intel®

S.A. Distributor

ELECTRONIC BUILDING ELEMENTS (PTY) LTD
PURVEYORS OF ALL ELECTRONIC COMPONENTS
Telephone: 78-9221/6
P.O. Box 4609, Pretoria
Telex: 3-0181 SA
Pine Square (2nd Floor)
Berea Street Hazelwood Pretoria

The 8086 Family User's Manual

October 1979

© Intel Corporation 1978, 1979
9800722-03/$7.50

intel

The 8086 Family User's Manual

October 1979

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

i	iSBC	Multimodule
ICE	Library Manager	PROMPT
iCS	MCS	Promware
Insite	Megachassis	RMX
Intel	Micromap	UPI
Intelevision	Multibus	μScope
Intellec		

and the combination of ICE, iCS, iSBC, MCS, or RMX and a numerical suffix.

Table of Contents

CHAPTER 1
INTRODUCTION

PAGE

Manual Organization 1-1
8086 Family Architecture 1-1
Functional Distribution 1-1
Microprocessors 1-2
Interrupt Controller 1-3
Bus Interface Components 1-3
Multiprocessing 1-3
Bus Organization 1-4
Local Bus 1-4
System Bus 1-5
Processing Modules 1-6
Bus Implementation Examples 1-6
Development Aids 1-12

CHAPTER 2
THE 8086 AND 8088 CENTRAL PROCESSING UNITS

Processor Overview 2-1
Processor Architecture 2-3
Execution Unit 2-5
Bus Interface Unit 2-5
General Registers 2-6
Segment Registers 2-7
Instruction Pointer 2-7
Flags 2-7
8080/8085 Register and Flag Correspondence ... 2-8
Mode Selection 2-8
Memory 2-8
Storage Organization 2-8
Segmentation 2-10
Physical Address Generation 2-11
Dynamically Relocatable Code 2-13
Stack Implementation 2-14
Dedicated and Reserved Memory Locations.... 2-14
8086/8088 Memory Access Differences 2-15
Input/Output 2-15
Input/Output Space 2-16
Restricted I/O Locations 2-16
8086/8088 Memory Access Differences 2-16
Memory-Mapped I/O 2-16
Direct Memory Access 2-17
8089 Input/Output Processor (IOP) 2-17
Multiprocessing Features 2-17
Bus Lock 2-17
WAIT and $\overline{\text{TEST}}$ 2-18
Escape 2-19
Request/Grant Lines 2-20
Multibus™ Architecture 2-21

PAGE

8289 Bus Arbiter 2-22
Processor Control and Monitoring 2-22
Interrupts 2-22
External Interrupts 2-22
Internal Interrupts 2-24
Interrupt Pointer Table 2-25
Interrupt Procedures 2-26
Single-Step (Trap) Interrupt 2-28
Breakpoint Interrupt 2-28
System Reset 2-29
Instruction Queue Status 2-29
Processor Halt 2-29
Status Lines 2-30
Instruction Set 2-30
Data Transfer Instructions 2-31
General Purpose Data Transfers 2-31
Address Object Transfers 2-32
Flag Transfers 2-32
Arithmetic Instructions 2-33
Arithmetic Data Formats 2-33
Arithmetic Instructions and Flags 2-34
Addition 2-35
Subtraction 2-36
Multiplication 2-36
Division 2-37
Bit Manipulation Instructions 2-38
Logical 2-38
Shifts 2-39
Rotates 2-39
String Instructions 2-40
Program Transfer Instructions 2-43
Unconditional Transfers 2-43
Conditional Transfers 2-45
Iteration Control 2-45
Interrupt Instructions 2-46
Processor Control Instructions 2-47
Flag Operations 2-47
External Synchronization 2-48
No Operation 2-48
Instruction Set Reference Information 2-48
Addressing Modes 2-68
Register and Immediate Operands 2-68
Memory Addressing Modes 2-68
The Effective Address 2-68
Direct Addressing 2-69
Register Indirect Addressing 2-69
Based Addressing 2-70
Indexed Addressing 2-70

PAGE
Based Indexed Addressing 2-71
String Addressing..... 2-72
I/O Port Addressing 2-72
Programming Facilities 2-72
Software Development Overview..... 2-73
PL/M-86..... 2-75
Statements and Comments 2-75
Data Definition 2-75
Assignment Statement..... 2-77
Program Flow Statements..... 2-79
Procedures 2-81
ASM-86..... 2-83
Statements 2-83
Constants 2-84
Defining Data..... 2-85
Records..... 2-85
Structures 2-87
Addressing Modes 2-87
Segment Control 2-88
Procedures 2-90
LINK-86 2-90
LOC-86..... 2-90
LIB-86..... 2-91
OH-86 2-91
CONV-86 2-92
Sample Programs..... 2-92
Programming Guidelines and Examples..... 2-96
Programming Guidelines 2-96
Segments and Segment Registers 2-96
Self-Modifying Code..... 2-96
Input/Output 2-97
Operating Systems..... 2-97
Interrupt Service Procedures 2-99
Stack-Based Parameters 2-100
Flag Images..... 2-100
Programming Examples 2-100
Procedures 2-100
Jumps and Calls..... 2-105
Records..... 2-110
Dynamic Code Relocation 2-113
Memory-Mapped I/O 2-115
Breakpoints 2-117
Interrupt Procedures..... 2-119
String Operations..... 2-125

CHAPTER 3
THE 8089 INPUT/OUTPUT PROCESSOR

Processor Overview..... 3-1
Evolution 3-1
Principles of Operation..... 3-2
CPU/IOP Communications..... 3-2
Channels..... 3-4
Channel Programs (Task Blocks)..... 3-4
DMA Transfers 3-5
Bus Configurations 3-5
A Sample Transaction..... 3-10
Applications..... 3-12

Processor Architecture..... 3-13
Common Control Unit (CCU) 3-13
Arithmetic/Logic Unit (ALU) 3-13
Assembly/Disassembly Registers..... 3-14
Instruction Fetch Unit..... 3-14
Bus Interface Unit (BIU)..... 3-16
Channels..... 3-16
I/O Control 3-16
Registers 3-17
Program Status Word 3-18
Tag Bits..... 3-19
Concurrent Channel Operation 3-20
Memory 3-21
Storage Organization..... 3-22
Dedicated and Reserved Memory Locations.... 3-23
Dynamic Relocation 3-23
Memory Access 3-24
Input/Output 3-25
Programmed I/O..... 3-25
I/O Instructions..... 3-25
Device Addressing 3-26
I/O Bus Transfers 3-26
DMA Transfers 3-27
Preparing the Device Controller..... 3-27
Preparing the Channel..... 3-27
Beginning the Transfer 3-31
DMA Transfer Cycle..... 3-32
Following the Transfer 3-33
Multiprocessing Features 3-34
Bus Arbitration 3-34
Request/Grant Line..... 3-35
8289 Bus Arbiter 3-36
Bus Arbitration for IOP Configurations..... 3-36
Bus Load Limit 3-36
Bus Lock..... 3-37
Processor Control and Monitoring 3-37
Initialization..... 3-37
Channel Commands 3-40
DRQ (DMA Request) 3-43
EXT (External Terminate) 3-43
Interrupts 3-43
Status Lines 3-43
Instruction Set 3-44
Data Transfer Instructions 3-44
Arithmetic Instructions..... 3-45
Logical and Bit Manipulation Instructions..... 3-46
Program Transfer Instructions..... 3-48
Processor Control Instructions 3-49
Instruction Set Reference Information 3-51

PAGE
Addressing Modes 3-59
Register and Immediate Operands 3-59
Memory Addressing Modes 3-59
The Effective Address 3-60
Based Addressing 3-60
Offset Addressing 3-60
Indexed Addressing 3-60
Indexed Auto-Increment Addressing 3-61
Programming Facilities 3-63
ASM-89 3-63
Statements 3-63
Constants 3-66
Defining Data 3-66
Structures 3-67
Addressing Modes 3-68
Program Transfer Targets 3-68
Procedures 3-69
Segment Control 3-69
Intermodule Communication 3-70
Sample Program 3-73
Linking and Locating ASM-89 Modules 3-76
Programming Guidelines and Examples 3-79
Programming Guidelines 3-79
Segments 3-79
Self-Modifying Code 3-79
I/O System Design 3-79
Programming Examples 3-81
Initialization and Dispatch 3-81
Memory-to-Memory Transfer 3-85
Saving and Restoring Registers 3-85

CHAPTER 4
HARDWARE REFERENCE INFORMATION

Introduction 4-1
8086 and 8088 CPUs 4-1
CPU Architecture 4-1
Bus Operation 4-5
Clock Circuit 4-10
Minimum/Maximum Mode 4-10
Minimum Mode 4-11
Maximum Mode 4-11
External Memory Addressing 4-14
I/O Interfacing 4-15
Interrupts 4-16
Machine Instruction Encoding and Decoding 4-18
8086 Instruction Sequence 4-37
8089 I/O Processor 4-38
System Configuration 4-39
Local Mode 4-39
Remote Mode 4-40
Bus Operation 4-41
Initialization 4-44
I/O Dispatching 4-46
PAGE
DMA Transfers 4-47
DMA Termination 4-50
Peripheral Interfacing 4-50
Instruction Encoding 4-52

APPENDIX A
APPLICATION NOTES

AP-67 8086 System Design A-3
AP-61 Multitasking for the 8086 ..... A-67
AP-50 Debugging Strategies and Considerations for 8089 Systems A-85
AP-51 Designing 8086, 8088, 8089 Multiprocessing Systems with the 8289 Bus Arbiter A-111
AP-59 Using the 8259A Programmable Interrupt Controller A-135
AP-28A Intel®Multibus™ Interfacing A-175
AP-43 Using the iSBC-957™ Execution Vehicle for Executing 8086 Program Code A-209

APPENDIX B
DEVICE SPECIFICATIONS

8086 Family
8086/8086-2/8086-4 16-Bit HMOS Microprocessor B-1
M8086 16-Bit HMOS Microprocessor B-22
I8086 16-Bit HMOS Microprocessor B-23
8088 8-Bit HMOS Microprocessor B-24
8089 8/16-Bit HMOS I/O Processor B-46
8282/8283 Octal Latch B-59
8284 Clock Generator and Driver for 8086, 8088, 8089 Processors B-63
M8284 Clock Generator and Driver for 8086, 8088, 8089 Processors B-69
I8284 Clock Generator and Driver for 8086, 8088, 8089 Processors B-70
8286/8287 Octal Bus Transceiver B-71
8288 Bus Controller for 8086, 8088, 8089 Processors B-75
8289 Bus Arbiter B-81
8237/8237-2 High Performance Programmable DMA Controller B-92
8259A/8259A-2/8289A-8 Programmable Interrupt Controller B-106
8085 Peripherals
8155/8156/8155-2/8156-2 2048 Bit Static MOS RAM with I/O Ports and Timer B-124
8185/8185-2 1024 x 8-Bit Static RAM for MCS-85™ B-125
8355/8355-2 16,384-Bit ROM with I/O B-126
8755A/8755A-2 16,384-Bit EPROM with I/O B-127

PAGE

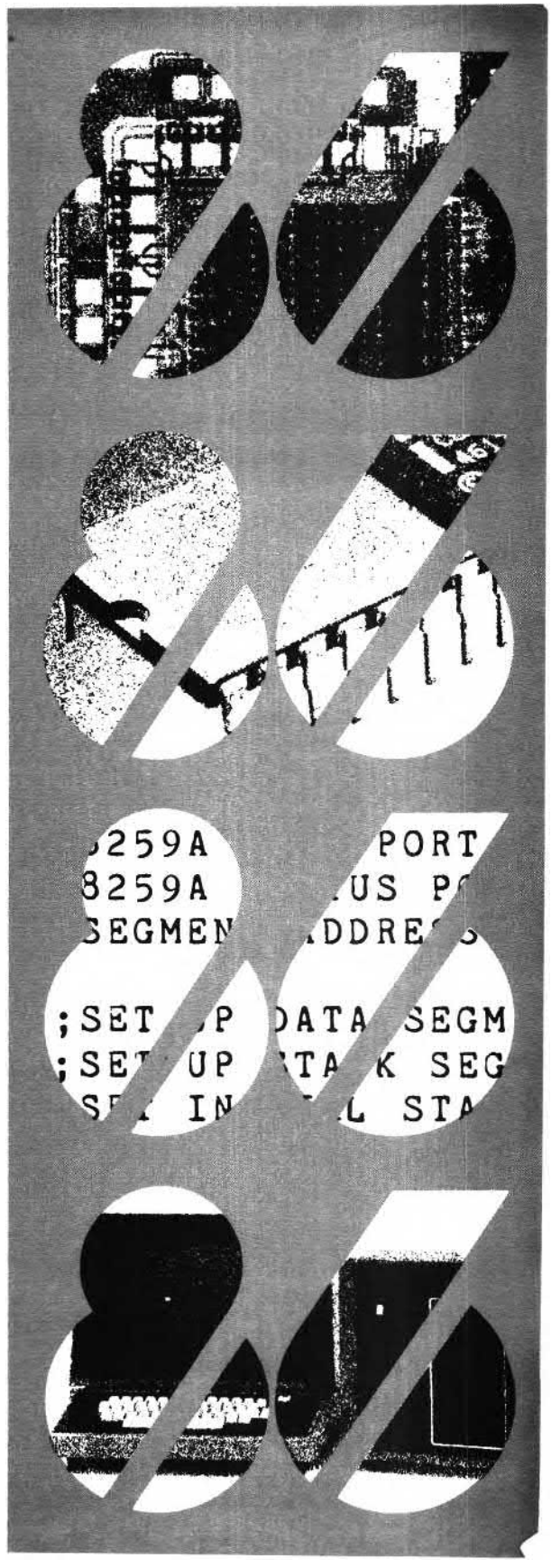
Standard Peripherals
8041A/8741A Universal Peripheral Interface 8-Bit Microcomputer B-128
8202 Dynamic RAM Controller..... B-129
8205 High Speed 1 Out of 8 Binary Decoder .. B-130
8251A Programmable Communication Interface..... B-131
8253/8253-5 Programmable Interval Timer... B-132
8255A/8255A-5 Programmable Peripheral Interface..... B-133
8271/8271-6/8271-8 Programmable Floppy Disk Controller B-134
8273 Programmable HDLC/SDLC Protocol Controller..... B-135
8275 Programmable CRT Controller B-136
8279/8279-5 Programmable Keyboard/Display Interface..... B-137
8291 GPIB Talker/Listener B-138
8292 GPIB Controller..... B-139
8293 GPIB Transceiver..... B-140
8294 Data Encryption Unit B-141
8295 Dot Matrix Printer Controller B-142

RAM Memories
2114A 1024 x 4 Bit Static RAM B-143
2142 1024 x 4 Bit Static RAM..... B-144
2148 1024 x 4 Bit Static RAM..... B-145

EPROM Memories
2716 16K (2K x 8) UV Erasable PROM..... B-146
2732 32K (4K x 8) UV Erasable PROM..... B-147
2758 8K (1K x 8) UV Erasable Low Power PROM B-148

Development Tools
Model 230 Intellec® Series II Microcomputer Development System.... B-149
8086/8088 Software Development Package B-153
8089 Assembler Support Package B-163
ICE-86™ 8086 In-Circuit Emulator B-165
iSBC 86/12A™ Single Board Computer B-171
iSBC 957™ Intellec®-iSBC 86/12A™ Interface and Execution Package..... B-179
iSBC 300/340™ iSBC 300™ 32K-Byte RAM Expansion Module iSBC 340™ 16K-Byte EPROM/ROM Expansion Module B-184
SDK-86 MCS-86™ System Design Kit B-188
SDK-C86 MCS-86™ System Design Kit B-194

Chapter 1
Introduction



CHAPTER 1
INTRODUCTION

This publication describes the Intel® 8086 family of microcomputing components, concentrating on the 8086, 8088 and 8089 microprocessors. It is written for hardware and software engineers and technicians who understand microcomputer operating principles. The manual is intended to introduce the product line and to serve as a reference during system design and implementation.

Recognizing that successful microcomputer-based products are judicious blends of hardware and software, the *User's Manual* addresses both subjects, although at different levels of detail. This publication is the definitive source for information describing the 8086 family components. Software topics, such as programming languages, utilities and examples, are given moderately detailed, but by no means complete, coverage. Additional references, available from Intel's Literature Department, are cited in the programming sections.

1.1 Manual Organization

The manual contains four chapters and three appendices. The remainder of this chapter describes the architecture of the 8086 family, and subsequent chapters cover the individual components in detail.

Chapter 2 describes the 8086 and 8088 Central Processing Units, and Chapter 3 covers the 8089 Input/Output Processor. These two chapters are identically organized and focus on providing a *functional* description of the 8086, 8088 and 8089, plus related Intel hardware and software products. Hardware reference information—electrical characteristics, timing and physical interfacing considerations—for all three processors is concentrated in Chapter 4.

Appendix A is a collection of 8086 family application notes; these provide design and debugging examples. Appendix B contains complete data sheets for all the 8086 family components and system development aids; summary data sheets covering compatible components from other Intel product lines are also reproduced in Appendix B.

1.2 8086 Family Architecture

Considered individually, the 8086, 8088 and 8089 are advanced third-generation microprocessors. Moreover, these processors are elements of a larger design, that of the 8086 family. This systems architecture specifies how the processors and other components relate to each other, and is the key to the exceptional versatility of these products.

The components in the 8086 family have been designed to operate together in diverse combinations within the systematic framework of the overall family architecture. In this way a single family of components can be used to solve a wide array of microcomputing problems. A component mix can be tailored to fit the performance needs of an application precisely, without having to pay for unneeded capabilities that may be bundled into more monolithic, CPU-centered architectures. Using the same family of components across multiple systems limits the learning curve problem and builds on past experience. Finally, the modular structure of the family architecture provides an orderly way for systems to grow and change.

The 8086 family architecture is characterized by three major principles:

1. System functions are distributed among specialized components.
2. Multiprocessing capabilities are inherent in the hardware.
3. A hierarchical bus organization provides for the complex data flows required by high-performance systems without burdening simpler systems with unneeded capabilities.

Functional Distribution

Table 1-1 lists the components that constitute the 8086 microprocessor family. All components are contained in standard dual in-line packages and require single +5V power sources.

Table 1-1. 8086 Component Family

Microprocessor	Technology	Pins	Description
8086 Central Processing Unit (CPU)	HMOS	40	8/16 bit general-purpose microprocessor; 16-bit external data path.
8088 Central Processing Unit (CPU)	HMOS	40	8/16 bit general-purpose microprocessor; 8-bit external data path.
8089 Input/Output Processor (IOP)	HMOS	40	8/16 bit microprocessor optimized for high-speed I/O operations; 8-bit and 16-bit external data paths.

Support Component	Technology	Pins	Function
8259A Programmable Interrupt Controller (PIC)	NMOS	28	Identifies highest-priority interrupt request.
8282 Octal Latch 8283 Octal Latch (Inverting)	Bipolar	20	Demultiplexes and increases drive of address bus.
8284 Clock Generator and Driver	Bipolar	18	Provides time base.
8286 Octal Bus Transceiver 8287 Octal Bus Transceiver (Inverting)	Bipolar	20	Increases drive on data bus.
8288 Bus Controller	Bipolar	20	Generates bus command signals.
8289 Bus Arbiter	Bipolar	20	Controls access of microprocessors to multimaster system bus.

Microprocessors

At the core of the product line are three microprocessors that share these characteristics:

- Standard operating speed is 5 MHz (200 ns cycle time); a selected 8 MHz version of the 8086 CPU is also available.
- Chips are housed in reliable 40-pin packages.
- Processors operate on both 8- and 16-bit data types; internal data paths are at least 16 bits wide.
- Up to 1 megabyte of memory can be addressed, along with a separate 64k byte I/O space.
- The address/data and status interfaces of the processors are compatible (the address and data buses are time-multiplexed at the processor, i.e., an address transmission is followed by a data transmission over a subset of the same physical lines).

The 8086 and 8088 are third-generation central processing units (CPUs) that differ primarily in their external data paths. The 8088 transfers data between itself and other system components 8 bits at a time. The 8086 can transfer either 8 or 16 bits in one bus cycle and is therefore capable of greater throughput. Both processors have two operating modes, selectable by a strapping pin. In minimum mode, the CPUs emit the bus control signals needed by memory and I/O peripheral components. In maximum mode, an 8288 Bus Controller assumes responsibility for controlling devices attached to the system bus. CPU pins no longer needed for bus control are then redefined to provide signals that support multiprocessing systems.

The 8089 Input/Output Processor (IOP) is an independent microprocessor whose design has been optimized for transferring data. The 8089

typically runs under the direction of a CPU, but it executes a separate instruction stream and can operate in parallel with other system processors. The IOP contains two independent I/O channels that combine attributes of both CPUs and advanced DMA (direct memory access) controllers. The channels can execute programs and perform programmed I/O operations similar to CPUs. They may also transfer data by DMA, at rates up to 1.25 megabytes per second (5 MHz version). The channels can support mixes of 8- and 16-bit I/O devices and memory. Combining speed with programmable intelligence, the 8089 can assume the bulk of I/O processing overhead and thereby free a CPU to perform other tasks.

Interrupt Controller

The 8259A Programmable Interrupt Controller (PIC) is a new, 8086 family-compatible version of the familiar 8259 that has been enhanced to operate with the advanced interrupt facilities of the 8086 and 8088 CPUs. The 8259A accepts interrupt requests from up to eight sources; up to 64 sources may be accommodated by "cascading" additional 8259As. Each interrupt source is assigned a priority number that typically reflects its "criticality" in the system. The 8259A has several built-in, priority-resolving mechanisms that are selectable by software commands from the CPU. These modes operate somewhat differently, but in general the 8259A continuously identifies the highest-priority active interrupt request and generates an interrupt request to the CPU if this request has higher priority than the request currently being processed. When the CPU recognizes the interrupt request, the 8259A transfers a code to the CPU that identifies the interrupt source.

Bus Interface Components

Components may be selected from this modular group to implement different system bus configurations. Except for the 8284, all components are optional; their inclusion in a system is based on the needs of the application. All of the bus interface components are implemented using bipolar technology to provide high-quality, high-drive signals and very fast internal switching.

The 8284 Clock Generator and Driver provides the time base for the 8086 family microprocessors. It divides the frequency signal from an external crystal or TTL signal by three and outputs the 5 MHz or 8 MHz processor clock signal. It also provides the microprocessors with reset and ready signals.

8282 or 8283 Octal Latches may be added to a system to demultiplex the combined address/data bus generated by the 8086 family microprocessors. A demultiplexed bus provides separate stable address and data lines required by many peripheral components. Two latches demultiplex 16 bits of the bus to provide an address space of up to 64k bytes, while three latches generate the full 20-bit (megabyte) address space. The latches also provide the high drive on the address lines needed in larger systems.

8286 and 8287 Octal Bus Transceivers are used to provide more drive on data lines than the processors themselves are capable of providing. One or two transceivers may be used depending on the width of the data bus (8 or 16 bits).

The 8288 Bus Controller decodes status signals output by an 8089, or a maximum mode 8086 or 8088. When these signals indicate that the processor is to run a bus cycle, the 8288 issues a bus command that identifies the bus cycle as memory read, memory write, I/O read, I/O write, etc. It also provides a signal that strobes the address into 8282/83 latches. The 8288 provides the drive levels needed for the bus control lines in medium to large systems.

The 8289 Bus Arbiter controls the access of a processor to a multimaster system bus. A multimaster bus is a path to system resources (typically memory) that is shared by two or more microprocessors (masters). Arbiters for each master may use one of several priority-resolving techniques to ensure that only one master drives the shared bus.

Multiprocessing

Employing multiple processors in medium to large systems offers several significant advantages over the centralized approach that relies on a single CPU and extremely fast memory:

- system tasks may be allocated to special-purpose processors whose designs are optimized to perform certain types of tasks simply and efficiently;

- very high levels of performance can be attained when multiple processors can execute simultaneously (parallel processing);
- robustness can be improved by isolating system functions so that a failure or error in one part of the system has a limited effect on the rest of the system;
- the natural partitioning of the system promotes parallel development of subsystems, breaks the application into smaller, more manageable tasks, and helps isolate the effects of system modifications.

The 8086 family architecture is explicitly designed to simplify the development of multiple processor systems by providing facilities for coordinating the interaction of the processors.

The architecture supports two types of processors: independent processors and coprocessors. An independent processor is one that executes its own instruction stream. The 8086, 8088 and 8089 are examples of independent processors. An 8086 or 8088 typically executes a program in response to an interrupt. The 8089 starts its channels in response to an interrupt-like signal called a channel attention; this signal is typically issued by a CPU.

The 8086 architecture also supports a second type of processor, called a coprocessor. Coprocessor "hooks" have been designed into the 8086 and 8088 so that this type of processor can be accommodated in the future. A coprocessor differs from an independent processor in that it obtains its instructions from another processor, called a host. The coprocessor monitors instructions fetched by the host and recognizes certain of these as its own and executes them. A coprocessor, in effect, extends the instruction set of its host processor.

The 8086 family architecture provides built-in solutions to two classic multiprocessing coordination problems: bus arbitration and mutual exclusion. Bus arbitration may be performed by the bus request/grant logic contained in each of the processors, by 8289 Bus Arbiters, or by a combination of the two when processors have access to multiple shared buses. In all cases, the arbitration mechanism operates invisibly to software.

For mutual exclusion, each processor has a LOCK (bus lock) signal which a program may activate to prevent other processors from obtaining a shared system bus. The 8089 may lock the bus during a DMA transfer to ensure that both the transfer completes in the shortest possible time and that another processor does not access the target of the transfer (e.g., a buffer) while it is being updated. Each of the processors has an instruction that examines and updates a memory byte with the bus locked. This instruction can be used to implement a semaphore mechanism for controlling the access of multiple processors to shared resources. (A semaphore is a variable that indicates whether a resource, such as a buffer or a pointer, is "available" or "in use"; section 2.5 discusses semaphores in more detail).

Bus Organization

Figure 1-1 summarizes the 8086 family bus structure. There are two different types of buses: system and local. Both buses may be shared by multiple processors, i.e., both are multimaster buses. Microprocessors are always connected to a local bus, and memory and I/O components usually reside on a system bus. The 8086 family bus interface components link a local bus to a system bus.

Local Bus

The local bus is optimized for use by the 8086 family microprocessors. Since standard memory and I/O components are not attached to the local bus, information can be multiplexed and encoded to make very efficient use of processor pins (certain MCS-85™ peripheral components can be directly connected to the local bus). This allows several pins to be dedicated to coordinating the activity of multiple processors sharing the local bus. Multiple processors connected to the same local bus are said to be local to each other; processors on different local buses are said to be remote to each other, or configured remotely. Both independent processors and coprocessors may share a local bus; on-chip arbitration logic determines which processor drives the bus. Because the processors on the local bus share the same bus interface components, the local configuration of multiple processors provides a compact and inexpensive multiprocessing system.

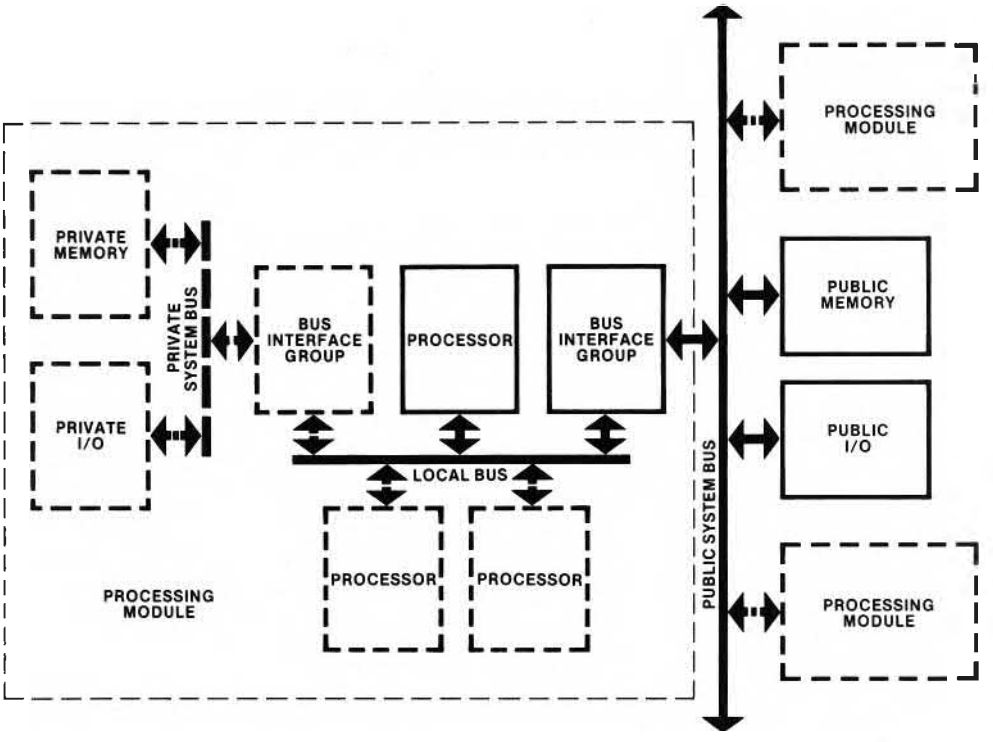


Figure 1-1. Generalized 8086 Family Bus Structure

System Bus

A full implementation of an 8086 system bus consists of the following five sets of signals:

1. address bus,
2. data bus,
3. control lines,
4. interrupt lines, and
5. arbitration lines.

These signals are designed to meet the needs of standard memory and I/O devices; the address and data buses are demultiplexed and traditional control signals (memory read/write, I/O read/write, etc.) are provided on the system bus.

The system bus design is modular and subsets may be implemented according to the needs of the application. For example, the arbitration lines are not needed in single-processor systems or in multiple-processor systems that perform arbitration at the local-bus level.

A group of bus interface components transforms the signals of a local bus into a system bus. The number of bus interface components required to generate a system bus depends on the size and complexity of the system; reduced application needs translate directly into reduced component counts. These main variables determine the configuration of a bus interface group: address space size (number of latches), data bus width (number of transceivers), and arbitration needs (presence of a bus arbiter).

The 8086 family system bus is functionally and electrically compatible with the Multibus™ multimaster system bus used in Intel's iSBC™ line of single board computing products. This compatability gives system designers access to a wide variety of computer, memory, communications and other modules that may be incorporated into products, used for evaluation or for test vehicles.

Processing Modules

The processor(s) and bus interface group(s) that are connected by a local bus constitute a processing module. A simple processing module could consist of a single CPU and one bus interface group. A more complex module would contain multiple processors, such as two IOPs, or a CPU and one or two IOPs. One bus interface group typically links the processors in the module to a public system bus. If there are multiple processing modules in the system, all memory or I/O connected to the public bus is accessible to all processing modules on the public bus. 8289 Bus Arbiters in each processing module control the access of the modules to the public bus and hence to the public memory and I/O.

A second bus interface group may be connected to a processing module's local bus, generating a second bus. This bus can provide the processing module with a private address space that is not accessible to other processing modules. Distributing memory and I/O resources in this manner can improve system robustness by isolating the effects of failures. It can also increase system throughput dramatically. If processor programs and local data are placed in private memory, contention for use of the public system bus can be held to a minimum to ensure that shared resources are quickly available when they are needed. In addition, processors in separate modules can simultaneously fetch instructions from private memory spaces to allow multiple system tasks to proceed in parallel.

Bus Implementation Examples

This section summarizes the 8086 family bus organization by showing how components from the family can be combined to implement diverse bus configurations. The first two examples illustrate special cases that extend the applicability of the 8086 family to smaller systems. The remaining examples add and recombine the same basic components to form progressively more complex bus configurations. Note that these examples are intended to be illustrative rather than exhaustive; many different combinations of components can be tailored to fit the needs of individual applications.

In its minimum mode configuration, the 8088 time-multiplexes its 8-bit data bus with the lower eight bits of its 20-bit address bus (figure 1-2). This multiplexed address/data bus, and the bus control signals emitted by the 8088, are directly compatible with the multiplexed bus components of Intel's 8085 family. These peripherals contain on-chip logic that demultiplexes a combined address/data bus. In addition, many of these devices are multifunctional, combining, for example, RAM, I/O ports and a timer on a single chip. By using these components, it is possible to build small (as few as four chips) economical systems that are nonetheless capable of performing significant computing tasks.

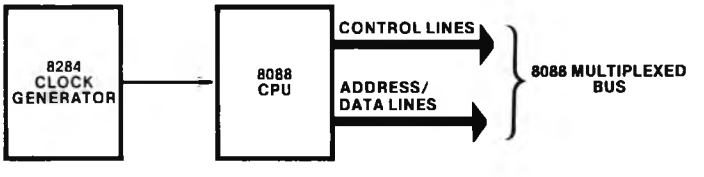


Figure 1-2. 8088 Multiplexed Bus

Combining 8282/83 latches with a minimum mode 8086 or 8088 produces a minimum mode system bus (figure 1-3). Two latches provide an address space of up to 64k bytes; adding a third latch provides access to the full megabyte of memory. An 8288 Bus Controller is not required for this implementation as the CPUs themselves emit the bus control signals when they are configured in the minimum mode. This demultiplexed bus structure is compatible with the wide array of memory and I/O components that have been developed for the industry-standard 8080A CPU. Eight-bit peripherals may be connected to both the upper and lower halves of the 8086's 16-bit data bus. 8286/87 transceivers may be added to provide additional drive on the data lines, where required. Including an 8259A gives the CPU the ability to respond to multiple interrupt sources without polling. The minimum mode system bus configuration is well-suited to a variety of systems whose computational requirements can be met by a single 8086 or 8088 CPU.

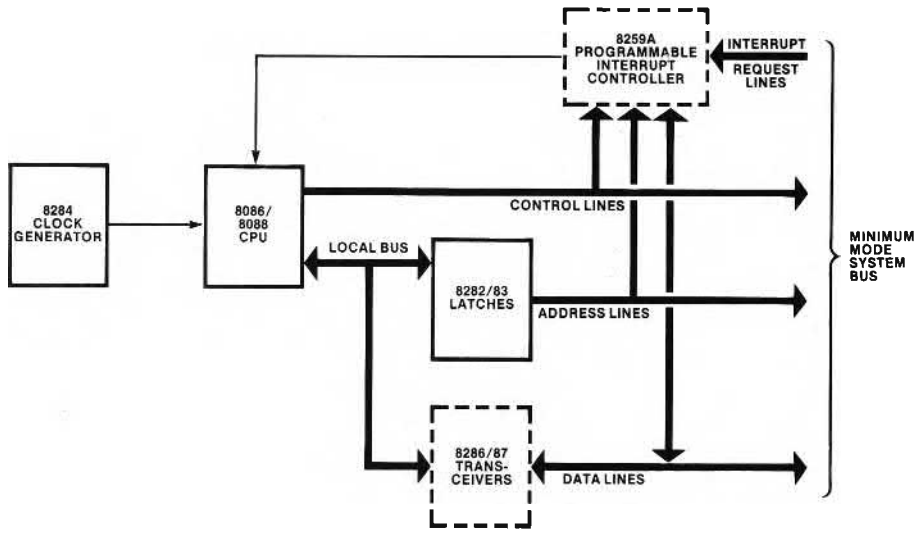


Figure 1-3. Minimum Mode System Bus

When an 8086 or 8088 is configured in maximum mode and an 8288 is added to control the system bus, one or two 8089s may be directly connected to the CPU (figure 1-4). The processors all share the same latches, transceivers, clock and bus controller, via the local bus. Arbitration logic built into the 8086, 8088 and 8089 coordinates use of the local bus, and thus of the system bus. This bus configuration enables the powerful I/O handling capabilities of the 8089 to be incorporated into systems of moderate size and cost.

The 8289 enables high-performance systems to be designed as a series of independent processing modules whose activities are coordinated via a shared system bus. Figure 1-5 shows the multimaster system bus interface; this bus structure is electrically compatible with the Multibus™ architecture used in Intel iSBC™ single-board computing systems.

Several different combinations of processors may be attached to the local bus of a multimaster computing module:

- a single 8086 or 8088
- a single 8089
- two 8089s
- an 8086 or 8088 and one 8089
- an 8086 or 8088 and two 8089s

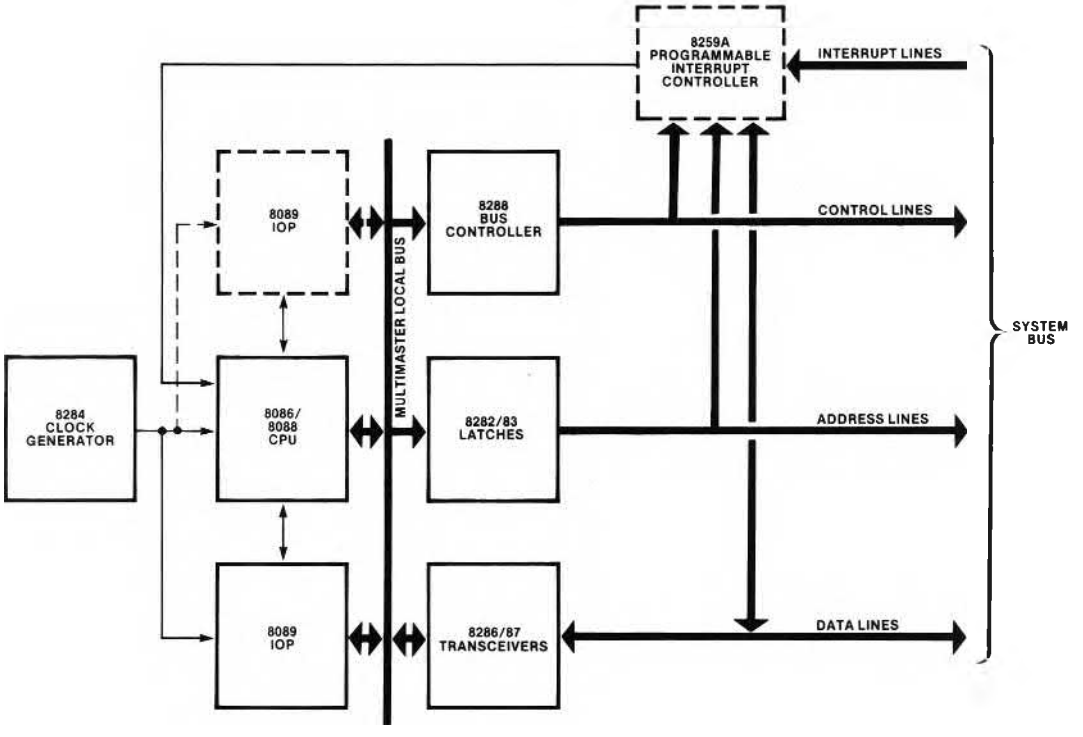


Figure 1-4. Multimaster Local Bus

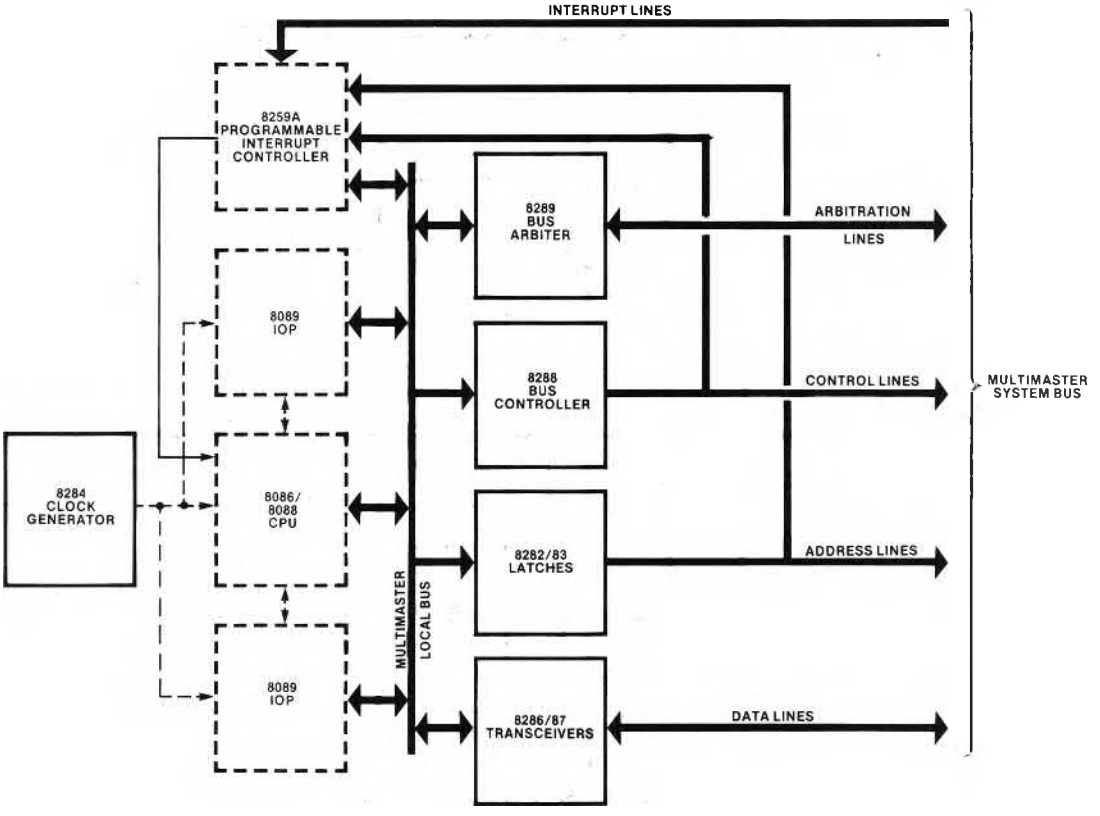


Figure 1-5. Basic Multimaster Processing Module

All of the processors on the local bus obtain access to the system bus through a single set of interface components.

One or two 8089s in a multimaster processing module may be configured with a private I/O bus as shown in figure 1-6. In this configuration, memory access commands are directed to the public multimaster system bus, while I/O commands use the private I/O bus. Memory, containing the 8089's programs, as well as I/O devices, may be connected to the private I/O bus. Taking this approach can greatly reduce the 8089's use of the system bus as most memory and I/O accesses can be made to the private address space. The system bus is thus made available for use by other processors, and the 8089 can execute in parallel with other processors for extended periods. A limited private I/O bus may be implemented using the 8-bit multiplexed peripherals of the 8085 family, eliminating the latches and transceivers shown in figure 1-6.

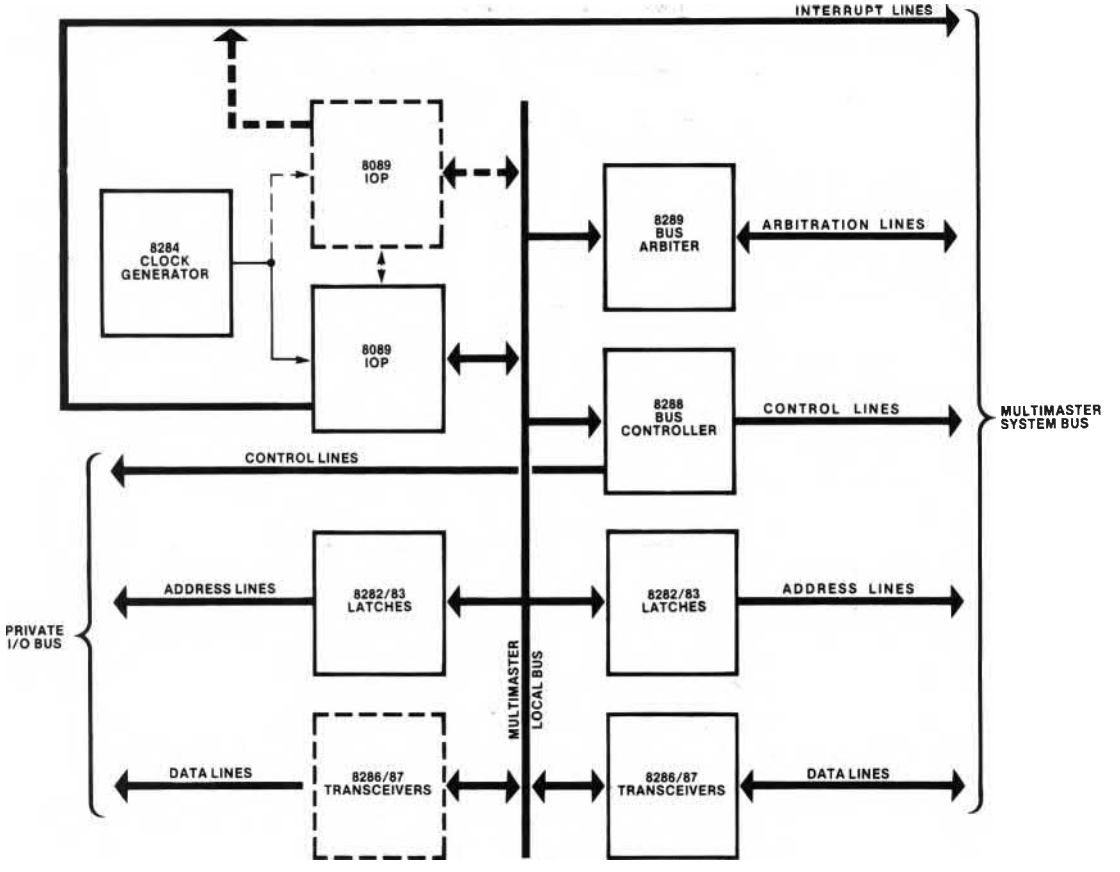


Figure 1-6. Private I/O Bus

Adding a second 8288 to the local bus allows an 8086 or 8088 in a processing module to divide its address space into system and resident sections (figure 1-7). A PROM or decoder is used to direct an address reference to the system bus or to the resident bus. The resident bus allows the CPU to run out of its own address space to minimize its use of the system bus. Since no other processors can access the private memory on the CPU's resident bus, operating system code and data in this space is protected from errors in other processor programs. If a second 8289 is added to a resident bus module, the resident bus becomes a second multimaster system bus.

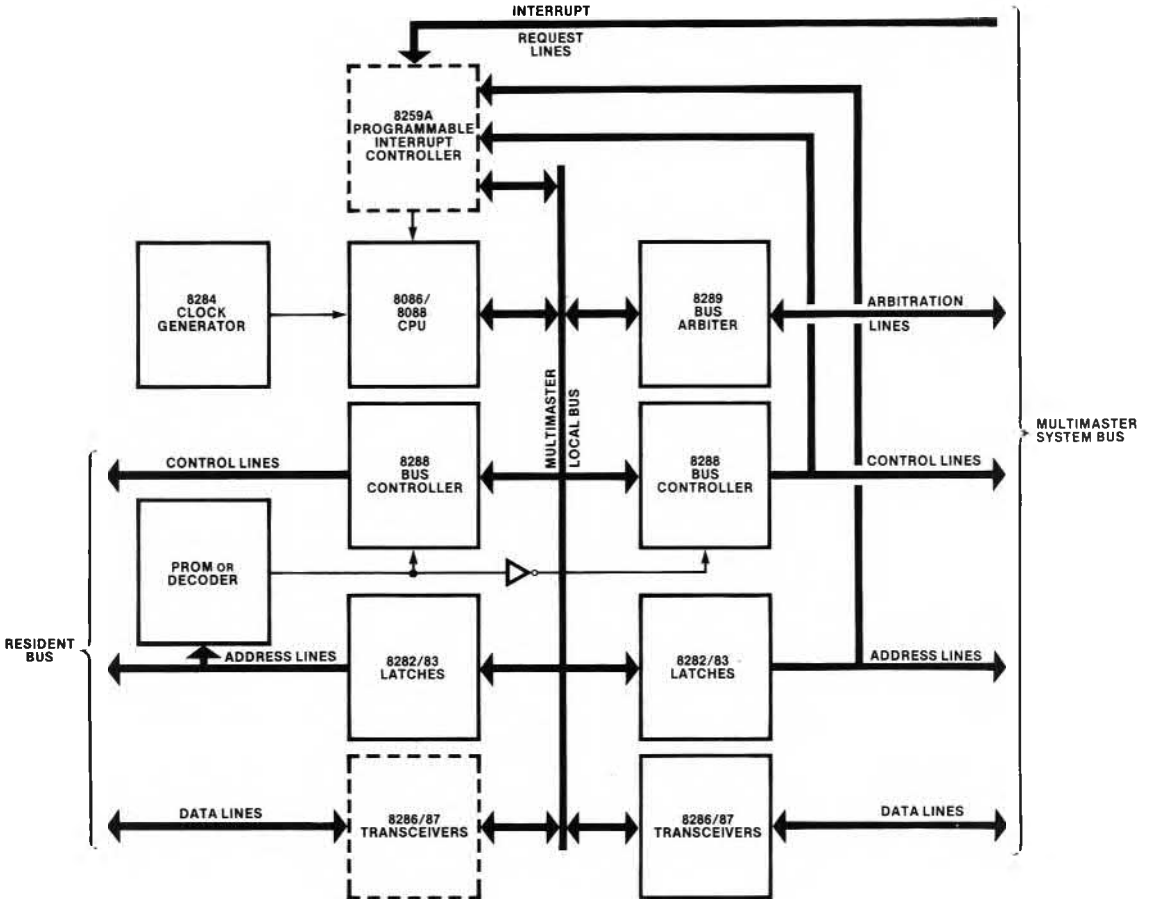


Figure 1-7. Resident Bus

As an alternative to the resident bus, a private read-only memory space can be implemented using the RD (read) signal provided by the CPUs in lieu of an 8288 Bus Controller.

Multiprocessing systems of widely varying complexity can be constructed from multimaster processing modules. Each module can be designed and implemented separately and can be optimized to perform a given task. The modules can communicate with each other by means of interrupts and messages placed in system memory. Additional functions can be added to a system by incorporating the new functions into modules and connecting the modules to the system bus.

Figure 1-8 illustrates a hypothetical system in which nine processors are distributed among five multimaster processing modules. (For clarity, bus interface components are not shown in figure 1-8.) A supervisor module controls the system, primarily responding to interrupts and dispatching other modules to perform tasks. The supervisor CPU, like the other processors in the system, executes code from private memory that is inaccessible to other modules. System memory, which is accessible to all the processors, is used only for messages, common buffers, etc. This helps to "protect" the processors from each other and to keep system bus contention at a minimum. The database module is responsible for maintaining all system files. Each of the three graphics modules supports a graphics CRT terminal. An 8089 in each module performs data transfers and CRT refresh and calls upon an 8088 for intensive computational routines.

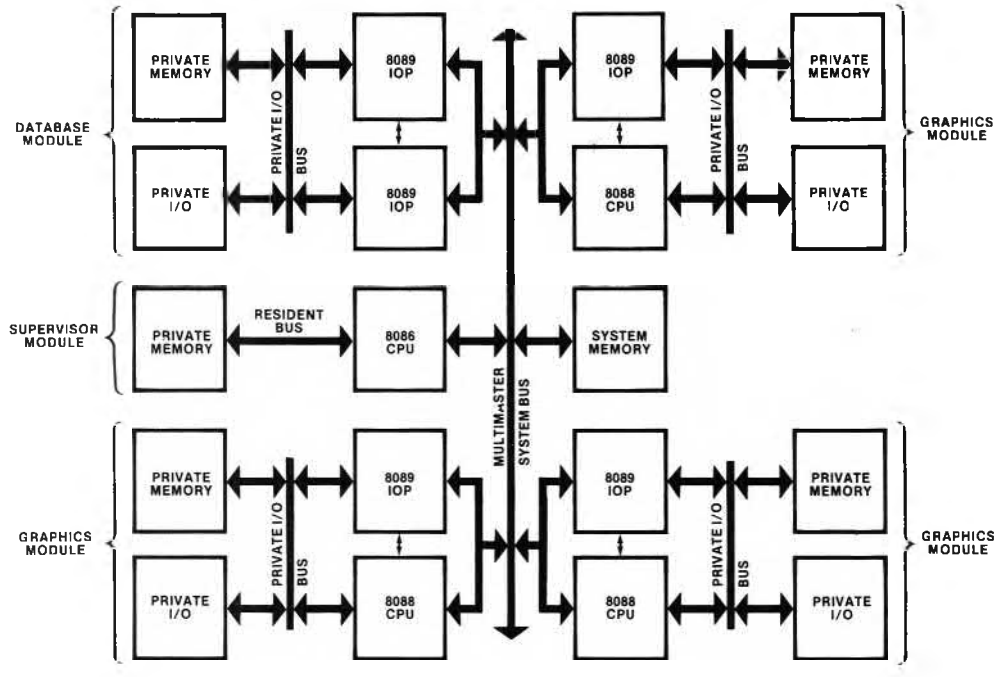


Figure 1-8. Multimaster Design Example

1.3 Development Aids

Intel provides the sophisticated tools needed for timely and economical development of products based on the 8086 family. The 8086 family system development environment is focused on the Intellec® Series II Microcomputer Development System (figure 1-9). The Intellec system is a multiple-microprocessor system that runs ISIS-II, a disk-based operating system that has been proven in thousands of installations. The Intellec has built-in interfaces for a printer, a PROM programmer and a paper tape reader/punch. This same hardware and operating system may be used to develop systems based on other Intel microprocessor families such as the 8085 and the 8048.

Three language translators support 8086 family programming. PL/M-86 is a high-level language for the 8086 and 8088 that supports structured programming techniques. It is upward-compatible with PL/M-80, the most widely used high-level microprocessor language. ASM-86 may be used to write assembly language programs for the 8086 and the 8088 CPUs and gives the programmer access to the full power of these CPUs. 8089 programs are written in ASM-89, the 8089 assembly language.

The language translators produce compatible outputs that can be manipulated by the software development utilities. LINK-86, for example, can combine programs written in ASM-86 with PL/M-86 programs. LIB-86 allows related programs to be stored in libraries to simplify storage and retrival. LOC-86 assigns absolute memory addresses to programs. OH-86 changes the format of an executable program for PROM programming or for loading into the RAM of a test vehicle.

The UPP-301 Universal PROM Programmer can burn programs into any of Intel's PROM memories; the UPP plugs into the Intellec® system and allows program data to be manipulated from the console before it is programmed into the PROM.

The SDK-86 is an (minimum mode) 8086-based prototyping and evaluation kit. It includes the CPU, RAM, I/O ports and a breadboard area for interfacing customer circuits. A ROM-based monitor program is supplied with the kit. Monitor commands may be entered from an on-board keypad or from a terminal; the monitor returns results to the SDK-86's on-board LED display or to a terminal. Monitor commands allow programs to be entered, run, stopped, and single-stepped; memory contents can be altered as well as displayed. The SDK-C86 Software and Cable Interface connects an SDK-86 to an Intellec® system. The software supplied with the cable enables programs to be transferred between the development system and the SDK-86 to allow users to develop programs using the text editor, translators and utilities of the Intellec system and then download the program to the SDK-86 for execution.

The iSBC 86/12™ board is a high-performance single board computer based on a maximum mode 8086 CPU. The board contains 32k of dual-port RAM that is accessible to the CPU via the on-board bus and to other processors via the built-in Multibus™ interface. The board also has an asynchronous serial port, parallel ports with sockets for drivers and terminators, two timers and sockets for 16k of ROM.

An iSBC 86/12™ can be linked to an Intellec® system using the iSBC 957™ Intellec-iSBC 86/12 Interface and Execution Package. The package includes a ROM-based monitor for the iSBC 86/12 board, software for the Intellec system and cabling to connect the two. The package supports data transfers between Intellec diskettes and iSBC 86/12 memory, full speed execution of customer programs on the iSBC 86/12 board, breakpoints, single-stepping, and data moves, replacements, searches and compares. All commands are entered from the Intellec console.

The ICE-86™ module is an in-circuit emulator for the 8086 microprocessor. A 40-pin probe replaces the 8086 in the system under test. This probe is connected to ICE-86 circuit boards that in turn plug into the Intellec® chassis. The ICE-86 module emulates the 8086 in the system under test in response to commands entered through the Intellec console. These commands allow the user to debug the system by setting breakpoints, tracing the flow of execution, single-stepping, examining and altering memory and I/O, etc. All references to program variables and labels are symbolic (i.e., their PL/M-86 or ASM-86 names). Software testing can also map "system under test" memory into the Intellec memory to permit software testing to begin before prototype hardware has been developed.

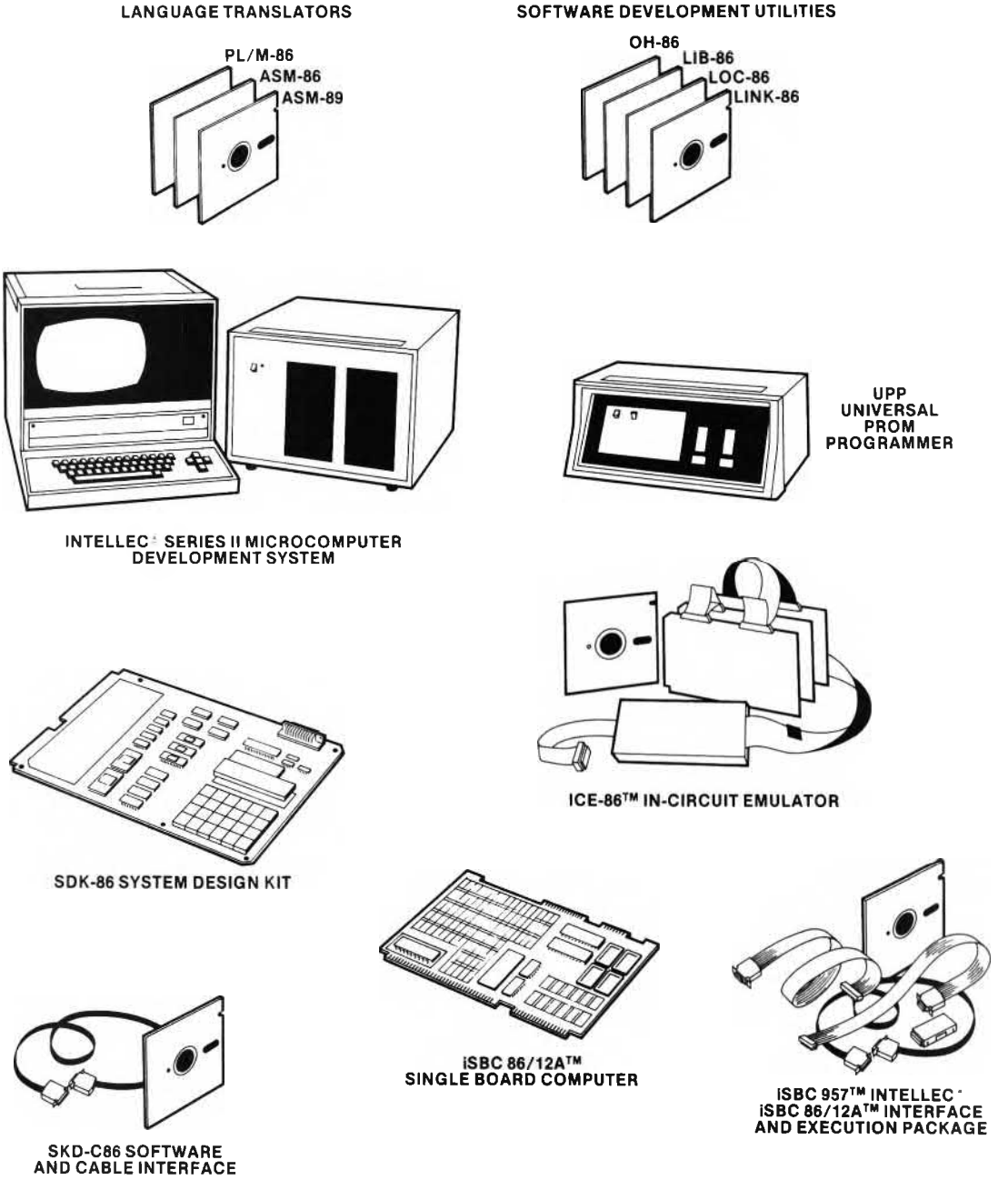
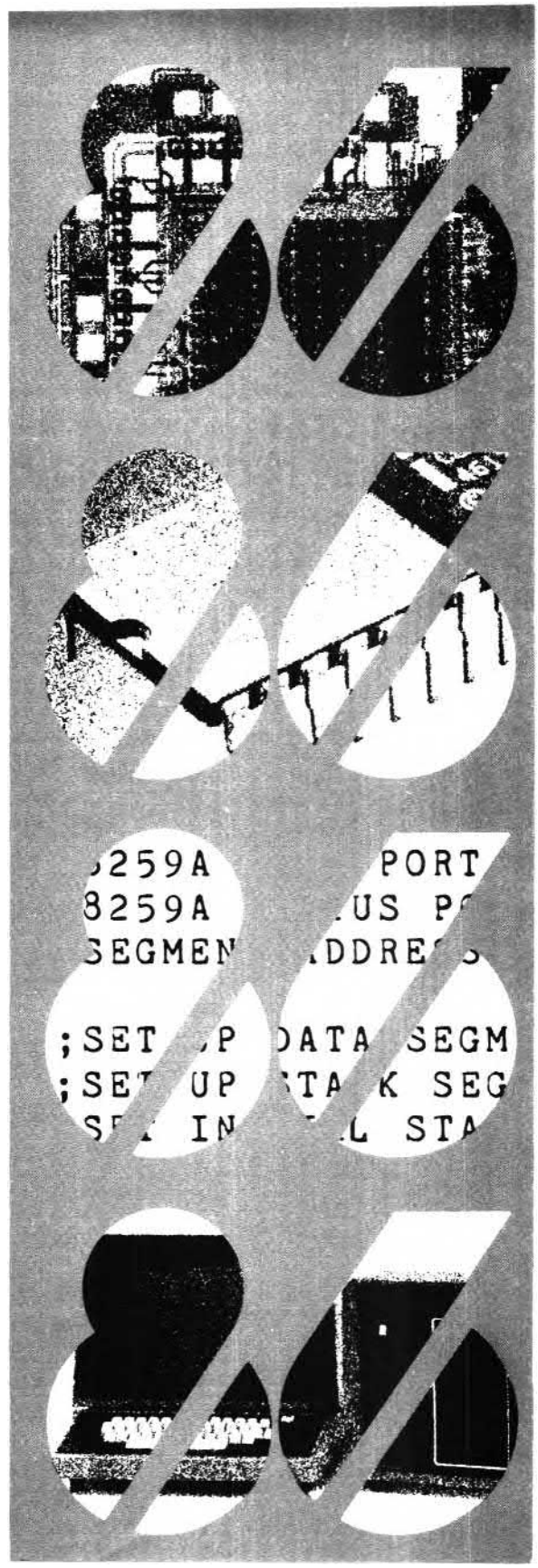


Figure 1-9. 8086 Family Development Aids

Chapter 2
The 8086 and 8088 Central Processing Units



CHAPTER 2
THE 8086 AND 8088 CENTRAL PROCESSING UNITS

This chapter describes the mainstays of the 8086 microprocessor family: the 8086 and 8088 central processing units (CPUs). The material is divided into ten sections and generally proceeds from hardware to software topics as follows:

1. Processor Overview
2. Processor Architecture
3. Memory
4. Input/Output
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Addressing Modes
9. Programming Facilities
10. Programming Guidelines and Examples

The chapter describes the internal operation of the CPUs in detail. The interaction of the processors with other devices is discussed in functional terms; electrical characteristics, timing, and other information needed to actually interface other devices with the 8086 and 8088 are provided in Chapter 4.

2.1 Processor Overview

The 8086 and 8088 are closely related third-generation microprocessors. The 8088 is designed with an 8-bit external data path to memory and I/O, while the 8086 can transfer 16 bits at a time. In almost every other respect the processors are identical; software written for one CPU will execute on the other without alteration. The chips are contained in standard 40-pin dual in-line packages (figure 2-1) and operate from a single +5V power source.

The 8086 and 8088 are suitable for an exceptionally wide spectrum of microcomputer applications, and this flexibility is one of their most outstanding characteristics. Systems can range from uniprocessor minimal-memory designs implemented with a handful of chips (figure 2-2), to multiprocessor systems with up to a megabyte of memory (figure 2-3).

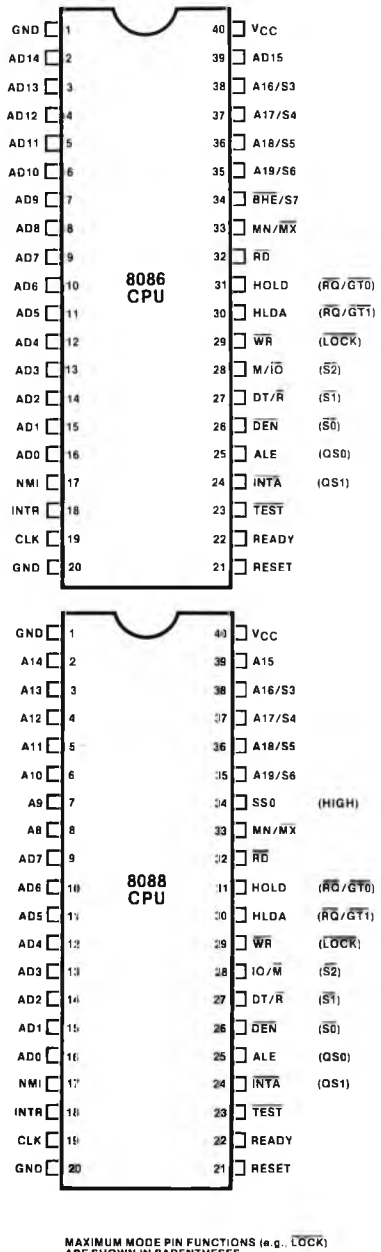


Figure 2-1. 8086 and 8088 Central Processing Units

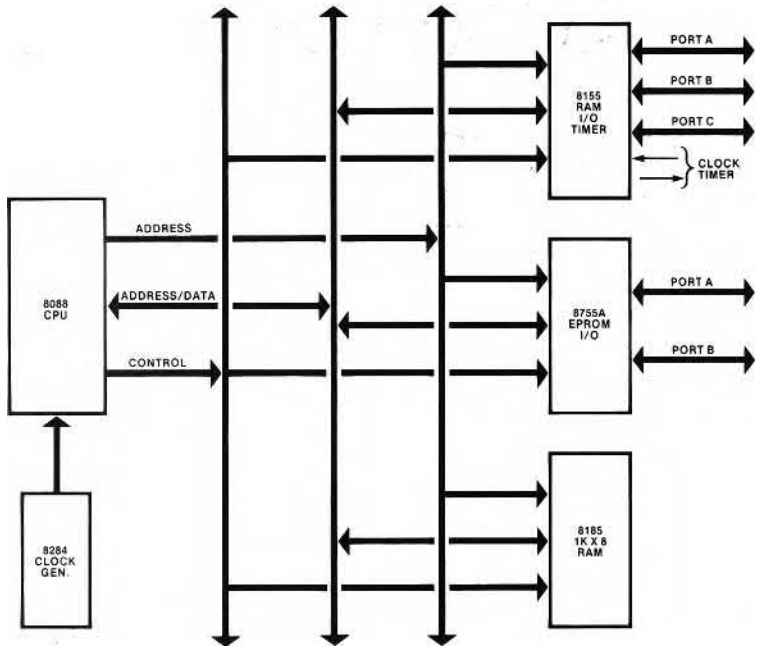


Figure 2-2. Small 8088-Based System

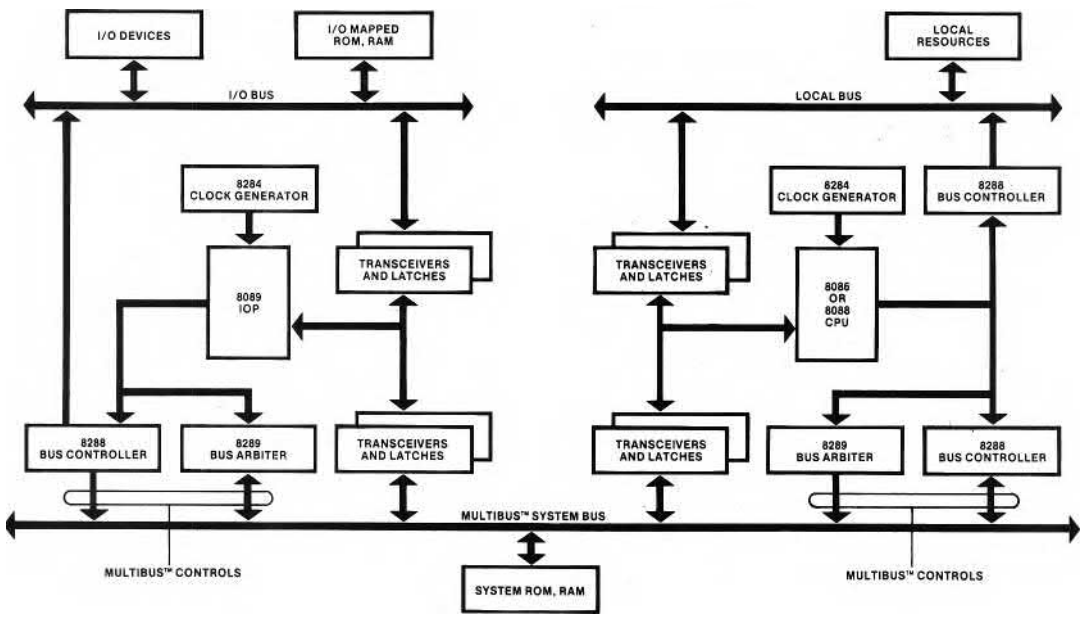


Figure 2-3. 8086/8088/8089 Multiprocessing System

The large application domain of the 8086 and 8088 is made possible primarily by the processors' dual operating modes (minimum and maximum mode) and built-in multiprocessing features. Several of the 40 CPU pins have dual functions that are selected by a strapping pin. Configured in minimum mode, these pins transfer control signals directly to memory and input/output devices. In maximum mode these same pins take on different functions that are helpful in medium to large ystems, especially systems with multiple processors. The control functions assigned to these pins in minimum mode are assumed by a support chip, the 8288 Bus Controller.

The CPUs are designed to operate with the 8089 Input/Output Processor (IOP) and other processors in multiprocessing and distributed processing systems. When used in conjunction with one or more 8089s, the 8086 and 8088 expand the applicability of microprocessors into I/O-intensive data processing systems. Built-in coordinating signals and instructions, and electrical compatibility with Intel's Multibus™ shared bus architecture, simplify and reduce the cost of developing multiple-processor designs.

Both CPUs are substantially more powerful than any microprocessor previously offered by Intel. Actual performance, of course, varies from application to application, but comparisons to the industry standard 2-MHz 8080A are instructive. The 8088 is from four to six times more powerful than the 8080A; the 8086 provides seven to ten times the 8080A's performance (see figure 2-4).

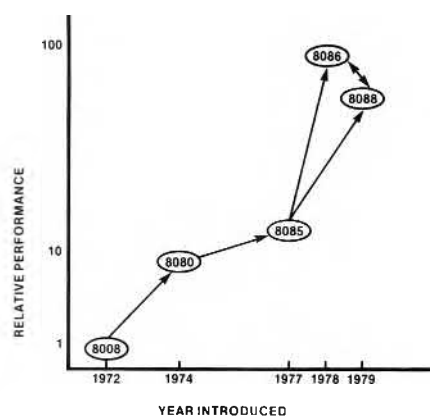


Figure 2-4. Relative Performance of the 8086 and 8088

The 8086's advantage over the 8088 is attributable to its 16-bit external data bus. In applications that manipulate 8-bit quantities extensively, or that are execution-bound, the 8088 can approach to within 10% of the 8086's processing throughput.

The high performance of the 8086 and 8088 is realized by combining a 16-bit internal data path with a pipelined architecture that allows instructions to be prefetched during spare bus cycles. Also contributing to performance is a compact instruction format that enables more instructions to be fetched in a given amount of time.

Software for high-performance 8086 and 8088 systems need not be written in assembly language. The CPUs are designed to provide direct hardware support for programs written in high-level languages such as Intel's PL/M-86. Most high-level languages store variables in memory; the 8086/8088 symmetrical instruction set supports direct operation on memory operands, including operands on the stack. The hardware addressing modes provide efficient, straightforward implementations of based variables, arrays, arrays of structures and other high-level language data constructs. A powerful set of memory-to-memory string operations is available for efficient character data manipulation. Finally, routines with critical performance requirements that cannot be met with PL/M-86 may be written in ASM-86 (the 8086/8088 assembly language) and linked with PL/M-86 code.

While the 8086 and 8088 are totally new designs, they make the most of users' existing investments in systems designed around the 8080/8085 microprocessors. Many of the standard Intel memory, peripheral control and communication chips are compatible with the 8086 and the 8088. Software is developed in the familiar Intellec® Microcomputer Development System environment, and most existing programs, whether written in ASM-80 or PL/M-80, can be directly converted to run on the 8086 and 8088.

2.2 Processor Architecture

Microprocessors generally execute a program by repeatedly cycling through the steps shown below (this description is somewhat simplified):

1. Fetch the next instruction from memory.
2. Read an operand (if required by the instruction).

3. Execute the instruction.
4. Write the result (if required by the instruction).

In previous CPUs, most of these steps have been performed serially, or with only a single bus cycle fetch overlap. The architecture of the 8086 and 8088 CPUs, while performing the same steps, allocates them to two separate processing units within the CPU. The execution unit (EU) executes instructions; the bus interface unit (BIU) fetches instructions, reads operands and writes results. The two units can operate independently of one another and are able, under most circumstances, to extensively overlap instruction fetch with execution. The result is that, in most cases, the time normally required to fetch instructions "disappears" because the EU executes instructions that have already been fetched by the BIU. Figure 2-5 illustrates this overlap and compares it with traditional microprocessor operation. In the example, overlapping reduces the elapsed time required to execute three instructions, and allows two additional instructions to be prefetched as well.

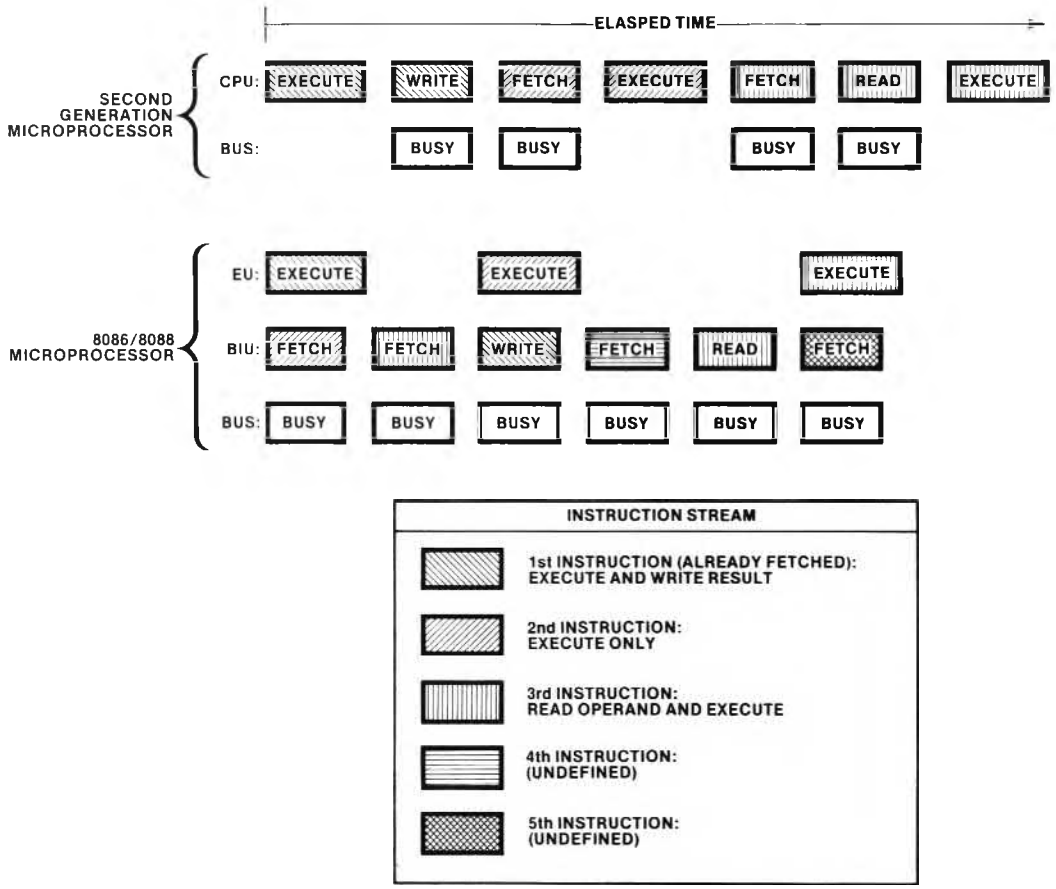


Figure 2-5. Overlapped Instruction Fetch and Execution

Execution Unit

The execution units of the 8086 and 8088 are identical (figure 2-6). A 16-bit arithmetic/logic unit (ALU) in the EU maintains the CPU status and control flags, and manipulates the general registers and instruction operands. All registers and data paths in the EU are 16 bits wide for fast internal transfers.

The EU has no connection to the system bus, the "outside world." It obtains instructions from a queue maintained by the BIU. Likewise, when an instruction requires access to memory or to a peripheral device, the EU requests the BIU to obtain or store the data. All addresses manipulated by the EU are 16 bits wide. The BIU, however, performs an address relocation that gives the EU access to the full megabyte of memory space (see section 2.3).

Bus Interface Unit

The BIUs of the 8086 and 8088 are functionally identical, but are implemented differently to match the structure and performance characteristics of their respective buses.

The BIU performs all bus operations for the EU. Data is transferred between the CPU and memory or I/O devices upon demand from the EU. Sections 2.3 and 2.4 describe the interaction of the BIU with memory and I/O devices.

In addition, during periods when the EU is busy executing instructions, the BIU "looks ahead" and fetches more instructions from memory. The instructions are stored in an internal RAM array called the instruction stream queue. The 8088 instruction queue holds up to four bytes of the instruction stream, while the 8086 queue can store

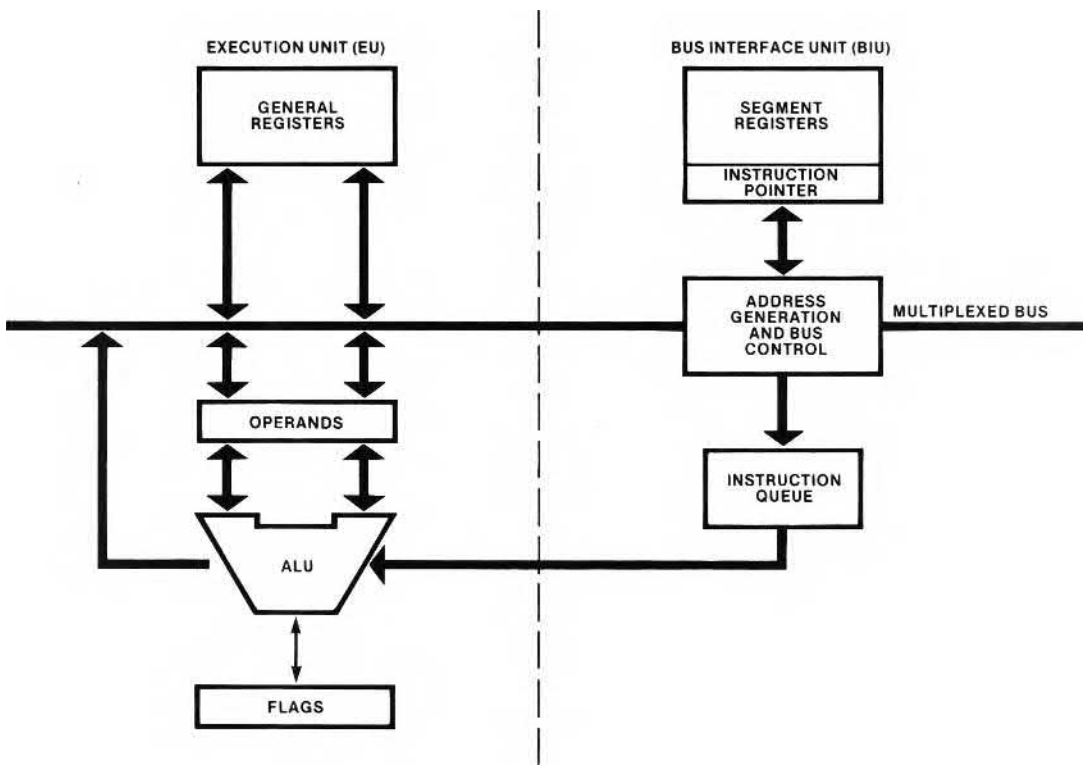


Figure 2-6. Execution and Bus Interface Units (EU and BIU)

up to six instruction bytes. These queue sizes allow the BIU to keep the EU supplied with prefetched instructions under most conditions without monopolizing the system bus. The 8088 BIU fetches another instruction byte whenever one byte in its queue is empty and there is no active request for bus access from the EU. The 8086 BIU operates similarly except that it does not initiate a fetch until there are two empty bytes in its queue. The 8086 BIU normally obtains two instruction bytes per fetch; if a program transfer forces fetching from an odd address, the 8086 BIU automatically reads one byte from the odd address and then resumes fetching two-byte words from the subsequent even addresses.

Under most circumstances the queues contain at least one byte of the instruction stream and the EU does not have to wait for instructions to be fetched. The instructions in the queue are those stored in the memory locations immediately adjacent to and higher than the instruction currently being executed. That is, they are the next logical instructions so long as execution proceeds serially. If the EU executes an instruction that transfers control to another location, the BIU resets the queue, fetches the instruction from the new address, passes it immediately to the EU, and then begins refilling the queue from the new location. In addition, the BIU suspends instruction fetching whenever the EU requests a memory or I/O read or write (except that a fetch already in progress is completed before executing the EU's bus request).

General Registers

Both CPUs have the same complement of eight 16-bit general registers (figure 2-7). The general registers are subdivided into two sets of four registers each: the data registers (sometimes called the H & L group for "high" and "low"), and the pointer and index registers (sometimes called the P & I group).

The data registers are unique in that their upper (high) and lower halves are separately addressable. This means that each data register can be used interchangeably as a 16-bit register, or as two 8-bit registers. The other CPU registers always are accessed as 16-bit units only. The data registers can be used without constraint in most arithmetic and logic operations. In addition,

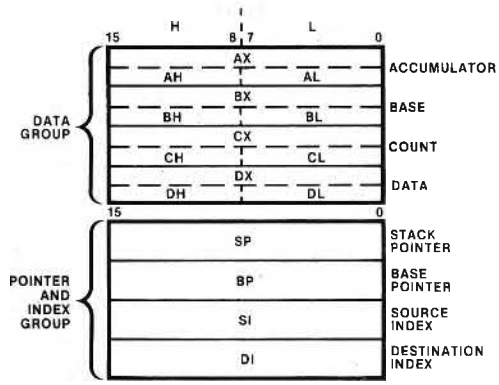


Figure 2-7. General Registers

some instructions use certain registers implicitly (see table 2-1) thus allowing compact yet powerful encoding.

Table 2-1. Implicit Use of General Registers

REGISTER	OPERATIONS
AX	Word Multiply, Word Divide, Word I/O
AL	Byte Multiply, Byte Divide, Byte I/O, Translate, Decimal Arithmetic
AH	Byte Multiply, Byte Divide
BX	Translate
CX	String Operations, Loops
CL	Variable Shift and Rotate
DX	Word Multiply, Word Divide, Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

The pointer and index registers can also participate in most arithmetic and logic operations. In fact, all eight general registers fit the definition of "accumulator" as used in first and second generation microprocessors. The P & I registers (except for BP) also are used implicitly in some instructions as shown in table 2-1.

Segment Registers

The megabyte of 8086 and 8088 memory space is divided into logical segments of up to 64k bytes each. (Memory segmentation is described in section 2.3.) The CPU has direct access to four segments at a time; their base addresses (starting locations) are contained in the segment registers (see figure 2-8). The CS register points to the current code segment; instructions are fetched from this segment. The SS register points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data segment; it generally contains program variables. The ES register points to the current extra segment, which also is typically used for data storage.

The segment registers are accessible to programs and can be manipulated with several instructions. Good programming practice and consideration of compatibility with future Intel hardware and software products dictate that the segment registers be used in a disciplined fashion. Section 2.10 provides guidelines for segment register use.

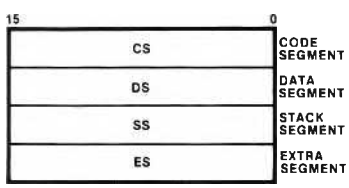


Figure 2-8. Segment Registers

Instruction Pointer

The 16-bit instruction pointer (IP) is analogous to the program counter (PC) in the 8080/8085 CPUs. The instruction pointer is updated by the BIU so that it contains the offset (distance in bytes) of the next instruction from the beginning of the current code segment; i.e., IP points to the next instruction. During normal execution, IP contains the offset of the next instruction to be *fetched* by the BIU; whenever IP is saved on the stack, however, it first is automatically adjusted to point to the next instruction to be *executed*. Programs do not have direct access to the instruction pointer, but instructions cause it to change and to be saved on and restored from the stack.

Flags

The 8086 and 8088 have six 1-bit status flags (figure 2-9) that the EU posts to reflect certain properties of the result of an arithmetic or logic operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags, that is, on the result of a prior operation. Different instructions affect the status flags differently; in general, however, the flags reflect the following conditions:

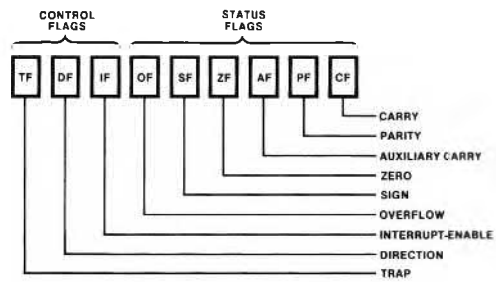


Figure 2-9. Flags

1. If AF (the auxiliary carry flag) is set, there has been a carry out of the low nibble into the high nibble or a borrow from the high nibble into the low nibble of an 8-bit quantity (low-order byte of a 16-bit quantity). This flag is used by decimal arithmetic instructions.
2. If CF (the carry flag) is set, there has been a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit). The flag is used by instructions that add and subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.
3. If OF (the overflow flag) is set, an arithmetic overflow has occurred; that is, a significant digit has been lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that will generate an interrupt in this situation.

4. If SF (the sign flag) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented in the 8086 and 8088 in standard two's complement notation, SF indicates the sign of the result (0 = positive, 1 = negative).
5. If PF (the parity flag) is set, the result has even parity, an even number of 1-bits. This flag can be used to check for data transmission errors.
6. If ZF (the zero flag) is set, the result of the operation is 0.

Three additional control flags (figure 2-9) can be set and cleared by programs to alter processor operations:

1. Setting DF (the direction flag) causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses, or from "right to left." Clearing DF causes string instructions to auto-increment, or to process strings from "left to right."
2. Setting IF (the interrupt-enable flag) allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no affect on either non-maskable external or internally generated interrupts.
3. Setting TF (the trap flag) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction. Section 2.10 contains an example showing the use of TF in a single-step and breakpoint routine.

8080/8085 Registers and Flag Correspondence

The registers, flags and program counter in the 8080/8085 CPUs all have counterparts in the 8086 and 8088 (see figure 2-10). The A register (accumulator) in the 8080/8085 corresponds to the AL register in the 8086 and 8088. The 8080/8085 H & L, B & C, and D & E registers correspond to registers BH, BL, CH, CL, DH and DL, respectively, in the 8086 and 8088. The 8080/8085 SP (stack pointer) and PC (program counter) have their counterparts in the 8086/8088 SP and IP. The AF, CF, PF, SF, and ZF flags are the same in both CPU families. The remaining flags and registers are unique to the 8086 and 8088. This 8080/8085 to 8086 mapping allows most existing 8080/8085 program code to be directly translated into 8086/8088 code.

Mode Selection

Both processors have a strap pin (MN/$\overline{MX}$) that defines the function of eight CPU pins in the 8086 and nine pins in the 8088. Connecting MN/$\overline{MX}$ to +5V places the CPU in minimum mode. In this configuration, which is designed for small systems (roughly one or two boards), the CPU itself provides the bus control signals needed by memory and peripherals. When MN/$\overline{MX}$ is strapped to ground, the CPU is configured in maximum mode. In this configuration the CPU encodes control signals on three lines. An 8288 Bus Controller is added to decode the signals from the CPU and to provide an expanded set of control signals to the rest of the system. The CPU uses the remaining free lines for a new set of signals designed to help coordinate the activities of other processors in the system. Sections 2.5 and 2.6 describe the functions of these signals.

2.3 Memory

The 8086 and 8088 can accommodate up to 1,048,576 bytes of memory in both minimum and maximum mode. This section describes how memory is functionally organized and used. There are substantial differences in the way memory components are actually accessed by the two processors; these differences, which are invisible to programs, are covered in section 4.2, External Memory Addressing.

Storage Organization

From a storage point of view, the 8086 and 8088 memory spaces are organized as identical arrays of 8-bit bytes (see figure 2-11). Instructions, byte data and word data may be freely stored at any byte address without regard for alignment thereby saving memory space by allowing code to be densely packed in memory (see figure 2-12). Odd-addressed (unaligned) word variables, however,

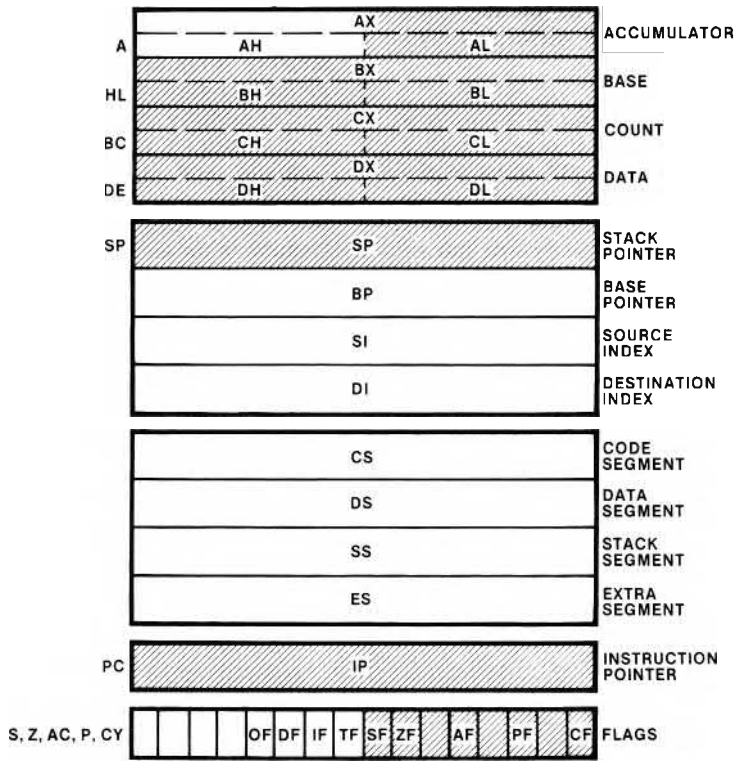


Figure 2-10. 8080/8085 Register Subset (Shaded)

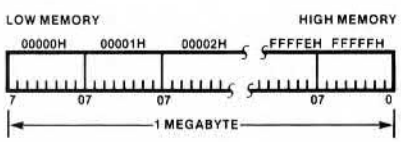


Figure 2-11. Storage Organization

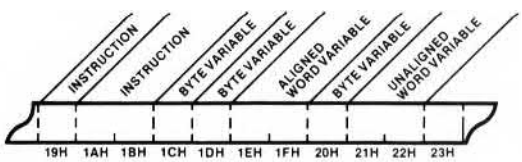


Figure 2-12. Instruction and Variable Storage

do not take advantage of the 8086's ability to transfer 16-bits at a time. Instruction alignment does not materially affect the performance of either processor.

Following Intel convention, word data always is stored with the most-significant byte in the higher memory location (see figure 2-13). Most of the time this storage convention is "invisible" to anyone working with the processors; exceptions may occur when monitoring the system bus or when reading memory dumps.

A special class of data is stored as doublewords; i.e., two consecutive words. These are called pointers and are used to address data and code that are outside the currently-addressable segments. The lower-addressed word of a pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally with the higher-addressed byte containing the most-significant eight bits of the word (see figure 2-14).

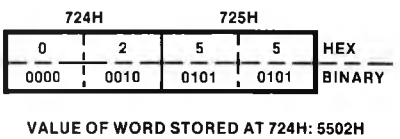


Figure 2-13. Storage of Word Variables

Segmentation

8086 and 8088 programs "view" the megabyte of memory space as a group of segments that are defined by the application. A segment is a logical unit of memory that may be up to 64k bytes long. Each segment is made up of contiguous memory locations and is an independent, separately-addressable unit. Every segment is assigned (by software) a base address, which is its starting location in the memory space. All segments begin on 16-byte memory boundaries. There are no other restrictions on segment locations; segments may be adjacent, disjoint, partially overlapped, or fully overlapped (see figure 2-15). A physical memory location may be mapped into (contained in) one or more logical segments.

The segment registers point to (contain the base address values of) the four currently addressable segments (see figure 2-16). Programs obtain access to code and data in other segments by changing the segment registers to point to the desired segments.

Every application will define and use segments differently. The currently addressable segments provide a generous work space: 64k bytes for code, a 64k byte stack and 128k bytes of data storage. Many applications can be written to simply initialize the segment registers and then forget them. Larger applications should be designed with careful consideration given to segment definition.

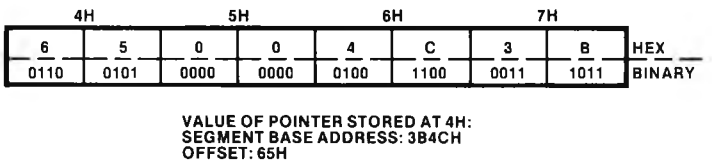


Figure 2-14. Storage of Pointer Variables

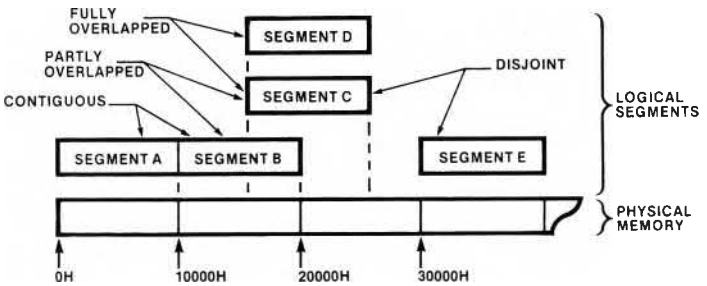


Figure 2-15. Segment Locations in Physical Memory

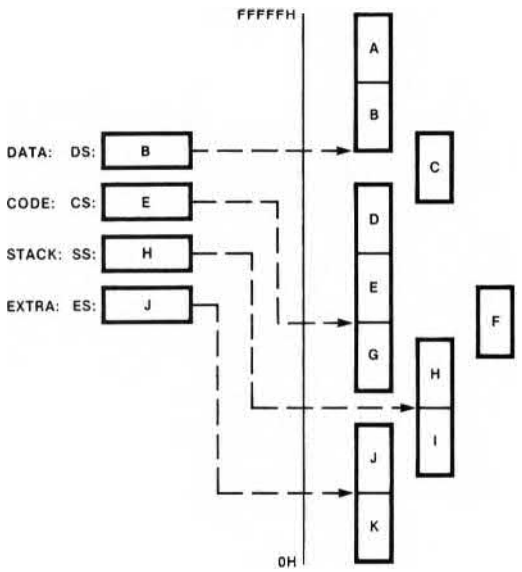


Figure 2-16. Currently Addressable Segments

The segmented structure of the 8086/8088 memory space supports modular software design by discouraging huge, monolithic programs. The segments also can be used to advantage in many programming situations. Take, for example, the case of an editor for several on-line terminals. A 64k text buffer (probably an extra segment) could be assigned to each terminal. A single program could maintain all the buffers by simply changing register ES to point to the buffer of the terminal requiring service.

Physical Address Generation

It is useful to think of every memory location as having two kinds of addresses, physical and logical. A physical address is the 20-bit value that uniquely identifies each byte location in the megabyte memory space. Physical addresses may range from 0H through FFFFFH. All exchanges between the CPU and memory components use this physical address.

Programs deal with logical, rather than physical addresses and allow code to be developed without prior knowledge of where the code is to be located in memory and facilitate dynamic management of memory resources. A logical address consists of a segment base value and an offset value. For any given memory location, the segment base value

locates the first byte of the containing segment and the offset value is the distance, in bytes, of the target location from the beginning of the segment. Segment base and offset values are unsigned 16-bit quantities; the lowest-addressed byte in a segment has an offset of 0. Many different logical addresses can map to the same physical location as shown in figure 2-17. In figure 2-17, physical memory location 2C3H is contained in two different overlapping segments, one beginning at 2B0H and the other at 2C0H.

Whenever the BIU accesses memory—to fetch an instruction or to obtain or store a variable—it generates a physical address from a logical address. This is done by shifting the segment base value four bit positions and adding the offset as illustrated in figure 2-18. Note that this addition process provides for modulo 64k addressing (addresses wrap around from the end of a segment to the beginning of the same segment).

The BIU obtains the logical address of a memory location from different sources depending on the type of reference that is being made (see table 2-2). Instructions always are fetched from the current code segment; IP contains the offset of the target instruction from the beginning of the segment. Stack instructions always operate on the current stack segment; SP contains the offset of the top of the stack. Most variables (memory operands) are assumed to reside in the current data segment, although a program can instruct the BIU to access a variable in one of the other currently addressable segments. The offset of a memory variable is calculated by the EU. This calculation is based on the addressing mode specified in the instruction; the result is called the operand's effective address (EA). Section 2.8 covers addressing modes and effective address calculation in detail.

Strings are addressed differently than other variables. The source operand of a string instruction is assumed to lie in the current data segment, but another currently addressable segment may be specified. Its offset is taken from register SI, the source index register. The destination operand of a string instruction always resides in the current

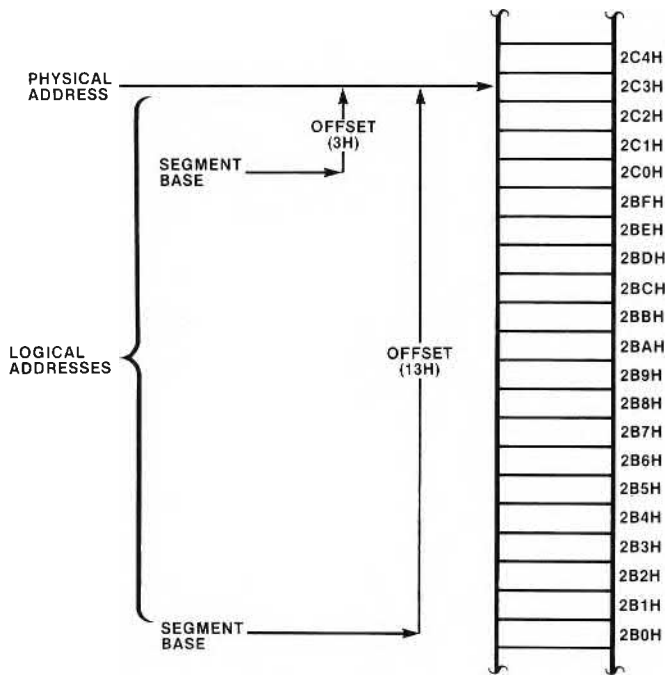


Figure 2-17. Logical and Physical Addresses

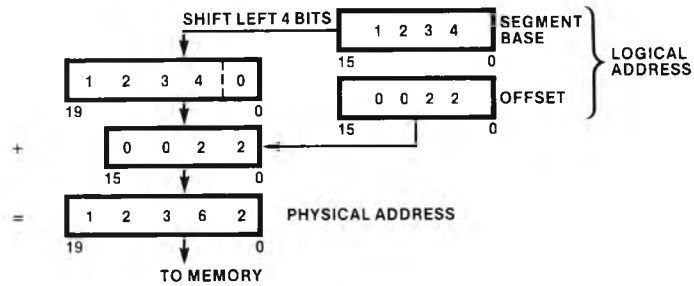


Figure 2-18. Physical Address Generation

Table 2-2. Logical Address Sources

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction Fetch	CS	NONE	IP
Stack Operation	SS	NONE	SP
Variable (except following)	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	NONE	DI
BP Used As Base Register	SS	CS,DS,ES	Effective Address

extra segment; its offset is taken from DI, the destination index register. The string instructions automatically adjust SI and DI as they process the strings one byte or word at a time.

When register BP, the base pointer register, is designated as a base register in an instruction, the variable is assumed to reside in the current stack segment. Register BP thus provides a convenient way to address data on the stack; BP can be used, however, to access data in any of the other currently addressable segments.

In most cases, the BIU's segment assumptions are a convenience to programmers. It is possible, however, for a programmer to explicitly direct the BIU to access a variable in any of the currently addressable segments (the only exception is the destination operand of a string instruction which must be in the extra segment). This is done by preceding an instruction with a segment override prefix. This one-byte machine instruction tells the BIU which segment register to use to access a variable referenced in the following instruction.

Dynamically Relocatable Code

The segmented memory structure of the 8086 and 8088 makes it possible to write programs that are position-independent, or dynamically relocatable. Dynamic relocation allows a multiprogramming or multitasking system to make particularly effective use of available memory. Inactive programs can be written to disk and the space they occupied allocated to other programs. If a disk-resident program is needed later, it can be read back into any available memory location and restarted. Similarly, if a program needs a large contiguous block of storage, and the total amount is available only in nonadjacent fragments, other program segments can be compacted to free up a continuous space. This process is shown graphically in figure 2-19.

In order to be dynamically relocatable, a program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment. In other words, all offsets in the program must be relative to fixed values

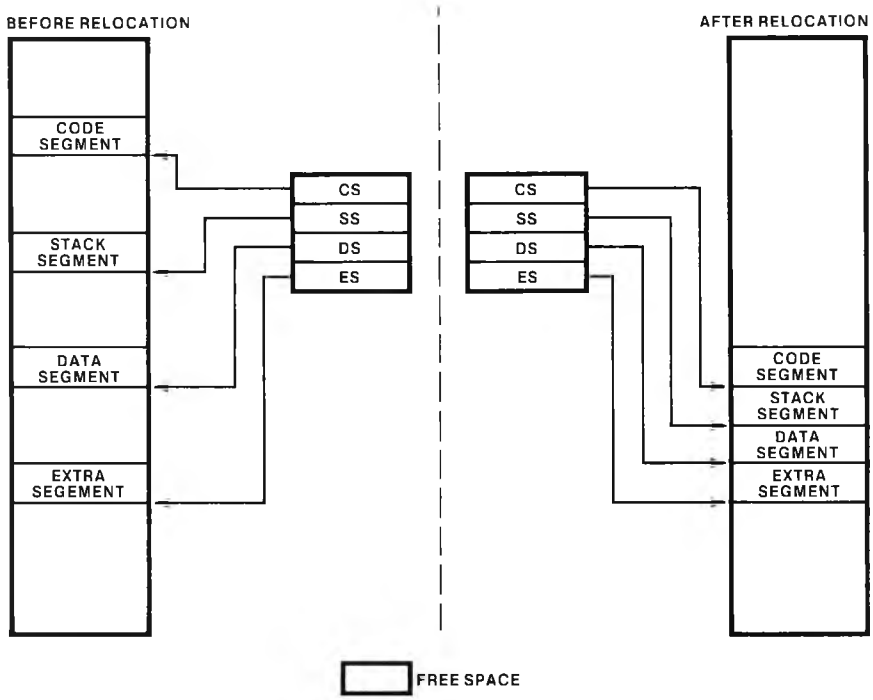


Figure 2-19. Dynamic Code Relocation

contained in the segment registers. This allows the program to be moved anywhere in memory as long as the segment registers are updated to point to the new base addresses. Section 2.10 contains an example that illustrates dynamic code relocation.

Stack Implementation

Stacks in the 8086 and 8088 are implemented in memory and are located by the stack segment register (SS) and the stack pointer register (SP). A system may have an unlimited number of stacks, and a stack may be up to 64k bytes long, the maximum length of a segment. (An attempt to expand a stack beyond 64k bytes overwrites the beginning of the stack.) One stack is directly addressable at a time; this is the current stack, often referred to simply as "the" stack. SS contains the base address of the current stack and SP points to the top of the stack (TOS). In other words, SP contains the offset of the top of the stack from the stack segment's base address. Note, however, that the stack's base address (contained in SS) is not the "bottom" of the stack.

8086 and 8088 stacks are 16 bits wide; instructions that operate on a stack add and remove stack items one word at a time. An item is pushed onto the stack (see figure 2-20) by *decrementing* SP by 2 and writing the item at the new TOS. An item is popped off the stack by copying it from TOS and then *incrementing* SP by 2. In other words, the stack grows *down* in memory toward its base address. Stack operations never move items on the stack, nor do they erase them. The top of the stack changes only as a result of updating the stack pointer.

Dedicated and Reserved Memory Locations

Two areas in extreme low and high memory are dedicated to specific processor functions or are reserved by Intel Corporation for use by Intel

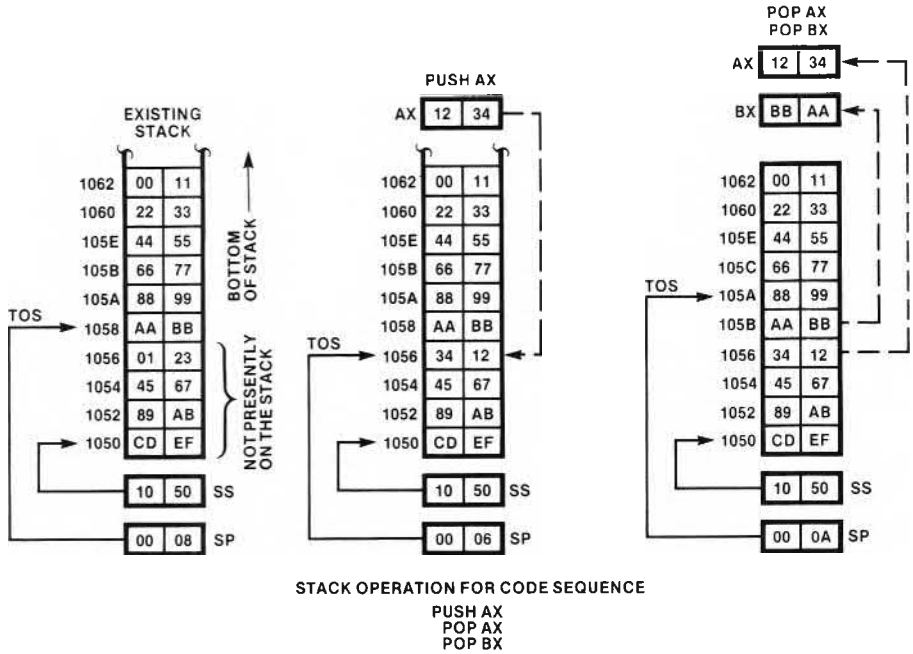


Figure 2-20. Stack Operation

hardware and software products. As shown in figure 2-21, the location are: 0H throgh 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes). These areas are used for interrupt and system reset processing 8086 and 8088 application systems should not use these areas for any other purpose. Doing so may make these systems incompatible with future Intel products.

8086/8088 Memory Access Differences

The 8086 can access either 8 or 16 bits of memory at a time. If an instruction refers to a word variable and that variable is located at an even-numbered address, the 8086 accesses the complete word in one bus cycle. If the word is located at an odd-numbered address, the 8086 accesses the word one byte at a time in two consecutive bus cycles.

To maximize throughput in 8086-based systems, 16-bit data should be stored at even addresses (should be word-aligned). This is particularly true of stacks. Unaligned stacks can slow a system's response to interrupts. Nevertheless, except for the performance penalty, word alignment is totally transparent to software. This allows maximum data packing where memory space is constrained.

The 8086 always fetches the instruction stream in words from even addresses except that the first fetch after a program transfer to an odd address obtains a byte. The instruction stream is disassembled inside the processor and instruction alignment will not materially affect the performance of most systems.

The 8088 always accesses memory in bytes. Word operands are accessed in two bus cycles regardless of their alignment. Instructions also are fetched one byte at a time. Although alignment of word operands does not affect the performance of the 8088, locating 16-bit data on even addresses will insure maximum throughput if the system is ever transferred to an 8086.

2.4 Input/Output

The 8086 and 8088 have a versatile set of input/output facilities. Both processors provide a large I/O space that is separate from the memory

Mnemonics © Intel, 1978

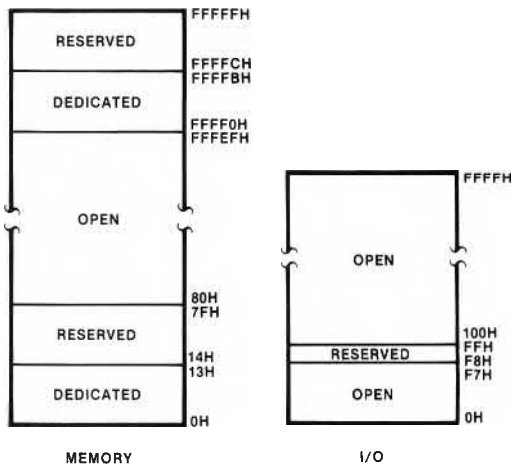


Figure 2-21. Reserved and Dedicated Memory and I/O Locations

space, and instructions that transfer data between the CPU and devices located in the I/O space. I/O devices also may be placed in the memory space to bring the power of the full instruction set and addressing modes to input/output processing. For high-speed transfers, the CPUs may be used with traditional direct memory access controllers or the 8089 Input/Output Processor.

Input/Output Space

The 8086/8088 I/O space can accommodate up to 64k 8-bit ports or up to 32k 16-bit ports. The IN and OUT (input and output) instructions transfer data between the accumulator (AL for byte transfers, AX for word transfers) and ports located in the I/O space.

The I/O space is not segmented; to access a port, the BIU simply places the port address (0-64k) on the lower 16 lines of the address bus. Different forms of the I/O instructions allow the address to be specified as a fixed value in the instruction or as a variable taken from register DX.

Restricted I/O Locations

Locations F8H through FFH (eight of the 64k locations) in the I/O space are reserved by Intel Corporation for use by future Intel hardware and software products. Using these locations for any other purpose may inhibit compatibility with future Intel products.

8086/8088 I/O Access Differences

The 8086 can transfer either 8 or 16 bits at a time to a device located in the I/O space. A 16-bit device should be located at an even address so that the word will be transferred in a single bus cycle. An 8-bit device may be located at either an even or odd address; however, the internal registers in a given device must be assigned all-even or all-odd addresses.

The 8088 transfers one byte per bus cycle. If a 16-bit device is used in the 8088 I/O space, it must be capable of transferring words in the same fashion, i.e., eight bits at a time in two bus cycles. (The 8089 Input/Output Processor can provide a straightforward interface between the 8088 and a 16-bit I/O device.) An 8-bit device may be located at odd or even addresses in the 8088 I/O space and internal registers may be assigned consecutive addresses (e.g., 1H, 2H, 3H). Assigning all-odd or all-even addresses to these registers, however, will simplify transferring the system to an 8086 CPU.

Memory-Mapped I/O

I/O devices also may be placed in the 8086/8088 memory space. As long as the devices respond like memory components, the CPU does not know the difference.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV (move) instruction can transfer data between any 8086/8088 register and a port, or the AND, OR and TEST instructions may be used to manipulate bits in I/O device registers. In addition, memory-mapped I/O can take advantage of the 8086/8088 memory addressing modes. A group of terminals, for example, could be treated as an array in memory with an index register

selecting a terminal in the array. Section 2.10 provides examples of using the instruction set and addressing modes with memory-mapped I/O.

Of course, a price must be paid for the added programming flexibility that memory-mapped I/O provides. Dedicating part of the memory space to I/O devices reduces the number of addresses available for memory, although with a megabyte of memory space this should rarely be a constraint. Memory reference instructions also take longer to execute and are somewhat less compact than the simpler IN and OUT instructions.

Direct Memory Access

When configured in minimum mode, the 8086 and 8088 provide HOLD (hold) and HLDA (hold acknowledge) signals that are compatible with traditional DMA controllers such as the 8257 and 8237. A DMA controller can request use of the bus for direct transfer of data between an I/O device and memory by activating HOLD. The CPU will complete the current bus cycle, if one is in progress, and then issue HLDA, granting the bus to the DMA controller. The CPU will not attempt to use the bus until HOLD goes inactive.

The 8086 addresses memory that is physically organized in two separate banks, one containing even-addressed bytes and one containing odd-addressed bytes. An 8-bit DMA controller must alternately select these banks to access logically adjacent bytes in memory. The 8089 provides a simple way to interface a high-speed 8-bit device to an 8086-based system (see Chapter 3).

8089 Input/Output Processor (IOP)

The 8086 and 8088 are designed to be used with the 8089 in high-performance I/O applications. The 8089 conceptually resembles a microprocessor with two DMA channels and an instruction set specifically tailored for I/O operations. Unlike simple DMA controllers, the 8089 can service I/O devices directly, removing this task from the CPU. In addition, it can transfer data on its own bus or on the system bus, can match 8- or 16-bit peripherals to 8- or 16-bit buses, and can transfer data from memory to memory and from I/O device to I/O device. Chapter 3 describes the 8089 in detail.

2.5 Multiprocessing Features

As microprocessor prices have declined, multiprocessing (using two or more coordinated processors in a system) has become an increasingly attractive design alternative. Performance can be substantially improved by distributing system tasks among separate, concurrently executing processors. In addition, multiprocessing encourages a modular approach to design, usually resulting in systems that are more easily maintained and enhanced. For example, figure 2-22 shows a multiprocessor system in which I/O activities have been delegated to an 8089 IOP. Should an I/O device in the system be changed (e.g., a hard disk substituted for a floppy), the impact of the modification is confined to the I/O subsystem and is transparent to the CPU and to the application software.

The 8086 and 8088 are designed for the multiprocessing environment. They have built-in features that help solve the coordination problems that have discouraged multiprocessing system development in the past.

Bus Lock

When configured in maximum mode, the 8086 and 8088 provide the $\overline{\text{LOCK}}$ (bus lock) signal. The BIU activates $\overline{\text{LOCK}}$ when the EU executes the one-byte LOCK prefix instruction. The $\overline{\text{LOCK}}$ signal remains active throughout execution of the instruction that follows the LOCK prefix. Interrupts are *not* affected by the LOCK prefix. If another processor requests use of the bus (via the request/grant lines, which are discussed shortly), the CPU records the request, but does not honor it until execution of the locked instruction has been completed.

Note that the $\overline{\text{LOCK}}$ signal remains active for the duration of a *single* instruction. If two consecutive instructions are each preceded by a LOCK prefix, there will still be an unlocked period between these instructions. In the case of a locked repeated string instruction, $\overline{\text{LOCK}}$ does remain active for the duration of the block operation.

When the 8086 or 8088 is configured in minimum mode, the $\overline{\text{LOCK}}$ signal is not available. The LOCK prefix can be used, however, to delay the

Mnemonics © Intel, 1978

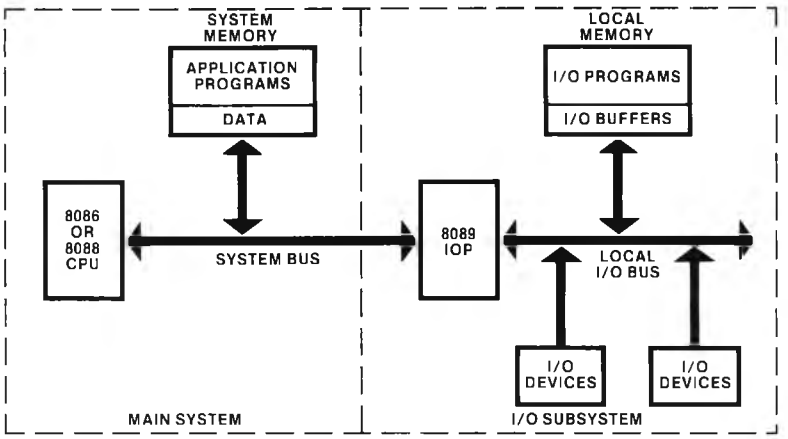


Figure 2-22. Multiprocessing System

generation of an HLDA response to a HOLD request until execution of the locked instruction is completed.

The $\overline{\text{LOCK}}$ signal provides information only. It is the responsibility of other processors on the shared bus to not attempt to obtain the bus while $\overline{\text{LOCK}}$ is active. If the system uses 8289 Bus Arbiters to control access to the shared bus, the 8289's accept $\overline{\text{LOCK}}$ as an input and do not relinquish the bus while this signal is active.

$\overline{\text{LOCK}}$ may be used in multiprocessing systems to coordinate access to a common resource, such as a buffer or a pointer. If access to the resource is not controlled, one processor can read an erroneous value from the resource when another processor is updating it (see figure 2-23).

Access can be controlled (see figure 2-24) by using the LOCK prefix in conjunction with the XCHG (exchange register with memory) instruction. The basis for controlling access to a given resource is a semaphore, a software-settable flag or switch that indicates whether the resource is "available" (semaphore=0) or "busy" (semaphore=1). Processors that share the bus agree by convention not to use the resource unless the semaphore indicates that it is available. They likewise agree to set the semaphore when they are using the resource and to clear it when they are finished.

The XCHG instruction can obtain the current value of the semaphore and set it to "busy" in a single instruction. The instruction, however, requires two bus cycles to swap 8-bit values. It is possible for another processor to obtain the bus between these two cycles and to gain access to the partially-updated semaphore. This can be prevented by preceding the XCHG instruction with a LOCK prefix, as illustrated in figure 2-25. The bus lock establishes control over access to the semaphore and thus to the shared resource.

WAIT and $\overline{\text{TEST}}$

The 8086 and 8088 (in either maximum or minimum mode) can be synchronized to an external event with the WAIT (wait for $\overline{\text{TEST}}$) instruction and the $\overline{\text{TEST}}$ input signal. When the EU executes a WAIT instruction, the result depends on the state of the $\overline{\text{TEST}}$ input line. If $\overline{\text{TEST}}$ is inactive, the processor enters an idle state and repeatedly retests the $\overline{\text{TEST}}$ line at five-clock intervals. If $\overline{\text{TEST}}$ is active, execution continues with the instruction following the WAIT.

BUS CYCLE	SHARED POINTER IN MEMORY	PROCESSOR ACTIVITIES
0	05, 22 | 4C, 1B	
1	C2, 59 | 4C, 1B	"A" UPDATES 1 WORD
2	C2, 59 | 4C, 1B	"B" READS PARTIALLY UPDATED VALUE
3	C2, 59 | 31, 05	"A" COMPLETES UPDATE

Figure 2-23. Uncontrolled Access to Shared Resource

Escape

The ESC (escape) instruction provides a way for another processor to obtain an instruction and/or a memory operand from an 8086/8088 program. When used in conjunction with WAIT and TEST, ESC can initiate a "subroutine" that executes concurrently in another processor (see figure 2-26).

Six bits in the ESC instruction may be specified by the programmer when the instruction is written. By monitoring the 8086/8088 bus and control lines, another processor can capture the ESC instruction when it is fetched by the BIU. The six bits may then direct the external processor to perform some predefined activity.

If the 8086/8088 is configured in maximum mode, the external processor, having determined that an ESC has been fetched, can monitor QS0

BUS CYCLE	SEMAPHORE	SHARED POINTER IN MEMORY	PROCESSOR ACTIVITIES
0	0	05, 22 | 4C, 1B	
1	1	05, 22 | 4C, 1B	"A" OBTAINS EXCLUSIVE USE
2	1	C2, 59 | 4C, 1B	"A" UPDATES 1 WORD
3	1	C2, 59 | 4C, 1B	"B" TESTS SEMAPHORE AND WAITS
4	1	C2, 59 | 31, 05	"A" COMPLETES UPDATE
5	1	C2, 59 | 31, 05	"B" TESTS SEMAPHORE AND WAITS
6	0	C2, 59 | 31, 05	"A" RELEASES RESOURCE
7	1	C2, 59 | 31, 05	"B" OBTAINS EXCLUSIVE USE
8	1	C2, 59 | 31, 05	"B" READS UPDATED VALUE
9	0	C2, 59 | 31, 05	"B" RELEASES RESOURCE

Figure 2-24. Controlled Access to Shared Resource

Mnemonics © Intel, 1978

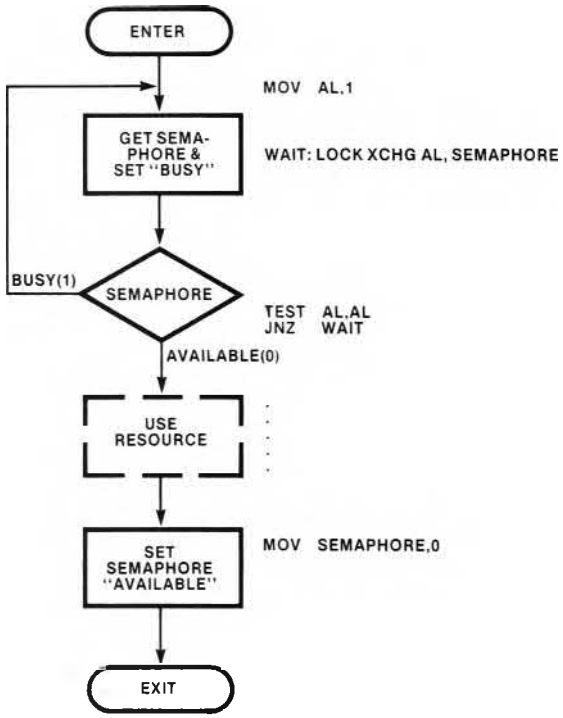


Figure 2-25. Using XCHG and LOCK

and QS1 (the queue status lines, discussed in section 2.6) and determine when the ESC instruction is executed. If the instruction references memory the external processor can then monitor the bus and capture the operand's physical address and/or the operand itself.

Note that fetching an ESC instruction is not tantamount to executing it. The ESC may be preceded by a jump that causes the queue to be reinitialized. This event also can be determined from the queue status lines.

Request/Grant Lines

When the 8086 or 8088 is configured in maximum mode, the HOLD and HLDA lines evolve into two more sophisticated signals called $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$. These are bidirectional lines that can be used to share a local bus between an 8086 or 8088 and two other processors via a handshake sequence.

The request/grant sequence is a three-phase cycle: request, grant and release. First, the processor desiring the bus pulses a request/grant line. The CPU returns a pulse on the same line indicating that it is entering the "hold acknowledge" state and is relinquishing the bus. The BIU is logically disconnected from the bus during this period. The

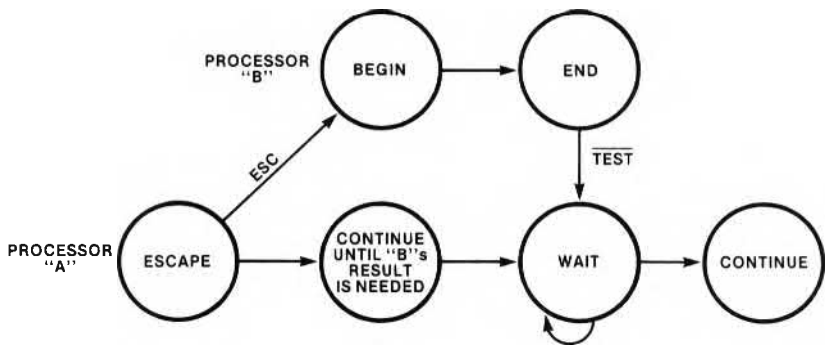


Figure 2-26. Using ESC with WAIT and TEST

EU, however, will continue to execute instructions until an instruction requires bus access or the queue is emptied, whichever occurs first. When the other processor has finished with the bus, it sends a final pulse to the 8086/8088 indicating that the request has ended and that the CPU may reclaim the bus.

$\overline{RQ}/\overline{GT0}$ has higher priority than $\overline{RQ}/\overline{GT1}$. If requests arrive simultaneously on both lines, the grant goes to the processor on $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$ is acknowledged after the bus has been returned to the CPU. If, however, a request arrives on $\overline{RQ}/\overline{GT0}$ while the CPU is processing a prior request on $\overline{RQ}/\overline{GT1}$, the second request is not honored until the processor on $\overline{RQ}/\overline{GT1}$ releases the bus.

Multibus™ Architecture

Intel has designed a general-purpose multiprocessing bus called the Multibus. This is the standard design used in iSBC™ single-board microcomputer products. Many other manufacturers offer products that are compatible with the Multibus architecture as well. When the 8086 and 8088 are configured in maximum mode, the 8288 Bus Controller outputs signals that are electrically compatible with the Multibus protocol. Designers of multiprocessing systems may want to consider using the Multibus architecture in the design of their products to reduce development cost and time, and to obtain compatibility with the wide variety of boards available in the iSBC product line.

The Multibus architecture provides a versatile communications channel that can be used to coordinate a wide variety of computing modules (see figure 2-27). Modules in a Multibus system are designated as masters or slaves. Masters may obtain use of the bus and initiate data transfers on it. Slaves are the objects of data transfers only. The Multibus architecture allows both 8- and 16-bit masters to be intermixed in a system. In addition to 16 data lines, the bus design provides 20 address lines, eight multilevel interrupt lines, and control and arbitration lines. An auxiliary power bus also is provided to route standby power to memories if the normal supply fails.

The Multibus architecture maintains its own clock, independent of the clocks of the modules it links together. This allows different speed masters to share the bus and allows masters to operate asynchronously with respect to each other. The arbitration logic of the bus permit slow-speed masters to compete equably for use of the bus. Once a module has obtained the bus, however, transfer speeds are dependent only on the capabilities of the transmitting and receiving modules. Finally, the Multibus standard defines the form factors and physical requirements of modules that communicate on this bus. For a complete description of the Multibus architec-

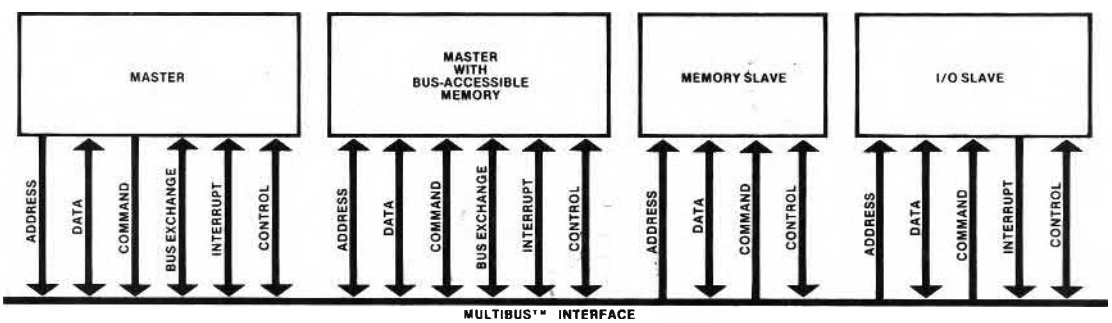


Figure 2-27. Multibus™-Based System

ture, refer to the Intel Multibus Specification (document number 9800683) and Application Note 28A, "Intel Multibus Interfacing."

8289 Bus Arbiter

Multiprocessor systems require a means of coordinating the processors' use of the shared bus. The 8289 Bus Arbiter works in conjunction with the 8288 Bus Controller to provide this control for 8086- and 8088-based systems. It is compatible with the Multibus architecture and can be used in other shared-bus designs as well.

The 8289 eliminates race conditions, resolves bus contention and matches processors operating asynchronously with respect to each other. Each processor on the bus is assigned a different priority. When simultaneous requests for the bus arrive, the 8289 resolves the contention and grants the bus to the processor with the highest priority; three different prioritizing techniques may be used. Chapter 4 discusses the 8289 in more detail.

2.6 Processor Control and Monitoring

Interrupts

The 8086 and 8088 have a simple and versatile interrupt system. Every interrupt is assigned a type code that identifies it to the CPU. The 8086 and 8088 can handle up to 256 different interrupt types. Interrupts may be initiated by devices external to the CPU; in addition, they also may be triggered by software interrupt instructions and, under certain conditions, by the CPU itself (see figure 2-28). Figure 2-29 illustrates the basic response of the 8086 and 8088 to an interrupt. The next sections elaborate on the information presented in this drawing.

External Interrupts

The 8086 and 8088 have two lines that external devices may use to signal interrupts (INTR and NMI). The INTR (Interrupt Request) line is usually driven by an Intel® 8259A Programmable Interrupt Controller (PIC), which is in turn connected to the devices that need interrupt services. The 8259A is a very flexible circuit that is controlled by software commands from the 8086 or 8088 (the PIC appears as a set of I/O ports to the software). Its main job is to accept interrupt requests from the devices attached to it, determine which requesting device has the highest priority, and then activate the 8086/8088 INTR line if the selected device has higher priority than the device currently being serviced (if there is one).

When INTR is active, the CPU takes different action depending on the state of the interrupt-enable flag (IF). No action takes place, however, until the currently-executing instruction has been

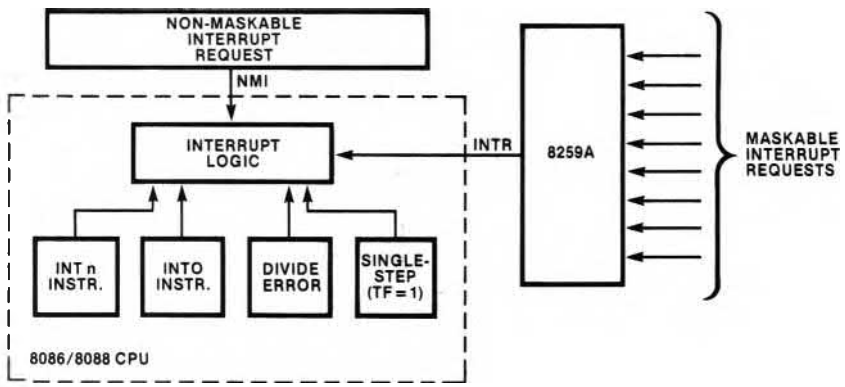


Figure 2-28. Interrupt Sources

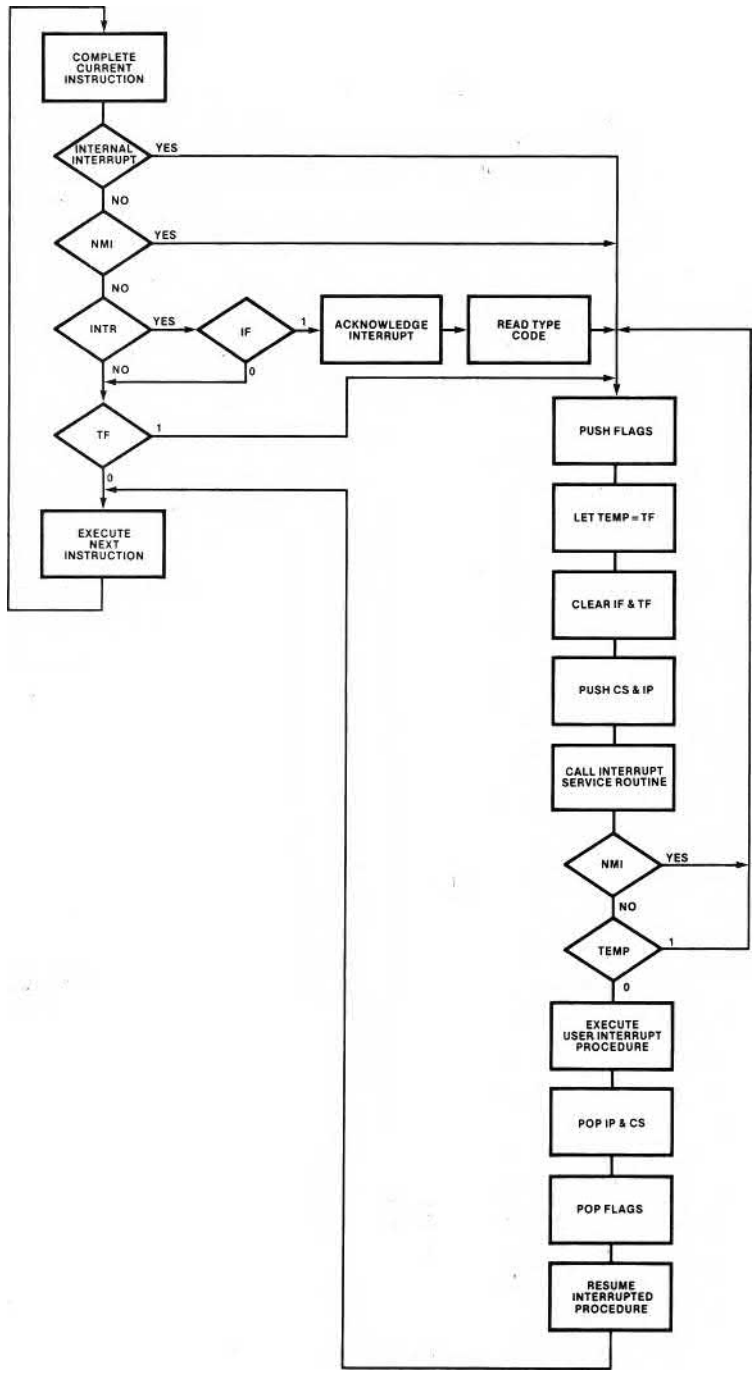


Figure 2-29. Interrupt Processing Sequence

completed.* Then, if IF is clear (meaning that interrupts signaled on INTR are masked or disabled), the CPU ignores the interrupt request and processes the next instruction. The INTR signal is not latched by the CPU, so it must be held active until a response is received or the request is withdrawn. If interrupts on INTR are enabled (if IF is set), then the CPU recognizes the interrupt request and processes it. Interrupt requests arriving on INTR can be enabled by executing an STI (set interrupt-enable flag) instruction, and disabled by executing a CLI (clear interrupt-enable flag) instruction. They also may be selectively masked (some types enabled, some disabled) by writing commands to the 8259A. It should be noted that in order to reduce the likelihood of excessive stack buildup, the STI and IRET instructions will reenable interrupts only after the end of the following instruction.

The CPU acknowledges the interrupt request by executing two consecutive interrupt acknowledge (INTA) bus cycles. If a bus hold request arrives (via the HOLD or request/grant lines) during the INTA cycles, it is not honored until the cycles have been completed. In addition, if the CPU is configured in maximum mode, it activates the $\overline{\text{LOCK}}$ signal during these cycles to indicate to other processors that they should not attempt to obtain the bus. The first cycle signals the 8259A that the request has been honored. During the second INTA cycle, the 8259A responds by placing a byte on the data bus that contains the interrupt type (0-255) associated with the device requesting service. (The type assignment is made when the 8259A is initialized by software in the 8086 or 8088.) The CPU reads this type code and uses it to call the corresponding interrupt procedure.

An external interrupt request also may arrive on another CPU line, NMI (non-maskable interrupt). This line is edge-triggered (INTR is level-triggered) and is generally used to signal the CPU of a "catastrophic" event, such as the imminent loss of power, memory error detection or bus parity error. Interrupt requests arriving on NMI cannot be disabled, are latched by the CPU, and have higher priority than an interrupt request on INTR. If an interrupt request arrives on both lines during the execution of an instruction, NMI will be recognized first. Non-maskable interrupts are predefined as type 2; the processor does not need to be supplied with a type code to call the NMI procedure, and it does not run the INTA bus cycles in response to a request on NMI.

The time required for the CPU to recognize an external interrupt request (interrupt latency) depends on how many clock periods remain in the execution of the current instruction. On the average, the longest latency occurs when a multiplication, division or variable-bit shift or rotate instruction is executing when the interrupt request arrives (see section 2.7 for detailed instruction timing data). As mentioned previously, in a few cases, worst-case latency will span two instructions rather than one.

Internal Interrupts

An INT (interrupt) instruction generates an interrupt immediately upon completion of its execution. The interrupt type coded into the instruction supplies the CPU with the type code needed to call the procedure to process the interrupt. Since any type code may be specified, software interrupts may be used to test interrupt procedures written to service external devices.

*There are a few cases in which an interrupt request is not recognized until after the *following* instruction. Repeat, LOCK and segment override prefixes are considered "part of" the instructions they prefix; no interrupt is recognized between execution of a prefix and an instruction. A MOV (move) to segment register instruction and a POP segment register instruction are treated similarly: no interrupt is recognized until after the following instruction. This mechanism protects a program that is changing to a new stack (by updating SS and SP). If an interrupt were recognized after SS had been changed, but before SP had been altered, the processor would push the flags, CS and IP into the wrong area of memory. It follows from this that whenever a segment register and another value must be updated together, the segment register should be changed first, followed immediately by the instruction that changes the other value. There are also two cases, WAIT and repeated string instructions, where an interrupt request is recognized in the middle of an instruction. In these cases, interrupts are accepted after any completed primitive operation or wait test cycle.

Mnemonics © Intel, 1978

If the overflow flag (OF) is set, an INTO (interrupt on overflow) instruction generates a type 4 interrupt immediately upon completion of its execution.

The CPU itself generates a type 0 interrupt immediately following execution of a DIV or IDIV (divide, integer divide) instruction if the calculated quotient is larger than the specified destination.

If the trap flag (TF) is set, the CPU automatically generates a type 1 interrupt following *every* instruction. This is called single-step execution and is a powerful debugging tool that is discussed in more detail shortly.

All internal interrupts (INT, INTO, divide error, and single-step) share these characteristics:

1. The interrupt type code is either contained in the instruction or is predefined.
2. No INTA bus cycles are run.
3. Internal interrupts cannot be disabled, except for single-step.
4. Any internal interrupt (except single-step) has higher priority than any external interrupt (see table 2-3). If interrupt requests arrive on NMI and/or INTR during execution of an instruction that causes an internal interrupt (e.g., divide error), the internal interrupt is processed first.

Interrupt Pointer Table

The interrupt pointer (or interrupt vector) table (figure 2-30) is the link between an interrupt type code and the procedure that has been designated to service interrupts associated with that code. The interrupt pointer table occupies up to the first 1k bytes of low memory. There may be up to 256 entries in the table, one for each interrupt type

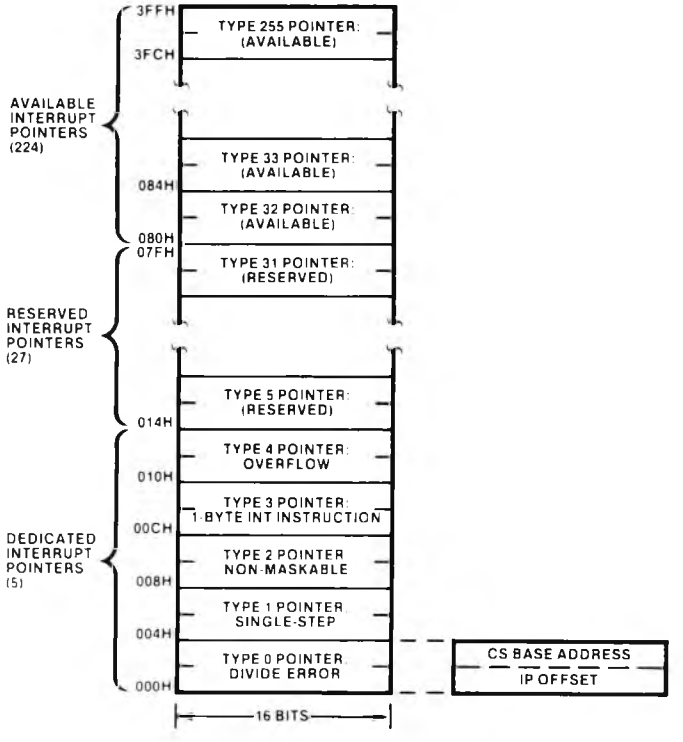


Figure 2-30. Interrupt Pointer Table

that can occur in the system. Each entry in the table is a doubleword pointer containing the address of the procedure that is to service interrupts of that type. The higher-addressed word of the pointer contains the base address of the segment containing the procedure. The lower-addressed word contains the procedure's offset from the beginning of the segment. Since each entry is four bytes long, the CPU can calculate the location of the correct entry for a given interrupt type by simply multiplying (type*4).

Table 2-3. Interrupt Priorities

INTERRUPT	PRIORITY
Divide error, INT n, INTO	highest
NMI	
INTR	
Single-step	lowest

Space at the high end of the table that would be occupied by entries for interrupt types that cannot occur in a given application may be used for other purposes. The dedicated and reserved portions of the interrupt pointer table (locations 0H through 7FH), however, should not be used for any other purpose to insure proper system operation and to preserve compatibility with future Intel hardware and software products.

After pushing the flags onto the stack, the 8086 or 8088 activates an interrupt procedure by executing the equivalent of an intersegment indirect CALL instruction. The target of the "CALL" is the address contained in the interrupt pointer table element located at (type*4). The CPU saves the address of the next instruction by pushing CS and IP onto the stack. These are then replaced by the second and first words of the table element, thus transferring control to the procedure.

If multiple interrupt requests arrive simultaneously, the processor activates the interrupt procedures in priority order. Figure 2-31 shows how procedures would be activated in an extreme case. The processor is running in single-step mode with external interrupts enabled. During execution of a divide instruction, INTR is activated. Furthermore the instruction generates a divide error interrupt. Figure 2-31 shows that the interrupts are recognized in turn, in the order of their priorities except for INTR. INTR is not recognized until after the following instruction because recognition of the earlier interrupts cleared IF. Of couse interrupts could be reenabled in any of the interrupt response routines if earlier response to INTR is desired.

As figure 2-31 shows, all main-line code is executed in single-step mode. Also, because of the order of interrupt processing, the opportunity exists in each occurrence of the single-step routine to select whether pending interrupt routines (divide error and INTR routines in this example) are executed at full speed or in single-step mode.

Interrupt Procedures

When an interrupt service procedure is entered, the flags, CS, and IP are pushed onto the stack and TF and IF are cleared. The procedure may reenable external interrupts with the STI (set interrupt-enable flag) instruction, thus allowing itself to be interrupted by a request on INTR. (Note, however, that interrupts are not actually enabled until the instruction *following* STI has executed.) An interrupt procedure always may be interrupted by a request arriving on NMI. Software- or processor-initiated interrupts occurring within the procedure also will interrupt the procedure. Care must be taken in interrupt procedures that the type of interrupt being serviced by the procedure does not itself inadvertently occur within the procedure. For example, an attempt to divide by 0 in the divide error (type 0) interrupt procedure may result in the procedure being reentered endlessly. Enough stack space must be available to accommodate the maximum depth of interrupt nesting that can occur in the system.

Like all procedures, interrupt procedures should save any registers they use before updating them, and restore them before terminating. It is good practice for an interrupt procedure to enable external interrupts for all but "critical sections" of code (those sections that cannot be interrupted without risking erroneous results). If external interrupts are disabled for too long in a procedure, interrupt requests on INTR can potentially be lost.

Mnemonics © Intel, 1978

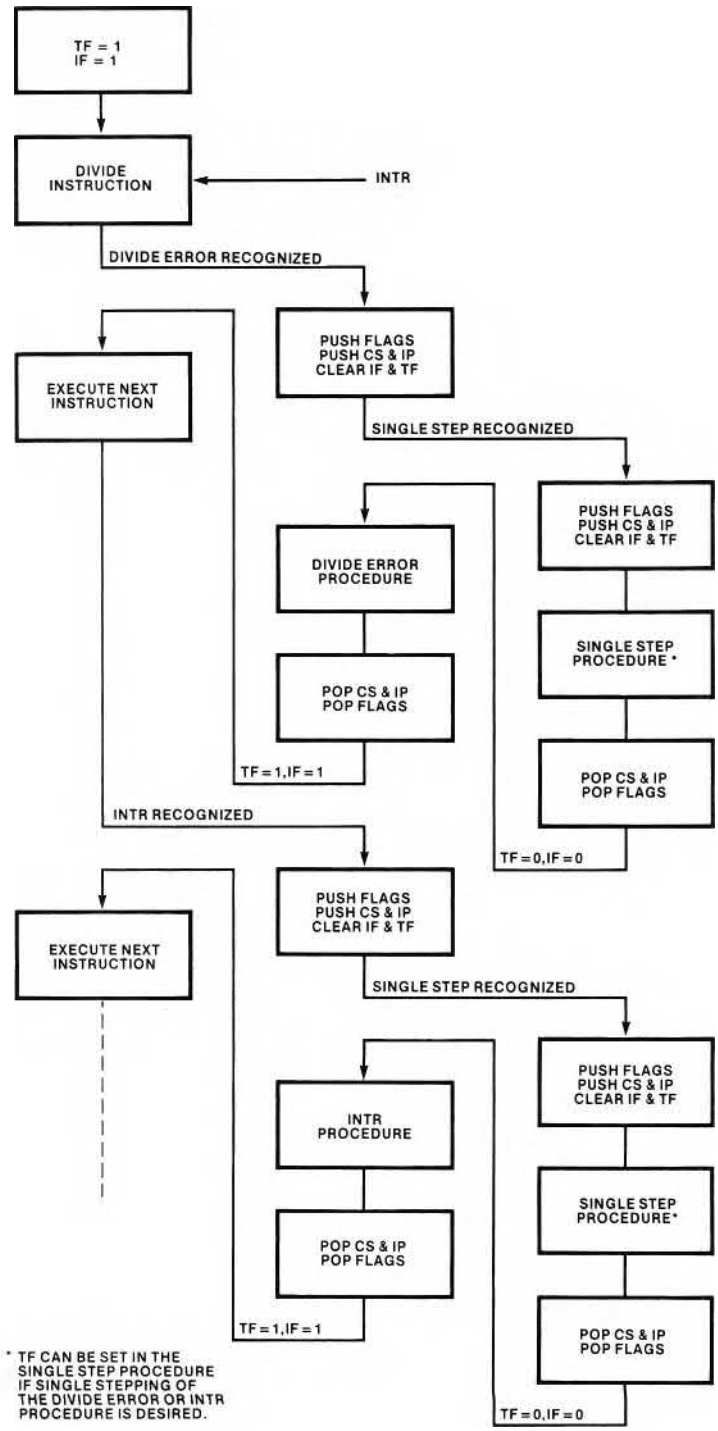


Figure 2-31. Processing Simultaneous Interrupts

Mnemonics © Intel, 1978

All interrupt procedures should be terminated with an IRET (interrupt return) instruction. The IRET instruction assumes that the stack is in the same condition as it was when the procedure was entered. It pops the top three stack words into IP, CS and the flags, thus returning to the instruction that was about to be executed when the interrupt procedure was activated.

The actual processing done by the procedure is dependent upon the application. If the procedure is servicing an external device, it should output a command to the device instructing it to remove its interrupt request. It might then read status information from the device, determine the cause of the interrupt and then take action accordingly. Section 2.10 contains three typical interrupt procedure examples.

Software-initiated interrupt procedures may be used as service routines ("supervisor calls") for other programs in the system. In this case, the interrupt procedure is activated when a program, rather than an external device, needs attention. (The "attention" might be to search a file for a record, send a message to another program, request an allocation of free memory, etc.) Software interrupt procedures can be advantageous in systems that dynamically relocate programs during execution. Since the interrupt pointer table is at a fixed storage location, procedures may "call" each other through the table by issuing software interrupt instructions. This provides a stable communication "exchange" that is independent of procedure addresses. The interrupt procedures may themselves be moved so long as the interrupt pointer table always is updated to provide the linkage from the "calling" program via the interrupt type code.

Single-Step (Trap) Interrupt

When TF (the trap flag) is set, the 8086 or 8088 is said to be in single-step mode. In this mode, the processor automatically generates a type 1 interrupt after each instruction. Recall that as part of its interrupt processing, the CPU automatically pushes the flags onto the stack and then clears TF and IF. Thus the processor is *not* in single-step mode when the single-step interrupt procedure is entered; it runs normally. When the single-step procedure terminates, the old flag image is restored from the stack, placing the CPU back into single-step mode.

Single-stepping is a valuable debugging tool. It allows the single-step procedure to act as a "window" into the system through which operation can be observed instruction-by-instruction. A single-step interrupt procedure, for example, can print or display register contents, the value of the instruction pointer (it is on the stack), key memory variables, etc., as they change after each instruction. In this way the exact flow of a program can be traced in detail, and the point at which discrepancies occur can be determined. Other possible services that could be provided by a single-step routine include:

- Writing a message when a specified memory location or I/O port changes value (or equals a specified value).
- Providing diagnostics selectively (only for certain instruction addresses for instance).
- Letting a routine execute a number of times before providing diagnostics.

The 8086 and 8088 do not have instructions for setting or clearing TF directly. Rather, TF can be changed by modifying the flag-image on the stack. The PUSHF and POPF instructions are available for pushing and popping the flags directly (TF can be set by ORing the flag-image with 0100H and cleared by ANDing it with FEFFH). After TF is set in this manner, the first single-step interrupt occurs after the first instruction following the IRET from the single-step procedure.

If the processor is single-stepping, it processes an interrupt (either internal or external) as follows. Control is passed normally (flags, CS and IP are pushed) to the procedure designated to handle the type of interrupt that has occurred. However, before the first instruction of that procedure is executed, the single-step interrupt is "recognized" and control is passed normally (flags, CS and IP are pushed) to the type 1 interrupt procedure. When single-step procedure terminates, control returns to the previous interrupt procedure. Figure 2-31 illustrates this process in a case where two interrupts occur when the processor is in single-step mode.

Breakpoint Interrupt

A type 3 interrupt is dedicated to the breakpoint interrupt. A breakpoint is generally any place in a program where normal execution is arrested so

that some sort of special processing may be performed. Breakpoints typically are inserted into programs during debugging as a way of displaying registers, memory locations, etc., at crucial points in the program.

The INT 3 (breakpoint) instruction is one byte long. This makes it easy to "plant" a breakpoint anywhere in a program. Section 2.10 contains an example that shows how a breakpoint may be set and how a breakpoint procedure may be used to place the processor into single-step mode.

The breakpoint instruction also may be used to "patch" a program (insert new instructions) without recompiling or reassembling it. This may be done by saving an instruction byte, and replacing it with an INT 3 (CCH) machine instruction. The breakpoint procedure would contain the new machine instructions, plus code to restore the saved instruction byte and decrement IP on the stack before returning, so that the displaced instruction would be executed after the patch instructions. The breakpoint example in section 2.10 illustrates these principles.

Note that patching a program requires machine-instruction programming and should be undertaken with considerable caution; it is easy to add new bugs to a program in an attempt to correct existing ones. Note also that a patch is only a temporary measure to be used in exceptional conditions. The affected code should be updated and retranslated as soon as possible.

System Reset

The 8086/8088 RESET line provides an orderly way to start or restart an executing system. When the processor detects the positive-going edge of a pulse on RESET, it terminates all activities until the signal goes low, at which time it initializes the system as shown in table 2-4.

Since the code segment register contains FFFFH and the instruction pointer contains 0H, the processor executes its first instruction following system reset from absolute memory location FFFF0H. This location normally contains an intersegment direct JMP instruction whose target is the actual beginning of the system program. The LOC-86 utility supplies this JMP instruction from information in the program that identifies its first instruction. As external (maskable) interrupts are disabled by system reset, the system software should reenable interrupts as soon as the system is initialized to the point where they can be processed.

Table 2-4. CPU State Following RESET

CPU COMPONENT	CONTENT
Flags	Clear
Instruction Pointer	0000H
CS Register	FFFFH
DS Register	0000H
SS Register	0000H
ES Register	0000H
Queue	Empty

Instruction Queue Status

When configured in maximum mode, the 8086 and 8088 provide information about instruction queue operations on lines QS0 and QS1. Table 2-5 interprets the four states that these lines can represent.

The queue status lines are provided for external processors that receive instructions and/or operands via the 8086/8088 ESC (escape) instruction (see sections 2.5 and 2.8). Such a processor may monitor the bus to see when an ESC instruction is fetched and then track the instruction through the queue to determine when (and if) the instruction is executed.

Table 2-5. Queue Status Signals (Maximum Mode Only)

QS_0	QS_1	QUEUE OPERATION IN LAST CLK CYCLE
0	0	No operation; default value
0	1	First byte of an instruction was taken from the queue
1	0	Queue was reinitialized
1	1	Subsequent byte of an instruction was taken from the queue

Processor Halt

When the HLT (halt) instruction (see section 2.7) is executed, the 8086 or 8088 enters the halt state. This condition may be interpreted as "stop all

Mnemonics © Intel, 1978

operations until an external interrupt occurs or the system is reset." No signals are floated during the halt state, and the content of the address and data buses is undefined. A bus hold request arriving on the HOLD line (minimum mode) or either request/grant line (maximum mode) is acknowledged normally while the processor is halted.

The halt state can be used when an event prevents the system from functioning correctly. An example might be a power-fail interrupt. After recognizing that loss of power is imminent, the CPU could use the remaining time to move registers, flags and vital variables to (for example) a battery-powered CMOS RAM area and then halt until the return of power was signaled by an interrupt or system reset.

Status Lines

When configured in maximum mode, the 8086 and 8088 emit eight status signals that can be used by external devices. Lines $\overline{S0}$, $\overline{S1}$ and $\overline{S2}$ identify the type of bus cycle that the CPU is starting to execute (table 2-6). These lines are typically decoded by the 8288 Bus Controller. S3 and S4 indicate which segment register was used to construct the physical address being used in this bus cycle (see table 2-7). Line S5 reflects the state of the interrupt-enable flag. S6 is always 0. S7 is a spare line whose content is undefined.

Table 2-6. Bus Cycle Status Signals

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	TYPES OF BUS CYCLE
0	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	HALT
1	0	0	Instruction Fetch
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive; no bus cycle

Table 2-7. Segment Register Status Lines

S_4	S_3	SEGMENT REGISTER
0	0	ES
0	1	SS
1	0	CS or none (I/O or Interrupt Vector)
1	1	DS

2.7 Instruction Set

The 8086 and 8088 execute exactly the same instructions. This instruction set includes equivalents to the instructions typically found in previous microprocessors, such as the 8080/8085. Significant new operations include:

- multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers,
- move, scan and compare operations for strings up to 64k bytes in length,
- non-destructive bit testing,
- byte translation from one code to another,
- software-generated interrupts, and
- a group of instructions that can help coordinate the activities of multiprocessor systems.

These instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory and immediate operands may be specified interchangeably in most instructions (except, of course, that immediate values may only serve as "source" and not "destination" operands). In particular, memory variables can be added to, subtracted from, shifted, compared, and so on, in place, without moving them in and out of registers. This saves instructions, registers, and execution time in assembly language programs. In high-level languages, where most variables are memory based, compilers, such as PL/M-86, can produce faster and shorter object programs.

The 8086/8088 instruction set can be viewed as existing at two levels: the assembly level and the machine level. To the assembly language programmer, the 8086 and 8088 appear to have a repertoire of about 100 instructions. One MOV (move) instruction, for example, transfers a byte or a word from a register or a memory location or an immediate value to either a register or a memory location. The 8086 and 8088 CPUs, however, recognize 28 different MOV machine instructions ("move byte register to memory," "move word immediate to register," etc.). The ASM-86 assembler translates the assembly-level instructions written by a programmer into the

machine-level instructions that are actually executed by the 8086 or 8088. Compilers such as PL/M-86 translate high-level language statements directly into machine-level instructions.

The two levels of the instruction set address two different requirements: efficiency and simplicity. The numerous—there are about 300 in all—forms of machine-level instructions allow these instructions to make very efficient use of storage. For example, the machine instruction that increments a memory operand is three or four bytes long because the address of the operand must be encoded in the instruction. To increment a register, however, does not require as much information, so the instruction can be shorter. In fact, the 8086 and 8088 have eight different machine-level instructions that increment a different 16-bit register; these instructions are only one byte long.

If a programmer had to write one instruction to increment a register, another to increment a memory variable, etc., the benefit of compact instructions would be offset by the difficulty of programming. The assembly-level instructions simplify the programmer's view of the instruction set. The programmer writes one form of the INC (increment) instruction and the ASM-86 assembler examines the operand to determine which machine-level instruction to generate.

This section presents the 8086/8088 instruction set from two perspectives. First, the assembly-level instructions are described in functional terms. The assembly-level instructions are then presented in a reference table that breaks out all permissible operand combinations with execution times and machine instruction length, plus the effect that the instruction has on the CPU flags. Machine-level instruction encoding and decoding are covered in section 4.2.

Data Transfer Instructions

The 14 data transfer instructions (table 2-8) move single bytes and words between memory and registers as well as between register AL or AX and I/O ports. The stack manipulation instructions are included in this group as are instructions for transferring flag contents and for loading segment registers.

Table 2-8. Data Transfer Instructions

GENERAL PURPOSE	
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte
INPUT/OUTPUT	
IN	Input byte or word
OUT	Output byte or word
ADDRESS OBJECT	
LEA	Load effective address
LDS	Load pointer using DS
LES	Load pointer using ES
FLAG TRANSFER	
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack

General Purpose Data Transfers

MOV *destination,source*

MOV transfers a byte or a word from the source operand to the destination operand.

PUSH *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP *destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

Mnemonics © Intel, 1978

XCHG *destination,source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors (see section 2.5).

XLAT *translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

IN *accumulator,port*

IN transfers a byte or a word from an input port to the AL register or the AX register, respectively. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

OUT *port,accumulator*

OUT transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in register DX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

Address Object Transfers

These instructions manipulate the *addresses* of variables rather than the contents or values of variables. They are most useful for list processing, based variables, and string operations.

LEA *destination,source*

LEA (load effective address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general register. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to load these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

LDS *destination,source*

LDS (load pointer using DS) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

LES *destination,source*

LES (load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

Flag Transfers

LAHF

LAHF (load register AH from flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH

(see figure 2-32). The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

SAHF

SAHF (store register AH into flags) transfers bits 7, 6, 4, 2 and 0 from register AH into SF, ZF, AF, PF and CF, respectively, replacing whatever values these flags previously had. OF, DF, IF and TF are not affected. This instruction is provided for 8080/8085 compatibility.

PUSHF

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to by SP (see figure 2-32). The flags themselves are not affected.

POPF

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (see figure 2-32). SP is then incremented by two to point to the new top of stack. PUSHF and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory-image and then popping the flags.

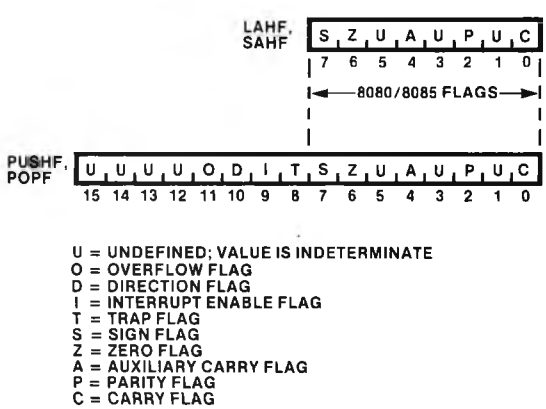


Figure 2-32. Flag Storage Formats

Arithmetic Instructions

Arithmetic Data Formats

8086 and 8088 arithmetic operations (table 2-9) may be performed on four types of numbers: unsigned binary, signed binary (integers), unsigned packed decimal and unsigned unpacked decimal (see table 2-10). Binary numbers may be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor always assumes that the operands specified in arithmetic instructions contain data that represent valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

Table 2-9. Arithmetic Instructions

ADDITION	
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
SUBTRACTION	
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
MULTIPLICATION	
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
DIVISION	
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword

Mnemonics © Intel, 1978

Table 2-10. Arithmetic Interpretation of 8-Bit Numbers

HEX	BIT PATTERN	UNSIGNED BINARY	SIGNED BINARY	UNPACKED DECIMAL	PACKED DECIMAL
07	0 0 0 0 0 1 1 1	7	+7	7	7
89	1 0 0 0 1 0 0 1	137	−119	invalid	89
C5	1 1 0 0 0 1 0 1	197	−59	invalid	invalid

Unsigned binary numbers may be either 8 or 16 bits long; all bits are considered in determining a number's magnitude. The value range of an 8-bit unsigned binary number is 0-255; 16 bits can represent values from 0 through 65,535. Addition, subtraction, multiplication and division operations are available for unsigned binary numbers.

Signed binary numbers (integers) may be either 8 or 16 bits long. The high-order (leftmost) bit is interpreted as the number's sign: 0 = positive and 1 = negative. Negative numbers are represented in standard two's complement notation. Since the high-order bit is used for a sign, the range of an 8-bit integer is −128 through +127; 16-bit integers may range from −32,768 through +32,767. The value zero has a positive sign. Multiplication and division operations are provided for signed binary numbers. Addition and subtraction are performed with the unsigned binary instructions. Conditional jump instructions, as well as an "interrupt on overflow" instruction, can be used following an unsigned operation on an integer to detect overflow into the sign bit.

Packed decimal numbers are stored as unsigned byte quantities. The byte is treated as having one decimal digit in each half-byte (nibble); the digit in the high-order half-byte is the most significant. Hexadecimal values 0-9 are valid in each half-byte, and the range of a packed decimal number is 0-99. Addition and subtraction are performed in two steps. First an unsigned binary instruction is used to produce an intermediate result in register AL. Then an adjustment operation is performed which changes the intermediate value in AL to a final correct packed decimal result. Multiplication and division adjustments are not available for packed decimal numbers.

Unpacked decimal numbers are stored as unsigned byte quantities. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers. The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction. Arithmetic on unpacked decimal numbers is performed in two steps. The unsigned binary addition, subtraction and multiplication operations are used to produce an intermediate result in register AL. An adjustment instruction then changes the value in AL to a final correct unpacked decimal number. Division is performed similarly, except that the adjustment is carried out on the numerator operand in register AL first, then a following unsigned binary division instruction produces a correct result.

Unpacked decimal numbers are similar to the ASCII character representations of the digits 0-9. Note, however, that the high-order half-byte of an ASCII numeral is always 3H. Unpacked decimal arithmetic may be performed on ASCII numeric characters under the following conditions:

- the high-order half-byte of an ASCII numeral must be set to 0H prior to multiplication or division.
- unpacked decimal arithmetic leaves the high-order half-byte set to 0H; it must be set to 3H to produce a valid ASCII numeral.

Arithmetic Instructions and Flags

The 8086/8088 arithmetic instructions post certain characteristics of the result of the operation to six flags. Most of these flags can be tested by following the arithmetic instruction with a conditional jump instruction; the INTO (interrupt on overflow) instruction also may be used. The

various instructions affect the flags differently, as explained in the instruction descriptions. However, they follow these general rules:

- CF (carry flag): If an addition results in a carry out of the high-order bit of the result, then CF is set; otherwise CF is cleared. If a subtraction results in a borrow into the high-order bit of the result, then CF is set; otherwise CF is cleared. Note that a *signed* carry is indicated by CF ≠ OF. CF can be used to detect an unsigned overflow. Two instructions, ADC (add with carry) and SBB (subtract with borrow), incorporate the carry flag in their operations and can be used to perform multibyte (e.g., 32-bit, 64-bit) addition and subtraction.

- AF (auxiliary carry flag): If an addition results in a carry out of the low-order half-byte of the result, then AF is set; otherwise AF is cleared. If a subtraction results in a borrow into the low-order half-byte of the result, then AF is set; otherwise AF is cleared. The auxiliary carry flag is provided for the decimal adjust instructions and ordinarily is not used for any other purpose.

- SF (sign flag): Arithmetic and logical instructions set the sign flag equal to the high-order bit (bit 7 or 15) of the result. For signed binary numbers, the sign flag will be 0 for positive results and 1 for negative results (so long as overflow does not occur). A conditional jump instruction can be used following addition or subtraction to alter the flow of the program depending on the sign of the result. Programs performing unsigned operations typically ignore SF since the high-order bit of the result is interpreted as a digit rather than a sign.

- ZF (zero flag): If the result of an arithmetic or logical operation is zero, then ZF is set; otherwise ZF is cleared. A conditional jump instruction can be used to alter the flow of the program if the result is or is not zero.

- PF (parity flag): If the low-order eight bits of an arithmetic or logical result contain an even number of 1-bits, then the parity flag is set; otherwise it is cleared. PF is provided for 8080/8085 compatibility; it also can be used to check ASCII characters for correct parity.

- OF (overflow flag): If the result of an operation is too large a positive number, or too small a negative number to fit in the destination operand (excluding the sign bit), then OF is set; otherwise OF is cleared. OF thus indicates signed arithmetic overflow; it can be tested with a conditional jump or the INTO (interrupt on overflow) instruction. OF may be ignored when performing unsigned arithmetic.

Addition

ADD *destination,source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF and ZF.

ADC *destination,source*

ADC (Add with Carry) sums the operands, which may be bytes or words, adds one if CF is set and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADC updates AF, CF, OF, PF, SF and ZF. Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.

INC *destination*

INC (Increment) adds one to the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF and ZF; it does not affect CF.

AAA

AAA (ASCII Adjust for Addition) changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAA updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAA.

Mnemonics © Intel, 1978

DAA

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAA.

Subtraction

SUB *destination,source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF and ZF.

SBB *destination,source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

DEC *destination*

DEC (Decrement) subtracts one from the destination, which may be a byte or a word. DEC updates AF, OF, PF, SF, and ZF; it does not affect CF.

NEG *destination*

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an integer. If the operand is zero, its sign is not changed. Attempting to negate a byte containing −128 or a word containing −32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF and ZF. CF is always set except when the operand is zero, in which case it is cleared.

CMP *destination,source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF, SF and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (jump if greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

AAS

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF and ZF is undefined following execution of AAS.

DAS

DAS (Decimal Adjust for Subtraction) corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). DAS changes the content of AL to a pair of valid packed decimal digits. DAS updates AF, CF, PF, SF and ZF; the content of OF is undefined following execution of DAS.

Multiplication

MUL *source*

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length

result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is nonzero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of MUL.

IMUL *source*

IMUL (Integer Multiply) performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of the result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF and ZF is undefined following execution of IMUL.

AAM

AAM (ASCII Adjust for Multiply) corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result. AAM updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAM.

Division

DIV *source*

DIV (divide) performs an unsigned division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH. The single-length quotient is returned in AL, and the single-length remainder is returned in AH. If the source operand is a word, it is divided into the double-length dividend in registers AX and DX. The single-length quotient is returned in AX, and the single-length remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. The content of AF, CF, OF, PF, SF and ZF is undefined following execution of DIV.

IDIV *source*

IDIV (Integer Divide) performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127 (7FH) and the minimum negative quotient is −127 (81H). If the source operand is a word, it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is −32,767 (8001H). If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend. The content of AF, CF, OF, PF, SF and ZF is undefined following IDIV.

AAD

AAD (ASCII Adjust for Division) modifies the numerator in AL *before* dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subse-

quent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. AAD updates PF, SF and ZF; the content of AF, CF and OF is undefined following execution of AAD.

CBW

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout register AH. CBW does not affect any flags. CBW can be used to produce a double-length (word) dividend from a byte prior to performing byte division.

CWD

CWD (Convert Word to Doubleword) extends the sign of the word in register AX throughout register DX. CWD does not affect any flags. CWD can be used to produce a double-length (doubleword) dividend from a word prior to performing word division.

Bit Manipulation Instructions

The 8086 and 8088 provide three groups of instructions (table 2-11) for manipulating bits within both bytes and words: logical, shifts and rotates.

Table 2-11. Bit Manipulation Instructions

LOGICALS	
NOT	"Not" byte or word
AND	"And" byte or word
OR	"Inclusive or" byte or word
XOR	"Exclusive or" byte or word
TEST	"Test" byte or word
SHIFTS	
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
ROTATES	
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

Logical

The logical instructions include the boolean operators "not," "and," "inclusive or," and "exclusive or," plus a TEST instruction that sets the flags, but does not alter either of its operands.

AND, OR, XOR and TEST affect the flags as follows: The overflow (OF) and carry (CF) flags are always cleared by logical instructions, and the content of the auxiliary carry (AF) flag is always undefined following execution of a logical instruction. The sign (SF), zero (ZF) and parity (PF) flags are always posted to reflect the result of the operation and can be tested by conditional jump instructions. The interpretation of these flags is the same as for arithmetic instructions. SF is set if the result is negative (high-order bit is 1), and is cleared if the result is positive (high-order bit is 0). ZF is set if the result is zero, cleared otherwise. PF is set if the result contains an even number of 1-bits (has even parity) and is cleared if the number of 1-bits is odd (the result has odd parity). Note that NOT has no effect on the flags.

NOT *destination*

NOT inverts the bits (forms the one's complement) of the byte or word operand.

AND *destination,source*

AND performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared.

OR *destination,source*

OR performs the logical "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.

XOR *destination,source*

XOR (Exclusive Or) performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the

result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

TEST *destination,source*

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result, i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (jump if not zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.

Shifts

The bits in bytes and words may be shifted arithmetically or logically. Up to 255 shifts may be performed, according to the value of the count operand coded in the instruction. The count may be specified as the constant 1, or as register CL, allowing the shift count to be a variable supplied at execution time. Arithmetic shifts may be used to multiply and divide binary numbers by powers of two (see note in description of SAR). Logical shifts can be used to isolate bits in bytes or words.

Shift instructions affect the flags as follows. AF is always undefined following a shift operation. PF, SF and ZF are updated normally, as in the logical instructions. CF always contains the value of the last bit shifted out of the destination operand. The content of OF is always undefined following a multibit shift. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation; if the sign bit retains its original value, OF is cleared.

SHL/SAL *destination,count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then OF is cleared.

SHR *destination,source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

SAR *destination,count*

SAR (Shift Arithmetic Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an "equivalent" IDIV instruction if the destination operand is negative and 1-bits are shifted out. For example, shifting −5 right by one bit yields −3, while integer division of −5 by 2 yields −2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.

Rotates

Bits in bytes and words also may be rotated. Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand. As in the shift instructions, the number of bits to be rotated is taken from the count operand, which may specify either a constant of 1, or the CL register. The carry flag may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated in CF and then tested by a JC (jump if carry) or JNC (jump if not carry) instruction.

Rotates affect only the carry and overflow flags. CF always contains the value of the last bit rotated out. On multibit rotates, the value of OF is always undefined. In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared.

ROL *destination,count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

Mnemonics © Intel, 1978

ROR *destination,count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

RCL *destination,count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and itself is replaced by the high-order bit of the destination.

RCR *destination,count*

RCR (Rotate through Carry Right) operates exactly like RCL except that the bits are rotated right instead of left.

String Instructions

Five basic string operations, called primitives, allow strings of bytes or words to be operated on, one element (byte or word) at a time. Strings of up to 64k bytes may be manipulated with these instructions. Instructions are available to move, compare and scan for a value, as well as for moving string elements to and from the accumulator (see table 2-12). These basic operations may be preceded by a special one-byte prefix that causes the instruction to be repeated by the hardware, allowing long strings to be processed much faster than would be possible with a software loop. The repetitions can be terminated by a variety of conditions, and a repeated operation may be interrupted and resumed.

The string instructions operate quite similarly in many respects; the common characteristics are covered here and in table 2-13 and figure 2-33 rather than in the descriptions of the individual instructions. A string instruction may have a source operand, a destination operand, or both. The hardware assumes that a source string resides in the current data segment; a segment prefix byte may be used to override this assumption. A destination string must be in the current extra segment. The assembler checks the attributes of the operands to determine if the elements of the strings are bytes or words. The assembler does not, however, use the operand names to address the strings. Rather, the content of register SI (source index) is used as an offset to address the current element of the source string, and the content of register DI (destination index) is taken as the offset of the current destination string element. These registers must be initialized to point to the source/destination strings before executing the string instruction; the LDS, LES and LEA instructions are useful in this regard.

Table 2-12. String Instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVS	Move byte or word string
MOVSB/MOVSW	Move byte or word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string

Table 2-13. String Instruction Register and Flag Use

SI	Index (offset) for source string
DI	Index (offset) for destination string
CX	Repetition counter
AL/AX	Scan value Destination for LODS Source for STOS
DF	0 = auto-increment SI, DI 1 = auto-decrement SI, DI
ZF	Scan/compare terminator

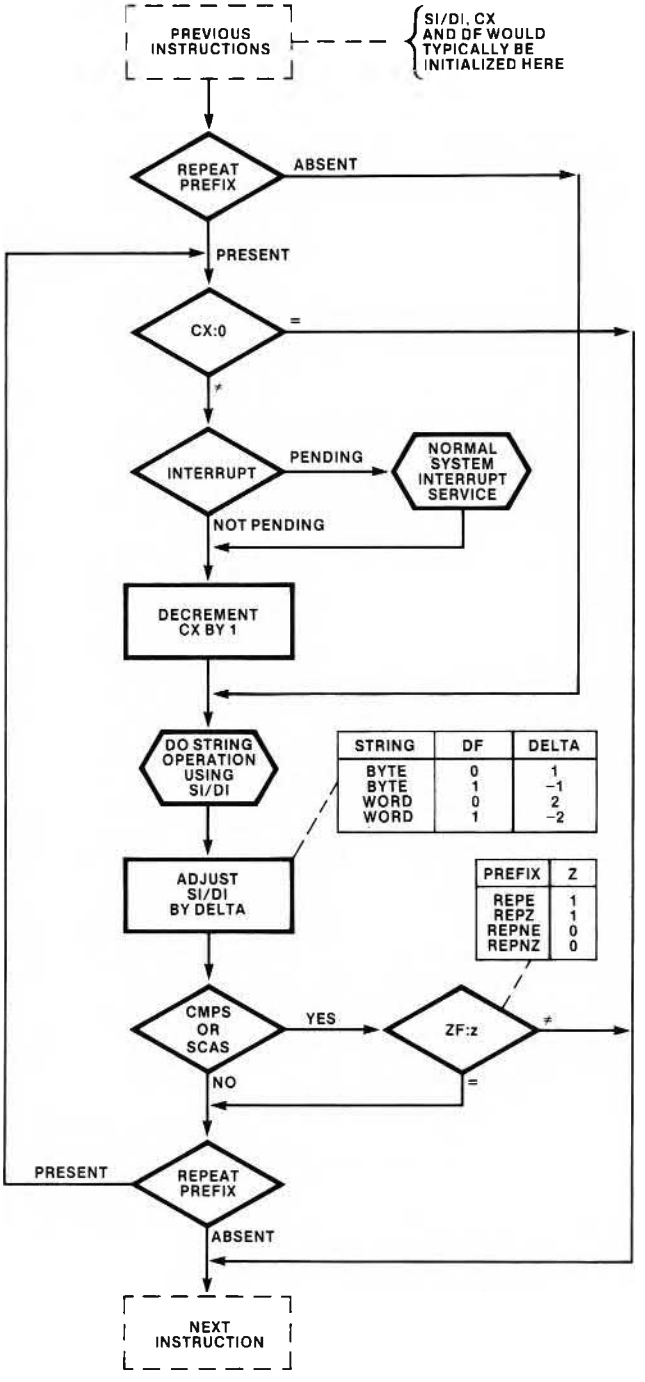


Figure 2-33. String Operation Flow

Mnemonics © Intel, 1978

The string instructions automatically update SI and/or DI in anticipation of processing the next string element. The setting of DF (the direction flag) determines whether the index registers are auto-incremented (DF = 0) or auto-decremented (DF = 1). If byte strings are being processed, SI and/or DI is adjusted by1; the adjustment is 2 for word strings.

If a Repeat prefix has been coded, then register CX (count register) is decremented by 1 after each repetition of the string instruction; therefore, CX must be initialized to the number of repetitions desired before the string instruction is executed. If CX is 0, the string instruction is not executed, and control goes to the following instruction.

Section 2.10 contains examples that illustrate the use of all the string instructions.

REP/REPE/REPZ/REPNE/REPNZ

Repeat, Repeat While Equal, Repeat While Zero, Repeat While Not Equal and Repeat While Not Zero are five mnemonics for two forms of the prefix byte that controls repetition of a subsequent string instruction. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are two mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ except that the zero flag must be cleared or the repetition is terminated. Note that ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. Note, however, that execution does *not* resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. The processor "remembers" only one prefix in effect at the time of the interrupt, the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes at this point, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a non-maskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

MOVS *destination-string,source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

MOVSB/MOVSW

These are alternate mnemonics for the move string instruction. These mnemonics are coded without operands; they explicitly tell the assembler that a byte string (MOVSB) or a word string (MOVSW) is to be moved (when MOVS is coded, the assembler determines the string type from the attributes of the operands). These mnemonics are useful when the assembler cannot determine the attributes of a string, e.g., a section of code is being moved.

CMPS *destination-string,source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects the flags but does not alter either operand, updates SI and DI to point to the next string element and updates AF, CF, OF, PF, SF and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the destination element is greater than the source element. If CMPS is prefixed with REPE

or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF = 1)." If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF = 0)." Thus, CMPS can be used to find matching or differing string elements.

SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element = scan-value (ZF = 1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.

LODS *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

Program Transfer Instructions

The sequence of execution of instructions in an 8086/8088 program is determined by the content of the code segment register (CS) and the instruction pointer (IP). The CS register contains the base address of the current code segment, the 64k portion of memory from which instructions are presently being fetched. The IP is used as an offset from the beginning of the code segment; the combination of CS and IP points to the memory location from which the next instruction is to be fetched. (Recall that under most operating conditions, the next instruction to be *executed* has already been fetched from memory and is waiting in the CPU instruction queue.) The program transfer instructions operate on the instruction pointer and on the CS register; changing the content of these causes normal sequential execution to be altered. When a program transfer occurs, the queue no longer contains the correct instruction, and the BIU obtains the next instruction from memory using the new IP and CS values, passes the instruction directly to the EU, and then begins refilling the queue from the new location.

Four groups of program transfers are available in the 8086/8088 (see table 2-14): unconditional transfers, conditional transfers, iteration control instructions and interrupt-related instructions. Only the interrupt-related instructions affect any CPU flags. As will be seen, however, the execution of many of the program transfer instructions is affected by the states of the flags.

Unconditional Transfers

The unconditional transfer instructions may transfer control to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). (The ASM-86 assembler terms an intrasegment target NEAR and an intersegment target FAR.) The transfer is made unconditionally any time the instruction is executed.

CALL *procedure-name*

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The

Table 2-14. Program Transfer Instructions

UNCONDITIONAL TRANSFERS	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
CONDITIONAL TRANSFERS	
JA/JNBE	Jump if above/not below nor equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
ITERATION CONTROLS	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	Jump if register CX = 0
INTERRUPTS	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Interrupt return

assembler generates a different type of CALL instruction depending on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be *executed* before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The relative displacement (up to ±32k) of the target procedure from the CALL instruction is then added to the instruction pointer. This form of the CALL instruction is "self-relative" and is appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are in the same segment and are moved together.

An intrasegment indirect CALL may be made through memory or through a register. SP is decremented by two and IP is pushed onto the stack. The offset of the target procedure is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and is replaced by the offset word contained in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the content of the first word of the doubleword pointer referenced by the instruction.

RET *optional-pop-value*

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates an intrasegment RET if the programmer has defined the procedure NEAR, or an intersegment RET if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

JMP *target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack, and no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte form of this instruction called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within ±32k. Intrasegment direct JMPS are self-relative and are appropriate in position-independent (dynamically relocatable) routines in which the JMP and its target are in the same segment and are moved together.

An intrasegment indirect JMP may be made either through memory or through a 16-bit general register. In the first case, the content of the word referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP, and the second word replaces CS.

Conditional Transfers

The conditional transfer instructions are jumps that may or may not transfer control depending on the state of the CPU flags at the time the instruction is executed. These 18 instructions (see table 2-15) each test a different combination of flags for a condition. If the condition is "true," then control is transferred to the target specified in the instruction. If the condition is "false," then control passes to the instruction that follows the conditional jump. All conditional jumps are SHORT, that is, the target must be in the current code segment and within −128 to +127 bytes of the first byte of the next instruction (JMP 00H jumps to the first byte of the next instruction). Since the jump is made by adding the relative displacement of the target to the instruction pointer, all conditional jumps are self-relative and are appropriate for position-independent routines.

Iteration Control

The iteration control instructions can be used to regulate the repetition of software loops. These instructions use the CX register as a counter. Like the conditional transfers, the iteration control instructions are self-relative and may only transfer to targets that are within −128 to +127 bytes of themselves, i.e., they are SHORT transfers.

LOOP *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

LOOPE/LOOPZ *short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and

Mnemonics © Intel, 1978

Table 2-15. Interpretation of Conditional Transfers

MNEMONIC	CONDITION TESTED	"JUMP IF ..."
JA/JNBE	(CF OR ZF)=0	above/not below nor equal
JAE/JNB	CF=0	above or equal/not below
JB/JNAE	CF=1	below/not above nor equal
JBE/JNA	(CF OR ZF)=1	below or equal/not above
JC	CF=1	carry
JE/JZ	ZF=1	equal/zero
JG/JNLE	((SF XOR OF) OR ZF)=0	greater/not less nor equal
JGE/JNL	(SF XOR OF)=0	greater or equal/not less
JL/JNGE	(SF XOR OF)=1	less/not greater nor equal
JLE/JNG	((SF XOR OF) OR ZF)=1	less or equal/not greater
JNC	CF=0	not carry
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	overflow
JP/JPE	PF=1	parity/parity equal
JS	SF=1	sign

Note: "above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE/LOOPZ is executed.

LOOPNE/LOOPNZ *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is clear; otherwise the next sequential instruction is executed.

JCXZ *short-label*

JCXZ (Jump If CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

Interrupt Instructions

The interrupt instructions allow interrupt service routines to be activated by programs as well as by external hardware devices. The effect of software interrupts is similar to hardware-initiated interrupts. However, the processor does not execute an interrupt acknowledge bus cycle if the interrupt originates in software or with an NMI. The effect of the interrupt instructions on the flags is covered in the description of each instruction.

INT *interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap (TF) and interrupt-enable (IF) flags to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word of the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type = 3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as "supervisor calls," i.e., requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

INTO

INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt procedure (its type is 4) through the interrupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

IRET

IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

Processor Control Instructions

These instructions (see table 2-16) allow programs to control various CPU functions. One group of instructions updates flags, and another group is used primarily for synchronizing the 8086 or 8088 with external events. A final instruction causes the CPU to do nothing. Except for the flag operations, none of the processor control instructions affect the flags.

Flag Operations

CLC

CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no other flags. It (and CMC and STC) is useful in conjunction with the RCL and RCR instructions.

Table 2-16. Processor Control Instructions

FLAG OPERATIONS	
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
EXTERNAL SYNCHRONIZATION	
HLT	Halt until interrupt or reset
WAIT	Wait for $\overline{\text{TEST}}$ pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
NO OPERATION	
NOP	No operation

CMC

CMC (Complement Carry flag) "toggles" CF to its opposite state and affects no other flags.

STC

STC (Set Carry flag) sets CF to 1 and affects no other flags.

CLD

CLD (Clear Direction flag) zeroes DF causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

STD

STD (Set Direction flag) sets DF to 1 causing the string instructions to auto-decrement the SI and/or DI index registers. STD does not affect any other flags.

Mnemonics © Intel, 1978

CLI

CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words maskable interrupts are disabled. A non-maskable interrupt appearing on the NMI line, however, is honored, as is a software interrupt. CLI does not affect any other flags.

STI

STI (Set Interrupt-enable flag) sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed. STI does not affect any other flags.

External Synchronization

HLT

HLT (Halt) causes the 8086/8088 to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a non-maskable interrupt request on NMI, or, if interrupts are enabled, upon receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

WAIT

WAIT causes the CPU to enter the wait state while its $\overline{\text{TEST}}$ line is not active. WAIT does not affect any flags. This instruction is described more completely in section 2.5.

ESC *external-opcode, source*

ESC (Escape) provides a means for an external processor to obtain an opcode and possibly a memory operand from the 8086 or 8088. The external opcode is a 6-bit immediate constant that the assembler encodes in the machine instruction it builds (see table 2-26). An external processor may monitor the system bus and capture this opcode when the ESC is fetched. If the source operand is a register, the processor does nothing. If the source operand is a memory variable, the processor obtains the operand from memory and discards it. An external processor may capture the memory operand when the processor reads it from memory.

LOCK

LOCK is a one-byte prefix that causes the 8086/8088 (configured in maximum mode) to assert its bus $\overline{\text{LOCK}}$ signal while the following instruction executes. LOCK does not affect any flags. See section 2.5 for more information on LOCK.

No Operation

NOP

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

Instruction Set Reference Information

Table 2-21 provides detailed operational information for the 8086/8088 instruction set. The information is presented from the point of view of utility to the assembly language programmer. Tables 2-17, 2-18 and 2-19 explain the symbols used in table 2-21. Machine language instruction encoding and decoding information is given in Chapter 4.

Instruction timings are presented as the number of clock periods required to execute a particular form (register-to-register, immediate-to-memory, etc.) of the instruction. If a system is running with a 5 MHz maximum clock, the maximum clock period is 200 ns; at 8 MHz, the clock period is 125 ns. Where memory operands are used, "+EA" denotes a variable number of additional clock periods needed to calculate the operand's effective address (discussed in section 2.8). Table 2-20 lists all effective address calculation times.

Table 2-17. Key to Instruction Coding Formats

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location or immediate value that is used in the operation, but is not altered by the instruction.
source-table	XLAT	Name of memory translation table addressed by register BX.
target	JMP, CALL	A label to which control is to be transferred directly, or a register or memory location whose *content* is the address of the location to which control is to be transferred indirectly.
short-label	cond. transfer, iteration control	A label to which control is to be conditionally transferred; must lie within −128 to +127 bytes of the first byte of the next instruction.
accumulator	IN, OUT	Register AX for word transfers, AL for bytes.
port	IN, OUT	An I/O port number; specified as an immediate value of 0-255, or register DX (which contains port number in range 0-64k).
source-string	string ops.	Name of a string in memory that is addressed by register SI; used only to identify string as byte or word and specify segment override, if any. This string is used in the operation, but is not altered.
dest-string	string ops.	Name of string in memory that is addressed by register DI; used only to identify string as byte or word. This string receives (is replaced by) the result of the operation.
count	shifts, rotates	Specifies number of bits to shift or rotate; written as immediate value 1 or register CL (which contains the count in the range 0-255).
interrupt-type	INT	Immediate value of 0-255 identifying interrupt pointer number.
optional-pop-value	RET	Number of bytes (0-64k, ordinarily an even number) to discard from stack.
external-opcode	ESC	Immediate value (0-63) that is encoded in the instruction for use by an external processor.

Mnemonics © Intel, 1978

Table 2-18. Key to Flag Effects

IDENTIFIER	EXPLANATION
(blank)	not altered
0	cleared to 0
1	set to 1
X	set or cleared according to result
U	undefined—contains no reliable value
R	restored from previously-saved value

For control transfer instructions, the timings given include any additional clocks required to reinitialize the instruction queue as well as the time required to fetch the target instruction. For instructions executing on an 8086, four clocks should be added for each instruction reference to a word operand located at an odd memory address to reflect any additional operand bus cycles required. Similarly for instructions executing on an 8088, four clocks should be added to each instruction reference to a 16-bit memory operand; this includes all stack operations. The required number of data references is listed in table 2-21 for each instruction to aid in this calculation.

Several additional factors can increase actual execution time over the figures shown in table 2-21. The time provided assumes that the instruction has already been prefetched and that it is waiting in the instruction queue, an assumption that is valid under most, but not all, operating conditions. A series of fast executing (fewer than two clocks per opcode byte) instructions can drain the queue and increase execution time. Execution time also is slightly impacted by the interaction of the EU and BIU when memory operands must be read or written. If the EU needs access to memory, it may have to wait for up to one clock if the BIU has already started an instruction fetch bus cycle. (The EU can detect the need for a memory operand and post a bus request far enough in advance of its need for this operand to avoid waiting a full 4-clock bus cycle). Of course the EU does not have to wait if the queue is full, because the BIU is idle. (This discussion assumes

Table 2-19. Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	An 8- or 16-bit general register
reg 16	A 16-bit general register
seg-reg	A segment register
accumulator	Register AX or AL
immediate	A constant in the range 0-FFFFH
immed8	A constant in the range 0-FFH
memory	An 8- or 16-bit memory location[1]
mem8	An 8-bit memory location[1]
mem16	A 16-bit memory location[1]
source-table	Name of 256-byte translate table
source-string	Name of string addressed by register SI
dest-string	Name of string addressed by register DI
DX	Register DX
short-label	A label within −128 to +127 bytes of the end of the instruction
near-label	A label in current code segment
far-label	A label in another code segment
near-proc	A procedure in current code segment
far-proc	A procedure in another code segment
memptr16	A word containing the offset of the location in the current code segment to which control is to be transferred[1]
memptr32	A doubleword containing the offset and the segment base address of the location in another code segment to which control is to be transferred[1]
regptr16	A 16-bit general register containing the offset of the location in the current code segment to which control is to be transferred
repeat	A string instruction repeat prefix

[1] Any addressing mode—direct, register indirect, based, indexed, or based indexed—may be used (see section 2.8).

Table 2-20. Effective Address Calculation Time

EA COMPONENTS		CLOCKS*
Displacement Only		6
Base or Index Only	(BX,BP,SI,DI)	5
Displacement + Base or Index	(BX,BP,SI,DI)	9
Base + Index	BP + DI, BX + SI	7
	BP + SI, BX + DI	8
Displacement + Base + Index	BP + DI + DISP BX + SI + DISP	11
	BP + SI + DISP BX + DI + DISP	12

*Add 2 clocks for segment override

that the BIU can obtain the bus on demand, i.e., that no other processors are competing for the bus.)

With typical instruction mixes, the time actually required to execute a sequence of instructions will typically be within 5-10% of the sum of the individual timings given in table 2-21. Cases can be constructed, however, in which execution time may be much higher than the sum of the figures provided in the table. The execution time for a given sequence of instructions, however, is always repeatable, assuming comparable external conditions (interrupts, coprocessor activity, etc.). If the execution time for a given series of instructions must be determined exactly, the instructions should be run on an execution vehicle such as the SDK-86 or the iSBC 86/12™ board.

Table 2-21. Instruction Set Reference Data

AAA	**AAA** (no operands) ASCII adjust for addition				**Flags** O D I T S Z A P C U U U X U X
Operands		**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)		4	—	1	AAA

AAD	**AAD** (no operands) ASCII adjust for division				**Flags** O D I T S Z A P C U X X U X U
Operands		**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)		60	—	2	AAD

AAM	**AAM** (no operands) ASCII adjust for multiply				**Flags** O D I T S Z A P C U X X U X U
Operands		**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)		83	—	1	AAM

AAS	**AAS** (no operands) ASCII adjust for subtraction				**Flags** O D I T S Z A P C U U U X U X
Operands		**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)		4	—	1	AAS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

ADC

ADC destination,source
Add with carry

Flags

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	ADC AX, SI
register, memory	9+EA	1	2-4	ADC DX, BETA [SI]
memory, register	16+EA	2	2-4	ADC ALPHA [BX] [SI], DI
register, immediate	4	—	3-4	ADC BX, 256
memory, immediate	17+EA	2	3-6	ADC GAMMA, 30H
accumulator, immediate	4	—	2-3	ADC AL, 5

ADD

ADD destination,source
Addition

Flags

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	ADD CX, DX
register, memory	9+EA	1	2-4	ADD DI, [BX].ALPHA
memory, register	16+EA	2	2-4	ADD TEMP, CL
register, immediate	4	—	3-4	ADD CL, 2
memory, immediate	17+EA	2	3-6	ADD ALPHA, 2
accumulator, immediate	4	—	2-3	ADD AX, 200

AND

AND destination,source
Logical and

Flags

O	D	I	T	S	Z	A	P	C
0				X	X	U	X	0

Operands	Clocks	Transfers*	Bytes	Coding Example
register, register	3	—	2	AND AL,BL
register, memory	9+EA	1	2-4	AND CX,FLAG_WORD
memory, register	16+EA	2	2-4	AND ASCII [DI],AL
register, immediate	4	—	3-4	AND CX,0F0H
memory, immediate	17+EA	2	3-6	AND BETA, 01H
accumulator, immediate	4	—	2-3	AND AX, 01010000B

CALL

CALL target
Call a procedure

Flags

O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Examples
near-proc	19	1	3	CALL NEAR_PROC
far-proc	28	2	5	CALL FAR_PROC
memptr 16	21+EA	2	2-4	CALL PROC_TABLE [SI]
regptr 16	16	1	2	CALL AX
memptr 32	37+EA	4	2-4	CALL [BX].TASK [SI]

CBW

CBW (no operands)
Convert byte to word

Flags

O	D	I	T	S	Z	A	P	C

Operands	Clocks	Transfers*	Bytes	Coding Example
(no operands)	2	—	1	CBW

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

CLC	CLC (no operands) Clear carry flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLC

CLD	CLD (no operands) Clear direction flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLD

CLI	CLI (no operands) Clear interrupt flag			Flags O D I T S Z A P C 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CLI

CMC	CMC (no operands) Complement carry flag			Flags O D I T S Z A P C X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	CMC

CMP	CMP destination,source Compare destination to source			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	CMP BX, CX
register, memory	9+EA	1	2-4	CMP DH, ALPHA
memory, register	9+EA	1	2-4	CMP [BP+2], SI
register, immediate	4	—	3-4	CMP BL, 02H
memory, immediate	10+EA	1	3-6	CMP [BX].RADAR [DI], 3420H
accumulator, immediate	4	—	2-3	CMP AL, 00010000B

CMPS	CMPS dest-string,source-string Compare string			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	22	2	1	CMPS BUFF1, BUFF2
(repeat) dest-string, source-string	9+22/rep	2/rep	1	REPE CMPS ID, KEY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

CWD	CWD (no operands) Convert word to doubleword			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	5	—	1	CWD

DAA	DAA (no operands) Decimal adjust for addition			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	DAA

DAS	DAS (no operands) Decimal adjust for subtraction			Flags O D I T S Z A P C U X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	DAS

DEC	DEC destination Decrement by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	DEC AX
reg8	3	—	2	DEC AL
memory	15+EA	2	2-4	DEC ARRAY [SI]

DIV	DIV source Division, unsigned			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-90	—	2	DIV CL
reg16	144-162	—	2	DIV BX
mem8	(86-96) +EA	1	2-4	DIV ALPHA
mem16	(150-168) +EA	1	2-4	DIV TABLE [SI]

ESC	ESC external-opcode,source Escape			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immediate, memory	8+EA	1	2-4	ESC 6,ARRAY [SI]
immediate, register	2	—	2	ESC 20,AL

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

HLT	HLT (no operands) Halt			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	HLT

IDIV	IDIV source Integer division			Flags O D I T S Z A P C U U U U U U
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	101-112	—	2	IDIV BL
reg16	165-184	—	2	IDIV CX
mem8	(107-118) +EA	1	2-4	IDIV DIVISOR_BYTE [SI]
mem16	(171-190) +EA	1	2-4	IDIV [BX].DIVISOR_WORD

IMUL	IMUL source Integer multiplication			Flags O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	80-98	—	2	IMUL CL
reg16	128-154	—	2	IMUL BX
mem8	(86-104) +EA	1	2-4	IMUL RATE_BYTE
mem16	(134-160) +EA	1	2-4	IMUL RATE_WORD [BP] [DI]

IN	IN accumulator,port Input byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
accumulator, immed8	10	1	2	IN AL, 0FFEAH
accumulator, DX	8	1	1	IN AX, DX

INC	INC destination Increment by 1			Flags O D I T S Z A P C X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16	2	—	1	INC CX
reg8	3	—	2	INC BL
memory	15+EA	2	2-4	INC ALPHA [DI] [BX]

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

INT	INT interrupt-type Interrupt			Flags	O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
immed8 (type = 3) immed8 (type ≠ 3)	52 51	5 5	1 2	INT 3 INT 67	

INTR†	INTR (external maskable interrupt) Interrupt if INTR and IF=1			Flags	O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
(no operands)	61	7	N/A	N/A	

INTO	INTO (no operands) Interrupt if overflow			Flags	O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
(no operands)	53 or 4	5	1	INTO	

IRET	IRET (no operands) Interrupt Return			Flags	O D I T S Z A P C R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
(no operands)	24	3	1	IRET	

JA/JNBE	JA/JNBE short-label Jump if above/Jump if not below nor equal			Flags	O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
short-label	16 or 4	—	2	JA ABOVE	

JAE/JNB	JAE/JNB short-label Jump if above or equal/Jump if not below			Flags	O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
short-label	16 or 4	—	2	JAE ABOVE_EQUAL	

JB/JNAE	JB/JNAE short-label Jump if below/Jump if not above nor equal			Flags	O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
short-label	16 or 4	—	2	JB BELOW	

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†INTR is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

JBE/JNA	JBE/JNA short-label Jump if below or equal/Jump if not above			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNA NOT_ABOVE

JC	JC short-label Jump if carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JC CARRY_SET

JCXZ	JCXZ short-label Jump if CX is zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	JCXZ COUNT_DONE

JE/JZ	JE/JZ short-label Jump if equal/Jump if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JZ ZERO

JG/JNLE	JG/JNLE short-label Jump if greater/Jump if not less nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JG GREATER

JGE/JNL	JGE/JNL short-label Jump if greater or equal/Jump if not less			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JGE GREATER_EQUAL

JL/JNGE	JL/JNGE short-label Jump if less/Jump if not greater nor equal			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JL LESS

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

JLE/JNG	JLE/JNG short-label Jump if less or equal/Jump if not greater			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNG NOT_GREATER

JMP	JMP target Jump			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	15	—	2	JMP SHORT
near-label	15	—	3	JMP WITHIN_SEGMENT
far-label	15	—	5	JMP FAR_LABEL
memptr16	18+EA	1	2-4	JMP [BX].TARGET
regptr16	11	—	2	JMP CX
memptr32	24+EA	2	2-4	JMP OTHER.SEG [SI]

JNC	JNC short-label Jump if not carry			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNC NOT_CARRY

JNE/JNZ	JNE/JNZ short-label Jump if not equal/Jump if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNE NOT_EQUAL

JNO	JNO short-label Jump if not overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNO NO_OVERFLOW

JNP/JPO	JNP/JPO short-label Jump if not parity/Jump if parity odd			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPO ODD_PARITY

JNS	JNS short-label Jump if not sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JNS POSITIVE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

JO	JO short-label Jump if overflow			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JO SIGNED_OVRFLW

JP/JPE	JP/JPE short-label Jump if parity/Jump if parity even			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JPE EVEN_PARITY

JS	JS short-label Jump if sign			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	16 or 4	—	2	JS NEGATIVE

LAHF	LAHF (no operands) Load AH from flags			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	LAHF

LDS	LDS destination,source Load pointer using DS			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Example**
reg16, mem32	16 + EA	2	2-4	LDS SI,DATA.SEG [DI]

LEA	LEA destination,source Load effective address			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem16	2 + EA	—	2-4	LEA BX, [BP] [DI]

LES	LES destination,source Load pointer using ES			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg16, mem32	16 + EA	2	2-4	LES DI, [BX].TEXT_BUFF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

LOCK	LOCK (no operands) Lock bus			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	LOCK XCHG FLAG,AL

LODS	LODS source-string Load string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
source-string (repeat) source-string	12 9+13/rep	1 1/rep	1 1	LODS CUSTOMER_NAME REP LODS NAME

LOOP	LOOP short-label Loop			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	17/5	—	2	LOOP AGAIN

LOOPE/LOOPZ	LOOPE/LOOPZ short-label Loop if equal/Loop if zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	18 or 6	—	2	LOOPE AGAIN

LOOPNE/LOOPNZ	LOOPNE/LOOPNZ short-label Loop if not equal/Loop if not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
short-label	19 or 5	—	2	LOOPNE AGAIN

NMI†	NMI (external nonmaskable interrupt) Interrupt if NMI = 1			Flags O S I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†NMI is not an instruction; it is included in table 2-21 only for timing information.

Table 2-21. Instruction Set Reference Data (Cont'd.)

MOV	**MOV** destination,source Move			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
memory, accumulator	10	1	3	MOV ARRAY [SI], AL
accumulator, memory	10	1	3	MOV AX, TEMP_RESULT
register, register	2	—	2	MOV AX,CX
register, memory	8+EA	1	2-4	MOV BP, STACK_TOP
memory, register	9+EA	1	2-4	MOV COUNT [DI], CX
register, immediate	4	—	2-3	MOV CL, 2
memory, immediate	10+EA	1	3-6	MOV MASK [BX] [SI], 2CH
seg-reg, reg16	2	—	2	MOV ES, CX
seg-reg, mem16	8+EA	1	2-4	MOV DS, SEGMENT_BASE
reg16, seg-reg	2	—	2	MOV BP, SS
memory, seg-reg	9+EA	1	2-4	MOV [BX].SEG_SAVE, CS

MOVS	**MOVS** dest-string,source-string Move string			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string, source-string	18	2	1	MOVS LINE EDIT_DATA
(repeat) dest-string, source-string	9+17/rep	2/rep	1	REP MOVS SCREEN, BUFFER

MOVSB/MOVSW	**MOVSB/MOVSW** (no operands) Move string (byte/word)			**Flags** O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	18	2	1	MOVSB
(repeat) (no operands)	9+17/rep	2/rep	1	REP MOVSW

MUL	**MUL** source Multiplication, unsigned			**Flags** O D I T S Z A P C X U U U U X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
reg8	70-77	—	2	MUL BL
reg16	118-133	—	2	MUL CX
mem8	(76-83) +EA	1	2-4	MUL MONTH [SI]
mem16	(124-139) +EA	1	2-4	MUL BAUD_RATE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

NEG	NEG destination Negate			Flags O D I T S Z A P C X X X X X 1*
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	3	—	2	NEG AL
memory	16+EA	2	2-4	NEG MULTIPLIER

*0 if destination = 0

NOP	NOP (no operands) No Operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	3	—	1	NOP

NOT	NOT destination Logical not			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	3	—	2	NOT AX
memory	16+EA	2	2-4	NOT CHARACTER

OR	OR destination,source Logical inclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	OR AL, BL
register, memory	9+EA	1	2-4	OR DX, PORT_ID [DI]
memory, register	16+EA	2	2-4	OR FLAG_BYTE, CL
accumulator, immediate	4	—	2-3	OR AL, 01101100B
register, immediate	4	—	3-4	OR CX,01H
memory, immediate	17+EA	2	3-6	OR [BX].CMD_WORD,0CFH

OUT	OUT port,accumulator Output byte or word			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
immed8, accumulator	10	1	2	OUT 44, AX
DX, accumulator	8	1	1	OUT DX, AL

POP	POP destination Pop word off stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	8	1	1	POP DX
seg-reg (CS illegal)	8	1	1	POP DS
memory	17+EA	2	2-4	POP PARAMETER

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

POPF	POPF (no operands) Pop flags off stack			Flags O D I T S Z A P C R R R R R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	8	1	1	POPF

PUSH	PUSH source Push word onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register	11	1	1	PUSH SI
seg-reg (CS legal)	10	1	1	PUSH ES
memory	16 + EA	2	2-4	PUSH RETURN_CODE [SI]

PUSHF	PUSHF (no operands) Push flags onto stack			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	10	1	1	PUSHF

RCL	RCL destination,count Rotate left through carry			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCL CX, 1
register, CL	8 + 4/bit	—	2	RCL AL, CL
memory, 1	15 + EA	2	2-4	RCL ALPHA, 1
memory, CL	20 + EA + 4/bit	2	2-4	RCL [BP].PARM, CL

RCR	RCR designation,count Rotate right through carry			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	RCR BX, 1
register, CL	8 + 4/bit	—	2	RCR BL, CL
memory, 1	15 + EA	2	2-4	RCR [BX].STATUS, 1
memory, CL	20 + EA + 4/bit	2	2-4	RCR ARRAY [DI], CL

REP	REP (no operands) Repeat string operation			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REP MOVS DEST, SRCE

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

REPE/REPZ	REPE/REPZ (no operands) Repeat string operation while equal/while zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPE CMPS DATA, KEY

REPNE/REPNZ	REPNE/REPNZ (no operands) Repeat string operation while not equal/not zero			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	REPNE SCAS INPUT_LINE

RET	RET optional-pop-value Return from procedure			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(intra-segment, no pop)	8	1	1	RET
(intra-segment, pop)	12	1	3	RET 4
(inter-segment, no pop)	18	2	1	RET
(inter-segment, pop)	17	2	3	RET 2

ROL	ROL destination,count Rotate left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers**	**Bytes**	**Coding Examples**
register, 1	2	—	2	ROL BX, 1
register, CL	8+4/bit	—	2	ROL DI, CL
memory, 1	15+EA	2	2-4	ROL FLAG_BYTE [DI],1
memory, CL	20+EA+4/bit	2	2-4	ROL ALPHA , CL

ROR	ROR destination,count Rotate right			Flags O D I T S Z A P C X X
Operand	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	ROR AL, 1
register, CL	8+4/bit	—	2	ROR BX, CL
memory, 1	15+EA	2	2-4	ROR PORT_STATUS, 1
memory, CL	20+EA+4/bit	2	2-4	ROR CMD_WORD, CL

SAHF	SAHF (no operands) Store AH into flags			Flags O D I T S Z A P C R R R R R
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	4	—	1	SAHF

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

SAL/SHL	SAL/SHL destination,count Shift arithmetic left/Shift logical left			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Examples**
register,1	2	—	2	SAL AL,1
register, CL	8+4/bit	—	2	SHL DI, CL
memory,1	15+EA	2	2-4	SHL [BX].OVERDRAW, 1
memory, CL	20+EA+4/bit	2	2-4	SAL STORE_COUNT, CL

SAR	SAR destination,source Shift arithmetic right			Flags O D I T S Z A P C X X X U X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SAR DX, 1
register, CL	8+4/bit	—	2	SAR DI, CL
memory, 1	15+EA	2	2-4	SAR N_BLOCKS, 1
memory, CL	20+EA+4/bit	2	2-4	SAR N_BLOCKS, CL

SBB	SBB destination,source Subtract with borrow			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	SBB BX, CX
register, memory	9+EA	1	2-4	SBB DI, [BX].PAYMENT
memory, register	16+EA	2	2-4	SBB BALANCE, AX
accumulator, immediate	4	—	2-3	SBB AX, 2
register, immediate	4	—	3-4	SBB CL, 1
memory, immediate	17+EA	2	3-6	SBB COUNT [SI], 10

SCAS	SCAS dest-string Scan string			Flags O D I T S Z A P C X X X X X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	15	1	1	SCAS INPUT_LINE
(repeat) dest-string	9+15/rep	1/rep	1	REPNE SCAS BUFFER

SEGMENT†	SEGMENT override prefix Override to specified segment			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	MOV SS:PARAMETER, AX

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†ASM-86 incorporates the segment override prefix into the operand specification and not as a separate instruction. SEGMENT is included in table 2-21 only for timing information.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

SHR	SHR destination,count Shift logical right			Flags O D I T S Z A P C X X
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, 1	2	—	2	SHR SI, 1
register, CL	8+4/bit	—	2	SHR SI, CL
memory, 1	15+EA	2	2-4	SHR ID_BYTE [SI] [BX], 1
memory, CL	20+EA+4/bit	2	2-4	SHR INPUT_WORD, CL

SINGLE STEP†	SINGLE STEP (Trap flag interrupt) Interrupt if TF = 1			Flags O D I T S Z A P C 0 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	50	5	N/A	N/A

STC	STC (no operands) Set carry flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STC

STD	STD (no operands) Set direction flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STD

STI	STI (no operands) Set interrupt enable flag			Flags O D I T S Z A P C 1
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
(no operands)	2	—	1	STI

STOS	STOS dest-string Store byte or word string			Flags O D I T S Z A P C
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
dest-string	11	1	1	STOS PRINT_LINE
(repeat) dest-string	9+10/rep	1/rep	1	REP STOS DISPLAY

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

†SINGLE STEP is not an instruction; it is included in table 2-21 only for timing information.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

SUB	SUB destination,source Subtraction			Flags O D I T S Z A P C X X X X X X	
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
register, register	3	—	2	SUB CX, BX	
register, memory	9+EA	1	2-4	SUB DX, MATH_TOTAL [SI]	
memory, register	16+EA	2	2-4	SUB [BP+2], CL	
accumulator, immediate	4	—	2-3	SUB AL, 10	
register, immediate	4	—	3-4	SUB SI, 5280	
memory, immediate	17+EA	2	3-6	SUB [BP].BALANCE, 1000	

TEST	TEST destination,source Test or non-destructive logical and			Flags O D I T S Z A P C 0 X X U X 0	
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
register, register	3	—	2	TEST SI, DI	
register, memory	9+EA	1	2-4	TEST SI, END_COUNT	
accumulator, immediate	4	—	2-3	TEST AL, 00100000B	
register, immediate	5	—	3-4	TEST BX, 0CC4H	
memory, immediate	11+EA	—	3-6	TEST RETURN_CODE, 01H	

WAIT	WAIT (no operands) Wait while $\overline{\text{TEST}}$ pin not asserted			Flags O D I T S Z A P C	
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
(no operands)	3 + 5n	—	1	WAIT	

XCHG	XCHG destination,source Exchange			Flags O D I T S Z A P C	
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
accumulator, reg16	3	—	1	XCHG AX, BX	
memory, register	17+EA	2	2-4	XCHG SEMAPHORE, AX	
register, register	4	—	2	XCHG AL, BL	

XLAT	XLAT source-table Translate			Flags O D I T S Z A P C	
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**	
source-table	11	1	1	XLAT ASCII_TAB	

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

Mnemonics © Intel, 1978

Table 2-21. Instruction Set Reference Data (Cont'd.)

XOR	XOR destination,source Logical exclusive or			Flags O D I T S Z A P C 0 X X U X 0
Operands	**Clocks**	**Transfers***	**Bytes**	**Coding Example**
register, register	3	—	2	XOR CX, BX
register, memory	9+EA	1	2-4	XOR CL, MASK_BYTE
memory, register	16+EA	2	2-4	XOR ALPHA [SI], DX
accumulator, immediate	4	—	2-3	XOR AL, 01000010B
register, immediate	4	—	3-4	XOR SI, 00C2H
memory, immediate	17+EA	2	3-6	XOR RETURN_CODE, 0D2H

*For the 8086, add four clocks for each 16-bit word transfer with an odd address. For the 8088, add four clocks for each 16-bit word transfer.

2.8 Addressing Modes

The 8086 and 8088 provide many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory or in I/O ports. In addition, the addresses of memory and I/O port operands can be calculated in several different ways. These addressing modes greatly extend the flexibility and convenience of the instruction set. This section briefly describes register and immediate operands and then covers the 8086/8088 memory and I/O addressing modes in detail.

Register and Immediate Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all instruction forms. This is because the register "addresses" are encoded in instructions in just a few bits, and because these operations are performed entirely within the CPU (no bus cycles are run). Registers may serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue; like a register operand, no bus cycles need to be run to obtain an immediate operand. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Memory Addressing Modes

Whereas the EU has direct access to register and immediate operands, memory operands must be transferred to or from the CPU over the bus. When the EU needs to read or write a memory operand, it must pass an offset value to the BIU. The BIU adds the offset to the (shifted) content of a segment register producing a 20-bit physical address and then executes the bus cycle(s) needed to access the operand.

The Effective Address

The offset that the EU calculates for a memory operand is called the operand's effective address or EA. It is an unsigned 16-bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides. The EU can calculate the effective address in several different ways. Information encoded in the second byte of the instruction tells the EU how to calculate the effective address of each memory operand. A compiler or assembler derives this information from the statement or instruction written by the programmer. Assembly language programmers have access to all addressing modes.

Figure 2-34 shows that the execution unit calculates the EA by summing a displacement, the content of a base register and the content of an index register. The fact that any combination of these three components may be present in a given instruction gives rise to the variety of 8086/8088 memory addressing modes.

Mnemonics © Intel, 1978

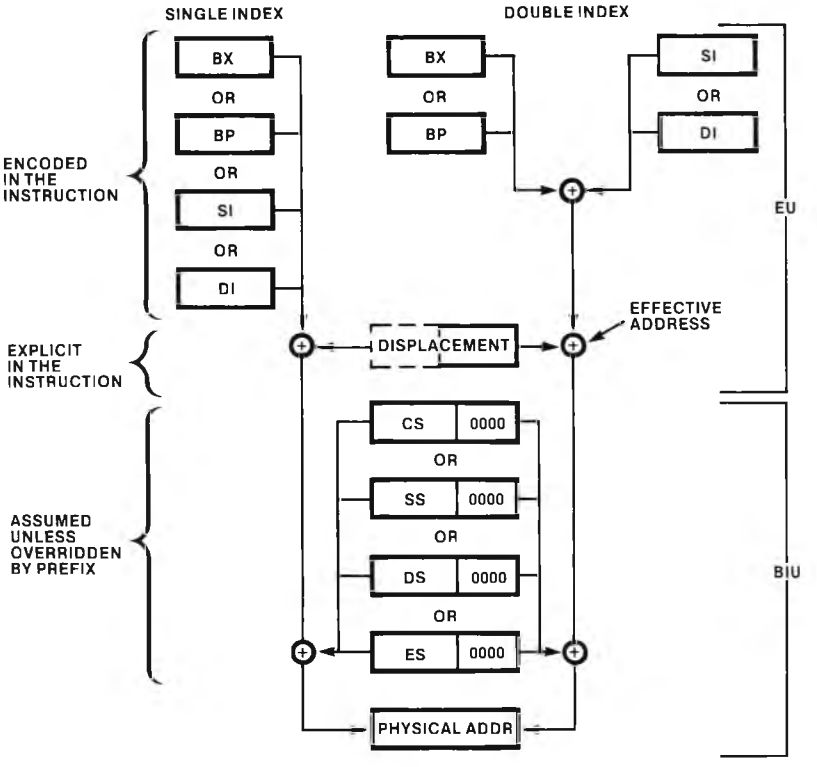


Figure 2-34. Memory Address Computation

The displacement element is an 8- or 16-bit number that is contained in the instruction. The displacement generally is derived from the position of the operand name (a variable or label) in the program. It also is possible for a programmer to modify this value or to specify the displacement explicitly.

A programmer may specify that either BX or BP is to serve as a base register whose content is to be used in the EA computation. Similarly, either SI or DI may be specified as an index register. Whereas the displacement value is a constant, the contents of the base and index registers may change during execution. This makes it possible for one instruction to access different memory locations as determined by the current values in the base and/or index registers.

It takes time for the EU to calculate a memory operand's effective address. In general, the more elements in the calculation, the longer it takes. Table 2-20 shows how much time is required to compute an effective address for any combination of displacement, base register and index register.

Direct Addressing

Direct addressing (see figure 2-35) is the simplest memory addressing mode. No registers are involved; the EA is taken directly from the displacement field of the instruction. Direct addressing typically is used to access simple variables (scalars).

Register Indirect Addressing

The effective address of a memory operand may be taken directly from one of the base or index registers as shown in figure 2-36. One instruction can operate on many different memory locations if the value in the base or index register is updated

appropriately. The LEA (load effective address) and arithmetic instructions might be used to change the register value.

Note that *any* 16-bit general register may be used for register indirect addressing with the JMP or CALL instructions.

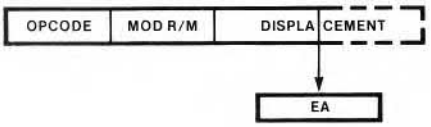


Figure 2-35. Direct Addressing

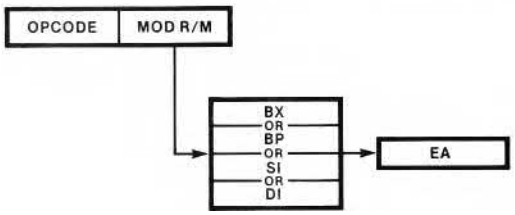


Figure 2-36. Register Indirect Addressing

Based Addressing

In based addressing (figure 2-37), the effective address is the sum of a displacement value and the content of register BX or register BP. Recall that specifying BP as a base register directs the BIU to obtain the operand from the current stack segment (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data (see section 2.10 for examples).

Based addressing also provides a straightforward way to address structures which may be located at different places in memory (see figure 2-38). A base register can be pointed at the base of the structure and elements of the structure addressed by their displacements from the base. Different copies of the same structure can be accessed by simply changing the base register.

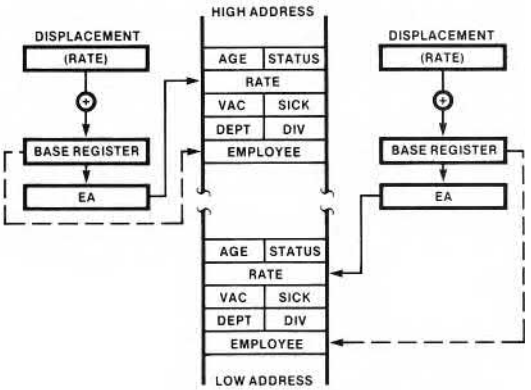


Figure 2-38. Accessing a Structure With Based Addressing

Indexed Addressing

In indexed addressing, the effective address is calculated from the sum of a displacement plus the content of an index register (SI or DI) as shown in figure 2-39. Indexed addressing often is

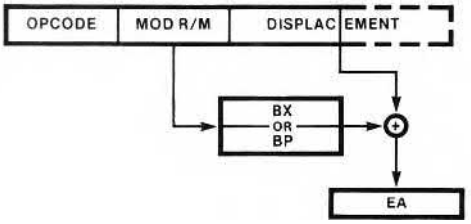


Figure 2-37. Based Addressing

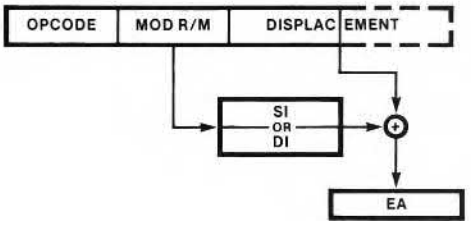


Figure 2-39. Indexed Addressing

used to access elements in an array (see figure 2-40). The displacement locates the beginning of the array, and the value of the index register selects one element (the first element is selected if the index register contains 0). Since all array elements are the same length, simple arithmetic on the index register will select any element.

Based Indexed Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register and a displacement (see figure 2-41). Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack (see figure 2-42). Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

Arrays contained in structures and matrices (two-dimension arrays) also could be accessed with based indexed addressing.

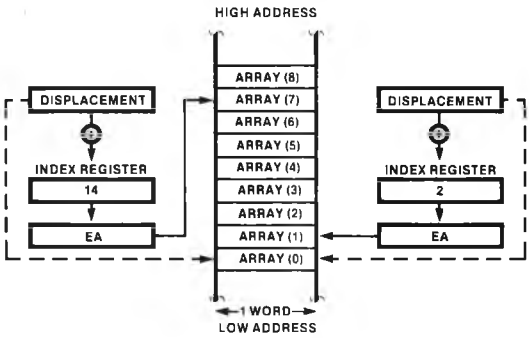


Figure 2-40. Accessing an Array With Indexed Addressing

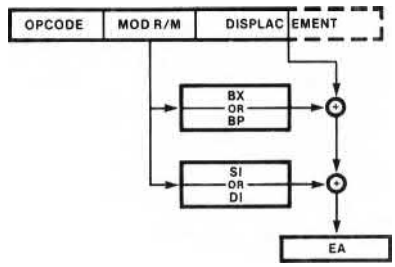


Figure 2-41. Based Indexed Addressing

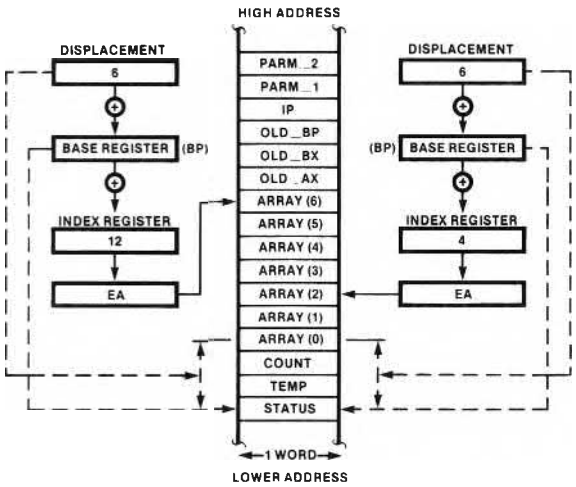


Figure 2-42. Accessing a Stack Array With Based Indexed Addressing

String Addressing

String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly as shown in figure 2-43. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation, the CPUs automatically adjust SI and DI to obtain subsequent bytes or words.

I/O Port Addressing

If an I/O port is memory mapped, any of the memory operand addressing modes may be used to access the port. For example, a group of terminals can be accessed as an "array." String instructions also can be used to transfer data to memory-mapped ports with an appropriate hardware interface. Section 2.10 contains examples of addressing memory-mapped I/O ports.

Two different addressing modes can be used to access ports located in the I/O space; these are illustrated in figure 2-44. In direct port addressing, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0-255. Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and can range from 0 to 65,535. By previously adjusting the content of register DX, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value in DX.

2.9 Programming Facilities

A comprehensive integrated set of tools supports 8086/8088 software development. These tools are programs that run on Intellec® 800 or Series II Microcomputer Development Systems under the ISIS-II operating system, the same hardware and operating system used to develop software for the 8080 and the 8085. Since the 8086 and 8088 are software-compatible with one another, the same tools are used for both processors to provide programmers with a uniform development environment.

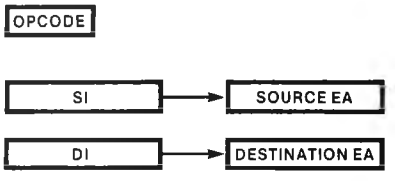


Figure 2-43. String Operand Addressing

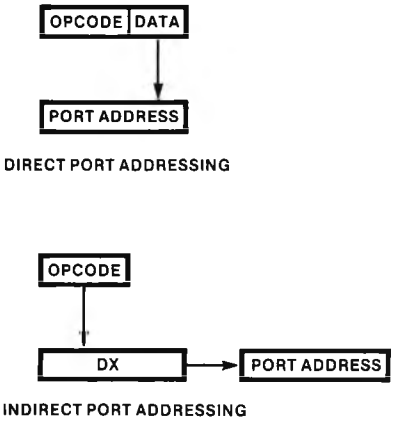


Figure 2-44. I/O Port Addressing

Software Development Overview

A program that will ultimately execute on an 8086- or 8088-based system is developed in steps (see figure 2-45). The overall program is composed of functional units called modules. For purposes of this discussion, a module is a section of code that is separately created, edited, and compiled or assembled. A very small program might consist of a single module; a large program could be comprised of 100 or more modules. The 8086/8088 LINK-86 utility binds modules together into a single program. (The module structure of a program is critical to its successful development and maintenance; see section 2.10 for guidelines.)

8086 and 8088 modules can be written in either PL/M-86 or ASM-86 (see table 2-22). PL/M-86 is a high-level language suitable for most microprocessor applications. It is easy to use, even by programmers who have little experience with microprocessors. Because it reduces software development time, PL/M-86 is ideal for most of the programming in any application, especially applications that must get to market quickly.

ASM-86 is the 8086/8088 assembly language. ASM-86 provides the programmer who is familiar with the CPU architecture, access to all processor features. For critical code segments within programs that make sophisticated use of the hardware, have extremely demanding performance or memory constraints, ASM-86 is the best choice.

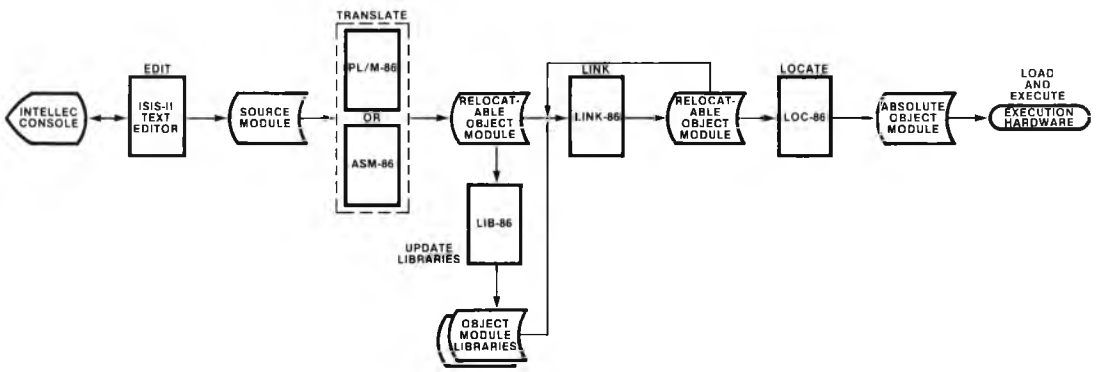


Figure 2-45. Software Development Process

Table 2-22. PL/M-86/ASM-86 Characteristics

PL/M-86	ASM-86
• Fast Development	• Fastest Execution Speed
• Less Programmer Training	• Smallest Memory Requirements
• Detailed Hardware Knowledge Not Required	• Access To All Processor Facilities

The languages are completely compatible, and a judicious combination of the two often makes good sense. Prototype software can be developed rapidly with PL/M-86. When the system is operating correctly, it can be analyzed to see which sections can best profit from being written in ASM-86. Since the logic of these sections already has been debugged, selective rewriting can be done quickly and with low risk.

Each PL/M-86 or ASM-86 module (called a source moduel) is keyed into the Intellec® system using the ISIS-II text editor and is stored as a diskette file. This source file is then input to the appropriate language translator (ASM-86 assembler or PL/M-86 compiler). The language translator creates a diskette file from the source file, which is called a relocatable object module. The translator also lists the program and flags any errors detected during the translation. The relocatable object module contains the 8086/8088 machine instructions that the translator created from the statements in the source module. The term "relocatable" refers to the fact that all references to memory locations in the module are relative, rather than being absolute memory addresses. The module generally is not executable until the relative references are changed to the actual memory locations where the module will reside in the execution system's memory. The process of changing the relative references to absolute memory locations is called locating.

There are very good reasons for not locating modules when they are translated. First, the execution system's physical memory configuration (where RAM and ROM/PROM segments are actually located in the megabyte memory space) may not be known at the time the modules are written. Second, it is desirable to be able to use a common module (e.g., a square root routine) in more than one system. If absolute addresses were assigned at translation time, the common module would either have to occupy the same physical addresses in every system, or separate versions with different addresses would have to be maintained for each system. When locating is deferred, a single version of a common routine can be used by any number of systems. Finally, the locations of modules typically change as a system is developed, maintained and enhanced. Separating the location process from the translation process means that as modifications are made, unchanged modules only need to be relocated, not retranslated.

Relocatable object modules may be placed into special files called libraries, using the LIB-86 library manager program. Libraries provide a convenient means of collecting groups of related modules so that they can be accessed automatically by the LINK-86 program.

When enough relocatable object modules have been created to test the system, or part of it, the modules are linked and located. Linking combines all the separate modules into a single program. Locating changes the relative memory references in the program to the actual memory locations where the program will be loaded in the execution system. The link and locate process also is referred to as R & L, for relocation and linkage.

Two other programs round out the software development tools available for the 8086 and 8088. OH-86 converts an absolute object file into a hexadecimal format used by some PROM programmers and system loaders (for example, the SDK-86 and iSBC 957™ loaders). CONV-86 can do most of the conversion work required to translate 8080/8085 assembly language source modules into ASM-86 source modules.

The 8086/8088 software development facilities are covered in more detail in the remainder of this section. However, these are only introductions to

the use of these tools. Complete documentation is available in the following publications available from Intel's Literature Department:

ISIS-II:

ISIS-II System User's Guide, Order No. 9800306

ASM-86:

MCS-86 Assembly Language Reference Manual, Order No. 9800640

MCS-86 Assembler Operating Instructions for ISIS-II Users, Order No. 9800641

PL/M-86:

PL/M-86 Programming Manual, Order No. 9800466

ISIS-II PL/M-86 Compiler Operator's Manual, Order No. 9800478

LINK-86, LOC-86, LIB-86, OH-86:

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, Order No. 9800639

CONV-86:

MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users, Order No. 9800642

PL/M-86

PL/M-86 is a general-purpose, high-level language for programming the 8086 and 8088 microprocessors. It is an extension of PL/M-80, the most widely-used, high-level programming language for microprocessors. (PL/M-80 source programs can be processed by the PL/M-86 compiler; the resulting object program is generally reduced by 15-30% in size.) PL/M-86 is suitable for all types of microprocessor software from operating systems to application programs.

PL/M-86's purpose is simple: to reduce the time and cost of developing and maintaining software for the 8086 and 8088. It accomplishes this by creating a programming environment that, for the most part, is distinct from the architecture of the CPUs. Registers, segments, addressing modes, stacks, etc., are effectively "invisible" to the PL/M-86 programmer. Instead, the processors appear to respond to simple commands and familiar algebraic expressions. The responsibility for translating these source statements into the machine instructions ultimately required to execute on the 8086/8088 is assumed by the PL/M-86 compiler. By "hiding" the details of the machine architecture, PL/M-86 encourages programmers to concentrate on solving the problem at hand. Furthermore, because PL/M-86 is closer to natural language, it is easier to "think in PL/M-86" than it is to "think in assembly language." This speeds up the expression of a program solution, and, equally important, makes that solution easier for someone other than the original programmer to understand. PL/M-86 also contains all the constructs necessary for structured programming.

Statements and Comments

A programmer builds a PL/M-86 program by writing statements and comments (see figure 2-46). There are several different types of statements in PL/M-86; they always end with a semicolon. Blanks can be used freely before, within, and after statements to improve readability. A statement also may span more than one line.

The characters "/*" start a comment, and the characters "*/" end it; any characters may be used in between. Comments do not affect the execution of a PL/M-86 program, but all good programs are thoughtfully commented. Comments are notes that document and clarify the program's operation; they may be written virtually anywhere in a PL/M-86 program.

Data Definition

Most PL/M-86 programs begin by defining the data items (variables) with which they are going to work. An individual PL/M-86 data element is called a scalar. Every scalar variable has a programmer-supplied name up to 31 characters long, and a type. PL/M-86 supports five types of scalars: byte, word, integer, real, and pointer. Table 2-23 lists the characteristics of these PL/M-86 data types.

```
/*TRAFFIC DATA RECORDER CONTROL PROGRAM*
 *VERSION 2.2, RELEASE 5, 23APR79.*
 *THIS RELEASE FIXES THREE BUGS*
 *DOCUMENTED IN PROBLEM REPORT #16.*/

/*COMPUTE TOTAL PAYMENT DUE*/
TOTAL = PRINCIPAL + INTEREST;

IF TERMINAL$READY
    THEN CALL FILL$BUFFER;
    ELSE CALL WAIT (50);      /*WAIT 50 MS FOR RESPONSE*/
```

Figure 2-46. PL/M-86 Statements and Comments

Table 2-23. PL/M-86 Data Types

TYPE	BYTES	RANGE	USAGE
BYTE	1	0 to 255	Unsigned Integer, Character
WORD	2	0 to 65,535	Unsigned Integer
INTEGER	2	−32,768 to +32,767	Signed Integer
REAL	4	1×10^{-38} to $3.37 \times 10^{+38}$	Floating Point
POINTER	2/4	N/A	Address Manipulation

Variables are defined by writing a DECLARE statement of this form:

```
DECLARE scalar-name type;
```

Options of the DECLARE statement can be used to specify an initial value for the scalar and to define a series of items in a shorthand form.

Besides scalar variables, scalar constants may be used in PL/M-86 programs (see figure 2-47). Constants may be written "as is" or may be given names to improve program clarity.

Scalars can be aggregated into named collections of data such as arrays and structures. An array is a collection of scalars of the same type (all integer, all real, etc.). Arrays are useful for representing data that has a repetitive nature. For example, monthly rainfall samples could be represented as an array of 12 elements, one for each month:

```
DECLARE RAINFALL (12) REAL;
```

Each element in an array is accessible by a number called a subscript which is the element's relative location in the array. In PL/M-86, the first element in an array has a subscript of 0; it is considered the "0th" element. Thus, RAINFALL (11) refers to December's sample. The subscript need not be a constant; variables and expressions also may be used as subscripts.

Strings of character data are typically defined as byte arrays. Characters can be accessed with subscripts or with powerful string-handling functions built into PL/M-86.

```
        10  /*DECIMAL NUMBER*/
       0AH  /*HEXADECIMAL NUMBER*/
       12Q  /*OCTAL NUMBER*/
  00001010B  /*BINARY NUMBER*/
      10.0  /*FLOATING POINT NUMBER*/
     1.0E1  /*FLOATING POINT NUMBER*/
       'A'  /*CHARACTER*/

/*CONSTANTS MAY BE GIVEN NAMES*/
DECLARE STATUS$PORT LITERALLY '0FFEH';
DECLARE THRESHOLD LITERALLY '98.6';
```

Figure 2-47. PL/M-86 Constants

A structure is a collection of related data elements that do not necessarily have the same type. The elements are related by virtue of "belonging" to the entity represented by the structure. Here is a simple structure declaration:

```
DECLARE BRIDGE STRUCTURE
        (SPAN        WORD,
        YR$BUILT     BYTE,
        AVG$TRAFFIC REAL);
```

The year the bridge was built could be accessed by writing BRIDGE.YR$BUILT; the structure element name is "qualified" by the dot and the structure name. This allows structures with the same element names to be distinguished from each other (e.g., HIGHWAY.YR$BUILT).

Arrays and structures can be combined into more complex data aggregates:

- array elements may be structures rather than scalars,
- a structure element may be an array,
- structures in arrays may themselves contain arrays.

Figure 2-48 provides sample PL/M-86 data declarations.

Assignment Statement

Data that has been defined can be operated on with PL/M-86 executable statements. The fundamental executable statement is the assignment statement, written in this form:

```
variable-name = expression;
```

This means "evaluate the expression and assign (move) the result to the variable."

There are three basic classes of expressions in PL/M-86; arithmetic, relational and logical (see table 2-24 and figure 2-49). All expressions are combinations of operands and operators, although an expression can consist of a single operand. Operands are variables and constants; operators vary according to the type of expression. Evaluation of an expression always yields a single result; different classes of expressions yield different types of results.

Table 2-24. Characteristics of PL/M-86 Expressions

EXPRESSION	OPERATORS	RESULT
ARITHMETIC	+, −, *, /, MOD	NUMBER
RELATIONAL	>, <, =, >=, <=	"TRUE" - FFH "FALSE" - 0H
LOGICAL	AND, OR, XOR, NOT	8/16-BIT STRING

```
    /****SCALARS****/
DECLARE SWITCH          BYTE;
DECLARE COUNT           WORD,                   /*1 SCALAR*/
        INDEX           INTEGER;                /*1 SCALAR*/
DECLARE (NET, GROSS,  TOTAL)   REAL;            /*3 SCALARS*/

    /****ARRAYS****/
DECLARE MONTH (12)      BYTE;
DECLARE TERMINAL_LINE (80)          BYTE;

    /****STRUCTURE****/
DECLARE EMPLOYEE STRUCTURE
        (ID_NUMBER                  WORD,
        DEPARTMENT                  BYTE
        RATE                        REAL);

    /****ARRAY OF STRUCTURES****/
DECLARE INVENTORY_ITEM (100)        STRUCTURE
        (PART_NUMBER                WORD,
        ON_HAND                     WORD,
        RE_ORDER                    BYTE);

    /****ARRAY WITHIN STRUCTURE****/
DECLARE COUNTY_DATA                 STRUCTURE
        (NAME (20)                  BYTE,
        TEN_YR_RAINFALL(10)         BYTE,
        PER CAPITA_INCOME           REAL);
```

Figure 2-48. PL/M-86 Data Declarations

```
/*ARITHMETIC*/
A = 2; B = 3;
B = B+ 1;                   /*B CONTAINS 4*/
C = (A*B) -2;               /*C CONTAINS 6*/
C = ((A*B) + 3) MOD 3;      /*C CONTAINS 2*/

/*RELATIONAL*/
A =2; B= 3
C = B > A;                  /*C CONTAINS 0FFH*/
C = B < > A;                /*C CONTAINS 0FFH*/
C = B = (A+1);              /*C CONTAINS 0FFH*/

/*LOGICAL*/
A = 0011$0001B;             /*$ IS FOR READABILITY*/
B = 1000$0001B;
C = NOT B;                  /*C CONTAINS 0111$1110B*/
C = A AND B;                /*C CONTAINS 0000$0001B*/
C = A OR B;                 /*C CONTAINS 1011$0001B*/
C = B XOR A;                /*C CONTAINS 1011$0000B*/
C = (A AND B) OR 0F0H;      /*C CONTAINS 1111$0001B*/
```

Figure 2-49. Expressions in PL/M-86 Assignment Statements

Program Flow Statements

Simple PL/M-86 programs can be written with just DECLARE and assignment statements. Such programs, however, execute exactly the same sequence of statements every time they are run and would not prove very useful. PL/M-86 provides statements that change the flow of control through a program. These statements allow sections of the program to be executed selectively, repeated, skipped entirely, etc.

The IF statement (figure 2-50) selects one or the other of two statements for execution depending on the result of a relational expression. The IF statement is written:

```
IF relational-expression

   THEN statement1;

   ELSE statement2;
```

Statement1 is executed if the expression is "true"; statement2 is not executed in this case. If the relation is "false," statement1 is skipped and statement2 is executed. In determining the "truth" of an expression, the IF statement only examines the low-order bit of the result (1="true"). Therefore, arithmetic and logical expressions also may be used in an IF statement.

```
A = 3; B = 5;
IF A < B
  THEN MINIMUM = 1;        /*EXECUTED*/
  ELSE MINIMUM = 2;        /*SKIPPED*/

MORE_DATA = 0FFH;
IF NOT MORE_DATA
  THEN DONE = 1;           /*SKIPPED*/
  ELSE DONE = 0;           /*EXECUTED*/

/*NESTED IF STATEMENTS*/
CLOCK_ON = 1; HOUR=24; ALARM=OFF;
IF CLOCK_ON
  THEN IF HOUR = 24
     THEN IF ALARM = OFF
        THEN HOUR = 0; /*EXECUTED*/
```

Figure 2-50. PL/M-86 IF Statements

A DO block begins with a DO statement and ends with an END statement. All intervening statements are part of the block. A DO block can appear anywhere in a program that an executable statement can appear. There are four kinds of DO statements in PL/M-86: simple DO, DO CASE, interative DO, and DO WHILE.

A simple DO statement (figure 2-51) causes all the statements in the block to be treated as though they were a single statement. Simple DOs enable a single IF statement to cause multiple statements to be executed (the alternative would be to repeat the IF statement for every statement to be executed).

```
/*SIMPLE DO*/
A=5; B=9;
IF (A+2) < B THEN DO;
                X=X-1;      /*EXECUTED*/
                Y(X)=0;     /*EXECUTED*/
                END;
        ELSE    DO;
                X=X+1;      /*SKIPPED*/
                Y(X)=1;     /*SKIPPED*/
                END;


/*DO CASE*/
A = 2;
DO CASE (A);
    X = X+1;    /*SKIPPED*/
    X = X+2;    /*SKIPPED*/
    X = X+3;    /*EXECUTED*/
    X = X+4;    /*SKIPPED*/
    END;
```

Figure 2-51. PL/M-86 Simple DO and DO CASE

DO CASE (figure 2-51) causes one statement in the DO block to be selected and executed depending on the result of the expression (usually arithmetic) written immediately following DO CASE:

```
DO CASE arithmetic-expression;
```

If the expression yields 0, the first statement in the DO block is executed; if the expression yields 1, the second statement is executed, etc. A statement in the DO block may be null (consist of only a semicolon) to cause no action for selected cases. DO CASE provides a rapid and easily-understood way to respond to data like "transaction codes"

where a different action is required for each of many values a code might assume (an alternative would be an IF statement for every value the code could assume).

An iterative DO block (figures 2-52 and 2-53) is executed from 0 to an infinite number of times based on the relationship of an index variable to an expression that terminates execution. The general form is:

```
DO index = start-expr TO stop-expr BY step-expr;
```

The "BY step-expr" is optional, and the step is assumed to be 1 if not supplied (the typical case). When control first reaches the DO statement, start-expr is evaluated and is assigned to index. Then index is compared to stop-expr; if index exceeds stop-expr, control goes to the statement following the DO block, otherwise the block is executed. At the end of the block, the result of step-expr is added to index, and it is compared to stop-expr again, etc. (The iterative DO is quite flexible—this is a simplified explanation.) Iterative DOs are handy for "stepping through" an array. For example, an array of 10 elements could be zeroed by:

```
DO I = 0 TO 9;
    ARRAY(I) = 0;
    END;
```

In a DO WHILE (figures 2-52 and 2-54), the statements are executed repeatedly as long as the expression following WHILE evaluates to "true." DO WHILE often can be applied in situations where an interative DO will not work, or is clumsy, such as where repetition must be controlled by a non-integer value. Like an iterative DO, DO WHILE may be executed from 0 times to an infinite number of times.

```
/*ITERATIVE DO*/
DO I = 0 TO 5;
    ARRAY (I) = I;              /*EXECUTED 6 TIMES*/
    TOTAL = TOTAL+1;            /*EXECUTED 6 TIMES*/
    END;
/*I = 6 AT THIS POINT*/

/*DO WHILE*/
MORE = 0; SPACE_OK =1;
DO WHILE (MORE AND SPACE_OK);
    ITEMS = ITEMS + 1;          /*SKIPPED*/
    N_TRACKS =
    N_TRACKS + 10;              /*SKIPPED*/
    IF N_TRACKS >= 999          /*SKIPPED*/
        THEN SPACE_OK = 0;
    END;


/*DO WHILE*/
CODE = 'A';
DO WHILE (CODE = 'A');
    TEMP = TEMP * STEP;         /*EXECUTION STOPS*/
    IF TEMP > 98.6              /*AFTER TEMP*/
        THEN CODE = 'B';        /*EXCEEDS 98.6*/
        N_STEPS = N_STEPS + 1;
    END;
```

Figure 2-52. PL/M-86 Iterative DO and DO WHILE

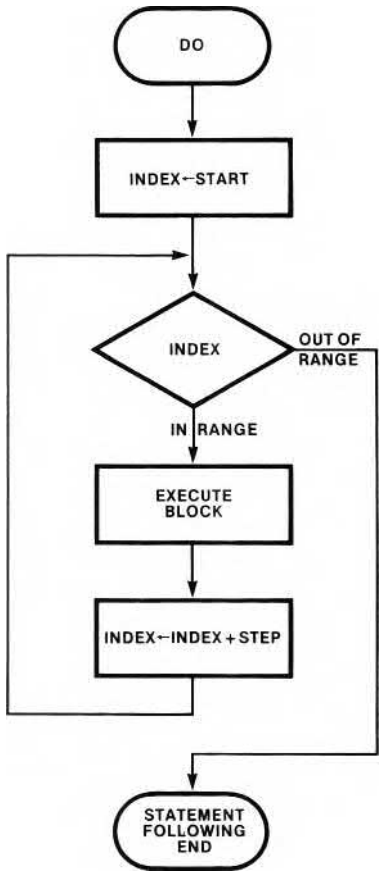


Figure 2-53. PL/M-86 Iterative DO Flowchart

A GOTO written in the form

```
GOTO target;
```

causes an unconditional transfer (branch) to another statement in the program. The statement receiving control would be written

```
target: statement;
```

where "target" is a label identifying the statement.

A CALL statement written in the form

```
CALL proc-name (parm-list);
```

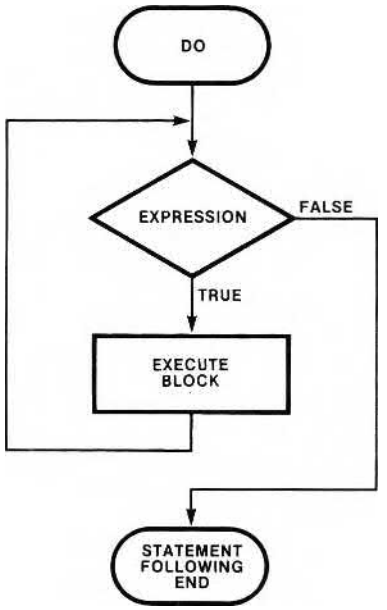


Figure 2-54. PL/M-86 DO WHILE Flowchart

activates a procedure defined earlier in the program. The variables listed in "parm-list" are passed to the procedure, the procedure is executed, and then control returns to the statement following the CALL. Thus, unlike a GOTO, a CALL brings control back to the point of departure.

Procedures

Procedures are "subprograms" that make it possible to simplify the design of complex programs and to share a single copy of a routine among programs. A procedure usually is designed to perform one function; i.e., to solve one part of the total problem with which the program is dealing. For example, a program to calculate paychecks could be broken down into separate procedures for calculating gross pay, income tax, Social Security and net pay. The organization of the "main" program then could be understood at a glance:

```
CALL GROSS_PAY;
CALL INCOME_TAX;
CALL SOCIAL_SECURITY;
CALL NET_PAY;
```

Furthermore, the income tax procedure could be divided into separate procedures for calculating state and federal taxes. Procedures, then, provide a mechanism by which a large, complex problem can be attacked with a "divide and conquer" strategy.

A procedure usually is defined early in a program, but it is only executed when it is referred to by name in a later PL/M-86 statement. A procedure can accept a list of variables, called parameters, that it will use in performing its function. These parameters may assume different values each time the procedure is executed.

PL/M-86 provides two classes of procedures, typed and untyped. A typed procedure returns a value to the statement that activates it and, in addition, may accept parameters from that statement. A typed procedure is activated whenever its name appears in a statement; the value it returns effectively takes the place of the procedure name in the statement. Typed procedures can be used in all kinds of PL/M-86 expressions. Untyped procedures may accept parameters, but do not return a value. Untyped procedures are activated by CALL statements. Figure 2-55 shows how simple typed and untyped procedures may be declared and then activated.

The statements forming the body of a procedure need not exist within the module that activates the procedure. The activating module can declare the procedure EXTERNAL, and the LINK-86 utility will connect the two modules.

PL/M-86 procedures can be written to handle interrupts. Procedures also may be declared REENTRANT, making them concurrently usable by different tasks in a multitasking system. PL/M-86 also has about 50 procedures built into the language, including facilities for:

- converting variables from one type to another
- shifting and rotating bits
- performing input and output
- manipulating strings
- activating the CPU $\overline{\text{LOCK}}$ signal.

```
/*DECLARATION OF A TYPED PROCEDURE THAT
  ACCEPTS TWO REAL PARAMETERS AND RETURNS A REAL VALUE*/
  AVG: PROCEDURE (X,Y) REAL;
       DECLARE (X,Y) REAL;
       RETURN (X+Y)/2.0;
       END AVG;

/*ACTIVATING A TYPED PROCEDURE*/
LOW = 2.0;
HIGH = 3.0;
TOTAL = TOTAL + AVG (LOW,HIGH);  /*2.5 IS ADDED TO TOTAL*/

/*DECLARATION OF AN UNTYPED PROCEDURE
  THAT ACCEPTS ONE PARAMETER*/
TEST: PROCEDURE (X);
  DECLARE X BYTE;
  IF X = 0H THEN
       COUNT = COUNT + 1;
  END TEST;

/*ACTIVATING AN UNTYPED PROCEDURE*/
CALL TEST (ALPHA);  /*COUNT IS INCREMENTED
                      IF ALPHA = 0*/
```

Figure 2-55. PL/M-86 Procedures

ASM-86

Programmers who are familiar with the CPU architecture can obtain complete access to all processor facilities with ASM-86. Since the execution unit on both the 8086 and the 8088 is identical, both processors use the same assembly language. Examples of processor features not accessible through PL/M-86 that can be utilized in ASM-86 programs include: software interrupts, the WAIT and ESC instructions and explicit control of the segment registers.

An ASM-86 program often can be written to execute faster and/or to use less memory than the same program written in PL/M-86. This is because the compiler has a limited "knowledge" of the entire program and must generate a generalized set of machine instructions that will work in all situations, but may not be optimal in a particular situation. For example, assume that the elements of an array are to be summed and the result placed in a variable in memory. The machine instructions generated by the PL/M-86 compiler would move the next array element to a register and then add the register to the sum variable in memory. An ASM-86 programmer, knowing that a register will be "safe" while the array is summed, could instead add all the array elements to a register and then move the register to the sum variable, saving one instruction execution per array element.

It is easier to write assembly language programs in ASM-86 than it is in many assembly languages. ASM-86 contains powerful data structuring facilities that are usually found only in high-level languages. ASM-86 also simplifies the programmer's "view" of the 8086/8088 machine instruction set. For example, although there are 28 different types of MOV machine instructions, the programmer always writes a single form of the instruction:

MOV destination-operand, source-operand

The assembler generates the correct machine-instruction form based on the attributes of the source and destination operands (attributes are covered later in this section). Finally, the ASM-86 assembler performs extensive checks on the consistency of operand definition versus operand use in instructions, catching many common types of clerical errors.

Statements

Compared to many assemblers, ASM-86 accepts a relaxed statement format (see figure 2-56). This helps to reduce clerical errors and allows programmers to format their programs for better readability. Variable and label names may be up to 31 characters long and are not restricted to alphabetic and numeric characters. In particular, the underscore (_) may be used to improve the readability of long names. Blanks may be inserted freely between identifiers (there are no "column" requirements), and statements also may span multiple lines.

All ASM-86 statements are classified as instructions or directives. A clear distinction must be made here between ASM-86 instructions and

```
; THIS STATEMENT CONTAINS A COMMENT ONLY

MOV     AX, [BX + 3]                        ; TYPICAL ASM-86 INSTRUCTION
    MOV AX,         [BX + 3]                ; BLANKS NOT SIGNIFICANT
MOV     AX,
&       [BX + 3]                            ; CONTINUED STATEMENTS

ZERO    EQU     0                           ; SIMPLE ASM-86 DIRECTIVE
CUR_PROJ  EQU     PROJECT [BX] [SI]         ; MORE COMPLEX DIRECTIVE
THE_STACK_STARTS_HERE  SEGMENT              ; LONG IDENTIFIER
TIGHT_LOOP:  JMP TIGHT_LOOP                 ; LABELLED STATEMENT
MOV  ES: DATA_STRING [SI], AL               ; SEGMENT OVERRIDE PREFIX
WAIT:  LOCK XCHG  AX,SEMAPHORE              ; LABEL & LOCK PREFIX
```

Figure 2-56. ASM-86 Statements

Mnemonics © Intel, 1978

8086/8088 machine instructions. The assembler generates machine instructions from ASM-86 instructions written by a programmer. Each ASM-86 instruction produces one machine instruction, but the form of the generated machine instruction will vary according to the operands written in the ASM-86 instruction. For example, writing

```
MOV BL,1
```

produces a byte-immediate-to-register MOV, while writing

```
MOV TERMINAL_NO,BX
```

produces a word-register-to-memory MOV. To the programmer, though, there is simply a MOV source-to-destination instruction.

ASM-86 instructions are written in the form:

```
(label:) (prefix) mnemonic (operand(s)) (;comment)
```

where parentheses denote optional fields (the parentheses are not actually written by programmers). The label field names the storage location containing the machine instruction so that it can be referred to symbolically as the target of a JMP instruction elsewhere in the program. Writing a prefix causes ASM-86 to generate one of the special prefix bytes (segment override, bus lock or repeat) immediately preceding the machine instruction. The mnemonic identifies the type of instruction (MOV for move, ADD for add, etc.) that is to be generated. Zero, one or two operands may be written next, separated by commas, according to the requirements of the instruction. Finally, writing a semicolon signifies that what follows is a comment. Comments do not affect the execution of a program, but they can greatly improve its clarity; all good ASM-86 programs are thoughtfully commented.

Writing a directive gives ASM-86 information to use in generating instructions, but does not itself produce a machine instruction. About 20 different directives are available in ASM-86. Directives are written like this:

```
(name) mnemonic (operand(s)) (;comment)
```

Some directives require a name to be present, while others prohibit a name. ASM-86 recognizes the directive from the mnemonic keyword written in the next field. Any operands required by the directive are written next, separated by commas. A comment may be written as the last field of a directive.

Some of the more commonly used directives define procedures (PROC), allocate storage for variables (DB, DW, DD) give a descriptive name to a number or an expression (EQU), define the bounds of segments (SEGMENT and ENDS), and force instructions and data to be aligned at word boundaries (EVEN).

Constants

Binary, decimal, octal and hexadecimal numeric constants (see figure 2-57) may be written in ASM-86 statements; the assembler can perform basic arithmetic operations on these as well. All numbers must, however, be integers and must be representable in 16 bits including a sign bit. Negative numbers are assembled in standard two's complement notation.

Character constants are enclosed in single quotes and may be up to 255 characters long when used

```
MOV         STRING [SI], 'A'      ; CHARACTER
MOV         STRING [SI], 41H      ; EQUIVALENT IN HEX
ADD         AX, 0C4H              ; HEX CONSTANT MUST START WITH NUMERAL
OCTAL_8     EQU  100              ; OCTAL
OCTAL_9     EQU  10Q              ; OCTAL ALTERNATE
ALL_ONES    EQU  11111111B        ; BINARY
MINUS_5     EQU  -5               ; DECIMAL
MINUS_6     EQU  -6D              ; DECIMAL ALTERNATE
```

Figure 2-57. ASM-86 Constants

to initialize storage. When used as immediate operands, character constants may be one or two bytes long to match the length of the destination operand.

Defining Data

Most ASM-86 programs begin by defining the variables with which they will work. Three directives, DB, DW and DD, are used to allocate and name data storage locations in ASM-86 (see figure 2-58). The directives are used to define storage in three different units: DB means "define byte," DW means "define word," and DD means "define doubleword." The operands of these directives tell the assembler how many storage units to allocate and what initial values, if any, with which to fill the locations.

```
A_SEG     SEGMENT
ALPHA     DB      ?          ; NOT INITIALIZED
BETA      DW      ?          ; NOT INITIALIZED
GAMMA     DD      ?          ; NOT INITIALIZED
DELTA     DB      ?          ; NOT INITIALIZED
EPSILON   DW      5          ; CONTAINS 05H
A_SEG     ENDS

B_SEG     SEGMENT AT 55H ; SPECIFYING BASE ADDRESS
IOTA      DB      'HELLO'    ; CONTAINS 48 45 4C 4C 4F H
KAPPA     DW      'AB'       ; CONTAINS 42 41 H
LAMBDA    DD      B_SEG      ; CONTAINS 0000 5500 H
MU        DB      100 DUP 0  ; CONTAINS (100 X) 00H
B_SEG     ENDS
```

	ATTRIBUTES			OPERATORS	
VARIABLE	SEGMENT	OFFSET	TYPE	LENGTH	SIZE
ALPHA	A_SEG	0	1	1	1
BETA	A_SEG	1	2	1	2
GAMMA	A_SEG	3	4	1	4
DELTA	A_SEG	7	1	1	1
EPSILON	A_SEG	8	2	1	2
IOTA	B_SEG	0	1	5	5
KAPPA	B_SEG	5	2	1	2
LAMBDA	B_SEG	7	4	1	4
MU	B_SEG	11	1	100	100

Figure 2-58. ASM-86 Data Definitions

For every variable in an ASM-86 program, the assembler keeps track of three attributes: segment, offset and type. Segment identifies the segment that contains the variable (segment control is covered shortly). Offset is the distance in bytes of the variable from the beginning of its containing segment. Type identifies the variable's allocation unit (1 = byte, 2 = word, 4 = doubleword). When a variable is referenced in an instruction, ASM-86 uses these attributes to determine what form of the instruction to generate. If the variable's attributes conflict with its usage in an instruction, ASM-86 produces an error message. For example, attempting to add a variable defined as a word to a byte register is an error. There are cases where the assembler must be explicitly told an operand's type. For example, writing MOVE [BX],5 will produce an error message because the assembler does not know if [BX] refers to a byte, a word or a doubleword. The following operators can be used to provide this information: BYTE PTR, WORD PTR and DWORD PTR. In the previous example, a word could be moved to the location referenced by [BX] by writing MOVE WORD PTR [BX],5.

ASM-86 also provides two built-in operators, LENGTH and SIZE, that can be written in ASM-86 instructions along with attribute information. LENGTH causes the assembler to return the number of storage units (bytes, words or doublewords) occupied by an array. SIZE causes ASM-86 to return the total number of bytes occupied by a variable or an array. These operators and attributes make it possible to write generalized instruction sequences that need not be changed (only reassembled) if the attributes of the variables change (e.g., a byte array is changed to a word array). See figure 2-59 for an example of using the attributes and attribute operators.

Records

ASM-86 provides a means of symbolically defining individual bits and strings of bits within a byte or a word. Such a definition is called a record, and each named bit string (which may consist of a single bit) in a record is called a field. Records promote efficient use of storage while at the same time improving the readability of the program and reducing the likelihood of clerical errors. Defining a record does not allocate storage; rather, a record is a template that tells the assembler the name and location of each bit field within the byte or word. When a field name is written later in an instruction, ASM-86 uses the record to generate an immediate mask for instructions like TEST, AND, OR, etc., or an immediate count for shifts and rotates. See figure 2-60 for an example of using a record.

```
; SUM THE CONTENTS OF TABLE INTO AX
TABLE        DW        50  DUP(?)
; NOTE SAME INSTRUCTIONS WOULD WORK FOR
; TABLE      DB        25  DUP(?)
; TABLE      DW       118  DUP(?),   ETC.

             SUB       AX,AX                ; CLEAR SUM
             MOV       CX, LENGTH TABLE ; LOOP TERMINATOR
             MOV       SI, SIZE TABLE       ;POINT SUBSCRIPT
                                            ; TO END OF TABLE
ADD_NEXT:    SUB       SI, TYPE TABLE       ; BACK UP ONE ELEMENT
             ADD       AX, TABLE [SI]       ; ADD ELEMENT
             LOOP      ADD_NEXT             ; UNTIL CX = 0
             ; AX CONTAINS SUM
```

Figure 2-59. Using ASM-86 Attributes and Attribute Operators

```
EMP_BYTE   DB   ?                  ; 1 BYTE, UNINITIALIZED
; BIT DEFINITIONS:
;     7-2     : YEARS EMPLOYED
        1     : SEX (1 = FEMALE)
        0     : STATUS (1 = EXEMPT)
EMP_BITSRECORD                     ;RECORD DEFINED HERE
&           YRS_EMP : 6,
&           SEX : 1,
&           STATUS : 1
.
.
.
; SELECT NONEXEMPT FEMALES EMPLOYED 10 + YEARS

MOV         AL, EMP_BYTE           ; KEEP ORIGINAL INTACT
TEST        AL, MASK SEX           ; FEMALE ?
JZ          REJECT                 ; NO, QUITE
TEST        AL, MASK STATUS        ; NONEXEMPT?
JNZ         REJECT                 ;NO, QUIT
SHR         AL, CL                 ; ISOLATE YEARS
CMP         AL, 11                 ; >=10 YEARS?
JL          REJECT                 ; NO, QUIT
; PROCESS SELECTED EMPLOYEE
.
.
.
REJECT:     ; PROCESS REJECTED EMPLOYEE
.
.
.                                  ; RECORD USED HERE
MOV         CL, YRS_EMP            ; GET SHIFT COUNT
```

Figure 2-60. Using an ASM-86 RECORD Definition

Structures

An ASM-86 structure is a map, or template, that gives names and attributes (length, type, etc.) to a collection of fields. Each field in a structure is defined using DB, DW and DD directives; however, no storage is allocated to the structure. Instead, the structure becomes associated with a particular area of memory when a field name is referenced in an instruction along with a base value. The base value "locates" the structure; it may be a variable name or a base register (BX or BP). The structure may be associated with another area of memory by specifying a different base value. Figure 2-61 shows how a simple structure may be defined and used. Note that a structure field may itself be a structure, allowing much more complex organizations to be laid out.

Structures are particularly useful in situations where the same storage format is at multiple locations, where the location of a collection of variables is not known at assembly-time, and where the location of a collection of variables changes during execution. Applications include multiple buffers for a single file, list processing and stack addressing.

Addressing Modes

Figure 2-62 provides sample ASM-86 coding for each of the 8086/8088 addressing modes. The assembler interprets a bracketed reference to BX, BP, SI or DI as a base or index register to be used to construct the effective address of a memory operand. An unbracketed reference means the register itself is the operand.

The following cases illustrate typical ASM-86 coding for accessing arrays and structures, and show which addressing mode the assembler specifies in the machine instruction it generates:

- If ALPHA is an array, then ALPHA [SI] is the element indexed by SI, and ALPHA [SI + 1] is the following byte (indexed).
- If ALPHA is the base address of a structure and BETA is a field in the structure, then ALPHA.BETA selects the BETA field (direct).
- If register BX contains the base address of a structure and BETA is a field in the structure, then [BX].BETA refers to the BETA field (based).

```
EMPLOYEE        STRUC
    SSN         DB  9     DUP(?)
    RATE        DB  1     DUP(?)
    DEPT        DW  1     DUP(?)
    YR_HIRED    DB  1     DUP(?)
    EMPLOYEE    ENDS

MASTER          DB  12    DUP(?)
TXN             DB  12    DUP(?)

; CHANGE RATE IN MASTER TO VALUE IN TXN.
                MOV       AL, TXN.RATE
                MOV       MASTER.RATE, AL

; ASSUME BX POINTS TO AN AREA CONTAINING
;     DATA IN THE SAME FORMAT AS THE EMPLOYEE
;     STRUCTURE. ZERO THE SECOND DIGIT
;     OF SSN
                MOV       SI, 1   ; INDEX VALUE OF 2ND DIGIT
                MOV       [BX].SSN[SI],0
```

Figure 2-61. Using an ASM-86 Structure

Mnemonics © Intel, 1978

```
ADD   AX, BX                ; REGISTER ← REGISTER
ADD   AL, 5                 ; REGISTER ← IMMEDIATE
ADD   CX, ALPHA             ; REGISTER ← MEMORY (DIRECT)
ADD   ALPHA, 6              ; MEMORY (DIRECT) ← IMMEDIATE
ADD   ALPHA, DX             ; MEMORY (DIRECT) ← REGISTER
ADD   BL, [BX]              ; REGISTER ← MEMORY (REGISTER INDIRECT)
ADD   [SI], BH              ; MEMORY (REGISTER INDIRECT) ← IMMEDIATE
ADD   [PP].ALPHA, AH        ; MEMORY (BASED) ← REGISTER
ADD   CX, ALPHA [SI]        ; REGISTER ← MEMORY (INDEXED)
ADD   ALPHA [DI+2], 10      ; MEMORY (INDEXED) ← IMMEDIATE
ADD   [BX].ALPHA [SI], AL   ; MEMORY (BASED INDEXED) ← REGISTER
ADD   SI, [BP+4] [DI]       ; REGISTER ← MEMORY (BASED INDEXED)
IN    AL, 30                ; DIRECT PORT
OUT   DX, AX                ; INDIRECT PORT
```

Figure 2-62. ASM-86 Addressing Mode Examples

- If register BX contains the address of an array, then [BX] [SI] refers to the element indexed by SI (based indexed).
- If register BX points to a structure whose ALPHA field is an array, then [BX] .ALPHA [SI] selects the element indexed by SI (based indexed).
- If register BX points to a structure whose ALPHA field is itself a structure, then [BX].ALPHA.BETA refers to the BETA field of the ALPHA substructure (based).
- If register BX points to a structure and the ALPHA field of the structure is an array and each element of ALPHA is a structure, then [BX].ALPHA[SI + 3].BETA refers to the field BETA in the element of ALPHA indexed by [SI + 3] (based indexed).

Note that DI may be used in place of SI in these cases and that BP may be substituted for BX. Without a segment override prefix, expressions containing BP refer to the current stack segment, and expressions containing BX refer to the current data segment.

Segment Control

An ASM-86 program is organized into a series of named segments. These are "logical" segments; they are eventually mapped into 8086/8088 memory segments, but this usually is not done until the program is located. A SEGMENT directive starts a segment, and an ENDS directive ends the segment (see figure 2-63). All data and instructions written between SEGMENT and ENDS are part of the named segment. In small programs, variables often are defined in one or two segment(s), stack space is allocated in another segment, and instructions are written in a third or fourth segment. It is perfectly possible, however, to write a complete program in one segment; if this is done, all the segment registers will contain the same base address; that is, the memory segments will completely overlap. Large programs may be divided into dozens of segments.

The first instructions in a program usually establish the correspondence between segment names and segment registers, and then load each segment register with the base address of its corresponding segment. The ASSUME directive tells the assembler what addresses will be in the segment registers at execution time. The assembler checks each memory instruction operand, determines which segment it is in and which segment register contains the address of that segment. If the assumed register is the register expected by the hardware for that instruction type, then the assembler generates the machine instruction normally. If, however, the hardware expects one segment register to be used, and the operand is *not* in the segment pointed to by that register, then the assembler automatically precedes the machine instruction with a segment override prefix byte. (If the segment cannot be overridden, the assembler produces an error message.) An example may clarify this. If register BP is used in an instruction, the 8086 and 8088 CPUs expect, as a default, that the memory operand will be located in the segment pointed to by SS—in the current

```
DATA_SEG      SEGMENT
   ; DATA DEFINITIONS GO HERE
DATA_SEG      ENDS

STACK_SEG   SEGMENT
   ; ALLOCATE 100 WORDS FOR A STACK AND
   ;    LABEL THE INITIAL TOS FOR LOADING SP.
          DW 100 DUP(?)
STACK TOP LABEL WORD
STACK_SEG      ENDS

CODE_SEG      SEGMENT
   ; GIVE ASSEMBLER INITIAL REGISTER-TO-SEGMENT
   ;    CORRESPONDENCE. NOTE THAT IN THIS
   ;    PROGRAM THE EXTRA SEGMENT INITIALLY
   ;    OVERLAPS THE DATA SEGMENT ENTIRELY.
ASSUME  CS: CODE_SEG,
&          DS: DATA_SEG,
&          ES: DATA_SEG,
&          SS: STACK_SEG

START:   ; THIS IS THE BEGINNING OF THE PROGRAM.
         ; LOC-86 WILL PLACE A JMP TO THIS
         ; LOCATION AT ADDRESS FFFF0H.

   ; LOAD THE SEGMENT REGISTERS. CS DOES NOT
   ;    HAVE TO BE LOADED BECAUSE SYSTEM
   ;    RESET SETS IT TO FFFFH, AND THE
   ;    LONG JMP INSTRUCTION AT THAT ADDRESS
   ;    UPDATES IT TO THE ADDRESS OF CODE_SEG.
   ;    SEGMENT REGISTERS ARE LOADED FROM AX
   ;    BECAUSE THERE IS NO IMMEDIATE-TO-
   ;    SEGMENT_REGISTER FORM OF THE MOV
   ;    INSTRUCTION.

                    MOV   AX, DATA_SEG
                    MOV   DS, AX
                    MOV   ES, AX
                    MOV   AX, STACK_SEG
                    MOV   SS, AX
; SET STACK POINTER TO INITIAL TOS.
                    MOV   SP, OFFSET STACK_TOP

; SEGMENTS ARE NOW ADDRESSABLE.
; MAIN PROGRAM CODE GOES HERE.
CODE_SEG                ENDS

; NEXT STATEMENT ENDS ASSEMBLY AND TELLS
;    LOC-86 THE PROGRAMS STARTING ADDRESS.

                    END   START
```

Figure 2-63. Setting Up ASM-86 Segments

Mnemonics © Intel, 1978

stack segment. A programmer may, however, choose to use BP to address a variable in the current data segment—the segment pointed to by DS. The ASSUME directive enables the assembler to detect this situation and to automatically generate the needed override prefix.

It also is possible for a programmer to explicitly code segment override prefixes rather than relying on the assembler. This may result in a somewhat better-documented program since attention is called to the override. The disadvantage of explicit segment overrides is that the assembler does not check whether the operand is in fact addressable through the overriding segment register.

ASM-86, in conjunction with the relocation and linkage facilities, provides much more sophisticated segment handling capabilities than have been described in this introduction. For example, different logical segments may be combined into the same physical segment, and segments may be assigned the same physical locations (allowing a "common" area to be accessed by different programs using different variable and label names).

Procedures

Procedures may be written in ASM-86 as well as in PL/M-86. In fact, procedures written in one language are callable from the other, provided that a few simple conventions are observed in the ASM-86 program. The purpose of ASM-86 procedures is the same as in PL/M-86: to simplify the design of complex programs and to make a single copy of a commonly-used routine accessible from anywhere in the program.

An ASM-86 program activates a procedure with a CALL instruction. The procedure terminates with a RET instruction, which transfers control to the instruction following the CALL. Parameters may be passed in registers or pushed onto the stack before calling the procedure. The RET instruction can discard stack parameters before returning to the caller.

Unlike PL/M-86 procedures, ASM-86 procedures are executable where they are coded, as well as by a CALL instruction. Therefore, ASM-86 procedures often are defined following the main program logic, rather than preceding it as in PL/M-86. Figure 2-64 shows how procedures may be defined and called in ASM-86. Section 2-10 contains examples of procedures that accept parameters on the stack.

LINK-86

Fundamentally, LINK-86 combines separate relocatable object modules into a single program. This process consists primarily of combining (logical) segments of the same name into single segments, adjusting relative addresses when segments are combined, and resolving external references.

A programmer can use a procedure that is actually contained in another module by naming the procedure in an ASM-86 EXTRN directive, or declaring the procedure to be EXTERNAL in PL/M-86. The procedure is defined or declared PUBLIC in the module where it actually resides, meaning that it can be used by other modules. When LINK-86 encounters such an external reference, it searches through the other modules in its input, trying to find the matching PUBLIC declaration. If it finds the referenced object, it links it to the reference, "satisfying" the external reference. If it cannot satisfy the reference, LINK-86 prints a diagnostic message. LINK-86 also checks PL/M-86 procedure calls and function references to insure that the parameters passed to a procedure are the type expected by the procedure.

LINK-86 gives the programmer, particularly the ASM-86 programmer, great control over segments (segments may be combined end to end, renamed, assigned the same locations, etc.). LINK-86 also produces a map that summarizes the link process and lists any unusual conditions encountered. While the output of LINK-86 is generally input to LOC-86, it also may again be input to LINK-86 to permit modules to be linked in incremental groups.

LOC-86

LOC-86 accepts the single relocatable object module produced by LINK-86 and binds the memory references in the module to actual memory addresses. Its output is an absolute object module ready for loading into the memory of an execution vehicle. LOC-86 also inserts a

```
FREQUENCY       DB      256 DUP (0)
.
.
USART_DATA      EQU     0FF0H           ; DATA PORT ADDRESS
USART_STAT      EQU     0FF2H           ; STATUS PORT ADDRESS
.
.
NEXT:           CALL    CHAR_IN
                CALL    COUNT_IT
                JMP     NEXT

CHAR_IN         PROC
    ; THIS PROCEDURE DOES NOT TAKE PARAMETERS.
    ;   IT SAMPLES THE USART STATUS PORT
    ;   UNTIL A CHARACTER IS READY, AND
    ;   THEN READS THE CHARACTER INTO AL
                MOV     DX, USART_STAT
    AGAIN:      IN      AL, DX          ; READ STATUS
                AND     AL, 2           ; CHARACTER PRESENT?
                JZ      AGAIN           ; NO, TRY AGAIN
                MOV     DX, USART_DATA
                IN      AL, DX          ; YES, READ CHARACTER
                RET
CHAR_IN         ENDP

COUNT_IT        PROC
    ; THIS PROCEDURE EXPECTS A CHARACTER IN AL.
    ;   IT INCREMENTS A COUNTER IN A FREQUENCY
    ;   TABLE BASED ON THE BINARY VALUE OF
    ;   THE CHARACTER.
                XOR     AH, AH          ; CLEAR HIGH BYTE
                MOV     SI, AL          ; INDEX INTO TABLE
                INC     FREQUENCY [S]; BUMP THE COUNTER
                RET
COUNT_IT        ENDP
```

Figure 2-64. ASM-86 Procedures

direct intersegment JMP instruction at location FFFF0H. The target of the JMP instruction is the logical beginning of the program. When the 8086 or 8088 is reset, this instruction is automatically executed to restart the system. LOC-86 produces a memory map of the absolute object module and a table showing the address of every symbol defined in the program.

LIB-86

LIB-86 is a valuable adjunct to the R & L programs. It is used to maintain relocatable object modules in special files called libraries. Libraries are a convenient way to make collections of modules available to LINK-86. When a module being linked refers to "external" data or instructions, LINK-86 can automatically search a series of libraries, find the referenced module, and include it in the program being created.

OH-86

OH-86 converts an absolute object module into Intel's standard hexadecimal format. This format is used by some PROM programmers and system loaders, such as the iSBC 957™ and SDK-86 loaders.

Mnemonics © Intel, 1978

CONV-86

Users who have developed substantial, fully-tested assembly language programs for the 8080/8085 microprocessors may want to use CONV-86 to automatically convert large amounts of this code into ASM-86 source code (see figure 2-65). CONV-86 accepts an ASM-80 source program as input and produces an ASM-86 source program as output, plus a print file that documents the conversion and lists any diagnostic messages.

Some programs cannot be completely converted by CONV-86. Exceptions include:

- self-modifying code,
- software timing loops,
- 8085 RIM and SIM instructions,
- interrupt code, and
- macros.

By using the diagnostic messages produced by CONV-86, the converted ASM-86 source file can be manually edited to clean up any sections not converted. A converted program is typically 10-20% larger than the ASM-80 version and does not take full advantage of the 8086/8088 architecture. However, the development time saved by using CONV-86 can make it an attractive alternative to rewriting working programs from scratch.

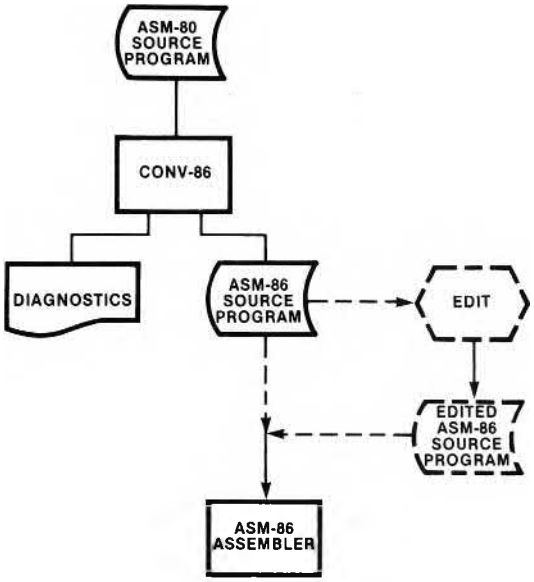


Figure 2-65. ASM-80/ASM-86 Conversion

Sample Programs

Figures 2-66 and 2-67 show how a simple program might be written in PL/M-86 and ASM-86. The program simulates a pair of rolling dice and executes on an Intel SDK-86 System Design Kit. The SDK-86 is an 8086-based computer with memory, parallel and serial I/O ports, a keypad and a display. The SDK-86 is implemented on a single PC board which includes a large prototype area for system expansion and experimentation. A ROM-based monitor program provides a user interface to the system; commands are entered through the keypad and monitor responses are written on the display. With the addition of a cable and software interface (called SDK-C86), the SDK-86 may be connected to an Intellec® Microcomputer Development System. In this mode, the user enters monitor commands from the Intellec keyboard and receives replies on the Intellec CRT display.

The dice program runs on an SDK-86 that is connected to an Intellec® Microcomputer Development System. The program displays two continuously changing digits in the upper left corner of the Intellec display. The digits are random numbers in the range 1-6. A roll is started by entering a monitor GO command. Pressing the INTR key on the SDK-86 keypad stops the roll.

There are two procedures in the PL/M-86 version of the dice program. The first is called CO for console output. This is an untyped PUBLIC procedure that is supplied on an SDK-C86 diskette. CO is written in PL/M-86 and outputs one character to the Intellec console. It is declared EXTERNAL in the dice program because it exists in another module. LINK-86 searches the SDK-C86 library for CO and includes it in the single relocatable object module it builds.

RANDOM is an internal typed procedure; it is contained in the dice module and returns a word value that is a random number between 1 and 6. RANDOM does not use any parameters and is activated in the parameter list passed to CO. When CO is called like this, first RANDOM is activated, then 30 is added to the number it returns and the sum is passed to CO.

```
PL/M-86 COMPILER    DICE

ISIS-II PL/M-86 V1.2 COMPILATION OF MODULE DICE
OBJECT MODULE PLACED IN :F1:DICE.OBJ
COMPILER INVOKED BY:  PLM86 :F1:DICE.P86 XREF

   1          DICE:  DO;
              /* THIS PROGRAM SIMULATES THE ROLL OF A PAIR OF DICE */

              /* GIVE NAMES TO CONSTANTS */
   2   1      DECLARE CLEAR$CRT1       LITERALLY '01BH';  /* INTELLEC  */
   3   1      DECLARE CLEAR$CRT2       LITERALLY '045H';  /*  CRT      */
   4   1      DECLARE HOME$CURSOR1     LITERALLY '01BH';  /*  CONTROL  */
   5   1      DECLARE HOME$CURSOR2     LITERALLY '048H';  /*  CODES    */
   6   1      DECLARE SPACE            LITERALLY '020H';  /*ASCII BLANK*/

              /* PROGRAM VARIABLES */
   7   1      DECLARE (RANDOM$NUMBER,SAVE)  WORD;

              /* CONSOLE OUTPUT PROCEDURE */
   8   1      CO:  PROCEDURE(X) EXTERNAL;
   9   2           DECLARE X    BYTE;
  10   2           END CO;

              /* RANDOM NUMBER GENERATOR PROCEDURE                */
              /* ALGORITHM FOR 16-BIT RANDOM NUMBER FROM:         */
              /*     "A GUIDE TO PL/M PROGRAMMING FOR             */
              /*      MICROCOMPUTER APPLICATIONS,"                */
              /*     DANIEL D. MCCRACKEN,                         */
              /*     ADDISON-WESLEY, 1978                         */
  11   1      RANDOM:  PROCEDURE WORD;
  12   2         RANDOM$NUMBER = SAVE;         /*START WITH OLD NUMBER*/
  13   2         RANDOM$NUMBER = 2053 * RANDOM$NUMBER + 13849;
  14   2         SAVE = RANDOM$NUMBER;         /*SAVE FOR NEXT TIME*/
                 /*FORCE 16-BIT NUMBER INTO RANGE 1-6*/
  15   2         RANDOM$NUMBER = RANDOM$NUMBER MOD 6  + 1;
  16   2         RETURN RANDOM$NUMBER;
  17   2         END RANDOM;

              /* MAIN ROUTINE */
              /* CLEAR THE SCREEN*/
  18   1      CALL CO(CLEAR$CRT1);
  19   1      CALL CO(CLEAR$CRT2);

              /* ROLL THE DICE UNTIL INTERRUPTED */
  20   1      DO WHILE 1;   /*"DO FOREVER"*/
                 /*NOTE THAT ADDING 30 TO THE DIE VALUE */
                 /*  CONVERTS IT TO ASCII.              */
  21   2         CALL CO(RANDOM + 030H);       /*1ST DIE*/
  22   2         CALL CO(SPACE);               /*BLANK*/
  23   2         CALL CO(RANDOM + 030H);       /*2ND DIE*/
                 /* HOME THE CURSOR */
  24   2         CALL CO(HOME$CURSOR1);
  25   2         CALL CO(HOME$CURSOR2);
  26   2         END;

  27   1      END DICE;


CROSS-REFERENCE LISTING
-----------------------

   DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
  ----- ------ -----  ---------------------------------
      2                CLEARCRT1              LITERALLY
                                                18

      3                CLEARCRT2              LITERALLY
                                                19

      8  0000H         CO                     PROCEDURE EXTERNAL(0) STACK=0000H
                                                18   19   21   22   23   24   25

      1  0002H    71  DICE                    PROCEDURE STACK=0004H

      4                HOMECURSOR1            LITERALLY
                                                24

      5                HOMECURSOR2            LITERALLY
                                                25

     11  0049H    44  RANDOM                  PROCEDURE WORD STACK=0002H
                                                21   23
```

Figure 2-66. Sample PL/M-86 Program

```
                                                  12    13    14    15    16

        7  0002H     2  SAVE                    WORD
                                                  12    14

        6               SPACE                   LITERALLY
                                                  22

        8  0000H     1  X                       BYTE PARAMETER
                                                   9

MODULE INFORMATION:

     CODE AREA SIZE      = 0075H    117D
     CONSTANT AREA SIZE  = 0000H      0D
     VARIABLE AREA SIZE  = 0004H      4D
     MAXIMUM STACK SIZE  = 0004H      4D
     51 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure 2-66. Sample PL/M-86 Program (Cont'd.)

```
MCS-86 MACRO ASSEMBLER    DICE

ISIS-II MCS-86 MACRO ASSEMBLER V2.0 ASSEMBLY OF MODULE DICE
OBJECT MODULE PLACED IN :F1:DICE.OBJ
ASSEMBLER INVOKED BY: ASM86 :F1:DICE.A86 XREF

LOC  OBJ                  LINE     SOURCE

                             1     ; THIS PROGRAM SIMULATES THE ROLL OF A PAIR OF DICE
                             2
                             3     ; CONSOLE OUTPUT PROCEDURE
                             4             EXTRN   CO:NEAR
                             5
                             6     ; SEGMENT GROUP DEFINITIONS NEEDED FOR PL/M-86 COMPATIBILITY
                             7     CGROUP  GROUP   CODE
                             8     DGROUP  GROUP   DATA,STACK
                             9
                            10     ; INFORM ASSEMBLER OF SEGMENT REGISTER CONTENTS.
                            11             ASSUME  CS:CGROUP,DS:DGROUP,SS:DGROUP,ES:NOTHING
                            12
                            13     ; ALLOCATE DATA
----                        14     DATA    SEGMENT PUBLIC  'DATA'
                            15     ; NOTE THAT THE FOLLOWING ARE PASSED ON THE STACK TO THE PL/M-86
                            16     ;  PROCEDURE 'CO'.  BY CONVENTION, A BYTE PARAMETER IS PASSED IN
                            17     ;  THE LOW-ORDER 8-BITS OF A WORD ON THE STACK.  HENCE, THESE ARE
                            18     ;  DEFINED AS WORD VALUES, THOUGH THEY OCCUPY 1 BYTE ONLY.
0000 1B00                   19     CLEAR_CRT1      DW      01BH    ; INTELLEC
0002 4500                   20     CLEAR_CRT2      DW      045H    ;   CRT
0004 1B00                   21     HOME_CURSOR1    DW      01BH    ;   CONTROL
0006 4800                   22     HOME_CURSOR2    DW      048H    ;   CODES
0008 2000                   23     SPACE           DW      020H    ; ASCII BLANK
000A ????                   24     SAVE            DW      ?       ; HOLDS LAST 16-BIT RANDOM NUMBER
----                        25     DATA    ENDS
                            26
                            27
                            28     ; ALLOCATE STACK SPACE
----                        29     STACK   SEGMENT STACK   'STACK'
0000 (20                    30             DW      20 DUP (?)
     ????
     )
                            31     ; LABEL INITIAL TOS: FOR LATER USE.
0028                        32     STACK_TOP       LABEL   WORD
----                        33     STACK   ENDS
                            34
                            35
                            36     ; PROGRAM CODE
----                        37     CODE    SEGMENT PUBLIC  'CODE'
                            38
                            39
                            40     ; RANDOM NUMBER GENERATOR PROCEDURE
                            41     ; ALGORITHM FOR 16-BIT RANDOM NUMBER FROM:
                            42     ;   "A GUIDE TO PL/M PROGRAMMING FOR
                            43     ;   MICROCOMPUTER APPLICATIONS,"
                            44     ;   DANIEL D. MCCRACKEN
                            45     ;   ADDISON-WESLEY, 1978
0000                        46     RANDOM  PROC
0000 A10A00          R      47             MOV     AX,SAVE         ; NEW NUMBER =
```

Figure 2-67. ASM-86 Sample Program

```
MCS-86 MACRO ASSEMBLER    DICE

LOC  OBJ                  LINE     SOURCE

0003 B90508                 48             MOV     CX,2053          ;   OLD NUMBER * 2053
0006 F7E1                   49             MUL     CX               ;   + 13849
0008 051936                 50             ADD     AX,13849         ;
000B A30A00          R      51             MOV     SAVE,AX          ; SAVE FOR NEXT TIME
                            52             ; FORCE 16-BIT NUMBER INTO RANGE 1 - 6
                            53             ;   BY MODULO 6 DIVISION + 1
000E 2BD2                   54             SUB     DX,DX            ; CLEAR UPPER DIVIDEND
0010 B90600                 55             MOV     CX,6             ; SET DIVISOR
0013 F7F1                   56             DIV     CX               ; DIVIDE BY 6
0015 8BC2                   57             MOV     AX,DX            ; REMAINDER TO AX
0017 40                     58             INC     AX               ; ADD 1
0018 C3                     59             RET                      ; RESULT IN AX
                            60     RANDOM  ENDP
                            61
                            62
                            63     ; MAIN PROGRAM
                            64
                            65     ; LOAD SEGMENT REGISTERS
                            66     ; NOTE PROGRAM DOES NOT USE ES; CS IS INITIALIZED BY HARDWARE RESET;
                            67     ;  DATA & STACK ARE MEMBERS OF SAME GROUP, SO ARE TREATED AS A SINGLE
                            68     ;  MEMORY SEGMENT POINTED TO BY BOTH DS & SS.
0019 B8----          R      69     START:  MOV     AX,DGROUP
001C 8ED8                   70             MOV     DS,AX
001E 8ED0                   71             MOV     SS,AX
                            72
                            73     ; INITIALIZE STACK POINTER
0020 BC2800          R      74             MOV     SP,OFFSET DGROUP:STACK_TOP
                            75
                            76     ; CLEAR THE SCREEN
0023 FF360000        R      77             PUSH    CLEAR_CRT1
0027 E80000          E      78             CALL    CO
002A FF360200        R      79             PUSH    CLEAR_CRT2
002E E80000          E      80             CALL    CO
                            81
                            82     ; ROLL THE DICE UNTIL INTERRUPTED
0031 E8CCFF                 83     ROLL:   CALL    RANDOM           ; GET 1ST DIE IN AL
0034 0430                   84             ADD     AL,030H          ; CONVERT TO ASCII
0036 50                     85             PUSH    AX               ; PASS IT TO
0037 E80000          E      86             CALL    CO               ;   CONSOLE OUTPUT
003A FF360800        R      87             PUSH    SPACE            ; OUTPUT
003E E80000          E      88             CALL    CO               ;   A BLANK
0041 E8BCFF                 89             CALL    RANDOM           ; GET 2ND DIE IN AL
0044 0430                   90             ADD     AL,030H          ; CONVERT TO ASCII
0046 50                     91             PUSH    AX               ; PASS IT TO
0047 E80000          E      92             CALL    CO               ;   CONSOLE OUTPUT
                            93     ; HOME THE CURSOR
004A FF360400        R      94             PUSH    HOME_CURSOR1
004E E80000          E      95             CALL    CO
0051 FF360600        R      96             PUSH    HOME_CURSOR2
0055 E80000          E      97             CALL    CO
                            98     ; CONTINUE FOREVER
0058 EBD7                   99             JMP     ROLL
----                       100     CODE    ENDS
                           101


XREF SYMBOL TABLE LISTING
---- ------ ----- -------

NAME            TYPE     VALUE  ATTRIBUTES, XREFS

??SEG . . . . SEGMENT          SIZE=0000H PARA PUBLIC
CGROUP. . . . GROUP            CODE   7# 11
CLEAR_CRT1. . V WORD    0000H  DATA  19# 77
CLEAR_CRT2. . V WORD    0002H  DATA  20# 79
CO. . . . . . L NEAR    0000H  EXTRN  4# 78 80 86 88 92 95 97
CODE. . . . . SEGMENT          SIZE=005AH PARA PUBLIC 'CODE'  7# 37 100
DATA. . . . . SEGMENT          SIZE=000CH PARA PUBLIC 'DATA'  8# 14 25
DGROUP. . . . GROUP            DATA STACK   8# 11 11 69 74
HOME_CURSOR1. V WORD    0004H  DATA  21# 94
HOME_CURSOR2. V WORD    0006H  DATA  22# 96
RANDOM. . . . L NEAR    0000H  CODE  46# 60 83 89
ROLL. . . . . L NEAR    0031H  CODE  83# 99
SAVE. . . . . V WORD    000AH  DATA  24# 47 51
SPACE . . . . V WORD    0008H  DATA  23# 87
STACK . . . . SEGMENT          SIZE=0028H PARA STACK 'STACK'
STACK_TOP . . V WORD    0028H  STACK  32# 74
START . . . . L NEAR    0019H  CODE  69# 104


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 2-67. ASM-86 Sample Program (Cont'd.)

Mnemonics © Intel, 1978

The ASM-86 version of the dice program operates like the PL/M-86 version. Since the program uses the PL/M-86 CO procedure for writing data to the Intellec console, it adheres to certain conventions established by the PL/M-86 compiler. The program's logical segments (called CODE, DATA and STACK—the program does not use an extra segment) are organized into two groups called CGROUP and DGROUP. All the members of a group of logical segments are located in the same 64k byte physical memory segment. Physically, the program's DATA and STACK segments can be viewed as "subsegments" of DGROUP.

PL/M-86 procedures expect parameters to be passed on the stack, so the program pushes each character before calling CO. Note that the stack will be "cleaned up" by the PL/M-86 procedure before returning (i.e., the parameter will be removed from the stack by CO).

2.10 Programming Guidelines and Examples

This section addresses 8086/8088 programming from two different perspectives. A series of general guidelines is presented first. These guidelines apply to all types of systems and are intended to make software easier to write, and particularly, easier to maintain and enhance. The second part contains a number of specific programming examples. Written primarily in ASM-86, these examples illustrate how the instruction set and addressing modes may be utilized in various, commonly encountered programming situations.

Programming Guidelines

These guidelines encourage the development of 8086/8088 software that is adaptable to change. Some of the guidelines refer to specific processor features and others suggest approaches to general software design issues. PL/M-86 programmers need not be concerned with the discussions that deal with specific hardware topics; they should, however, give careful attention to the system design subjects. **Systems that are designed in accordance with these recommendations should be less costly to modify or extend. In addition, they should be better-positioned to take advantage of new hardware and software products that are constantly being introduced by Intel.**

Segments and Segment Registers

Segments should be considered as independent logical units whose physical locations in memory *happen* to be defined by the contents of the segment registers. Programs should be independent of the actual contents of the segment registers and of the physical locations of segments in memory. For example, a program should not take advantage of the "knowledge" that two segments are physically adjacent to each other in memory. The single exception to this fully-independent treatment of segments is that a program may set up more than one segment register to point to the same segment in memory, thereby obtaining addressability through more than one segment register. For example, if both DS and ES point to the same segment, a string located in that segment may be used as a source operand in one string instruction and as a destination string in another instruction (recall that a destination string must be located in the extra segment).

Any data aggregate or construct such as an array, a structure, a string or a stack should be restricted to 64k bytes in length and should be wholly contained in one segment (i.e., should not cross a segment boundary).

Segment registers should only contain values supplied by the relocation and linkage facilities. Segment register values may be moved to and from memory, pushed onto the stack and popped from the stack. Segment registers should never be used to hold temporary variables nor should they be altered in any other way.

As an additional guideline, code should *not* be written within six bytes of the end of physical memory (or the end of the code segment if this segment is dynamically relocatable). Failure to observe this guideline could result in an attempted opcode prefetch from non-existent memory, hanging the CPU if READY is not returned.

Self-Modifying Code

It is possible to write a program that deliberately changes some of its own machine instructions

during execution. While this technique may save a few bytes or machine cycles, it does so at the expense of program clarity. This is particularly true if the program is being examined at the machine instruction level; the machine instructions shown in the assembly listing may not match those found in memory or monitored from the bus. It also precludes executing the code from ROM. Also, because of the prefetch queue within the 8086 and 8088, code that is self-modified within six bytes of the current point of execution cannot be guaranteed to execute as intended. (This code may already have been fetched.) Finally, a self-modifying program may prove incompatible with future Intel products that assume that the content of a code segment remains constant during execution.

A corrollary to this requirement is that variable data should not be placed in a code segment. Constant data may be written in a code segment, but this is not recommended for two reasons. First, programs are simpler to understand if they are uniformly subdivided into segments of code, data and stack. Second, placing data in a code segment can restrict the segment's position independence. This is because, in general, the segment base address of a data item may be changed, but the offset (displacement) of the data item may not. This means that the entire segment must be moved as a unit to avoid changing the offset of the constant data. If the constant data were located in a data segment or an extra segment, individual procedures within the code segment could be moved independently.

Input/Output

Since I/O devices vary so widely in their capabilities and their interface designs, I/O software is inevitably device dependent. Substituting a hard disk for a floppy disk, for example, necessitates software changes even though the disks are functionally identical. I/O software can, however, be designed to minimize the effect of device changes on programs.

Figure 2-68 illustrates a design concept that structures an I/O system into a hierarchy of separately compiled/assembled modules. This approach isolates application modules that use the input/output devices from all physical characteristics of the hardware with which they ultimately communicate. An application module that reads a disk file, for example, should have no knowledge of where the file is located on the disk, what size the disk sectors are, etc. This allows these characteristics to change without affecting the application module. To an application module, the I/O system appears to be a series of file-oriented commands (e.g., Open, Close, Read, Write). An application module would typically issue a command by calling a file system procedure.

The file system processes I/O command requests, perhaps checking for gross errors, and calls a procedure in the I/O supervisor. The I/O supervisor is a bridge between the functional I/O request of the application module and the physical I/O performed by the lowest-level modules in the hierarchy. There should be separate modules in the supervisor for different types of devices and some device-dependent code may be unavoidable at this level. The I/O supervisor would typically perform overhead activities such as maintaining disk directories.

The modules that actually communicate with the I/O devices (or their controllers) are at the lowest level in the hierarchy. These modules contain the bulk of the system's device-dependent code that will have to be modified in the event that a device is changed.

The 8089 Input/Output Processor is specifically designed to encourage the development of modular, hierarchical I/O systems. The 8089 allows knowledge of device characteristics to be "hidden" from not only application programs, but also from the operating system that controls the CPU. The CPU's I/O supervisor can simply prepare a message in memory that describes the nature of the operation to be performed, and then activate the 8089. The 8089 independently performs all physical I/O and notifies the CPU when the operation has been completed.

Operating Systems

Operating systems also should be organized in a hierarchy similar to the concept illustrated in figure 2-69. Application modules should "see" only the upper level of the operating system. This level might provide services like sending messages between application modules, providing time delays, etc. An intermediate level might consist of housekeeping routines that dispatch tasks, alter

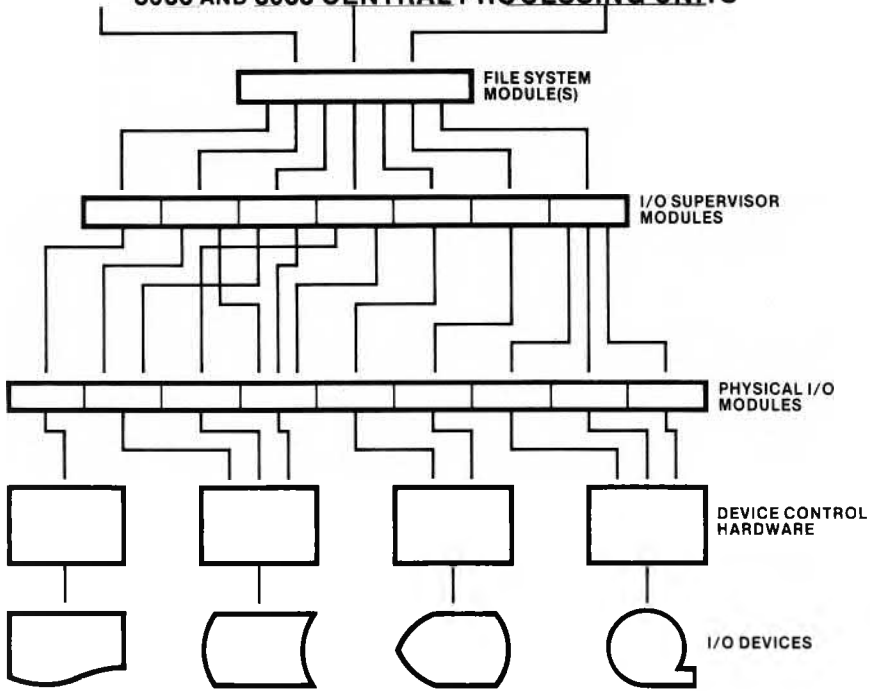


Figure 2-68. I/O System Hierarchy Concept

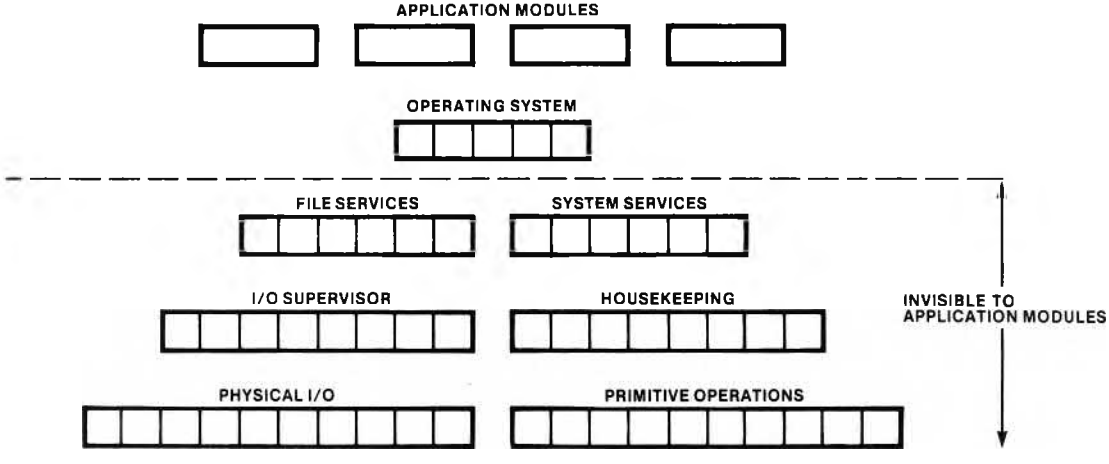


Figure 2-69. Operating System Hierarchy

priorities, manage memory, etc. At the lowest level would be the modules that implement primitive operations such as adding and removing tasks or messages from lists, servicing timer interrupts, etc.

Interrupt Service Procedures

Procedures that service external interrupts should be considered differently than those that service internal interrupts. A service procedure that is activated by an internal interrupt, may, and often should, be made reentrant. External interrupt procedures, on the other hand, should be viewed as temporary tasks. In this sense, a task is a single sequential thread of execution; it should not be reentered. The processor's response to an external interrupt may be viewed as the following sequence of events:

- the running (active) task is suspended,
- a new task, the interrupt service procedure, is created and becomes the running task,
- the interrupt task ends, and is deleted,
- the suspended task is reactived and becomes the running task from the point where it was suspended.

An external interrupt procedure should only be interruptable by a request that activates a different interrupt procedure. When the number of interrupt sources is not too large, this can be accomplished by assigning a different type code and corresponding service procedure to each source. In systems where a large number of similar sources can generate closely spaced interrupts (e.g., 500 communication lines), an approach similar to that illustrated in figure 2-70, may be used to insure that the interrupt service procedure is not reentered, and yet, interrupts arriving in bursts are not missed. The basic technique is to divide the code required to service an interrupt into two parts. The interrupt service procedure itself is kept as short as possible; it performs the absolute minimum amount of processing necessary to service the device. It then builds a message that contains enough information to permit another task, the interrupt message processor, to complete the interrupt service. It adds the message to a queue (which might be implemented as a linked list), and terminates so that it is available to service the next interrupt. The interrupt message processor, which is not reentrant, obtains a message from the queue, finishes processing the interrupt associated with that message, obtains the next message (if there is one), etc. When a burst of interrupts occurs, the queue will lengthen, but interrupts will not be missed so long as there is time for the interrupt service procedure to be activated and run between requests.

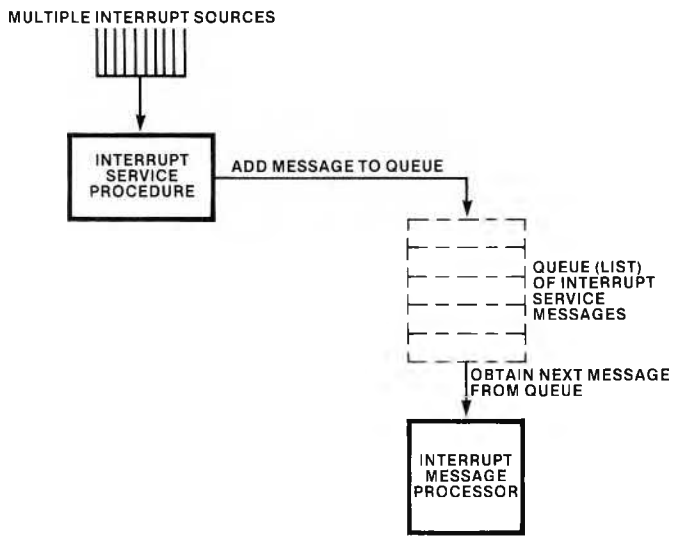


Figure 2-70. Interrupt Message Processor

Stack-Based Parameters

Parameters are frequently passed to procedures on a stack. Results produced by the procedure, however, should be returned in other memory locations or in registers. In other words, the called procedure should "clean up" the stack by discarding the parameters before returning. The RET instruction can perform this function. PL/M-86 procedures always follow this convention.

Flag-Images

Programs should make no assumptions about the contents of the undefined bits in the flag-images stored in memory by the PUSHF and SAHF instructions. These bits always should be masked out of any comparisons or tests that use these flag-images. The undefined bits of the word flag-image can be cleared by ANDing the word with FD5H. The undefined bits of the byte flag-image can be cleared by ANDing the byte with D5H.

Programming Examples

These examples demonstrate the 8086/8088 instruction set and addressing modes in common programming situations. The following topics are addressed:

- procedures (parameters, reentrancy)
- various forms of JMP and CALL instructions
- bit manipulation with the ASM-86 RECORD facility
- dynamic code relocation
- memory mapped I/O
- breakpoints
- interrupt handling
- string operations

These examples are written primarily in ASM-86 and will be of most interest to assembly language programmers. The PL/M-86 compiler generates code that handles many of these situations automatically for PL/M-86 programs. For example, the compiler takes care of the stack in PL/M-86 procedures, allowing the programmer to concentrate on solving the application problem. PL/M-86 programmers, however, may want to examine the memory mapped I/O and interrupt handling examples, since the concepts illustrated are generally applicable; one of the interrupt procedures is written in PL/M-86.

The examples are intended to show one way to use the instruction set, addressing modes and features of ASM-86. They do not demonstrate the "best" way to solve any particular problem. The flexibility of the 8086 and 8088, application differences plus variations in programming style usually add up to a number of ways to implement a programming solution.

Procedures

The code in figure 2-71 illustrates several techniques that are typically used in writing ASM-86 procedures. In this example a calling program invokes a procedure (called EXAMPLE) twice, passing it a different byte array each time. Two parameters are passed on the stack; the first contains the number of elements in the array, and the second contains the address (offset in DATA_SEG) of the first array element. This same technique can be used to pass a variable-length parameter list to a procedure (the "array" could be any series of parameters or parameter addresses). Thus, although the procedure always receives two parameters, these can be used to indirectly access any number of variables in memory.

Any results returned by a procedure should be placed in registers or in memory, but not on the stack. AX or AL is often used to hold a single word or byte result. Alternatively, the calling program can pass the address (or addresses) of a result area to the procedure as a parameter. It is good practice for ASM-86 programs to follow the calling conventions used by PL/M-86; these are documented in *MCS-86 Assembler Operating Instructions For ISIS-II Users*, Order No. 9800641.

EXAMPLE is defined as a FAR procedure, meaning it is in a different segment than the calling program. The calling program must use an intersegment CALL to activate the procedure. Note that this type of CALL saves CS and IP on the stack. If EXAMPLE were defined as NEAR (in the same segment as the caller) then an intrasegment CALL would be used, and only IP would be saved on the stack. It is the responsibility of the calling program to know how the procedure is defined and to issue the correct type of CALL.

```
STACK_SEG       SEGMENT
                DW          20 DUP (?)      ; ALLOCATE 20-WORD STACK

STACK_TOP       LABEL       WORD            ; LABEL INITIAL TOS
STACK_SEG       ENDS

DATA_SEG        SEGMENT
ARRAY_1         DB          10 DUP (?)      ; 10-ELEMENT BYTE ARRAY

ARRAY_2         DB          5 DUP (?)       ; 5-ELEMENT BYTE ARRAY

DATA_SEG        ENDS


PROC_SEG        SEGMENT
ASSUME  CS:PROC_SEG,DS:DATA_SEG,SS:STACK_SEG,ES:NOTHING

EXAMPLE         PROC        FAR             ; MUST BE ACTIVATED BY
                                            ;     INTERSEGMENT CALL

; PROCEDURE PROLOG
                PUSH        BP              ; SAVE BP
                MOV         BP, SP          ; ESTABLISH BASE POINTER
                PUSH        CX              ; SAVE CALLER'S
                PUSH        BX              ;     REGISTERS
                PUSHF                       ;     AND FLAGS
                SUB         SP, 6           ; ALLOCATE 3 WORDS LOCAL STORAGE
                ; END OF PROLOG
; PROCEDURE BODY
                MOV         CX, [BP+8]      ; GET ELEMENT COUNT
                MOV         BX, [BP+6]      ; GET OFFSET OF 1ST ELEMENT
                ; PROCEDURE CODE GOES HERE
                ; FIRST PARAMETER CAN BE ADDRESSED:
                ;   [BX]
                ; LOCAL STORAGE CAN BE ADDRESSED:
                ;     [BP-8], [BP-10], [BP-12]
                ; END OF PROCEDURE BODY
; PROCEDURE EPILOG
                ADD         SP, 6           ; DE-ALLOCATE LOCAL STORAGE
                POPF                        ; RESTORE CALLER'S
                POP         BX              ;   REGISTERS
                POP         CX              ;   AND
                POP         BP              ;   FLAGS
                ; END OF EPILOG
; PROCEDURE RETURN
                RET         4               ; DISCARD 2 PARAMETERS

EXAMPLE         ENDP                        ; END OF PROCEDURE "EXAMPLE"

PROC_SEG        ENDS
```

Figure 2-71. Procedure Example 1

Mnemonics © Intel, 1978

```
CALLER_SEG      SEGMENT
; GIVE ASSEMBLER SEGMENT/REGISTER CORRESPONDENCE
ASSUME          CS:CALLER_SEG,
&               DS:DATA_SEG,
&               SS:STACK_SEG,
&               ES:NOTHING              ; NO EXTRA SEGMENT IN THIS PROGRAM

; INITIALIZE SEGMENT REGISTERS
START:          MOV     AX,DATA_SEG
                MOV     DS,AX
                MOV     AX,STACK_SEG
                MOV     SS,AX
                MOV     SP,OFFSET STACK_TOP  ; POINT SP TO TOS

; ASSUME ARRAY_1 IS INITIALIZED
;
; CALL "EXAMPLE", PASSING ARRAY_1, THAT IS, THE NUMBER OF ELEMENTS
;   IN THE ARRAY, AND THE LOCATION OF THE FIRST ELEMENT.
                MOV     AX,SIZE ARRAY_1
                PUSH    AX
                MOV     AX,OFFSET ARRAY_1
                PUSH    AX
                CALL    EXAMPLE

; ASSUME ARRAY_2 IS INITIALIZED
;
; CALL "EXAMPLE" AGAIN WITH DIFFERENT SIZE ARRAY.
                MOV     AX,SIZE ARRAY_2
                PUSH    AX
                MOV     AX,OFFSET ARRAY_2
                PUSH    AX
                CALL    EXAMPLE
CALLER_SEG              ENDS

                END     START
```

Figure 2-71. Procedure Example 1 (Cont'd.)

Figure 2-72 shows the stack before the caller pushes the parameters onto it. Figure 2-73 shows the stack as the procedure receives it after the CALL has been executed.

EXAMPLE is divided into four sections. The "prolog" sets up register BP so it can be used to address data on the stack (recall that specifying BP as a base register in an instruction automatically refers to the stack segment unless a segment override prefix is coded). The next step in the prolog is to save the "state of the machine" as it existed when the procedure was activated. This is done by pushing any registers used by the procedure (only CX and BP in this case) onto the stack. If the procedure changes the flags, and the caller expects the flags to be unchanged following execution of the procedure, they also may be saved on the stack. The last instruction in the prolog allocates three words on the stack for the procedure to use as local temporary storage. Figure 2-74 shows the stack at the end of the prolog. Note that PL/M-86 procedures assume that all registers except SP and BP can be used without saving and restoring.

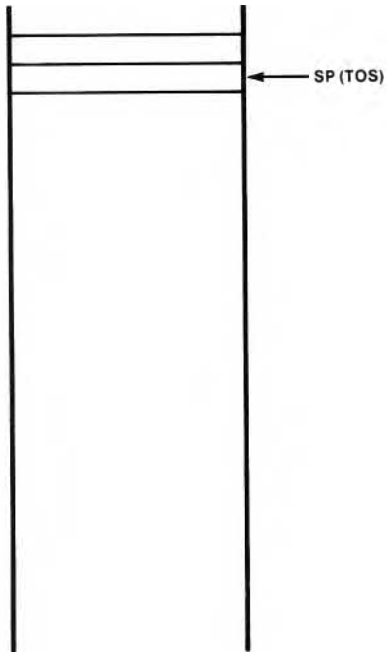


Figure 2-72. Stack Before Pushing Parameters

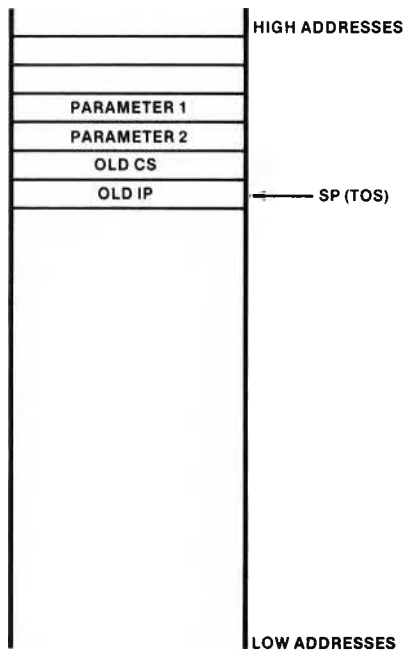


Figure 2-73. Stack at Procedure Entry

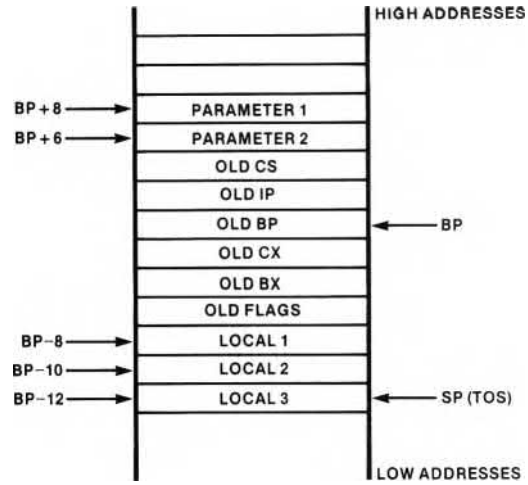


Figure 2-74. Stack Following Procedure Prolog

The procedure "body" does the actual processing (none in the example). The parameters on the stack are addressed relative to BP. Note that if EXAMPLE were a NEAR procedure, CS would not be on the stack and the parameters would be two bytes "closer" to BP. BP also is used to address the local variables on the stack. Local constants are best stored in a data or extra segment.

The procedure "epilog" reverses the activities of the prolog, leaving the stack as it was when the procedure was entered (see figure 2-75).

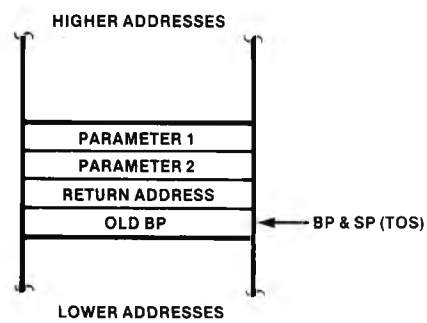


Figure 2-75. Stack Following Procedure Epilog

The procedure "return" restores CS and IP from the stack and discards the parameters. As figure 2-76 shows, when the calling program is resumed, the stack is in the same state as it was before any parameters were pushed onto it.

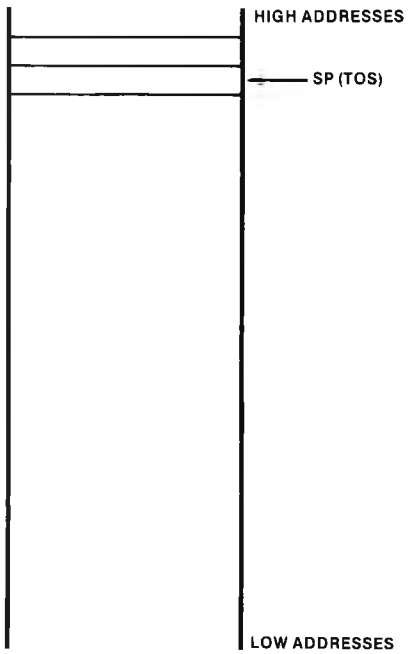


Figure 2-76. Stack Following Procedure Return

Figure 2-77 shows a simple procedure that uses an ASM-86 structure to address the stack. Register BP is pointed to the base of the structure, which is the top of the stack since the stack grows toward lower addresses (see figure 2-78). Any structure element can then be addressed by specifying BP as a base register:

```
[BP].structure_element.
```

Figure 2-79 shows a different approach to using an ASM-86 structure to define the stack layout. As shown in figure 2-80, register BP is pointed at the middle of the structure (at OLD_BP) rather than at the base of the structure. Parameters and the return address are thus located at positive displacements (high addresses) from BP, while local variables are at negative displacements (lower addresses) from BP. This means that the local variables will be "closer" to the beginning of the stack segment and increases the likelihood that the assembler will be able to produce shorter instructions to access these variables, i.e., their offsets from SS may be 255 bytes or less and can be expressed as a 1-byte value rather than a 2-byte value. Exit from the subroutine also is slightly faster because a MOV instruction can be used to deallocate the local storage instead of an ADD (compare figure 2-71).

It is possible for a procedure to be activated a second time before it has returned from its first activation. For example, procedure A may call procedure B, and an interrupt may occur while procedure B is executing. If the interrupt service procedure calls B, then procedure B is *reentered* and must be written to handle this situation correctly, i.e., the procedure must be made reentrant.

In PL/M-86 this can be done by simply writing:

```
B: PROCEDURE (PARM1, PARM2) REENTRANT;
```

An ASM-86 procedure will be reentrant if it uses the stack for storing all local variables. When the procedure is reentered, a new "generation" of variables will be allocated on the stack. The stack will grow, but the sets of variables (and the parameters and return addresses as well) will automatically be kept straight. The stack must be large enough to accommodate the maximum "depth" of procedure activation that can occur under actual running conditions. In addition, any procedure called by a reentrant procedure must itself be reentrant.

A related situation that also requires reentrant procedures is recursion. The following are examples of recursion:

- A calls A (direct recursion),
- A calls B, B calls A (indirect recursion),
- A calls B, B calls C, C calls A (indirect recursion).

Mnemonics © Intel, 1978

```
CODE            SEGMENT
                ASSUME CS:CODE
MAX             PROC
; THIS PROCEDURE IS CALLED BY THE FOLLOWING
;       SEQUENCE:
;               PUSH PARM1
;               PUSH PARM2
;               CALL MAX
; IT RETURNS THE MAXIMUM OF THE TWO WORD
;    PARAMETERS IN AX.

; DEFINE THE STACK LAYOUT AS A STRUCTURE.
STACK_LAYOUT STRUC
OLD_BP          DW ?        ; SAVED BP VALUE—BASE OF STRUCTURE
RETURN_ADDR     DW ?        ; RETURN ADDRESS
PARM_2          DW ?        ; SECOND PARAMETER
PARM_1          DW ?        ; FIRST PARAMETER
STACK_LAYOUT ENDS

; PROLOG
                PUSH        BP                  ; SAVE IN OLD_BP
                MOV         BP, SP              ; POINT TO OLD_BP
; BODY
                MOV         AX, [BP].PARM_1     ; IF FIRST
                CMP         AX, [BP].PARM_2     ; > SECOND
                JG          FIRST_IS_MAX        ; THEN RETURN FIRST
                MOV         AX, [BP].PARM_2     ; ELSE RETURN SECOND
; EPILOG
FIRST_IS_MAX:   POP         BP                  ; RESTORE BP (& SP)
; RETURN
                RET         4                   ; DISCARD PARAMETERS
MAX             ENDP

CODE            ENDS
                END
```

Figure 2-77. Procedure Example 2

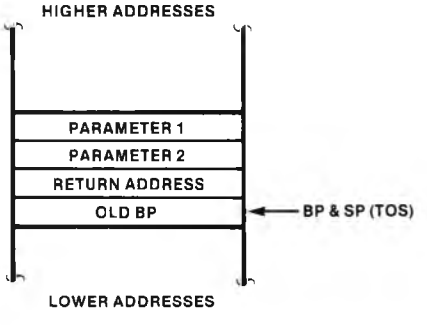


Figure 2-78. Procedure Example 2 Stack Layout

Jumps and Calls

The 8086/8088 instruction set contains many different types of JMP and CALL instructions (e.g., direct, indirect through register, indirect through memory, etc.). These varying types of transfer provide efficient use of space and execution time in different programming situations. Figure 2-81 illustrates typical use of the different forms of these instructions. Note that the ASM-86 assembler uses the terms "NEAR" and "FAR" to denote intrasegment and intersegment transfers, respectively.

Mnemonics © Intel, 1978

```
EXTRA           SEGMENT
; CONTAINS STRUCTURE TEMPLATE THAT "NEARPROC"
;     USES TO ADDRESS AN ARRAY PASSED BY ADDRESS.
DUMMY           STRUC
PARM_ARRAY      DB          256 DUP ?
DUMMY           ENDS
EXTRA           ENDS

CODE            SEGMENT
                ASSUME CS:CODE,ES:EXTRA
NEARPROC        PROC
; LAY OUT THE STACK (THE DYNAMIC STORAGE AREA OR DSA).
DSASTRUC        STRUC
I               DW          ?                   ; LOCAL VARIABLES FIRST
LOC_ARRAY       DW          10 DUP (?)          ;
OLD_BP          DW          ?                   ; ORIGINAL BP VALUE
RETADDR         DW          ?                   ; RETURN ADDRESS
POINTER         DD          ?                   ; 2ND PARM—POINTER TO "PARM_ARRAY"
COUNT           DB          ?                   ; 1ST PARM—A BYTE OCCUPIES
                DB          ?                   ;     A WORD ON THE STACK
DSASTRUC        ENDS

; USE AN EQU TO DEFINE THE BASE ADDRESS OF THE
;     DSA. CANNOT SIMPLY USE BP BECAUSE IT WILL
;     BE POINTING TO "OLD_BP" IN THE MIDDLE OF
;     THE DSA.
DSA             EQU         [BP – OFFSET OLD_BP]

; PROCEDURE ENTRY
                PUSH        BP                  ; SAVE BP
                MOV         BP, SP              ; POINT BP AT OLD_BP
                SUB         SP, OFFSET OLD_BP ; ALLOCATE LOC_ARRAY & I

; PROCEDURE BODY
                ; ACCESS LOCAL VARIABLE I
                MOV         AX,DSA.I

                ; ACCESS LOCAL ARRAY (3) I.E., 4TH ELEMENT
                MOV         SI,6                ; WORD ARRAY-INDEX IS 3*2
                MOV         AX,DSA.LOC_ARRAY [SI]

                ; LOAD POINTER TO ARRAY PASSED BY ADDRESS
                LES         BX,DSA.POINTER

                ; ES:BX NOW POINTS TO PARM_ARRAY (0)
                ; ACCESS SI'TH ELEMENT OF PARM_ARRAY
                MOV         AL,ES:[BX].PARM_ARRAY [SI]

                ; ACCESS THE BYTE PARAMETER
                MOV         AL,DSA.COUNT
```

Figure 2-79. Procedure Example 3

```
; PROCEDURE EXIT
                MOV     SP,BP           ; DE-ALLOCATE LOCALS
                POP     BP              ; RESTORE BP
                ; STACK NOW AS RECEIVED FROM CALLER
                RET     6               ; DISCARD PARAMETERS

NEARPROC        ENDP
CODE            ENDS
                END
```

Figure 2-79. Procedure Example 3 (Cont'd.)

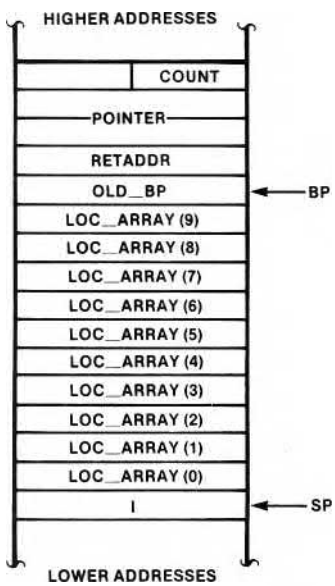


Figure 2-80. Procedure Example 3 Stack Layout

The procedure in figure 2-81 illustrates how a PL/M-86 DO CASE construction may be implemented in ASM-86. It also shows:

- an indirect CALL through memory to a procedure located in another segment,
- a direct JMP to a label in another segment,
- an indirect JMP though memory to a label in the same segment,
- an indirect JMP through a register to a label in the same segment,
- a direct CALL to a procedure in another segment,
- a direct CALL to a procedure in the same segment,
- direct JMPs to labels in the same segment, within −128 to +127 bytes ("SHORT") and farther than −128 to +127 bytes ("NEAR").

Mnemonics © Intel, 1978

```
DATA            SEGMENT
; DEFINE THE CASE TABLE (JUMP TABLE) USED BY PROCEDURE
;     "DO_CASE." THE OFFSET OF EACH LABEL WILL
;     BE PLACED IN THE TABLE BY THE ASSEMBLER.
CASE_TABLE      DW          ACTION0, ACTION1, ACTION2,
&                           ACTION3, ACTION4, ACTION5
DATA            ENDS

; DEFINE TWO EXTERNAL (NOT PRESENT IN THIS
;     ASSEMBLY BUT SUPPLIED BY R & L FACILITY)
;     PROCEDURES. ONE IS IN THIS CODE SEGMENT
;     (NEAR) AND ONE IS IN ANOTHER SEGMENT (FAR).
                EXTRN       NEAR_PROC: NEAR, FAR_PROC: FAR

; DEFINE AN EXTERNAL LABEL (JUMP TARGET) THAT
;     IS IN ANOTHER SEGMENT.
                EXTRN       ERR_EXIT: FAR

CODE            SEGMENT
                ASSUME      CS: CODE, DS: DATA
; ASSUME DS HAS BEEN SET UP
;     BY CALLER TO POINT TO "DATA" SEGMENT.

DO_CASE         PROC        NEAR
; THIS EXAMPLE PROCEDURE RECEIVES TWO
;   PARAMETERS ON THE STACK. THE FIRST
;   PARAMETER IS THE "CASE NUMBER" OF
;   A ROUTINE TO BE EXECUTED (0-5). THE SECOND
;   PARAMETER IS A POINTER TO AN ERROR
;   PROCEDURE THAT IS EXECUTED IF AN INVALID
;   CASE NUMBER (>5) IS RECEIVED.

; LAY OUT THE STACK.
STACK_LAYOUT STRUC
OLD_BP          DW      ?
RETADDR         DW      ?
ERR_PROC_ADDR   DD      ?
CASE_NO         DB      ?
                DB      ?
STACK_LAYOUT ENDS

; SET UP PARAMETER ADDRESSING
                PUSH        BP
                MOV         BP, SP

; CODE TO SAVE CALLER'S REGISTERS COULD GO HERE.

; CHECK THE CASE NUMBER
                MOV         BH, 0
                MOV         BL, [BP].CASE_NO
                CMP         BX, LENGTH CASE_TABLE
                JLE         OK              ; ALL CONDITIONAL JUMPS
                                            ; ARE SHORT DIRECT
```

Figure 2-81. JMP and CALL Examples

```
; CALL THE ERROR ROUTINE WITH A FAR
;    INDIRECT CALL. A FAR INDIRECT CALL
;    IS INDICATED SINCE THE OPERAND HAS
;    TYPE "DOUBLEWORD."
                    CALL        [BP].ERR_PROC_ADDR

; JUMP DIRECTLY TO A LABEL IN ANOTHER SEGMENT.
;    A FAR DIRECT JUMP IS INDICATED SINCE
;    THE OPERAND HAS TYPE "FAR."
                    JMP         ERR_EXIT

OK:
; MULTIPLY CASE NUMBER BY 2 TO GET OFFSET
;     INTO CASE_TABLE (EACH ENTRY IS 2 BYTES).
                    SHL         BX, 1
; NEAR INDIRECT JUMP THROUGH SELECTED
;    ELEMENT OF CASE_TABLE. A NEAR
;    INDIRECT JUMP IS INDICATED SINCE THE
;    OPERAND HAS TYPE "WORD."
                    JMP         CASE_TABLE [BX]

ACTION0:            ; EXECUTED IF CASE_NO = 0
     ; CODE TO PROCESS THE ZERO CASE GOES HERE.
     ; FOR ILLUSTRATION PURPOSES, USE A
     ;    NEAR INDIRECT JUMP THROUGH A
     ;    REGISTER TO BRANCH TO THE POINT
     ;    WHERE ALL CASES CONVERGE.
     ;    A DIRECT JUMP (JMP ENDCASE) IS
     ;    ACTUALLY MORE APPROPRIATE HERE.
                    MOV         AX, OFFSET ENDCASE
                    JMP         AX

ACTION1:            ; EXECUTED IF CASE_NO = 1
     ; CALL A FAR EXTERNAL PROCEDURE. A FAR
     ;    DIRECT CALL IS INDICATED SINCE OPERAND
     ;    HAS TYPE "FAR."
                    CALL        FAR_PROC
     ; CALL A NEAR EXTERNAL PROCEDURE.
                    CALL        NEAR_PROC
     ; BRANCH TO CONVERGENCE POINT USING NEAR
     ;    DIRECT JUMP. NOTE THAT "ENDCASE"
     ;    IS MORE THAN 127 BYTES AWAY
     ;    SO A NEAR DIRECT JUMP WILL BE USED.
                    JMP         ENDCASE

ACTION2:            ; EXECUTED IF CASE_NO = 2
     ; CODE GOES HERE
                    JMP         ENDCASE     ; NEAR DIRECT JUMP
```

Figure 2-81. JMP and CALL Examples (Cont'd.)

Mnemonics © Intel, 1978

```
ACTION3:        ; EXECUTED IF CASE_NO = 3
    ; CODE GOES HERE
                JMP         ENDCASE     ; NEAR DIRECT JMP

; ARTIFICIALLY FORCE "ENDCASE" FURTHER AWAY
;   SO THAT ABOVE JUMPS CANNOT BE "SHORT."
                ORG         500

ACTION4:        ; EXECUTED IF CASE_NO = 4
    ; CODE GOES HERE
                JMP         ENDCASE     ; NEAR DIRECT JUMP

ACTION5:        ; EXECUTED IF CASE_NO = 5
    ; CODE GOES HERE.
    ; BRANCH TO CONVERGENCE POINT USING
    ;   SHORT DIRECT JUMP SINCE TARGET IS
    ;   WITHIN 127 BYTES. MACHINE INSTRUCTION
    ;   HAS 1-BYTE DISPLACEMENT RATHER THAN
    ;   2-BYTE DISPLACEMENT REQUIRED FOR
    ;   NEAR DIRECT JUMPS. "SHORT" IS
    ;   WRITTEN BECAUSE "ENDCASE" IS A FORWARD
    ;   REFERENCE, WHICH ASSEMBLER ASSUMES IS
    ;   "NEAR." IF "ENDCASE" APPEARED PRIOR
    ;   TO THE JUMP, THE ASSEMBLER WOULD
    ;   AUTOMATICALLY DETERMINE IF IT WERE REACHABLE
    ;   WITH A SHORT JUMP.
                JMP         SHORT ENDCASE

ENDCASE:        ; ALL CASES CONVERGE HERE.

    ; POP CALLER'S REGISTERS HERE.
    ; RESTORE BP & SP, DISCARD PARAMETERS
    ;   AND RETURN TO CALLER.
                MOV         SP, BP
                POP         BP
                RET         6

DO_CASE         ENDP
CODE            ENDS
                END         ; OF ASSEMBLY
```

Figure 2-81. JMP and CALL Examples (Cont'd.)

Records

Figure 2-82 shows how the ASM-86 RECORD facility may be used to manipulate bit data. The example shows how to:

- right-justify a bit field,
- test for a value,
- assign a constant known at assembly time,
- assign a variable,
- set or clear a bit field.

```
DATA            SEGMENT
; DEFINE A WORD ARRAY
XREF            DW 3000 DUP (?)
; EACH ELEMENT OF XREF CONSISTS OF 3 FIELDS:
;       A 2-BIT TYPE CODE,
;       A 1-BIT FLAG,
;       A 13-BIT NUMBER.
; DEFINE A RECORD TO LAY OUT THIS ORGANIZATION.
LINE_REC        RECORD  LINE_TYPE: 2,
&                       VISIBLE: 1,
&                       LINE_NUM: 13
DATA            ENDS

CODE            SEGMENT
                ASSUME CS: CODE,  DS: DATA
; ASSUME SEGMENT REGISTERS ARE SET UP PROPERLY
;     AND THAT SI INDEXES AN ELEMENT OF XREF.

; A RECORD FIELD-NAME USED BY ITSELF RETURNS
;   THE SHIFT COUNT REQUIRED TO RIGHT-JUSTIFY
;   THE FIELD. ISOLATE "LINE_TYPE" IN THIS
;   MANNER.
                MOV     AL, XREF [SI]
                MOV     CL, LINE_TYPE
                SHR     AX, CL

; THE "MASK" OPERATOR APPLIED TO A RECORD
;    FIELD-NAME RETURNS THE BIT MASK
;    REQUIRED TO ISOLATE THE FIELD WITHIN
;    THE RECORD. CLEAR ALL BITS EXCEPT
;    "LINE_NUM."
                MOV     DX, XREF[SI]
                AND     DX, MASK LINE_NUM

; DETERMINE THE VALUE OF THE "VISIBLE" FIELD
                TEST    XREF[SI], MASK VISIBLE
                JZ      NOT_VISIBLE
; NO JUMP IF VISIBLE = 1
NOT_VISIBLE:    ; JUMP HERE IF VISIBLE = 0

; ASSIGN A CONSTANT KNOWN AT ASSEMBLY-TIME
;     TO A FIELD, BY FIRST CLEARING THE BITS
;     AND THEN OR'ING IN THE VALUE. IN
;     THIS CASE "LINE_TYPE" IS SET TO 2 (10B).
                AND     XREF[SI], NOT MASK LINE_TYPE
                OR      XREF[SI],2 SHL LINE_TYPE
; THE ASSEMBLER DOES THE MASKING AND SHIFTING.
; THE RESULT IS THE SAME AS:
                AND     XREF[SI], 3FFFH
                OR      XREF[SI], 8000H
;    BUT IS MORE READABLE AND LESS SUBJECT
;    TO CLERICAL ERROR.
```

Figure 2-82. RECORD Example

Mnemonics © Intel, 1978

```
                MOV     CL, LINE_TYPE   ; SHIFT COUNT
                SHL     AX, CL   ; SHIFT TO "LINE UP" BITS
                AND     XREF[SI], NOT MASK LINE_TYPE   ; CLEAR BITS
                OR      XREF[SI], AX   ; OR IN NEW VALUE

; NO SHIFT IS REQUIRED TO ASSIGN TO THE
;     RIGHT-MOST FIELD. ASSUMING AX CONTAINS
;     A VALID NUMBER (HIGH 3 BITS ARE 0),
;     ASSIGN AX TO "LINE_NUM."
                AND     XREF[SI], NOT MASK LINE_NUM
                OR      XREF[SI], AX

; A FIELD MAY BE SET OR CLEARED WITH
;   ONE INSTRUCTION. CLEAR THE "VISIBLE"
;   FLAG AND THEN SET IT.
                AND     XREF[SI], NOT MASK VISIBLE
                OR      XREF[SI], MASK VISIBLE

CODE            ENDS
                END     ; OF ASSEMBLY
```

Figure 2-82. RECORD Example (Cont'd.)

The following considerations apply to position-independent code sequences:

- A label that is referenced by a direct FAR (intersegment) transfer is not moveable.
- A label that is referenced by an indirect transfer (either NEAR or FAR) is moveable so long as the register or memory pointer to the label contains the label's current address.
- A label that is referenced by a SHORT (e.g., conditional jump) or a direct NEAR (intrasegment) transfer is moveable so long as the referencing instruction is moved with the label as a unit. These transfers are self-relative; that is they require only that the label maintain the same distance from the referencing instruction, and actual addresses are immaterial.
- Data is segment-independent, but not offset-independent. That is, a data item may be moved to a different segment, but it must maintain the same offset from the beginning of the segment. Placing constants in a unit of code also effectively makes the code offset-dependent, and therefore is not recommended.
- A procedure should not be moved while it is active or while any procedure it has called is active.
- A section of code that has been interrupted should not be moved.

The segment that is receiving a section of code must have "room" for the code. If the MOVS (or MOVSB or MOVSW) instruction attempts to auto-increment DI past 64k, it wraps around to 0 and causes the beginning of the segment to be overwritten. If a segment override is needed for the source operand, code similar to the following can be used to properly resume the instruction if it is interrupted:

```
RESUME:  REP   MOVS      DESTINATION, ES:SOURCE
  ;IF CX NOT = 0 THEN INTERRUPT HAS OCCURRED
         AND   CX,CX     ; CX=0?
         JNZ   RESUME    ;NO, FINISH EXECUTION
  ;CONTROL COMES HERE WHEN STRING HAS BEEN MOVED.
```

If the MOVS is interrupted, the CPU "remembers" the segment override, but "forgets" the presence of the REP prefix when execution resumes. Testing CX indicates whether the instruction is completed or not. Jumping back to the instruction resumes it where it left off. Note that a segment override cannot be specified with MOVSB or MOVSW.

Dynamic Code Relocation

Figure 2-83 illustrates one approach to moving programs in memory at execution time. A "supervisor" program (which is not moved) keeps a pointer variable that contains the current location (offset and segment base) of a position-independent procedure. The supervisor always calls the procedure through this pointer. The supervisor also has access to the procedure's length in bytes. The procedure is moved with the MOVSB instruction. After the procedure is moved, its pointer is updated with the new location. The ASM-86 WORD PTR operator is written to inform the assembler that one word of the doubleword pointer is being updated at a time.

```
MAIN_DATA       SEGMENT
; SET UP POINTERS TO POSITION-INDEPENDENT PROCEDURE
;    AND FREE SPACE.
PIP_PTR         DD          EXAMPLE
FREE_PTR        DD          TARGET_SEG
; SET UP SIZE OF PROCEDURE IN BYTES
PIP_SIZE        DW          EXAMPLE_LEN
MAIN_DATA       ENDS

STACK           SEGMENT
                DW          20 DUP (?)              ; 20 WORDS FOR STACK

STACK_TOP       LABEL       WORD                    ; TOS BEGINS HERE
STACK           ENDS

SOURCE_SEG      SEGMENT
; THE POSITION-INDEPENDENT PROCEDURE IS INITIALLY IN THIS SEGMENT.
; OTHER CODE MAY PRECEDE IT, I.E., ITS OFFSET NEED NOT BE ZERO.
ASSUME          CS:SOURCE_SEG
EXAMPLE         PROC        FAR
   ; THIS PROCEDURE READS AN 8-BIT PORT UNTIL
   ; BIT 3 OF THE VALUE READ IS FOUND SET. IT
   ; THEN READS ANOTHER PORT. IF THE VALUE READ
   ; IS GREATER THAN 10H IT WRITES THE VALUE TO
   ; A THIRD PORT AND RETURNS; OTHERWISE IT STARTS
   ; OVER.
STATUS_PORT     EQU         0D0H
PORT_READY      EQU         008H
INPUT_PORT      EQU         0D2H
THRESHOLD       EQU         010H
OUTPUT_PORT     EQU         0D4H
CHECK_AGAIN:    IN          AL,STATUS_PORT          ; GET STATUS
                TEST        AL,PORT_READY           ; DATA READY?
                JNE         CHECK_AGAIN             ; NO, TRY AGAIN
                IN          AL,INPUT_PORT           ; YES, GET DATA
                CMP         AL,THRESHOLD            ; > 10H?
                JLE         CHECK_AGAIN             ; NO, TRY AGAIN
                OUT         OUTPUT_PORT,AL          ; YES, WRITE IT
```

Figure 2-83. Dynamic Code Relocation Example

Mnemonics © Intel, 1978

```
                RET         ; RETURN TO CALLER
; GET PROCEDURE LENGTH
EXAMPLE_LEN     EQU         (OFFSET THIS BYTE)—(OFFSET CHECK_AGAIN)
                ENDP        EXAMPLE   ENDP
SOURCE_SEG      ENDS

TARGET_SEG      SEGMENT
; THE POSITION-INDEPENDENT PROCEDURE
;    IS MOVED TO THIS SEGMENT, WHICH IS
;    INITIALLY "EMPTY."
; IN TYPICAL SYSTEMS, A "FREE SPACE MANAGER" WOULD
;   MAINTAIN A POOL OF AVAILABLE MEMORY SPACE
; FOR ILLUSTRATION PURPOSES, ALLOCATE ENOUGH
;    SPACE TO HOLD IT
                DB          EXAMPLE_LEN DUP (?)

TARGET_SEG      ENDS

MAIN_CODE       SEGMENT
; THIS ROUTINE CALLS THE EXAMPLE PROCEDURE
; AT ITS INITIAL LOCATION, MOVES IT, AND
; CALLS IT AGAIN AT THE NEW LOCATION.

ASSUME          CS:MAIN_CODE,SS:STACK,
&               DS:MAIN_DATA,ES:NOTHING

; INITIALIZE SEGMENT REGISTERS & STACK POINTER.
START:          MOV         AX,MAIN_DATA
                MOV         DS,AX
                MOV         AX,STACK
                MOV         SS,AX
                MOV         SP,OFFSET STACK_TOP

; CALL EXAMPLE AT INITIAL LOCATION.
                CALL        PIP_PTR

; SET UP CX WITH COUNT OF BYTES TO MOV
                MOV         CX,PIP_SIZE
; SAVE DS, SET UP DS/SI AND ES/DI TO
;    POINT TO THE SOURCE AND DESTINATION
;    ADDRESSES.
                PUSH        DS
                LES         DI,FREE_PTR
                LDS         SI,PIP_PTR
; MOVE THE PROCEDURE.
                CLD                             ; AUTO INCREMENT
                REP MOVSB

; RESTORE OLD ADDRESSABILITY.
                MOV         AX,DS               ; HOLD TEMPORARILY
                POP         DS
; UPDATE POINTER TO POSITION-INDEPENDENT PROCEDURE
                MOV         WORD PTR PIP_PTR+2,ES
                SUB         DI,PIP_SIZE         ; PRODUCES OFFSET
                MOV         WORD PTR PIP_PTR,DI
```

Figure 2-83. Dynamic Code Relocation Example (Cont'd.)

```
; UPDATE POINTER TO FREE SPACE
                MOV     WORD PTR FREE_PTR+2,AX
                SUB     SI,PIP_SIZE         ; PRODUCES OFFSET
                MOV     WORD PTR FREE_PTR,SI

; CALL POSITION-INDEPENDENT PROCEDURE AT
;    NEW LOCATION AND STOP
                CALL    PIP_PTR
MAIN_CODE       ENDS
                END     START
```

Figure 2-83. Dynamic Code Relocation Example (Cont'd.)

Memory-Mapped I/O

Figure 2-84 shows how memory-mapped I/O can be used to address a group of communication lines as an "array." In the example, indexed addressing is used to poll the array of status ports, one port at a time. Any of the other 8086/8088 memory addressing modes may be used in conjunction with memory-mapped I/O devices as well.

In figure 2-85 a MOVS instruction is used to perform a high-speed transfer to a memory-mapped line printer. Using this technique requires the hardware to be set up as follows. Since the MOVS instruction transfers characters to successive memory addresses, the decoding logic must select the line printer if any of these locations is written. One way of accomplishing this is to have the chip select logic decode only the upper 12 lines of the address bus (A19-A8), ignoring the contents of the lower eight lines (A7-A0). When data is written to any address in this 256-byte block, the upper 12 lines will not change, so the printer will be selected.

If an 8086 is being used with an 8-bit printer, the 8086's 16-bit data bus must be mapped into 8-bits by external hardware. Using an 8088 provides a more direct interface.

```
COM_LINES       SEGMENT   AT 800H
; THE FOLLOWING IS A MEMORY MAPPED "ARRAY"
;     OF EIGHT 8-BIT COMMUNICATIONS CONTROLLERS
;     (E.G., 8251 USARTS). PORTS HAVE ALL-ODD
;     OR ALL-EVEN ADDRESSES (EVERY OTHER BYTE
;     IS SKIPPED) FOR 8086-COMPATIBILITY.

COM_DATA        DB    ?
                DB    ?                 ; SKIP THIS ADDRESS
COM_STATUS      DB    ?
                DB    ?                 ; SKIP THIS ADDRESS
                DB    28    DUP (?)     ; REST OF "ARRAY"
COM_LINES       ENDS

CODE            SEGMENT
; ASSUME STACK IS SET UP, AS ARE SEGMENT
;     REGISTERS (DS POINTING TO COM_LINES).
;     FOLLOWING CODE POLLS THE LINES.

CHAR_RDY        EQU       00000010B     ; CHARACTER PRESENT
START_POLL:     MOV       CX, 8         ; POLL 8 LINES ZERO
                SUB       SI, SI        ; ARRAY INDEX
```

Figure 2-84. Memory Mapped I/O "Array"

Mnemonics © Intel, 1978

```
                JE          READ__CHAR ; READ IF PRESENT
                ADD         SI, 4      ; ELSE BUMP TO NEXT LINE
                LOOP        POLL__NEXT ; CONTINUE POLLING UNTIL
                                       ;     ALL 8 HAVE BEEN CHECKED
                JMP         START__POLL; START OVER

READ__CHAR:     MOV         AL,COM__DATA [SI]   ;GET THE DATA
; ETC.
CODE            ENDS
                END
```

Figure 2-84. Memory Mapped I/O "Array" (Cont'd.)

```
PRINTER         SEGMENT
; THIS SEGMENT CONTAINS A "STRING" THAT
;    IS ACTUALLY A MEMORY-MAPPED LINE PRINTER.
;    THE SEGMENT (PRINTER) MUST BE ASSIGNED (LOCATED)
;    TO A BLOCK OF THE ADDRESS SPACE SUCH
;    THAT WRITING TO ANY ADDRESS IN THE
;    BLOCK SELECTS THE PRINTER.

PRINT__SELECT   DB 133      DUP (?)          ; "STRING" REPRESENTING PRINTER
                DB 123      DUP (?)          ; REST OF 256-BYTE BLOCK
PRINTER         ENDS

DATA            SEGMENT
PRINT__BUF      DB 133      DUP (?)          ; LINE TO BE PRINTED
PRINT__COUNT    DB 1        ?                ; LINE LENGTH
; OTHER PROGRAM DATA
DATA            ENDS

CODE            SEGMENT
; ASSUME STACK AND SEGMENT REGISTERS HAVE
;     BEEN SET UP (DS POINTS TO DATA SEGMENT).
;     FOLLOWING CODE TRANSFERS A LINE TO
;     THE PRINTER.

                ASSUME      ES: PRINTER
                MOV         AX, PRINTER      ; PREVENT SEGMENT OVERRIDE
                MOV         ES, AX
                SUB         DI, DI           ; CLEAR SOURCE AND
                SUB         SI, SI           ;     DESTINATION POINTERS
                MOV         CX, PRINT__COUNT
                CLD         ; AUTO-INCREMENT
    REP         MOVS        PRINT__SELECT, PRINT__BUF
                ; ETC.
CODE            ENDS
                END
```

Figure 2-85. Memory Mapped Block Transfer Example

Breakpoints

Figure 2-86 illustrates how a program may set a breakpoint. In the example, the breakpoint routine puts the processor into single-step mode, but the same general approach could be used for other purposes as well. A program passes the address where the break is to occur to a procedure that saves the byte located at that address and replaces it with an INT 3 (breakpoint) instruction. When the CPU encounters the breakpoint instruction, it calls the type 3 interrupt procedure. In the example, this procedure places the processor into single-step mode starting with the instruction where the breakpoint was placed.

```
INT_PTR_TAB   SEGMENT
; INTERRUPT POINTER TABLE-LOCATE AT 0H
TYPE_0        DD        ?                    ; NOT DEFINED IN EXAMPLE
TYPE_1        DD        SINGLE_STEP
TYPE_2        DD        ?                    ; NOT DEFINED IN EXAMPLE
TYPE_3        DD        BREAKPOINT
INT_PTR_TAB   ENDS

SAVE_SEG      SEGMENT
SAVE_INSTR    DB 1      DUP (?)              ; INSTRUCTION REPLACED
                                             ; BY BREAKPOINT
SAVE_SEG      ENDS

MAIN_CODE     SEGMENT
; ASSUME STACK AND SEGMENT REGISTERS ARE SET UP.

; ENABLE SINGLE-STEPPING WITH INSTRUCTION AT
;    LABEL "NEXT" BY PASSING SEGMENT AND
;    OFFSET OF "NEXT" TO "SET_BREAK" PROCEDURE
              PUSH      CS
              LEA       AX, CS: NEXT
              PUSH      AX
              CALL      FAR SET_BREAK
; ETC.

NEXT:         IN        AL, 0FFFH            ; BREAKPOINT SET HERE
              ; ETC.

MAIN_CODE     ENDS

BREAK         SEGMENT
SET_BREAK     PROC      FAR
; THIS PROCEDURE SAVES AN INSTRUCTION BYTE (WHOSE
;    ADDRESS IS PASSED BY THE CALLER) AND WRITES
;    AN INT 3 (BREAKPOINT) MACHINE INSTRUCTION
:    AT THE TARGET ADDRESS.

TARGET        EQU       DWORD PTR [BP + 6]
```

Figure 2-86. Breakpoint Example

Mnemonics © Intel, 1978

```
; SET UP BP FOR PARM ADDRESSING & SAVE REGISTERS
                PUSH    BP
                MOV     BP, SP
                PUSH    DS
                PUSH    ES
                PUSH    AX
                PUSH    BX
; POINT DS/BX TO THE TARGET INSTRUCTION
                LDS     BX, TARGET
; POINT ES TO THE SAVE AREA
                MOV     AX, SAVE_SEG
                MOV     ES, AX
; SWAP THE TARGET INSTRUCTION FOR INT 3 (0CCH)
                MOV     AL, 0CCH
                XCHG    AL, DS: [BX]
; SAVE THE TARGET INSTRUCTION
                MOV     ES: SAVE_INSTR, AL
; RESTORE AND RETURN
                POP     BX
                POP     AX
                POP     ES
                POP     DS
                POP     BP
                RET     4
SET_BREAK       ENDP

BREAKPOINT      PROC    FAR
; THE CPU WILL ACTIVATE THIS PROCEDURE WHEN IT
;   EXECUTES THE INT 3 INSTRUCTION SET BY THE
;   SET_BREAK PROCEDURE. THIS PROCEDURE
;   RESTORES THE SAVED INSTRUCTION BYTE TO ITS
;   ORIGINAL LOCATION AND BACKS UP THE
;   INSTRUCTION POINTER IMAGE ON THE STACK
;   SO THAT EXECUTION WILL RESUME WITH
;   THE RESTORED INSTRUCTION. IT THEN SETS
;   TF (THE TRAP FLAG) IN THE FLAG-IMAGE
;   ON THE STACK. THIS PUTS THE PROCESSOR
;   IN SINGLE-STEP MODE WHEN EXECUTION
;   RESUMES.
    FLAG_IMAGE  EQU     WORD PTR [BP+6]
    IP_IMAGE    EQU     WORD PTR [BP+2]
NEXT_INSTR      EQU     DWORD PTR [BP+2]
; SET UP BP TO ADDRESS STACK AND SAVE REGISTERS
                PUSH    BP
                MOV     BP, SP
                PUSH    DS
                PUSH    ES
                PUSH    AX
                PUSH    BX
; POINT ES AT THE SAVE AREA
                MOV     AX, SAVE_SEG
                MOV     ES, AX
; GET THE SAVED BYTE
                MOV     AL, ES: SAVE_INSTR
```

Figure 2-86. Breakpoint Example (Cont'd.)

```
; GET THE ADDRESS OF THE TARGET + 1
;    (INSTRUCTION FOLLOWING THE BREAKPOINT)
                LDS         BX, NEXT_INSTR
; BACK UP IP-IMAGE (IN BX) AND REPLACE ON STACK
                DEC         BX
                MOV         IP_IMAGE, BX

; RESTORE THE SAVED INSTRUCTION
                MOV         DS: [BX], AL
; SET TF ON STACK
                AND         FLAG_IMAGE, 0100H
; RESTORE EVERYTHING AND EXIT
                POP         BX
                POP         AX
                POP         ES
                POP         DS
                POP         BP
                IRET
BREAKPOINT      ENDP

SINGLE  STEP    PROC        FAR
; ONCE SINGLE-STEP MODE HAS BEEN ENTERED,
;    THE CPU "TRAPS" TO THIS PROCEDURE
;    AFTER EVERY INSTRUCTION THAT IS NOT IN
;    AN INTERRUPT PROCEDURE. IN THE CASE
;    OF THIS EXAMPLE, THIS PROCEDURE WILL
;    BE EXECUTED IMMEDIATELY FOLLOWING THE
;    "IN AL, 0FFFH" INSTRUCTION (WHERE THE
;    BREAKPOINT WAS SET) AND AFTER EVERY
;    SUBSEQUENT INSTRUCTION. THE PROCEDURE
;    COULD "TURN ITSELF OFF" BY CLEARING
;    TF ON THE STACK.
; SINGLE-STEP CODE GOES HERE.
; SINGLE_STEP   ENDP

BREAK           ENDS

                END         ;
```

Figure 2-86. Breakpoint Example (Cont'd.)

Interrupt Procedures

Figure 2-87 is a block diagram of a hypothetical system that is used to illustrate three different examples of interrupt handling: an external (maskable) interrupt, an external non-maskable interrupt and a software interrupt.

In this hypothetical system, an 8253 Programmable Interval Timer is used to generate a time base. One of the three timers on the 8253 is programmed to repeatedly generate interrupt requests at 50 millisecond intervals. The output from this timer is tied to one of the eight interrupt request lines of an 8259A Programmable Interrupt Controller. The 8259A, in turn, is connected to the INTR line of an 8086 or 8088.

Mnemonics © Intel, 1978

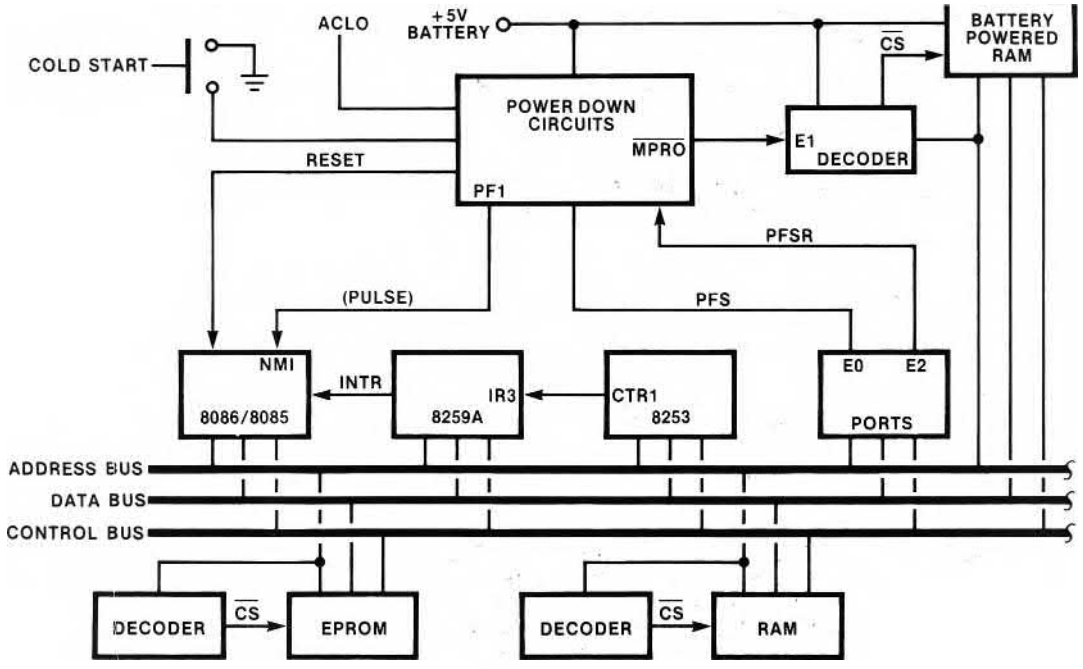


Figure 2-87. Interrupt Example Block Diagram

A power-down circuit is used in the system to illustrate one application of the 8086/8088 NMI (non-maskable interrupt) line. If the ac line voltage drops below a certain threshold, the power supply activates ACLO. The power-down circuit then sends a power-fail interrupt (PFI) pulse to the CPU's NMI input. After 5 milliseconds, the power-down circuit activates MPRO (memory protect) to disable reading from and writing to the system's battery-powered RAM. This protects the RAM from fluctuations that may occur when power is actually lost 7.5 milliseconds after the power failure is detected. The system software must save all vital information in the battery-powered RAM segment within 5 milliseconds of the activation of NMI.

When power returns, the power-down circuit activates the system RESET line. Pressing the "cold start" switch also produces a system RESET. The PFS (power fail status) line, which is connected to the low-order bit of port E0, identifies the source of the RESET. If the bit is set, the software executes a "warm start" to restore the information saved by the power-fail routine. If the PFS bit is cleared, the software executes a "cold start" from the beginning of the program. In either case, the software writes a "one" to the low-order bit of port E2. This line is connected to the power-down circuit's PFSR (power fail status reset) signal and is used to enable the battery-powered RAM segment.

A software interrupt is used to update a simple real-time clock. This procedure is written in PL/M-86, while the rest of the system is written in ASM-86 to demonstrate the interrupt handling capability of both languages. The system's main program simply initializes the system following receipt of a RESET and then waits for an interrupt. An example of this interrupt procedure is given in figure 2-88.

```
INT_POINTERS                    SEGMENT
; INTERRUPT POINTER TABLE, LOCATE AT 0H, ROM-BASED
TYPE_0          DD              ?               ; DIVIDE-ERROR NOT SUPPLIED IN EXAMPLE.
TYPE_1          DD              ?               ; SINGLE-STEP NOT SUPPLIED IN EXAMPLE.
TYPE_2          DD              POWER_FAIL      ; NON-MASKABLE INTERRUPT
TYPE_3          DD              ?               ; BREAKPOINT NOT SUPPLIED IN EXAMPLE.
TYPE_4          DD              ?               ; OVERFLOW NOT SUPPLIED IN EXAMPLE.
; SKIP RESERVED PART OF EXAMPLE
                ORG             32*4
TYPE_32         DD              ?               ; 8259A IR0 - AVAILABLE
TYPE_33         DD              ?               ; 8259A IR1 - AVAILABLE
TYPE_34         DD              ?               ; 8259A IR2 - AVAILABLE
TYPE_35         DD              TIMER_PULSE     ; 8259A IR3
TYPE_36         DD              ?               ; 8259A IR4 - AVAILABLE
TYPE_37         DD              ?               ; 8259A IR5 - AVAILABLE
TYPE_38         DD              ?               ; 8259A IR6 - AVAILABLE
TYPE_39         DD              ?               ; 8259A IR7 - AVAILABLE
;
; POINTER FOR TYPE 40 SUPPLIED BY PL/M-86 COMPILER
;
INT_POINTERS                    ENDS

BATTERY                         SEGMENT
; THIS RAM SEGMENT IS BATTERY-POWERED. IT CONTAINS VITAL DATA
;     THAT MUST BE MAINTAINED DURING POWER OUTAGES.
STACK_PTR       DW              ?               ; SP SAVE AREA
STACK_SEG       DW              ?               ; SS SAVE AREA
; SPACE FOR OTHER VARIABLES COULD BE DEFINED HERE.
BATTERY                         ENDS

DATA                            SEGMENT
; RAM SEGMENT THAT IS NOT BACKED UP BY BATTERY
N_PULSES        DB              1 DUP (0)       ; # TIMER PULSES

; ETC.
DATA                            ENDS

STACK                           SEGMENT
; LOCATED IN BATTERY-POWERED RAM
                DW              100 DUP (?)     ; THIS IS AN ARBITRARY STACKSIZE

STACK_TOP       LABEL           WORD            ; LABEL THE INITIAL TOS
STACK                           ENDS

INTERRUPT_HANDLERS      SEGMENT
; INTERRUPT PROCEDURES EXCEPT TYPE 40 (PL/M-86)

                ASSUME:         CS:INTERRUPT_HANDLERS,DS:DATA,SS:STACK,ES:BATTERY

POWER_FAIL                      PROC            ; TYPE 2 INTERRUPT
; POWER FAIL DETECT CIRCUIT ACTIVATES NMI LINE ON CPU IF POWER IS
;     ABOUT TO BE LOST. THIS PROCEDURE SAVES THE PROCESSOR STATE IN
;     RAM (ASSUMED TO BE POWERED BY AN AUXILIARY SOURCE) SO THAT IT
;     CAN BE RESTORED BY A WARM START ROUTINE IF POWER RETURNS
```

Figure 2-88. Interrupt Procedures Example

Mnemonics © Intel, 1978

```
; IP, CS, AND FLAGS ARE ALREADY ON THE STACK.
;   SAVE THE OTHER REGISTERS.
                    PUSH        AX
                    PUSH        BX
                    PUSH        CX
                    PUSH        DX
                    PUSH        SI
                    PUSH        DI
                    PUSH        BP
                    PUSH        DS
                    PUSH        ES

; CRITICAL MEMORY VARIABLES COULD ALSO BE SAVED ON THE STACK AT THIS
;    POINT. ALTERNATIVELY, THEY COULD BE DEFINED IN THE "BATTERY"
;    SEGMENT, WHERE THEY WILL AUTOMATICALLY BE PROTECTED IF MAIN POWER
;    IS LOST.

; SAVE SP AND SS IN FIXED LOCATIONS THAT ARE KNOWN BY WARM START ROUTINE.
                    MOV         AX,BATTERY
                    MOV         ES,AX
                    MOV         ES:STACK_PTR,SP
                    MOV         ES:STACK_SEG,SS
; STOP GRACEFULLY
                    HLT
POWER_FAIL                      ENDP

TIMER_PULSE                     PROC                ; TYPE 35 INTERRUPT
; THIS PROCEDURE HANDLES THE 50MS INTERRUPTS GENERATED BY THE 8253.
;    IT COUNTS THE INTERRUPTS AND ACTIVATES THE TYPE 40 INTERRUPT
;    PROCEDURE ONCE PER SECOND.
;
; DS IS ASSUMED TO BE POINTING TO THE DATA SEGMENT
;
; THE 8253 IS RUNNING FREE, AND AUTOMATICALLY LOWERS ITS INTERRUPT
;    REQUEST. IF A DEVICE REQUIRED ACKNOWLEDGEMENT, THE CODE MIGHT GO HERE.
;
; NOW PERFORM PROCESSING THAT MUST NOT BE INTERRUPTED (EXCEPT FOR NMI).
                    INC         N_PULSES
; ENABLE HIGHER-PRIORITY INTERRUPTS AND DO LESS CRITICAL PROCESSING
                    STI
                    CMP         N_PULSES,200        ; 1 SECOND PASSED?
                    JBE         DONE                ; NO, GO ON.
                    MOV         N_PULSES,0          ; YES, RESET COUNT.
                    INT         40                  ; UPDATE CLOCK
; SEND NON-SPECIFIC END-OF-INTERRUPT COMMAND TO 8259A, ENABLING EQUAL
;    OR LOWER PRIORITY INTERRUPTS.
DONE:               MOV         AL,020H             ; EOI COMMAND
                    OUT         0C0H,AL             ; 8259A PORT
                    IRET
TIMER_PULSE                     ENDP

INTERRUPT_HANDLERS              ENDS


CODE                SEGMENT
; THIS SEGMENT WOULD NORMALLY RESIDE IN ROM.

                    ASSUME      CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
INIT            PROC        NEAR
; THIS PROCEDURE IS CALLED FOR BOTH WARM AND COLD STARTS TO INITIALIZE
;     THE 8253 AND THE 8259A. THIS ROUTINE DOES NOT USE STACK, DATA, OR
;     EXTRA SEGMENTS, AS THEY ARE NOT SET PREDICTABLY DURING A WARM START.
;     INTERRUPTS ARE DISABLED BY VIRTUE OF THE SYSTEM RESET.

; INITIALIZE 8253 COUNTER 1 - OTHER COUNTERS NOT USED.
; CLK INPUT TO COUNTER IS ASSUMED TO BE 1.23 MHZ.

LO50MS          EQU         000H            ; COUNT VALUE IS
HI50MS          EQU         0F0H            ;     61440 DECIMAL.
CONTROL         EQU         0D6H            ; CONTROL PORT ADDRESS
COUNT_1         EQU         0D2H            ; COUNTER 1 ADDRESS
MODE2           EQU         01110100B       ; MODE 2, BINARY

                MOV         DX,CONTROL      ; LOAD CONTROL BYTE
                MOV         AL,MODE2
                OUT         DX,AL
                MOV         DX,COUNT_1      ; LOAD 50MS DOWNCOUNT
                MOV         AL,LO50MS
                OUT         DX,AL
                MOV         AL,HI50MS
                OUT         DX,AL
                ; COUNTER NOW RUNNING, INTERRUPTS STILL DISABLED.

; INITIALIZE 8259A TO: SINGLE INTERRUPT CONTROLLER, EDGE-TRIGGERED,
;     INTERRUPT TYPES 32-40 (DECIMAL) TO BE SENT TO CPU FOR INTERRUPT
;     REQUESTS 0-7 RESPECTIVELY, 8086 MODE, NON-AUTOMATIC END-OF-INTERRUPT.
;     MASK OFF UNUSED INTERRUPT REQUEST LINES.

ICW1            EQU         00010011B       ; EDGE-TRIGGERED, SINGLE 8259A, ICW4 REQUIRED.
ICW2            EQU         00100000B       ; TYPE 20H, 32 - 40D
ICW4            EQU         00000001B       ; 8086 MODE, NORMAL EOI
OCW1            EQU         11110111B       ; MASK ALL BUT IR3
PORT_A          EQU         0C0H            ; ICW1 WRITTEN HERE
PORT_B          EQU         0C2H            ; OTHER ICW'S WRITTEN HERE

                MOV         DX,PORT_A       ; WRITE 1ST ICW
                MOV         AL,ICW1
                OUT         DX,AL
                MOV         DX,PORT_B       ; WRITE 2ND ICW
                MOV         AL,ICW2
                OUT         DX,AL
                MOV         AL,ICW4         ; WRITE 4TH ICW
                OUT         DX,AL
                MOV         AL,0CW1         ; MASK UNUSED IR'S
                OUT         DX,AL
; INITIALIZATION COMPLETE, INTERRUPTS STILL DISABLED
                RET
INIT            ENDP


USER_PGM:
; "REAL" CODE WOULD GO HERE. THE EXAMPLE EXECUTES AN ENDLESS LOOP
;     UNTIL AN INTERRUPT OCCURS.
                JMP         USER_PGM


; EXECUTION STARTS HERE WHEN CPU IS RESET.
POWER_FAIL_STATUS           EQU     0E0H        ; PORT ADDRESS
ENABLE_RAM                  EQU     0E2H        ; PORT ADDRESS
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

Mnemonics © Intel, 1978

```
; DETERMINE WARM OR COLD START
                IN          AL,POWER_FAIL_STATUS
                RCR         AL,1                ; ISOLATE LOW BIT
                JC          WARM_START

COLD_START:
; INITIALIZE SEGMENT REGISTERS AND STACK POINTER.
                ASSUME   CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
                ; RESET TAKES CARE OF CS AND IP.
                MOV         AX,DATA
                MOV         DS,AX
                MOV         AX,STACK
                MOV         SS,AX
                MOV         SP,OFFSET STACK_TOP

; INITIALIZE 8253 AND 8259A.
                CALL        INIT

; ENABLE INTERRUPTS
                STI

; START MAIN PROCESSING
                JMP         USER_PGM


WARM_START:
; INITIALIZE 8253 AND 8259A.
                CALL        INIT

; RESTORE SYSTEM TO STATE AT THE TIME POWER FAILED
                ; MAKE BATTERY SEGMENT ADDRESSABLE
                      MOV     AX,BATTERY
                      MOV     DX,AX
                ; VARIABLES SAVED IN THE "BATTERY" SEGMENT WOULD BE MOVED
                ;    BACK TO UNPROTECTED RAM NOW. SEGMENT REGISTERS AND
                ;    "ASSUME" DIRECTIVES WOULD HAVE TO BE WRITTEN TO GAIN
                ;    ADDRESSABILITY.

                ; RESTORE THE OLD STACK
                      MOV     SS,DS:STACK_SEG
                      MOV     SP,DS:STACK_PTR
                ; RESTORE THE OTHER REGISTERS
                      POP     ES
                      POP     DS
                      POP     BP
                      POP     DI
                      POP     SI
                      POP     DX
                      POP     CX
                      POP     BX
                      POP     AX
                ; RESUME THE ROUTINE THAT WAS EXECUTING WHEN NMI WAS ACTIVATED.
                ;    I.E., POP CS, IP, & FLAGS, EFFECTIVELY "RETURNING" FROM THE
                ;    NMI PROCEDURE.
                      IRET
CODE            ENDS

                ; TERMINATE ASSEMBLY AND MARK BEGINNING OF THE PROGRAM.
                                END         START
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

```
TYPE$40:  DO;
  DECLARE (HOUR, MIN, SEC) BYTE PUBLIC;
  UPDATE$TOD: PROCEDURE INTERRUPT 40;
    /*THE PROCESSOR ACTIVATES THIS PROCEDURE
     *TO HANDLE THE SOFTWARE INTERRUPT
     *GENERATED EVERY SECOND BY THE TYPE 35
     *EXTERNAL INTERRUPT PROCEDURE. THIS
     *PROCEDURE UPDATES A REAL-TIME CLOCK.
     *IT DOES NOT PRETEND TO BE "REALISTIC"
     *AS THERE IS NO WAY TO SET THE CLOCK.*/

  SEC = SEC + 1;
  IF SEC = 60 THEN DO;
    SEC = 0;
    MIN = MIN + 1;
    IF MIN = 60 THEN DO;
      MIN = 0;
      HOUR = HOUR + 1;
      IF HOUR = 24 THEN DO;
        HOUR = 0;
        END;
      END;
    END;
  END UPDATE$TOD;
END;
```

Figure 2-88. Interrupt Procedures Example (Cont'd.)

String Operations

Figure 2-89 illustrates typical use of string instructions and repeat prefixes. The XLAT instruction also is demonstrated. The first example simply moves 80 words of a string using MOVS. Then two byte strings are compared to find the alphabetically lower string, as might be done in a sort. Next a string is scanned from right to left (the index register is auto-decremented) to find the last period (".") in the string. Finally a byte string of EBCDIC characters is translated to ASCII. The translation is stopped at the end of the string or when a carriage return character is encountered, whichever occurs first. This is an example of using the string primitives in combination with other instructions to build up more complex string processing operations.

```
ALPHA          SEGMENT
; THIS IS THE DATA THE STRING INSTRUCTIONS WILL USE
OUTPUT         DW 100     DUP (?)
INPUT          DW 100     DUP (?)
NAME_1         DB 'JONES, JONA'
NAME_2         DB 'JONES, JOHN'
SENTENCE       DB 80      DUP (?)
EBCDIC_CHARS   DB 80      DUP (?)
ASCII_CHARS    DB 80      DUP (?)
CONV_TAB       DB 64      DUP(0H)            ; EBCDIC TO ASCII
```

Figure 2-89. String Examples

Mnemonics © Intel, 1978

```
        ; ASCII NULLS ARE SUBSTITUTED FOR "UNPRINTABLE" CHARS
                DB 1        20H
                DB 9        DUP (0H)
                DB 7        '¢', '.', '<', '(', '+', 0H, '&'
                DB 9        DUP (0H)
                DB 8        '!', '$', '*', ')', ';', ' ', '-', '/'
                DB 8        DUP (0H)
                DB 6        ' ', ',', '%', '_', '>', '?'
                DB 9        DUP (0H)
                DB 17       ' ', ':', '#', '@', ' ' ', '=', ' " ',
                            0H, 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'
                DB 7        DUP (0H)
                DB 9        'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r'
                DB 7        DUP (0H)
                DB 9        '≃', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'
                DB 22       DUP (0H)
                DB 10       ' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'
                DB 6        DUP (0H)
                DB 10       ' ', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R'
                DB 6        DUP (0H)
                DB 10       ' ', 0H, 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'
                DB 6        DUP (0H)
                DB 10       '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
                DB 6        DUP (0H)

ALPHA           ENDS

STACK           SEGMENT
                DW 100      DUP (?)             ; THIS IS AN ARBITRARY STACK SIZE
                                                ; FOR ILLUSTRATION ONLY.
STACK_BASE      LABEL       WORD                ; INITIAL TOS
STACK           ENDS

CODE            SEGMENT
BEGIN:          ; SET UP SEGMENT REGISTERS. NOTICE THAT
                ; ES & DS POINT TO THE SAME SEGMENT, MEANING
                ; THAT THE CURRENT EXTRA & DATA
                ; SEGMENTS FULLY OVERLAP. THIS ALLOWS
                ; ANY STRING IN "ALPHA" TO BE USED
                ; AS A SOURCE OR A DESTINATION.
                ASSUME CS: CODE, SS: STACK,
&                       DS: ALPHA, ES: ALPHA
                MOV         AX, STACK
                MOV         SS, AX
                MOV         SP, OFFSET STACK_BASE ; INITIAL TOS
                MOV         AX, ALPHA
                MOV         DS, AX
                MOV         ES, AX

; MOVE THE FIRST 80 WORDS OF "INPUT" TO
;     THE LAST 80 WORDS OF "OUTPUT".
                LEA         SI, INPUT           ; INITIALIZE
                LEA         DI, OUTPUT+20       ; INDEX REGISTERS
```

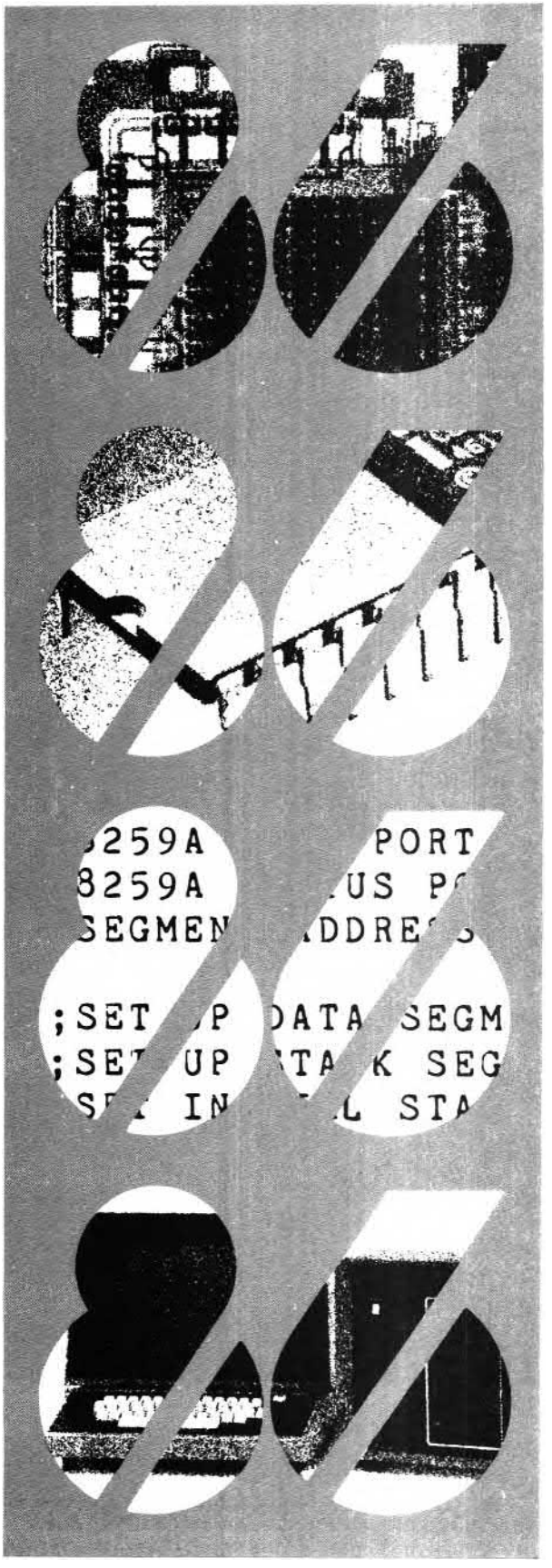
Figure 2-89. String Examples (Cont'd.)

```
                MOV     CX, 80                  ; REPETITION COUNT
                CLD                             ; AUTO-INCREMENT
        REP     MOVS    OUTPUT, INPUT

; FIND THE ALPHABETICALLY LOWER OF 2 NAMES.
                MOV     SI, OFFSET NAME__1      ; ALTERNATIVE
                MOV     DI, OFFSET NAME__2      ; TO LEA
                MOV     CX, SIZE NAME__2        ; CHAR. COUNT
                CLD                             ; AUTO-INCREMENT
        REPE    CMPS    NAME__2, NAME__1        "WHILE EQUAL"
                JB      NAME__2__LOW
NAME__1__LOW:           ; NOT IN THIS EXAMPLE
NAME__2__LOW:           ; CONTROL COMES HERE IN THIS EXAMPLE.
                        ; DI POINTS TO BYTE ('H') THAT
                        ; COMPARED UNEQUAL.

; FIND THE LAST PERIOD ('.') IN A TEXT STRING.
                MOV     DI, OFFSET SENTENCE +
&                           LENGTH SENTENCE     ; START AT END
                MOV     CX, SIZE SENTENCE
                STD                             ; AUTO-DECREMENT
                MOV     AL, '.'                 ; SEARCH ARGUMENT
       REPNE    SCAS    SENTENCE                ; "WHILE NOT ="
                JCXZ    NO__PERIOD              ; IF CX=0, NO PERIOD FOUND
PERIOD:         ; IF CONTROL COMES HERE THEN
                ;   DI POINTS TO LAST PERIOD IN SENTENCE.
NO__PERIOD:     ; ETC.

; TRANSLATE A STRING OF EBCDIC CHARACTERS
;     TO ASCII, STOPPING IF A CARRIAGE RETURN
;     (0DH ASCII) IS ENCOUNTERED.
                MOV     BX, OFFSET CONV__TAB    ; POINT TO TRANSLATE TABLE
                MOV     SI, OFFSET EBCDIC__CHARS    ; INITIALIZE
                MOV     DI, OFFSET ASCII__CHARS     ;    INDEX REGISTERS
                MOV     CX, SIZE ASCII__CHARS       ;    AND COUNTER
                CLD                             ; AUTO-INCREMENT
NEXT:           LODS    EBCDIC__CHARS           ; NEXT EBCDIC CHAR IN AL
                XLAT    CONV__TAB               ; TRANSLATE TO ASCII
                STOS    ASCII__CHARS            ; STORE FROM AL
                TEST    AL, 0DH                 ; IS IT CARRIAGE RETURN?
                LOOPNE  NEXT                    ; NO, CONTINUE WHILE CX NOT 0
                JE      CR__FOUND               ; YES, JUMP
                ; CONTROL COMES HERE IF ALL CHARACTERS
                ;     HAVE BEEN TRANSLATED BUT NO
                ;     CARRIAGE RETURN IS PRESENT.
                ; ETC.

CR__FOUND:
                ; DI-1 POINTS TO THE CARRIAGE RETURN
                ;   IN ASCII__CHARS.

CODE            ENDS
                END
```

Figure 2-89. String Examples (Cont'd.)

Mnemonics © Intel, 1978

Chapter 3
The 8089
Input/Output Processor



CHAPTER 3
THE 8089 INPUT/OUTPUT PROCESSOR

This chapter describes the 8089 Input/Output Processor (IOP). Its organization parallels Chapter 2; that is, sections generally proceed from hardware to software topics as follows:

1. Processor Overview
2. Processor Architecture
3. Memory
4. Input/Output
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Addressing Modes
9. Programming Facilities
10. Programming Guidelines and Examples

As in Chapter 2, the discussion is confined to covering the hardware in functional terms; timing, electrical characteristics and other physical interfacing data are provided in Chapter 4.

3.1 Processor Overview

The 8089 Input/Output Processor is a high-performance, general-purpose I/O system implemented on a single chip. Within the 8089 are two independent I/O channels, each of which combines attributes of a CPU with those of a very flexible DMA (direct memory access) controller. For example, channels can execute programs like CPUs; the IOP instruction set has about 50 different types of instructions specifically designed for efficient input/output processing. Each channel also can perform high-speed DMA transfers; a variety of optional operations allow the data to be manipulated (e.g., translated or searched) as it is transferred. The 8089 is contained in a 40-pin dual in-line package (figure 3-1) and operates from a single +5V power source. An integral member of the 8086 family, the IOP is directly compatible with both the 8086 and 8088 when these processors are configured in maximum mode. The IOP also may be used in any system that incorporates Intel's Multibus™ shared bus architecture, or a superset of the Multibus™ design.

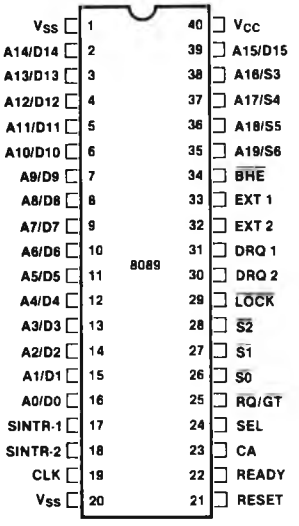


Figure 3-1. 8089 Input/Output Processor Pin Diagram

Evolution

Figure 3-2 depicts the general trend in CPU and I/O device relationships in the first three generations of microprocessors. First generation CPUs were forced to deal directly with substantial numbers of TTL components, often performing transfers at the bit level. Only a very limited number of relatively slow devices could be supported.

Single-chip interface controllers were introduced in the second generation. These devices removed the lowest level of device control from the CPU and let the CPU transfer whole bytes at once. With the introduction of DMA controllers, high-speed devices could be added to a system, and whole blocks of data could be transferred without CPU intervention. Compared to the previous generation, I/O device and DMA controllers allowed microprocessors to be applied to problems that required moderate levels of I/O, both in terms of the numbers of devices that could be supported and the transfer speeds of those devices.

required a considerable amount of attention from the CPU, and in many cases the CPU had to respond to an interrupt with every byte read or written. The CPU also had to stop while DMA transfers were performed.

The 8089 introduces the third generation of input/output processing. It continues the trend of simplifying the CPU's "view" of I/O devices by removing another level of control from the CPU. The CPU performs an I/O operation by building a message in memory that describes the function to be performed; the IOP reads the message, carries out the operation and notifies the CPU when it has finished. All I/O devices appear to the CPU as transmitting and receiving whole blocks of data; the IOP can make both byte- and word-level transfers invisible to the CPU. The IOP assumes all device controller overhead, performs both programmed and DMA transfers, and can recover from "soft" I/O errors without CPU intervention; all of these activities may be performed while the CPU is attending to other tasks.

Since the 8089 is a new concept in microprocessor components, this section surveys the basic operation of the IOP as background to the detailed descriptions provided in the rest of the chapter. This summary deliberately omits some operating details in order to provide an integrated overview of basic concepts.

CPU/IOP Communications

A CPU communicates with an IOP in two distinct modes: initialization and command. The initialization sequence is typically performed when the system is powered-up or reset. The CPU initializes the IOP by preparing a series of linked message blocks in memory. On a signal from the CPU, the IOP reads these blocks and determines from them how the data buses are configured and how access to the buses is to be controlled.

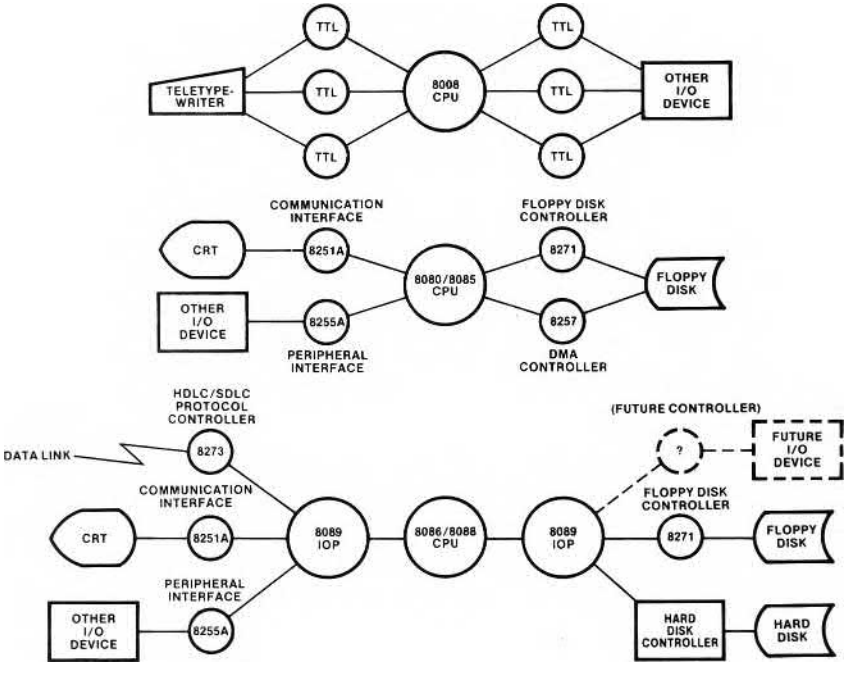


Figure 3-2. IOP Evolution

Following initialization, the CPU directs all communications to either of the IOP's two channels; indeed, during normal operation the IOP appears to be two separate devices—channel 1 and channel 2. All CPU-to-channel communications center on the channel control block (CB) illustrated in figure 3-3. The CB is located in the CPU's memory space, and its address is passed to the IOP during initialization. Half of the block is dedicated to each channel. The channel maintains the BUSY flag that indicates whether it is in the midst of an operation or is available for a new command. The CPU sets the CCW (channel command word) to indicate what kind of operation the IOP is to perform. Six different commands allow the CPU to start and stop programs, remove interrupt requests, etc.

If the CPU is dispatching a channel to run a program, it directs the channel to a parameter block (PB) and a task block (TB); these are also shown in figure 3-3. The parameter block is analogous to a parameter list passed by a program to a subroutine; it contains variable data that the channel program is to use in carrying out its assignment. The parameter block also may contain space for variables (results) that the channel is to return to the CPU. Except for the first two words, the format and size of a parameter block are completely open; the PB may be set up to exchange any kind of information between the CPU and the channel program.

A task block is a channel program—a sequence of 8089 instructions that will perform an operation. A typical channel program might use parameter block data to set up the IOP and a device controller for a transfer, perform the transfer, return the results, and then halt. However, there are no restrictions on what a channel program can do; its function may be simple or elaborate to suit the needs of the application.

Before the CPU starts a channel program, it links the program (TB) to the parameter block and the parameter block to the CB as shown in figure 3-3. The links are standard 8086/8088 doubleword pointer variables; the lower-addressed word contains an offset, and the higher-addressed word contains a segment base value. A system may have many different parameter and task blocks; however, only one of each is ever linked to a channel at any given time.

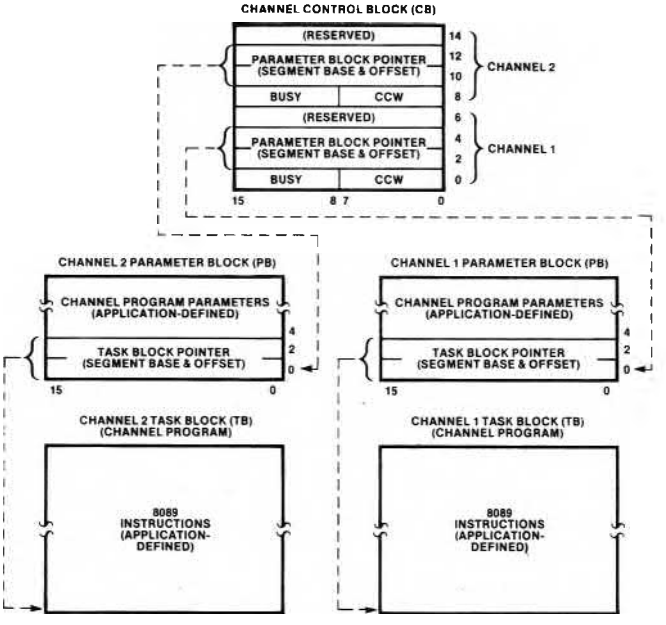


Figure 3-3. Command Communication Blocks

After the CPU has filled in the CCW and has linked the CB to a parameter block and a task block, if appropriate, it issues a channel attention (CA). This is done by activating the IOP's CA (channel attention) and SEL (channel select) pins. The state of SEL at the falling edge of CA directs the channel attention to channel 1 or channel 2. If the IOP is located in the CPU's I/O space, it appears to the CPU as two consecutive I/O ports (one for each channel), and an OUT instruction to the port functions as a CA. If the IOP is memory-mapped, the channels appear as two consecutive memory locations, and any memory reference instruction (e.g., MOV) to these locations causes a channel attention.

An IOP channel attention is functionally similar to a CPU interrupt. When the channel recognizes the CA, it stops what it is doing (it will typically be idle) and examines the command in the CCW. If it is to start a program, the channel loads the addresses of the parameter and task blocks into internal registers, sets its BUSY flag and starts executing the channel program. After it has issued the CA, the CPU is free to perform other processing; the channel can perform its function in parallel, subject to limitations imposed by bus configurations (discussed shortly).

When the channel has completed its program, it notifies the CPU by clearing its BUSY flag in the CB. Optionally, it may issue an interrupt request to the CPU.

The CPU/IOP communication structure is summarized in figure 3-4. Most communication takes place via "message areas" shared in common memory. The only direct hardware communications between the devices are channel attentions and interrupt requests.

Channels

Each of the two IOP channels operates independently, and each has its own register set, channel attention, interrupt request and DMA control signals. At a given point in time, a channel may be idle, executing a program, performing a DMA transfer, or responding to a channel attention. Although only one channel actually runs at a time, the channels can be active concurrently, alternating their operations (e.g., channel 1 may execute instructions in the periods between successive DMA transfer cycles run by channel 2). A built-in priority system allows high-priority activities on one channel to preempt less critical operations on the other channel. The CPU is able to further adjust priorities to handle special cases. The CPU starts the channel and can halt it, suspend it, or cause it to resume a suspended operation by placing different values in the CCW.

Channel Programs (Task Blocks)

Channel programs are written in ASM-89, the 8089 assembly language. About 50 basic instructions are available. These instructions operate on bit, byte, word and doubleword (pointer) variable types; a 20-bit physical address variable type (not used by the 8086/8088) can also be manipulated. Data may be taken from registers, immediate constants and memory. Four memory addressing modes allow flexible access to both memory variables and I/O devices located anywhere in either the CPU's megabyte memory space or in the 8089's 64k I/O space.

The IOP instruction set contains general purpose instructions similar to those found in CPUs as well as instructions specifically tailored for I/O

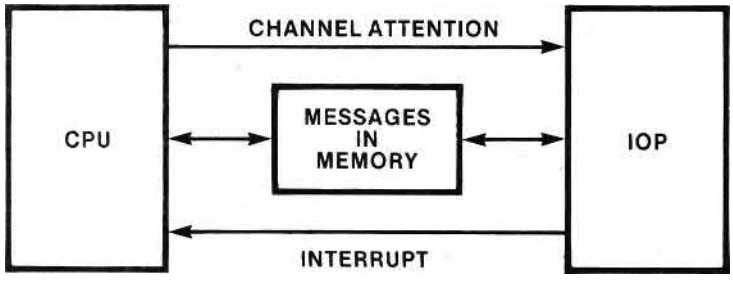


Figure 3-4. CPU/IOP Communication

operations. Data transfer, simple arithmetic, logical and address manipulation operations are available. Unconditional jump and call instructions also are provided so that channel programs can link to each other. An individual bit may be set or cleared with a single instruction. Conditional jumps can test a bit and jump if it is set (or cleared), or can test a value and jump if it is zero (or non-zero). Other instructions initiate DMA transfers, perform a locked test-and-set semaphore operation, and issue an interrupt request to the CPU.

DMA Transfers

The 8089 XFER (transfer) instruction prepares the channel for a DMA transfer. It executes one additional instruction, then suspends program execution and enters the DMA transfer mode. The transfer is governed by channel registers setup by the program prior to executing the XFER instruction.

Data is transferred from a source to a destination. The source and destination may be any locations in the CPU's memory space or in the IOP's I/O space; the IOP makes no distinction between memory components and I/O devices. Thus transfers may be made from I/O device to memory, memory to I/O device, memory to memory and I/O device to I/O device. The IOP automatically matches 8- and 16-bit components to each other.

Individual transfer cycles (i.e., the movement of a byte or a word) may be synchronized by a signal (DMA request) from the source or from the destination. In the synchronized mode, the channel waits for the synchronizing signal before starting the next transfer cycle. The transfer also may be unsynchronized, in which case the channel begins the next transfer cycle immediately upon completion of the previous cycle.

A transfer cycle is performed in two steps: fetching a byte or word from the source into the IOP and then storing it from the IOP into the destination. The IOP automatically optimizes the transfer to make best use of the available data bus widths. For example, if data is being transferred from an 8-bit device to memory that resides on a 16-bit bus (e.g., 8086 memory), the IOP will normally run two one-byte fetch cycles and then store the full word in a single cycle.

Between the fetch and store cycles, the IOP can operate on the data. A byte may be translated to another code (e.g., EBCDIC to ASCII), or compared to a search value, or both, if desired.

A transfer can be terminated by several programmer-specified conditions. The channel can stop the transfer when a specified number (up to 64k) of bytes has been transferred. An external device may stop a transfer by signaling on the channel's external terminate pin. The channel can stop the transfer when a byte (possibly translated) compares equal, or unequal, to a search value. Single-cycle termination, which stops unconditionally after one byte or word has been stored, is also available.

When the transfer terminates, the channel automatically resumes program execution. The channel program can determine the cause of the termination in situations where multiple terminations are possible (e.g., terminating when 80 bytes are transferred or a carriage return character is encountered, whichever occurs first). As an example of post-transfer processing, the channel program could read a result register from the I/O device controller to determine if the transfer was performed successfully. If not (e.g., a CRC error was detected by the controller), the channel program could retry the operation without CPU intervention.

A channel program typically ends by posting the result of the operation to a field supplied in the parameter block, optionally interrupting the CPU, and then halting. When the channel halts, its BUSY flag in the channel control block is cleared to indicate its availability for another operation. As an alternative to being interrupted by the channel, the CPU can poll this flag to determine when the operation has been completed.

Bus Configurations

As shown in figure 3-5, the IOP can access memory or ports (I/O devices) located in a 1-megabyte system space and memory or ports located in a 64-kilobyte I/O space. Although the IOP only has one physical data bus, it is useful to think of the IOP as accessing the system space via a system data bus and the I/O space over an I/O data bus. The distinction between the "two" buses is based on the type-of-cycle signals output

Mnemonics © Intel, 1979

memory write signals, whether they are memory or I/O devices. Components in the I/O space respond to the I/O read and I/O write signals. Thus I/O devices located in the system space are memory-mapped and memory in the I/O space is I/O-mapped. The two basic configuration options differ in the degree to which the IOP shares these buses with the CPU. Both configurations require an 8086/8088 CPU to be strapped in maximum mode.

In the local configuration, shown in figure 3-6, the IOP (or IOPs if two are used) shares both buses with the CPU. The system bus and the I/O bus are the same width (8 bits if the CPU is an space, and the IOP I/O space corresponds to the CPU I/O space. Channel programs are located in the system space; I/O devices may be located in either space. The IOP requests use of the bus for channel program instruction fetches as well as for DMA and programmed transfers. In the local configuration, either the IOP or the CPU may use the buses, but not both simultaneously. The advantage of the local configuration is that intelligent DMA may be added to a system with no additional components beyond the IOP. The disadvantage is that parallel operation of the processors is limited to cases in which the CPU has instruction in its queue that can be executed without using the bus.

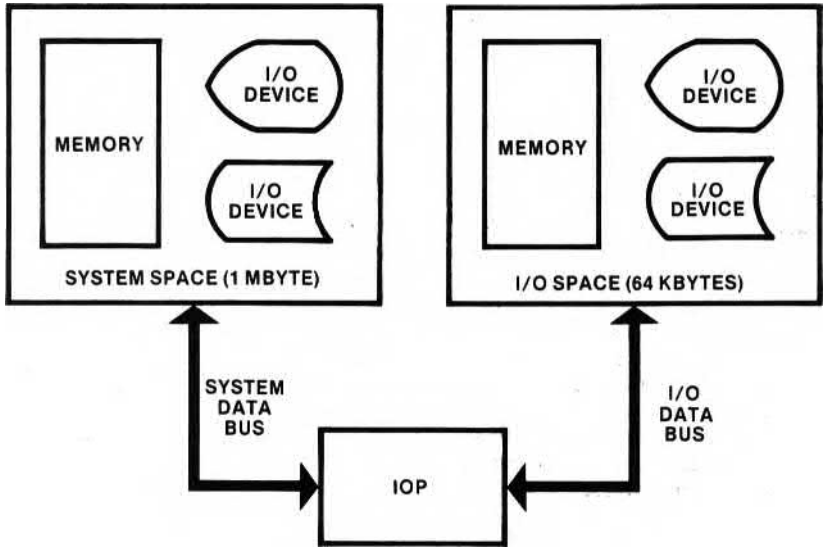


Figure 3-5. IOP Data Buses

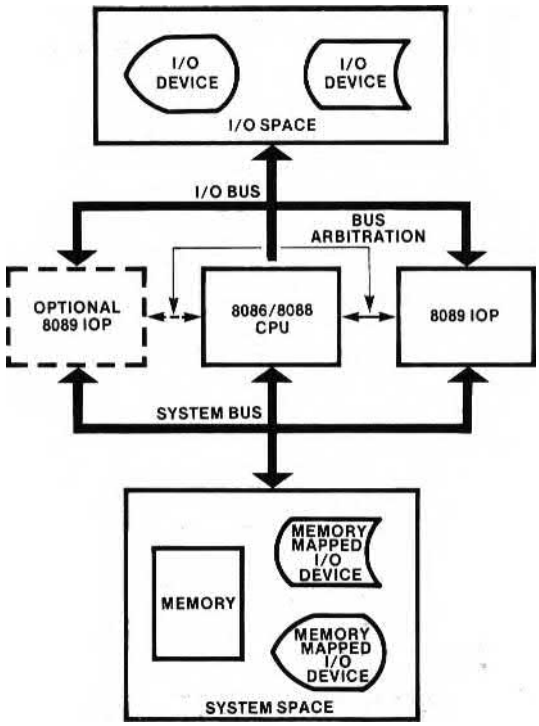


Figure 3-6. Local Configuration

In the remote configuration (figure 3-7), the IOP (or IOPs) shares a common system bus with the CPU. Access to this bus is controlled by 8289 Bus Arbiters. The IOP's I/O bus, however, is physically separated from the CPU in the remote configuration. Two IOPs can share the local I/O bus. Any number of remote IOPs may be contained in a system, configured in remote clusters of one or two. The local I/O bus need not be the same physical width as the shared system bus, allowing an IOP, for example, to interface 8-bit peripherals to an 8086. In the remote configuration, the IOP can access local I/O devices and memory without using the shared system bus, thereby reducing bus contention with the CPU. Contention can further be reduced by locating the IOP's channel programs in the local I/O space. The IOP can then also fetch instructions without accessing the system bus. Parameter, channel control and other CPU/IOP communication blocks must be located in system memory, however, so that both processors can access them. The remote configuration thus increases the degree to which an IOP and a CPU can operate in parallel and thereby increases a system's throughput potential. The price paid for this is that additional hardware must be added to arbitrate use of the shared bus, and to separate the shared and local buses (see Chapter 4 for details).

It is also possible to configure an IOP remote to one CPU, and local to another CPU (see figure 3-8). The local CPU could be used to perform heavy computational routines for the IOP.

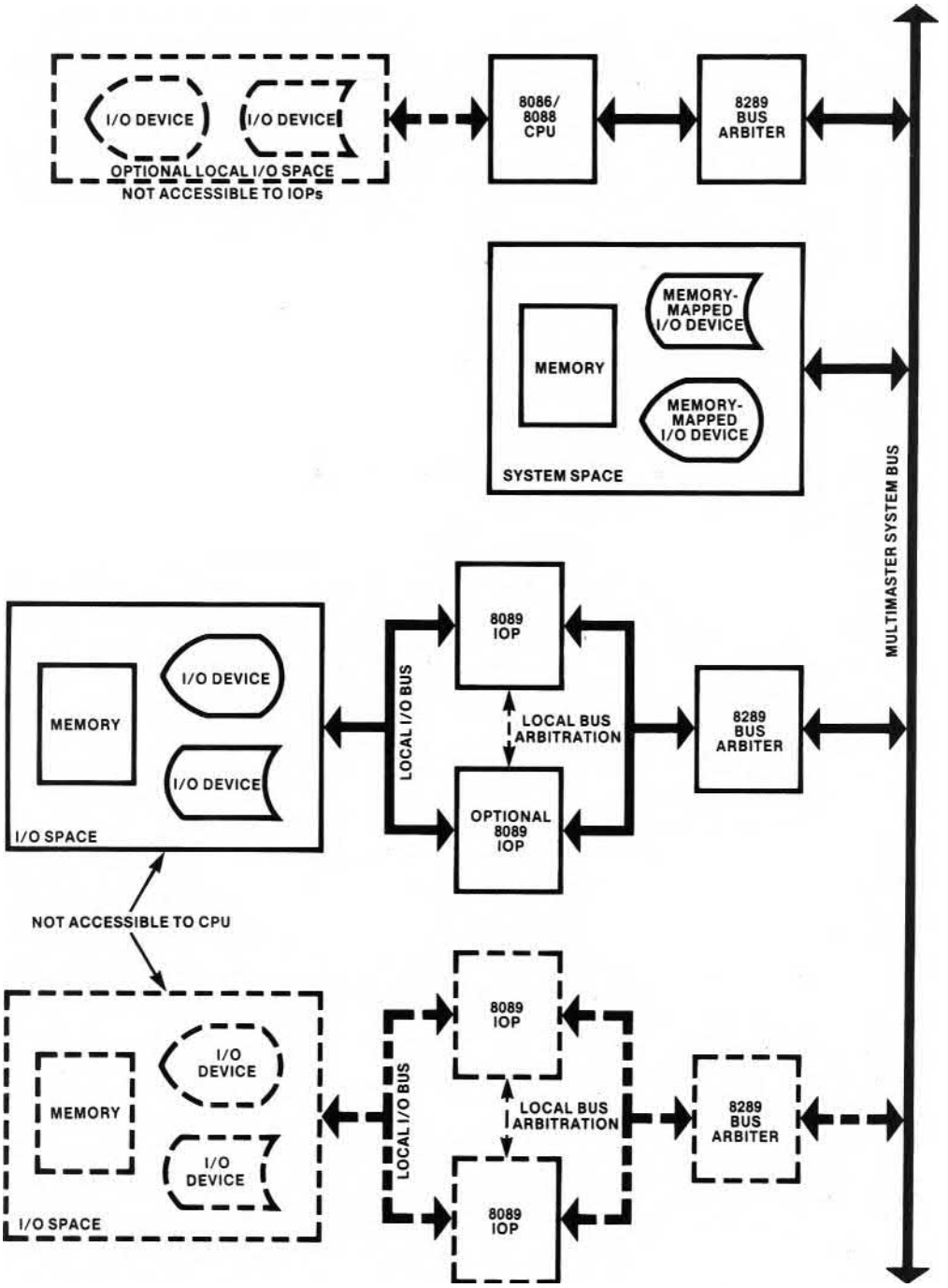


Figure 3-7. Remote Configuration

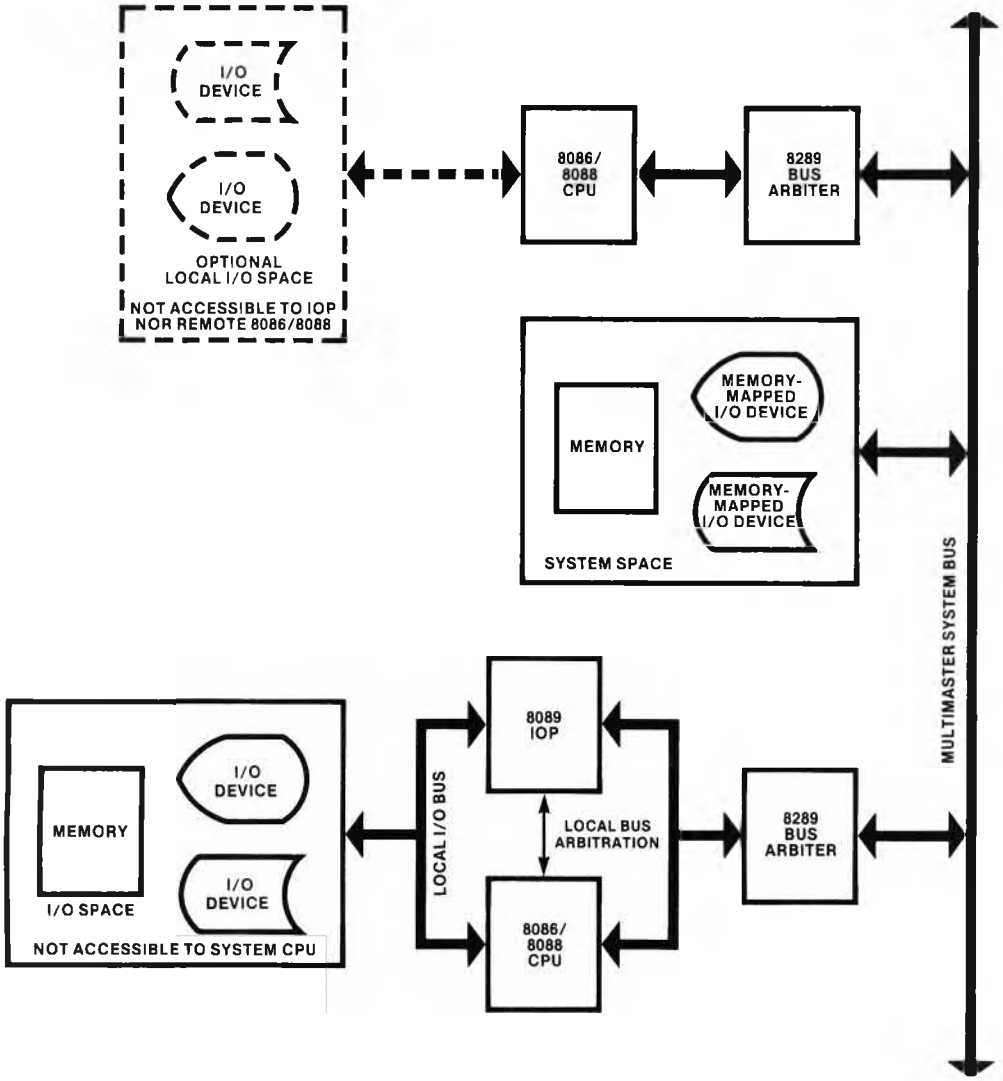


Figure 3-8. Remote IOP Configured With Local 8086/8088

Figure 3-9 shows how a CPU and an IOP might work together to read a record (sector) from a floppy disk. This example is not illustrative of the IOP's full capabilities, but it does review its basic operation and its interaction with a CPU.

The CPU must first obtain exclusive use of a channel. This can be done by performing a "test and set lock" operation on the selected channel's BUSY flag. Assuming the CPU wants to use channel 1, this could be accomplished in PL/M-86 by coding similar to the following:

```
DO WHILE LOCKSET (@CH1.BUSY,0FFH);
    END;
```

In ASM-86 a loop containing the XCHG instruction prefixed by LOCK would accomplish the same thing, namely testing the BUSY flag until it is clear (0H), and immediately setting it to FFH (busy) to prevent another task or processor from obtaining use of the channel.

Having obtained the channel, the CPU fills in a parameter block (see figure 3-10). In this case, the CPU passes the following parameters to the channel: the address of the floppy disk controller, the address of the buffer where the data is to be placed, and the drive, track and sector to be read. It also supplies space for the IOP to return the result of the operation. Note that this is quite a "low-level" parameter block in that it implies that the CPU has detailed knowledge of the I/O system. For a "real" system, a higher-level parameter block would isolate the CPU from I/O device characteristics. Such a block might contain more general parameters such as file name and record key.

After setting up the parameter block, the CPU writes a "start channel program" command in channel 1's CCW. Then the CPU places the address of the desired channel program in the parameter block and writes the parameter block address in the CB. Notice that in this simple example, the CPU "knows" the address of the channel program for reading from the disk, and presumably also "knows" the address of another program for writing, etc. A more general solution would be to place a function code (read, write, channel program execute different routines depending on which function is requested.

After the communication blocks have been setup, the CPU dispatches the channel by issuing a channel attention, typically by an OUT instruction for an I/O-mapped 8089, or a MOV or other memory reference instruction for a memory-mapped 8089.

The channel begins executing the channel program (task block) whose address has been placed in the parameter block by the CPU. In this case the program initializes the 8271 Floppy Disk Controller by sending it a "read data" command followed by a parameter indicating the track to be read. The program initializes the channel registers that define and control the DMA transfer.

Having prepared the 8271 and the channel itself, the channel program executes a XFER instruction and sends a final parameter (the sector to be read) to the 8271. (The 8271 enters DMA transfer mode immediately upon receiving the last of a series of parameters; sending the last parameter after the XFER instruction gives the channel time to setup for the transfer.) The DMA transfer begins when the 8271 issues a DMA request to the channel. The transfer continues until the 8271 issues an interrupt request, indicating that the data has been transferred or that an error has occurred. The 8271's interrupt request line is tied to the IOP's EXT1 (external terminate on channel 1) pin so that the channel interprets an interrupt request as an external terminate condition. Upon termination of the transfer, the channel resumes executing instructions and reads the 8271 result register to determine if the data was read successfully. If a soft (correctable) error is indicated, the IOP retries the transfer. If a hard (uncorrectable) error is detected, or if the transfer has been successful, the IOP posts the content of the result register to the parameter block result field, thus passing the result back to the CPU. The channel then interrupts the CPU (to inform the CPU that the request has been processed) and halts.

When the CPU recognizes the interrupt, it inspects the result field in the parameter block to see if the content of the buffer is valid. If so, it uses the data; otherwise it typically executes an error routine.

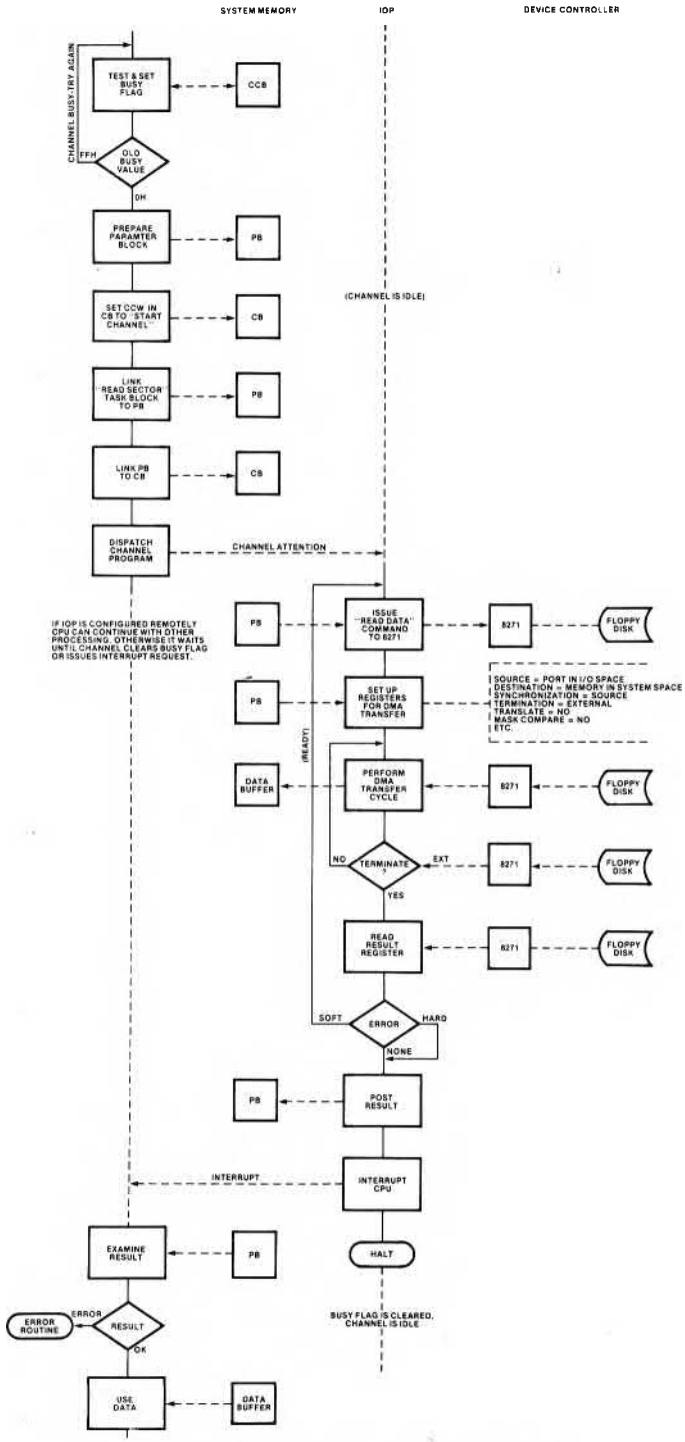


Figure 3-9. Sample CPU/IOP Transaction

POINTER TO CHANNEL PROGRAM		0
(OFFSET & SEGMENT)		2
DEVICE ADDRESS		4
POINTER TO BUFFER		6
(OFFSET & SEGMENT)		8
TRACK	DRIVE	10
RESULT	SECTOR	12

Figure 3-10. Sample Parameter Block

Applications

Combining the raw speed and responsiveness of a traditional DMA controller, an I/O-oriented instruction set, and a flexible bus organization, the 8089 IOP is a very versatile I/O system. Applications with demanding I/O requirements, previously beyond the abilities of microcomputer systems, can be undertaken with the IOP. These kinds of I/O-intensive applications include:

- systems that employ high-bandwidth, low-latency devices such as hard disks and graphics terminals;
- systems with many devices requiring asynchronous service; and
- systems with high-overhead peripherals such as intelligent CRTs and graphics terminals.

In addition, virtually every application that performs a moderate amount of I/O can benefit from the design philosophy embodied in the IOP: system functions should be distributed among special-purpose processors. An IOP channel program is likely to be both faster and smaller than an equivalent program implemented with a CPU. Programming also is more straightforward with the IOP's specialized instruction set.

Removing I/O from the CPU and assigning it to one or more IOPs simplifies and structures a system's design. The main interface to the I/O system can be limited to the parameter blocks. Once these are defined, the I/O system can be designed and implemented in parallel with the rest of the system. I/O specialists can work on the I/O system without detailed knowledge of the application; conversely, the operating system and application teams do not need to be expert in the operation of I/O devices. Standard high-level I/O systems can be used in multiple application systems. Because the application and I/O systems are almost independent, application system changes can be introduced without affecting the I/O system. New peripherals can similarly be incorporated into a system without impacting applications or operating system software. The IOP's simple CPU interface also is designed to be compatible with future Intel CPUs.

Keeping in mind the true general-purpose nature of the IOP, some of the situations where it can be used to advantage are:

- Bus matching - The IOP can transfer data between virtually any combination of 8- and 16-bit memory and I/O components. For example, it can interface a 16-bit peripheral to an 8-bit CPU bus, such as the 8088 bus. The IOP also provides a straightforward means of performing DMA between an 8-bit peripheral and 8086 memory that is split into odd- and even-addressed banks. The 8089 can access both 8- and 16-bit peripherals connected to a 16-bit bus.
- String processing - The 8089 can perform a memory move, translate, scan-for-match or scan-for-nonmatch operation much faster than the equivalent instructions in an 8086 or 8088. Translate and scan operations can be setup so that the source and destination refer to the same addresses to permit the string to be operated on in place.
- Spooling - Data from low-speed devices such as terminals and paper tape readers can be read by the 8089 and placed in memory or on disk until the transmission is complete. The IOP can then transfer the data at high speed when it is needed by an application program. Conversely, output data ultimately destined for a low-speed device such as a printer, can be temporarily spooled to disk and then printed later. This permits batches of data to be gathered or distributed by low-priority programs that run in the background, essentially using up "spare" CPU and IOP cycles. Application programs that use or produce the data can execute faster because they are not bound by the low-speed devices.

- Multitasking operating systems - A multitasking operating system can dispatch I/O tasks to channels with an absolute minimum of overhead. Because a remote channel can run in parallel with the CPU, the operating system's capacity for servicing application tasks can increase dramatically, as can its ability to handle more, and faster, I/O devices. If both channels of an IOP are active concurrently, the IOP automatically gives preference to the higher-priority activity (e.g., DMA normally preempts channel program execution). The operating system can adjust the priority mechanism and also can halt or suspend a channel to take care of a critical asynchronous event.

- Disk systems - The IOP can meet the speed and latency requirements of hard disks. It can be used to implement high-level, file-oriented systems that appear to application programs as simple commands: OPEN, READ, WRITE, etc. The IOP can search and update disk directories and maintain free space maps. "Hierarchical memory" systems that automatically transfer data among memory, high-speed disks and low-speed disks, based on frequency of use, can be built around IOPs. Complex database searches (reading data directly or following pointer chains) can appear to programs as simple commands and can execute in parallel with application programs if an IOP is configured remotely.

- Display terminals - The 8089 is well suited to handling the DMA requirements of CRT controllers. The IOP's transfer bandwidth is high enough to support both alphanumeric and graphic displays. The 8089 can assume responsibility for refreshing the display from memory data; in the remote configuration, the refresh overhead can be removed from the system bus entirely. Linked-list display algorithms may be programmed to perform sophisticated modes of display.

  Each time it performs a refresh operation, the IOP can scan a keyboard for input and translate the key's row-and-column format into an ASCII or EBCDIC character. The 8089 can buffer the characters, scanning the stream until an end-of-message character (e.g., carriage return) is detected, and then interrupt the CPU.

  A single IOP can concurrently support an alphanumeric CRT and keyboard on one channel and a floppy disk on the other channel. This configuration makes use of approximately 30 percent of the available bus bandwidth. Performance can be increased within the available bus bandwidth by adding an 8086 or 8088 CPU to a remote IOP configuration. This configuration can provide scaling, rotation or other sophisticated display transformations.

3.2 Processor Architecture

The 8089 is internally divided into the functional units depicted schematically in figure 3-11. The units are connected by a 20-bit data path to obtain maximum internal transfer rates.

Common Control Unit (CCU)

All IOP operations (instructions, DMA transfer cycles, channel attention responses, etc.) are composed of sequences of more basic processes called internal cycles. A bus cycle takes one internal cycle; the execution of an instruction may require several internal cycles. There are 23 different types of internal cycles each of which takes from two to eight clocks to execute, not including possible wait states and bus arbitration times.

The common control unit (CCU) coordinates the activities of the IOP primarily by allocating internal cycles to the various processor units; i.e., it determines which unit will execute the next internal cycle. For example, when both channels are active, the CCU determines which channel has priority and lets that channel run; if the channels have equal priority, the CCU "interleaves" their execution (this is discussed more fully later in this section). The CCU also initializes the processor.

Arithmetic/Logic Unit (ALU)

The ALU can perform unsigned binary arithmetic on 8- and 16-bit binary numbers. Arithmetic results may be up to 20 bits in length. Available arithmetic instructions include addition, increment and decrement. Logical operations ("and," "or" and "not") may be performed on either 8- or 16-bit quantities.

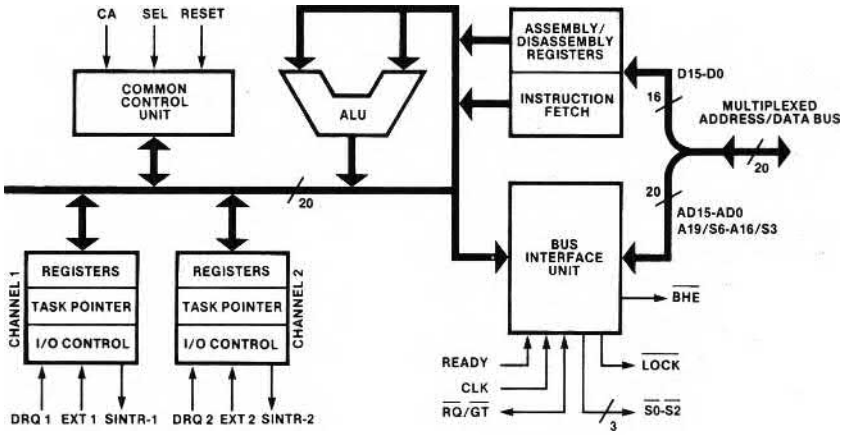


Figure 3-11. 8089 Block Diagram

Assembly/Disassembly Registers

All data entering the chip flows through these registers. When data is being transferred between different width buses, the 8089 uses the assembly/disassembly registers to effect the transfer in the fewest possible bus cycles. In a DMA transfer from an 8-bit peripheral to 16-bit memory, for example, the IOP runs two bus cycles, picking up eight bits in each cycle, assembles a 16-bit word, and then transfers the word to memory in a single bus cycle. (The first and last cycles of a transfer may be performed differently to accommodate odd-addressed words; the IOP automatically adjusts for this condition.)

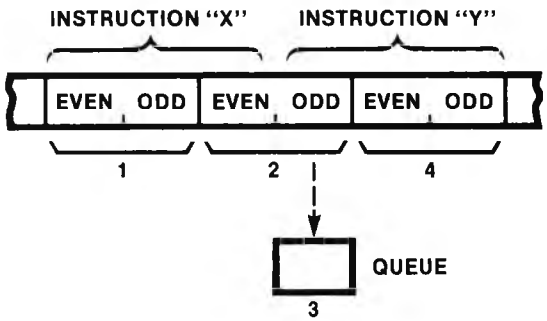
Instruction Fetch Unit

This unit controls instruction fetching for the executing channel (one channel actually runs at a time). If the bus over which the instructions are being fetched is eight bits wide, then the instructions are obtained one byte at a time, and each fetch requires one bus cycle. If the instructions are being fetched over a 16-bit bus, then the instruction fetch unit automatically employs a 1-byte queue to reduce the number of bus cycles. Each channel has its own queue, and the activity of one channel does not affect the other's queue.

During sequential execution, instructions are fetched one word at a time from even addresses; each fetch requires one bus cycle. This process is shown graphically in figure 3-12. When the last byte of an instruction falls on an even address, the odd-addressed byte (the first byte of the following instruction) of the fetched word is saved in the queue. When the channel begins execution of the next instruction, it fetches the first byte from the queue rather than from memory. The queue, then, keeps the processor fetching words, rather than bytes, thereby reducing its use of the bus and increasing throughput.

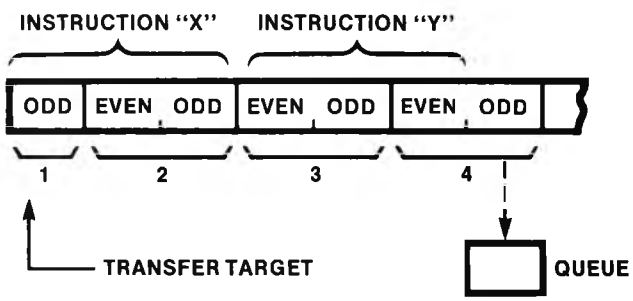
The processor fetches bytes rather than words in two cases. If a program transfer instruction (e.g., JMP or CALL) directs the processor to an instruction located at an odd address, the first byte of the instruction is fetched by itself as shown in figure 3-13. This is because the program transfer invalidates the content of the queue by changing the serial flow of execution.

The second case arises when an LPDI instruction is located at an odd address. In this situation, the six-byte LPDI instruction is fetched: byte, word, byte, byte, byte, and the queue is not used. The first byte of the following instruction is fetched in one bus cycle as if it had been the target of a program transfer. Word fetching resumes with this instruction's second byte.



FETCH	INSTRUCTION BYTES
1	FIRST TWO BYTES OF "X"
2	THIRD BYTE OF "X" PLUS FIRST BYTE OF "Y", WHICH IS SAVED IN QUEUE
3	FIRST BYTE OF "Y" FROM QUEUE—NO BUS CYCLE
4	LAST TWO BYTES OF "Y"

Figure 3-12. Sequential Instruction Fetching (16-Bit Bus)



FETCH	INSTRUCTION BYTES
1	FIRST (ODD-ADDRESSED) BYTE OF "X" (8-BIT BUS CYCLE)
2	SECOND AND THIRD BYTES OF "X"
3	FIRST AND SECOND BYTES OF "Y".
4	THIRD BYTE OF "Y" PLUS FIRST BYTE OF NEXT INSTRUCTION, WHICH IS SAVED IN QUEUE

Figure 3-13. Instruction Fetching Following a Program Transfer to an Odd Address (16-Bit Bus)

Bus Interface Unit (BIU)

The BIU runs all bus cycles, transferring instructions and data between the IOP and external memory or peripherals. Every bus access is associated with a register tag bit that indicates to the BIU whether the system or I/O space is to be addressed. The BIU outputs the type of bus cycle (instruction fetch from I/O space, data store into system space, etc.) on status lines $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$. An 8288 Bus Controller decodes these lines and provides signals that selectively enable one bus or the other (see Chapter 4 for details).

The BIU further distinguishes between the physical and logical widths of the system and I/O buses. The physical widths of the buses are fixed and are communicated to the BIU during initialization. In the local configuration, both buses must be the same width, either 8 or 16 bits (matching the width of the host CPU bus). In the remote configuration, the IOP system bus must be the same physical width as the bus it shares with the CPU. The width of the IOP's I/O bus, which is local to the 8089, may be selected independently. If any 16-bit peripherals are located in the I/O space, then a 16-bit I/O bus must be used. If only 8-bit devices reside on the I/O bus, then either an 8- or a 16-bit I/O bus may be selected. A 16-bit I/O bus has the advantage of easy accommodation of future 16-bit devices and fewer instruction fetches if channel programs are placed in the I/O space.

For a given DMA transfer, a channel program specifies the logical width of the system and the I/O buses; each channel specifies logical bus widths independently. The logical width of an 8-bit physical bus can only be eight bits. A 16-bit physical bus, however, can be used as either an 8- or 16-bit logical bus. This allows both 8- and 16-bit devices to be accessed over a single 16-bit physical bus. Table 3-1 lists the permissible physical and logical bus widths for both locally and remotely configured IOPs. Logical bus width pertains to DMA transfers only. Instructions are fetched and operands are read and written in bytes or words depending on physical bus width.

In addition to performing transfers, the BIU is responsible for local bus arbitration. In the local configuration, the BIU uses the $\overline{RQ}/\overline{GT}$ (request/grant) line to obtain the bus from the CPU and to return it after a transfer has been performed. In the remote configuration, the BIU uses $\overline{RQ}/\overline{GT}$ to coordinate use of the local I/O bus with another IOP or a local CPU, if present. System bus arbitration in the remote configuration is performed by an 8289 Bus Arbiter that operates invisibly to the IOP. The BIU automatically asserts the $\overline{LOCK}$ (bus lock) signal during execution of a TSL (test and set lock) instruction and, if specified by the channel program, can assert the $\overline{LOCK}$ signal for the duration of a DMA transfer. Section 3.5 contains a complete discussion of bus arbitration.

Table 3-1. Physical/Logical Bus Combinations

Configuration	System Bus Physical:Logical	I/O Bus Physical:Logical
Local	8:8 16:8/16	8:8 16:8/16
Remote	8:8 16:8/16 16:8/16 8:8	8:8 16:8/16 8:8 16:8/16

Channels

Although the 8089 is a single processor, under most circumstances it is useful to think of it as two independent channels. A channel may perform DMA transfers and may execute channel programs; it also may be idle. This section describes the hardware features that support these operations.

I/O Control

Each channel contains its own I/O control section that governs the operation of the channel during DMA transfers. If the transfer is synchronized, the channel waits for a signal on its DRQ (DMA request) line before performing the next fetch-store sequence in the transfer. If the transfer is to be terminated by an external signal, the channel monitors its EXT (external terminate) line and stops the transfer when this line goes active. Between the fetch and store cycles (when the data is in the IOP) the channel optionally counts,

translates, and scans the data, and may terminate the transfer based on the results of these operations. Each channel also has a SINTR (system interrupt) line that can be activated by software to issue an interrupt request to the CPU.

Registers

Figure 3-14 illustrates the channel register set, and table 3-2 summarizes the uses of each register. Each channel has an independent set of registers; they are not accessible to the other channel. Most of the registers play different roles during channel program execution than in DMA transfers. Channel programs must be careful to save these registers in memory prior to a DMA transfer if their values are needed following the transfer.

General Purpose A (GA). A channel program may use GA for a general register or a base register. A general register can be an operand of most IOP instructions; a base register is used to address memory operands (see section 3.8). Before initiating a DMA transfer, the channel program points GA to either the source or destination address of the transfer.

General Purpose B (GB). GB is functionally interchangeable with GA. If GA points to the source of a DMA transfer, then GB points to the destination, and vice versa.

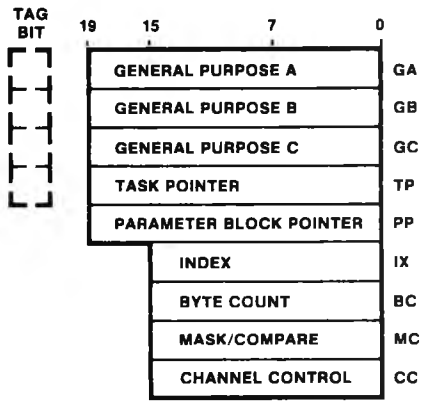


Figure 3-14. Channel Register Set

General Purpose C (GC). GC may be used as a general register or a base register during channel program execution. If data is to be translated during a DMA transfer, then the channel program loads GC with the address of the first byte of a translation table before initiating the transfer. GC is not altered by a transfer operation.

Task Pointer (TP). The CCU loads TP from the parameter block when it starts or resumes a channel program. During program execution, the channel automatically updates TP to point to the

Table 3-2. Channel Register Summary

Register	Size	Program Access	System or I/O Pointer	Use by Channel Programs	Use in DMA Transfers
GA	20	Update	Either	General, base	Source/destination pointer
GB	20	Update	Either	General, base	Source/destination pointer
GC	20	Update	Either	General, base	Translate table pointer
TP	20	Update	Either	Procedure return, instruction pointer	Adjusted to reflect cause of termination
PP	20	Reference	System	Base	N/A
IX	16	Update	N/A	General, auto-increment	N/A
BC	16	Update	N/A	General	Byte counter
MC	16	Update	N/A	General, masked compare	Masked compare
CC	16	Update	N/A	Restricted use recommended	Defines transfer options

next instruction to be executed; i.e., TP is used as an instruction pointer or program counter. Program transfer instructions (JMP, CALL, etc.) update TP to cause nonsequential execution. A procedure (subroutine) returns to the calling program by loading TP with an address previously saved by the CALL instruction. The task pointer is fully accessible to channel programs; it can be used as a general register or as a base register. Such use is not recommended, however, as it can make programs very difficult to understand.

Parameter Block Pointer (PP). The CCU loads this register with the address of the parameter block before it starts a channel program. The register cannot be altered by a channel program, but is very useful as a base register for accessing data in the parameter block. PP is not used during DMA transfers.

Index (IX). IX may be used as a general register during channel program execution. It also may be used as an index register to address memory operands (the address of the operand is computed by adding the content of IX to the content of a base register). When specified as an index register, IX may be optionally auto-incremented as the last step in the instruction to provide a convenient means of "stepping" through arrays or strings. IX is not used in DMA transfers.

Byte Count (BC). BC may be used as a general register during channel program execution. If DMA is to be terminated when a specific number of bytes has been transferred, BC should be loaded with the desired byte count before initiating the transfer. During DMA, BC is decremented for each byte transferred, whether byte count termination has been selected or not. If BC reaches zero, the transfer is stopped only if byte count termination has been specified. If byte count termination has not been selected, BC "wraps around" from 0H to FFFFH and continues to be decremented.

Mask/Compare (MC). A channel program may use MC for a general register. This register also may be used in either a channel program or in a DMA transfer to perform a masked compare of a byte value. To use MC in this way, the program loads a compare value in the low-order eight bits of the register and a mask value in the upper eight bits (see figure 3-15). A "1" in a mask bit *selects* the bit in the corresponding position in the compare value; a "0" in a mask bit *masks* the corresponding bit in the compare value. In figure 3-15, a value compared with MC will be considered equal if its low-order five bits contain the value 00100; the upper three bits may contain any value since they are masked out of the comparison.

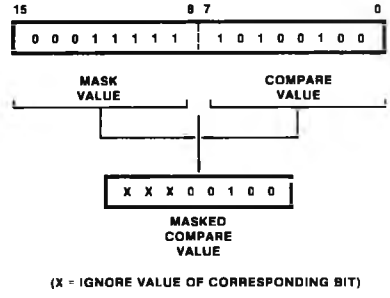


Figure 3-15. Mask/Compare Register

Channel Control (CC). The content of the channel control register governs a DMA transfer (see figure 3-16). A channel program loads this register with appropriate values before beginning the transfer operation; section 3.4 covers the encoding of each field in detail. Bit 8 (the chain bit) of CC pertains to channel program execution rather than to a DMA transfer. When this bit is zero, the channel program runs at normal priority; when it is one, the priority of the program is raised to the same level as DMA (priorities are covered later in this section). Although a channel program may use CC as a general register, such use is not recommended because of the side effects on the chain bit and thus on the priority of the channel program. Channel programs should restrict their use of CC to loading control values in preparation for a DMA transfer, setting and clearing the chain bit, and storing the register.

Program Status Word (PSW)

Each channel maintains its own program status word (PSW) as shown in figure 3-17. Channel programs do not have access to the PSW. The PSW records the state of the the channel so that channel operation may be suspended and then resumed later. When the CPU issues a "suspend" command, the channel saves the PSW, task pointer, and task pointer tag bit in the first four bytes of the channel's parameter block as shown in figure 3-18. Upon receipt of a subsequent

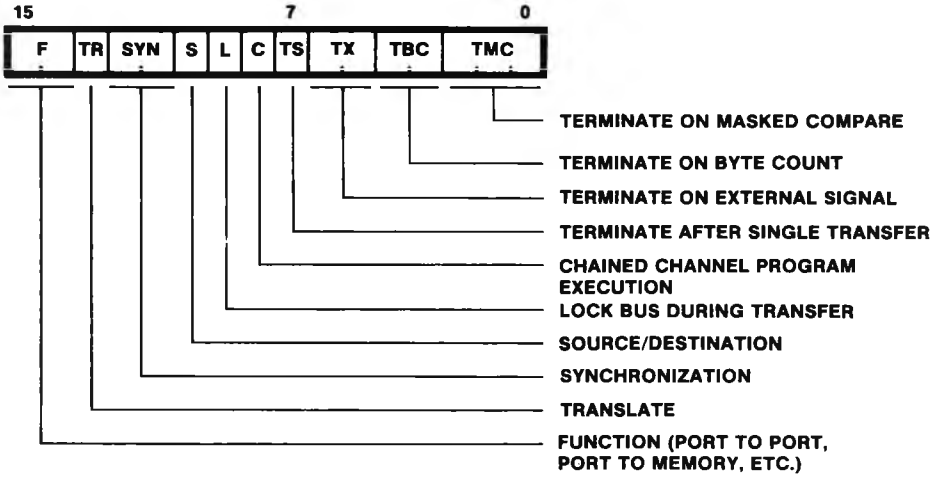


Figure 3-16. Channel Control Register

"resume" command, the PSW, TP, and TP tag bit are restored from the parameter block save area and execution resumes.

Two conditions override the normal channel priority mechanism. If one channel is performing DMA (priority 1) and the channel receives a channel attention (priority 2), the channel attention is serviced at the end of the current DMA transfer cycle. This override prevents a synchronized DMA transfers from "shutting out" a channel attention. DMA terminations and chained channel programs postpone recognition of a CA on the *other* channel; the CA is latched, however, and is serviced as soon as priorities permit.

The IOP's $\overline{\text{LOCK}}$ (bus lock) signal also supersedes channel switching. A running channel will not relinquish control of the processor while $\overline{\text{LOCK}}$ is active, regardless of the priorities of the activities on the two channels. This is consistent with the purpose of the $\overline{\text{LOCK}}$ signal: to guarantee exclusive access to a shared resource in a multiprocessing system. Refer to sections 3.5 and 3.7 for futher information on the $\overline{\text{LOCK}}$ signal and the TSL instruction.

Tag Bits

Registers GA, GB, GC, and TP are called pointer registers because they may be used to access, or

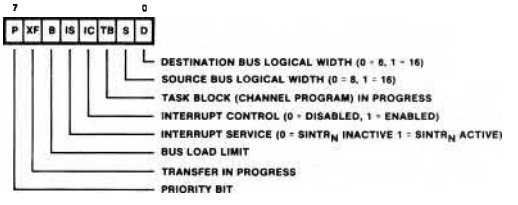


Figure 3-17. Program Status Word

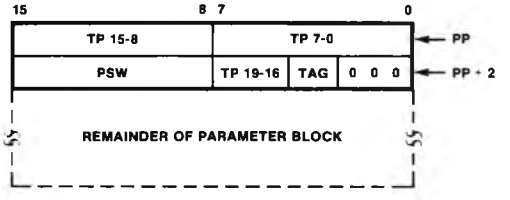


Figure 3-18. Channel State Save Area

point to, addresses in either the system space or the I/O space. The pointer registers may address either memory or I/O devices (IOP instructions do not distinguish between memory and I/O devices since the latter are memory-mapped). The tag bit associated with each register (figure 3-14) determines whether the register points to an address in the system space (tag=0) or the I/O space (tag=1).

The CCU sets or clears TP's tag bit depending on whether the command it receives from the CPU is "start channel program in system space," or "start channel program in I/O space." Channel programs alter the tag bits of GA, GB, GC, and TP by using different instructions for loading the registers. Briefly, a "load pointer" instruction clears a tag bit, a "move" instruction sets a tag bit, and a "move pointer" instruction moves a memory value (either 0 or 1) to a tag bit. Section 3.9 covers these instructions in detail.

If a register points to the system space, all 20 bits are placed on the address lines to allow the full megabyte to be directly addressed. If a register points to the I/O space, the upper four bits of the address lines are undefined; the lower 16 bits are sufficient to access any location in the 64k byte I/O space.

Concurrent Channel Operation

Both channels may be active concurrently, but only one can actually run at a time. At the end of each internal cycle, the CCU lets one channel or the other execute the next internal cycle. No extra overhead is incurred by this channel switching. The basis for making the determination is a priority mechanism built into the IOP. This mechanism recognizes that some kinds of activities (e.g., DMA) are more important than others. Each activity that a channel can perform has a priority that reflects its relative importance (see table 3-3).

Two new activities are introduced in table 3-3. When a DMA transfer terminates, the channel executes a short internal channel program. This DMA termination program adjusts TP so that the user's program resumes at the instruction specified when the transfer was setup (this is discussed in detail in section 3.4). Similarly, when a channel attention is recognized, the channel executes an internal program that examines the CCW and carries out its command. Both of these programs consist of standard 8089 instructions that are fetched from internal ROM. Intel Application Note AP-50, *Debugging Strategies and Considerations for 8089 Systems*, lists the instructions in these programs. Users monitoring the bus during debugging may see operands read or written by the termination or channel attention programs. The instructions themselves, however, will not appear on the bus as they are resident in the chip.

Notice also that, according to table 3-3, a channel program may run at priority 3 or at priority 1.

Table 3-3. Channel Priorities and Interleave Boundaries

Channel Activity	Priority (1 = highest)	Interleave Boundary	
		By DMA	By Instruction
DMA transfer	1	Bus cycle[1]	Bus cycle[1]
DMA termination sequence	1	Internal cycle	None
Channel program (chained)	1	Internal cycle[2]	Instruction
Channel attention sequence	2	Internal cycle	None
Channel program (not chained)	3	Internal cycle[2]	Instruction
Idle	4	Two clocks	Two clocks

[1]DMA is not interleaved while $\overline{\text{LOCK}}$ is active.
[2]Except TSL instruction; see section 3.7.

Channel program priority is determined by the chain bit in the channel control register. If this bit is cleared, the program runs at normal priority (3); if it is set, the program is said to be chained, and it runs at the same priority as DMA. Thus, the chain bit provides a way to raise the priority of a critical channel program.

The CCU lets the channel with the highest priority run. If both channels are running activities with the same priority, the CCU examines the priority bits in the PSWs. If the priority bits are unequal, the channel with the higher value (1) runs. Thus, the priority bit serves as a "tie breaker" when the channels are otherwise at the same priority level. The value of the priority bit in the PSW is loaded from a corresponding bit in the CCW; therefore, the CPU can control which channel will run when the channels are at the same priority level. The priority bit has no effect when the channel priorities are different. If both channels are at the same priority level and if both priority bits are equal, the channels run alternately without any additional overhead.

The CCU switches channels only at certain points called interleave boundaries; these vary according to the type of activity running in each channel and are shown in table 3-3. In table 3-3 and in the following discussion, the terms "channel A" and "channel B" are used to identify two active channels that are bidding for control of an IOP. "Channel A" is the channel that last ran and will run again unless the CCU switches to "channel B." Where the CCU switches from one channel (channel A) to another (channel B) depends on whether channel B is performing DMA or is executing instructions. For this determination, instructions in the internal ROM are considered the same as instructions executed in user-written channel programs (chained or not chained). Table 3-3 shows that a switch from channel A to channel B will occur sooner if channel B is running DMA. DMA, then, interleaves instruction execution at internal cycle boundaries. Since instructions are often composed of several internal cycles, instruction execution on channel A can be suspended by DMA on channel B (when channel A next runs, the instruction is resumed from the point of suspension). DMA on channel A is interleaved by DMA on channel B after any bus cycle (when channel A runs again, the DMA transfer sequence is resumed from the point of suspension). If both channels are executing programs, the interleave boundaries are extended to instruction boundaries: a program on channel B will not run until channel A reaches the end of an instruction. Note that a DMA termination sequence or channel attention sequence on channel A cannot be interleaved by instructions on channel B, regardless of channel B's priority. These internal programs are short, however, and will not delay channel B for long (see Chapter 4 for timing information).

Table 3-4 summarizes the channel switching mechanism with several examples. It is important to remember that channel switching occurs only when both channels are ready to run. In typical applications, one of the channels will be idle much of the time, either because it is waiting to be dispatched by the CPU or because it is waiting for a DMA request in a synchronized transfer. (During a synchronized transfer, the channel is idle between DMA requests; for many peripherals, the channel will spend much more time idling than executing DMA cycles.) The real potential for one channel "shutting out" a priority 1 activity on the other channel is largely limited to unsynchronized DMA transfers and locked transfers (synchronized or unsynchronized). Long, chained channel programs and high-speed synchronized DMA will slow a priority 1 activity on the other channel, but will not shut it out because the channels will alternate (assuming their priority bits are equal). A chained channel program will shut out any lower priority activity on the other channel, including a channel attention. (The channel attention is latched by the IOP, however, so it will execute when the other channel drops to a lower priority.) Chained channel programs should therefore be used with discretion and should be made as short as possible.

3.3 Memory

The 8089 can access memory components located in two different address spaces. The system space, which coincides with the CPU's memory space, may contain up to 1,048,576 bytes. The I/O space, which may either coincide with the CPU's I/O space or be local (private) to the IOP, may contain up to 65,536 bytes. Memory components in the system space should respond to the memory read and write commands issued by the 8288 Bus Controller. Memory components in the I/O space must respond to 8288 I/O read and write commands. Memory in either space may be

Table 3-4. Channel Switching Examples

Channel A (Ran Last)				Channel B			Result
Activity	Chain Bit	Priority Bit	LOCK	Activity	Chain Bit	Priority Bit	
DMA transfer	X	X	Inactive	Idle	X	X	A runs.
DMA transfer	X	X	Inactive	Channel attention	X	X	A runs until end of current transfer cycle; then B runs.
Channel program	X	0	Inactive	Channel program	X	1	B runs.
Channel program	X	0	Inactive	Channel program	X	0	A and B alternate by instruction.
Channel program	1	X	Inactive	Channel program	0	X	A runs.
DMA transfer	X	1	Inactive	Channel program	1	1	B runs one bus or internal cycle following each bus cycle run by A.*
Channel attention	X	X	Inactive	Channel program	1	X	A runs if it has started the sequence; otherwise B runs.
DMA transfer	X	X	Active	Channel attention	X	X	A runs until DMA terminates.
Channel program (TSL instruction)	0	X	Active	DMA transfer	X	X	A completes TSL instruction, LOCK goes inactive and B runs.

*If transfer is synchronized, B also runs when A goes idle between transfer cycles.

implemented like 8086 memory (16-bit words split into even- and odd-addressed 8-bit banks) or 8088 memory (a single 8-bit bank). See Chapter 4 for physical implementation considerations.

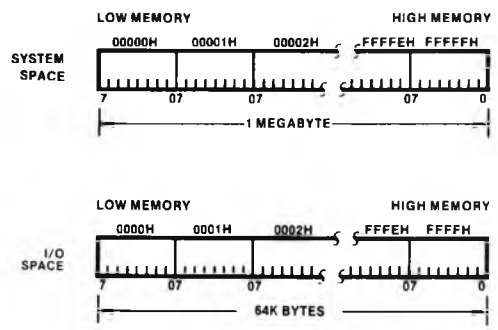


Figure 3-19. Storage Organization

Storage Organization

From a software point of view, both 8089 memory spaces are organized as unsegmented arrays of individually addressable 8-bit bytes (figure 3-19). Instructions and data may be stored at any address without regard for alignment (figure 3-20).

The IOP views the system space differently from the 8086 or 8088 with which it typically shares the space. The 8086 and 8088 differentiate between a location's logical (segment and offset) address and its physical (20-bit) address.

The 8089 does not "see" the logically segmented structure of the memory space; it uses its 20-bit pointer registers to access all locations in the system space by their physical addresses. Memory in the 8089 I/O space is treated similarly except that only 16 bits are needed to address any location.

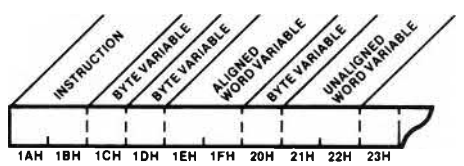


Figure 3-20. Instruction and Variable Storage

Following Intel convention, word data is stored with the most-significant byte in the higher address (see figure 3-21). The 8089 recognizes the doubleword pointer variable used by the 8086 and 8088 (figure 3-22). The lower-addressed word of the pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally, with the higher-addressed byte containing the most-significant eight bits of the word. The 8089 can convert a doubleword pointer into a 20-bit physical address when it is loaded into a pointer register to address system memory. A special 3-byte variable, called a physical address pointer (figure 3-23), is used to save and restore pointer registers and their associated tag bits.

Dedicated and Reserved Memory Locations

The extreme low and high addresses of the system space are dedicated to specific processor functions or are reserved for use by other Intel hardware and software products; the locations are 0H through 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes), as shown in figure 3-24. The low addresses are used for part of the 8086/8088 interrupt pointer table. Locations FFFF0H-FFFFBH are used for 8086, 8088 and 8089 startup sequences; the remaining locations are reserved by Intel.

If an IOP is configured locally, its I/O space coincides with the CPU's I/O space, and it must respect the reserved addresses F8H-FFH. The entire I/O space of a remotely-configured IOP may be used without restriction.

Using any dedicated or reserved addresses may inhibit the compatibility of a system with current or future Intel hardware and software products.

Dynamic Relocation

The 8089 is very well-suited to environments in which programs do not occupy static memory locations, but are moved about during execution. Dynamic code relocation allows systems to make efficient use of limited memory resources by transferring programs between external storage and memory, and by combining scattered free areas of memory into larger, more useful, continuous spaces.

IOP channel programs are inherently position-independent, the only restriction being that channel programs that transfer to each other or share data must be moved as a unit. Since the IOP

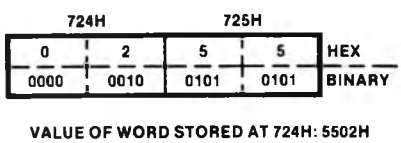


Figure 3-21. Storage of Word Variables

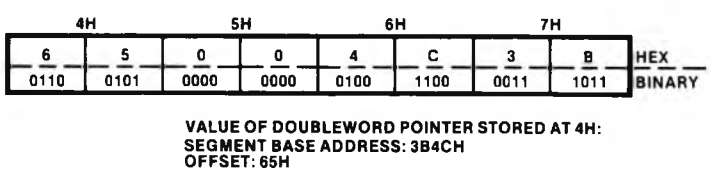


Figure 3-22. Storage of Doubleword Pointer Variables

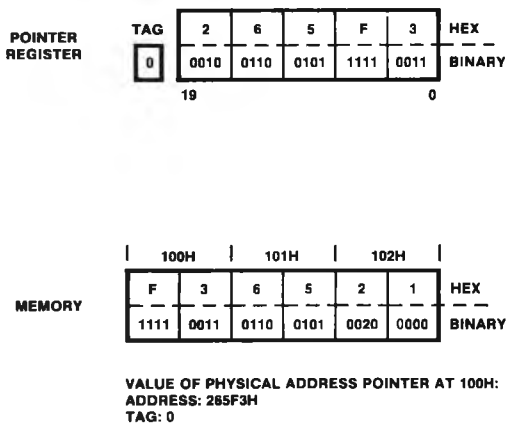


Figure 3-23. Storage of Physical Address Pointer Variables

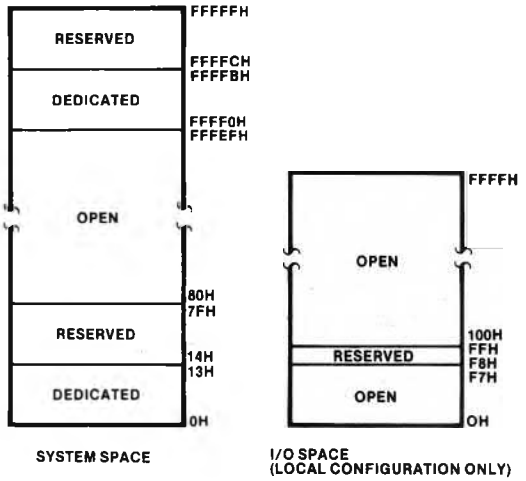


Figure 3-24. Reserved Memory Locations

receives the address of a channel program and its associated parameter block when it is dispatched by the CPU, the location of these blocks is immaterial and can change from one dispatch to the next. (Note, however, that the channel control block cannot be moved without reinitializing the IOP.) Typically, then, the CPU would direct the movement of IOP channel programs and parameter blocks. These blocks, of course, cannot be moved while they are in use.

While the CPU may be in charge of relocation, the IOP is an excellent vehicle for performing the actual transfer of channel programs, parameter blocks, and CPU programs as well. A very simple channel program can transfer code between memory locations by DMA much faster than the equivalent CPU instructions, and transfers between disk and memory also can be performed more efficiently.

Memory Access

Memory accesses are always performed using a pointer register and its associated tag bit. The tag bit indicates whether the access is to the system space (tag=0) or the I/O space (tag=1). The pointer register contains the base address of the location; i.e., the pointer register is used as a base register. Only the low-order 16 bits of the pointer register are used for I/O space locations; all 20 bits are used for system space addresses. Different types of memory accesses use base registers as shown in table 3-5. The 8089 addressing modes allow the base address of a memory operand to be modified by other registers and constant values to yield the effective address of the operand (see section 3.8).

Notice that table 3-5 indicates that memory operands may be addressed using register PP in addition to GA, GB, and GC. PP is maintained by the IOP and can neither be read nor written by a channel program; it can be used, however, to access data in the parameter block. PP has no associated tag bit; a reference to it implies the system space, where a parameter block always resides.

Table 3-5. Base Register Use in Memory Access

Memory Access	Base Register
Instruction Fetch	TP
DMA Source	GA or GB[1]
DMA Destination	GA or GB[1]
DMA Translate Table	GC
Memory Operand	GA or GB or GC or PP[2]

[1]As specified in CC register
[2]As specified in instruction

The IOP is told the physical widths of the system and I/O buses when it is initialized. If a bus is eight bits wide, the IOP accesses memory on this bus like an 8088. Instruction fetches and operand reads and writes are performed one byte at a time; one bus cycle is run for each memory access. Word operands are accessed in two cycles, completely transparent to software. Instruction fetches are made as needed, and the instruction stream is not queued.

The IOP accesses memory on a 16-bit bus like an 8086. As mentioned in the previous section, the instruction stream is generally fetched in words from even addresses with the second byte held in the one-byte queue. If a word operand is aligned (i.e., located at an even address), the 8089 will access it in a single 16-bit bus cycle. If a word operand is unaligned (i.e., located at an odd address), the word will be accessed in two consecutive 8-bit bus cycles. Byte operands are always accessed in 8-bit bus cycles.

For memory on 16-bit buses, performance is improved and bus contention is reduced if word operands are stored at even addresses. The instruction queue tends to reduce the effect of alignment on instructions fetched on a 16-bit bus. In tight loops, performance can be increased by word-aligning transfer targets.

Notice that the correct operation of a program is completely independent of memory bus width. A channel program written for one system that uses an 8-bit memory bus will execute without modification if the bus is increased to 16 bits. It is good practice, though, to write all programs as though they are to run on 16-bit systems; i.e., to align word operands. Such programs will then make optimal use of the bus in whatever system they are run.

3.4 Input/Output

The 8089 combines the programmed I/O capabilities of a CPU with the high-speed block transfer facility of a DMA controller. It also provides additional features (e.g., compare and translate during DMA) and is more flexible than a typical CPU or DMA controller. The 8089 transfers data from a source address to a destination address. Whether the component mapped into a given address is actually memory or I/O is immaterial. All addresses in both the system and I/O spaces are equally accessible, and transfers may be made between the two spaces as well as within either address space.

Programmed I/O

A channel program performs I/O similar to the way a CPU communicates with memory-mapped I/O devices. Memory reference instructions perform the transfer rather than "dedicated" I/O instructions, such as the 8086/8088 IN and OUT instructions. Programmed I/O is typically used to prepare a device controller for a DMA transfer and to obtain status/result information from the controller following termination of the transfer. It may be used, however, with any device whose transfer rate does not require DMA.

I/O Instructions

Since the 8089 does not distinguish between memory components and I/O devices, any instruction that accepts a byte or word memory operand can be used to access an I/O device. Most memory reference instructions take a source operand or a destination operand, or both. The instructions generally obtain data from the source operand, operate on the data, and then place the result of the operation in the destination operand. Therefore, when a source operand refers to an address where an I/O device is located, data is input from the device. Similarly, when a destination operand refers to an I/O device address, data is output to the device.

Most I/O device controllers have one or more internal registers that accept commands and supply status or result information. Working with these registers typically involves:

- reading or writing the entire register;
- setting or clearing some bits in a register while leaving others alone; or
- testing a single bit in a register.

Table 3-6 shows some of the 8089 instructions that are useful for performing these kinds of operations. Section 3.7 covers the 8089 instruction set in detail.

Table 3-6. Memory Reference Instructions Used for I/O

Instruction	Effect on I/O Device
MOV/MOVB	Read or write word/byte
AND/ANDB	Clear multiple bits in word/byte
OR/ORB	Set multiple bits in word/byte
CLR	Clear single bit (in byte)
SET	Set single bit (in byte)
JBT	Read (byte) and jump if single bit =1
JNBT	Read (byte) and jump if single bit =0

Device Addressing

Since memory reference instructions are used to perform programmed I/O, device addressing is very similar to memory addressing. An operand that refers to an I/O device always specifies one of the pointer registers GA, GB, or GC (PP is legal, but an I/O device would not normally be mapped into a parameter block). The base address of the device is taken from the specified pointer register. Any of the memory addressing modes (see section 3.8) may be used to modify the base address to produce the effective (actual) address of the device. The pointer register's tag bit locates the device in the system space (tag=0) or in the I/O space (tag=1). If the device is in the I/O space, only the low-order 16 bits of the pointer register are used for the base address; all 20 bits are used for a system space address. The IOP's system and I/O spaces are fully compatible with the corresponding address spaces of the other 8086 family processors.

I/O Bus Transfers

Table 3-7 shows the number of bus cycles the IOP runs for all combinations of bus size, transfer size (byte or word), and transfer address (even or odd). Bus width refers to the physical bus implementation; the instruction mnemonic determines whether a byte or a word is transferred.

Both 8- and 16-bit devices may reside on a 16-bit bus. All 16-bit devices should be located at even addresses so that transfers will be performed in one bus cycle. The 8-bit devices on a 16-bit bus may be located at odd or even addresses. The internal registers in an 8-bit device on a 16-bit bus must be assigned all-odd or all-even addresses that are two bytes apart (e.g., 1H, 3H, 5H, or 2H, 4H, 6H). All 8-bit peripherals should be referenced with byte instructions, and 16-bit devices should be referenced with word instructions. Odd-addressed 8-bit devices must be able to transfer data on the upper eight bits of the 16-bit physical data bus.

Only 8-bit devices should be connected to an 8-bit bus, and these should only be referenced with byte instructions. An 8-bit device on an 8-bit bus may be located at an odd or even address, and its internal registers may be assigned consecutive addresses (e.g., 1H, 2H, 3H). Assigning all-odd or all-even addresses, however, will simplify conversion to a 16-bit bus at a later date.

Table 3-7. Programmed I/O Bus Transfers

Bus Width:	8				16			
Instruction:	byte		word*		byte		word	
Device Address:	even	odd	even	odd	even	odd	even	odd*
Bus Cycles:	1	1	2	2	1	1	1	2

* not normally used

DMA Transfers

In addition to byte- and word-oriented programmed I/O, the 8089 can transfer blocks of data by direct memory access. A block may be transferred between any two addresses; memory-to-memory transfers are performed as easily as memory-to-port, port-to-memory or port-to-port exchanges. There is no limitation on the size of the block that can be transferred except that the block cannot exceed 64k bytes if byte count termination is used. A channel program typically prepares for a DMA transfer by writing commands to a device controller and initializing channel registers that are used during the transfer. No instructions are executed during the transfer, however, and very high throughput speeds can be achieved.

Preparing the Device Controller

Most controllers that can peform DMA transfers are quite flexible in that they can perform several different types of operations. For example, an 8271 Floppy Disk Controller can read a sector, write a sector, seek to track 0, etc. The controller typically has one or more internal registers that are "programmed" to perform a given operation. Often, certain registers will contain status information that can be read to determine if the controller is busy, if it has detected an error, etc.

An 8089 channel program views these device registers as a series of memory locations. The channel program typically places the device's base address in a pointer register and uses programmed I/O to communicate with the registers.

Some controllers start a DMA transfer immediately upon receiving the last of a series of parameters. If this type of controller is being used, the channel program instruction that sends the last parameter should *follow* the 8089 XFER instruction. (The XFER instruction places the channel in DMA mode after the next instruction; this is explained in more detail later in this section.)

Preparing the Channel

For a channel to perform a DMA transfer, it must be provided with information that describes the operation. The channel program provides this information by loading values into channel registers and, in one case, by executing a special instruction (see table 3-8).

Source and Destination Pointers. One register is loaded to point to the transfer source; the other points to the destination. A bit in the channel control register is set to indicate which register is the source pointer. If a register is pointed at a memory location, it should contain the address where the transfer is to begin — i.e., the lowest address in the buffer. The channel automatically increments a memory pointer as the transfer proceeds. If the tag bit selects the I/O space, the upper four bits of the register are ignored; if the tag selects the system space, all 20 bits are used. The source and destination may be located in the same or in different address spaces.

Translate Table Pointer. If the data is to be translated as it is transferred, GC should be pointed at the first (lowest-addressed) byte in a 256-byte translation table. The table may be located in either the system or I/O space, and GC

Table 3-8. DMA Transfer Control Information

Information	Register or Instruction	Required or Optional
Source Pointer	GA or GB	Required
Destination Pointer	GA or GB	Required
Translate Table Pointer	GC	Optional
Byte Count	BC	Optional
Mask/Compare Values	MC	Optional
Logical Bus Width	WID	Optional*
Channel Control	CC	Required

*Must be executed once following processor RESET.

Mnemonics © Intel, 1979

should be loaded by an instruction that sets or clears its tag bit as appropriate. The translate operation is only defined for byte data; source and destination logical bus widths must both be set to eight bits.

The channel translates a byte by treating it as an unsigned 8-bit binary number. This number is added to the content of register GC to form a memory address; GC is not altered by the operation. If GC points to the I/O space, its upper four bits are ignored in the operation. The byte at this address (which is in the translate table) is then fetched from memory, replacing the source byte. Figure 3-25 illustrates the translate process.

Byte Count. If the transfer is to be terminated on byte count— i.e., after a specific number of bytes have been transferred—the desired count should be loaded into register BC as an unsigned 16-bit number. The channel decrements BC as the transfer proceeds, whether or not byte count termination has been specified. There are cases (discussed later in this section) where the difference between BC's value before and after the transfer does not accurately reflect the number of bytes transferred to the destination.

Mask/Compare Values. If the transfer is to be terminated when a byte (possibly translated) is found equal or unequal to a search value, MC should be loaded as described in section 3.2. MC is not altered during the transfer. Normally, the logical destination bus width is set to eight bits when transferred data is being compared. If the logical destination width is 16 bits, only the low-order byte of each word is compared.

Logical Bus Width. The 8089 WID (logical bus width) instruction is used to set the logical width of the source and destination buses for a DMA transfer. Any bus whose physical width is eight bits can only have a logical width of eight bits. A 16-bit physical bus, however, can have a logical width of 8 or 16 bits; i.e., it can be used as either an 8-bit or 16-bit bus in any given transfer. Logical bus widths are set independently for each channel.

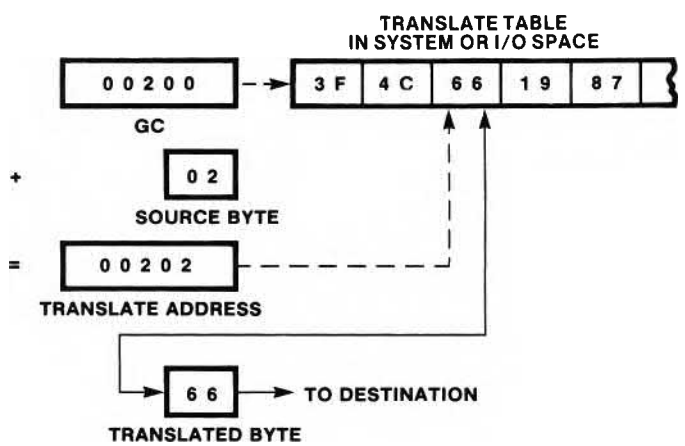


Figure 3-25. Translate Operation

Mnemonics © Intel, 1979

For a transfer to or from an I/O device on a 16-bit physical bus, the logical bus width should be set equal to the peripheral's width; i.e., 8 or 16 bits. Transfers to or from 16-bit memory will run at maximum speed if the logical bus width is set to 16 since the channel will fetch/store words. In the following cases, however, the logical width should be set to 8:

- the data is being translated,
- the data is being compared under mask, and the 16-bit memory is the destination of the transfer.

The WID instruction sets both logical widths and remains in effect until another WID instruction is executed. Following processor reset, the settings of the logical bus widths are unpredictable. Therefore, the WID instruction must be executed before the first DMA transfer.

Channel Control. The 16 bits of the CC register are divided into 10 fields that specify how the DMA transfer is to be executed (see figure 3-26). A channel program typically sets these fields by loading a word into the register.

The *function field* (bits 15-14) identifies the source and destination as memory or ports (I/O devices). During the transfer, the channel increments source/destination pointer registers that refer to memory so that the data will be placed in successive locations. Pointers that refer to I/O devices remain constant throughout the transfer.

The *translate field* (bit 13) controls data translation. If it is set, each incoming byte is translated using the table pointed to by register GC. Translate is defined only for byte transfers; the destination bus must have a logical width of eight.

The *synchronization field* (bits 12-11) specifies how the transfer is to be synchronized. Unsynchronized ("free running") transfers are typically used in memory-to-memory moves. The channel begins the next transfer cycle immediately upon completion of the current cycle (assuming it has the bus). Slow memories, which cannot run as fast as the channel, can extend bus cycles by signaling "not ready" to the 8284 Clock Generator, which will insert wait states into the bus cycle. A similar technique may be used with peripherals whose speed exceeds the channel's ability to execute a synchronized transfer: in effect, the peripheral synchronizes the transfer through the use of wait states. Chapter 4 discusses synchronization in more detail.

Source synchronization is typically selected when the source is an I/O device and the destination is memory. The I/O device starts the next transfer cycle by activating the channel's DRQ (DMA request) line. The channel then runs one transfer cycle and waits for the next DRQ.

Destination synchronization is most often used when the source is memory and the destination is an I/O device. Again, the I/O device controls the transfer frequency by signaling on DRQ when it is ready to receive the next byte or word.

The *source field* (bit 10) identifies register GA or GB as the source pointer (and the other as the destination pointer).

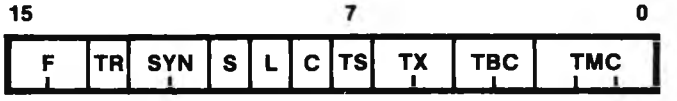
The *lock field* (bit 9) may be used to instruct the channel to assert the processor's bus lock ($\overline{\text{LOCK}}$) signal during the transfer. In a source-synchronized transfer, $\overline{\text{LOCK}}$ is active from the time the first DMA request is received until the channel enters the termination sequence. In a destination-synchronized transfer $\overline{\text{LOCK}}$ is active from the first fetch (which precedes the first DMA request) until the channel enters the termination sequence.

The *chain field* (bit 8) is not used during the transfer. As discussed previously, setting this bit raises channel program execution to priority level 1.

The *terminate on single transfer field* (bit 7) can be used to cause the channel to run one complete transfer cycle only—i.e., to transfer one byte or word and immediately resume channel program execution. When single transfer is specified, any other termination conditions are ignored. Single transfer termination can be used with low-speed devices, such as keyboards and communication lines, to translate and/or compare one byte as it transferred.

The *three low-order fields* in register CC instruct the channel when to terminate the transfer, assuming that single transfer has not been selected. Three termination conditions may be specified singly or in combination.

Mnemonics © Intel, 1979



F	FUNCTION
00	PORT TO PORT
01	MEMORY TO PORT
10	PORT TO MEMORY
11	MEMORY TO MEMORY

TR	TRANSLATE
0	NO TRANSLATE
1	TRANSLATE

SYN	SYNCHRONIZATION
00	NO SYNCHRONIZATION
01	SYNCHRONIZE ON SOURCE
10	SYNCHRONIZE ON DESTINATION
11	RESERVED BY INTEL

S	SOURCE
0	GA POINTS TO SOURCE
1	GB POINTS TO SOURCE

L	LOCK
0	NO LOCK
1	ACTUATE LOCK DURING TRANSFER

C	CHAIN
0	NO CHAINING
1	CHAINED: RAISE TB TO PRIORITY 1

TS	TERMINATE ON SINGLE TRANSFER
0	NO SINGLE TRANSFER TERMINATION
1	TERMINATE AFTER SINGLE TRANSFER

TX	TERMINATE ON EXTERNAL SIGNAL
00	NO EXTERNAL TERMINATION
01	TERMINATE ON EXT ACTIVE; OFFSET = 0
10	TERMINATE ON EXT ACTIVE; OFFSET = 4
11	TERMINATE ON EXT ACTIVE; OFFSET = 8

TBC	TERMINATE ON BYTE COUNT
00	NO BYTE COUNT TERMINATION
01	TERMINATE ON BC = 0; OFFSET = 0
10	TERMINATE ON BC = 0; OFFSET = 4
11	TERMINATE ON BC = 0; OFFSET = 8

TMC	TERMINATE ON MASKED COMPARE
000	NO MASK/COMPARE TERMINATION
001	TERMINATE ON MATCH; OFFSET = 0
010	TERMINATE ON MATCH; OFFSET = 4
011	TERMINATE ON MATCH; OFFSET = 8
100	(NO EFFECT)
101	TERMINATE ON NON-MATCH; OFFSET = 0
110	TERMINATE ON NON-MATCH; OFFSET = 4
111	TERMINATE ON NON-MATCH; OFFSET = 8

Figure 3-26. Channel Control Register Fields

External termination allows an I/O device (typically, the one that is synchronizing the transfer) to stop the transfer by activating the channel's EXT (external terminate) line. If byte count termination is selected, the channel will stop when BC=0. If masked compare termination is specified, the channel will stop the transfer when a byte is found that is equal or unequal (two options are available) to the low-order byte in MC as masked by MC's high-order byte. The byte that stops the termination is transferred. If translate has been specified, the translated byte is compared.

When a DMA transfer ends, the channel adds a value called the termination offset to the task pointer and resumes channel program execution at that point in the program. The termination offset may assume a value of 0, 4, or 8. Single transfer termination always results in a termination offset of 0. Figure 3-27 shows how the termination offsets can be used as indices into a three-element "jump table" that identifies the condition that caused the termination.

As an example of using the jump table, consider a case in which a transfer is to terminate when 80 bytes have been transferred or a linefeed character is detected, whichever occurs first. The program would load 80H into BC and 000AH into MC (ASCII line feed, no bits masked). The channel program could assign byte count termination an offset of 0 and masked compare termination an offset of 4. If the transfer is terminated by byte count (no linefeed is found), the instruction at location TP+0 will be executed first after the termination. If the linefeed is found before the byte count expires, the instruction at TP+4 will be executed first. The LJMP (long unconditional jump, see section 3.7) instruction is four bytes long and can be placed at TP+0 and TP+4 to cause the channel program to jump to a different routine, depending on how the transfer terminates.

If the transfer can only terminate in one way and that condition is assigned an offset of 0, there is no need for the jump table. Code which is to be unconditionally executed when the transfer ends can immediately follow the instruction after XFER. This is also the case when single transfer is specified (execution always resumes at TP+0).

It is possible, however, for two, or even three, termination conditions to arise at the same time. In

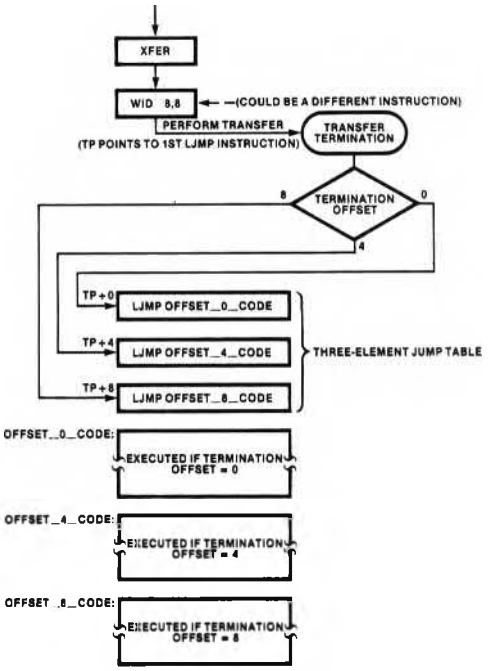


Figure 3-27. Termination Jump Table

the preceding example, this would occur if the 80th character were a linefeed. When multiple terminations occur simultaneously, the channel indicates that termination resulted from the condition with the largest offset value. In the preceding example, if byte count and search termination occur at the same time, the channel program resumes at TP+4.

Beginning the Transfer

The 8089 XFER (transfer) instruction puts the channel into DMA transfer mode after the *following instruction* has been executed. This technique gives the channel time to set itself up when it is used with device controllers, such as the 8271 Floppy Disk Controller, that begin transferring immediately upon receipt of the last in a series of parameters or commands. If the transfer is to or from such a device, the last parameter should be sent to the device after the XFER instruction. If this type of device is not being used, the instruction following XFER would

Mnemonics © Intel, 1979

typically send a "start" command to the controller. If a memory-to-memory transfer is being made, any instruction may follow XFER except one that alters GA, GB, or CC. The HLT instruction should normally not be coded after the XFER; doing so clears the channel's BUSY flag, but allows the DMA transfer to proceed.

DMA Transfer Cycle

A DMA transfer cycle is illustrated in figure 3-28; a complete transfer is a series of these cycles run until a termination condition is encountered. The figure is deliberately simplified to explain the general operation of a DMA transfer; in particular, the updating of the source and destination pointers (GA and GB) can be more complex than the figure indicates. Notice that it is possible to start an unending transfer by not specifying a termination condition in CC or by specifying a condition that never occurs; it is the programmer's responsibility to ensure that the transfer eventually stops.

If the transfer is source-synchronized, the channel waits until the synchronizing device activates the channel's DRQ line. The other channel is free to run during this idle period. The channel fetches a byte or a word, depending on the source address (contained in GA or GB) and the logical bus width. Table 3-9 shows how a channel performs the fetch/store sequence for all combinations of addresses and bus widths. If the destination is on a 16-bit logical bus and the source is on an 8-bit logical bus, and the transfer is to an even address, the channel fetches a second byte and assembles a word internally. During each fetch, the channel decrements BC according to whether a byte or word is obtained. Thus BC always indicates the number of bytes fetched.

The channel samples its EXT line after every bus cycle in the transfer. If EXT is recognized after the first of two scheduled fetches, the second fetch is not run. After the fetch sequence has been completed, the channel translates the data if this option is specified in CC.

If a word has been fetched or assembled, and bytes are to be stored (destination bus is eight bits or transfer is to an odd address), the channel disassembles the word into two bytes. If the transfer is destination-synchronized (only one

Table 3-9. DMA Transfer Assembly/Disassembly

Address (Source→ Destination)	Logical Bus Width (Source→Destination)			
	8→8	8→16	16→8	16→16
EVEN→EVEN	B→B	B/B→W	W→B/B	W→W
EVEN→ODD	B→B	B→B	W→B/B	W→B/B
ODD→EVEN	B→B	B/B→W	B→B	B/B→W
ODD→ODD	B→B	B→B	B→B	B→B

B= Byte Fetched or Stored in 1 Bus Cycle
W= Word Fetched or Stored in 1 Bus Cycle
B/B= 2 Bytes Fetched or Stored in 2 Bus Cycles

type of synchronization may be specified for a given transfer), the channel waits for DRQ before running a store cycle. It stores a word or the lower-addressed byte (which may be the only byte or the first of two bytes). Table 3-9 shows the possible combinations of even/odd addresses and logical bus widths that define the store cycle. Whenever stores are to memory on a 16-bit logical bus, the channel stores words, except that bytes may be stored on the first and last cycles.

The channel samples EXT again after the first store cycle and, if it is active, the channel prevents the second store cycle from running. If specified in the CC register, the low-order byte is compared to the value in MC. A "hit" on the comparison (equal or unequal, as indicated in CC) also prevents the second of two scheduled store cycles from running. In both of these cases, one byte has been "overfetched," and this is reflected in BC's value. It would be unusual, however, for a synchronizing device to issue EXT in the midst of a DMA cycle. Note also that EXT is valid only when DRQ is inactive. Chapter 4 covers the timing requirements for these two signals in detail.

GA and GB are updated next. Only memory pointers are incremented; pointers to I/O devices remain constant throughout the transfer.

If any termination condition has occurred during this cycle, the channel stops the transfer. It uses the content of the CC register to assign a value to the termination offset, to reflect the cause of the termination. The channel adds this offset to TP and resumes channel program execution at the location now addressed by TP. This offset will

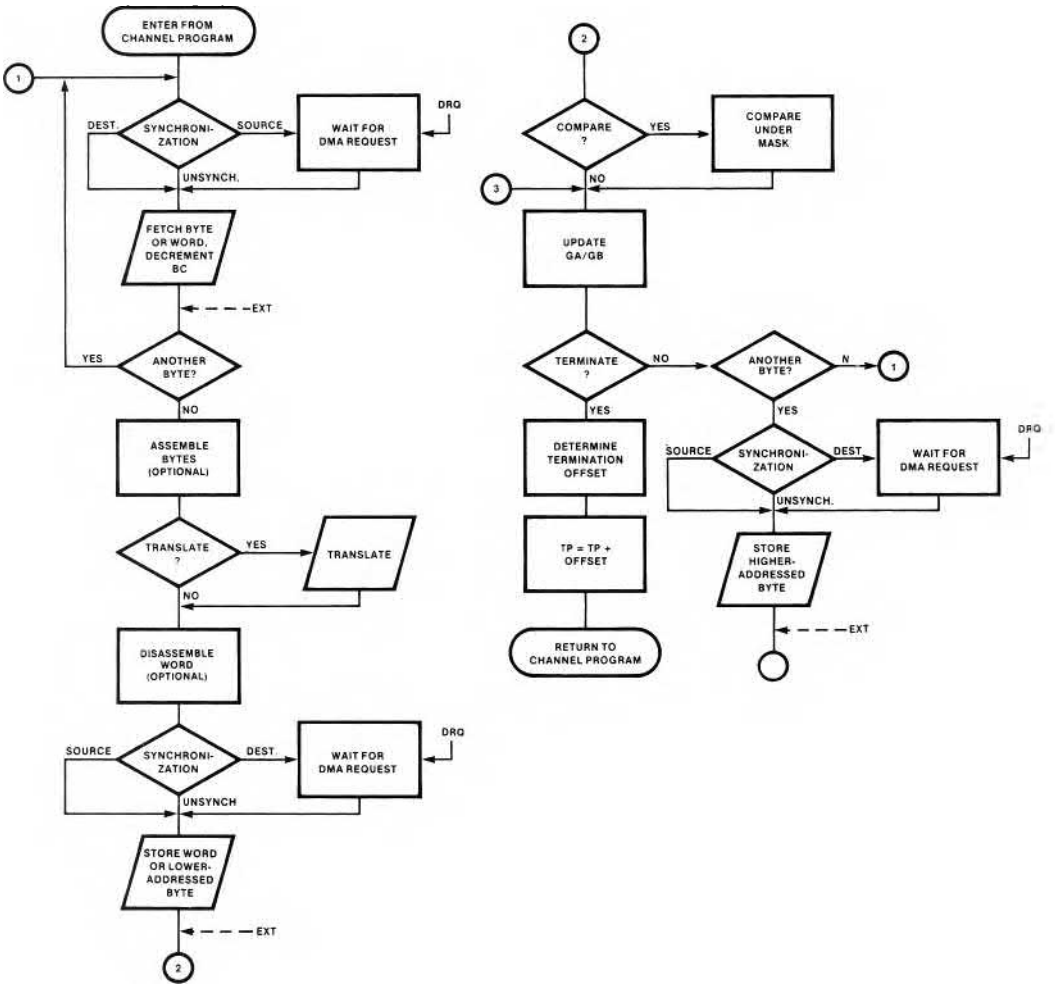


Figure 3-28. Simplified DMA Transfer Flowchart

always be zero, four, or eight bytes past the end of the instruction following the XFER instruction.

If no termination condition is detected and another byte remains to be stored, the channel stores this byte, waiting for DRQ if necessary, and updates the source and destination pointers. After the store, it again checks for termination.

Following the Transfer

A DMA transfer updates register BC, register GA (if it points to memory), and register GB (if it points to memory). If the original contents of these registers are needed following the transfer, the contents should be saved in memory prior to executing the XFER instruction.

Mnemonics © Intel, 1979

A program may determine the address of the last byte stored by a DMA transfer by inspecting the pointer registers as shown in table 3-10. The number of bytes stored is equal to:

last_byte_address — first_byte_address + 1.

For port-to-port transfers, the number of bytes transferred can be determined by subtracting the final value of BC from its original value provided that:

- the original BC > final BC,
- a transfer cycle is not "chopped off" before it completes by a masked compare or external termination.

In general, programs should not use the contents of GA, GB and BC following a transfer except as noted above and in table 3-10. This is because the contents of the registers are affected by numerous conditions, particularly when the transfer is terminated by EXT. In particular, when a program is performing a sequence of transfers, it should reload these registers before each transfer.

3.5 Multiprocessing Features

The 8089 shares the multiprocessing facilities common to the 8086 family of processors. It has on-chip logic for arbitrating the use of the local bus with a CPU or another IOP; system bus arbitration is delegated to an 8289 Bus Arbiter. The 8089's TSL (test and set while locked) instruction enables it to share a resource, such as a buffer, with other processors by means of semaphore (see section 2.5 for a discussion of the use of semaphores to control access to shared resources). Finally, the 8089 can lock the system bus for the duration of a DMA transfer to ensure that the transfer completes without interference from other processors on the bus.

In the remote configuration, the 8089 is electrically compatible with Intel's Multibus™ multimaster bus design. This means that the power and convenience of 8089 I/O processing can be used in 8080- or 8085-based systems that implement the Multibus protocol or a superset of it. This includes single-board computers such as Intel's iSBC 80/20™ and iSBC 80/30™ boards. In addition, the IOP can access other iSBC board products such as memory and communications controllers.

Bus Arbitration

The 8089 shares its system bus with a CPU, and may also share its I/O bus with an IOP or another CPU. Only one processor at a time may drive a bus. When two (or more) processors want to use a shared bus, the system must provide an arbitration mechanism that will grant the bus to one of the processors. This section describes the bus arbitration facilities that may be used with the 8089 and covers their applicability to different IOP configurations.

Table 3-10. Address of Last Byte Stored

Termination	Source	Destination	Synchronization	Last Byte Stored
byte count	memory memory port	memory port memory	any any any	destination pointer[1] source pointer destination pointer
masked compare	memory memory port	memory port memory	any any any	destination pointer source pointer destination pointer
external	memory memory port	memory port memory	unsynchronized destination source	destination pointer source pointer[2] destination pointer

[1]Source pointer may also be used.
[2]If transfer is B/B→W, source pointer must be decremented by 1 to point to last byte transferred.

Request/Grant Line

When an 8089 is directly connected to another 8089, an 8086 or an 8088, the $\overline{RQ}/\overline{GT}$ (request/grant) lines built into all of these processors are used to arbitrate use of a local bus. In the local mode, $\overline{RQ}/\overline{GT}$ is used to control access to both the system and the I/O bus.

As discussed in section 2.6, the CPU's request/grant lines ($\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$) operate as follows:

- an external processor sends a pulse to the CPU to request use of the bus;
- the CPU finishes its current bus cycle, if one is in progress, and sends a pulse to the processor to indicate that it has been granted the bus; and
- when the external processor is finished with the bus, it sends a final pulse to the CPU, to indicate that it is releasing the bus.

The 8089's request/grant circuit can operate in two modes; the mode is selected when the IOP is initialized (see section 3.6). Mode 0 is compatible with the 8086/8088 request/grant circuit and must be specified when the 8089's $\overline{RQ}/\overline{GT}$ line is connected to $\overline{RQ}/\overline{GT0}$ or $\overline{RQ}/\overline{GT1}$ of one of these CPUs. Mode 0 *may* be specified when $\overline{RQ}/\overline{GT}$ of one 8089 is tied to $\overline{RQ}/\overline{GT}$ of another 8089. When mode 0 is used with a CPU, the CPU is designated the master, and the IOP is designated a slave. When mode 0 is used with another IOP, one IOP is the master, and the other is the slave. Master/slave designation also is made at initialization time as discussed in section 3.6. The master has the bus when the system is initialized and keeps the bus until it is requested by the slave. When the slave requests the bus, the master grants it if the master is idle. In this sense, the CPU becomes idle at the end of the current bus cycle. An IOP master, on the other hand, does not become idle until both channels have halted program execution or are waiting for DMA requests. Once granted the bus, the slave (always an IOP) uses it until both channels are idle, and then releases it to the master. In mode 0, the master has no way of requesting the slave to return the bus.

Mode 1 operation of the request/grant lines may only be used to arbitrate use of a private I/O bus between two IOPs. In this case, one IOP is designated the master, and the other is designated the slave. However, the only difference between a master and a slave running in mode 1 is that the master has the bus at initialization time. Both processors may request the bus from each other at any time. The processor that has the bus will grant it to the requester as soon as one of the following occurs on either channel:

- an unchained channel program instruction is completed, or
- a channel goes idle due to a program halt or the completion of a synchronized transfer cycle (the channel waits for a DMA request).

Execution of a chained channel program, a DMA termination sequence, a channel attention sequence, or a synchronized DMA transfer (i.e., a high-priority operation) on either channel prevents the IOP from granting the bus to the requesting IOP.

The handshaking sequence in mode 1 is:

- the requesting processor pulses once on $\overline{RQ}/\overline{GT}$;
- the processor with the bus grants it by pulsing once; and
- if the processor granting the bus wants it back immediately (for example, to fetch the next instruction), it will pulse $\overline{RQ}/\overline{GT}$ again, two clocks after the grant pulse.

The fundamental difference between the two modes is the frequency with which the bus can be switched between the two processors when both are active. In mode 0, the processor that has the bus will tend to keep it for relatively long periods if it is executing a channel program. Mode 1 in effect places unchained channel programs at a lower priority since the processor will give up the bus at the end of the next instruction. Therefore, when both processors are running channel programs or synchronized DMA, they will share the bus more or less equally. When a processor changes to what would typically be considered a higher-priority activity such as chained program execution or DMA termination, it will generally be able to obtain the bus quickly and keep the bus for the duration of the more critical activity.

8289 Bus Arbiter

When an IOP is configured remotely, an 8289 Bus Arbiter is used to control its access to the shared system bus (the CPU also has its own 8289). In a remote cluster of two IOPs or an IOP and a CPU, one 8289 controls access to the system bus for both processors in the cluster. The 8289 has several operating modes; when used with an 8089, the 8289 is usually strapped in its IOB (I/O Peripheral Bus) mode.

The 8289 monitors the IOP's status lines. When these indicate that the IOP needs a cycle on the system bus, and the IOP does not presently have the bus, the 8289 activates a bus request signal. This signal, along with the bus request lines of other 8289s on the same bus, can be routed to an external priority-resolving circuit. At the end of the current bus cycle, this circuit grants the bus to the requesting 8289 with the highest priority. Several different prioritizing techniques may be used; in a typical system, an IOP would have higher bus priority than a CPU. If the 8289 does not obtain the bus for its processor, it makes the bus appear "not ready" as if a slow memory were being accessed. The processor's clock generator responds to the "not ready" condition by inserting wait states into the IOP's bus cycle, thereby extending the cycle until the bus is acquired.

Bus Arbitration for IOP Configurations

When the CPU initializes an IOP, it must inform the IOP whether it is a master or a slave, and which request/grant mode is to be used. This section covers the requirements and options available for each IOP configuration; section 3.6 describes how the information is communicated at initialization time.

Table 3-11 summarizes the bus arbitration requirements and options by IOP configuration. In the local configuration, all bus arbitration is performed by the request/grant lines without additional hardware. One IOP may be connected to each of the CPU's $\overline{RQ}/\overline{GT}$ lines. The IOP connected to $\overline{RQ}/\overline{GT0}$ will obtain the bus if both processors make simultaneous requests.

Since a single IOP in a remote configuration does not use $\overline{RQ}/\overline{GT}$, its mode may be set to 0 or 1 without affect. The single remote IOP, however, must be initialized as a master. If two remote IOPs share an I/O bus, one must be a master and the other a slave; both must be initialized to use the same request/grant mode. Normally, mode 1 will be selected for its improved responsiveness, and the designation of master will be arbitrary. If one IOP must have the I/O bus when the system comes up, it should be initialized as the master.

When a remote IOP shares its I/O bus with a local CPU, it must be a slave and must use request/grant mode 0.

Bus Load Limit

A locally configured IOP effectively has higher bus priority than the CPU since the CPU will grant the bus upon request from the IOP. One or two local IOPs can potentially monopolize the bus at the expense of the CPU. Of course, if the IOP activities are time-critical, this is exactly what should happen. On the other hand, there may be low-priority channel programs that have less demanding performance requirements.

In such cases, the CPU may set a CCW bit called bus load limit to constrain the channel's use of the bus during normal (unchained) channel program

Table 3-11. Bus Arbitration Requirements and Options

IOP	Local		Remote		Remote With Local CPU	
	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode
IOP1	Slave	0	Master	0 or 1	Slave	0
IOP2	Slave	0	Slave	Same as Master	N/A	N/A

execution. When this bit is set, the channel decrements a 7-bit counter from 7F (127) to 0H with each instruction executed. Since the counter is decremented once per clock period, the channel waits a minimum of 128 clock cycles before it executes the next instruction. By forcing the execution time of all instructions to 128 clocks, the use of the bus is reduced to between 3 and 25 percent of the available bus cycles.

Setting the bus load limit effectively enables a CPU to slow the execution of a normal channel program, thus freeing up bus cycles. This is of most use in local configurations, but also may be effective in remote configurations, particularly when channel programs are executed from system memory. Bus load limit has no effect on chained channel programs, DMA transfers, DMA termination, or channel attention sequences.

Bus Lock

Like the 8086 and 8088, the 8089 has a $\overline{\text{LOCK}}$ (bus lock) signal which can be activated by software. The $\overline{\text{LOCK}}$ output is normally connected to the $\overline{\text{LOCK}}$ input of an 8289 Bus Arbiter. When $\overline{\text{LOCK}}$ is active, the bus arbiter will not release the bus to another processor regardless of its priority. A channel automatically locks the bus during execution of the TSL (test and set while locked) instruction and may lock the bus for the duration of a DMA transfer.

If bit 9 of register CC is set, the 8089 activates its $\overline{\text{LOCK}}$ output during a DMA transfer on that channel. If the transfer is synchronized, $\overline{\text{LOCK}}$ is active from the time that the first DRQ is recognized. If the transfer is unsynchronized, $\overline{\text{LOCK}}$ is active throughout the entire transfer (there are no idle periods in an unsynchronized transfer). $\overline{\text{LOCK}}$ goes inactive when the channel begins the DMA termination sequence.

A locked transfer ensures that the transfer will be completed in the shortest possible time and that the transferring channel has exclusive use of the bus. Once the channel obtains the bus and starts a locked transfer, the channel, in effect, becomes the highest-priority processor on that bus.

The 8089 TSL (test and set while locked) instruction can be used to implement a semaphore. (See section 2.5 for a discussion of how a semaphore may be used to control the access of multiple processors to a shared resource.) The instruction activates $\overline{\text{LOCK}}$ and inspects the value of a byte in memory. If the value of the byte is 0H, it is changed (set) to a value specified in the instruction and the following instruction is executed. If the byte does not contain 0H, control is transferred to another location specified in the instruction. The bus is locked from the time the byte is read until it is either written or control is transferred to ensure that another processor does not access the variable after TSL has read it, but before it has updated it (i.e., between bus cycles). The following line of code will repeatedly test a semaphore pointed to by GA until it is found to contain zero:

```
TEST_FLAG: TSL [GA], 0FFH, TEST_FLAG
```

When the semaphore is found to be zero, it is set to FFH and the program continues with the next instruction.

3.6 Processor Control and Monitoring

This section focuses on IOP/CPU interaction, i.e., how the CPU initializes the IOP and subsequently sends commands to channels, and how the channels may interrupt the CPU. It also covers the channels' DMA control signals and the status signals that external devices can use to monitor IOP activities.

Initialization

Before the 8089 channels can be dispatched to perform I/O tasks, the IOP must be initialized. The initialization sequence (figure 3-29) provides the IOP with a definition of the system environment: physical bus widths, request/grant mode, and the location of the channel control block.

The sequence begins when the IOP's RESET line is activated. This halts any operation in progress, but does not affect any registers. Upon the first

Mnemonics © Intel, 1979

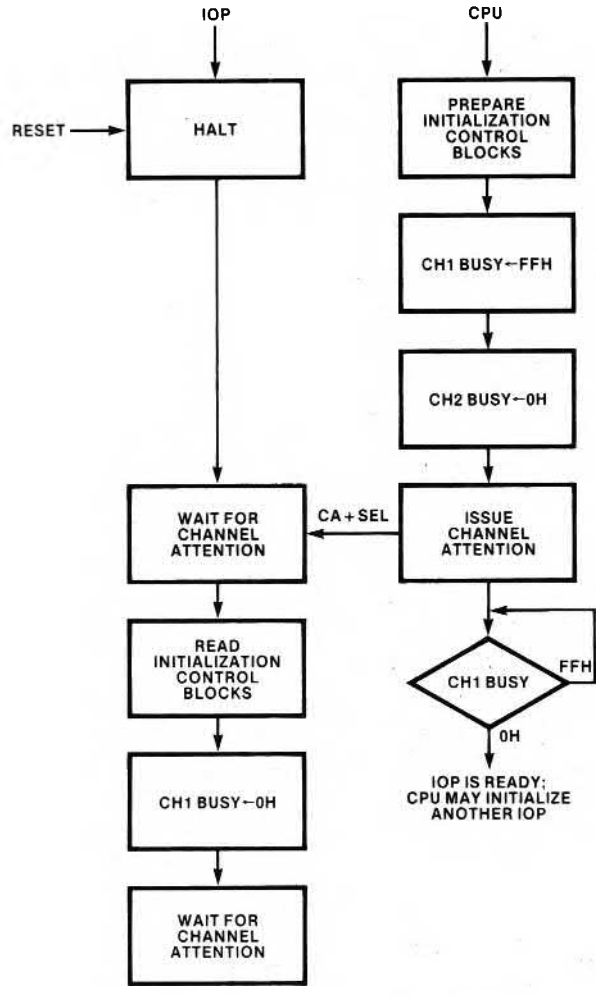


Figure 3-29. Initialization Sequence

RESET after power-up, the content of all IOP registers is undefined. Register contents are preserved if the IOP is subsequently RESET, except that RESET always clears the chain bit in register CC.

The IOP initializes itself by reading information from initialization control blocks located in the system space (see figure 3-30). The three blocks are the SCP (system configuration pointer), SCB (system configuration block) and the CB (channel control block). The CB is normally RAM-based; the SCP and the SCB may be in RAM or ROM. It is the CPU's responsibility to properly setup the control blocks.

The CPU starts the initialization sequence by issuing a channel attention to channel 1 (SEL low) or to channel 2 (SEL high). The CPU typically accesses the channels as two consecutive addresses in its I/O or memory space. An OUT instruction (for an I/O-mapped IOP) or a memory reference instruction (such as MOV) then issues the channel attention.

Mnemonics © Intel, 1979

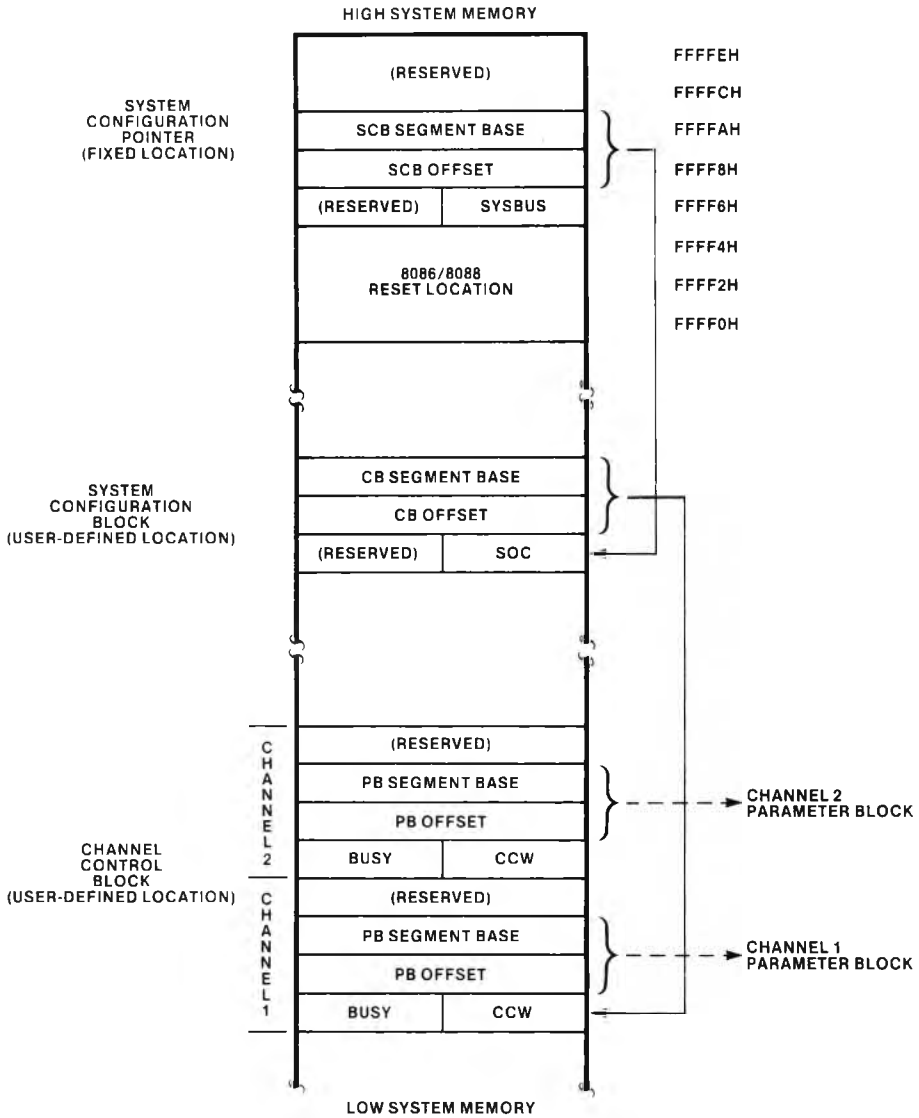


Figure 3-30. Initialization Control Blocks

If channel 1 is selected (SEL=low), the IOP considers itself a master (as discussed in section 3.5). If channel 2 is selected (SEL=high), the IOP operates as a slave. The IOP ignores, and does not latch, any subsequent channel attentions that occur during initialization.

If the IOP is a master, it assumes that it has the bus immediately. If it is a slave, it pulses $\overline{RQ}/\overline{GT}$ to request the bus from the CPU (local configuration) or the other IOP (remote configuration). When the IOP has obtained the bus, it assumes that the system bus is eight bits wide and reads the

SYSBUS field (figure 3-31) from location FFFF6H in system memory. This byte tells the IOP the actual physical width of the system bus; all subsequent accesses take advantage of a 16-bit bus if it is available; i.e., even-addressed words are fetched in single bus cycles. It is therefore advantageous to word-align the control blocks.

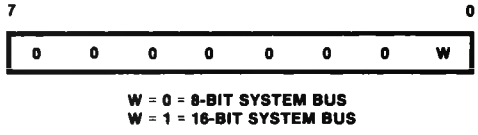


Figure 3-31. SYSBUS Encoding

Next, the IOP reads the SCB address located at FFFF8H. This is a standard doubleword pointer, and the IOP constructs a 20-bit physical address from it by shifting the segment base left four bits and adding the offset word of the pointer.

Having obtained the SCB address, the IOP reads the SOC (system operation command). This byte (see figure 3-32) tells the IOP the request/grant mode and the width of the I/O bus.

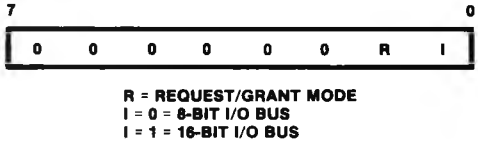


Figure 3-32. SOC Encoding

Then the IOP reads the doubleword pointer to the channel control block, converts the pointer into a 20-bit physical address, and stores it in an internal register. This register is not accessible to channel programs and is only loaded during initialization. The CB, therefore, cannot be moved during execution except by reinitializing the IOP.

After loading the address of the CB, the IOP clears the channel 1 BUSY flag to 0H. The other fields in the CB are used when a channel is dispatched and are not read or altered in the initialization sequence.

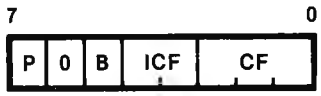
After the CPU has started the initialization sequence, it should monitor channel 1's BUSY flag in the CB to determine when the sequence has been completed. When the BUSY flag has been cleared, the CPU can dispatch either channel. It also can begin the initialization of another IOP. Since each IOP normally has a separate CB, the CPU must allocate the CB and update the pointer in the SCB before initializing the next IOP. Alternatively, multiple SCBs could be employed, each pointing to a different CB area. In this case the CPU would update the pointer in the SCP before initializing the next IOP. It follows from this that in multi-IOP systems, either the SCB or SCP, or both, must be RAM-based. When all IOPs have been initialized, the CPU may use RAM occupied by the SCB for another purpose.

Channel Commands

After initialization, any channel attention is interpreted as a command to channel 1 (SEL=low) or to channel 2 (SEL=high). As discussed in section 3.2, the channel attention, depending on the activities of both channels, may not be recognized immediately. The channel attention is latched, however, so that it will be serviced as soon as priorities allow.

When the channel recognizes the CA, it sets its BUSY flag in the CB to FFH. This does not prevent the CPU from issuing another CA, but provides status information only. In its response to a CA, the channel reads various control fields from system memory. It is the responsibility of the CPU to ensure that the appropriate fields are properly initialized before issuing the CA.

After setting its BUSY flag, the channel reads its CCW from the CB. It examines the command field (see figure 3-33) and executes the command encoded there by the CPU.



CF COMMAND FIELD
000 UPDATE PSW
001 START CHANNEL PROGRAM LOCATED IN I/O SPACE.
010 (RESERVED)
011 START CHANNEL PROGRAM LOCATED IN SYSTEM SPACE.
100 (RESERVED)
101 RESUME SUSPENDED CHANNEL OPERATION
110 SUSPEND CHANNEL OPERATION
111 HALT CHANNEL OPERATION

ICF INTERRUPT CONTROL FIELD
00 IGNORE, NO EFFECT ON INTERRUPTS.
01 REMOVE INTERRUPT REQUEST; INTERRUPT IS ACKNOWLEDGED.
10 ENABLE INTERRUPTS.
11 DISABLE INTERRUPTS.

B BUS LOAD LIMIT
0 NO BUS LOAD LIMIT
1 BUS LOAD LIMIT

P PRIORITY BIT

Figure 3-33. Channel Command Word Encoding

Figure 3-34 illustrates the channel's response to each type of command. Note that if CF contains a reserved value (010 or 100), the channel's response is unpredictable.

The CPU can use the "update PSW" command to alter the bus load limit and priority bits in the PSW (see figure 3-17) without otherwise affecting the channel. This command also allows the CPU to control interrupts originating in the channel; this topic is discussed in more detail later in this section.

The two "start program" commands differ only in their affect on the TP tag bit. If CF=001, the channel sets the tag to 1 to indicate that the program resides in the I/O space. If CF=011, the tag is cleared to 0, and the program is assumed to be in the system space. The channel converts the doubleword parameter block pointer to a 20-bit physical address and loads this into PP. It loads the doubleword task block (channel program) pointer into TP, updates the PSW as specified by the ICF, B and P fields of the CCW and starts the program with the instruction pointed to by TP.

The CPU may suspend a channel operation (either program execution or DMA transfer) by setting CF to 110. The channel saves its state (TP, its tag bit, and PSW) in the first two words of the parameter block (see figure 3-18 for format) and clears its BUSY flag to 0H. Note the following in regard to a suspended operation:

- The content of the doubleword pointer to the beginning of the channel program is replaced by the channel state save data. Therefore, a suspended operation may be resumed, but cannot be started from the beginning without recreating the doubleword pointer.

- TP is the only register saved by this operation. If another channel program is started on this channel, the other registers, including PP, are subject to being overwritten. In general, suspend is used to temporarily halt a channel, not to "interrupt" it with another program. Section 3.10 provides an example of a program that can be used to save another program's registers.

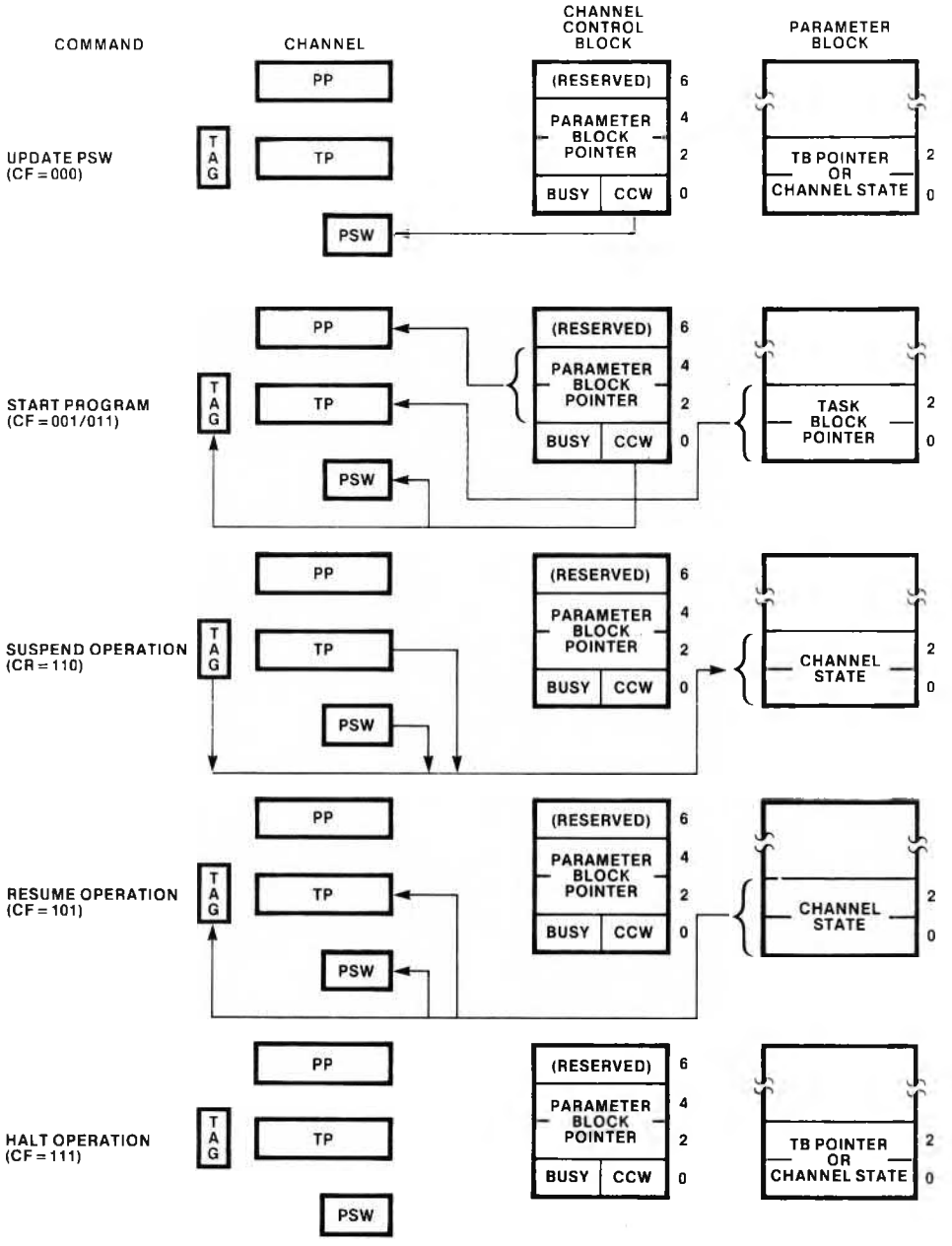


Figure 3-34. Channel Commands

- Suspending a DMA transfer does not affect any I/O devices (an I/O device will act as though the transfer is proceeding). The CPU must provide for conditions that may arise if, for example, a device requests a DMA transfer, but the channel does not acknowledge the request because it has been suspended. Similarly, an I/O device may be in a different condition when the operation is resumed.

A suspended operation may be resumed by setting CF to 101. This command causes the channel to reload TP, its tag bit, and the PSW from the first two words of PB. Resuming an operation that has not been suspended will give unpredictable results since the first two words of PB will not contain the required channel state data. A resume command does not affect any channel registers other than TP.

The CPU may abort a channel operation by issuing a "halt" command (CF=111). The channel clears its BUSY flag to 0H and then idles. Again, the CPU must be prepared for the effect aborting a DMA transfer may have on an I/O device.

DRQ (DMA Request)

The synchronizing device in a DMA transfer uses the DRQ line to indicate when it is ready to send or receive the next byte or word. The channel recognizes a signal on this line only during a DMA transfers, i.e., after the instruction following XFER has been executed and before a termination condition has occurred. The channels have separate DMA request lines (DRQ1 and DRQ2).

EXT (External Terminate)

An external device (typically the synchronizing device) can terminate a DMA transfer by signaling on this line. Each channel has its own external terminate line (EXT1 and EXT2). The channel stops the transfer as soon as the current fetch or store cycle is completed. An external terminate in an unsynchronized transfer could result in a loss of data, although this would not be a typical use of EXT. In a synchronized transfer, the synchronizing device will normally issue EXT instead of DRQ following the last transfer cycle. If EXT is activated during a transfer cycle, a fetched byte may not be stored as explained in section 3.4.

A channel does not recognize EXT if it is not performing a DMA transfer. If EXT1 and EXT2 are activated simultaneously, EXT1 is recognized first.

Interrupts

Each channel has a separate system interrupt line (SINTR1 and SINTR2). A channel program may generate a CPU interrupt request by executing a SINTR instruction. Whether this instruction actually activates the SINTR line, however, depends upon the state of the interrupt control bit (bit 3 of the PSW; see figure 3-17). If this bit is set, interrupts from the channel are enabled, and execution of the SINTR instruction activates SINTR. If the interrupt control bit is cleared, the SINTR instruction has no effect; interrupts from the channel are disabled.

The CPU can alter a channel's interrupt control bit by sending any command to the channel with the value of ICF (interrupt control field) in the CCW set to 10 (enable) or 11 (disable). Thus, the CPU can prevent interrupts from either channel.

Once activated, SINTR remains active until the CPU sends a channel command with ICF set to 01 (interrupt acknowledge). When the channel receives this command, it clears the interrupt service bit in the PSW (figure 3-17) and removes the interrupt request. Disabling interrupts also clears the interrupt service bit and lowers SINTR.

Status Lines

The IOP emits signals on the $\overline{S0}$-$\overline{S2}$ status lines to indicate to external devices the type of bus cycle the processor is starting. Table 3-12 shows the signals that are output for each type of cycle. These status lines are connected to an 8288 Bus Controller. The bus controller decodes these lines and outputs the signals that control components attached to the bus. The IOP indicates "instruction fetch" on these lines when it is reading and writing memory operands as well as when it is fet-

Mnemonics © Intel, 1979

ched instructions. In the remote configuration, an 8289 Bus Arbiter monitors the $\overline{S0}$-$\overline{S2}$ status lines to determine when a system bus access is required.

Table 3-12. Status Signals S0-S2

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Type of Bus Cycle
0	0	0	Instruction fetch from I/O space
0	0	1	Data fetch from I/O space
0	1	0	Data store to I/O space
0	1	1	(not used)
1	0	0	Instruction fetch from system space
1	0	1	Data fetch from system space
1	1	0	Data store to system space
1	1	1	Passive; no bus cycle run

Status lines S3-S6 indicate whether the bus cycle is DMA or non-DMA, and which channel is running the cycle (see table 3-13). Note that when the IOP is not running a bus cycle (e.g., when it is idle or when it is executing an internal cycle that does not use the bus), the status lines reflect the last bus cycle run.

Table 3-13. Status Signals S3-S6

S6	S5	S4	S3	Bus Cycle
1	1	0	0	DMA cycle on channel 1
1	1	0	1	DMA cycle on channel 2
1	1	1	0	Non-DMA cycle on channel 1
1	1	1	1	Non-DMA cycle on channel 2

3.7 Instruction Set

This section divides the IOP's 53 instructions into five functional categories:

1. data transfer,
2. arithmetic,
3. logic and bit manipulation,
4. program transfer,
5. processor control.

The description of each instruction in these categories explains how the instruction operates and how it may be used in channel programs. Instructions that perform essentially the same operation (e.g., ADD and ADDB, which add words and bytes respectively), are described together. A reference table at the end of the section lists every instruction alphabetically and provides execution time, encoded length, and sample ASM-89 coding for each permissable operand combination. For information on how the 8089 machine instructions are encoded in memory, see section 4.3.

In reading this section, it is important to recall that the instruction set does not differentiate between memory addresses and I/O device addresses. Instructions that are described as accepting byte and word memory operands may also be used to read and write I/O devices.

Data Transfer Instructions

These instructions move data between memory and channel registers. Traditional byte and word moves (including memory-to-memory) are available, as are special instructions that load addresses into pointer registers and update tag bits in the process.

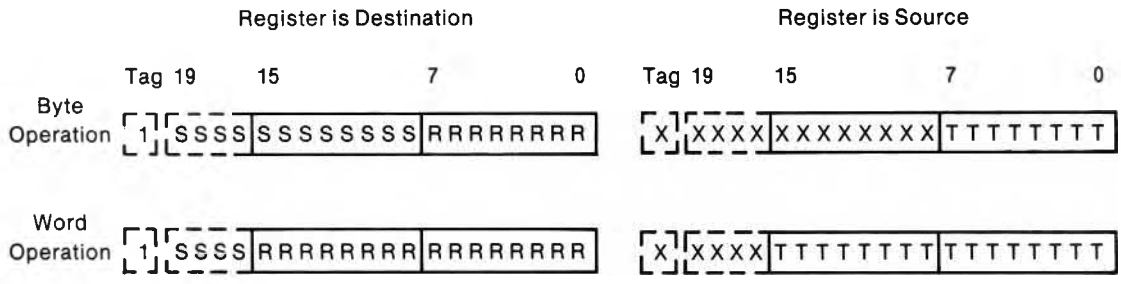
MOV *destination, source*

MOV transfers a byte or word from the source to the destination. Four instructions are provided:

MOV	Move Word Variable,
MOVB	Move Byte Variable,
MOVI	Move Word Immediate,
MOVBI	Move Byte Immediate.

Figure 3-35 shows how these instructions affect register operands. Notice that when a pointer register is specified as the destination of a MOV, its tag bit is unconditionally set to 1. MOV instructions are therefore used to load I/O space addresses into pointer registers.

Mnemonics © Intel, 1979



T = bit is transferred to destination operand
R = bit is replaced by source operand
S = bit is sign extension of high-order bit transferred
X = bit is ignored
1 = bit is unconditionally set

Figure 3-35. Register Operands in MOV Instructions

MOVP *destination, source*

MOVP (move pointer) transfers a physical address variable between a pointer register and memory. If the source is a pointer register, its content and tag bit are converted to a physical address pointer (see figure 3-23). If the source is a memory location, the three bytes are converted to a 20-bit physical address and a tag value, and are loaded into the pointer register and its tag bit. MOVP is typically used to save and restore pointer registers.

LPD *destination, source*

LPD (load pointer with doubleword) converts a doubleword pointer (see figure 3-22) to a 20-bit physical address and loads it into the destination, which must be a pointer register. The pointer register's tag bit is unconditionally cleared to 0, indicating a system address. Two instructions are provided:

LPD	Load Pointer With Doubleword Variable
LPDI	Load Pointer With Doubleword Immediate

An 8086 or 8088 can pass any address in its megabyte memory space to a channel program in the form of a doubleword pointer. The channel program can access the location by using LPD to load the location address into a pointer register.

Arithmetic Instructions

The arithmetic instructions interpret all operands as unsigned binary numbers of 8, 16 or 20 bits. Signed values may be represented in standard two's complement notation with the high-order bit representing the sign (0=positive, 1=negative). The processor, however, has no way of detecting an overflow into a sign bit so this possibility must be provided for in the user's software.

The 8089 performs arithmetic operations to 20 significant bits as follows. Byte and word operands are sign-extended to 20 bits (e.g., bit 7 of a byte operand is propagated through bits 8-19 of an internal register). Sign extension does not affect the magnitude of the operand. The operation is then performed, and the 20-bit result is

Mnemonics © Intel, 1979

returned to the destination operand. High-order bits are truncated as necessary to fit the result in the available space. A carry out of, or borrow into, the high-order bit of the result is not detected. However, if the destination is a register that is larger than the source operand, carries will be reflected in the upper register bits, up to the size of the register.

Figure 3-36 shows how the arithmetic instructions treat registers when they are specified as source and destination operands.

ADD *destination, source*

The sum of the two operands replaces the destination operand. Four addition instructions are provided:

ADD	Add Word Variable
ADDB	Add Byte Variable
ADDI	Add Word Immediate
ADDBI	Add Byte Immediate

INC *destination*

The destination is incremented by 1. Two instructions are available:

INC	Increment Word
INCB	Increment Byte

DEC *destination*

The destination is decremented by 1. Word and byte instructions are provided:

DEC	Decrement Word
DECB	Decrement Byte

Logical and Bit Manipulation Instructions

The logical instructions include the boolean operators AND, OR and NOT. Two bit manipulation instructions are provided for setting or

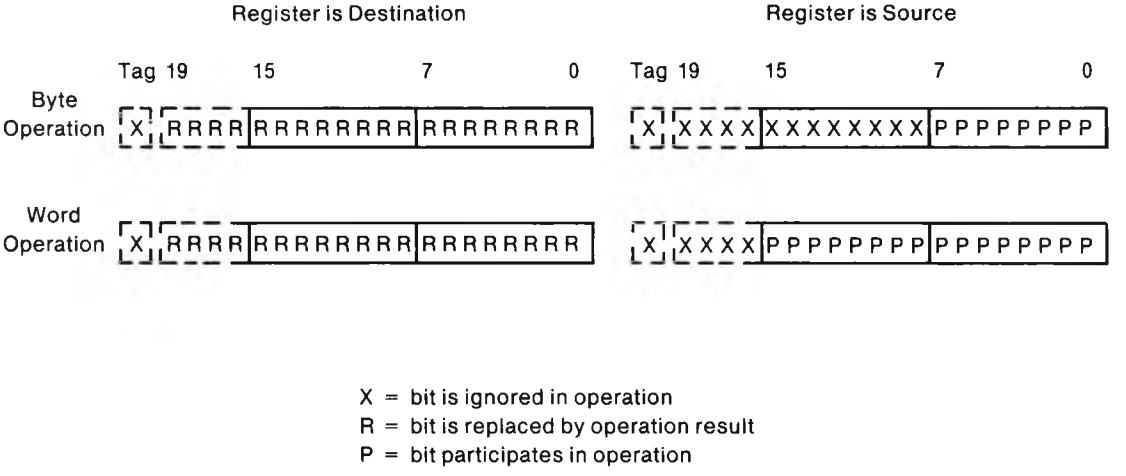


Figure 3-36. Register Operands in Arithmetic Instructions

clearing a single bit in memory or in an I/O device register. As shown in figure 3-37, the logical operations always leave the upper four bits of 20-bit destination registers undefined. These bits should not be assumed to contain reliable values or the same values from one operation to the next. Notice also that when a register is specified as the destination of a byte operation, bits 8-15 are overwritten by bit 7 of the result. Bits 8-15 can be preserved in AND and OR instructions by using word operations in which the upper byte of the source operand is FFH or 00H, respectively.

AND *destination, source*

The two operands are logically ANDed and the result replaces the destination operand. A bit in the result is set if the bits in the corresponding positions of the operands are both set, otherwise the result bit is cleared. The following AND instructions are available:

AND	Logical AND Word Variable
ANDB	Logical AND Byte Variable
ANDI	Logical AND Word Immediate
ANDBI	Logical AND Byte Immediate

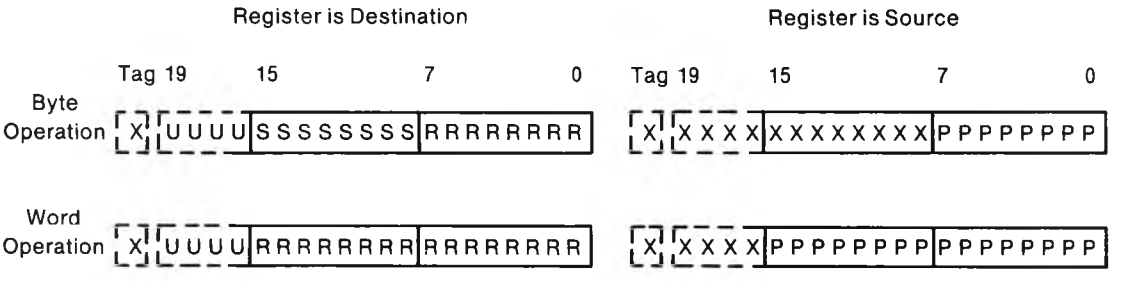
AND is useful when more than one bit of a device register must be cleared while leaving the remaining bits intact. For example, ANDing an 8-bit register with EEH only clears bits 0 and 4.

OR *destination, source*

The two operands are logically ORed, and the result replaces the destination operand. A bit in the result is set if either or both of the corresponding bits of the operands are set; if both operand bits are cleared, the result bit is cleared. Four types of OR instructions are provided:

OR	Logical OR Word Variable
ORB	Logical OR Byte Variable
ORI	Logical OR Word Immediate
ORBI	Logical OR Byte Immediate

OR can be used to selectively set multiple bits in a device register. For example, ORing an 8-bit register with 30H sets bits 4 and 5, but does not affect the other bits.



X = bit is ignored in operation
U = bit is undefined following operation
R = bit participates in operation and is replaced by result
S = bit is sign-extension of high-order result bit
P = bit participates in operation, but is unchanged

Figure 3-37. Register Operands in Logical Instructions

Mnemonics © Intel, 1979

NOT *destination/destination, source*

NOT inverts the bits of an operand. If a single operand is coded, the inverted result replaces the original value. If two operands are coded, the inverted bits of the source replace the destination value (which must be a register), but the source retains its original value. In addition to these two operand forms, separate mnemonics are provided for word and byte values:

NOT	Logical NOT Word
NOTB	Logical NOT Byte

NOT followed by INC will negate (create the two's complement of) a positive number.

SETB *destination, bit-select*

The bit-select operand specifies one bit in the destination, which must be a memory byte, that is unconditionally set to 1. A bit-select value of 0 specifies the low-order bit of the destination while the high-order bit is set if bit-select is 7. SETB is handy for setting a single bit in an 8-bit device register.

CLR *destination, bit-select*

CLR operates exactly like SETB except that the selected bit is unconditionally cleared to 0.

Program Transfer Instructions

Register TP controls the sequence in which channel program instructions are executed. As each instruction is executed, the length of the instruction is added to TP so that it points to the next sequential instruction. The program transfer instructions can alter this sequential execution by adding a signed displacement value to TP. The displacement is contained in the program transfer instruction and may be either 8 or 16 bits long. The displacement is encoded in two's complement notation, and the high-order bit indicates the sign (0=positive displacement, 1=negative displacement). An 8-bit displacement may cause a transfer to a location in the range −128 through +127 bytes from the end of the transfer instruction, while a 16-bit displacement can transfer to any location within −32,768 through +32,767 bytes. An instruction containing an 8-bit displacement is called a short transfer and an instruction containing a 16-bit displacement is called a long transfer.

The program transfer instructions have alternate mnemonics. If the mnemonic begins with the letter "L," the transfer is long, and the distance to the transfer target is expressed as a 16-bit displacement regardless of how far away the target is located. If the mnemonic does not begin with "L," the ASM-89 assembler may build a short or long displacement according to rules discussed in section 3.9.

The "self-relative" addressing technique used by program transfer instructions has two important consequences. First, it promotes position-independent code, i.e., code that can be moved in memory and still execute correctly. The only restriction here is that the entire program must be moved as a unit so that the distance between the transfer instruction and its target does not change. Second, the limited addressing range of these instructions must be kept in mind when designing large (over 32k bytes of code) channel programs.

CALL/LCALL *TPsave, target*

CALL invokes an out-of-line routine, saving the value of TP so that the subroutine can transfer back to the instruction following the CALL. The instruction stores TP and its tag bit in the TPsave operand, which must be a physical address variable, and then transfers to the target address formed by adding the target operand's displacement to TP. The subroutine can return to the instruction following the CALL by using a MOVP instruction to load TPsave back into TP.

Notice that the 8089's facilities for implementing subroutines, or procedures, is less sophisticated than its counterparts in the 8086/8088. The principal difference is that the 8089 does not have a built in stack mechanism. 8089 programs can implement a stack using a base register as a stack pointer. On the other hand, since channel programs are not subject to interrupts, a stack will not be required for most channel programs.

JMP/LJMP *target*

JMP causes an unconditional transfer (jump) to the target location. Since the task pointer is not saved, no return to the instruction following the JMP is implied.

JZ/LJZ *source, target*

JZ (jump if zero) effects a transfer to the target location if the source operand is zero; otherwise the instruction following JZ is executed. Word and byte values may be tested by alternate instructions:

JZ/LJZ	Jump/Long Jump if Word Zero
JZB/LJZB	Jump/Long Jump if Byte Zero

If the source operand is a register, only the low-order 16 bits are tested; any additional high-order bits in the register are ignored. To test the low-order byte of a register, clear bits 8-15 and then use the word form of the instruction.

JNZ/LJNZ *source, target*

JNZ operates exactly like JZ except that control is transferred to the target if the source operand does not contain all 0-bits. Word and byte sources may be tested using these mnemonics:

JNZ/LJNZ	Jump/Long Jump if Word Not Zero
JNZB/LJNZB	Jump/Long Jump if Byte Not Zero.

JMCE/LJMCE *source, target*

This instruction (jump if masked compare equal) effects a transfer to the target location if the source (a memory byte) is equal to the lower byte in register MC as masked by the upper byte in MC. Figure 3-15 illustrates how 0-bits in the upper half of MC cause the corresponding bits in the lower half of MC and the source operand to compare equal, regardless of their actual values. For example, if bits 8-15 of MC contain the value 01H, then the transfer will occur if bit 0 of the source and register MC are equal. This instruction is useful for testing multiple bits in 8-bit device registers.

JMCNE/LJMCNE *source, target*

This instruction causes a jump to the target location if the source is not equal to the mask/compare value in MC. It otherwise operates identically to JMCE.

JBT/LJBT *source, bit-select, target*

JBT (jump if bit true) tests a single bit in the source operand and jumps to the target if the bit is a 1. The source must be a byte in memory or in an I/O device register. The bit-select value may range from 0 through 7, with 0 specifying the low-order bit. This instruction may be used to test a bit in an 8-bit device register. If the target is the JBT instruction itself, the operation effectively becomes "wait until bit is 0."

JNBT/LJNBT *source, bit-select, target*

This instruction operates exactly like JBT, except that the transfer is made if the bit is not true, i.e., if the bit is 0.

Processor Control Instructions

These instructions enable channel programs to control IOP hardware facilities such as the $\overline{\text{LOCK}}$ and SINTR1-2 pins, logical bus width selection, and the initiation of a DMA transfer.

TSL *destination, set-value, target*

Figure 3-38 illustrates the operation of the TSL (test and set while locked) instruction. TSL can be used to implement a semaphore variable that controls access to a shared resource in a multiprocessor system (see section 2.5). If the target operand specifies the address of the TSL instruction, the instruction is repetively executed until the semaphore (destination) is found to contain zero. Thus the channel program does not proceed until the resource is free.

WID *source-width, dest-width*

WID (set logical bus widths) alters bits 0 and 1 of the PSW, thus specifying logical bus widths for a DMA transfer. The operands may be specified as

Mnemonics © Intel, 1979

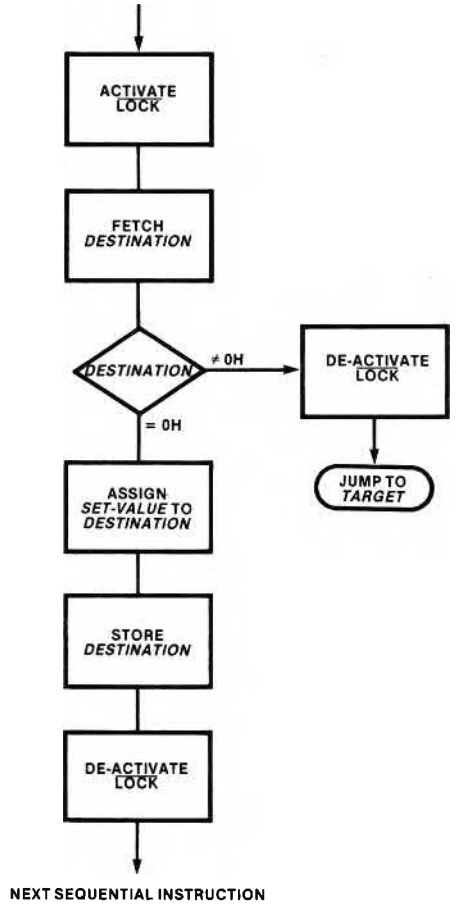


Figure 3-38. Operation of TSL Instruction

8 or 16 (bits), with the restriction that the logical width of a bus cannot exceed its physical width. The logical bus widths are undefined following a processor RESET; therefore the WID instruction must be executed before the first transfer. Thereafter the logical widths retain their values until the next WID instruction or processor RESET.

XFER *(no operands)*

XFER (enter DMA transfer mode after following instruction) prepares the channel for a DMA transfer operation. In a synchronized transfer, the instruction following XFER may ready the synchronizing device (e.g., send a "start" command or the last of a series of parameters). Any instruction, including NOP and WID, may follow XFER, except an instruction that alters GA, GB or GC.

SINTR *(no operands)*

This instruction sets the interrupt service bit in the PSW and activates the channel's SINTR line if the interrupt control bit in the PSW is set. If the

interrupt control bit is cleared (interrupts from this channel are disabled), the interrupt service bit is set, but SINTR1-2 is not activated. A channel program may use this instruction to interrupt a CPU.

NOP *(no operands)*

This instruction consumes clock cycles but performs no operation. As such, it is useful in timing loops.

HLT *(no operands)*

This instruction concludes a channel program. The channel clears its BUSY flag and then idles.

Instruction Set Reference Information

Table 3-16 lists every 8089 instruction alphabetically by its ASM-89 mnemonic. The ASM-89 coding format is shown (see table 3-14 for an explanation of operand identifiers) along with the instruction name. For every combination of operand types (see table 3-15 for key), the instruction's execution time and its length in bytes, and a coding example are provided.

The instruction timing figures are the number of clock periods required to execute the instruction with the given combination of operands. At 5 MHz, one clock period is 200 ns; at 8 MHz a clock period is 125 ns. Two timings are provided when an instruction operates on a memory word. The first (lower) figure indicates execution time when the word is aligned on an even address and is accessed over a 16-bit bus. The second figure is for odd-addressed words on 16-bit buses and any word accessed via an 8-bit bus.

Instruction fetch time is shown in table 3-17 and should be added to the execution times shown in table 3-16 to determine how long a sequence of instructions will take to run. (Section 3.2 explains the effect of the instruction queue on 16-bit instruction fetches.) External delays such as bus arbitration, wait states and activity on the other channel will increase the elapsed time over the figures shown in tables 3-16 and 3-17. These delays are application dependent.

Table 3-14. Key to ASM-89 Operand Identifiers

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, arithmetic, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location, or immediate value that is used in the operation, but is not altered by the instruction.
target	program transfer	Location to which control is to be transferred.
TPsave	program transfer	A 24-bit memory location where the address of the next sequential instruction is to be saved.
bit-select	bit manipulation	Specification of a bit location within a byte; 0=least-significant (rightmost) bit, 7=most-significant (leftmost) bit.
set-value	TSL	Value to which destination is set if it is found 0.
source-width	WID	Logical width of source bus.
dest-width	WID	Logical width of destination bus.

Mnemonics © Intel, 1979

Table 3-15. Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	Any general register
ptr-reg	A pointer register
immed8	A constant in the range 0-FFH
immed16	A constant in the range 0-FFFFH
mem8	An 8-bit memory location (byte)
mem16	A 16-bit memory location (word)
mem24	A 24-bit memory location (physical address pointer)
mem32	A 32-bit memory location (doubleword pointer)
label	A label within −32,768 to +32,767 bytes of the end of the instruction
short-label	A label within −128 to +127 bytes of the end of the instruction
0-7	A constant in the range: 0-7
8/16	The constant 8 or the constant 16

Table 3-16. Instruction Set Reference Data

ADD destination, source		Add Word Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	ADD BC, [GA].LENGTH
mem16, register	16/26	2-3	ADD [GB], GC

ADDB destination, source		Add Byte Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	ADDB GC, [GA].N_CHARS
mem8, register	16	2-3	ADDB [PP].ERRORS, MC

ADDBI destination, source		Add Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	ADDBI MC,10
mem8, immed8	16	3-4	ADDBI [PP+IX+].RECORDS, 2CH

ADDI destination, source		Add Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	ADDI GB, 0C25BH
mem16, immed16	16/26	4-5	ADDI [GB].POINTER, 5899

Table 3-16. Instruction Set Reference Data (Cont'd.)

AND destination, source		Logical AND Word Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	AND MC, [GA].FLAG_WORD
mem16, register	16/26	2-3	AND [GC].STATUS, BC

ANDB destination, source		Logical AND Byte Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	AND BC, [GC]
mem8, register	16	2-3	AND [GA+IX].RESULT, GA

ANDBI destination, source		Logical AND Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	GA, 01100000B
mem8, immed8	16	3-4	[GC+IX], 2CH

ANDI destination, source		Logical AND Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	IX, 0H
mem16, immed16	16/26	4-5	[GB+IX].TAB, 40H

CALL TPsave, target		Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	3-5	CALL [GC+IX].SAVE, GET_NEXT

CLR destination, bit select		Clear Bit To Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7	16	2-3	CLR [GA], 3

DEC destination		Decrement Word By 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
register	3	2	
mem16	16/26	2-3	DEC [PP].RETRY

Mnemonics © Intel, 1979

Table 3-16. Instruction Set Reference Data (Cont'd.)

DECB destination		Decrement Byte By 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	DECB [GA+IX+].TAB

HLT (no operands)		Halt Channel Program	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	11	2	HLT

INC destination		Increment Word by 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
register	3	2	INC GA
mem16	16/26	2-3	INC [GA].COUNT

INCB destination		Increment Byte by 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	INCB [GB].POINTER

JBT source, bit-select, target		Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JBT [GA].RESULT_REG, 3, DATA_VALID

JMCE source, target		Jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	3-5	JMCE [GB].FLAG, STOP_SEARCH

JMCNE source, target		Jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	3-5	JMCNE [GB+IX], NEXT_ITEM

JMP target		Jump Unconditionally	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	3-4	JMP READ_SECTOR

Table 3-16. Instruction Set Reference Data (Cont'd.)

JNBT source, bit-select, target		Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JNBT [GC], 3, RE_READ

JNZ source, target		Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JNZ BC, WRITE_LINE
mem16, label	12/16	3-5	JNZ [PP].NUM_CHARS, PUT_BYTE

JNZB source, target		Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JNZB [GA], MORE_DATA

JZ source, target		Jump if Word is Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JZ BC, NEXT_LINE
mem16, label	12/16	3-5	JZ [GC+IX].INDEX, BUF_EMPTY

JZB source, target		Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JZB [PP].LINES_LEFT, RETURN

LCALL TPsave, target		Long Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	4-5	LCALL [GC].RETURN_SAVE, INIT_8279

LJBT source, bit-select, target		Long Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJBT [GA].RESULT, 1, DATA_OK

LJMCE source, target		Long jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCE [GB], BYTE_FOUND

Mnemonics © Intel, 1979

Table 3-16. Instruction Set Reference Data (Cont'd.)

LJMCNE source, target		Long jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCNE [GC+IX+], SCAN_NEXT

LJMP target		Long Jump Unconditional	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	4	LJMP GET_CURSOR

LJNBT source, bit-select, target		Long Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJNBT [GC], 6, CRCC_ERROR

LJNZ source, target		Long Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJNZ BC, PARTIAL_XMIT
mem16, label	12/16	4-5	LJNZ [GA+IX].N_LEFT, PUT_DATA

LJNZB source, target		Long Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJNZB [GB+IX+].ITEM, BUMP_COUNT

LJZ source, target		Long Jump if Word Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJZ IX, FIRST_ELEMENT
mem16, label	12/16	4-5	LJZ [GB].XMIT_COUNT, NO_DATA

LJZB source, target		Long Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJZB [GA], RETURN_LINE

LPD destination, source		Load Pointer With Doubleword Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem32	20/28*	2-3	LPD GA, [PP].BUF_START

*20 clocks if operand is on even address; 28 if on odd address

Mnemonics © Intel, 1979

Table 3-16. Instruction Set Reference Data (Cont'd.)

LPDI destination, source		Load Pointer With Doubleword Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, immed32	12/16*	6	LPDI GB, DISK_ADDRESS

*12 clocks if instruction is on even address; 16 if on odd address

MOV destination, source		Move Word	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	8/12	2-3	MOV IX, [GC]
mem16, register	10/16	2-3	MOV [GA].COUNT, BC
mem16, mem16	18/28	4-6	MOV [GA].READING, [GB]

MOVB destination, source		Move Byte	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	8	2-3	MOVB BC, [PP].TRAN_COUNT
mem8, register	10	2-3	MOVB [PP].RETURN_CODE, GC
mem8, mem8	18	4-6	MOVB [GB+IX+], [GA+IX+]

MOVBI destination, source		Move Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	MOVBI MC, 'A'
mem8, immed8	12	3-4	MOVBI [PP].RESULT, 0

MOVI destination, source		Move Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	MOVI BC, 0
mem16, immed16	12/18	4-5	MOVI [GB], 0FFFFH

MOVP destination, source		Move Pointer	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem24	19/27*	2-3	MOVP TP, [GC+IX]
mem24, ptr-reg	16/22*	2-3	MOVP [GB].SAVE_ADDR, GC

*First figure is for operand on even address; second is for odd-addressed operand.

NOP (no operands)		No Operation	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	NOP

Mnemonics © Intel, 1979

Table 3-16. Instruction Set Reference Data (Cont'd.)

NOT destination/destination, source		Logical NOT Word	
Operands	**Clocks**	**Bytes**	**Coding Example**
register	3	2	NOT MC
mem16	16/26	2-3	NOT [GA].PARM
register, mem16	11/15	2-3	NOT BC, [GA+IX].LINES_LEFT

NOTB destination/destination, source		Logical NOT Byte	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	NOTB [GA].PARM_REG
register, mem8	11	2-3	NOTB IX, [GB].STATUS

OR destination, source		Logical OR Word	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	OR MC, [GC].MASK
mem16, register	16/26	2-3	OR [GC], BC

ORB destination, source		Logical OR Byte	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	ORB IX, [PP].POINTER
mem8, register	16	2-3	ORB [GA+IX+], GB

ORBI destination, source		Logical OR Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	ORBI IX, 00010001B
mem8, immed8	16	3-4	ORBI [GB].COMMAND, 0CH

ORI destination, source		Logical OR Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	ORI MC, 0FF0DH
mem16,immed16	16/26	4-5	ORI [GA], 1000H

SETB destination, bit-select		Set Bit to 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7	16	2-3	SETB [GA].PARM_REG, 2

SINTR (no operands)		Set Interrupt Service Bit	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	SINTR

Mnemonics © Intel, 1979

Table 3-16. Instruction Set Reference Data (Cont'd.)

TSL destination, set-value, target		Test and Set While Locked	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, immed8, short-label	14/16*	4-5	TSL [GA].FLAG, 0FFH, NOT_READY

*14 clocks if destination ≠ 0; 16 clocks if destination = 0

WID source-width, dest-width		Set Logical Bus Widths	
Operands	**Clocks**	**Bytes**	**Coding Example**
8/16, 8/16	4	2	WID 8, 8

XFER (no operands)		Enter DMA Transfer Mode After Next Instruction	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	XFER

Table 3-17. Instruction Fetch Timings (Clock Periods)

INSTRUCTION LENGTH (BYTES)	BUS WIDTH		
	8	16	
		(1)	(2)
2	14	7	11
3	18	14	11
4	22	14	15
5	26	18	15

(1) First byte of instruction is on an even address.

(2) First byte of instruction is on an odd address. Add 3 clocks if first byte is not in queue (e.g., first instruction following program transfer).

3.8 Addressing Modes

8089 instruction operands may reside in registers, in the instruction itself or in the system or I/O address spaces. Operands in the system and I/O spaces may be either memory locations or I/O device registers and may be addressed in four different ways. This section describes how the channel processes different types of operands and how it calculates addresses using its addressing modes. Section 3.9 describes the ASM-89 conventions that programmers use to specify these operands and addressing modes.

Register and Immediate Operands

Registers may be specified as source or destination operands in many instructions. Instructions that operate on registers are generally both shorter and faster than instructions that specify immediate or memory operands.

Immediate operands are data contained in instructions rather than in registers or in memory. The data may be either 8 or 16 bits in length. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Memory Addressing Modes

Whereas the channel has direct access to register and immediate operands, operands in the system and I/O space must be transferred to or from the IOP over the bus. To do this, the IOP must calculate the address of the operand, called its

Mnemonics © Intel, 1979

effective address (EA). The programmer may specify that an operand's address be calculated in any of four different ways; these are the 8089's memory addressing modes.

The Effective Address

An operand in the system space has a 20-bit effective address, and an operand in the I/O space has a 16-bit effective address. These addresses are unsigned numbers that represent the distance (in bytes) of the low-order byte of the operand from the beginning of the address space. Since the 8089 does not "see" the segmented structure of the system space that it may share with an 8086 or 8088, 8089 effective addresses are equivalent to 8086/8088 physical addresses.

All memory addressing modes use the content of one of the pointer registers, and the state of that register's tag bit determines whether the operand lies in the system or the I/O space. If the operand is in the I/O space (tag = 1), bits 16-19 of the pointer register are ignored in the effective address calculation. Section 4.3 describes the two fields (AA and MM) in the encoded machine instruction that specify addressing mode and base (pointer) register.

Based Addressing

In based addressing (figure 3-39), the effective address is taken directly from the content of GA, GB, GC or PP. Using this addressing mode, one instruction may access different locations if the register is updated before the instruction executes. LPD, MOV, MOVP or arithmetic instructions might be used to change the value of the base register.

Offset Addressing

In this mode (figure 3-40) an 8-bit unsigned value contained in the instruction is added to the content of a base register to form the effective address. The offset mode provides a convenient way to address elements in structures (a parameter block is a typical example of a structure). As shown in figure 3-41, a base register can be pointed at the base (first element) in the structure, and then different offsets can be used to access the elements within the structure. By changing the base address, the same structure can be relocated elsewhere in memory.

Indexed Addressing

An indexed address is formed by adding the content of register IX (interpreted as an unsigned quantity) to a base register as shown in figure 3-42. Indexed addressing is often used to access

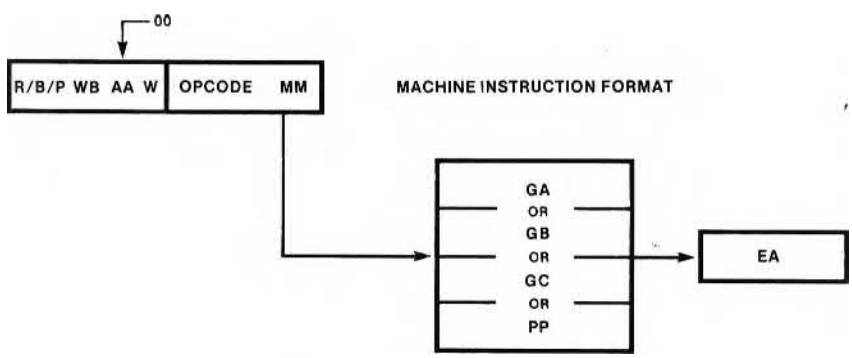


Figure 3-39. Based Addressing

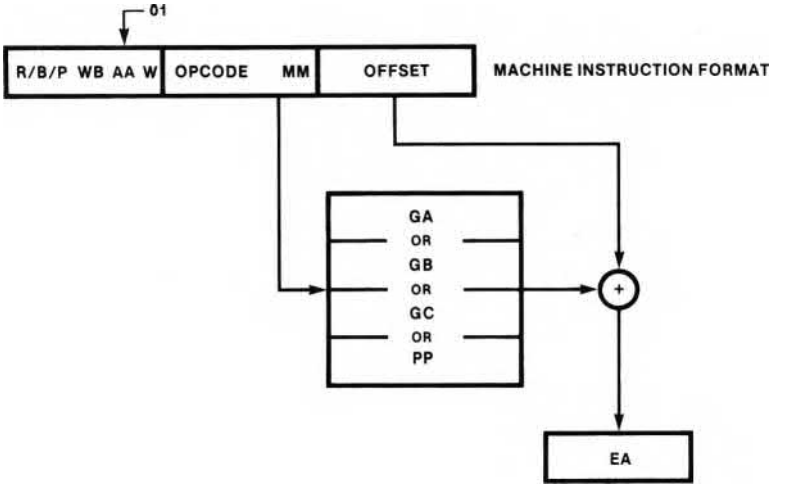


Figure 3-40. Offset Addressing

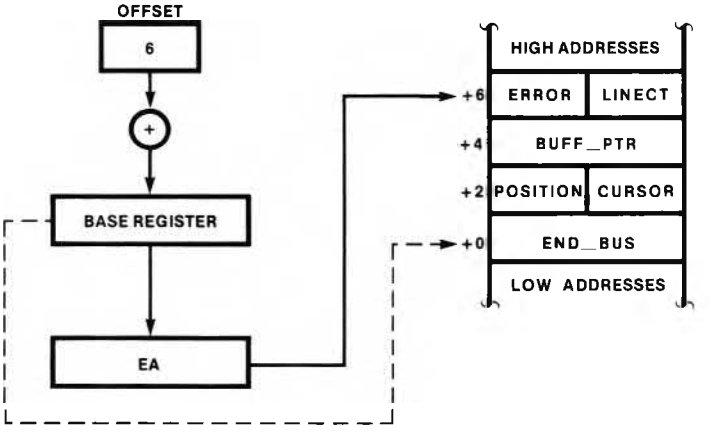


Figure 3-41. Accessing a Structure with Offset Addressing

array elements (see figure 3-43). A base register locates the beginning of the array and the value in IX selects one element, i.e., it acts as the array subscript. The *i*th element of a byte array is selected when IX contains $(i - 1)$. To access the *i*th element of a word array, IX should contain $((i - 1) * 2)$.

Indexed Auto-Increment Addressing

In this variation of indexed addressing, the effective address is formed by summing IX and a base register, and then IX is incremented automatically. (See figure 3-44.) The addition takes place

after the EA is calculated. IX is incremented by 1 for a byte operation, by 2 for a word operation and by 3 for a MOVP instruction. This addressing mode is very useful for "stepping through" successive elements of an array (e.g., a program loop that sums an array).

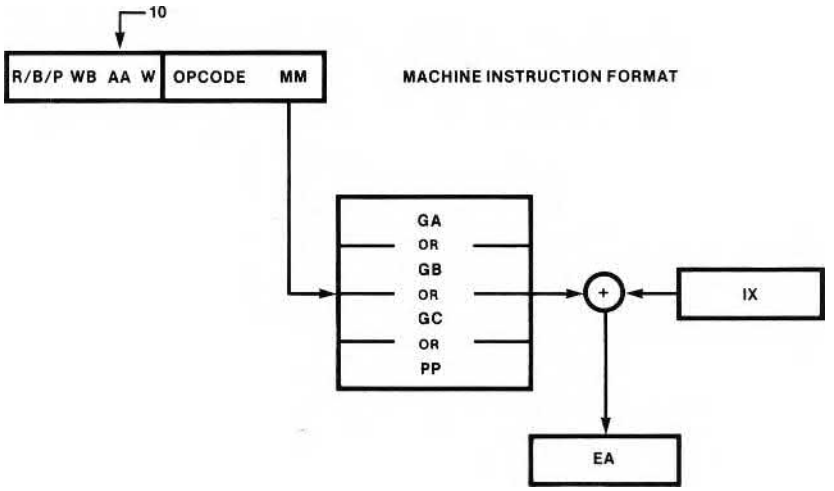


Figure 3-42. Indexed Addressing

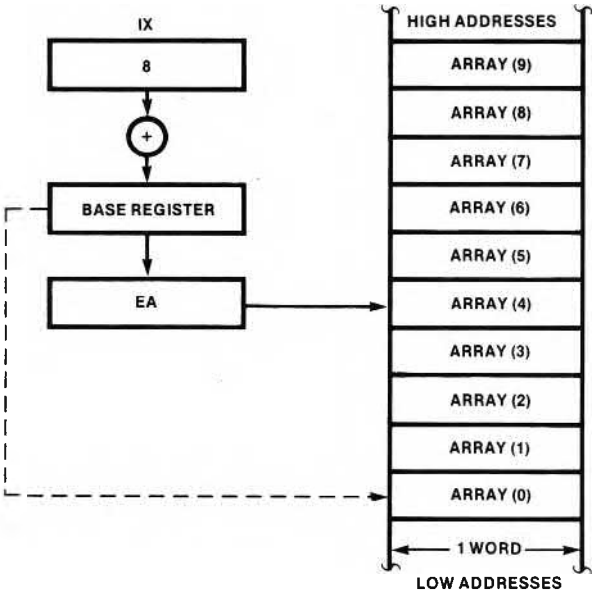


Figure 3-43. Accessing a Word Array with Indexed Addressing

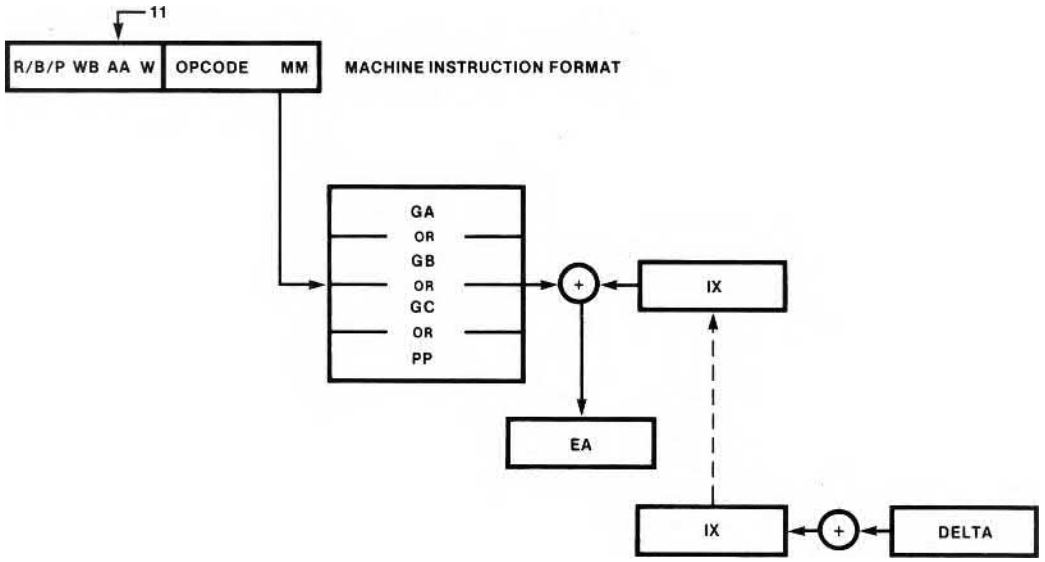


Figure 3-44. Indexed Auto-Increment Addressing

3.9 Programming Facilities

The compatibility of the 8089 with the 8086 and 8088 extends beyond the hardware interface. Comparing figure 3-45, with figure 2-45, one can see that, except for the translate step, the software development process is identical for both 8086/8088 and 8089 programs. The ASM-89 assembler produces a relocatable object module that is compatible with the 8086 family software development utilities LIB-86, LINK-86, LOC-86 and OH-86, described in section 2.9. All of these development tools run on an Intellec® 800 or Series II microcomputer development system.

This section surveys the facilities of the ASM-89 assembler and discusses how LINK-86 and LOC-86 can be used in 8089 software development. For a complete description of the 8089 assembly language, consult *8089 Assembly Language User's Guide*, Order No. 9800938, available from Intel's Literature Department.

ASM-89

The ASM-89 assembler reads a disk file containing 8089 assembly language statements, translates these statements into 8089 machine instructions, and writes the result into a second disk file. The assembly input is called a source module, and the principal output is a relocatable object module. The assembler also produces a file that lists the module and flags any errors detected during the assembly.

Statements

Statements are the building blocks of ASM-89 programs. Figure 3-46 shows several examples of ASM-89 statements. The ASM-89 assembler gives programmers considerable flexibility in formatting program statements. Variable names and labels (identifiers) may be up to 31 characters long, the underscore (_) character may be used to improve the readability of longer names (e.g.,

WAIT_UNTIL_READY). The component parts of statements (fields) need not be located at particular "columns" of the statement. Any number of blank characters may separate fields and multiple identifiers within the operand field. Long statements may be continued onto the next link by coding an ampersand (&) as the first character of the continued line.

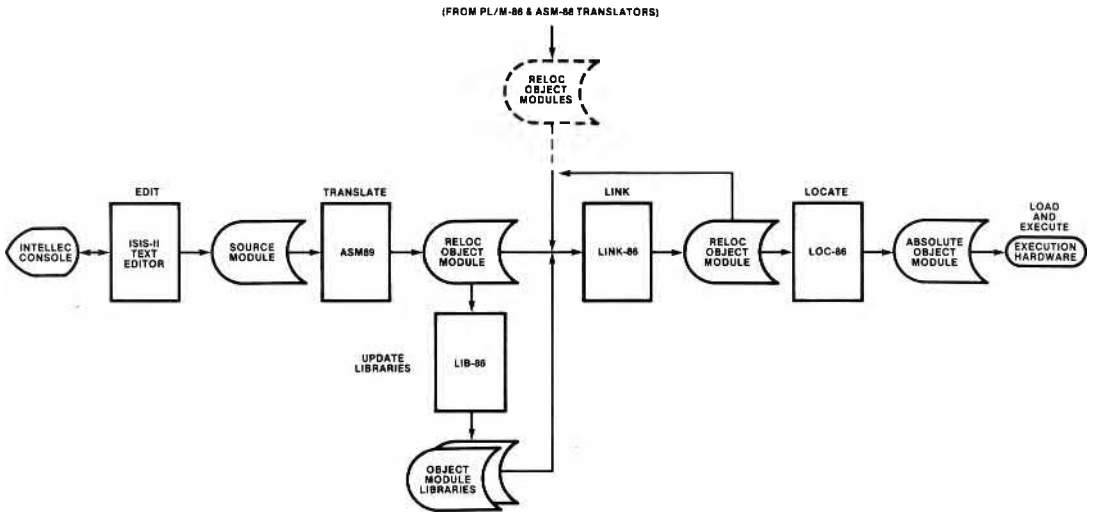


Figure 3-45. 8089 Software Development Process

```
; THIS STATEMENT CONTAINS A COMMENT FIELD ONLY
ADDI   BC,5                    ; TYPICAL ASM89 INSTRUCTION
  ADDI     BC,        5        ; NO "COLUMN" REQUIREMENTS
MOV      [GA].STATUS,
&        6                     ; A CONTINUED STATEMENT
SOURCE      EQU GA             ; A SIMPLE ASM89 DIRECTIVE
LINE_BUFFER_ADDRESS DD         ; A LONG IDENTIFIER
```

Figure 3-46. ASM-89 Statements

A statement whose first non-blank character is a semicolon is a comment statement. Comments have no affect on program execution and, in fact, are ignored by the ASM-89 assembler. Nevertheless, carefully selected comments are included in all well written ASM-89 programs. They summarize, annotate and clarify the logic of the program where the instructions are too "microscopic" to make the operation of the program self-evident.

An ASM-89 instruction statement (figure 3-47) directs the assembler to build an 8089 machine instruction. The optional label field assigns a symbolic identifier to the address where the instruction will be stored in memory. A labelled instruction can be the target of a program transfer; the transferring instruction specifies the label for its target operand. In figure 3-47 the labelled instruction conditionally transfers to itself; the program will loop on this one instruction as long as bit 3 of the byte addressed by [GA].STATUS is not true. The mnemonic field of an instruction statement specifies the type of 8089 machine instruction that the assembler is to build.

The operand field may contain no operands or one or more operands as required by the instruction. Multiple operands are separated by commas and, optionally, by blanks. Any instruction statement may contain a comment field (comment fields are initiated by a semicolon).

An ASM-89 directive statement (figure 3-48) does not produce an 8089 machine instruction. Rather, a directive gives the assembler information to use during the assembly. For example, the DS (define storage) directive in figure 3-48 tells the assembler to reserve 80 bytes of storage and to assign a symbolic identifier (INPUT_BUFFER) to the first (lowest-addressed) byte of this area. The ASM-89 assembler accepts 14 directives; the more commonly used directives are discussed in this section.

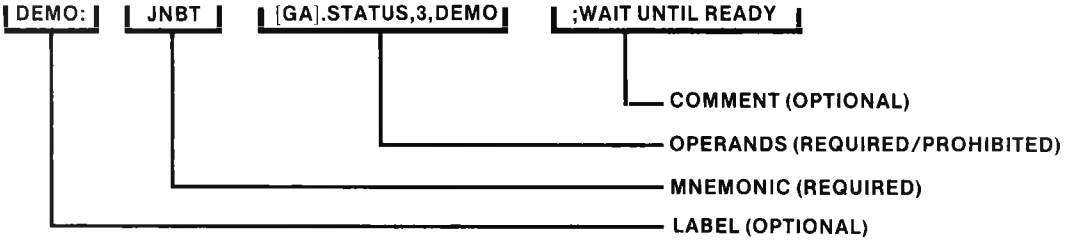


Figure 3-47. ASM-89 Instruction Format

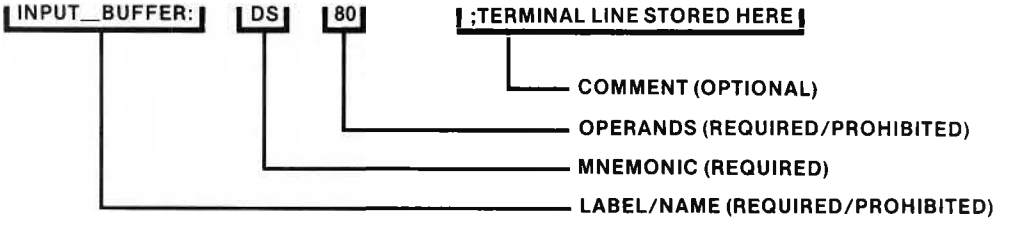


Figure 3-48. ASM-89 Directive Format

Mnemonics © Intel, 1979

The first field in a directive may be a label or a name; individual directives may require or prohibit names, while labels are optional for directives that accept them. A label ends in a colon like an instruction statement label. However, a directive label cannot be specified as the target of a program transfer. A name does not have a colon. The second field is the directive mnemonic, and the assembler distinguishes between instructions and directives by this field. Any operands required by the directive are written next; multiple operands are separated by commas and, optionally, by blanks. A comment may be included in any directive by beginning the text with a semicolon.

Constants

Binary, decimal, octal and hexadecimal numeric constants (figure 3-49) may be written in ASM-89 instructions and directives. The assembler can add and subtract constants at assembly time. Numeric constants, including the results of arithmetic operations, must be representable in 16 bits. Positive numbers cannot exceed 65,535 (decimal); negative numbers, which the assembler represents in two's complement notation, cannot be "more negative" than −32,768 (decimal).

Character constants are enclosed in single quote marks as shown in figure 3-49. Strings of characters up to 255 bytes long may be written when initializing storage. Instruction operands, however, can only be one or two characters long (for byte and word instructions respectively).

As an aid to program clarity, The EQU (equate) directive may be used to give names to constants (e.g., DISK_STATUS EQU 0FF20H).

Defining Data

Four ASM-89 directives reserve space for memory variables in the ASM-89 program (see figure 3-50). The DB, DW and DD directives allocate units of bytes, words and doublewords, respectively, initialize the locations, and optionally label them so that they may be referred to by name in instruction statements. The label of a storage directive always refers to the first (lowest-addressed) byte of the area reserved by the directive.

The DB and DW directives may be used to define byte- and word-constant scalars (individual data items) and arrays (sequences of the same type of item). For example, a character string constant could be defined as a byte array:

```
SIGN_ON_MSG: DB 'PLEASE ENTER PASSWORD'
```

The DD directive is typically used to define the address of a location in the system space, i.e., a doubleword pointer variable. The address may be loaded into a pointer register with the LPD instruction.

The DS directive reserves, and optionally names, storage in units of bytes, but does not initialize any of the reserved bytes. DS is typically used for RAM-based variables such as buffers. As there is no special directive for defining a physical address pointer, DS is typically used to reserve the three bytes used by the MOVP instruction.

```
MOVBI   GA, 'A'          ; CHARACTER
MOVBI   GA, 41H          ; HEXADECIMAL
MOVBI   GA, 65           ; DECIMAL
MOVBI   GA, 65D          ; DECIMAL ALTERNATIVE
MOVBI   GA, 101Q         ; OCTAL
MOVBI   GA, 101O         ; OCTAL ALTERNATIVE
MOVBI   GA, 01000001B    ; BINARY
; NEXT TWO STATEMENTS ARE EQUIVALENT AND
;     ILLUSTRATE TWO'S COMPLEMENT REPRESENTATION
;     OF NEGATIVE NUMBERS
MOVBI   GA, −5
MOVBI   GA, 11111011B
```

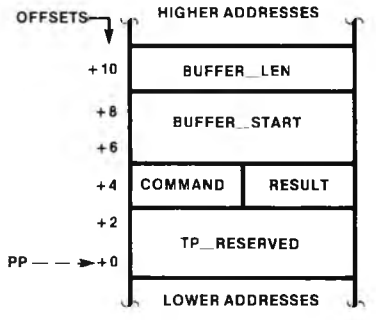
Figure 3-49. ASM89 Constants

```
; ASM89 DIRECTIVE                ; MEMORY CONTENT (HEX)
ALPHA:  DB      1                ; 01
        DB      -2               ; FE (TWO'S COMPLEMENT)
        DB      'A', 'B'         ; 4142
BETA:   DW      1                ; 0100
        DW      -5               ; FAFF
        DW      'AB'             ; 4241
        DW      400, 500         ; 2410F401
        DW      400H, 500H       ; 0004 0005
gamma:  DW      BETA             ; OFFSET OF BETA ABOVE,
                                 ; FROM BEGINNING OF PROGRAM
DELTA   DD      GAMMA            ; ADDRESS (SEGMENT & OFFSET)
                                 ; OF GAMMA
ZETA:   DS      80               ; 80 BYTES, UNINITIALIZED
```

Figure 3-50. ASM-89 Storage Directives

Structures

An ASM-89 structure is a map or template that gives names and relative locations to a collection of related variables that are called structure elements or members. Defining a structure, however, does not allocate storage. The structure is, in effect, overlaid on a particular area of memory when one of its elements is used as an instruction operand. Figure 3-51 shows how a structure representing a parameter block could be defined and then used in a channel program. The assembler uses the structure element name to produce an offset value (structures are used with the offset addressing mode). Compared to "hard coded" offsets, structures improve program clarity and simplify maintenance. If the layout of a memory block changes, only the structure definition must be modified. When the program is reassembled, all symbolic references to the structure are automatically adjusted. When multiple areas of memory are laid out identically, a single structure can be used to address any area by changing the content of the pointer (base) register that specifies the structure's "starting address."



STRUCTURE DEFINITION

```
PARM_BLOCK        STRUC
  TP_RESERVED:    DS   4
  COMMAND:        DS   1
  RESULT:         DS   1
  BUFFER_START:   DS   4
  BUFFER_LEN:     DS   2
PARM_BLOCK        ENDS
```

USING "HARD-CODED" OFFSETS

```
LPD GA, [PP].6
MOVBI   [PP].5,0
```

USING STRUCTURE ELEMENT NAMES

```
LPD GA, [PP].BUFFER_START
MOVBI   [PP].RESULT,0
```

Figure 3-51. ASM-89 Structure Definition and Use

Mnemonics © Intel, 1979

Addressing Modes

Table 3-18 summarizes the notation a programmer uses to specify how the effective address of a memory operand is to be computed. Examples of typical ASM-89 coding for each addressing mode, as well as register and immediate operands, are provided in figure 3-52. Notice that a bracketed reference to a register indicates that the content of the register is to be used to form the effective address of a memory operand, while an unbracketed register reference specifies that the register itself is the operand.

The following examples summarize how the memory addressing modes can be used to access simple variables, structures and arrays.

- If GA contains the address of a memory operand, then [GA] refers to that operand.
- If GA contains the base address of a structure, then [GA].DATA refers to the DATA element (field) in that structure. If DATA is six bytes from the beginning of the structure, then [GA].6 refers to the same location.
- If GA contains the starting address of an array, then [GA+IX] addresses the array element indexed by IX. For example, if IX contains the value 4H, the effective address refers to the fifth element of a byte array, or the third element of a word array. [GA+IX+] selects the same element and additionally auto-increments IX by 1 (byte operation), 2 (word operation) or 3 (MOVP instruction) in anticipation of accessing the next array element.

Note that any pointer register could have been substituted for GA in the previous examples.

Table 3-18. ASM-89 Memory Addressing Mode Notation

Notation	Addressing Mode
[*ptr-reg*]	Based
[*ptr-reg*].*offset*	Offset
[*ptr-reg* + IX]	Indexed
[*ptr-reg* + IX +]	Indexed Post Auto-increment

ptr-reg = GA, GB, GC or PP
offset = 8-bit signed value; may be structure element

Program Transfer Targets

As discussed in section 3.7, program transfer instructions operate by adding a signed byte or word displacement to the task pointer. Table 3-19 shows how the ASM-89 assembler determines the sign and size of the displacement value it places in a program transfer machine instruction. In the table, the terms "backward" and "forward" refer to the location of a label specified as a transfer target relative to the transfer instruction. "Backward" means the label physically precedes the instruction in the source module, and "forward" means the label follows the instruction in the source text. The distances are from the end of the transfer instruction; the distance to the instruction immediately following the transfer is 0 bytes.

```
ADDI    GA, 5            ; REGISTER, IMMEDIATE
ADD     GC, [GB]         ; REGISTER, MEMORY (BASED)
ADDBI   [PP],10          ; MEMORY (BASED), IMMEDIATE
ADDB    IX, [GB].5       ; REGISTER, MEMORY (OFFSET)
ADDB    BC, [GC].COUNT   ; REGISTER, MEMORY (OFFSET)
ADD     [GC+IX], BC      ; MEMORY (INDEXED), REGISTER
ADDI    [GA+IX+],5       ; MEMORY (INDEXED AUTO-INCREMENT), IMMED
ADDB    [PP].ERROR, [GA] ; MEMORY (OFFSET), MEMORY (BASED)
```

Figure 3-52. ASM-89 Operand Coding Examples

Two important points can be drawn from table 3-19. First, a target must lie within 32k bytes of a transfer instruction; this should not prove restrictive except in very large programs. Second, one byte can be saved in the assembled instruction by writing the short mnemonic when the target is known to be within −128 through +127 assembled bytes of the transfer.

It is also important to note that a program transfer target must reside in the same module as the transferring instruction, i.e., the target address must be known at assembly time.

Procedures

An ASM-89 program may invoke an out-of-line procedure (subroutine) with the CALL/LCALL instruction. The first instruction operand specifies a memory location where the content of TP will be stored as a physical address pointer before control is transferred to the procedure. The procedure may return to the instruction following the CALL/LCALL by using the MOVP instruction to restore TP from the save area. Figure 3-53 illustrates one approach to procedure linkage.

A channel program may use the first two words of its parameter block (pointed to by PP) as a task pointer save area. However, this is not recommended if there is any chance that the CPU will issue a "suspend" command to the channel; this command stores the current value of TP in the same location, possibly overwriting a return address.

As in any program transfer, the target of a CALL/LCALL instruction must be contained in the same module and within 32k bytes of the instruction.

Segment Control

The relocatable object module produced by the ASM-89 assembler consists of a single logical segment. (A segment is a storage unit up to 64k bytes long; for a more complete description, refer to sections 2.3 and 2.7.) The ASM-89 SEGMENT and ENDS directives name the segment as shown in figure 3-54. Typically, all instructions and most directives are coded in between these directives. The END directive, which terminates the assembly, is an exception.

The LOC-86 utility can assign this logical segment to any memory address that is a physical segment boundary (i.e., whose low-order four bits are 0000). In a ROM-based system, variable data (which must be in RAM) can be "clustered" together at one "end" of the program as shown in figure 3-55. The ORG directive can then be used to force assembly of the variables to start at a given offset from the beginning of the segment (2,000 hexadecimal bytes in figure 3-55). As the

Table 3-19. Program Transfer Displacement

Target Location			
Mnemonic Form	**Direction**	**Distance**	**Displacement Sign Bytes**
Short (e.g., JMP)	Backward	≤128	− 1
	Forward	≤127	+ 1
	Backward	≤32,768	− 2
	Forward	≤32,767	Error
	Backward	>32,768	Error
	Forward	>32,767	Error
Long (e.g., LJMP)	Backward	≤128	− 2
	Forward	≤127	+ 2
	Backward	≤32,768	− 2
	Forward	≤32,767	+ 2
	Backward	>32,768	Error
	Forward	>32,767	Error

Mnemonics © Intel, 1979

```
.
.
.
CALL SAVE:   DS   3   ; TP SAVE AREA
.
.
.
; SET UP TP SAVE AREA
;    NOTE: EXAMPLE ASSUMES PROGRAM
;            IS IN I/O SPACE. USE LPDI
;            IF IN SYSTEM SPACE.
;          MOVI   GC, CALLSAVE      ; LOAD ADDRESS TO GC
; CALL IT.
          LCALL   [GC],DEMO
.
.
.
          HLT   ; LOGICAL END OF PROGRAM

; DEFINE THE PROCEDURE.
DEMO:
; PROCEDURE INSTRUCTIONS GO HERE.
; NOTE: PROCEDURE MUST NOT UPDATE GC
;         AS IT POINTS TO THE RETURN ADDRESS.
.
.
.
; RETURN TO CALLER.
          MOVP   TP, [GC]
```

Figure 3-53. ASM-89 Procedure Example

```
CHANNEL1     SEGMENT     ; START OF SEGMENT
.
.
.
ASM89 SOURCE STATEMENTS
.
.
.
CHANNEL1     ENDS        ; END OF SEGMENT
             END         ; END OF ASSEMBLY
```

Figure 3-54. ASM-89 SEGMENT and ENDS Directives

figure shows, the segment can then be located so that instructions and constants fall into the ROM portion of memory, while the variable part of the segment is located in RAM. The entire segment, including any "unused" portions, of course, cannot exceed 64k bytes.

Intermodule Communication

An ASM-89 module can make some of its addresses available to other modules by defining symbols with the PUBLIC directive. At a minimum, a channel program must make the address of its first instruction available to the CPU module that starts the channel program. Figure 3-56 shows an ASM-89 module that contains three channel programs labelled READ, WRITE and DELETE. The example shows how a PL/M-86 program and an ASM-86 program could define these "entry points" as EXTERNAL and EXTRN symbols respectively. When the modules are linked together, LINK-86 will match the externals with the publics, thus providing the CPU programs with the addresses they need.

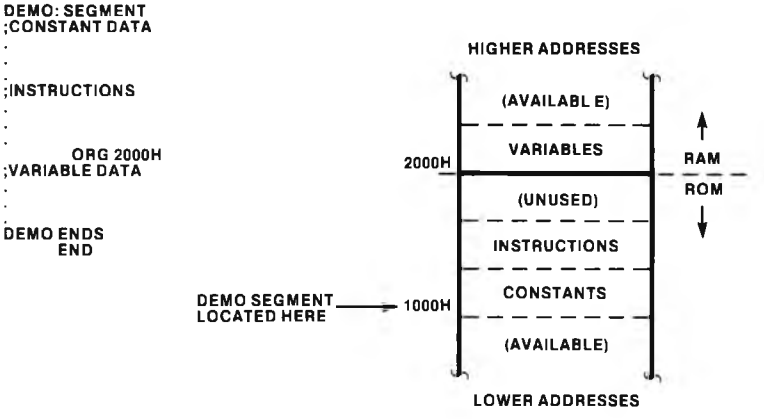


Figure 3-55. Using the ASM-89 ORG Directive

```
                ASM-89 MODULE DEFINES THREE PUBLIC SYMBOLS
.
.
.
PUBLIC      READ, WRITE, DELETE
.
.
.
.
READ:       ; ASM89 INSTRUCTIONS FOR "READ" OPERATION
.
.
            HLT
WRITE:      ; ASM89 INSTRUCTIONS FOR "WRITE" OPERATION
.
.
            HLT
DELETE:     ; ASM89 INSTRUCTIONS FOR "DELETE" OPERATION
.
.
            HLT
```

Figure 3-56. ASM-89 PUBLIC Directive

Mnemonics © Intel, 1979

PL/M-86 MODULE USES "WRITE" SYMBOL

```
DECLARE   (READ,WRITE,DELETE) POINTER EXTERNAL;
DECLARE   PARM$BLOCK   STRUCTURE
          (TP$START          POINTER,
          BUFFER$ADDR          POINTER,
          BUFFER$LEN          WORD);
.
.
/*SET UP "WRITE" CHANNEL OPERATION*/
PARM$BLOCK. TP$START = WRITE;
.
.
.
```

ASM-86 MODULE USES "READ" SYMBOL

```
                .
                .
                .
EXTRN           READ,WRITE,DELETE
.
.
.
READ_PTR        DD  READ
WRITE_PTR       DD  WRITE
DELETE_PTR      DD  DELETE
.
.
.
; PARM_BLOCK
                EVEN        ; FORCE TO EVEN ADDRESS
TP_START        DD ?
BUFFER_ADDRDD ?
BUFFER_LEN      DW ?
.
.
.
; SET UP "READ" CHANNEL OPERATION
      MOV   AX, WORD PTR READ_PTR          ; 1ST WORD
      MOV   WORD PTR TP_START, AX
      MOV   AX, WORD PTR READ_PTR          ; 2ND WORD
      MOV   WORD PTR TP_START + 2, AX
      .
      .
      .
```

Figure 3-56. ASM-89 PUBLIC Directive (Cont'd.)

Conversely, an ASM-89 module can obtain the address of a public symbol in another module by defining it with the EXTRN directive. An external symbol, however, can only appear as the initial value operand of a DD directive (see figure 3-57). This effectively means that an ASM-89 program's use of external symbols is limited to obtaining the addresses of data located in the system space. Another way of doing this, which may be preferable in many cases, is to have the CPU program place system space addresses in the parameter block.

PL/M-86 PROGRAM DECLARES PUBLIC SYMBOL "BUFFER"

```
.
.
.
DECLARE BUFFER (80) BYTE PUBLIC;
.
.
.
```

ASM-89 PROGRAM OBTAINS ADDRESS OF PUBLIC SYMBOL "BUFFER"

```
.
.
.
EXTRN BUFFER
.
.
.
BUF_ADDRESS   DD   BUFFER
.
.
.
LPD   GA, BUF_ADDRESS   ; POINT TO SYSTEM BUFFER
.
.
.
```

Figure 3-57. ASM-89 EXTRN Directive

Sample Program

Figure 3-58 diagrams the logic of a simple ASM-89 program; the code is shown in figure 3-59. The program reads one physical record (sector) from a diskette drive controlled by an 8271 Floppy Disk Controller. No particular system configuration is implied by the program, except that the 8271 resides in the IOP's I/O space.

Hardware address decoding logic is assumed to be set up as follows:

- reading location FF00H selects the 8271 status register,
- writing location FF00H selects the 8271 command register,
- reading location FF01H selects the 8271 result register
- writing location FF01H selects the 8271 parameter register
- decoding the address FF04H provides the 8271 DACK (DMA acknowledge) signal.

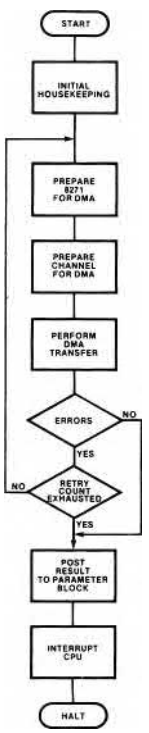


Figure 3-58. ASM-89 Sample Program Flow

Mnemonics © Intel, 1979

The program uses structures to address the parameter block and the 8271 registers. Register PP contains the address of the parameter block, and the program loads GC with FF00H to point to the 8271 registers. The program's entry point (the label START) is defined as a PUBLIC symbol so that the CPU program can place its address in the parameter block when it starts the program.

Register IX is used as a retry counter. If the transfer is not completed successfully (bit 3 of the 8271 result register ≠ 0), the program retries the transfer up to 10 times.

Since the 8271 automatically requests a DMA transfer upon receipt of the last parameter, this parameter is sent immediately following the XFER command.

```
8089 ASSEMBLER

ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE FLOPPY
OBJECT MODULE PLACED IN :F0:FLOPPY.OBJ
ASSEMBLER INVOKED BY ASM89 FLOPPY.A89

                                  1
0000                              2 FLOPPY            SEGMENT
                                  3 ;***
                                  4 ;*** 8089 PROGRAM TO READ SECTOR FROM FLOPPY DISK
                                  5 ;***
                                  6
                                  7 ;*** LAY OUT PARAMETER BLOCK.
                                  8 PARM BLOCK        STRUC
0000                              9    RESERVED TP:      DS    4
0004                             10    BUFF PTR:         DS    4
0008                             11    TRACK:            DS    1
0009                             12    SECTOR:           DS    1
000A                             13    RETURN CODE:      DS    1
000B                             14    PARM BLOCK     ENDS
                                 15
                                 16 ;***LAY OUT 8271 DEVICE REGISTERS.
0000                             17 FLOPPY REGS       STRUC
0001                             18    COMMAND STAT:     DS    1
0002                             19    PARM RESULT:      DS    1
                                 20    FLOPPY REGS    ENDS
                                 21
  FF00                           22 ;***8271 ADDRESSES.
  FF04                           23 FLOPPY REG ADDR   EQU    OFF00H        ;LOW-ADDRESSED REGISTER
                                 24 DACK 8271         EQU    OFF04H        ;DMA ACKNOWLEDGE
                                 25
                                 26 ;***MAKE PROGRAM ENTRY POINT ADDRESS
                                 27 ;       AVAILABLE TO OTHER MODULES.
                                 28 PUBLIC            START
                                 29
                                 30 ;***CLEAR RETURN CODE IN PARAMETER BLOCK.
0000    0A4F 0A 00               31 START:            MOVBI    [PP].RETURN CODE,0
                                 32
                                 33 ;***INITIALIZE RETRY COUNT.
0004    B130 0A00                34                   MOVI     IX,10
                                 35
                                 36 ;***POINT GC AT LOW-ORDER 8271 REGISTER.
0008    5130 00FF                37                   MOVI     GC,FLOPPY REG ADDR
                                 38
                                 39 ;***SEND COMMAND SEQUENCE TO 8271, HOLDING FINAL PARM.
                                 40 ;***WAIT UNTIL 8271 IS NOT BUSY.
000C    EABA 00 FC               41 RETRY:            JNBT     [GC].COMMAND STAT,7,RETRY
                                 42 ;***SEND "READ SECTOR, DRIVE 0" COMMAND.
0010    0A4E 00 12               43                   MOVBI    [GC].COMMAND STAT,012H
                                 44 ;***SEND TRACK ADDRESS PARAMETER.
0014    0293 08 02CE 01          45                   MOVB     [GC].PARM RESULT,[PP].TRACK
                                 46
                                 47 ;***LOAD CHANNEL CONTROL REGISTER SPECIFYING:
                                 48 ;    FROM PORT TO MEMORY,
                                 49 ;    SYNCHRONIZE ON SOURCE,
                                 50 ;    GA POINTS TO SOURCE,
                                 51 ;    TERMINATE ON EXT,
                                 52 ;    TERMINATION OFFSET = 0.
001A    D130 2088                53                   MOVI     CC,08820H
                                 54
```

Figure 3-59. ASM-89 Sample Program

Mnemonics © Intel, 1979

```
                                  55 ;***SET SOURCE BUS = 8, DEST BUS = 16.
001E   A000                       56                 WID     8,16
                                  57
                                  58 ;***POINT GB AT DESTINATION, GA AT SOURCE.
0020   238B 04                    59                 LPD     GB,[PP].BUFF_PTR
0023   1130 04FF                  60                 MOVI    GA,DACK_8271
                                  61
                                  62 ;***INSURE THAT 8271 IS READY FOR LAST PARAMETER.
0027   AABA 00 FC                 63 WAIT1:          JNBT    [GC].COMMAND_STAT,5,WAIT1
                                  64
                                  65 ;***PREPARE FOR DMA.
002B   6000                       66                 XFER
                                  67
                                  68 ;***START DMA BY SENDING FINAL PARAMETER TO 8271.
002D   0293 09 02CE 01            69                 MOVB    [GC].PARM_RESULT,[PP].SECTOR
                                  70
                                  71 ;***PROGRAM RESUMES HERE FOLLOWING EXT.
                                  72
                                  73 ;***IF TRANSFER IS OK THEN EXIT, ELSE TRY AGAIN.
0033   6ABE 01 05                 74                 JBT     [GC].PARM_RESULT,3,EXIT
                                  75
                                  76 ;***DECREMENT RETRY COUNT.
0037   A03C                       77                 DEC     IX
                                  78
                                  79 ;***TRY AGAIN IF COUNT NOT EXHAUSTED.
0039   A840 D0                    80                 JNZ     IX,RETRY
                                  81
                                  82 ;***WAIT UNTIL 8271 IS NOT BUSY.
003C   EABA 00 FC                 83 EXIT:           JNBT    [GC].COMMAND_STAT,7,EXIT
                                  84
                                  85 ;***SEND "READ RESULT" COMMAND TO 8271.
0040   0A4E 00 2C                 86                 MOVBI   [GC].COMMAND_STAT,02CH
                                  87
                                  88 ;***WAIT FOR RESULT.
0044   8ABA 00 FC                 89 WAIT2:          JNBT    [GC].COMMAND_STAT,4,WAIT2
                                  90
                                  91 ;***POST RESULT IN PARAMETER BLOCK FOR CPU.
0048   0292 01 02CF 0A            92                 MOVB    [PP].RETURN_CODE,[GC].PARM_RESULT
                                  93
                                  94 ;***INTERRUPT CPU.
004E   4000                       95                 SINTR
                                  96
                                  97 ;***STOP EXECUTION.
0050   2048                       98                 HLT
                                  99
0052                             100 FLOPPY          ENDS
                                 101                 END
```

```
SYMBOL TABLE
------------

 DEFN VALUE TYPE  NAME
----- ----- ----  ----

   10  0004  SYM  BUFF_PTR
   18  0000  SYM  COMMAND_STAT
   24  FF04  SYM  DACK_8271
   83  003C  SYM  EXIT
    2  0000  SYM  FLOPPY
   17  0000  STR  FLOPPY_REGS
   23  FF00  SYM  FLOPPY_REG_ADDR
    8  0000  STR  PARM_BLOCK
   19  0001  SYM  PARM_RESULT
    9  0000  SYM  RESERVED_TP
   41  000C  SYM  RETRY
   13  000A  SYM  RETURN_CODE
   12  0009  SYM  SECTOR
   31  0000  PUB  START
   11  0008  SYM  TRACK
   63  0027  SYM  WAIT1
   89  0044  SYM  WAIT2

ASSEMBLY COMPLETE; NO ERRORS FOUND
```

Figure 3-59. ASM-89 Sample Program (Cont'd.)

Linking and Locating ASM-89 Modules

The LINK-86 utility program combines multiple relocatable object modules into a single relocatable module. The input modules may consist of modules produced by any of the 8086 family language translators: ASM-89, ASM-86, or PL/M-86. LINK-86's principal function is to satisfy external references made in the modules. Any symbol that is defined with the EXTRN directive in ASM-89 or ASM-86 or is declared EXTERNAL in PL/M-86 is an external reference, i.e., a reference to an address contained in another module. Whenever LINK-86 encounters an external reference, it searches the other modules for a PUBLIC symbol of the same name. If it finds the matching symbol, it replaces the external reference with the address of the object.

The most common occurrence of an external reference in a system that employs one or more 8089s is the channel program address. In order for a CPU program to start a channel program, it must ensure that the address of the first channel program instruction is contained in the first two words of the parameter block. Since the channel program is assembled separately, the translator that processes the CPU program will not typically know its address. If this address is defined as an external symbol (see figure 3-56), LINK-86 will obtain the address from the ASM-89 channel program when the two are linked together. (The ASM-89 program must, of course, define the symbol in a PUBLIC directive.)

Other external references may arise when one module uses data (e.g., a buffer) that is contained in another module, and (in PL/M-86 and ASM-86 modules) when one module executes another module, typically by a CALL statement or instruction.

When an 8089 module (or modules) is to be located in the system space, it may be linked together with PL/M-86 or ASM-86 modules as described above and shown in figure 3-60. LINK-86 resolves external references and combines the input modules into a single relocatable object module. This module can be input to LOC-86 (LOC-86 assigns final absolute memory addresses to all of the instructions and data). This absolute object module may, in turn, be processed by the OH-86 utility to translate the module into the hexadecimal format. This format makes the module readable (the records are written in ASCII characters) and is required by some PROM programmers and RAM loaders. Intel's Universal PROM Programmer (UPP) and iSBC 957™ Execution Package (loader) use the hexadecimal format.

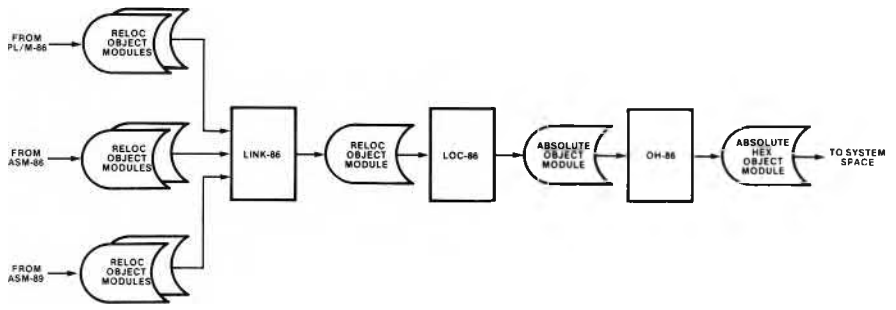


Figure 3-60. Creating a Single Absolute Object Module

If the 8089 code is to reside in its I/O space, a different technique is required since separate absolute object modules must be produced for the system and I/O spaces. Figure 3-61 shows how to link and locate when there are external references between I/O space modules and system space modules.

The normal link and locate sequence is followed and culminates in the production of an absolute module in hexadecimal format. Since the records in this file are human-readable, the file can be edited using the ISIS-II text editor. The editing task involves finding the 8089 I/O space records in the file, writing them to one file, and then writing the 8086/8088 records (destined for the system space) to another file. *MCS-86 Absolute Object File Formats*, Order No. 9800921, available from Intel's Literature Department, describes the records in absolute (including hexadecimal) object modules.

When using the previous method, it is likely that LOC-86 will issue messages warning that segments overlap. For example, the 8089 code would typically be located starting at absolute location 0H of the I/O space. However, the 8086/8088 interrupt pointer table occupies these low memory addresses in the system space. Since LOC-86 has no way to know that the segment will ultimately be located in different address spaces, it will warn of the conflict; the warning may be ignored.

An alternative to linking the modules together and then separating them is to link system space modules separately from I/O space modules as shown in figure 3-62. This approach avoids the manual edit of the absolute object module and the segment conflict messages from LOC-86. It requires, however, that modules in the two spaces not use the EXTRN/PUBLIC mechanism to refer to each other. Modules in the same space can define external and public symbols, however.

External references from I/O space modules to system space modules can be eliminated if the CPU programs pass all system space addresses in parameter blocks. In other words, a channel program can obtain any address in the system space if the address is in the parameter block. Using this approach allows the system space addresses to be changed during execution. If the addresses are constant values, they may also be altered as system development proceeds without relinking the channel programs.

External references from system space modules to addresses in the I/O space may be eliminated by assigning these addresses values that are known at assembly or compilation time. Figure 3-63 illustrates how the ASM-89 ORG directive can be used to force the first instruction (entry point) of a channel program to an absolute address. In the case of the example, one module contains two entry points labelled "READ" and "WRITE." Assuming the module is located at absolute address 0H in the I/O space, the channel programs will begin at 200H and 600H respectively. In the example, these values have been chosen arbitrarily; in a typical application they would be based on the length of the programs and the location of RAM and ROM areas. By starting the programs at fixed addresses that are known to the CPU programs that activate them, the channel programs can be reassembled without needing to relink the CPU programs.

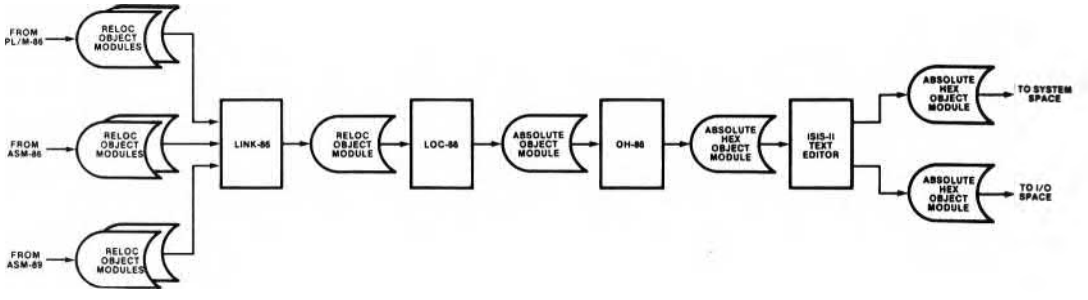


Figure 3-61. Creating Separate Absolute Object Modules—External References in Relocatable Modules

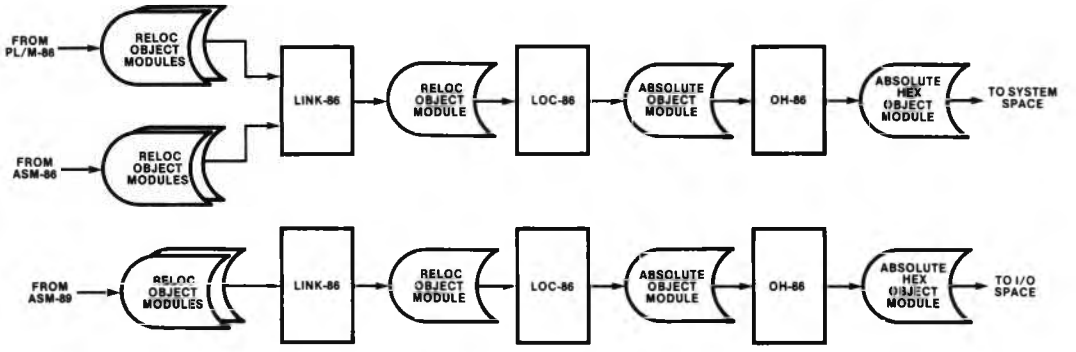


Figure 3-62. Creating Separate Absolute Object Modules—No External References in Relocatable Modules

```
ASM-89 ENTRY POINT DEFINITIONS
.
.
.
        ORG 200H
READ:
.
.
.
; INSTRUCTIONS FOR "READ" CHANNEL PROGRAM
.
.
.
        ORG 600H
WRITE:
.
.
.
; INSTRUCTIONS FOR "WRITE" CHANNEL PROGRAM
.
.
.

ASM-86 DEFINITION OF ENTRY POINT ADDRESSES
.
.
.
READ_ADDR       DD   200H
WRITE_ADDR      DD   600H
.
.
.

PL/M-86 DECLARATION OF ENTRY POINT ADDRESSES
.
.
.
DECLARE READ$ADDR POINTER;
DECLARE WRITE$ADDR POINTER;
READ$ADDR = 200H;
WRITE$ADDR = 600H;
```

Figure 3-63. Using Absolute Entry Point Addresses

3.10 Programming Guidelines and Examples

This section provides two types of 8089 programming information. A series of general guidelines, which apply to system and program design, is presented first. These guidelines are followed by specific coding examples that illustrate programming techniques that may be applied to many different types of applications.

Programming Guidelines

The practices in this section are recommended to simplify system development and, particularly, for system maintenance and enhancement. Software that is designed in accordance with these **guidelines will be adaptable to the changing environment in which most systems operate, and will be in the best position to take advantage of new Intel hardware and software products.**

Segments

Although the IOP does not "see" the segmented organization of system memory, it should respect this logical structure. The IOP should only address the system space through pointers passed by the CPU in the parameter block. It should not perform arithmetic on these addresses or otherwise manipulate them except for the automatic incrementing that occurs during DMA transfers. It is the responsibility of the CPU to pass addresses such that transfer operations do not cross segment boundaries.

Self-Modifying Code

Programs that alter their own instructions are difficult to understand and modify, and preclude placing the code in ROM. They may also inhibit compatibility with future Intel hardware and software products.

Note also that when the 8089 is on a 16-bit bus, its instruction fetch queue can interfere with the attempt of one instruction to modify the next sequential instruction. Although the instruction may be changed in memory, its unmodified first byte will be fetched from the queue rather than memory if it is on an odd address. The processor will thus execute a partially-modified instruction with unpredictable results.

I/O System Design

Section 2.10 notes that I/O systems should be designed hierarchically. Application programs "see" only the topmost level of the structure; all details pertaining to the physical characteristics and operation of I/O devices are relegated to lower levels. Figure 3-64 shows how this design approach might be employed in a system that uses an 8089 to perform I/O. The same concept can be expanded to larger systems with multiple IOPs.

The application system is clearly separated from the I/O system. No application programs perform I/O; instead they send an I/O request to the I/O supervisor. (In systems with file-oriented I/O, the request might be sent to a file system that would then invoke the I/O supervisor.) The I/O request should be expressed in terms of a logical block of data—a record, a line, a message, etc. It should also be devoid of any device-dependent information such as device address, sector size, etc.

The I/O supervisor transforms the application program's request for service into a parameter block and dispatches a channel program to carry out the operation. The I/O supervisor controls the channels; therefore, it knows the correspondence between channels and I/O devices, the locations of CBs and channel programs, and the format of all of the parameter blocks. The I/O supervisor also coordinates channel "events," monitoring BUSY flags and responding to channel-generated interrupt requests. The I/O supervisor does not, however, communicate with I/O devices that are controlled by the channels. If the CPU performs some I/O itself (this should be restricted to devices other than those run by the channels), the I/O supervisor invokes the equivalent of a channel program in the CPU to do the physical I/O. Note that although the I/O supervisor is drawn as a single box in figure 3-64, it is likely to be structured as a hierarchy itself, with separate modules performing its many functions.

The software interface between the CPU's I/O supervisor and an IOP channel program should be completely and explicitly defined in the

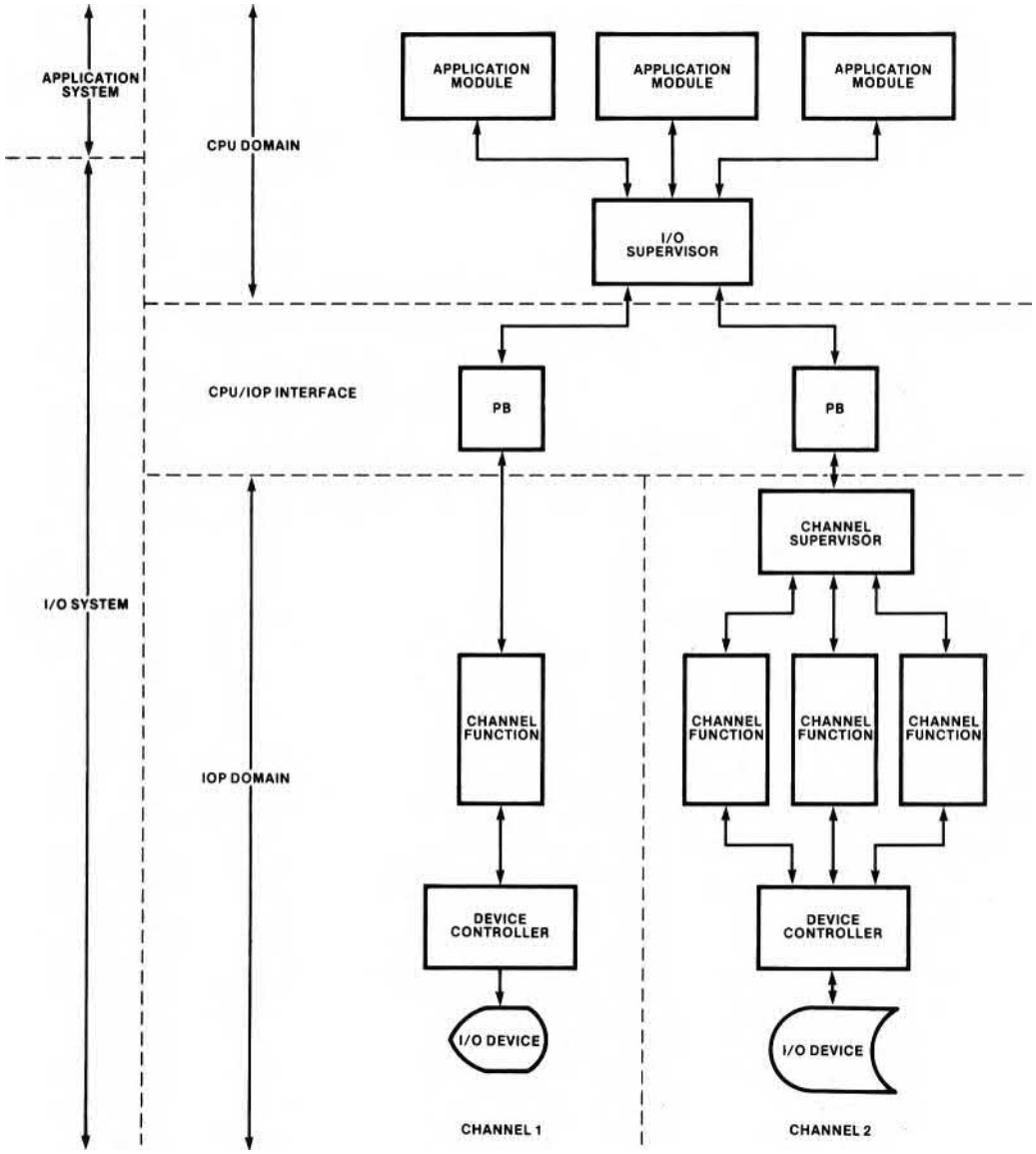


Figure 3-64. 8089-Based I/O System Design

parameter block. For example, the I/O supervisor should pass the addresses of all system memory areas that the channel program will use. The channel program should not be written so that it "knows" any of these addresses, even if they are constants. Concentrating the interface into one place like this makes the system easier to understand and reduces the likelihood of an undesirable side effect if it is modified. It also generalizes the design so that it may be used in other application systems.

Figure 3-64 shows a simple channel program running on channel 1 and a more complex program running on channel 2. Channel 1's program performs a single function and is therefore designed as a simple program. The program on channel 2 performs three functions (e.g., "read," "write," "delete") and is structured to separate its functions. The functions might be implemented as procedures called by the "channel supervisor" depending on the content of the parameter block. Notice that to the I/O supervisor, both programs appear alike; in particular, both have a single entry point.

In some channel programs, different functions will need different information passed to them in the parameter block. Figure 3-65 shows one technique that accommodates different formats while still allowing the channel supervisor to determine which procedure to call from the PB. The parameter block is divided into fixed and variable portions, and a function code in the fixed area indicates the type of operation that is to be performed. Part of the fixed area has been set aside so that additional parameters can be added in the future.

Programming Examples

The first example in this section illustrates how a CPU can initialize a group of IOPs and then dispatch channel programs. This code is written in PL/M-86.

The remaining examples, written in ASM-89, demonstrate the 8089 instruction set and addressing modes in various commonly-encountered programming situations. These include:

- memory-to-memory transfers
- saving and restoring registers

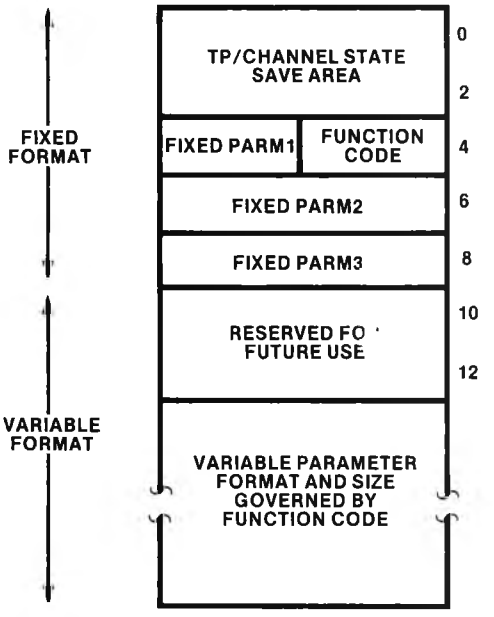


Figure 3-65. Variable Format Parameter Block

Initialization and Dispatch

The PL/M-86 code in figure 3-66 initializes two IOPs and dispatches two channel programs on one of the IOPs. The same general technique can be used to initialize any number of IOPs. The hypothetical system that this code runs on is configured as follows:

- 8086 CPU (16-bit system bus);
- two remote IOPs share an 8-bit local I/O bus via the request/grant lines operating in mode 1;
- 8089 channel attentions are mapped into four port addresses in the CPU's I/O space;
- channel programs reside in the 8089 I/O space;
- one 8089 controls a CRT terminal, one channel running the display, the other scanning the keyboard and building input messages;
- the function of the second 8089 is not defined in the example.

The code declares one CB (channel control block) for each 8089. The CBs are declared as two-element arrays, each element defining the structure of one channel's portion of the CB. The SCB (system configuration block) and SCP (system configuration pointer) are also declared as structures. The SCP is located at its dedicated system space address of FFFF6H. The other structures are not located at specific addresses since they are all linked together by a chain of pointers "anchored" at the SCP.

Two simple parameter blocks define messages to be transmitted between the PL/M-86 program and the CRT. Each PB contains a pointer to the beginning of the message area and the length of the message. In the case of the keyboard (input) message, the channel program builds the message in the buffer pointed to by the pointer in the PB and returns the length of the message in the PB.

The code initializes one IOP at a time since the chain of control blocks read by the IOP during initialization must remain static until the process is complete. To initialize the first IOP, the code fills in the SYSBUS and SOC fields and links the blocks to each other using the PL/M-86 @ (address) operator. It sets channel 1's BUSY flag to FFH so that it can monitor the flag to determine when the initialization has been completed (the IOP clears the flag to 0H when it has finished). Channel 2's BUSY flag is cleared, although this could just as well have been done after the initialization (the IOP does not alter channel 2's BUSY flag during initialization). The code starts the IOP by issuing a channel attention to channel 1 to indicate that the IOP is a bus master. PL/M-86's OUT function is used to select the port address to which the IOP's CA and SEL lines have been mapped. The data placed on the bus (0H) is ignored by the IOP. It then waits until the IOP clears the channel 1 BUSY flag.

The second IOP is initialized in the same manner, first changing the pointer in the SCB to point to the second IOP's channel control block. If this IOP were on a different I/O bus, the SOC field would have been altered if a different request/grant mode were being used or if the IOP had a 16-bit I/O bus. The second IOP is a slave so its initialization is started by issuing a CA to channel 2 rather than channel 1.

After both IOPs are ready, the code dispatches two channel programs (not coded in the example); one program is dispatched to each channel of one of the IOPs. To avoid external references, the system has been set up so that the PL/M-86 code "knows" the starting addresses of these channel programs (200H and 600H). The code uses the PL/M-86 LOCKSET function to:

- lock the system bus;
- read the BUSY flag;
- set the BUSY flag to FFH if it is clear;
- unlock the system bus.

This operation continues until the BUSY flag is found to be clear (indicating that the channel is available). Setting the flag immediately to FFH prevents another processor (or another task in this program activated as a result of an interrupt) from using the channel. The code fills in the parameter block with the address and length of the message to be displayed, sets the CCW and then links the channel program (task block) start address to the parameter block and links the parameter block to the CB. The channel is dispatched with the OUT function that effects a channel attention for channel 1.

A similar procedure is followed to start channel 2 scanning the terminal keyboard. In this case, the code allows channel 2 to generate an interrupt request (which it might do to signal that a message has been assembled). An interrupt procedure would then handle the interrupt request.

```
/*ASSIGN NAMES TO CONSTANTS*/
DECLARE     CHANNEL$BUSY        LITERALLY '0FFH';
DECLARE     CHANNEL$CLEAR       LITERALLY '0H';
DECLARE     CR /*CARR. RET.*/   LITERALLY '0DH';
DECLARE     LF /*LINE FEED*/    LITERALLY '0AH';
DECLARE     DISPLAY$TB          LITERALLY '200H';
DECLARE     KEYBD$TB            LITERALLY '600H';
```

Figure 3-66. Initialization and Dispatch Example

```
DECLARE   /*IOP CHANNEL ATTENTION ADDRESSES*/
IOP$A$CH1      LITERALLY              '0FFE0H',
IOP$A$CH2      LITERALLY              '0FFE1H',
IOP$B$CH1      LITERALLY              '0FFE2H',
IOP$B$CH2      LITERALLY              '0FFE3H';

DECLARE   /*CHANNEL CONTROL BLOCK FOR IOP$A)
               CB$A(2)          STRUCTURE
               (BUSY            BYTE,
               CCW              BYTE,
               PB$PTR           POINTER,
               RESERVED         WORD);

DECLARE   /*CHANNEL CONTROL BLOCK FOR IOP$B*/
               CB$B(2)          STRUCTURE
               (BUSY            BYTE,
               CCW              BYTE,
               PB$PTR           POINTER,
               RESERVED         WORD);

DECLARE   /*SYSTEM CONFIGURATION BLOCK*/
               SCB              STRUCTURE
               (SOC             BYTE,
               RESERVED         BYTE,
               CB$PTR           POINTER);

DECLARE   /*SYSTEM CONFIGURATION POINTER*/
               SCP              STRUCTURE
               (SYSBUS          BYTE,
               SCB$PTR          POINTER) AT (0FFFF6H);

DECLARE   MESSAGE$PB STRUCTURE
               (TB$PTR          POINTER,
               MSG$PTR          POINTER,
               MSG$LENGTH       WORD);

DECLARE   KEYBD$PB STRUCTUE
               (TP$PTR          POINTER,
               BUFF_PTR         POINTER,
               MSG$SIZE         WORD);

DECLARE   SIGN$ON BYTE (*) DATA
          (CR, LF, 'PLEASE ENTER USER ID');

DECLARE   KEYBD$BUFF BYTE (256);

/*
 *INITIALIZE IOP$A, THEN IOP$B
*/

/*PREPARE CONTROL BLOCKS FOR IOP$A*/
SCP.SCB$PTR = @ SCB;
SCP.SYSBUS = 01H; /*16-BIT SYSTEM BUS*/
SCB.SOC = 02H; /*RQ/GT MODE1, 8-BIT I/O BUS*/
SCB.CB$PTR = @ CB$A(0);
CB$A(0).BUSY = CHANNEL$BUSY
CB$A(1).BUSY = CHANNEL$CLEAR;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

```
/*ISSUE CA FOR CHANNEL1, INDICATING IOP IS MASTER*/
OUT (IOP$A$CH1) = 0H;

/*WAIT UNTIL FINISHED*/
DO WHILE CB$A(0).BUSY = CHANNEL$BUSY;
    END;

/*PREPARE CONTROL BLOCKS FOR IOP$B*/
SCB.CB$PTR = @CB$B(0);
CB$B(0).BUSY = CHANNEL$BUSY;
CB$B(1).BUSY = CHANNEL$CLEAR;

/*ISSUE CA FOR CHANNEL2, INDICATING SLAVE STATUS*/
OUT (IOP$B$CH2) = 0H;

/*WAIT UNTIL IOP IS READY*/
DO WHILE CB$B(0).BUSY = CHANNEL$BUSY;
    END;



/*
 *SEND SIGN ON MESSAGE TO CRT CONTROLLED
 *BY CHANNEL 1 OF IOP$A
 */
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/
DO WHILE LOCKSET (@CB$A(0).BUSY, CHANNEL$BUSY);
    END;

/*SET CCW AS FOLLOWS:
 *      PRIORITY = 1,
 *      NO BUS LOAD LIMIT,
 *      DISABLE INTERRUPTS,
 *      START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(0).CCW = 10011001B;

/*LINK MESSAGE PARAMETER BLOCK TO CB*/
CB$A(0).PB$PTR = @ MESSAGE$PB;

/*FILL IN PARAMETER BLOCK*/
MESSAGE$PB.TB$PTR = DISPLAY$TB;
MESSAGE$PB.MSG$PTR = @SIGN$ON;
MESSAGE$PB. MSB$LENGTH = LENGTH (SIGN$ON);

/*DISPATCH THE CHANNEL*/
OUT (IOP$A$CH1) = 0H;

/*
 *DISPATCH CHANNEL 2 OF IOP$A TO
 *CONTINUOUSLY SCAN KEYBOARD, INTERRUPTING
 *WHEN A COMPLETE MESSAGE IS READY
 */
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/
DO WHILE LOCKSET (@ CB$A(1).BUSY, CHANNEL$BUSY);
    END;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

```
/*SET CCW AS FOLLOWS:
 *     PRIORITY = 0
 *     BUS LOAD LIMIT,
 *     ENABLE INTERRUPTS,
 *     START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(1).CCW = 00110001B;
/*LINK KEYBOARD PARAMETER BLOCK TO CB*/
CB$A(1).PB$PTR = @ KEYBD$PB;
/*FILL IN PARAMETER BLOCK*/
KEYBD$PB.TB$PTR = KEYBD$TB;
KEYBD$PB.BUFF$PTR = @ KEYBD$BUFF;
KEYBD$PB.MSG$SIZE = 0H;
/*DISPATCH THE CHANNEL*/
OUT (IOP$A$CH2) = 0H;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

Memory-to-Memory Transfer

Figure 3-67 shows a channel program that performs a memory-to-memory block transfer in seven instructions. The program moves up to 64k bytes between any two locations in the system space. A 16-bit system bus is assumed, and the CPU is assumed to be monitoring the channel's BUSY flag to determine when the program has finished.

To attain maximum transfer speed, the program locks the bus during each transfer cycle. This ensures that another processor does not acquire the bus in the interval between the DMA fetch and store operations. By setting this channel's priority bit in the CCW to 1 and the other channel's to 0, the CPU could effectively prevent the other channel from running during the transfer. Byte count termination is selected so that the transfer will stop when the number of bytes specified by the CPU has been moved. Since there is only a single termination condition, a termination offset of 0 is specified. The transfer begins after the WID instruction, and the HLT instruction is executed immediately upon termination.

Saving and Restoring Registers

A CPU program can "interrupt" a channel program by issuing a "suspend" channel command. The channel responds to this command by saving the task pointer and PSW in the first two words of the parameter block. The suspended program can be restarted by issuing a "resume" command that loads TP and the PSW from the save area.

If the CPU wants to execute another channel program between the suspend and resume operations, the suspended program's registers will usually have to be saved first. If the "interrupting" program "knows" that the registers must be saved, it can perform the operation and also restore the registers before it halts.

A more general solution is shown in figure 3-68. This is a program that does nothing but save the contents of the channel registers. The registers are saved in the parameter block because PP is the only register that is known to point to an available area of memory. A similar program could be written to restore registers from the same parameter block.

Using this approach, the CPU would "interrupt" a running program as follows:

- suspend the running program,
- run the register save program,
- run the "interrupting" program,
- run the register restore program,
- resume the suspended program.

```
MEMEXAMP        SEGMENT
;**MEMORY-TO-MEMORY TRANSFER PROGRAM**
PB              STRUC
TP_RESERVED:    DS      4
FROM_ADDR:      DS      4
TO_ADDR:        DS      4
SIZE:           DS      2
PB              ENDS

;POINT GA AT SOURCE, GB AT DESTINATION.
                LPD             GA, [PP].FROM_ADDR
                LPD             GB, [PP].TO_ADDR
;LOAD BYTE COUNT INTO BC.
                MOV             BC, [PP].SIZE
;LOAD CC SPECIFYING:
;       MEMORY TO MEMORY,
;       NO TRANSLATE,
;       UNSYNCHRONIZED,
;       GA POINTS TO SOURCE,
;       LOCK BUS DURING TRANSFER,
;       NO CHAINING,
;       TERMINATING ON BYTE COUNT,OFFSET = 0.
                MOV             CC, 0C208H
;PREPARE CHANNEL FOR TRANSFER.
                XFER

;SET LOGICAL BUS WIDTH.
                WID             16,16

;STOP EXECUTION AFTER DMA.
                HLT
MEMEXAMP        ENDS
                END
```

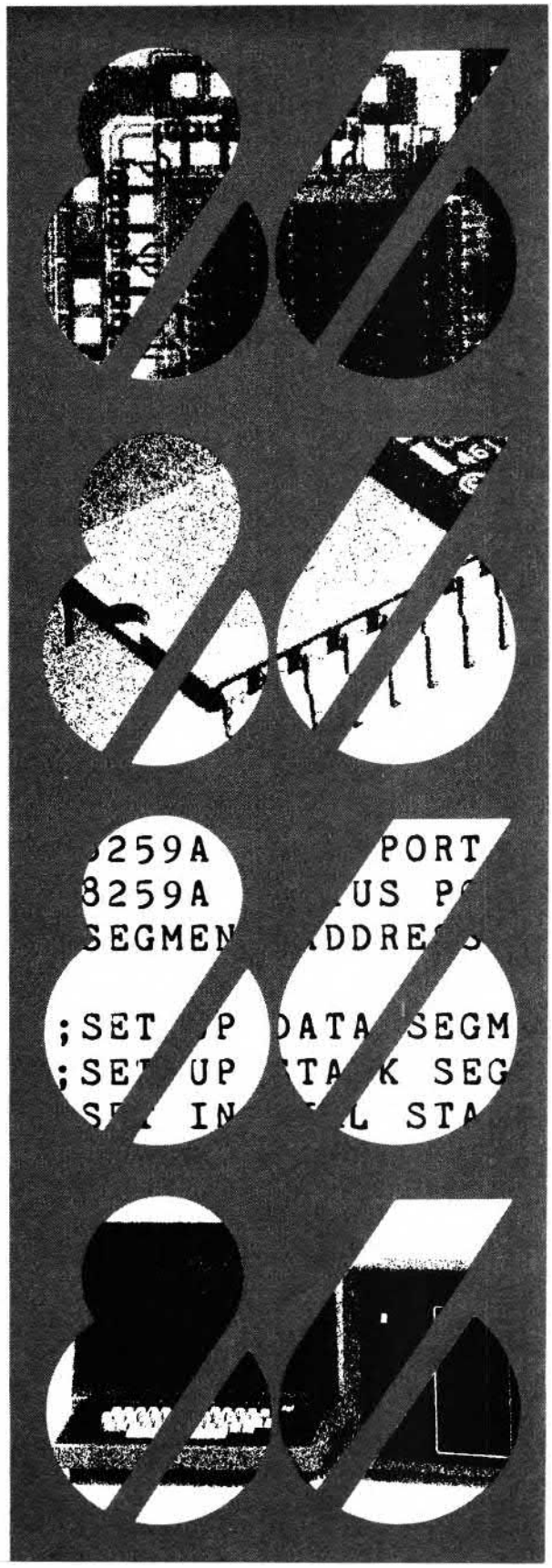
Figure 3-67. Memory-to-Memory Transfer Example

```
SAVEREGS        SEGMENT
;SAVE ANOTHER CHANNEL'S REGISTERS IN PB
PB              STRUC
TP_RESERVED:    DS      4
GA_SAVE:        DS      3
GB_SAVE:        DS      3
GC_SAVE:        DS      3
IX_SAVE:        DS      2
BC_SAVE:        DS      2
MC_SAVE:        DS      2
CC_SAVE:        DS      2
PB              ENDS

                MOVP    [PP].GA_SAVE, GA
                MOVP    [PP].GB_SAVE, GB
                MOVP    [PP].GC_SAVE, GC
                MOV     [PP].IX_SAVE, IX
                MOV     [PP].BC_SAVE, BC
                MOV     [PP].MC_SAVE, MC
                MOV     [PP].CC_SAVE, CC
                HLT
SAVEREGS        ENDS
                END
```

Figure 3-68. Register Save Example

Mnemonics © Intel, 1979

Chapter 4
Hardware Reference Information



CHAPTER 4
HARDWARE REFERENCE INFORMATION

4.1 Introduction

This chapter presents specific hardware information regarding the operation and functions of the 8086 family processors: the 8086 and 8088 Central Processing Units (CPUs) and the 8089 I/O Processor (IOP). Abbreviated descriptions of the 8086 family support circuits and their circuit functions appear where appropriate within the processor descriptions. For more specific information on any of the 8086 family support circuits, refer to the corresponding data sheets in Appendix B.

4.2 8086 and 8088 CPUs

The 8086 and 8088 CPUs are characterized by a 20-bit (1 megabyte) address bus and an identical instruction/function format, and differ essentially from one another by their respective data bus widths (the 8086 uses a 16-bit data bus, and the 8088 uses an 8-bit data bus). Except where expressly noted, the ensuing descriptions are applicable to both CPUs.

Both the 8086 and 8088 feature a combined or "time-multiplexed" address and data bus that permits a number of the pins to serve dual functions and consequently allows the complete CPU to be incorporated into a single, 40-pin package. As explained later in this chapter, a number of the CPU's control pins are defined according to the strapping of a single input pin (the MN/$\overline{MX}$ pin). In the "minimum mode," the CPU is configured for small, single-processor systems, and the CPU itself provides all control signals. In the "maximum mode," an Intel® 8288 Bus Controller, rather than the CPU, provides the control signal outputs and allows a number of the pins previously delegated to these control functions to be redefined in order to support multiprocessing applications. Figures 4-1 and 4-2 describe the pin assignments and signal definitions for the 8086 and 8088, respectively.

CPU Architecture

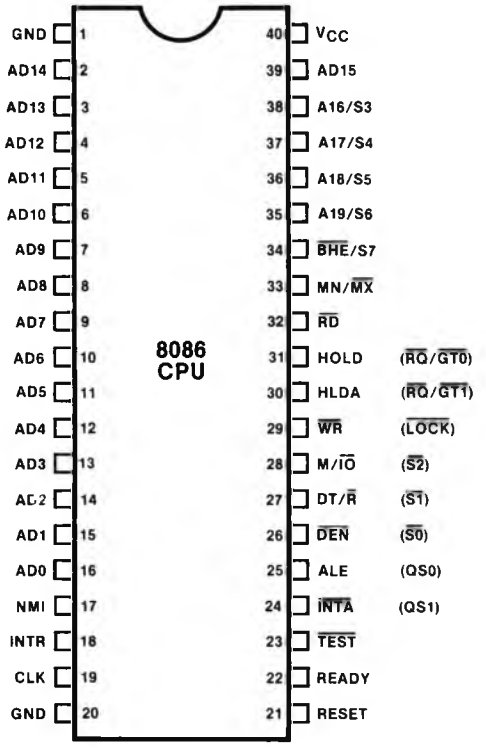
As shown in figures 4-3 and 4-4, both CPUs incorporate two separate processing units: the Execution Unit or "EU" and the Bus Interface Unit or "BIU." The EU for each processor is identical. The BIU for the 8086 incorporates a 16-bit data bus and a 6-byte instruction queue whereas the 8088 incorporates an 8-bit data bus and a 4-byte instruction queue.

The EU is responsible for the execution of all instructions, for providing data and addresses to the BIU, and for manipulating the general registers and the flag register. Except for a few control pins, the EU is completely isolated from the "outside world." The BIU is responsible for executing all external bus cycles and consists of the segment and communications registers, the instruction pointer and the instruction object code queue. The BIU combines segment and offset values in its dedicated adder to derive 20-bit addresses, transfers data to and from the EU on the ALU data bus and loads or "prefetches" instructions into the queue from which they are fetched by the EU.

The EU, when it is ready to execute an instruction, fetches the instruction object code byte from the BIU's instruction queue and then executes the instruction. If the queue is empty when the EU is ready to fetch an instruction byte, the EU waits for the instruction byte to be fetched. In the course of instruction execution, if a memory location or I/O port must be accessed, the EU requests the BIU to perform the required bus cycle.

The two processing sections of the CPU operate independently. In the 8086 CPU, when two or more bytes of the 6-byte instruction queue are empty and the EU does not require the BIU to perform a bus cycle, the BIU executes instruction fetch cycles to refill the queue. In the 8088 CPU, when one byte of the 4-byte instruction queue is empty, the BIU executes an instruction fetch cycle. Note that the 8086 CPU, since it has a 16-bit data bus, can access two instruction object code bytes in a single bus cycle, while the 8088 CPU, since it has an 8-bit data bus, accesses one instruction object code byte per bus cycle. If the EU issues a request for bus access while the BIU is in the process of an instruction fetch bus cycle, the BIU completes the cycle before honoring the EU's request.

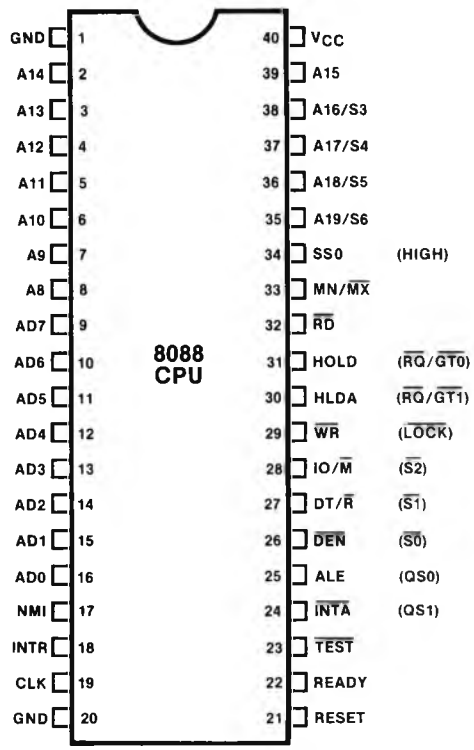
Common Signals		
Name	**Function**	**Type**
AD15-AD0	Address/Data Bus	Bidirectional, 3-State
A19/S6-A16/S3	Address/Status	Output, 3-State
$\overline{BHE}$/S7	Bus High Enable/ Status	Output, 3-State
MN/$\overline{MX}$	Minimum/Maximum Mode Control	Input
$\overline{RD}$	Read Control	Output, 3-State
$\overline{TEST}$	Wait On Test Control	Input
READY	Wait State Control	Input
RESET	System Reset	Input
NMI	Non-Maskable Interrupt Request	Input
INTR	Interrupt Request	Input
CLK	System Clock	Input
V_{CC}	+5V	Input
GND	Ground	
Minimum Mode Signals (MN/MX = V_{CC})		
Name	**Function**	**Type**
HOLD	Hold Request	Input
HLDA	Hold Acknowledge	Output
$\overline{WR}$	Write Control	Output, 3-State
M/$\overline{IO}$	Memory/IO Control	Output, 3-State
DT/$\overline{R}$	Data Transmit/ Receive	Output, 3-State
$\overline{DEN}$	Data Enable	Output, 3-State
ALE	Address Latch Enable	Output
$\overline{INTA}$	Interrupt Acknowledge	Output
Maximum Mode Signals (MN/MX = GND)		
Name	**Function**	**Type**
$\overline{RQ}$/$\overline{GT1, 0}$	Request/Grant Bus Access Control	Bidirectional
$\overline{LOCK}$	Bus Priority Lock Control	Output, 3-State
$\overline{S2}$-$\overline{S0}$	Bus Cycle Status	Output, 3-State
QS1, QS0	Instruction Queue Status	Output



MAXIMUM MODE PIN FUNCTIONS (e.g., $\overline{LOCK}$) ARE SHOWN IN PARENTHESES

Figure 4-1. 8086 Pin Definitions

Common Signals		
Name	**Function**	**Type**
AD7-AD0	Address/Data Bus	Bidirectional, 3-State
A15-A8	Address Bus	Output, 3-State
A19/S6-A16/S3	Address/Status	Output, 3-State
MN/$\overline{MX}$	Minimum/Maximum Mode Control	Input
$\overline{RD}$	Read Control	Output, 3-State
$\overline{TEST}$	Wait On Test Control	Input
READY	Wait State Control	Input
RESET	System Reset	Input
NMI	Non-Maskable Interrupt Request	Input
INTR	Interrupt Request	Input
CLK	System Clock	Input
V_{CC}	+5V	Input
GND	Ground	
Minimum Mode Signals (MN/MX = V_{CC})		
Name	**Function**	**Type**
HOLD	Hold Request	Input
HLDA	Hold Acknowledge	Output
$\overline{WR}$	Write Control	Output, 3-State
IO/$\overline{M}$	IO/Memory Control	Output, 3-State
DT/$\overline{R}$	Data Transmit/Receive	Output, 3-State
$\overline{DEN}$	Data Enable	Output, 3-State
ALE	Address Latch Enable	Output
$\overline{INTA}$	Interrupt Acknowledge	Output
SS0	S0 Status	Output, 3-State
Maximum Mode Signals (MN/MX = GND)		
Name	**Function**	**Type**
$\overline{RQ}$/$\overline{GT1, 0}$	Request/Grant Bus Access Control	Bidirectional
$\overline{LOCK}$	Bus Priority Lock Control	Output, 3-State
$\overline{S2}$-$\overline{S0}$	Bus Cycle Status	Output, 3-State
QS1, QS0	Instruction Queue Status	Output



MAXIMUM MODE PIN FUNCTIONS (e.g., $\overline{LOCK}$) ARE SHOWN IN PARENTHESES

Figure 4-2. 8088 Pin Definitions

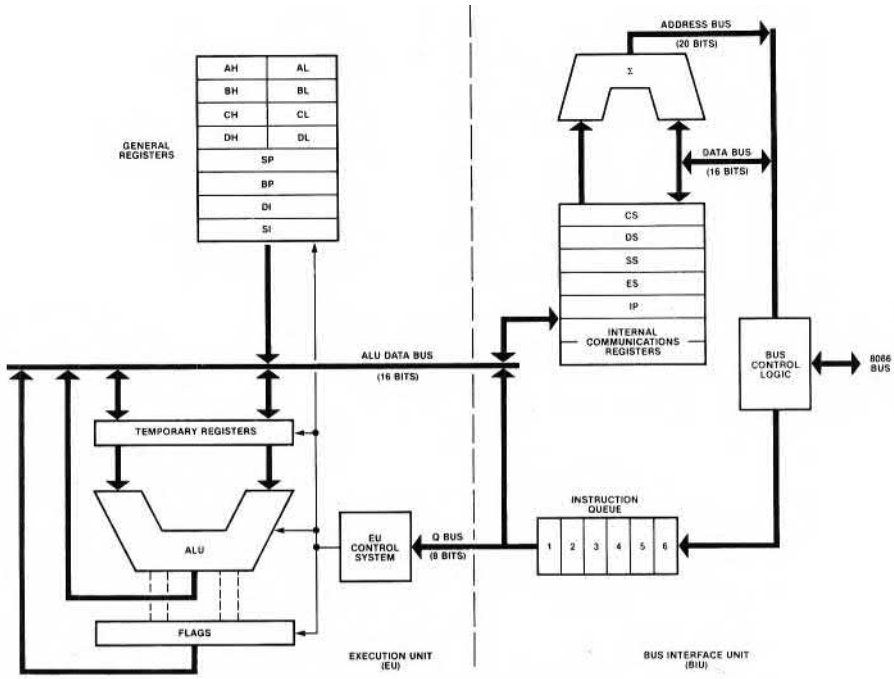


Figure 4-3. 8086 Elementary Block Diagram

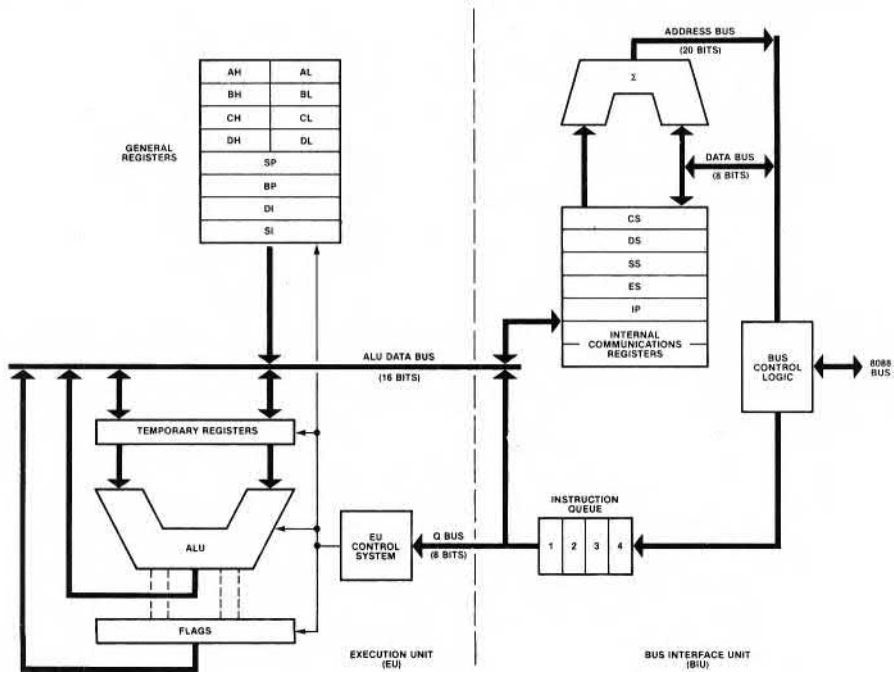


Figure 4-4. 8088 Elementary Block Diagram

Bus Operation

To explain the operation of the time-multiplexed bus, the BIU's bus cycle must be examined. Essentially, a bus cycle is an asynchronous event in which the address of an I/O peripheral or memory location is presented, followed by either a read control signal (to capture or "read" the data from the addressed device) or a write control signal and the associated data (to transmit or "write" the data to the addressed device). The selected device (memory or I/O peripheral) accepts the data on the bus during a write cycle or places the requested data on the bus during a read cycle. On termination of the cycle, the device latches the data written or removes the data read.

As shown in figure 4-5, all bus cycles consist of a minimum of four clock cycles or "T-states" identified as T_1, T_2, T_3 and T_4. The CPU places the address of the memory location or I/O device on the bus during state T_1. During a write bus cycle, the CPU places the data on the bus from state T_2 until state T_4. During a read bus cycle, the CPU accepts the data present on the bus in states T_3 and T_4, and the multiplexed address/data bus is floated in state T_2 to allow the CPU to change from the write mode (output address) to the read mode (input data).

It is important to note that the BIU executes a bus cycle only when a bus cycle is requested by the EU as part of instruction execution or when it must fill the instruction queue. Consequently, clock periods in which there is no BIU activity can occur between bus cycles. These inactive clock periods are referred to as idle states (T_I). While idle clock states result from several conditions (e.g., bus access granted to a coprocessor), as an example, consider the case of the execution of a "long" instruction. In the following example, an 8-bit register multiply (MUL) instruction (which requires between 70 and 77 clock cycles) is executed by the 8086. Assuming that the multiplication routine is entered as a result of a program jump (which causes the instruction queue to be reinitialized when the jump is executed) and, as will be explained later in this chapter, that the object code bytes are aligned on even-byte boundaries, the BIU's bus cycle sequence would appear as shown in figure 4-6.

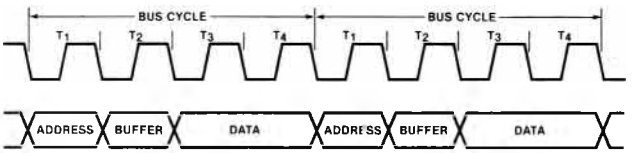


Figure 4-5. Typical BIU Bus Cycles

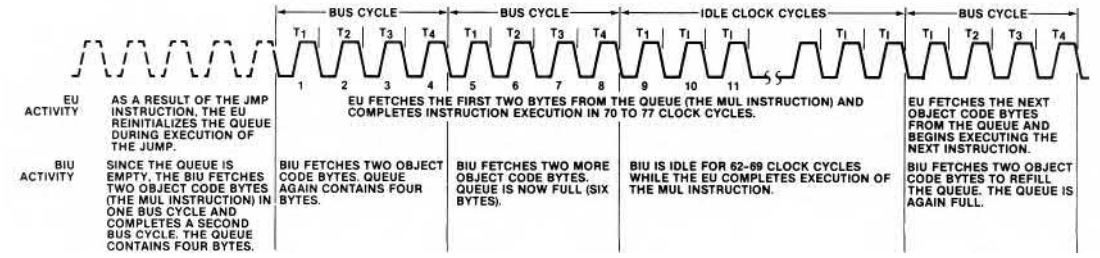


Figure 4-6. BIU Idle States

In addition to the idle state previously described, both the 8086 and 8088 CPUs include a mechanism for inserting additional T-states in the bus cycle to compensate for devices (memory or I/O) that cannot transfer data at the maximum rate. These extra T-states are called wait states (T_W) and, when required, are inserted between states T_3 and T_4. During a wait state, the data on the bus remains unchanged. When the device can complete the transfer (present or accept the data), it signals the CPU to exit the wait state and to enter state T_4.

As shown in the following timing diagrams, the actual bus cycle timing differs between a read and a write bus cycle and varies between the two CPUs. Note that the timing diagrams illustrated are for the minimum mode. (Maximum mode timing is described later in this chapter.)

Referring to figures 4-7 and 4-8, the 8086 CPU places a 20-bit address on the multiplexed address/data bus during state T_1. During state T_2, the CPU removes the address from the bus and either three-states (floats) the lower 16 address/data lines in preparation for a read cycle (figure 4-7) or places write data on these lines (figure 4-8). At this time, bus cycle status is available on the address/status lines. During state T_3, bus cycle status is maintained on the address/status lines and either the write data is maintained or read data is sampled on the lower 16 address/data lines. The bus cycle is terminated in state T_4 (control lines are disabled and the addressed device deselects from the bus).

The 8088 CPU, like the 8086, places a 20-bit address on the multiplexed address/data bus during state T_1 as shown in figures 4-9 and 4-10. Unlike the 8086, the 8088 maintains the address on the address lines (A_{15}-A_8) for the entire bus cycle. During state T_2, the CPU removes the address on the address/data lines (AD_7-AD_0) and either floats these lines in preparation for a read cycle (figure 4-9) or places write data on these lines (figure 4-10). At this time, bus cycle status is available on the address/status lines. During state T_3, bus cycle status is maintained on the address/status lines and either write data is maintained or read data is sampled on the address/data lines. The bus cycle is terminated in state T_4 (control lines are disabled and the addressed device deselects from the bus).

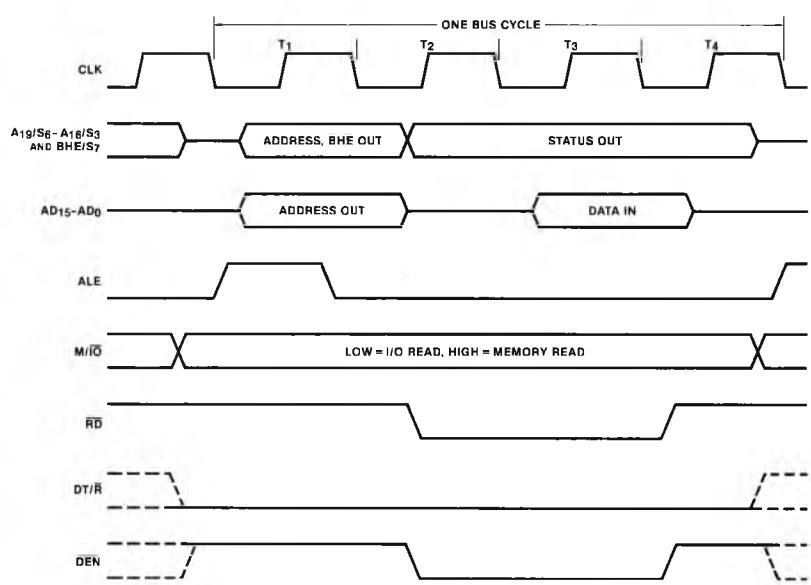


Figure 4-7. 8086 Read Bus Cycle

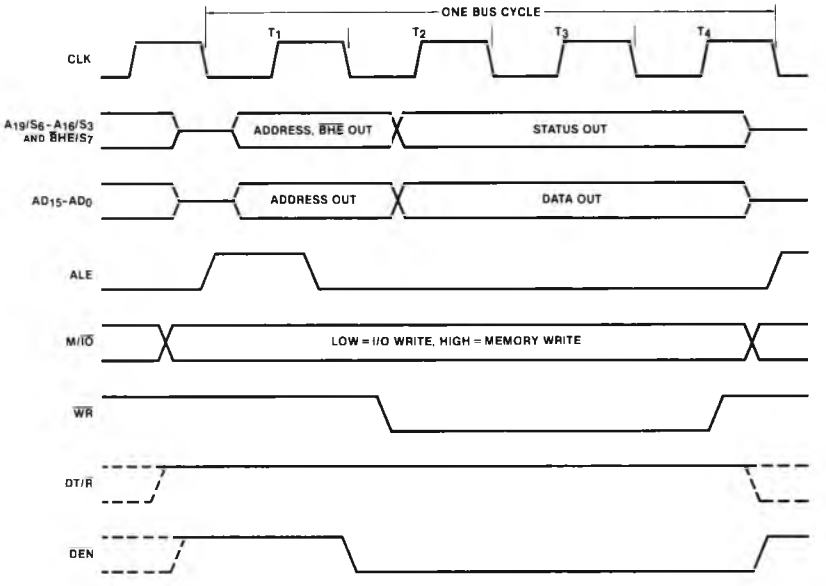


Figure 4-8. 8086 Write Bus Cycle

A majority of system memories and peripherals require a stable address for the duration of the bus cycle (certain MCS-85™ components can operate with a multiplexed address/data bus). During state T_1 of every bus cycle, the ALE (Address Latch Enable) control signal is output (either directly from the microprocessor in the minimum mode or indirectly through an 8288 Bus Controller in the maximum mode) to permit the address to be latched (the address is valid on the trailing-edge of ALE). This "demultiplexing" of the address/data bus can be done remotely at each device in the system or locally at the CPU and distributed throughout the system as a separate address bus. For optimum system performance and for compatibility with multiprocessor systems or with the Intel Multibus architecture, the locally-demultiplexed address bus is recommended. To latch the address, Intel® 8282 (non-inverting) or 8283 (inverting) Octal Latches are offered as part of the 8086 product family and are implemented as shown in figure 4-11. These circuits, in addition to providing the desired latch function, provide increased current drive capability and capacitive load immunity.

The data bus cannot be demultiplexed due to the timing differences between read and write cycles and the various read response times among peripherals and memories. Consequently, the multiplexed data bus either can be buffered or used directly. When memory and I/O peripherals are connected directly to an unbuffered bus, it is essential that during a read cycle, a device is prevented from corrupting the address present on the bus during state T_1. To ensure that the address is not corrupted, a device's output drivers should be enabled by an output enable function (rather than the device's chip select function) controlled by the CPU's read signal. (The MCS-86 family processors guarantee that the read signal will not be valid until after the address has been latched by ALE.) Many Intel peripheral, ROM/EPROM, and RAM circuits provide an output enable function to allow interface to an unbuffered multiplexed address/data bus. The alternative of using a buffered data bus should be considered since it simplifies the interfacing requirements and offers both increased drive current capability and capacitive load immunity. The Intel® 8286 (non-inverting) and 8287 (inverting)

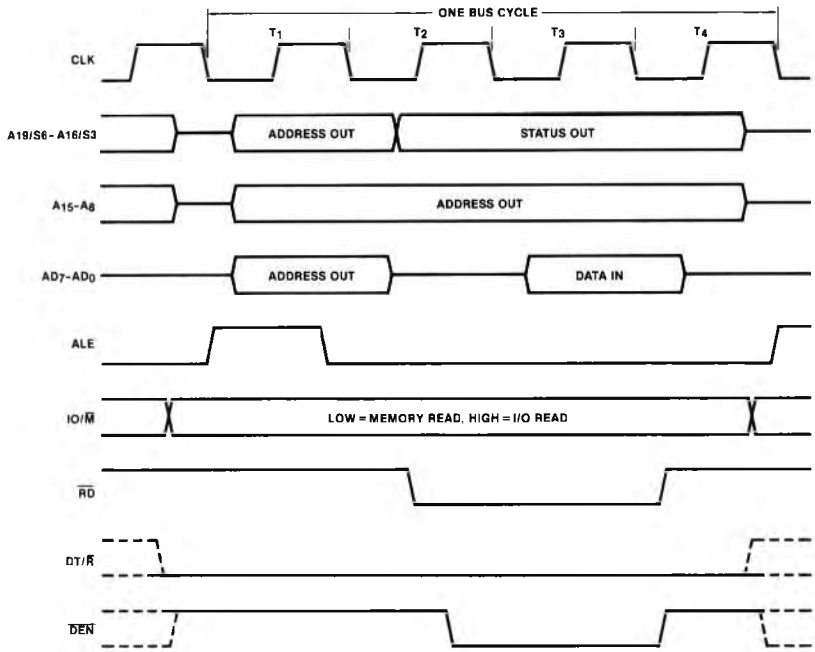


Figure 4-9. 8088 Read Bus Cycle

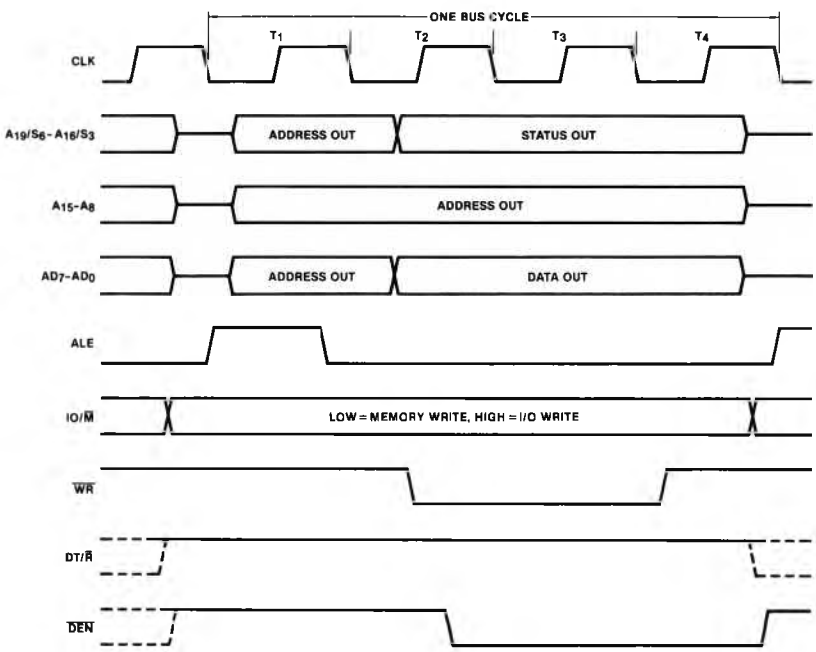


Figure 4-10. 8088 Write Bus Cycle

Octal Bus Transceivers, shown in figure 4-12, are expressly designed to buffer the data bus. These transceivers use the CPU's $\overline{DEN}$ (Data Enable) and DT/$\overline{R}$ (Data Transmit/Receive) control signals to enable and control the direction of data on the bus. These signals provide the proper timing relationship to guarantee isolation of the address that is present on the multiplexed bus during state T_1.

Except where noted, all subsequent discussions and examples in this chapter assume a locally demultiplexed address bus and a buffered data bus. The resultant address and data buses from the address latches and data transceivers to the memory and I/O devices will be referred to collectively as the "system" bus.

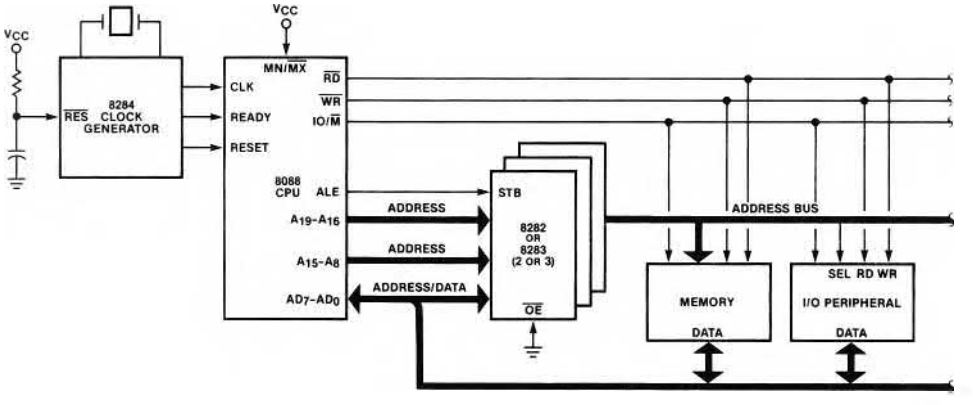


Figure 4-11. Minimum Mode 8088 Demultiplexed Address Bus

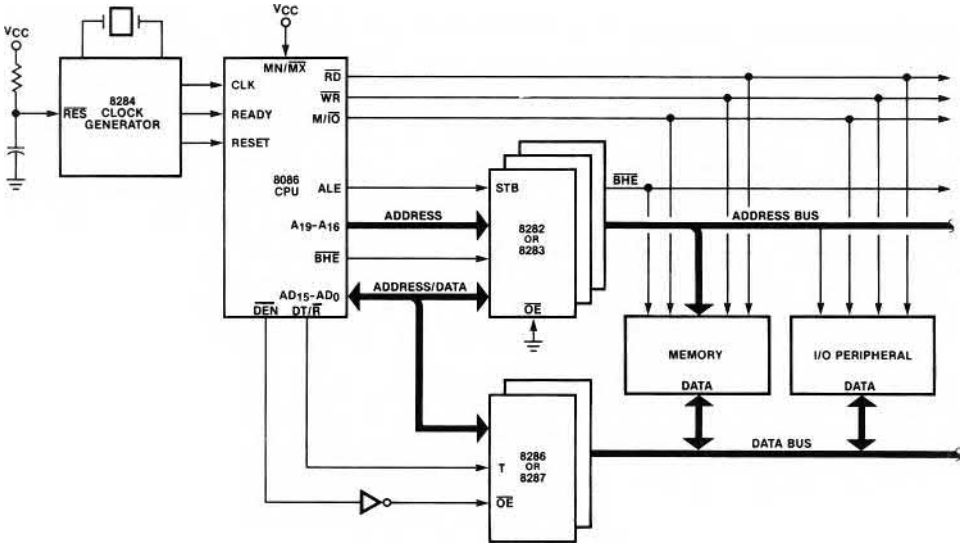


Figure 4-12. Minimum Mode 8086 Buffered Data Bus

Clock Circuit

To establish the bus cycle time, the CPU requires an external clock signal. As an integral part of the 8086 family, Intel offers the 8284 Clock Generator/Driver for this purpose. In addition to providing the primary (system) clock signal, this device provides both the hardware reset interface and the mechanism for the insertion of wait states in the bus cycle.

The clock generator/driver requires an external series-resonant crystal input (or external frequency source) at three times the required system clock frequency (i.e., to operate the CPU at 5 MHz, a 15 MHz fundamental frequency source is required). The divided-by-three output (CLK) from the 8284 is routed directly to the CPU's CLK input. The clock generator/driver provides a second clock output called PCLK (Peripheral Clock) at one half the frequency of the CLK output and a buffered TTL level OSC (oscillator) output at the applied crystal input frequency. These outputs are available for use by system devices.

The 8284's hardware reset function is accomplished with an internal Schmitt trigger circuit that is activated by the $\overline{RES}$ (Reset) input. When this input is pulled low (i.e., a contact closure to ground), the RESET output is activated synchronously with the CLK signal. This signal must be active for four clock cycles and causes the CPU to fetch and execute the instruction at location FFFF0H. An external RC circuit is connected to the RES input to provide the power-on reset function (on power-on, the $\overline{RES}$ input must be active for 50 microseconds). The RESET output is coupled directly to the RESET input of the CPU as well as being available to system peripherals as the system reset signal.

The insertion of wait states in the CPU's bus cycle is accomplished by deactivating one of the 8284's RDY inputs (RDY1 or RDY2). Either of these inputs, when enabled by its corresponding $\overline{AEN1}$ or $\overline{AEN2}$ input, can be deactivated directly by a peripheral device when it must extend the CPU's bus cycle (when it is not ready to present or accept data) or by a "wait state generator" circuit (a logic circuit that holds the RDY input inactive for a given number of clock cycles).

The READY output, which is synchronized to the CLK signal is coupled directly to the CPU's READY input. As shown in figure 4-13, when the addressed device needs to insert one or more wait states in a bus cycle, it deactivates the 8284's RDY input prior to the end of state T_2 which causes the READY output to be deactivated at the end of state T_2. The resultant wait state (T_W) is inserted between states T_3 and T_4. To exit the wait state, the device activates the 8284's RDY input which causes the READY input to the CPU to go active at the end of the current wait state and allows the CPU to enter state T_4.

Minimum/Maximum Mode

A unique feature of the 8086 and 8088 CPUs is the ability of a user to define a subset of the CPU's control signal outputs in order to tailor the CPU to its intended system environment. This "system tailoring" is accomplished by the strapping of the CPU's MN/$\overline{MX}$ (minimum/maximum) input pin. Table 4-1 defines the 8086 and 8088 pin assignments in both the minimum and maximum modes.

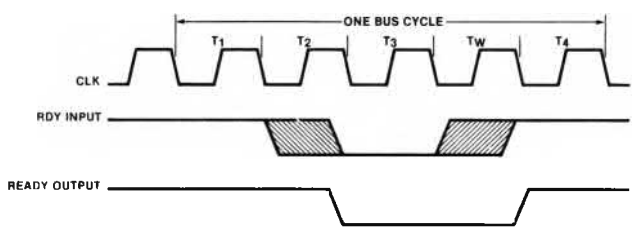


Figure 4-13. Wait State Timing

Table 4-1. Minimum/Maximum Mode Pin Assignments

8086			8088		
Pin	Mode		Pin	Mode	
	Minimum	Maximum		Minimum	Maximum
31	HOLD	$\overline{RQ}/\overline{GT0}$	31	HOLD	$\overline{RQ}/\overline{GT0}$
30	HLDA	$\overline{RQ}/\overline{GT1}$	30	HLDA	$\overline{RQ}/\overline{GT1}$
29	$\overline{WR}$	$\overline{LOCK}$	29	$\overline{WR}$	$\overline{LOCK}$
28	$M/\overline{IO}$	$\overline{S2}$	28	$IO/\overline{M}$	$\overline{S2}$
27	$DT/\overline{R}$	$\overline{S1}$	27	$DT/\overline{R}$	$\overline{S1}$
26	$\overline{DEN}$	$\overline{S0}$	26	$\overline{DEN}$	$\overline{S0}$
25	$\overline{ALE}$	QS0	25	$\overline{ALE}$	QS0
24	$\overline{INTA}$	QS1	24	$\overline{INTA}$	QS1
			34	SS0	High State

Minimum Mode

In the minimum mode ($MN/\overline{MX}$ pin strapped to +5V), the CPU supports small, single-processor systems that consist of a few devices and that use the system bus rather than support the Multibus™ architecture. In the minimum mode, the CPU itself generates all bus control signals ($DT/\overline{R}$, $\overline{DEN}$, ALE and either $M/\overline{IO}$ or $IO/\overline{M}$) and the command output signal ($\overline{RD}$, $\overline{WR}$ or $\overline{INTA}$), and provides a mechanism for requesting bus access (HOLD/HLDA) that is compatible with bus master type controllers (e.g., the Intel® 8237 and 8257 DMA Controllers).

In the minimum mode, when a bus master requires bus access, it activates the HOLD input to the CPU (through its request logic). The CPU, in response to the "hold" request, activates HLDA as an acknowledgement to the bus master requesting the bus and simultaneously floats the system bus and control lines. Since a bus request is asynchronous, the CPU samples the HOLD input on the positive transition of each CLK signal and, as shown in figure 4-14, activates HLDA at the end of either the current bus cycle (if a bus cycle is in progress) or idle clock period. The hold state is maintained until the bus master inactivates the HOLD input at which time the CPU regains control of the system bus. Note that during a "hold" state, the CPU will continue to execute instructions until a bus cycle is required.

Note that in the minimum mode, the I/O-memory control line for the 8088 CPU is the converse of the corresponding control line for the 8086 CPU ($M/\overline{IO}$ on the 8086 and $IO/\overline{M}$ on the 8088). This was done to provide the 8088 CPU, since it is an 8-bit device, compatibility with existing MCS-85™ systems and specific MCS-85™ family devices (e.g., the Intel® 8155/56).

Maximum Mode

In the maximum mode ($MN/\overline{MX}$ pin strapped to ground), an Intel® 8288 Bus Controller is added to provide a sophisticated bus control function and compatibility with the Multibus architecture (combining an Intel® 8289 Arbiter with the 8288 permits the CPU to support multiple processors on the system bus). As shown in figure 4-15, the bus controller, rather than the CPU, provides all bus control and command outputs, and allows the pins previously delegated to these functions to be redefined to support multiprocessing functions.

$\overline{S2}$, $\overline{S1}$ and $\overline{S0}$

Referring to figure 4-15, the 8288 Bus Controller uses the $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ status bit outputs from the CPU (and the 8089 IOP) to generate all bus control and command output signals required for a bus cycle. The status bit outputs are decoded as outlined in table 4-2. (For a detailed description of the operation of the 8288 Bus Controller, refer to the associated data sheet in Appendix B.)

The 8088 CPU, in the minimum mode, provides an SS0 status output. This output is equivalent to S0 in the maximum mode and can be decoded with $DT/\overline{R}$ and $IO/\overline{M}$ (inverted), which are equivalent to $\overline{S1}$ and $\overline{S2}$ respectively, to provide the same CPU cycle status information defined in table 4-2. This type of decoding could be used in a minimum mode 8088-based system to allow dynamic RAM refresh during passive CPU cycles.

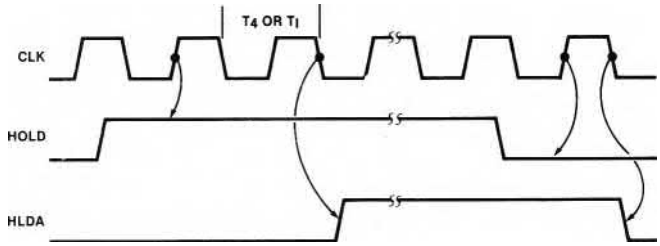


Figure 4-14. HOLD/HLDA Timing

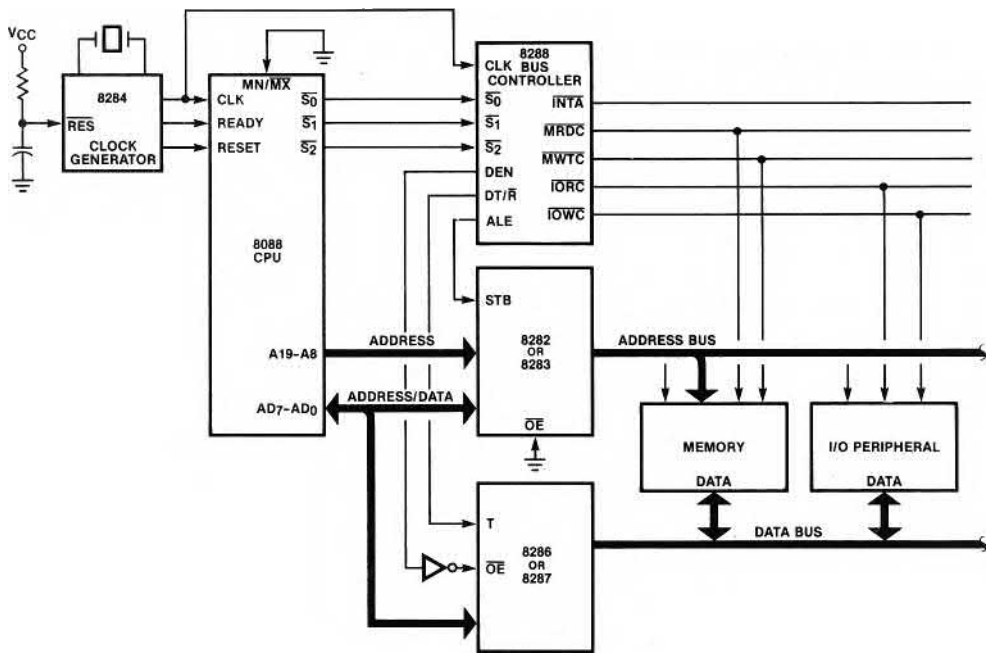


Figure 4-15. Elementary Maximum Mode System

Table 4-2. Status Bit Decoding

Status Inputs			CPU Cycle	8288 Command
$\overline{S2}$	$\overline{S1}$	$\overline{S0}$		
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Instruction Fetch	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

$\overline{RQ}/\overline{GT1}$, $\overline{RQ}/\overline{GT0}$

The Request/Grant signal lines ($\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$) provide the CPU's bus access mechanism in the maximum mode (replacing the HOLD/HLDA function available in the minimum mode) and are designed expressly for multiprocessor applications using the 8089 I/O Processor in its local mode or other processors that can support this function. These lines are unique in that the request/grant function is accomplished over a single line ($\overline{RQ}/\overline{GT0}$ or $\overline{RQ}/\overline{GT1}$) rather than the two-line HOLD/HLDA function.

As shown in figure 4-16, the request/grant sequence is a three-phase cycle: request, grant and release. The sequence is initiated by another processor on the system bus when it outputs a pulse on one of the $\overline{RQ}/\overline{GT}$ lines to request bus access (request phase). In response, the CPU outputs a pulse (on the same line) at the end of either the current bus cycle (if a bus cycle is in progress) or idle clock period to indicate to the requesting processor that it has floated the system bus and that it will logically disconnect from the bus controller on the next clock cycle (grant phase) and enter a "hold" state. Note that the CPU's execution unit (EU) continues to execute the instructions in the queue until an instruction requiring bus access is encountered or until the queue is empty. In the third (release) phase, the requesting processor again outputs a pulse on the $\overline{RQ}/\overline{GT}$ line. This pulse alerts the CPU that the processor is ready to release the bus. The CPU regains bus access on its next clock cycle. Note that the exchange of pulses is synchronized and, accordingly, both the CPU and requesting processor must be referenced to the same clock signal.

The request/grant lines are prioritized with $\overline{RQ}/\overline{GT0}$ taking precedence over $\overline{RQ}/\overline{GT1}$. If a request arrives on both lines simultaneously, the processor on $\overline{RQ}/\overline{GT0}$ is granted the bus (the request on $\overline{RQ}/\overline{GT1}$ is granted when the bus is released by the first processor following a one or two clock channel transfer delay). Both $\overline{RQ}/\overline{GT}$ lines (and the HOLD line in minimum mode) have a higher priority than a pending interrupt.

Request/grant latency (the time interval between the receipt of a request pulse and the return of a grant pulse) for several conditions is given in table 4-3.

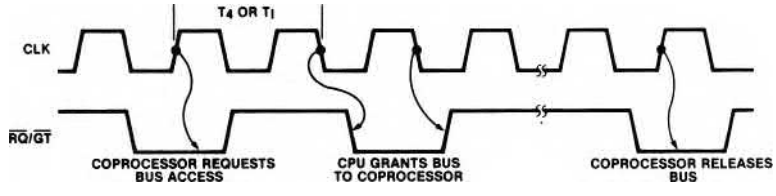


Figure 4-16. Request/Grant Timing

Table 4-3. Request/Grant Latency

Operating Condition	Request/Grant Delay	
	8086	8088
Normal Instruction Processing—$\overline{LOCK}$ inactive	3-6 (10*) clocks	3-10 clocks
$\overline{INTA}$ Cycle Executing—$\overline{LOCK}$ active	15 clocks	15 clocks
Locked XCHG Instruction Processing—$\overline{LOCK}$ active	24-31 (39*) clocks	24-39 clocks

*The number of clocks in parentheses applies when the instruction being executed references a word operand at an odd address boundary.

Latency during normal instruction processing ($\overline{LOCK}$ inactive) can be as short as three clock cycles (e.g., during execution of an instruction that does not reference memory) and no more than ten clock cycles. Whenever the $\overline{LOCK}$ output is active ($\overline{LOCK}$ is activated during an interrupt acknowledge cycle or during execution of an instruction with a Lock prefix), latency is increased. In the case of the execution of a locked XCHG instruction (used during semaphore examination), maximum latency is limited to 39 clock cycles. Greater latencies occur when a "long" instruction is locked. This, however, is neither necessary nor recommended.

At the end of processor activity, the 8086 or 8088 will not redirve its control and data buses until two clock cycles following receipt of the release pulse (or two clock cycles after HOLD goes inactive in the minimum mode).

A Hold request is honored immediately following CPU reset if the HOLD line is active when the RESET line goes inactive. This action facilitates the downloading of programs and, more specifically, the setting of memory location FFFF0H prior to CPU activation. Note that the same result can be effected in the maximum mode through the $\overline{RQ}/\overline{GT}$ line by generating the request pulse in the first or second clock cycle after RESET goes inactive.

$\overline{LOCK}$

The $\overline{LOCK}$ output is used in conjunction with an Intel 8289® Bus Arbiter to guarantee exclusive access of a shared system bus for the duration of an instruction. This output is software controlled and is effected by preceding the instruction requiring exclusive access with a one byte "lock" prefix (see instruction set description in Chapter 2).

When the lock prefix is decoded by the EU, the EU informs the BIU to activate the $\overline{LOCK}$ output during the next clock cycle. This signal remains active until one clock cycle after the execution of the associated instruction is concluded.

QS1, QS0

The QS1 and QS0 (Queue Status) outputs permit external monitoring of the CPU's internal instruction queue to allow instruction set extension processing by a coprocessor. (The corresponding Intel ICE modules use these status bits during "trace" operations.) The encoding of the QS1 and QS0 bits is shown in table 4-4.

Table 4-4. Queue Status Bit Decoding

QS1	QS0	Queue Status
0 (low)	0	No Operation. During the last clock cycle, nothing was taken from the queue.
0	1	First Byte. The byte taken from the queue was the first byte of the instruction.
1 (high)	0	Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction.
1	1	Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction.

The queue status is valid during the clock cycle after the indicated activity has occurred.

External Memory Addressing

The 8086 and 8088 CPUs have a 20-bit address bus and are capable of accessing one megabyte of memory address space.

The 8086 memory address space consists of a sequence of up to one million individual bytes in which any two consecutive bytes can be accessed as a 16-bit data word. As shown in figure 4-17, the memory address space is physically divided into two banks of up to 512k bytes each.

One bank is associated with the lower half of the CPU's 16-bit data bus (data bits D7-D0), and the other bank is associated with the upper half of the data bus (data bits D15-D8). Address bits A19 through A1 are used to simultaneously address a specific byte location in both the upper and lower banks, and the A0 address bit is *not* used in memory addressing. Instead, A0 is used in memory bank selection. The lower bank, which

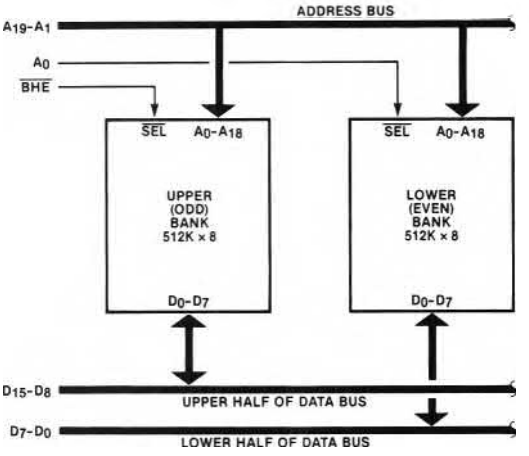


Figure 4-17. 8086 Memory Interface

contains even-address bytes, is selected when A0=0. The upper bank, containing odd address bytes (A0=1), is selected by a separate signal, Bus High Enable ($\overline{BHE}$). Table 4-5 defines the $\overline{BHE}$-A0 bank selection mechanism.

Table 4-5. Memory Bank Selection

$\overline{BHE}$	A0	Byte Transferred
0 (low)	0	Both bytes
0	1	Upper byte to/from odd address
1 (high)	0	Lower byte to/from even address
1	1	None

When accessing a data byte at an even address, the byte is transferred to or from the lower bank on the lower half of the data bus (D7-D0). In this case, the inactive level of the A0 address bit enables the addressed byte in the lower bank, and the inactive level of the $\overline{BHE}$ signal disables the addressed byte in the upper bank. Conversely, when performing a byte access at an odd address, the data byte is transferred to or from the upper bank on the upper half of the data bus (D15-D8). The active level of the $\overline{BHE}$ signal enables the upper bank, and the active level of the A0 address bit disables the lower bank.

As indicated in table 4-5, the 8086 can access a byte in both the upper and lower banks simultaneously as a 16-bit word. When the low-order byte of the word to be accessed is on an even address boundary (that is, when the low-order byte is in the lower bank), the word is said to be "aligned" and can be accessed in a single operation (a single bus cycle). As with the byte transfers previously described, address bits A19 through A1 address both banks, except that now $\overline{BHE}$ is active (selecting the upper bank) and A0 is inactive (selecting the lower bank) to access both bytes.

When the low-order byte of the word to be accessed is on an odd address boundary (when the low-order byte is in the upper bank), the word is "not aligned" and must be accessed in two bus cycles. During the first cycle, the low-order byte of the word is transferred to or from the upper bank as described for a byte access at an odd address (A0 and $\overline{BHE}$ active). The memory address is then incremented, which causes A0 to shift to an inactive level (selecting the lower bank), and a byte access at an even address is performed during the next bus cycle to transfer the word's high-order byte to or from the lower bank. The above sequence is initiated automatically by the 8086 whenever a word access at an odd address is performed. Also, the directing of the high- and low-order bytes of the 8086's internal word registers to the appropriate halves of the data bus is performed automatically and, except for the additional four clock cycles required to execute the second bus cycle, the entire operation is transparent to the program.

The 8088 memory address space is logically organized as a linear array of up to one million bytes. Since the 8088 uses an 8-bit-wide data bus, memory consists of a single bank. Address bit A0 is used to address memory, and a $\overline{BHE}$ signal is not provided.

Word (16-bit) operands can be located at odd- or even-address boundaries. The low-order byte of the word is stored in the lower-valued address location, and the high-order byte is stored in the next, higher-valued address location. The 8088 automatically executes two bus cycles when accessing word operands.

I/O Interfacing

The 8086 and 8088 CPUs support both I/O mapped I/O and memory mapped I/O. I/O mapped I/O permits an I/O device to reside in a separate address space (first 64k of address space), and the standard I/O instruction set is

available for device communications. Memory mapped I/O permits an I/O device to reside anywhere in memory and allows the complete CPU instruction set to be used for I/O operations.

The 8086 supports both 8-bit and 16-bit I/O devices. An 8-bit I/O device may be associated with either the upper or lower half of the data bus. (Assigning an equal number of devices to each half of the data bus distributes bus loading.) When an I/O device is assigned to the lower half of the bus (D7-D0), all I/O addresses must be even (A0 equal "0"), and when an I/O device is assigned to the upper half of the bus, all I/O addresses must be odd (A0 equal "1"). Note that since A0 always will be either a "1" or a "0" for a specific device, it cannot be used as an address input to select registers within the I/O device. When an I/O device on the upper half of the bus and an I/O device on the lower half of the bus are assigned addresses that differ only by the state of A0 (adjacent odd and even addresses), A0 and $\overline{BHE}$ both must be conditions of device selection to prevent a write operation to one device from overwriting data in the other device.

To permit data transfers to 16-bit I/O devices to be performed in a single bus cycle, the device is assigned an even address. To ensure that the I/O device is selected only for word transfers, A0 and $\overline{BHE}$ both must be conditions of device selection.

The 8088, since its data bus is eight bits wide, is designed to support 8-bit I/O devices and places no restrictions on odd or even addresses.

When the 8086 or the 8088 is operated in the minimum mode, the CPU's read and write commands ($\overline{RD}$ and $\overline{WR}$) are common for memory and I/O devices. If the memory and I/O address spaces overlap, device selection must be qualified by M/$\overline{IO}$ (8086) or IO/$\overline{M}$ (8088) to determine if the device is memory or I/O. This restriction does not apply to systems in which I/O and memory addresses do not overlap or to systems that use memory-mapped I/O exclusively. In the maximum mode, the CPU generates (through the bus controller) separate memory read/write and I/O read/write commands in place of the M/$\overline{IO}$ or IO/$\overline{M}$ signal. In a maximum mode system, an I/O device is assigned to an I/O address or to a memory address (memory mapped I/O) by connecting either the memory or I/O read/write command lines to the device's command inputs. When the I/O and memory address spaces overlap, device selection is determined by the appropriate read/write command set.

Interrupts

CPU interrupts can be software or hardware initiated. Software interrupts originate directly from program execution (i.e., execution of a breakpointed instruction) or indirectly through program logic (i.e., attempting to divide by zero). Hardware interrupts originate from external logic and are classified as either non-maskable or maskable. All interrupts, whether software or hardware initiated, result in the transfer of control to a new program location. A 256-entry vector table, which contains address pointers to the interrupt routines, resides in absolute locations 0 through 3FFH. Each entry in this table consists of two 16-bit address values (four bytes) that are loaded into the code segment (CS) and the instruction pointer (IP) registers as the interrupt routine address when an interrupt is accepted. Figure 4-18 illustrates the organization of the 256-entry vector table.

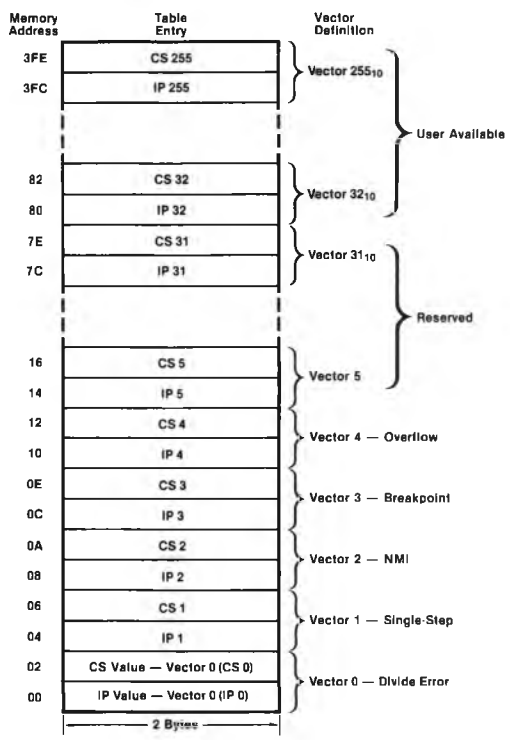


Figure 4-18. Interrupt Vector Table

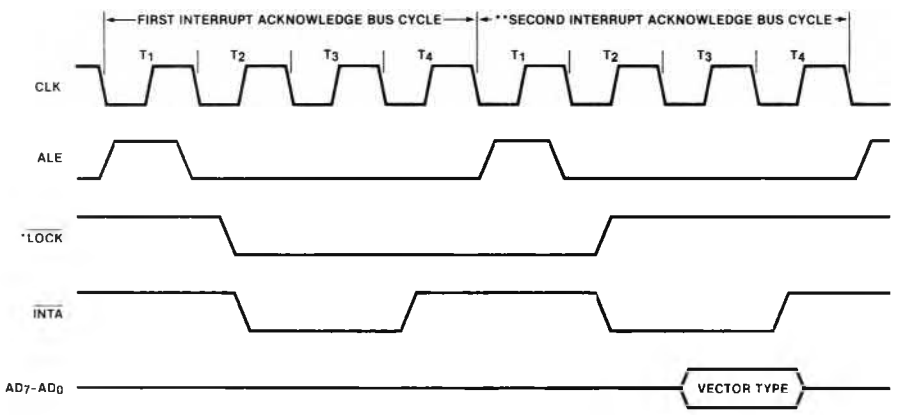
As shown in figure 4-18, the first five interrupt vectors are associated with the software-initiated interrupts and the hardware non-maskable interrupt (NMI). The next 27 interrupt vectors are reserved by Intel and should not be used if compatibility with future Intel products is to be maintained. The remaining interrupt vectors (vectors 32 thorugh 255) are available for user interrupt routines.

The non-maskable interrupt (NMI) occurs as a result of a positive transition at the CPU's NMI input pin. This input is asynchronous and, in order to ensure that it is recognized, is required to have a minimum duration of two clock cycles. NMI is typically used with power fail circuitry, error correcting memory or bus parity detection logic to allow fast response to these fault conditions. When NMI is activated, control is transferred to the interrupt service routine pointed to by vector 2 following execution of the current instruction. When a non-maskable interrupt is acknowledged, the current contents of the flags register are pushed onto the stack (the stack pointer is decremented by two), the interrupt enable and trap bits in the flags register are cleared (disabling maskable and single-step interrupts), and the vector 2 CS and IP address pointers are loaded into the CS and IP registers as the interrupt service routine address.

The CPU provides a single interrupt request input (INTR) that can be software masked by clearing the interrupt enable bit in the flags register through the execution of a CLI instruction. The INTR input is level triggered and is synchronized internally to the positive transition of the CLK signal. In order to be accepted before the next instruction, INTR must be active during the clock period preceding the end of the current instruction (and the interrupt enable bit must be set).

As shown in figure 4-19, when a maskable interrupt is acknowledged, the CPU executes two interrupt acknowledge bus cycles.

During the first bus cycle, the CPU floats the address/data bus and activates the $\overline{\text{INTA}}$ (Interrupt Acknowledge) command output during states T_2 through T_4. In the minimum mode, the CPU will not recognize a hold request from another bus master until the full interrupt acknowledge sequence is completed. In the maximum mode, the CPU activates the $\overline{\text{LOCK}}$ output from state T_2 of the first bus cycle until state T_2 of the second bus cycle to signal all 8289 Bus Arbiters in the system that the bus should not be accessed by any other processor. During the second bus cycle, the CPU again activates its $\overline{\text{INTA}}$ command output. In response to the



*MAXIMUM MODE ONLY
**SEVERAL (3 TYPICAL) IDLE CLOCK STATES OCCUR BETWEEN THE FIRST AND SECOND INTERRUPT ACKNOWLEDGE BUS CYCLES IN THE 8086 CPU (DURING THIS INTERVAL THE BUS IS DRIVEN). INTERRUPT ACKNOWLEDGE BUS CYCLES OCCUR BACK-TO-BACK IN THE 8088 CPU.

Figure 4-19. Interrupt Acknowledge Sequence

Mnemonics © Intel, 1978

second $\overline{\text{INTA}}$, the external interrupt system (e.g., an Intel® 8259A Programmable Interrupt Controller) places a byte on the data bus that identifies the source of the interrupt (the vector number or vector "type"). This byte is read by the CPU and then multiplied by four with the resultant value used as a pointer into the interrupt vector table. Before calling the corresponding interrupt routine, the CPU saves the machine status by pushing the current contents of the flags register onto the stack. The CPU then clears the interrupt enable and trap bits in the flags register to prevent subsequent maskable and single-step interrupts, and establishes the interrupt routine return linkage by pushing the current CS and IP register contents onto the stack before loading the new CS and IP register values from the vector table.

The four classes of interrupts are prioritized with software-initiated interrupts having the highest priority and with maskable and single-step interrupts sharing the lowest priority (see section 2.6). Since the CPU disables maskable and single-step interrupts when acknowledging any interrupt, if recognition of maskable interrupts or single-step operation is required as part of the interrupt routine, the routine first must set these bits.

The processing times for the various classes of interrupts are given in table 4-6. (These times also are included with the 8086/8088 instruction times cited in section 2.7.)

Table 4-6. Interrupt Processing Time

Interrupt Class	Processing Time
External Maskable Interrupt (INTR)	61 clocks
Non-Maskable Interrupt (NMI)	50 clocks
INT (with vector)	51 clocks
INT Type 3	52 clocks
INTO	53 clocks
Single Step	50 clocks

Note that the times shown in table 4-6 represent only the time required to process the interrupt request after it has been recognized. To determine interrupt latency (the time interval between the posting of the interrupt request and the execution of "useful" instructions within the interrupt routine), additional time must be included for the completion on an instruction being executed when the interrupt is posted (interrupts are generally processed only at instruction boundaries), for saving the contents of any additional registers prior to interrupt processing (interrupts automatically save only CS, IP and Flags) and for any wait states that may be incurred during interrupt processing.

Machine Instruction Encoding and Decoding

Writing a MOV instruction in ASM-86 in the form:

```
MOV destination,source
```

will cause the assembler to generate 1 of 28 possible forms of the MOV machine instruction. A programmer rarely needs to know the details of machine instruction formats or encoding. An exception may occur during debugging when it may be necessary to monitor instructions fetched on the bus, read unformatted memory dumps, etc. This section provides the information necessary to translate or decode an 8086 or 8088 machine instruction.

To pack instructions into memory as densely as possible, the 8086 and 8088 CPUs utilize an efficient coding technique. Machine instructions vary from one to six bytes in length. One-byte instructions, which generally operate on single registers or flags, are simple to identify. The keys to decoding longer instructions are in the first two bytes. The format of these bytes can vary, but most instructions follow the format shown in figure 4-20.

The first six bits of a multibyte instruction generally contain an opcode that identifies the basic instruction type: ADD, XOR, etc. The following bit, called the D field, generally specifies the "direction" of the operation: 1 = the REG field in the second byte identifies the destination operand, 0 = the REG field identifies the source operand. The W field distinguishes between byte and word operations: 0 = byte, 1 = word.

One of three additional single-bit fields, S, V or Z, appears in some instruction formats. S is used in conjunction with W to indicate sign extension

of immediate fields in arithmetic instructions. V distinguishes between single- and variable-bit shifts and rotates. Z is used as a compare bit with the zero flag in conditional repeat and loop instructions. All single-bit field settings are summarized in table 4-7.

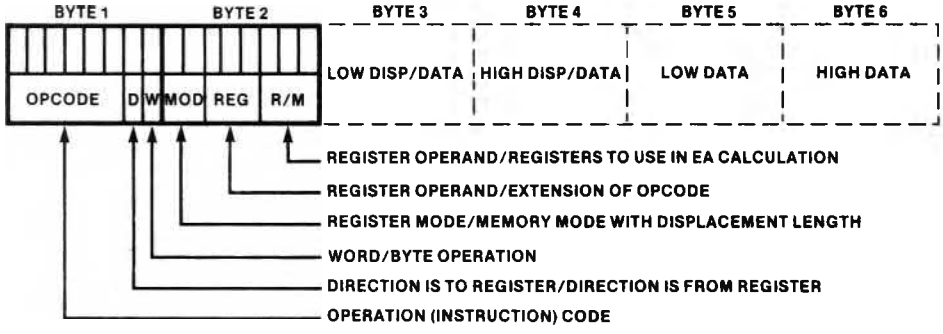


Figure 4-20. Typical 8086/8088 Machine Instruction Format

Table 4-7. Single-Bit Field Encoding

Field	Value	Function
S	0	No sign extension
	1	Sign extend 8-bit immediate data to 16 bits if W=1
W	0	Instruction operates on byte data
	1	Instruction operates on word data
D	0	Instruction source is specified in REG field
	1	Instruction destination is specified in REG field
V	0	Shift/rotate count is one
	1	Shift/rotate count is specified in CL register
Z	0	Repeat/loop while zero flag is clear
	1	Repeat/loop while zero flag is set

The second byte of the instruction usually identifies the instruction's operands. The MOD (mode) field indicates whether one of the operands is in memory or whether both operands are registers (see table 4-8). The REG (register) field identifies a register that is one of the instruction operands (see table 4-9). In a number of instructions, chiefly the immediate-to-memory variety, REG is used as an extension of the opcode to identify the type of operation. The encoding of the R/M (register/memory) field (see table 4-10) depends on how the mode field is set. If MOD = 11 (register-to-register mode), then R/M identifies the second register operand. If MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated. Effective address calculation is covered in detail in section 2.8.

Bytes 3 through 6 of an instruction are optional fields that usually contain the displacement value of a memory operand and/or the actual value of an immediate constant operand.

Table 4-8. MOD (Mode) Field Encoding

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

Table 4-9. REG (Register) Field Encoding

REG	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

There may be one or two displacement bytes; the language translators generate one byte whenever possible. The MOD field indicates how many displacement bytes are present. Following Intel convention, if the displacement is two bytes, the most-significant byte is stored second in the instruction. If the displacement is only a single byte, the 8086 or 8088 automatically sign-extends this quantity to 16-bits before using the information in further address calculations. Immediate values always follow any displacement values that may be present. The second byte of a two-byte immediate value is the most significant.

Table 4-12 lists the instruction encodings for all 8086/8088 instructions. This table can be used to predict the machine encoding of any ASM-86 instruction. Table 4-13 lists the 8086/8088 machine instructions in order by the binary value of their first byte. This table can be used to decode any machine instruction from its binary representation. Table 4-11 is a key to the abbreviations used in tables 4-12 and 4-13. Table 4-14 is a more compact instruction decoding guide.

Table 4-10. R/M (Register/Memory) Field Encoding

MOD=11			EFFECTIVE ADDRESS CALCULATION			
R/M	W=0	W=1	R/M	MOD=00	MOD=01	MOD=10
000	AL	AX	000	(BX)+(SI)	(BX)+(SI)+D8	(BX)+(SI)+D16
001	CL	CX	001	(BX)+(DI)	(BX)+(DI)+D8	(BX)+(DI)+D16
010	DL	DX	010	(BP)+(SI)	(BP)+(SI)+D8	(BP)+(SI)+D16
011	BL	BX	011	(BP)+(DI)	(BP)+(DI)+D8	(BP)+(DI)+D16
100	AH	SP	100	(SI)	(SI)+D8	(SI)+D16
101	CH	BP	101	(DI)	(DI)+D8	(DI)+D16
110	DH	SI	110	DIRECT ADDRESS	(BP)+D8	(BP)+D16
111	BH	DI	111	(BX)	(BX)+D8	(BX)+D16

Table 4-11. Key to Machine Instruction Encoding and Decoding

IDENTIFIER	EXPLANATION
MOD	Mode field; described in this chapter.
REG	Register field; described in this chapter.
R/M	Register/Memory field; described in this chapter.
SR	Segment register code: 00=ES, 01=CS, 10=SS, 11=DS.
W, S, D, V, Z	Single-bit instruction fields; described in this chapter.
DATA-8	8-bit immediate constant.
DATA-SX	8-bit immediate value that is automatically sign-extended to 16-bits before use.
DATA-LO	Low-order byte of 16-bit immediate constant.
DATA-HI	High-order byte of 16-bit immediate constant.
(DISP-LO)	Low-order byte of optional 8- or 16-bit unsigned displacement; MOD indicates if present.
(DISP-HI)	High-order byte of optional 16-bit unsigned displacement; MOD indicates if present.
IP-LO	Low-order byte of new IP value.
IP-HI	High-order byte of new IP value
CS-LO	Low-order byte of new CS value.
CS-HI	High-order byte of new CS value.
IP-INC8	8-bit signed increment to instruction pointer.
IP-INC-LO	Low-order byte of signed 16-bit instruction pointer increment.
IP-INC-HI	High-order byte of signed 16-bit instruction pointer increment.
ADDR-LO	Low-order byte of direct address (offset) of memory operand; EA not calculated.
ADDR-HI	High-order byte of direct address (offset) of memory operand; EA not calculated.
——	Bits may contain any value.
XXX	First 3 bits of ESC opcode.
YYY	Second 3 bits of ESC opcode.
REG8	8-bit general register operand.
REG16	16-bit general register operand.
MEM8	8-bit memory operand (any addressing mode).
MEM16	16-bit memory operand (any addressing mode).
IMMED8	8-bit immediate operand.
IMMED16	16-bit immediate operand.
SEGREG	Segment register operand.
DEST-STR8	Byte string addressed by DI.

Table 4-11. Key to Machine Instruction Encoding and Decoding (Cont'd.)

IDENTIFIER	EXPLANATION
SRC-STR8	Byte string addressed by SI.
DEST-STR16	Word string addressed by DI.
SRC-STR16	Word string addressed by SI.
SHORT-LABEL	Label within ±127 bytes of instruction.
NEAR-PROC	Procedure in current code segment.
FAR-PROC	Procedure in another code segment.
NEAR-LABEL	Label in current code segment but farther than −128 to +127 bytes from instruction.
FAR-LABEL	Label in another code segment.
SOURCE-TABLE	XLAT translation table addressed by BX.
OPCODE	ESC opcode operand.
SOURCE	ESC register or memory operand.

Table 4-12. 8086 Instruction Encoding

DATA TRANSFER

MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w = 1
Immediate to register	1 0 1 1 w reg	data	data if w = 1			
Memory to accumulator	1 0 1 0 0 0 0 w	addr-lo	addr-hi			
Accumulator to memory	1 0 1 0 0 0 1 w	addr-lo	addr-hi			
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 SR r/m	(DISP-LO)	(DISP-HI)		

PUSH = Push:				
Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			

POP = Pop:				
Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			

Table 4-12. 8086 Instruction Encoding (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
DATA TRANSFER (Cont'd.)						
XCHG = Exchange:						
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Register with accumulator	1 0 0 1 0 reg					
IN = Input from:						
Fixed port	1 1 1 0 0 1 0 w	DATA-8				
Variable port	1 1 1 0 1 1 0 w					
OUT = Output to:						
Fixed port	1 1 1 0 0 1 1 w	DATA-8				
Variable port	1 1 1 0 1 1 1 w					
XLAT = Translate byte to AL	1 1 0 1 0 1 1 1					
LEA = Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m	(DISP-LO)	(DISP-HI)		
LDS = Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m	(DISP-LO)	(DISP-HI)		
LES = Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m	(DISP-LO)	(DISP-HI)		
LAHF = Load AH with flags	1 0 0 1 1 1 1 1					
SAHF = Store AH into flags	1 0 0 1 1 1 1 0					
PUSHF = Push flags	1 0 0 1 1 1 0 0					
POPF = Pop flags	1 0 0 1 1 1 0 1					
ARITHMETIC						
ADD = Add:						
Reg/memory with register to either	0 0 0 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w=1			
ADC = Add with carry:						
Reg/memory with register to either	0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w=1			
INC = Increment:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)		
Register	0 1 0 0 0 reg					
AAA = ASCII adjust for add	0 0 1 1 0 1 1 1					
DAA = Decimal adjust for add	0 0 1 0 0 1 1 1					

Mnemonics © Intel, 1978

Table 4-12. 8086 Instruction Encoding (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ARITHMETIC (Cont'd.)						
SUB = Subtract:						
Reg/memory and register to either	0 0 1 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w=1			
SBB = Subtract with borrow:						
Reg/memory and register to either	0 0 0 1 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1			
DEC Decrement:						
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
Register	0 1 0 0 1 reg					
NEG Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		
CMP = Compare:						
Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)	data	data if s: w=1
Immediate with accumulator	0 0 1 1 1 1 0 w	data				
AAS ASCII adjust for subtract	0 0 1 1 1 1 1 1					
DAS Decimal adjust for subtract	0 0 1 0 1 1 1 1					
MUL Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
IMUL Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
AAM ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
DIV Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m	(DISP-LO)	(DISP-HI)		
IDIV Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
AAD ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0	(DISP-LO)	(DISP-HI)		
CBW Convert byte to word	1 0 0 1 1 0 0 0					
CWD Convert word to double word	1 0 0 1 1 0 0 1					
LOGIC						
NOT Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
SHL/SAL Shift logical/arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)		
SHR Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)		
SAR Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m	(DISP-LO)	(DISP-HI)		
ROL Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)		

Table 4-12. 8086 Instruction Encoding (Cont'd.)

LOGIC (Cont d.)	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)		
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

AND = And:						
Reg/memory with register to either	0 0 1 0 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w=1			

TEST = And function to flags no result:						
Register/memory and register	0 0 0 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data				

OR = Or:						
Reg/memory and register to either	0 0 0 0 1 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w=1			

XOR = Exclusive or:						
Reg/memory and register to either	0 0 1 1 0 0 d w	mod reg r/m	(DISP-LO)	(DISP-HI)		
Immediate to register/memory	0 0 1 1 0 1 0 w	data	(DISP-LO)	(DISP-HI)	data	data if w=1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w=1			

STRING MANIPULATION	
REP = Repeat	1 1 1 1 0 0 1 z
MOVS = Move byte/word	1 0 1 0 0 1 0 w
CMPS = Compare byte/word	1 0 1 0 0 1 1 w
SCAS = Scan byte/word	1 0 1 0 1 1 1 w
LODS = Load byte/wd to AL/AX	1 0 1 0 1 1 0 w
STDS = Stor byte/wd from AL/A	1 0 1 0 1 0 1 w

Mnemonics © Intel, 1978

Table 4-12. 8086 Instruction Encoding (Cont'd.)

CONTROL TRANSFER

CALL = Call:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	IP-INC-LO	IP-INC-HI			
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	(DISP-LO)	(DISP-HI)		
Direct intersegment	1 0 0 1 1 0 1 0	IP-lo	IP-hi			
		CS-lo	CS-hi			
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	(DISP-LO)	(DISP-HI)		

JMP = Unconditional Jump:				
Direct within segment	1 1 1 0 1 0 0 1	IP-INC-LO	IP-INC-HI	
Direct within segment-short	1 1 1 0 1 0 1 1	IP-INC8		
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	(DISP-LO)	(DISP-HI)
Direct intersegment	1 1 1 0 1 0 1 0	IP-lo	IP-hi	
		CS-lo	CS-hi	
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	(DISP-LO)	(DISP-HI)

RET = Return from CALL:			
Within segment	1 1 0 0 0 0 1 1		
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-lo	data-hi
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-lo	data-hi
JE/JZ = Jump on equal/zero	0 1 1 1 0 1 0 0	IP-INC8	
JL/JNGE = Jump on less/not greater or equal	0 1 1 1 1 1 0 0	IP-INC8	
JLE/JNG = Jump on less or equal/not greater	0 1 1 1 1 1 1 0	IP-INC8	
JB/JNAE = Jump on below/not above or equal	0 1 1 1 0 0 1 0	IP-INC8	
JBE/JNA = Jump on below or equal/not above	0 1 1 1 0 1 1 0	IP-INC8	
JP/JPE = Jump on parity/parity even	0 1 1 1 1 0 1 0	IP-INC8	
JO = Jump on overflow	0 1 1 1 0 0 0 0	IP-INC8	
JS = Jump on sign	0 1 1 1 1 0 0 0	IP-INC8	
JNE/JNZ = Jump on not equal/not zerO	0 1 1 1 0 1 0 1	IP-INC8	
JNL/JGE = Jump on not less/greater or equal	0 1 1 1 1 1 0 1	IP-INC8	
JNLE/JG = Jump on not less or equal/greater	0 1 1 1 1 1 1 1	IP-INC8	
JNB/JAE = Jump on not below/above or equal	0 1 1 1 0 0 1 1	IP-INC8	
JNBE/JA = Jump on not below or equal/above	0 1 1 1 0 1 1 1	IP-INC8	
JNP/JPO = Jump on not par/par odd	0 1 1 1 1 0 1 1	IP-INC8	
JNO = Jump on not overflow	0 1 1 1 0 0 0 1	IP-INC8	

Table 4-12. 8086 Instruction Encoding (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
CONTROL TRANSFER (Cont'd.)						
RET = Return from CALL:						
JNS = Jump on not sign	0 1 1 1 1 0 0 1	IP-INC8				
LOOP = Loop CX times	1 1 1 0 0 0 1 0	IP-INC8				
LOOPZ/LOOPE = Loop while zero/equal	1 1 1 0 0 0 0 1	IP-INC8				
LOOPNZ/LOOPNE = Loop while not zero/equal	1 1 1 0 0 0 0 0	IP-INC8				
JCXZ = Jump on CX zero	1 1 1 0 0 0 1 1	IP-INC8				
INT = Interrupt:						
Type specified	1 1 0 0 1 1 0 1	DATA-8				
Type 3	1 1 0 0 1 1 0 0					
INTO = Interrupt on overflow	1 1 0 0 1 1 1 0					
IRET = Interrupt return	1 1 0 0 1 1 1 1					
PROCESSOR CONTROL						
CLC = Clear carry	1 1 1 1 1 0 0 0					
CMC = Complement carry	1 1 1 1 0 1 0 1					
STC = Set carry	1 1 1 1 1 0 0 1					
CLD = Clear direction	1 1 1 1 1 1 0 0					
STD = Set direction	1 1 1 1 1 1 0 1					
CLI = Clear interrupt	1 1 1 1 1 0 1 0					
STI = Set interrupt	1 1 1 1 1 0 1 1					
HLT = Halt	1 1 1 1 0 1 0 0					
WAIT = Wait	1 0 0 1 1 0 1 1					
ESC = Escape (to external device)	1 1 0 1 1 x x x	m o d y y y r / m	(DISP-LO)	(DISP-HI)		
LOCK = Bus lock prefix	1 1 1 1 0 0 0 0					
SEGMENT = Override prefix	0 0 1 reg 1 1 0					

Table 4-13. Machine Instruction Decoding Guide

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3, 4, 5, 6	ASM-86 INSTRUCTION FORMAT	
00	0000 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8/MEM8,REG8
01	0000 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16/MEM16,REG16
02	0000 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG8,REG8/MEM8
03	0000 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADD	REG16,REG16/MEM16
04	0000 0100	DATA-8		ADD	AL,IMMED8
05	0000 0101	DATA-LO	DATA-HI	ADD	AX,IMMED16
06	0000 0110			PUSH	ES
07	0000 0111			POP	ES

Mnemonics © Intel, 1978

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
08	0000 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8/MEM8,REG8
09	0000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16/MEM16,REG16
0A	0000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG8,REG8/MEM8
0B	0000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	OR	REG16,REG16/MEM16
0C	0000 1100	DATA-8		OR	AL,IMMED8
0D	0000 1101	DATA-LO	DATA-HI	OR	AX,IMMED16
0E	0000 1110			PUSH	CS
0F	0000 1111			(not used)	
10	0001 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8/MEM8,REG8
11	0001 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16/MEM16,REG16
12	0001 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG8,REG8/MEM8
13	0001 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	ADC	REG16,REG16/MEM16
14	0001 0100	DATA-8		ADC	AL,IMMED8
15	0001 0101	DATA-LO	DATA-HI	ADC	AX,IMMED16
16	0001 0110			PUSH	SS
17	0001 0111			POP	SS
18	0001 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8/MEM8,REG8
19	0001 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16/MEM16,REG16
1A	0001 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG8,REG8/MEM8
1B	0001 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	SBB	REG16,REG16/MEM16
1C	0001 1100	DATA-8		SBB	AL,IMMED8
1D	0001 1101	DATA-LO	DATA-HI	SBB	AX,IMMED16
1E	0001 1110			PUSH	DS
1F	0001 1111			POP	DS
20	0010 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8/MEM8,REG8
21	0010 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16/MEM16,REG16
22	0010 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG8,REG8/MEM8
23	0010 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	AND	REG16,REG16/MEM16
24	0010 0100	DATA-8		AND	AL,IMMED8
25	0010 0101	DATA-LO	DATA-HI	AND	AX,IMMED16
26	0010 0110			ES:	(segment override prefix)
27	0010 0111			DAA	
28	0010 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8/MEM8,REG8
29	0010 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG16/MEM16,REG16
2A	0010 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	SUB	REG8,REG8/MEM8
2B	0010 1011	MOD REG R/M	(DISP-LO,(DISP-HI)	SUB	REG16,REG16/MEM16
2C	0010 1100	DATA-8		SUB	AL,IMMED8
2D	0010 1101	DATA-LO	DATA-HI	SUB	AX,IMMED16
2E	0010 1110			CS:	(segment override prefix)
2F	0010 1111			DAS	
30	0011 0000	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8/MEM8,REG8
31	0011 0001	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16/MEM16,REG16
32	0011 0010	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG8,REG8/MEM8
33	0011 0011	MOD REG R/M	(DISP-LO),(DISP-HI)	XOR	REG16,REG16/MEM16
34	0011 0100	DATA-8		XOR	AL,IMMED8
35	0011 0101	DATA-LO	DATA-HI	XOR	AX,IMMED16
36	0011 0110			SS:	(segment override prefix)

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
37	0011 0110			AAA	
38	0011 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG8/MEM8,REG8
39	0011 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG16/MEM16,REG16
3A	0011 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG8,REG8/MEM8
3B	0011 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	CMP	REG16,REG16/MEM16
3C	0011 1100	DATA-8		CMP	AL,IMMED8
3D	0011 1101	DATA-LO	DATA-HI	CMP	AX,IMMED16
3E	0011 1110			DS:	(segment override prefix)
3F	0011 1111			AAS	
40	0100 0000			INC	AX
41	0100 0001			INC	CX
42	0100 0010			INC	DX
43	0100 0011			INC	BX
44	0100 0100			INC	SP
45	0100 0101			INC	BP
46	0100 0110			INC	SI
47	0100 0111			INC	DI
48	0100 1000			DEC	AX
49	0100 1001			DEC	CX
4A	0100 1010			DEC	DX
4B	0100 1011			DEC	BX
4C	0100 1100			DEC	SP
4D	0100 1101			DEC	BP
4E	0100 1110			DEC	SI
4F	0100 1111			DEC	DI
50	0101 0000			PUSH	AX
51	0101 0001			PUSH	CX
52	0101 0010			PUSH	DX
53	0101 0011			PUSH	BX
54	0101 0100			PUSH	SP
55	0101 0101			PUSH	BP
56	0101 0110			PUSH	SI
57	0101 0111			PUSH	DI
58	0101 1000			POP	AX
59	0101 1001			POP	CX
5A	0101 1010			POP	DX
5B	0101 1011			POP	BX
5C	0101 1100			POP	SP
5D	0101 1101			POP	BP
5E	0101 1110			POP	SI
5F	0101 1111			POP	DI
60	0110 0000			(not used)	
61	0110 0001			(not used)	
62	0110 0010			(not used)	
63	0110 0011			(not used)	
64	0110 0100			(not used)	
65	0110 0101			(not used)	
66	0110 0110			(not used)	
67	0110 0111			(not used)	

Mnemonics © Intel, 1978

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT
68	0110 1000			(not used)
69	0110 1001			(not used)
6A	0110 1010			(not used)
6B	0110 1011			(not used)
6C	0110 1100			(not used)
6D	0110 1101			(not used)
6E	0110 1110			(not used)
6F	0110 1111			(not used)
70	0111 0000	IP-INC8		JO SHORT-LABEL
71	0111 0001	IP-INC8		JNO SHORT-LABEL
72	0111 0010	IP-INC8		JB/JNAE/JC SHORT-LABEL
73	0111 0011	IP-INC8		JNB/JAE/JNC SHORT-LABEL
74	0111 0100	IP-INC8		JE/JZ SHORT-LABEL
75	0111 0101	IP-INC8		JNE/JNZ SHORT-LABEL
76	0111 0110	IP-INC8		JBE/JNA SHORT-LABEL
77	0111 0111	IP-INC8		JNBE/JA SHORT-LABEL
78	0111 1000	IP-INC8		JS SHORT-LABEL
79	0111 1001	IP-INC8		JNS SHORT-LABEL
7A	0111 1010	IP-INC8		JP/JPE SHORT-LABEL
7B	0111 1011	IP-INC8		JNP/JPO SHORT-LABEL
7C	0111 1100	IP-INC8		JL/JNGE SHORT-LABEL
7D	0111 1101	IP-INC8		JNL/JGE SHORT-LABEL
7E	0111 1110	IP-INC8		JLE/JNG SHORT-LABEL
7F	0111 1111	IP-INC8		JNLE/JG SHORT-LABEL
80	1000 0000	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD REG8/MEM8,IMMED8
80	1000 0000	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-8	OR REG8/MEM8,IMMED8
80	1000 0000	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC REG8/MEM8,IMMED8
80	1000 0000	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB REG8/MEM8,IMMED8
80	1000 0000	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-8	AND REG8/MEM8,IMMED8
80	1000 0000	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB REG8/MEM8,IMMED8
80	1000 0000	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-8	XOR REG8/MEM8,IMMED8
80	1000 0000	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP REG8/MEM8,IMMED8
81	1000 0001	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADD REG16/MEM16,IMMED16
81	1000 0001	MOD 001 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	OR REG16/MEM16,IMMED16
81	1000 0001	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	ADC REG16/MEM16,IMMED16
81	1000 0001	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SBB REG16/MEM16,IMMED16

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
81	1000 0001	MOD 100 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	AND	REG16/MEM16,IMMED16
81	1000 0001	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	SUB	REG16/MEM16,IMMED16
81	1000 0001	MOD 110 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	XOR	REG16/MEM16,IMMED16
81	1000 0001	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	CMP	REG16/MEM16,IMMED16
82	1000 0010	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	ADD	REG8/MEM8,IMMED8
82	1000 0010	MOD 001 R/M		(not used)	
82	1000 0010	MOD 010 R/M	(DISP-LO),(DISP-HI), DATA-8	ADC	REG8/MEM8,IMMED8
82	1000 0010	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-8	SBB	REG8/MEM8,IMMED8
82	1000 0010	MOD 100 R/M		(not used)	
82	1000 0010	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-8	SUB	REG8/MEM8,IMMED8
82	1000 0010	MOD 110 R/M		(not used)	
82	1000 0010	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-8	CMP	REG8/MEM8,IMMED8
83	1000 0011	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-SX	ADD	REG16/MEM16, IMMED8
83	1000 0011	MOD 001 R/M		(not used)	
83	1000 0011	MOD 010 R/M	(DISP-LO), (DISP-HI), DATA-SX	ADC	REG16/MEM16,IMMED8
83	1000 0011	MOD 011 R/M	(DISP-LO),(DISP-HI), DATA-SX	SBB	REG16/MEM16,IMMED8
83	1000 0011	MOD 100 R/M		(not used)	
83	1000 0011	MOD 101 R/M	(DISP-LO),(DISP-HI), DATA-SX	SUB	REG16/MEM16,IMMED8
83	1000 0011	MOD 110 R/M		(not used)	
83	1000 0011	MOD 111 R/M	(DISP-LO),(DISP-HI), DATA-SX	CMP	REG16/MEM16,IMMED8
84	1000 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG8/MEM8,REG8
85	1000 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	TEST	REG16/MEM16,REG16
86	1000 0110	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG8,REG8/MEM8
87	1000 0111	MOD REG R/M	(DISP-LO),(DISP-HI)	XCHG	REG16,REG16/MEM16
88	1000 1000	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8/MEM8,REG8
89	1000 1001	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16/REG16
8A	1000 1010	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG8,REG8/MEM8
8B	1000 1011	MOD REG R/M	(DISP-LO),(DISP-HI)	MOV	REG16,REG16/MEM16
8C	1000 1100	MOD 0SR R/M	(DISP-LO),(DISP-HI)	MOV	REG16/MEM16,SEGREG
8C	1000 1100	MOD 1-- R/M		(not used)	
8D	1000 1101	MOD REG R/M	(DISP-LO),(DISP-HI)	LEA	REG16,MEM16
8E	1000 1110	MOD 0SR R/M	(DISP-LO),(DISP-HI)	MOV	SEGREG,REG16/MEM16
8E	1000 1110	MOD 1-- R/M		(not used)	
8F	1000 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	POP	REG16/MEM16
8F	1000 1111	MOD 001 R/M		(not used)	
8F	1000 1111	MOD 010 R/M		(not used)	

Mnemonics © Intel, 1978

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
8F	1000 1111	MOD 011 R/M		(not used)	
8F	1000 1111	MOD 100 R/M		(not used)	
8F	1000 1111	MOD 101 R/M		(not used)	
8F	1000 1111	MOD 110 R/M		(not used)	
8F	1000 1111	MOD 111 R/M		(not used)	
90	1001 0000			NOP	(exchange AX,AX)
91	1001 0001			XCHG	AX,CX
92	1001 0010			XCHG	AX,DX
93	1001 0011			XCHG	AX,BX
94	1001 0100			XCHG	AX,SP
95	1001 0101			XCHG	AX,BP
96	1001 0110			XCHG	AX,SI
97	1001 0111			XCHG	AX,DI
98	1001 1000			CBW	
99	1001 1001			CWD	
9A	1001 1010	DISP-LO	DISP-HI,SEG-LO, SEG-HI	CALL	FAR_PROC
9B	1001 1011			WAIT	
9C	1001 1100			PUSHF	
9D	1001 1101			POPF	
9E	1001 1110			SAHF	
9F	1001 1111			LAHF	
A0	1010 0000	ADDR-LO	ADDR-HI	MOV	AL,MEM8
A1	1010 0001	ADDR-LO	ADDR-HI	MOV	AX,MEM16
A2	1010 0010	ADDR-LO	ADDR-HI	MOV	MEM8,AL
A3	1010 0011	ADDR-LO	ADDR-HI	MOV	MEM16,AL
A4	1010 0100			MOVS	DEST-STR8,SRC-STR8
A5	1010 0101			MOVS	DEST-STR16,SRC-STR16
A6	1010 0110			CMPS	DEST-STR8,SRC-STR8
A7	1010 0111			CMPS	DEST-STR16,SRC-STR16
A8	1010 1000	DATA-8		TEST	AL,IMMED8
A9	1010 1001	DATA-LO	DATA-HI	TEST	AX,IMMED16
AA	1010 1010			STOS	DEST-STR8
AB	1010 1011			STOS	DEST-STR16
AC	1010 1100			LODS	SRC-STR8
AD	1010 1101			LODS	SRC-STR16
AE	1010 1110			SCAS	DEST-STR8
AF	1010 1111			SCAS	DEST-STR16
B0	1011 0000	DATA-8		MOV	AL,IMMED8
B1	1011 0001	DATA-8		MOV	CL,IMMED8
B2	1011 0010	DATA-8		MOV	DL,IMMED8
B3	1011 1011	DATA-8		MOV	BL,IMMED8
B4	1011 0100	DATA-8		MOV	AH,IMMED8
B5	1011 0101	DATA-8		MOV	CH,IMMED8
B6	1011 0110	DATA-8		MOV	DH,IMMED8
B7	1011 0111	DATA-8		MOV	BH,IMMED8
B8	1011 1000	DATA-LO	DATA-HI	MOV	AX,IMMED16
B9	1011 1001	DATA-LO	DATA-HI	MOV	CX,IMMED16
BA	1011 1010	DATA-LO	DATA-HI	MOV	DX,IMMED16
BB	1011 1011	DATA-LO	DATA-HI	MOV	BX,IMMED16

Mnemonics © Intel, 1978

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE		2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
HEX	BINARY				
BC	1011 1100	DATA-LO	DATA-HI	MOV	SP,IMMED16
BD	1011 1101	DATA-LO	DATA-HI	MOV	BP,IMMED16
BE	1011 1110	DATA-LO	DATA-HI	MOV	SI,IMMED16
BF	1011 1111	DATA-LO	DATA-HI	MOV	DI,IMMED16
C0	1100 0000			(not used)	
C1	1100 0001			(not used)	
C2	1100 0010	DATA-LO	DATA-HI	RET	IMMED16 (intraseg)
C3	1100 0011			RET	(intrasegment)
C4	1100 0100	MOD REG R/M	(DISP-LO),(DISP-HI)	LES	REG16,MEM16
C5	1100 0101	MOD REG R/M	(DISP-LO),(DISP-HI)	LDS	REG16,MEM16
C6	1100 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	MOV	MEM8,IMMED8
C6	1100 0110	MOD 001 R/M		(not used)	
C6	1100 0110	MOD 010 R/M		(not used)	
C6	1100 0110	MOD 011 R/M		(not used)	
C6	1100 0110	MOD 100 R/M		(not used)	
C6	1100 0110	MOD 101 R/M		(not used)	
C6	1100 0110	MOD 110 R/M		(not used)	
C6	1100 0110	MOD 111 R/M		(not used)	
C7	1100 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	MOV	MEM16,IMMED16
C7	1100 0111	MOD 001 R/M		(not used)	
C7	1100 0111	MOD 010 R/M		(not used)	
C7	1100 0111	MOD 011 R/M		(not used)	
C7	1100 0111	MOD 100 R/M		(not used)	
C7	1100 0111	MOD 101 R/M		(not used)	
C7	1100 0111	MOD 110 R/M		(not used)	
C7	1100 0111	MOD 111 R/M		(not used	
C8	1100 1000			(not used)	
C9	1100 1001			(not used)	
CA	1100 1010	DATA-LO	DATA-HI	RET	IMMED16 (intersegment)
CB	1100 1011			RET	(intersegment)
CC	1100 1100			INT	3
CD	1100 1101	DATA-8		INT	IMMED8
CE	1100 1110			INTO	
CF	1100 1111			IRET	
D0	1101 0000	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG8/MEM8,1
D0	1101 0000	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG8/MEM8,1
D0	1101 0000	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG8/MEM8,1
D0	1101 0000	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG8/MEM8,1
D0	1101 0000	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG8/MEM8,1
D0	1101 0000	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG8/MEM8,1
D0	1101 0000	MOD 110 R/M		(not used)	
D0	1101 0000	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR	REG8/MEM8,1
D1	1101 0001	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG16/MEM16,1
D1	1101 0001	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG16/MEM16,1
D1	1101 0001	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG16/MEM16,1
D1	1101 0001	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG16/MEM16,1
D1	1101 0001	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG16/MEM16,1

Mnemonics © Intel, 1978

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
D1	1101 0001	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG16/MEM16,1
D1	1101 0001	MOD 110 R/M		(not used)	
D1	1101 0001	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR	REG16/MEM16,1
D2	1101 0010	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG8/MEM8,CL
D2	1101 0010	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG8/MEM8,CL
D2	1101 0010	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG8/MEM8,CL
D2	1101 0010	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG8/MEM8,CL
D2	1101 0010	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG8/MEM8,CL
D2	1101 0010	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG8/MEM8,CL
D2	1101 0010	MOD 110 R/M		(not used)	
D2	1101 0010	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR	REG8/MEM8,CL
D3	1101 0011	MOD 000 R/M	(DISP-LO),(DISP-HI)	ROL	REG16/MEM16,CL
D3	1101 0011	MOD 001 R/M	(DISP-LO),(DISP-HI)	ROR	REG16/MEM16,CL
D3	1101 0011	MOD 010 R/M	(DISP-LO),(DISP-HI)	RCL	REG16/MEM16,CL
D3	1101 0011	MOD 011 R/M	(DISP-LO),(DISP-HI)	RCR	REG16/MEM16,CL
D3	1101 0011	MOD 100 R/M	(DISP-LO),(DISP-HI)	SAL/SHL	REG16/MEM16,CL
D3	1101 0011	MOD 101 R/M	(DISP-LO),(DISP-HI)	SHR	REG16/MEM16,CL
D3	1101 0011	MOD 110 R/M		(not used)	
D3	1101 0011	MOD 111 R/M	(DISP-LO),(DISP-HI)	SAR	REG16/MEM16,CL
D4	1101 0100	00001010		AAM	
D5	1101 0101	00001010		AAD	
D6	1101 0110			(not used)	
D7	1101 0111			XLAT	SOURCE-TABLE
D8	1101 1000	MOD 000 R/M			
	1XXX	MOD YYY R/M	(DISP-LO), (DISP-HI)	ESC	OPCODE,SOURCE
DF	1101 1111	MOD 111 R/M			
E0	1110 0000	IP-INC-8		LOOPNE/ LOOPNZ	SHORT-LABEL
E1	1110 0001	IP-INC-8		LOOPE/ LOOPZ	SHORT-LABEL
E2	1110 0010	IP-INC-8		LOOP	SHORT-LABEL
E3	1110 0011	IP-INC-8		JCXZ	SHORT-LABEL
E4	1110 0100	DATA-8		IN	AL,IMMED8
E5	1110 0101	DATA-8		IN	AX,IMMED8
E6	1110 0110	DATA-8		OUT	AL,IMMED8
E7	1110 0111	DATA-8		OUT	AX,IMMED8
E8	1110 1000	IP-INC-LO	IP-INC-HI	CALL	NEAR-PROC
E9	1110 1001	IP-INC-LO	IP-INC-HI	JMP	NEAR-LABEL
EA	1110 1010	IP-LO	IP-HI,CS-LO,CS-HI	JMP	FAR-LABEL
EB	1110 1011	IP-INC8		JMP	SHORT-LABEL
EC	1110 1100			IN	AL,DX
ED	1110 1101			IN	AX,DX
EE	1110 1110			OUT	AL,DX
EF	1110 1111			OUT	AX,DX
F0	1111 0000			LOCK	(prefix)
F1	1111 0001			(not used)	
F2	1111 0010			REPNE/REPNZ	
F3	1111 0011			REP/REPE/REPZ	
F4	1111 0100			HLT	
F5	1111 0101			CMC	

Table 4-13. Machine Instruction Decoding Guide (Cont'd.)

1ST BYTE HEX	1ST BYTE BINARY	2ND BYTE	BYTES 3,4,5,6	ASM-86 INSTRUCTION FORMAT	
F6	1111 0110	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-8	TEST	REG8/MEM8,IMMED8
F6	1111 0110	MOD 001 R/M		(not used)	
F6	1111 0110	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT	REG8/MEM8
F6	1111 0110	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG	REG8/MEM8
F6	1111 0110	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL	REG8/MEM8
F6	1111 0110	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL	REG8/MEM8
F6	1111 0110	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV	REG8/MEM8
F6	1111 0110	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV	REG8/MEM8
F7	1111 0111	MOD 000 R/M	(DISP-LO),(DISP-HI), DATA-LO,DATA-HI	TEST	REG16/MEM16,IMMED16
F7	1111 0111	MOD 001 R/M		(not used)	
F7	1111 0111	MOD 010 R/M	(DISP-LO),(DISP-HI)	NOT	REG16/MEM16
F7	1111 0111	MOD 011 R/M	(DISP-LO),(DISP-HI)	NEG	REG16/MEM16
F7	1111 0111	MOD 100 R/M	(DISP-LO),(DISP-HI)	MUL	REG16/MEM16
F7	1111 0111	MOD 101 R/M	(DISP-LO),(DISP-HI)	IMUL	REG16/MEM16
F7	1111 0111	MOD 110 R/M	(DISP-LO),(DISP-HI)	DIV	REG16/MEM16
F7	1111 0111	MOD 111 R/M	(DISP-LO),(DISP-HI)	IDIV	REG16/MEM16
F8	1111 1000			CLC	
F9	1111 1001			STC	
FA	1111 1010			CLI	
FB	1111 1011			STI	
FC	1111 1100			CLD	
FD	1111 1101			STD	
FE	1111 1110	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC	REG8/MEM8
FE	1111 1110	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC	REG8/MEM8
FE	1111 1110	MOD 010 R/M		(not used)	
FE	1111 1110	MOD 011 R/M		(not used)	
FE	1111 1110	MOD 100 R/M		(not used)	
FE	1111 1110	MOD 101 R/M		(not used)	
FE	1111 1110	MOD 110 R/M		(not used)	
FE	1111 1110	MOD 111 R/M		(not used)	
FF	1111 1111	MOD 000 R/M	(DISP-LO),(DISP-HI)	INC	MEM16
FF	1111 1111	MOD 001 R/M	(DISP-LO),(DISP-HI)	DEC	MEM16
FF	1111 1111	MOD 010 R/M	(DISP-LO),(DISP-HI)	CALL	REG16/MEM16 (intra)
FF	1111 1111	MOD 011 R/M	(DISP-LO),(DISP-HI)	CALL	MEM16 (intersegment)
FF	1111 1111	MOD 100 R/M	(DISP-LO),(DISP-HI)	JMP	REG16/MEM16 (intra)
FF	1111 1111	MOD 101 R/M	(DISP-LO),(DISP-HI)	JMP	MEM16 (intersegment)
FF	1111 1111	MOD 110 R/M	(DISP-LO),(DISP-HI)	PUSH	MEM16
FF	1111 1111	MOD 111 R/M		(not used)	

Mnemonics © Intel, 1978

Table 4-14. Machine Instruction Encoding Matrix

Hi \ Lo	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD b,f,r/m	ADD w,f,r/m	ADD b,t,r/m	ADD w,t,r/m	ADD b,ia	ADD w,ia	PUSH ES	POP ES	OR b,f,r/m	OR w,f,r/m	OR b,t,r/m	OR w,t,r/m	OR b,i	OR w,i	PUSH CS	
1	ADC b,f,r/m	ADC w,f,r/m	ADC b,t,r/m	ADC w,t,r/m	ADC b,i	ADC w,i	PUSH SS	POP SS	SBB b,f,r/m	SBB w,f,r/m	SBB b,t,r/m	SBB w,t,r/m	SBB b,i	SBB w,i	PUSH DS	POP DS
2	AND b,f,r/m	AND w,f,r/m	AND b,t,r/m	AND w,t,r/m	AND b,i	AND w,i	SEG ES	DAA	SUB b,f,r/m	SUB w,f,r/m	SUB b,t,r/m	SUB w,t,r/m	SUB b,i	SUB w,i	SEG CS	DAS
3	XOR b,f,r/m	XOR w,f,r/m	XOR b,t,r/m	XOR w,t,r/m	XOR b,i	XOR w,i	SEG SS	AAA	CMP b,f,r/m	CMP w,f,r/m	CMP b,t,r/m	CMP w,t,r/m	CMP b,i	CMP w,i	SEG DS	AAS
4	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI	DEC AX	DEC CX	DEC DX	DEC BX	DEC SP	DEC BP	DEC SI	DEC DI
5	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI	POP AX	POP CX	POP DX	POP BX	POP SP	POP BP	POP SI	POP DI
6																
7	JO	JNO	JB/ JNAE	JNB/ JAE	JE/ JZ	JNE/ JNZ	JBE/ JNA	JNBE/ JA	JS	JNS	JP/ JPE	JNP/ JPO	JL/ JNGE	JNL/ JGE	JLE/ JNG	JNLE/ JG
8	Immed b,r/m	Immed w,r/m	Immed b,r/m	Immed is,r/m	TEST b,r/m	TEST w,r/m	XCHG b,r/m	XCHG w,r/m	MOV b,f,r/m	MOV w,f,r/m	MOV b,t,r/m	MOV w,t,r/m	**MOV sr,f,r/m**	LEA	**MOV sr,t,r/m**	POP r/m
9	XCHG AX	XCHG CX	XCHG DX	XCHG BX	XCHG SP	XCHG BP	XCHG SI	XCHG DI	CBW	CWD	CALL l,d	WAIT	PUSHF	POPF	SAHF	LAHF
A	MOV m → AL	MOV m → AX	MOV AL → m	MOV AX → m	**MOVS**	**MOVS**	**CMPS**	**CMPS**	**TEST b,i,a**	**TEST w,i,a**	**STOS**	**STOS**	**LODS**	**LODS**	**SCAS**	**SCAS**
B	MOV i → AL	MOV i → CL	MOV i → DL	MOV i → BL	MOV i → AH	MOV i → CH	MOV i → DH	MOV i → BH	MOV i → AX	MOV i → CX	MOV i → DX	MOV i → BX	MOV i → SP	MOV i → BP	MOV i → SI	MOV i → DI
C			RET (i+SP)	RET	LES	LDS	MOV b,i,r/m	MOV w,i,r/m			RET l,(i+SP)	RET i	INT Type 3	INT (Any)	INTO	IRET
D	Shift b	Shift w	Shift b,v	Shift w,v	AAM	AAD		XLAT	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
E	LOOPNZ/ LOOPNE	LOOPZ/ LOOPE	LOOP	JCXZ	IN b	IN w	**OUT b**	**OUT w**	CALL d	JMP d	JMP l,d	JMP si,d	**IN v,b**	**IN v,w**	**OUT v,b**	**OUT v,w**
F	LOCK		REP	REP z	HLT	CMC	Grp 1 b,r/m	Grp 1 w,r/m	CLC	STC	CLI	STI	CLD	STD	Grp 2 b,r/m	Grp 2 w,r/m

where

mod□r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL/SAL	SHR	—	SAR
Grp 1	TEST	—	NOT	NEG	MUL	IMUL	DIV	IDIV
Grp 2	INC	DEC	CALL id	CALL l,id	JMP id	JMP l,id	PUSH	—

b = byte operation
d = direct
f = from CPU reg
i = immediate
ia = immed. to accum.
id = indirect
is = immed. byte, sign ext.
l = long ie. intersegment
m = memory
r/m = EA is second byte
si = short intrasegment
sr = segment register
t = to CPU reg
v = variable
w = word operation
z = zero

8086 Instruction Sequence

Figure 4-22 illustrates the internal operation and bus activity that occur as an 8086 CPU executes a sequence of instructions. This figure presents the signals and timing relationships that are important in understanding 8086 operation. The following discussion is intended to help in the interpretation of the figure.

Figure 4-22 shows the repeated execution of an instruction loop. This loop is defined in both machine code and assembly language by figure 4-21. A loop was chosen both to demonstrate the effects of a program jump on the queue and to make the instruction sequence easy to follow. The program sequence shown was selected for several reasons. First, consisting of seven instructions and 16 bytes, the sequence is typical of the tight loops found in many application programs. Second, this particular sequence contains several short, fast-executing instructions that demonstrate both the effect of the queue on CPU performance and the interaction between the execution unit (EU) fetching code from the queue and the bus interface unit (BIU) filling the queue and performing the requested bus cycles. Last, for the purpose of this discussion, code, stack, and memory data references were arranged to be aligned on even word boundaries.

ASSEMBLY LANGUAGE	MACHINE CODE
MOV AX, 0F802H	B802F8
PUSH AX	50
MOV CX, BX	8BCB
MOV DX, CX	8BD1
ADD AX, [SI]	0304
ADD SI, 8086H	81C68680
JMP $ −14	EBF0

Figure 4-21. Instruction Loop Sequence

Figure 4-22 can be more easily interpreted by keeping the following guidelines in mind.

- The queue status lines (QS0, QS1) are the key indicators of EU activity.
- Status lines $\overline{S2}$ through $\overline{S0}$ are the main indicators of 8086/8088 bus activity.
- Interaction of the BIU and EU is via the queue for prefetched opcodes and via the EU for requested bus cycles for data operands.

Keeping these guidelines in mind, the instruction sequence depicted in figure 4-22 can be described as follows. Starting the loop arbitrarily in clock cycle 1 with the queue reinitialization that occurs as part of the JMP instruction, JMP instruction execution is completed by the EU, while the BIU performs an opcode fetch to begin refilling the queue. (Note that a shorthand notation has been used in the figure to represent the two queue status lines and the three status lines—active periods on any of these lines are noted and the binary value of the lines is indicated above each active region.)

In clock cycle 8, the queue status lines indicate that the first byte of the MOV immediate instruction has been removed from the queue (one clock cycle after it was placed there by the BIU fetch) and that execution of this instruction has begun. The second byte of this instruction is taken from the queue in clock cycle 10 and then, in clock cycle 12, the EU pauses to wait one clock cycle for the BIU's second opcode fetch to be completed and for the third byte of the MOV immediate instruction to be available for execution (remember the queue status lines indicate queue activity that has occurred in the previous clock cycle).

Clock cycle 13 begins the execution of the PUSH AX instruction, and in clock cycle 15, the BIU begins the fourth opcode fetch. The BIU finishes the fourth fetch in clock cycle 18 and prepares for another fetch when it receives a request from the EU for a memory write (the stack push). Instead of completing the opcode fetch and forcing the EU to wait four additional clock cycles, the BIU immediately aborts the fetch cycle (resulting in two idle clock cycles (T_I) in clock cycles 19 and 20) and performs the required memory write. This interaction between the EU and BIU results in a single clock extension to the execution time of the PUSH AX instruction, the maximum delay that can occur in response to an EU bus cycle request.

Execution continues in clock cycle 24 with the execution of back-to-back, register-to-register MOV instructions. The first of these instructions takes full advantage of the prefetched opcode to complete this operation in two clock cycles. The second MOV instruction, however, depletes the queue and requires two additional clock cycles (clock cycles 28 and 29).

Mnemonics © Intel, 1978

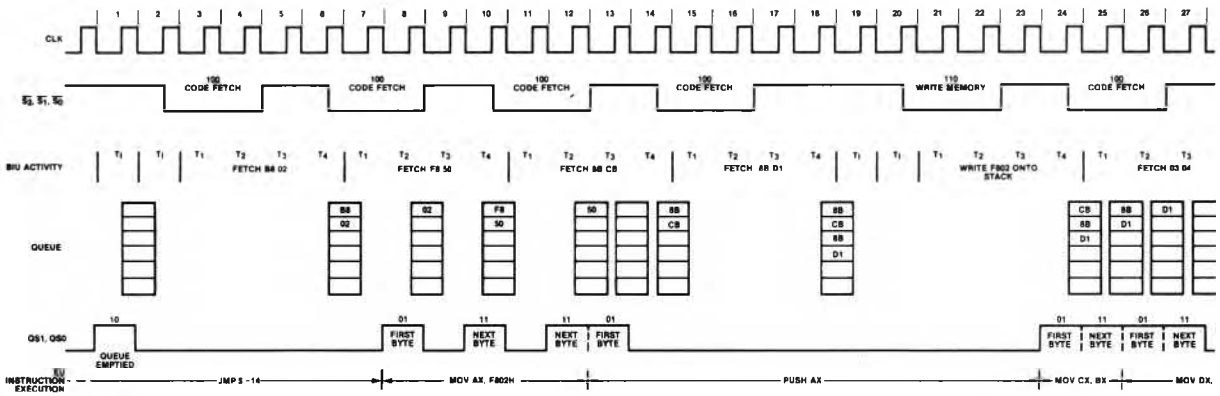


Figure 4-22. Sample Instruction Sequence Execution

In clock cycle 30, the ADD memory indirect to AX instruction begins. In the time required to execute this instruction, the BIU completes two opcode fetch cycles and a memory read and begins a fourth opcode fetch cycle. Note that in the case of the memory read, the EU's request for a bus cycle occurs at a point in the BIU fetch cycle where it can be incorporated directly (idle states are not required and no EU delay is imposed).

In clock cycle 44, the EU begins the ADD immediate instruction, taking four bytes from the queue and completing instruction execution in four clock cycles. Also during this time, the BIU senses a full queue in clock cycle 45 and enters a series of bus idle states (five or six bytes constitute a full queue in the 8086; the BIU waits until it can fetch a full word of opcode before accessing the bus).

At clock cycle 47, the BIU again begins a bus cycle sequence, one that is destined to be an "overfetch" since the EU is executing a JMP instruction. As part of the JMP instruction, the queue reinitialization (which began the instruction sequence) occurs.

The entire sequence of instructions has taken 55 clock cycles. Eighteen opcode bytes were fetched, one word memory read occurred, and one word stack write was performed.

This example was, by design, partially bus limited and indicates the types of EU and BIU interaction that can occur in this situation. Most application code sequences, however, use a higher proportion of more complex, longer-executing instructions and addressing modes, and therefore tend to be execution limited. In this case, less BIU-EU interaction is required, the queue more often is full, and more idle states occur on the bus.

The previous example sequence can be easily extended to incorporate wait states in the bus access cycles. In the case of a single wait state, each bus cycle would be lengthened to five clock cycles with a wait state (T_W) inserted between every T_3 and T_4 state of the bus cycle. As a first approximation, the instruction sequence exection time would appear to be lengthened by 10 clock cycles, one cycle for each useful read or write bus cycle that occurs. Actually, this approximation for the number of wait states inserted is incorrect since the queue can compensate for wait states by making use of previously idle bus time. For the example sequence, this compensation reduced the actual execution time by one wait state, and the sequence was completed in 64 clock cycles, one less than the approximated 65 clock cycles.

4.3 8089 I/O Processor

The Intel® 8089 I/O Processor (IOP) combines the functions of a DMA controller with the processing capabilities of a microprocessor. In addition to the normal DMA function of transferring data, the 8089 is capable of dynamically translating and comparing the data as it is

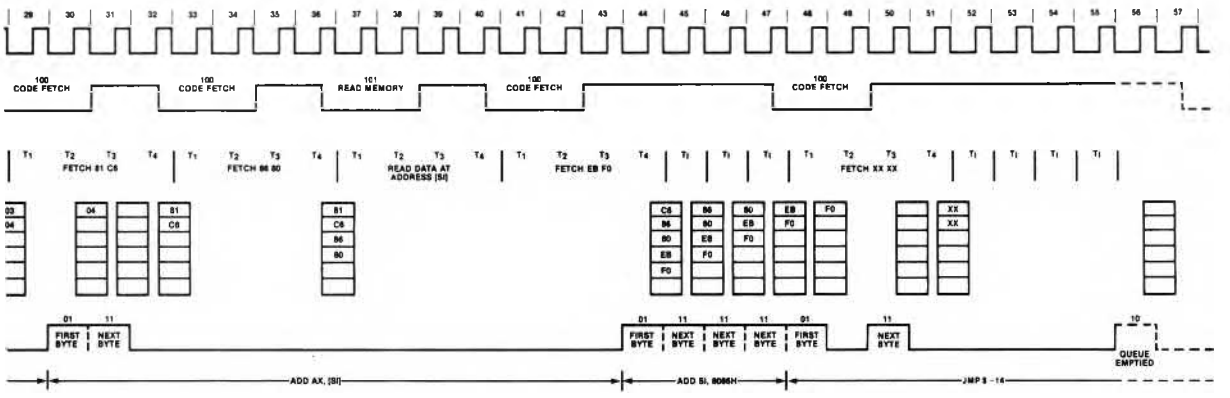


Figure 4-22. Sample Instruction Sequence Execution

transferred and of supporting a number of terminate conditions including byte count expired, data compare or miscompare and the occurrence of an external event. The 8089 contains two separate DMA channels, each with its own register set. Depending on the established priorities (both inherent and program determined), the two channels can alternate (interleave) their respective operations.

Designed expressly to relieve the 8086 or 8088 CPU of the overhead associated with I/O operations, the 8089, when configured in the remote mode, can perform a complete I/O task while the CPU is performing data processing tasks. The 8089, when it has completed its I/O task, can then interrupt the CPU.

Transfer flexibility is an integral part of the 8089's design. In addition to routine transfers between an I/O peripheral and memory, transfers can be performed between two I/O devices or between two areas of memory. Transfers between dissimilar bus widths are automatically handled by the 8089. When data is transferred from an 8-bit peripheral bus to a 16-bit memory bus, the 8089 reads two bytes from the peripheral, assembles the bytes into a 16-bit word and then writes the single word to the addressed memory location. Also, both 8- and 16-bit peripherals can reside on the same (16-bit) bus; byte transfers are performed with the 8-bit peripheral, and word transfers are performed with the 16-bit peripheral.

System Configuration

The 8089 can be implemented in one of two system configurations: a "local" mode in which the 8089 shares the system bus with an 8086 or 8088 CPU and a "remote" mode in which the 8089 has exclusive access to its own dedicated bus as well as access to the system bus. Note that in either the local or remote mode, the 8089 can address a full megabyte of system memory and 64k bytes of I/O space.

Local Mode

In the local mode, the 8089 acts as a slave to an 8086 or 8088 CPU that is operating in the maximum mode. In this configuration, the 8089 shares the system address latches, data transceivers and bus controller with the CPU as shown in figure 4-23.

Since the IOP and CPU share the system bus, either the IOP or the CPU will have access to the bus at any one time. When one processor is using the bus, the other processor floats its address/data and control lines. Bus access between the IOP and CPU is determined through the request/grant function. Recalling the CPU's request/grant sequence, the IOP requests the bus from the CPU, the CPU grants the bus to the IOP, and the IOP relinquishes the bus to the CPU when its operation is complete. Remember that the CPU cannot request the bus from the IOP (the CPU is only capable of granting the bus and

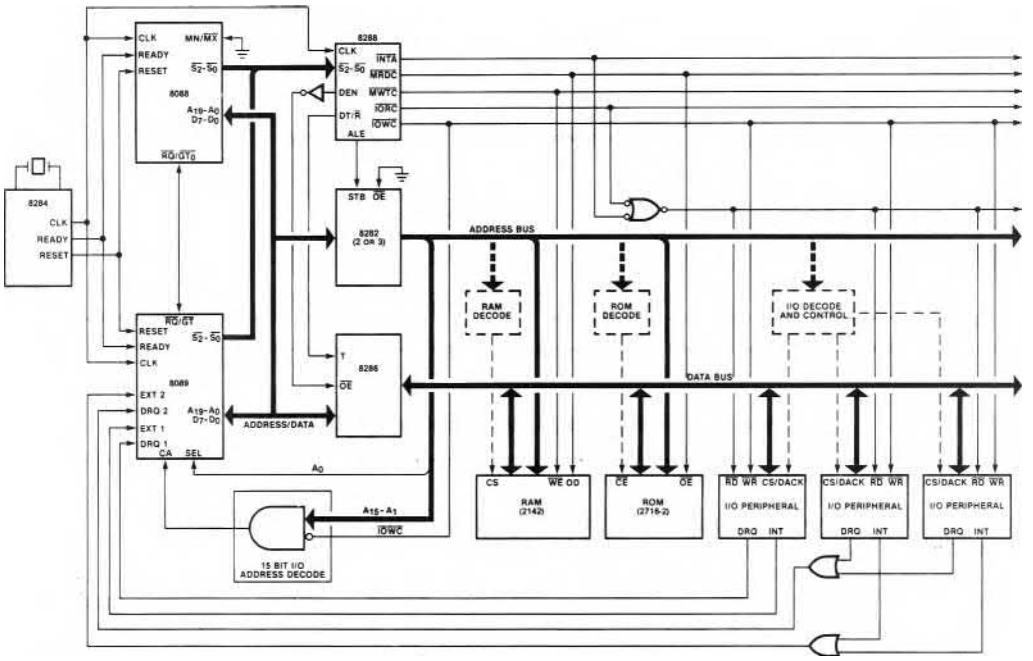


Figure 4-23. Typical 8088/8089 Local Mode Configuration

must wait for the IOP to release the bus). Also, since the request/grant pulse exchange must be synchronized, both the CPU and IOP must be referenced to the same clock signal.

The 8089 IOP, when used in the local mode, can be added to an 8086 or 8088 maximum mode configuration with little affect on component count (channel attention decoding logic as required) and offers the benefits of intelligent DMA (scan/match, translate, variable termination conditions), modular programming in a full megabyte of memory address space and a set of optimized I/O instructions that are unavailable to the 8086 and 8088 CPUs. The major disadvantage to the local configuration is that since the system bus is shared, bus contention always exists between the CPU and IOP. The use of the bus load limit field in the channel control word can help reduce IOP bus access during task block program execution (bus load limiting has no affect on DMA transfers) although, for I/O intensive systems, the remote mode should be considered.

Remote Mode

The 8089, when used in the remote mode, provides a multiprocessor system with true parallel processing. In this mode, the 8089 has a separate (local) bus and memory for I/O peripheral communications, and the system bus is completely isolated from the I/O peripheral(s). Accordingly, I/O transfers between an I/O peripheral and the IOP's local memory can occur simultaneously with CPU operations on the system bus.

As shown in figure 4-24, to interface the 8089 to the system bus, data transceivers and address latches are used to separate the IOP's local bus from the system bus, an 8288 Bus Controller is used to generate the bus control signals for both the local and system buses as well as to govern the operation of the transceivers/latches, and an 8289 Bus Arbiter is used to control access to the system bus (each processor in the system would have an associated 8289 Bus Arbiter). To interface the 8089 to its local bus, another set of address

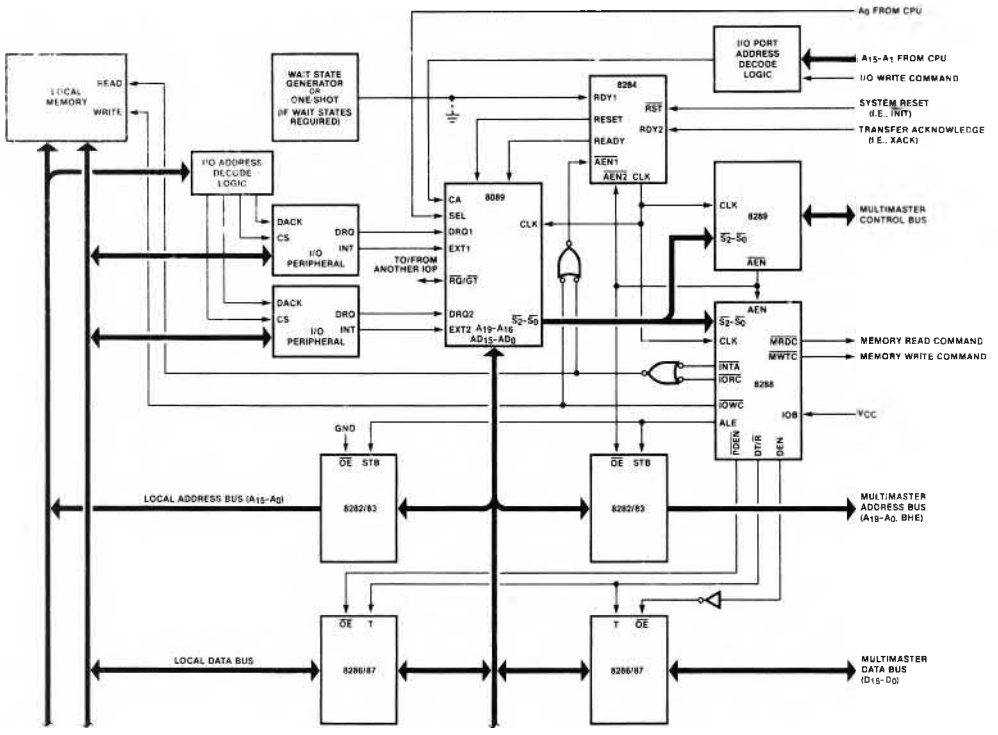


Figure 4-24. Typical 8089 Remote Mode Configuration

latches is required (unless MCS-85™ multiplexed address components are exclusively interfaced) and, depending on the bus loading demands, one (8-bit bus) or two (16-bit bus) data transceivers would be used.

In the remote mode, the IOP's local bus is treated as I/O space (up to 64k bytes), and the system bus is treated as memory space (1 megabyte). The 8288 Bus Controller's I/O command outputs control the local (I/O) bus, and its memory command outputs control the system (memory) bus. The 8289 Bus Arbiter, which is operated in its IOB (I/O peripheral bus) mode, also decodes the IOP's $\overline{S2}$ through $\overline{S0}$ status outputs. In this mode, the 8289 will not request the multimaster system bus when the IOP indicates an operation on its local bus. If the IOP's bus arbiter currently has access to the system bus, the CPU's arbiter (or any other arbiter in the system) can acquire use of the system bus at this time (a bus arbiter maintains bus access until another arbiter requests the bus).

Bus Operation

The 8089 utilizes the same bus structure as an 8086 or 8088 CPU that is configured in the maximum mode and performs a bus cycle only on demand (e.g., to fetch an instruction during task block execution or to perform a data transfer). The bus cycle itself is identical to an 8086 or 8088 CPU's bus cycle in that all cycles consist of four T-states and use the same time-multiplexing technique of the addressdata lines. As shown in the following timing diagrams, the address (and ALE signal) is output during state T_1 for either a read or write cycle. Depending on the type of cycle indicated, the address/data lines are floated during state T_2 for a read cycle (figure 4-25) or data is output on these lines during a write cycle (figure 4-26). During state T_3, write data is maintained or read data is sampled, and the busy cycle is concluded in state T_4.

Since the 8089 is capable of transferring data to or from both 8-bit and 16-bit buses, when an 8-bit physical bus is specified (bus width is specified

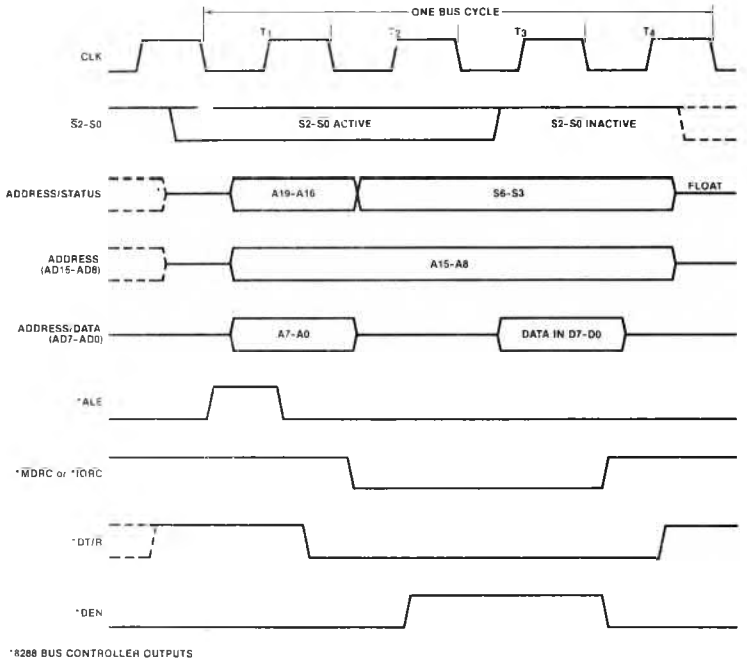


Figure 4-25. Read Bus Cycle (8-Bit Bus)

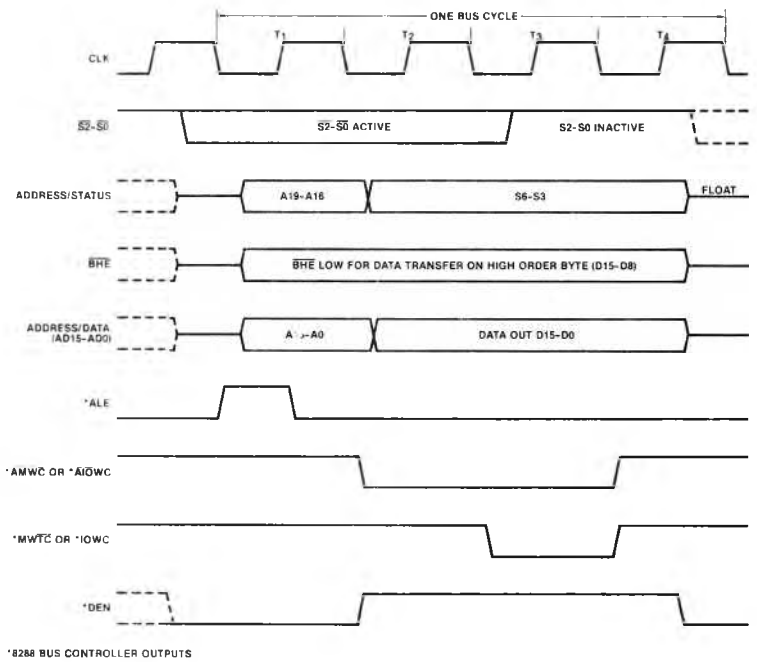


Figure 4-26. Write Bus Cycle (16-Bit Bus)

during the initialization sequence), the address present on the AD15 through AD8 address/data lines is maintained for the entire bus cycle as shown in figure 4-25 and, unless added drive capability is required, the associated address latch can be eliminated. An 8-bit data bus is compatible with the 8088 CPU and with the MCS-85™ multiplexed address peripherals (8155, 8185, etc.).

The 8089 operates identically to the 8086 CPU with respect to the use of the low- and high-order halves of the data bus. Table 4-14 defines the data bus use for the various combinations of bus width and address boundary.

The $\overline{S2}$ through $\overline{S0}$ status lines define the bus cycle to be performed. These lines are used by an 8288 Bus Controller to generate all memory and I/O command and control signals, and are decoded according to table 4-15.

Table 4-14. Data Bus Usage

Logical Bus Width[1]	Address Boundary	Physical Bus Width[2]		
		8	16	
			Byte Transfer	Word Transfer
8	Even	AD7-AD0 = DATA ($\overline{BHE}$ not used)	AD7-AD0 = DATA ($\overline{BHE}$ high)	N/A
	Odd	AD7-AD0 = DATA ($\overline{BHE}$ not used)	AD15-AD8 = DATA ($\overline{BHE}$ low)	N/A
16	Even	Illegal	AD7-AD0 = DATA ($\overline{BHE}$ high)	AD15-AD0 = DATA ($\overline{BHE}$ low)
	Odd	Illegal	AD15-AD8 = DATA ($\overline{BHE}$ low)	N/A[3]

Notes:

1. Logical bus width is specified by the WID instruction prior to the DMA transfer.
2. Physical bus width is specified when the 8089 is initialized.
3. A word transfer to or from an odd boundary is performed as two byte transfers. The first byte transferred is the low-order byte on the high-order data bus (AD15-AD8), and the second byte is the high-order byte on the low-order data bus (AD7-AD0). The 8089 automatically assembles the two bytes in their proper order.

Table 4-15. Bus Cycle Decoding

Status Output			Bus Cycle Indicated	Bus Controller Command Output
$\overline{S2}$	$\overline{S1}$	$\overline{S0}$		
0	0	0	Instruction fetch from I/O space	$\overline{INTA}$
0	0	1	Data read from I/O space	$\overline{IORC}$
0	1	0	Data write to I/O space	$\overline{IOWC}$, $\overline{AIOWC}$
0	1	1	Not used	None
1	0	0	Instruction fetch from system memory	$\overline{MRDC}$
1	0	1	Data read from system memory	$\overline{MRDC}$
1	1	0	Data write to system memory	$\overline{MWTC}$, $\overline{AMWC}$
1	1	1	Passive	None

Note that the 8089 indicates an instruction fetch from I/O space as a status of zero ($\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ equal 0). Since the 8288 Bus Controller decodes an input status value of zero as an interrupt acknowledge bus cycle, the bus controller's $\overline{INTA}$ output must be OR'ed with its $\overline{IORC}$ output to permit fetching of task block instructions from local 8089 memory (remote configuration) or system I/O space (local and remote configurations).

The $\overline{S2}$ through $\overline{S0}$ status lines become active in state T_4 if a subsequent bus cycle is to be performed. These lines are set to the passive state (all "ones") in the state immediately prior to state T_4 of the current bus cycle (state T_3 or T_W) and are floated when the 8089 does not have access to the bus.

The S6 through S3 status lines are multiplexed with the high-order address bits (A19-A16) and, accordingly, become valid in state T_2 of the bus cycle. The S4 and S3 status lines reflect the type of bus cycle being performed on the corresponding channel as indicated in table 4-16.

Table 4-16. Type of Cycle Decoding

Status Output		Type of Cycle
S4	S3	
0	0	DMA on Channel 1
0	1	DMA on Channel 2
1	0	Non-DMA on Channel 1
1	1	Non-DMA on Channel 2

The S6 and S5 status lines are always "1" on the 8089. Since these lines are not both "1" on the other processors in the 8086 family (S6 is always "0" on the 8086 and 8088 CPUs), these status lines can be used as a "signature" in a multiprocessor environment to identify the type of processor performing the bus cycle.

The 8089 includes the same provision as do the 8086 and 8088 CPUs for the insertion of wait states (T_W) in a bus cycle when the associated memory or I/O device cannot respond within the alloted time interval or when, in the remote mode, the 8089 must wait for access to the system bus. An 8284 Clock Generator/Driver is used to control the insertion of wait states which, when required, are inserted between states T_3 and T_4. The actual insertion of wait states is accomplished by deactivating one of the 8284's RDY inputs (RDY1 or RDY2). Either of these inputs, when enabled by its corresponding $\overline{AEN1}$ or $\overline{AEN2}$ input, can be deactivated directly by the memory or I/O device when it must extend the 8089's bus cycle (when the addressed device is not ready to present or accept data). The 8284's READY output, which is synchronized to the CLK signal, is directly connected to the 8089's READY input. As shown in figure 4-27, when the addressed device requires one or more wait states to be inserted into a bus cycle, it deactivates the 8284's RDY input prior to the end of state T_2. The READY output from the 8284 is subsequently deactivated at the end of state T_2 which causes the 8089 to insert wait states following state T_3. To exit the wait state, the device activates the 8284's RDY input which causes the READY input to the 8089 to go active on the next clock cycle and allows the 8089 to enter state T_4.

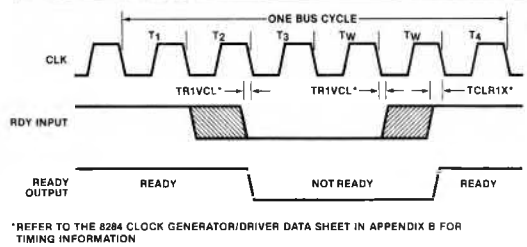


Figure 4-27. Wait State Timing

Periods of inactivity can occur between bus cycles. These inactive periods are referred to as idle states (T_I) and, as with the 8086 and 8088 CPUs, can result from the execution of a "long" instruction or the loss of the bus to another processor during task block instruction execution. Additionally, the 8089 can experience idle states when it is in the DMA mode and it is waiting for a DMA request from the addressed I/O device or when the bus load limit (BLL) function is enabled for a channel performing task block instruction execution and the other channel is idle.

Initialization

Initialization of the IOP is generally the responsibility of the host processor which, as stated in Chapter 3, prepares the communications data structure in shared memory. Initialization of the IOP itself begins with the activation of its RESET input. This input (originating typically from an

8284 Clock Generator/Driver) must be held active for at least five clock cycles to allow the 8089's internal reset sequence to be completed. Note that like the 8086 and 8088 CPUs, the RESET input must be held active for at least 50 microseconds when power is first applied. Following the reset interval, the host processor signals the IOP to begin its initialization sequence by activating the 8089's CA (Channel Attention) input. The 8089 will not recognize a pulse at its CA input until one clock cycle after the RESET input returns to an inactive level. Note that the minimum width for a CA pulse is one clock cycle and that this pulse may go active prior to RESET returning to an inactive level provided that the negative-going, trailing-edge of the CA pulse does not occur prior to one clock cycle after RESET goes inactive. Figure 4-28 illustrates the timing for this portion of the initialization sequence.

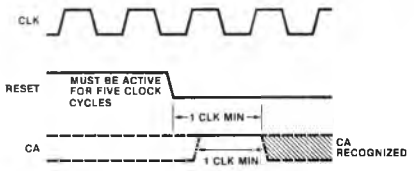


Figure 4-28. RESET-CA Initialization Timing

Coincident with the trailing edge of the first CA pulse following reset, the 8089 samples its SEL (Select) input from the host processor to determine master/slave status for its request/grant circuity. If the SEL input is low, the 8089 is designated a "master," and if the SEL input is high, the 8089 is designated a "slave." As a master, the 8089 assumes that it has the bus initially, and it will subsequently grant the bus to a requesting slave when the bus becomes available (i.e., the 8089 will respond to a "request" pulse on its $\overline{RQ}/\overline{GT}$ line with a "grant" pulse). A single 8089 in the remote configuration (or one of two 8089s in a remote configuration) would be designated a master. As a slave, the 8089 can only request the bus from a master processor (i.e., the 8089 initiates the request/grant sequence by outputting a "request" pulse on its $\overline{RQ}/\overline{GT}$ line). An 8089 that shares a bus with an 8086 or 8088 (or one of two 8089s in a remote configuration) would be designated a slave. Note that since the 8086 and 8088 CPUs can grant the bus only in response to a request, whenever an 8086 or 8088 and an 8089 share a common bus, the 8089 *must* be designated the slave. Also, when the $\overline{RQ}/\overline{GT}$ line is not used (i.e., a single 8089 in the remote configuration), the 8089 *must* be designated a master.

In addition to determining master/slave status, the CA pulse also causes the 8089 to begin execution of its internal ROM initialization sequence. Note that since the 8089 must have access to the *system* bus in order to perform this sequence, the 8089 immediately initiates a request/grant sequence (if designated a slave) and, if required, then requests the bus through the 8289 Arbiter. (If designated a master, the 8089 requests the bus through the 8289 Arbiter.) In the execution of the initialization sequence, the 8089 first fetches the SYSBUS byte from location FFFF6H. The W bit (bit 0) of this byte specifies the *physical* bus width of the *system* bus. Depending on the bus width specified, the 8089 then fetches the address of the system configuration block (SCB) contained in locations FFFF8H through FFFFBH in either two bus cycles (16-bit bus, W bit equal 1) or four bus cycles (8-bit bus, W bit equal 0). The SCB offset and segment address values fetched are combined into a 20-bit physical address that is stored in an internal register. Using this address, the 8089 next fetches the system operation command (SOC) byte. As explained in Chapter 3, this byte specifies both the request/grant operational mode (R bit) and the *physical* width of the I/O bus (I bit). After reading the SOC byte, the 8089 fetches the channel control block (CB) offset and segment address values. These values are combined into a 20-bit physical address and are stored in another internal register. To inform the host CPU that it has completed the initialization sequence, the 8089 clears the Channel 1 Busy flag in the channel control block by writing an all "zeroes" byte to CB + 1.

After the IOP has been initialized, the system configuration block may be altered in order to initialize another IOP. Once an IOP has been initialized, its channel control block in system memory cannot be moved since the CB address, which is internally stored by the IOP during the initialization sequence, is automatically accessed on every subsequent CA pulse.

and SEL inputs to the IOP's [illegible]

of the host CPU. Typically, these signals result from the CPU's execution of an I/O write instruction to one of two adjacent I/O ports (I/O port addresses that only differ by A0). Figure 4-29 illustrates a simple decoding circuit that could be used to generate the CA and SEL signals. Note that by qualifying the CA output with $\overline{IOWC}$, the SEL output, since it is latched for the entire I/O bus cycle, is guaranteed to be stable on the trailing edge of the CA pulse.

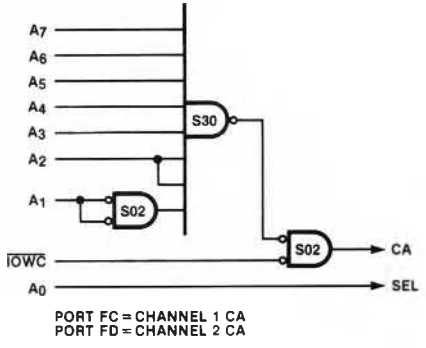


Figure 4-29. Channel Attention Decoding Circuit

I/O Dispatching

During normal operation, the I/O supervisory program running in the host CPU will receive a request to perform a specific I/O operation on one of the 8089's channels. In response to this request, the supervisory program will typically perform the following sequence of operations:

- Check the availability of the specified channel by examining the channel's busy flag in the Channel Control Block. If it is possible for another processor to access the channel, a semaphore operation (implemented by a locked XCHG instruction) is used to check channel availability.
- Load the variable parameters required for the intended operation into the channel's parameter block.
- Load the channel command word (CCW) into the channel control block.
- Establish the necessary linkages by writing the starting address of the channel program (task block) in the first four bytes of the block.
- Issue a channel attention (CA) to the specified channel.

In response to the CA, the 8089 interrupts any current activity at its first opportunity (see "Concurrent Channel Operation" in section 3.2) and begins execution of an internal instruction sequence that fetches and decodes the channel command word (CCW) and then performs the operation indicated (i.e., start, halt or continue channel program execution).

If the CCW specifies start channel program (start task block execution), the address of the parameter block is fetched from the channel control block, the address of the first channel program instruction (contained in the first four bytes of the parameter block) is fetched and then loaded into the TP (task pointer) register and, finally, task block execution is initiated from either system or I/O space. Task block execution continues, subject to the activity on the other channel as described in "Concurrent Channel Operation," until a XFER instruction is executed. Following execution of this instruction, the next sequential channel program instruction is executed before the channel enters the DMA transfer mode.

If the CCW specifies halt channel, the current operation on the specified channel is halted. If the channel is performing task block execution (either chained or not chained), channel operation is stopped at an instruction boundary, and if the channel is performing a DMA transfer, channel operation is stopped at a DMA transfer cycle boundary. Note that a channel will not stop a locked DMA transfer until the operation is completed. There are two unique halt channel commands. One command simply halts the channel and clears the busy flag in the channel control block. This command is used when the halted operation is to be discarded. The other command halts the channel, saves the task pointer and program status word (PSW) byte, and clears the busy flag. This command is used when the halted operation is to be resumed. Note that this halt command will not affect the integrity of resumed task block execution or a memory-to-memory DMA transfer, but could affect the integrity of a synchronized DMA transfer (a DMA request occuring while the channel is halted could be missed).

If the CCW specifies continue channel, an operation that has been previously halted is resumed (and the busy flag is set). Since this command restores the task pointer and PSW, it should be used only if the task pointer and PSW have been saved by a previous halt command.

Table 4-17 outlines the various CCW command execution times. Note that the times listed in the table for the halt commands do *not* include the time required to complete any current channel activity when the channel attention is received (completion of the current DMA transfer cycle or task block instruction).

DMA Transfers

The number of bytes transferred during a single DMA cycle is determined by both the source and destination logical bus widths as well as by the address boundary (odd or even address). The 8089 performs DMA transfers between dissimilar bus widths by assembling bytes or disassembling words in its internal assembly register file. As explained in Chapter 3, the DMA source and destination bus widths are defined by the execution of a WID instruction during task block (channel command) execution. Note that the bus widths specified remain in force until changed by a subsequent WID instruction. Table 4-18 defines the various byte (B) and word (W) source/destination transfer combinations based on address boundary and bus width specified.

The 8089 additionally optimizes bus accesses during transfers between dissimilar bus widths whenever possible. When either the source or destination is a 16-bit memory bus (auto-incrementing) that is initially aligned on an odd

Table 4-17. CCW Command Execution Times

CCW Command	Minimum Time*	Maximum Time**
CA NOP	48 + 2n clocks	48 + 2n clocks
CA Halt (no save)	48 + 2n clocks	48 + 2n clocks
CA Halt (with save)	94 + 5n clocks	100 + 6n clocks
CA Start (memory)	108 + 6n clocks	124 + 10n clocks
CA Start (I/O)	96 + 5n clocks	108 + 8n clocks
CA Continue	95 + 5n clocks	103 + 6n clocks

Notes:
n is the number of wait states per bus cycle.

* Minimum time occurs when both the channel control block and parameter block addresses are aligned on an even address boundary and a 16-bit bus is used.

** Maximum time occurs when both the channel control block and parameter block addresses are aligned on an odd address boundary on a 16-bit bus or when an 8-bit bus is used.

Table 4-18. DMA Assembly Register Operation

Address Boundary (Source → Destination)	Logical Bus Width (Source → Destination)			
	8 → 8	8 → 16	16 → 8	16 → 16
Even → Even	B → B	B/B → W	W → B/B	W → W
Even → Odd	B → B	B → B	W → B/B	W → B/B
Odd → Even	B → B	B/B → W	B → B	B/B → W
Odd → Odd	B → B	B → B	B → B	B → B

Mnemonics © Intel, 1979

address boundary (causing the first transfer cycle to be byte-to-byte), following the first transfer cycle, the memory address will be aligned on an even address boundary, and word transfers will subsequently occur. For example, when performing a memory-to-port transfer from a 16-bit bus to an 8-bit bus with the source beginning on an odd address boundary, the first transfer cycle will be byte-to-byte (B → B) as indicated in table 4-18, but subsequent transfers will be word-to-byte/byte (W → B/B).

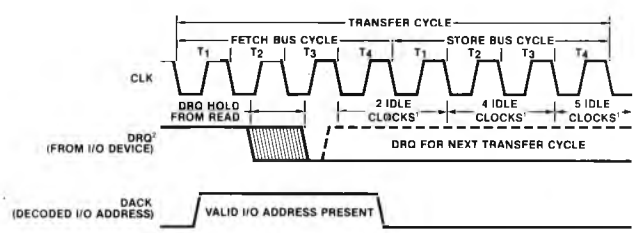
All DMA transfer cycles consist of at least two bus cycles; one bus cycle to fetch (read) the data form the source into the IOP, and one bus cycle to store (write) the data previously fetched from the IOP into the destination. Note that in all transfers, the data passes through the IOP to allow mask/compare and translate operations to be optionally performed during the transfer as well as to allow the data to be assembled or disassembled.

The IOP performs DMA transfers in one of three modes: unsynchronized, source synchronized or destination synchronized (the transfer mode is specified in the channel control register). The unsynchronized mode is used when both the source and destination devices do not provide a data request (DRQ) signal to the IOP as in the case of a memory-to-memory transfer. In the synchronized transfer modes, the source (source synchronized) or destination (destination synchronized) device initiates the transfer cycle by activating the IOP's DRQ1 (channel 1) or DRQ2 (channel 2) input.

The DRQ input is asynchronous and usually originates from an I/O device controller rather than from a memory circuit. This input is latched on the positive transition of the clock (CLK) signal and therefore must remain active for more than one clock period (more than 200 nanoseconds when using a 5 MHz clock) in order to guarantee that it is recognized.

During state T_1 of the associated fetch bus cycle (source synchronized) or store bus cycle (destination synchronized), the IOP outputs the address of the I/O device (the port address). This address must be decoded (by external circuitry) to generate the DMA acknowledge (DACK) signal to the I/O controller as the response to the controller's DMA request. An I/O controller will typically use DACK as a conditional input for the removal of DRQ. (After receipt of the DACK signal, most Intel peripheral controllers deactivate DRQ following receipt of the corresponding read or write signal.) Figures 4-30 and 4-31 illustrate the DRQ/DACK timing for both source synchronized (i.e., port-to-memory) and destination synchronized (i.e., memory-to-port) transfers.

Table 4-19 defines the DMA transfer cycles in terms of the number of bus and clock cycles required. Note that the number of clocks required to complete a transfer cycle does not take into account the effects of possible concurrent operations on the other channel or wait states within any of the bus cycles.



NOTES:

1. INDICATES THE NUMBER OF IDLE CLOCK CYCLES INSERTED BEFORE THE NEXT TRANSFER CYCLE BEGINS. IF DRQ IS RECEIVED PRIOR TO STATE T_4 OF THE CURRENT FETCH CYCLE, THE NEXT FETCH CYCLE BEGINS IMMEDIATELY FOLLOWING THE CURRENT STORE CYCLE.
2. IF THE 8089 IS IDLE WHEN DRQ IS RECOGNIZED, FIVE IDLE CLOCK CYCLES OCCUR BEFORE THE ASSOCIATED TRANSFER CYCLE IS INITIATED.

Figure 4-30. Source Synchronized Transfer Cycle

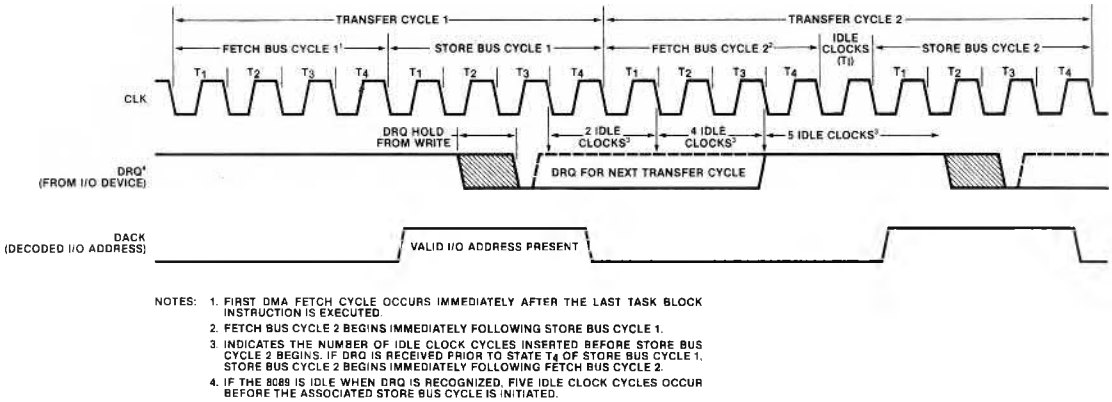


Figure 4-31. Destination Synchronized Transfer Cycle

Table 4-19. DMA Transfer Cycles

Logical Bus Width		Transfer Mode					
		Unsynchronized		Source Synchronized		Destination Synchronized	
Source	Destination	Bus Cycles Required	Total[1] Clocks	Bus Cycles Required	Total[1] Clocks	Bus Cycles Required	Total[1] Clocks
8	8	2 (1 fetch, 1 store)	8[2]	2 (1 fetch, 1 store)	8[2]	2 (1 fetch, 1 store)	8[2]
8	16[3]	3 (2 fetch, 1 store)	12	3 (2 fetch, 1 store)	16[4]	3 (2 fetch, 1 store)	12
16[3]	8	3 (1 fetch, 2 store)	12	3 (1 fetch, 2 store)	12	3 (1 fetch, 2 store)	16[4]
16[3]	16[3]	2 (1 fetch, 1 store)	8	2 (1 fetch, 1 store)	8	2 (1 fetch, 1 store)	8

Notes:

1. The "Total Clocks Required" does not include wait states. One clock cycle per wait state must be added to each fetch and/or store bus cycle in which a wait state is inserted. When performing a memory-to-memory transfer, three additional clocks must be added to the total clocks required (the first fetch cycle of any memory-to-memory transfer requires seven clock cycles).
2. When performing a translate operation, one additional 7-clock bus cycle must be added to the values specified in the table.
3. Word transfers in the table assume an even address word boundary. Word transfers to or from odd address boundaries are performed as indicated in table 4-18 and are subject to the bus cycle/clock requirements for byte-to-byte transfers.
4. Transfer cycles that include two synchronized bus cycles (i.e., synchronous transfers between dissimilar logical bus widths) insert four idle clock cycles between the two synchronized bus cycles to allow additional time for the synchronzing device to remove its initial DMA request.

the 8089 to acknowledge, by outputting the device's corresponding port address, a DMA request at its DRQ input. This response latency is dependent on a number of factors including the transfer cycle being performed, activity on the other channel, memory address boundaries, wait states present in either bus cycle and bus arbitration times.

Generally, when the other channel is idle, the maximum DACK latency is five clock cycles (1 microsecond at 5 MHz), excluding wait states and bus arbitration times. An exception occurs when performing a word transfer to or from an odd memory address boundary. This operation, since two store (source synchronized) or two fetch (destination synchronized) bus cycles are required to access memory, has a maximum possible latency of nine clock cycles. When the other channel is performing DMA transfers of equal priority ("P" bits equal), interleaving occurs at bus cycle boundaries, and the maximum latency is either nine clock cycles when the other channel is performing a normal 4-clock fetch or store bus cycle or twelve clock cycles when the other channel is performing the first fetch cycle of a memory-to-memory transfer. If the other channel is performing "chained" task block instruction execution of equal priority, maximum latency can be as high as 12 clock cycles (channel command instruction execution is interrupted at machine cycle boundaries which range from two to eight clock cycles).

DMA Termination

As stated in Chapter 3, a channel can exit the DMA transfer mode (and return to task block execution) on any of the following terminate conditions:

- Single cycle transfer
- Byte count expired
- Mask/compare match or mismatch
- External event

The terminate conditions are specified by individual fields in the channel control register. More than one terminate condition can be specified for a transfer (e.g., a transfer can be terminated when a specific byte count is reached or on the occurrence of an external event). When displacements (which are added to the task pointer register value) are specified to cause task block execution to resume at a unique entry point for each condition. Three reentry points are available: TP, TP + 4 and TP + 8. The time interval between the occurrence of a terminate condition and the resumption of task block execution is 12 clock cycles for reentry point TP and 15 clock cycles for reentry points TP + 4 and TP + 8.

Peripheral Interfacing

When interfacing a peripheral to an 8-bit physical data bus, the 8089 uses only the lower half of the address/data lines (AD7-AD0) as the bidirectional data bus, and the upper half of the address/data lines (AD15-AD8) maintain address information for the entire bus cycle. Consequently, with this bus configuration, only one octal latch (e.g., an Intel® 8282/83 Octal Latch) is required since only the lower half of the address/data lines is time-multiplexed (unless the address bus requires the increased current drive capability and capacitive load immunity provided by the latch).

When interfacing a peripheral to a 16-bit data bus, both the lower and upper halves of the address/data lines are time-multipelxed, and two octal latches are required. Note that unlike the 8086 and 8088 CPUs, the 8089 does not time-multiplex $\overline{BHE}$ (this signal is valid for the entire bus cycle). Both 8- and 16-bit peripherals can be interfaced to a 16-bit bus. An 8-bit peripheral can be connected to either the upper or lower half of the bus. An 8-bit peripheral on the lower half of the bus must use an even source/destination address, and an 8-bit peripheral on the upper half of the bus must use an odd source/destination address. To take advantage of word transfers, a 16-bit peripheral must use an even source/destination address.

To prepare a peripheral device for a DMA transfer, command and parameter data is written to the device's command/status port. This is usually accomplished using pointer register GC. Recalling that the 8089 executes one additional task block instruction following execution of the XFER instruction (the XFER instruction causes the 8089 to enter the DMA mode), this additional instruction is used to access the command port of an I/O device that immediately begins DMA

operation on receipt of the last command (the 8271 Floppy Disk Controller begins its DMA transfer on receipt of the last command parameter). Since a translate DMA operation requires the use of all three pointer registers (GA and GB specify the source and destination addresses; GC specifies the base address of the translation table), when it is necessary to use the last task block instruction to start the device, command port access can be accomplished relative to one of the pointer registers or relative to the PP register. If the device's data port address (GA or GB) is below the device's command port address, either an offset or an indexed reference can be used to access the command port.

A peripheral's (or peripheral controller's) DMA communication protocol with the 8089 is as follows:

- The peripheral (when source or destination synchronized) initiates a DMA transfer cycle by activating the 8089's DRQ (DMA request) input.
- The 8089 acknowledges the request by placing the peripheral's assigned data port address on the bus during state T_1 of the corresponding fetch (source synchronized) or store (destination synchronized) bus cycle. The peripheral is responsible for decoding this address as the DMA acknowledge (DACK) to its request.
- The data is transferred between the peripheral and the 8089 during the T_2 through T_4 state interval of the bus cycle. The peripheral must remove its DMA request during this interval.
- The peripheral, when ready, requests another DMA transfer cycle by again activating the DRQ input, and the above sequence is repeated.
- The peripheral can, as an option, end the DMA transfer by activating the 8089's EXT (external terminate) input.

The 8089 can support mulitple peripheral devices on a single channel provided that only one device is in the active transfer mode at any one time. To interface multiple devices, the DMA request (DRQ) lines are OR'ed together as are the external terminate (EXT) lines. Unique port addresses are, however, assigned to each device so that an individual DMA acknowledge (DACK) is returned to only the active device. DACK decoding can be accomplished with an Intel ® 8205 Binary Decoder or a ROM circuit. Note that the 8089 can only determine which device has requested service or terminated by the context of the task block program.

Most peripheral devices interfaced to the 8089 will use the decoded DMA acknowledge signal (DACK) as the "chip select" input. Peripheral devices that do not follow this convention must use DACK as a conditional input of chip select.

While most interrupts associated with the 8089 will be DMA requests or external terminates, non-DMA related interrupts can additionally be supported.

One technique that would be used when an 8089 is the local configuration (or when an 8086 or 8088 and an 8089 are locally connected as a remote module) is to allow the CPU to accept the interrupt and then direct the 8089 to the interrupt service routine. Another technique is to allow the 8089 to "poll" the device to determine when an interrupt has occurred (most peripheral controllers have an interrupt pending bit in a status word). The 8089's bit testing instructions are ideally suited for polling.

When the 8089 is in a remote configuration, non-DMA related interrupts can be supported with the addition of an Intel® 8259A Programmable Interrupt Controller. Systems that require this type of interrupt structure would dedicate one of the 8089's channels to interrupt servicing. In implementing this structure, the interrupt output from the 8259A is directly connected to the channel's external terminate (EXT) input, and the channel's DMA request (DRQ) input is not used. A task block program is initially executed to perform a source-synchronized DMA transfer (with an external terminate) on the "interrupt" channel to "arm" the interrupt mechanism. Since the DRQ input is not used, when the channel enters the DMA transfer mode, the channel idles while waiting for the first DMA request (which never occurs). The other channel, since the interrupt channel is idle, operates at maximum throughput. When an interrupt occurs, the "pseudo" DMA transfer is immediately terminated, and task block instruction execution is resumed. The task block program would write a "poll" command to the 8259A's command port and then read the

8259A's data port to acknowledge the interrupt and to determine the device responsible for the interrupt (the device is identified by a 3-bit binary number in the associated data byte). The device number read would be used by the task block program as a vector into a jump table for the device's interrupt service routine. Pertinent interrupt data could be written into the associated parameter block for subsequent examination by the host processor.

The interrupt mechanism previously described, since it uses the 8089's external terminate function, provides an extremely fast interrupt response time.

Note that when using dynamic RAM memory with the 8089, an Intel® 8202 Dynamic RAM Controller can be used to simplify the interface and to perform the RAM refresh cycle. When maximum transfer rates are required, the RAM refresh cycle can be externally initiated by the 8089. By connecting the decoded DACK (DMA acknowledge) signal to the 8202's REFRQ (refresh request) input, the refresh cycle will occur coincident with the I/O device bus cycle and therefore will not impose wait states in the memory bus cycle.

Instruction Encoding

Most 8089 programming will be performed at the assembly language level using ASM-89, the 8089 assembler. During program debugging, however, it may be necessary to work directly with machine instructions when monitoring the bus, reading unformatted memory dumps, etc. This section contains both a table to encode any ASM-89 instruction into its corresponding machine instruction (table 4-24) and a table to "disassemble" any machine instruction back into its associated assembly language equivalent (table 4-26).

Figure 4-32 shows the format of a typical 8089 machine instruction. Except for the LPDI and memory-to-memory forms of the MOV and MOVB instructions that are six bytes long, all 8089 machine instructions consist of from two to five bytes. The first two bytes are always present and are generally formatted as shown in figure 4-32 (table 4-24 contains the exact encoding of every instuction).

Bits 5 through 7 of the first byte of an instruction comprise the R/B/P field. This field identifies a register, bit select or pointer register operand as outlined in table 4-20.

Table 4-20. R/B/P Field Encoding

Code	Register	Bit	Pointer
000	GA	0	GA
001	GB	1	GB
010	GC	2	GC
011	BC	3	N/A
100	TP	4	TP
101	IX	5	N/A
110	CC	6	N/A
111	MC	7	N/A

The WB field (bits 3 and 4 of the first byte) indicates how many displacement/data bytes are present in the instruction as outlined in table 4-21. The displacement bytes are used in program transfers; one byte is present for short transfers, while long transfers contain a two-byte (word) displacement. As mentioned in Chapter 3, the

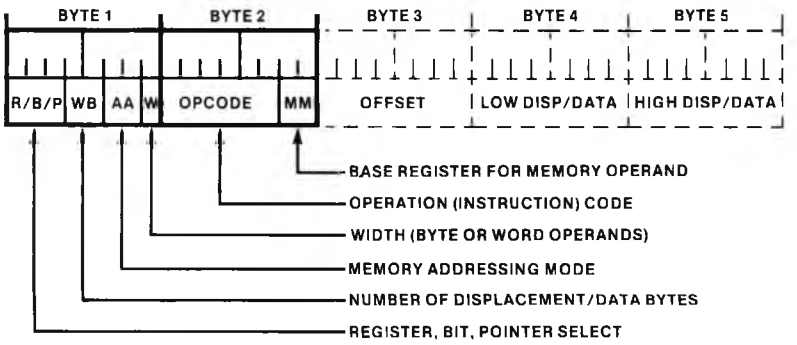


Figure 4-32. Typical 8089 Machine Instruction Format

displacement is stored in two's complement notation with the high-order bit indicating the sign. Data bytes contain the value of an immediate constant operand. A byte immediate instruction (e.g., MOVBI) will have one data byte, and a word immediate instruction (e.g., ADDI) will have two bytes (a word) of immediate data. An instruction may contain either displacement or data bytes, but not both (the TSL instruction is an exception and contains one byte of displacement and one byte of data). If an offset byte is present, the displacement/data byte(s) always follow the offset byte.

Table 4-21. WB Field Encoding

Code	Interpretation
00	No displacement/data bytes
01	One displacement/data byte
10	Two displacement/data bytes
11	TSL instruction only

The AA field specifies the addressing mode that the processor is to use in order to construct the effective address of a memory operand. Four addressing modes are available as outlined in table 4-22. (Address modes are described in detail in section 3.8.)

Table 4-22. AA Field Encoding

Code	Interpretation
00	Base register only
01	Base register plus offset
10	Base register plus IX
11	Base register plus IX, auto-increment

Bit 0 of the first instruction byte indicates whether the instruction operates on a byte (W=0) or a word (W=1).

Bits 7 through 2 of the second instruction byte specify the instruction opcode. The opcode, in conjunction with the W field of the first byte, identifies the instruction. For example, the opcode "111011" denotes the decrement instruction; if W=0, the assembly language instruction is DECB, while if W=1, the instruction is DEC. Table 4-26 lists, in hexadecimal order, the opcode of every assembly language instruction.

The MM field (bits 0 and 1) indicates which pointer (base) register is to be used to construct the effective address of a memory operand. Table 4-23 defines the MM field encoding. (Memory operand addressing is described in section 3.8.)

Table 4-23. MM Field Encoding

Code	Base Register
00	GA
01	GB
10	GC
11	PP

When the AA field value is "01" (base register + offset addressing), the third byte of the instruction contains the offset value. This unsigned value is added to the content of the base register specified by the MM field to form the effective address of the memory operand.

When the AA field value is "10," the IX register value is added to the content of the base register specified by the MM field to provide a 64k range of effective addresses. (Note that the upper four bits of the IX register are not sign-extended.)

When the AA field value is "11," the IX register value is added to the base register value to form the effective address as described for an AA field value of "10." In this addressing mode, however, the IX register value is incremented by one after every byte accessed.

Table 4-24. 8089 Instruction Encoding

DATA TRANSFER INSTRUCTIONS

MOV = Move word variable	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Memory to register	R R R 0 0 A A 1	1 0 0 0 0 0 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 1	1 0 0 0 0 1 M M	offset if AA=01			
Memory to memory	0 0 0 0 0 A A 1	1 0 0 1 0 0 M M	offset if AA=01	0 0 0 0 0 A A 1	1 1 0 0 1 1 M M	offset if AA=01

Mnemonics © Intel, 1979

DATA TRANSFER INSTRUCTIONS (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
MOVB = Move byte variable						
Memory to register	R R R 0 0 A A 0	1 0 0 0 0 0 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 0	1 0 0 0 0 1 M M	offset if AA=01			
Memory to memory	0 0 0 0 0 A A 0	1 0 0 1 0 0 M M	offset if AA=01	0 0 0 0 0 A A 0	1 1 0 0 1 1 M M	offset if AA=01
MOVBI = Move byte immediate						
Immediate to register	R R R 0 1 0 0 0	0 0 1 1 0 0 0 0	data-8			
Immediate to memory	0 0 0 0 1 A A 0	0 1 0 0 1 1 M M	offset if AA=01	data-8		
MOVI = Move word immediate						
Immediate to register	R R R 1 0 0 0 1	0 0 1 1 0 0 0 0	data-lo	data-hi		
Immediate to memory	0 0 0 1 0 A A 1	0 1 0 0 1 1 M M	offset if AA=01	data-lo	data-hi	
MOVP = Move pointer						
Memory to pointer register	P P P 0 0 A A 1	1 0 0 0 1 1 M M	offset if AA=01			
Pointer register to memory	P P P 0 0 A A 1	1 0 0 1 1 0 M M	offset if AA=01			
LPD = Load pointer with doubleword variable	P P P 0 0 A A 1	1 0 0 0 1 0 M M	offset if AA=01			
LPDI = Load pointer with doubleword immediate	P P P 1 0 0 0 1	0 0 0 0 1 0 0 0	offset-lo	offset-hi	segment-lo	segment-hi

ARITHMETIC INSTRUCTIONS

ADD = Add word variable						
Memory to register	R R R 0 0 A A 1	1 0 1 0 0 0 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 1	1 1 0 1 0 0 M M	offset if AA=01			
ADDB = Add byte variable						
Memory to register	R R R 0 0 A A 0	1 0 1 0 0 0 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 0	1 1 0 1 0 0 M M	offset if AA=01			
ADDI = Add word immediate						
Immediate to register	R R R 1 0 0 0 1	0 0 1 0 0 0 0 0	data-lo	data-hi		
Immediate to memory	0 0 0 1 0 A A 1	1 1 0 0 0 0 M M	offset if AA=01	data-lo	data-hi	

Table 4-24. 8089 Instruction Encoding (Cont'd.)

ARITHMETIC INSTRUCTIONS (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ADDBI = Add byte immediate						
Immedaite to register	R R R 0 1 0 0 0	0 0 1 0 0 0 0 0	data-8			
Immediate to memory	0 0 0 0 1 A A 0	1 1 0 0 0 0 M M	offset if AA=01	data-8		
INC = Increment word by 1						
Register	R R R 0 0 0 0 0	0 0 1 1 1 0 0 0				
Memory	0 0 0 0 0 A A 1	1 1 1 0 1 0 M M	offset if AA=01			
INCB = Increment byte by 1	0 0 0 0 0 A A 0	1 1 1 0 1 0 M M	offset if AA=01			
DEC = Decrement word by 1						
Register	R R R 0 0 0 0 0	0 0 1 1 1 1 0 0				
Memory	0 0 0 0 0 A A 1	1 1 1 0 1 1 M M	offset if AA=01			
DECB = Decrement byte by 1	0 0 0 0 0 A A 0	1 1 1 0 1 1 M M	offset if AA=01			

LOGICAL AND BIT MANIPULATION INSTRUCTIONS

AND = AND word variable						
Memory to register	R R R 0 0 A A 1	1 0 1 0 1 0 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 1	1 1 0 1 1 0 M M	offset if AA=01			
ANDB = AND byte variable						
Memory to register	R R R 0 0 A A 0	1 0 1 0 1 0 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 0	1 1 0 1 1 0 M M	offset if AA=01			
ANDI = AND word immediate						
Immediate to register	R R R 1 0 0 0 1	0 0 1 0 1 0 0 0	data-lo	data-hi		
Immediate to memory	0 0 0 1 0 A A 1	1 1 0 0 1 0 M M	offset if AA=01	data-lo	data-hi	
ANDBI = AND byte immediate						
Immediate to register	R R R 0 1 0 0 0	0 0 1 0 1 0 0 0	data-8			
Immediate to memory	0 0 0 0 1 A A 0	1 1 0 0 1 0 M M	offset if AA=01	data-8		
OR = OR word variable						
Memory to register	R R R 0 0 A A 1	1 0 1 0 0 1 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 1	1 1 0 1 0 1 M M	offset if AA=01			

Mnemonics © Intel, 1979

Table 4-24. 8089 Instruction Encoding (Cont'd.)

LOGICAL AND BIT MANIPULATION INSTRUCTIONS (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
ORB = OR byte variable						
Memory to register	R R R 0 0 A A 0	1 0 1 0 0 1 M M	offset if AA=01			
Register to memory	R R R 0 0 A A 0	1 1 0 1 0 1 M M	offset if AA=01			
ORI = OR word immediate						
Immediate to register	R R R 1 0 0 0 1	0 0 1 0 0 1 0 0	data-lo	data-hi		
Immediate to memory	0 0 0 1 0 A A 1	1 1 0 0 0 1 M M	offset if AA=01	data-lo	data-hi	
ORBI = OR byte immediate						
Immediate to register	R R R 0 1 0 0 0	0 0 1 0 0 1 0 0	data-8			
Immediate to memory	0 0 0 0 1 A A 0	1 1 0 0 0 1 M M	offset if AA=01	data-8		
NOT = NOT word variable						
Register	R R R 0 0 0 0 0	0 0 1 0 1 1 0 0				
Memory	0 0 0 0 0 A A 1	1 1 0 1 1 1 M M	offset if AA=01			
Memory to register	R R R 0 0 A A 1	1 0 1 0 1 1 M M	offset if AA=01			
NOTB = NOT byte variable						
Memory	0 0 0 0 0 A A 0	1 1 0 1 1 1 M M	offset if AA=01			
Memory to register	R R R 0 0 A A 0	1 0 1 0 1 1 M M	offset if AA=01			
SETB = Set bit to 1	B B B 0 0 A A 0	1 1 1 1 0 1 M M	offset if AA=01			
CLR = Clear bit to 0	B B B 0 0 A A 0	1 1 1 1 1 0 M M	offset if AA=01			

PROGRAM TRANSFER INSTRUCTIONS

***CALL** = Call	1 0 0 0 1 A A 1	1 0 0 1 1 1 M M	offset if AA=01	disp-8	
LCALL = Long call	1 0 0 1 0 A A 1	1 0 0 1 1 1 M M	offset if AA=01	disp-lo	disp-hi
***JMP** = Jump unconditional	1 0 0 0 1 0 0 0	0 0 1 0 0 0 0 0	disp-8		
LJMP = Long jump unconditional	1 0 0 1 0 0 0 1	0 0 1 0 0 0 0 0	disp-lo	disp-hi	

*The ASM-89 Assembler will automatically generate the long form of a program transfer instruction when the target is known to be beyond the byte-displacement range.

Table 4-24. 8089 Instruction Encoding (Cont'd.)

PROGRAM TRANSFER INSTRUCTIONS (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
***JZ** = Jump if word is 0						
Label to register	R R R 0 1 0 0 0	0 1 0 0 0 1 0 0	disp-8			
Label to memory	0 0 0 0 1 A A 1	1 1 1 0 0 1 M M	offset if AA=01	disp-8		
LJZ = Long jump if word is 0						
Label to register	R R R 1 0 0 0 0	0 1 0 0 0 1 0 0	disp-lo	disp-hi		
Label to memory	0 0 0 1 0 A A 1	1 1 1 0 0 1 M M	offset if AA=01	disp-lo	disp-hi	
***JZB** = Jump if byte is 0	0 0 0 0 1 A A 0	1 1 1 0 0 1 M M	offset if AA=01	disp-8		
LJZB = Long jump if byte is 0	0 0 0 1 0 A A 0	1 1 1 0 0 1 M M	offset if AA=01	disp-lo	disp-hi	
***JNZ** = Jump if word not 0						
Label to register	R R R 0 1 0 0 0	0 1 0 0 0 0 0 0	disp-8			
Label to memory	0 0 0 0 1 A A 1	1 1 1 0 0 0 M M	offset if AA=01	disp-8		
LJNZ = Long jump if word not 0						
Label to register	R R R 1 0 0 0 0	0 1 0 0 0 0 0 0	disp-lo	disp-hi		
Label to memory	0 0 0 1 0 A A 1	1 1 1 0 0 0 M M	offset if AA=01	disp-lo	disp-hi	
***JNZB** = Jump if byte not 0	0 0 0 0 1 A A 0	1 1 1 0 0 0 M M	offset if AA=01	disp-8		
LJNZB = Long jump if byte not 0	0 0 0 1 0 A A 0	1 1 1 0 0 0 M M	offset if AA=01	disp-lo	disp-hi	
***JMCE** = Jump if masked compare equal	0 0 0 0 1 A A 0	1 0 1 1 0 0 M M	offset if AA=01	disp-8		
LJMCE = Long jump if masked compare equal	0 0 0 1 0 A A 0	1 0 1 1 0 0 M M	offset if AA=01	disp-lo	disp-hi	
***JMCNE** = Jump if masked compare not equal	0 0 0 0 1 A A 0	1 0 1 1 0 1 M M	offset if AA=01	disp-8		
LJMCNE = Long jump if masked compare not equal	0 0 0 1 0 A A 0	1 0 1 1 0 1 M M	offset if AA=01	disp-lo	disp-hi	
***JBT** = Jump if bit is 1	B B B 0 1 A A 0	1 0 1 1 1 1 M M	offset if AA=01	disp-8		

*The ASM-89 Assembler will automatically generate the long form of a program transfer instruction when the target is known to be beyond the byte-displacement range.

Mnemonics © Intel, 1979

PROGRAM TRANSFER INSTRUCTIONS (Cont'd.)

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
LJBT = Long jump if bit is 1	B B B 1 0 A A 0	1 0 1 1 1 1 M M	offset if AA=01	disp-lo	disp-hi	
***JNBT** = Jump if bit is not 1	B B B 0 1 A A 0	1 0 1 1 1 0 M M	offset if AA=01	disp-8		
LJNBT = Long jump if bit is not 1	B B B 1 0 A A 0	1 0 1 1 1 0 M M	offset if AA=01	disp-lo	disp-hi	

PROCESSOR CONTROL INSTRUCTIONS

TSL = Test and set while locked	0 0 0 1 1 A A 0	1 0 0 1 0 1 M M	offset if AA=01	data-8	disp-8
WID = Set logical bus widths	1 S D* 0 0 0 0 0	0 0 0 0 0 0 0 0			

*S=source width, D=destination width; 0=8 bits, 1=16 bits

XFER = Enter DMA mode	0 1 1 0 0 0 0 0	0 0 0 0 0 0 0 0
SINTR = Set interrupt service bit	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0
HLT = Halt channel program	0 0 1 0 0 0 0 0	0 1 0 0 1 0 0 0
NOP = No operation	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

*The ASM-89 Assembler will automatically generate the long form of a program transfer instruction when the target is known to be beyond the byte-displacement range.

Table 4-26 lists all of the 8089 machine instructions in hexadecimal/binary order by their *second* byte. This table may be used to "decode" an assembled machine instruction into its ASM-89 symbolic form. The preceding table (table 4-25) defines the notation used in table 4-26.

Table 4-25. Key to 8089 Machine Instruction Decoding Guide

Identifier	Explanation
S	Logical width of source bus; 0=8, 1=16
D	Logical width of destination bus; 0=8, 1=16
PPP	Pointer register encoded in R/B/P field
RRR	Register encoded in R/B/P field
AA	AA (addressing mode) field
BBB	Bit select encoded in R/B/P field
offset-lo	Low-order byte of offset word in doubleword pointer
offset-hi	High-order byte of offset word in doubleword pointer
segment-lo	Low-order byte of segment word in doubleword pointer
segment-hi	High-order byte of segment word in doubleword pointer
data-8	8-bit immediate constant
data-lo	Low-order byte of 16-bit immediate constant
data-hi	High-order byte of 16-bit immediate constant
disp-8	8-bit signed displacement
disp-lo	Low-order byte of 16-bit signed displacement
disp-hi	High-order byte of 16-bit signed displacement
(offset)	Optional 8-bit offset used in offset addressing

Table 4-26. 8089 Machine Instruction Decoding Guide

Byte 1	Byte 2		Bytes 3, 4, 5, 6	ASM89 Instruction Format
	Hex	Binary		
00000000	00	00000000		NOP
01000000	00	00000000		SINTR
1SD00000	00	00000000		WID source-width,dest-width
01100000	00	00000000		XFER
	01	00000001		not used
	↓	↓		
	07	00000111		
PPP10001	08	00001000	offset-lo,offset-hi,segment-lo,segment-hi	LPDI ptr-reg,immed32
	09	00001001		not used
	↓	↓		
	1F	00011111		
RRR01000	20	00100000	data-8	ADDBI register,immed8
RRR10001	20	00100000	data-lo,data-hi	ADDI register,immed16
10001000	20	00100000	disp-8	JMP short-label
10010001	20	00100000	disp-lo,disp-hi	LJMP long-label
	21	00100001		not used
	↓	↓		
	23	00100011		
RRR01000	24	00100100	data-8	ORBI register,immed8
RRR10001	24	00100100	data-lo,data-hi	ORI register,immed16
	25	00100101		not used
	↓	↓		
	27	00100111		
RRR01000	28	00101000	data-8	ANDBI register,immed8

Mnemonics © Intel, 1979

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

Byte 1	Byte 2		Bytes 3, 4, 5, 6	ASM89 Instruction Format
	Hex	Binary		
RRR10001	28	00101000	data-lo,data-hi	ANDI register,immed16
	29	00101001		not used
	↓	↓		
	2B	00101011		
RRR00000	2C	00101100		NOT register
	2D	00101101		not used
	↓	↓		
	2F	00101111		
RRR01000	30	00110000	data-8	MOVBI register,immed8
RRR10001	30	00110000	data-lo,data-hi	MOVI register,immed16
	31	00110001		not used
	↓	↓		
	37	00110111		
RRR00000	38	00111000		INC register
	39	00111001		not used
	↓	↓		
	3B	00111011		
RRR00000	3C	00111100		DEC register
	3D	00111101		not used
	↓	↓		
	3F	00111111		
RRR01000	40	01000000	disp-8	JNZ register,short-label
RRR10000	40	01000000	disp-lo,disp-hi	LJNZ register,long-label
	41	01000001		not used
	↓	↓		
	43	01000011		
RRR01000	44	01000100	disp-8	JZ register,short-label
RRR10000	44	01000100	disp-lo,disp-hi	LJZ register,short-label
	45	01000101		not used
	↓	↓		
	47	01000111		
00100000	48	01001000		HLT
	49	01001001		not used
	↓	↓		
	4B	01001011		
00001AA0	4C	010011MM	(offset),data-8	MOVBI mem8,immed8
↓	↓	↓		
00001AA0	4F	010011MM		
00010AA1	4C	010011MM	(offset),data-lo,data-hi	MOVI mem16,immed16
↓	↓	↓		
00010AA1	4F	010011MM		
	50	01010000		not used
	↓	↓		
	7F	01111111		
RRR00AA0	80	100000MM	(offset)	MOVB register,mem8
↓	↓	↓		
RRR00AA0	83	100000MM		

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

Byte 1	Byte 2		Bytes 3, 4, 5, 6	ASM89 Instruction Format
	Hex	Binary		
RRR00AA1 ↓ RRR00AA1	80 ↓ 83	100000MM ↓ 100000MM	(offset)	MOV register,mem16
RRR00AA0 ↓ RRR00AA0	84 ↓ 87	100001MM ↓ 100001MM	(offset)	MOVB mem8,register
RRR00AA1 ↓ RRR00AA1	84 ↓ 87	100001MM ↓ 100001MM	(offset)	MOV mem16,register
PPP00AA1 ↓ PPP00AA1	88 ↓ 8B	100010MM ↓ 100010MM	(offset)	LPD ptr-reg,mem32
PPP00AA1 ↓ PPP00AA1	8C ↓ 8F	100011MM ↓ 100011MM	(offset)	MOVP ptr-reg,mem24
00000AA0 ↓ 00000AA0	90 ↓ 93	100100MM ↓ 100100MM	(offset),00000AA0,110011MM,(offset)	MOVB mem8,mem8
00000AA1 ↓ 00000AA1	90 ↓ 93	100100MM ↓ 100100MM	(offset),00000AA1,110011MM,(offset)	MOV mem16,mem16
00011AA0 ↓ 00011AA0	94 ↓ 97	100101MM ↓ 100101MM	(offset),data-8,disp-8	TSL mem8,immed8,short-label
PPP00AA1 ↓ PPP00AA1	98 ↓ 9B	100110MM ↓ 100110MM	(offset)	MOVP mem24,ptr-reg
10001AA1 ↓ 10001AA1	9C ↓ 9F	100111MM ↓ 100111MM	(offset),disp-8	CALL mem24,short-label
10010AA1 ↓ 10010AA1	9C ↓ 9F	100111MM ↓ 100111MM	(offset),disp-lo,disp-hi	LCALL mem24,long-label
RRR00AA0 ↓ RRR00AA0	A0 ↓ A3	101000MM ↓ 101000MM	(offset)	ADDB register,mem8
RRR00AA1 ↓ RRR00AA1	A0 ↓ A3	101000MM ↓ 101000MM	(offset)	ADD register,mem16
RRR00AA0 ↓ RRR00AA0	A4 ↓ A7	101001MM ↓ 101001MM	(offset)	ORB register,mem8
RRR00AA1 ↓ RRR00AA1	A4 ↓ A7	101001MM ↓ 101001MM	(offset)	OR register,mem16
RRR00AA0 ↓ RRR00AA0	A8 ↓ AB	101010MM ↓ 101010MM	(offset)	ANDB mem8,register

Mnemonics © Intel, 1979

Byte 1	Byte 2 Hex	Byte 2 Binary	Bytes 3, 4, 5, 6	ASM89 Instruction Format
RRR00AA1 ↓ RRR00AA1	A8 ↓ AB	101010MM ↓ 101010MM	(offset)	AND mem16,register
RRR00AA0 ↓ RRR00AA0	AC ↓ AF	101011MM ↓ 101011MM	(offset)	NOTB register,mem8
RRR00AA1 ↓ RRR00AA1	AC ↓ AF	101011MM ↓ 101011MM	(offset)	NOT register,mem16
00001AA0 ↓ 00001AA0	B0 ↓ B3	101100MM ↓ 101100MM	(offset),disp-8	JMCE mem8,short-label
00010AA0 ↓ 00010AA0	B0 ↓ B3	101100MM ↓ 101100MM	(offset),disp-lo,disp-hi	LJMCE mem8,long-label
00001AA0 ↓ 00001AA0	B4 ↓ B7	101101MM ↓ 101101MM	(offset),disp-8	JMCNE mem8,short-label
00010AA0 ↓ 00010AA0	B4 ↓ B7	101101MM ↓ 101101MM	(offset),disp-lo,disp-hi	LJMCNE mem8,long-label
BBB01AA0 ↓ BBB01AA0	B8 ↓ BB	101110MM ↓ 101110MM	(offset),disp-8	JNBT mem8,bit-select,short-label
BBB10AA0 ↓ BBB10AA0	B8 ↓ BB	101110MM ↓ 101110MM	(offset),disp-lo,disp-hi	LJNBT mem8,bit-select,long-label
BBB01AA0 ↓ BBB01AA0	BC ↓ BF	101111MM ↓ 101111MM	(offset),disp-8	JBT mem8,bit-select,short-label
BBB10AA0 ↓ BBB10AA0	BC ↓ BF	101111MM ↓ 101111MM	(offset),disp-lo,disp-hi	LJBT mem8,bit-select,long-label
00001AA0 ↓ 00001AA0	C0 ↓ C3	110000MM ↓ 110000MM	(offset),data-8	ADDBI mem8,immed8
00010AA1 ↓ 00010AA1	C0 ↓ C3	110000MM ↓ 110000MM	(offset),data-lo,data-hi	ADDI mem16,immed16
00001AA0 ↓ 00001AA0	C4 ↓ C7	110001MM ↓ 110001MM	(offset),data-8	ORBI mem8,immed8
00010AA1 ↓ 00010AA1	C4 ↓ C7	110001MM ↓ 110001MM	(offset),data-lo,data-hi	ORI mem16,immed16
00001AA0 ↓ 00001AA0	C8 ↓ CB	110010MM ↓ 110010MM	(offset),data-8	ANDBI mem8,immed8

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

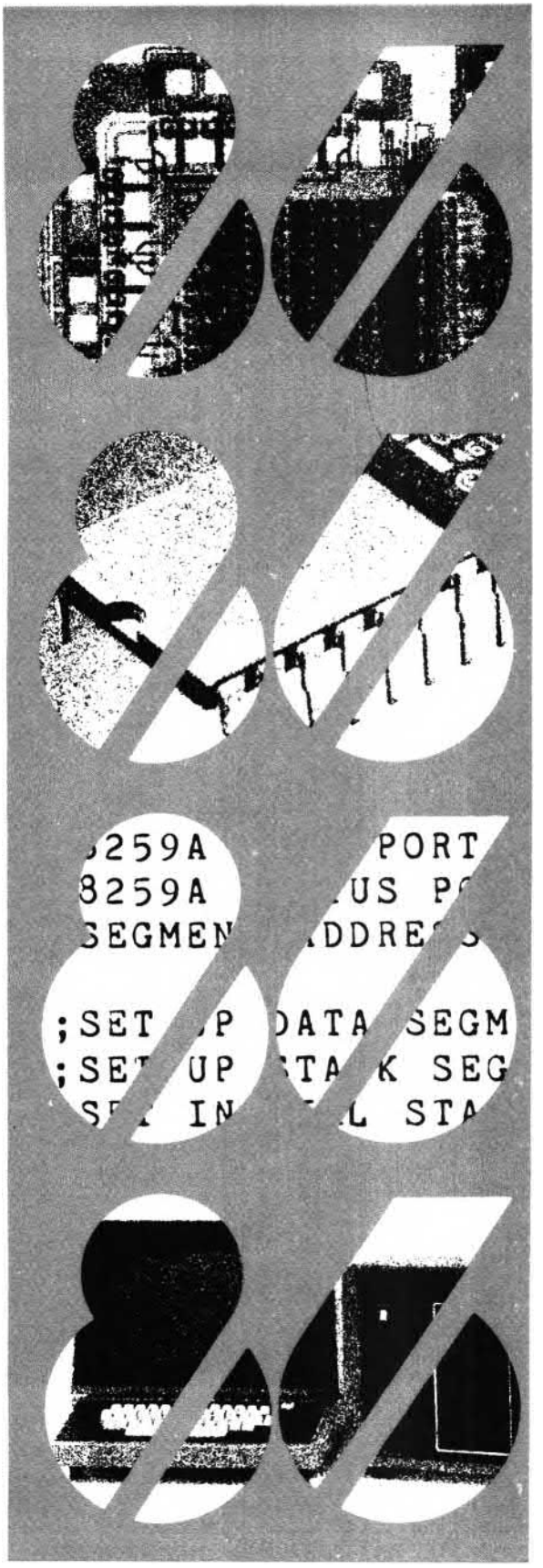
Byte 1	Byte 2 Hex	Byte 2 Binary	Bytes 3, 4, 5, 6	ASM89 Instruction Format
00010AA1 ↓ 00010AA1	C8 ↓ CB	110010MM ↓ 110010MM	(offset),data-lo,data-hi	ANDI mem16,immed16
	CC ↓ CF	11001100 ↓ 11001111		not used
RRR00AA0 ↓ RRR00AA0	D0 ↓ D3	110100MM ↓ 110100MM	(offset)	ADDB mem8,register
RRR00AA1 ↓ RRR00AA1	D0 ↓ D3	110100MM ↓ 110100MM	(offset)	ADD mem16,register
RRR00AA0 ↓ RRR00AA0	D4 ↓ D7	110101MM ↓ 110101MM	(offset)	ORB mem8,register
RRR00AA1 ↓ RRR00AA1	D4 ↓ D7	110101MM ↓ 110101MM	(offset)	OR mem16,register
RRR00AA0 ↓ RRR00AA0	D8 ↓ DB	110110MM ↓ 110110MM	(offset)	ANDB mem8,register
RRR00AA1 ↓ RRR00AA1	D8 ↓ DB	110110MM ↓ 110110MM	(offset)	AND mem16,register
RRR00AA0 ↓ RRR00AA0	DC ↓ DF	110111MM ↓ 110111MM	(offset)	NOTB mem8,register
RRR00AA1 ↓ RRR00AA1	DC ↓ DF	110111MM ↓ 110111MM	(offset)	NOT mem16,register
00001AA0 ↓ 00001AA0	E0 ↓ E3	111000MM ↓ 111000MM	(offset),disp-8	JNZB mem8,short-label
00001AA1 ↓ 00001AA1	E0 ↓ E3	111000MM ↓ 111000MM	(offset),disp-8	JNZ mem16,short-label
00010AA0 ↓ 00010AA0	E0 ↓ E3	111000MM ↓ 111000MM	(offset),disp-lo,disp-hi	LJNZB mem8,long-label
00010AA1 ↓ 00010AA1	E0 ↓ E3	111000MM ↓ 111000MM	(offset),disp-lo,disp-hi	LJNZ mem16,longlabel
00001AA0 ↓ 00001AA0	E4 ↓ E7	111001MM ↓ 111001MM	(offset),disp-8	JZB mem8,short-label
00001AA1 ↓ 00001AA1	E4 ↓ E7	111001MM ↓ 111001MM	(offset),disp-8	JZ mem16,short-label

Mnemonics © Intel, 1979

Table 4-26. 8089 Machine Instruction Decoding Guide (Cont'd.

Byte 1	Byte 2		Bytes 3, 4, 5, 6	ASM89 Instruction Format
	Hex	Binary		
00010AA0 ↓ 00010AA0	E4 ↓ E7	111001MM ↓ 111001MM	(offset),disp-lo,disp-hi	LJZB mem8,long-label
00010AA1 ↓ 00010AA1	E4 ↓ E7	111001MM ↓ 111001MM	(offset),disp-lo,disp-hi	LJZ mem16,long-label
00000AA0 ↓ 00000AA0	E8 ↓ EB	111010MM ↓ 111010MM	(offset)	INCB mem8
00000AA1 ↓ 00000AA1	E8 ↓ EB	111010MM ↓ 111010MM	(offset)	INC mem16
00000AA0 ↓ 00000AA0	EC ↓ EF	111011MM ↓ 111011MM	(offset)	DECB mem8
00000AA1 ↓ 00000AA1	EC ↓ EF	111011MM ↓ 111011MM	(offset)	DEC mem16
	F0 ↓ F3	11110000 ↓ 11110000		not used
BBB00AA0 ↓ BBB00AA0	F4 ↓ F7	111101MM ↓ 111101MM	(offset)	SETB mem8,0-7
BBB00AA0 ↓ BBB00AA0	F8 ↓ FB	111110MM ↓ 111110MM	(offset)	CLR mem8,0-7
	FC ↓ FF	11111100 ↓ 11111111		not used

Appendix A
Application Notes



APPENDIX A
APPLICATION NOTES

This appendix contains Intel application notes pertinent to the 8086 family microprocessors. The following application notes, in the order listed, have been included within this appendix:

AP-67	8086 System Design
AP-61	Multitasking for the 8086
AP-50	Debugging Strategies and Considerations for 8089 Systems
AP-51	Designing 8086, 8088, 8089 Multiprocessing Systems with the 8289 Bus Arbiter
AP-59	Using the 8259A Programmable Interrupt Controller
AP-28A	Intel® Multibus™ Interfacing
AP-43	Using the iSBC-957™ Execution Vehicle for Executing 8086 Program Code

intel

APPLICATION NOTE

AP-67

September 1979

8086 System Design

George Alexy
Microcomputer Applications

© Intel Corporation 1979

1. INTRODUCTION

2. 8086 OVERVIEW AND BASIC SYSTEM CONCEPTS

A. Bus Cycle Definition
B. Address and Data Bus Concepts
C. System Data Bus Concepts
D. Multiprocessor Environment

3. 8086 SYSTEM DETAILS

A. Operating Modes
B. Clock Generation
C. Reset
D. Ready Implementation and Timing
E. Interrupt Structure
F. Interpreting the 8086 Bus Timing Diagrams
G. Bus Control Transfer

4. INTERFACING WITH I/O

5. INTERFACING WITH MEMORIES

6. APPENDIX

1. INTRODUCTION

The 8086 family, Intel's new series of microprocessors and system components, offers the designer an advanced system architecture which can be structured to satisfy a broad range of applications. The variety of speed, configuration and component selections available within the family enables optimization of a specific design to both cost and performance objectives. More important however, the 8086 family concept allows the designer to develop a family of systems providing multiple levels of enhancement within a single design and a growth path for future designs.

This application note is directed toward the implementation of the system hardware and will provide an introduction to a representative sample of the systems configurable with the 8086 CPU member of the family. Application techniques and timing analysis will be given to aid the designer in understanding the system requirements, advantages and limitations. Additional Intel publications the reader may wish to reference are the 8086 User's Manual (9800722A), 8086 Assembly Language Reference Guide (9800749A), AP-28A MULTIBUS™ Interfacing (98005876B), INTEL MULTIBUS™ SPECIFICATION (9800683), AP-45 Using the 8202 Dynamic RAM Controller (9800809A), AP-51 Designing 8086, 8088, 8089 Multiprocessor Systems with the 8289 Bus Arbiter and AP-59 Using the 8259A Programmable Interrupt Controller. References to other Intel publications will be made throughout this note.

2. 8086 OVERVIEW AND BASIC SYSTEM CONCEPTS

2A. 8086 Bus Cycle Definition

The 8086 is a true 16-bit microprocessor with 16-bit internal and external data paths, one megabyte of memory address space (2**20) and a separate 64K byte (2**16) I/O address space. The CPU communicates with its external environment via a twenty-bit time multiplexed address, status and data bus and a command bus. To transfer data or fetch instructions, the CPU executes a bus cycle (Fig. 2A1). The minimum bus cycle consists of four CPU clock cycles called T states. During the first T state (T1), the CPU asserts an address on the twenty-bit

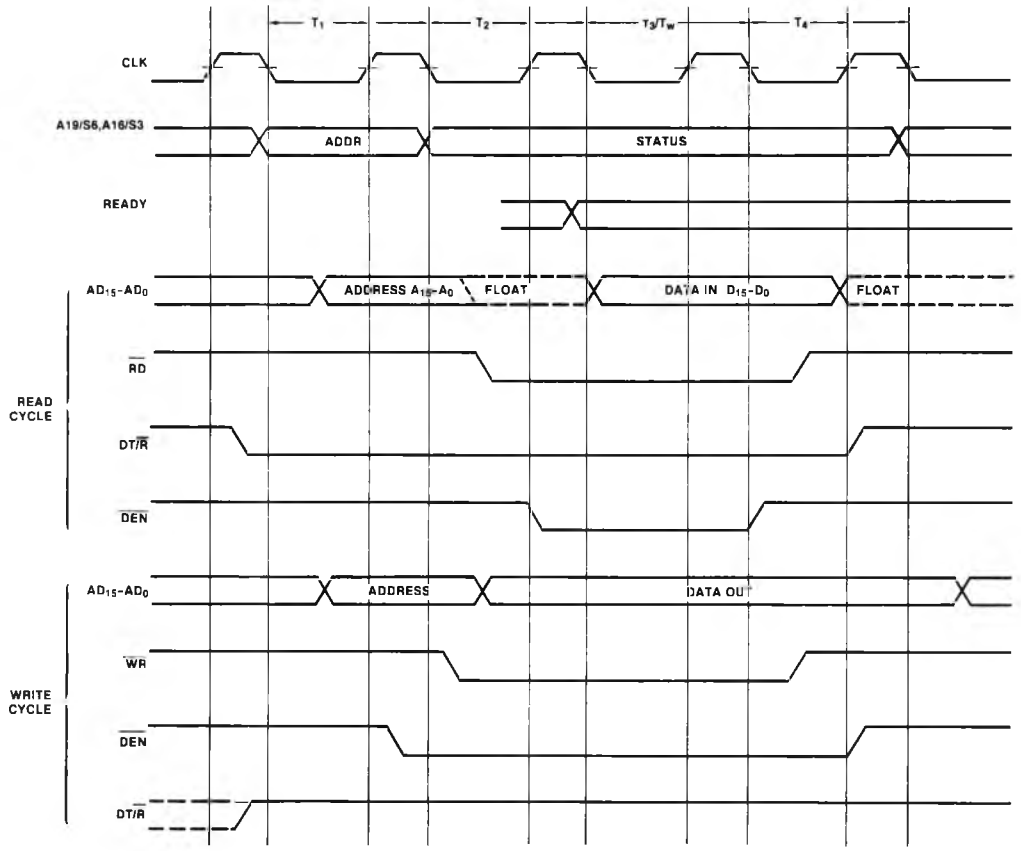


Figure 2A1. Basic 8086 Bus Cycle

multiplexed address/data/status bus. For the second T state (T2), the CPU removes the address from the bus and either three-states its outputs on the lower sixteen bus lines in preparation for a read cycle or asserts write data. Data bus transceivers are enabled in either T1 or T2 depending on the 8086 system configuration and the direction of the transfer (into or out of the CPU). Read, write or interrupt acknowledge commands are always enabled in T2. The maximum mode 8086 configuration (to be discussed later) also provides a write command enabled in T3 to guarantee data setup time prior to command activation.

During T2, the upper four multiplexed bus lines switch from address (A19-A16) to bus cycle status (S6,S5,S4,S3). The status information (Table 2A1) is available primarily for diagnostic monitoring. However, a decode of S3 and S4 could be used to select one of four banks of memory, one assigned to each segment register. This technique allows partitioning the memory by segment to expand the memory addressing beyond one megabyte. It also provides a degree of protection by preventing erroneous write operations to one segment from overlapping into another segment and destroying information in that segment.

The CPU continues to provide status information on the upper four bus lines during T3 and will either continue to assert write data or sample read data on the lower sixteen bus lines. If the selected memory or I/O device is not capable of transferring data at the maximum CPU transfer rate, the device must signal the CPU "not ready" and force the CPU to insert additional clock cycles (Wait states TW) after T3. The 'not ready' indication must be presented to the CPU by the start of T3. Bus activity during TW is the same as T3. When the selected device has had sufficient time to complete the transfer, it asserts "Ready" and allows the CPU to continue from the TW states. The CPU will latch the data on the bus during the last wait state or during T3 if no wait states are requested. The bus cycle is terminated in T4 (command lines are disabled and the selected external device deselects from the bus). The bus cycle appears to devices in the system as an asynchronous event consisting of an address to select the device followed by a read strobe or data and a write strobe. The selected device accepts bus data during a write cycle and drives the desired data onto the bus during a read cycle. On termination of the command, the device latches write data or disables its bus drivers. The only control the device has on the bus cycle is the insertion of wait cycles.

The 8086 CPU only executes a bus cycle when instructions or operands must be transferred to or from memory or I/O devices. When not executing a bus cycle, the bus interface executes idle cycles (TI). During the idle cycles, the CPU continues to drive status information from the previous bus cycle on the upper address lines. If the previous bus cycle was a write, the CPU continues to drive the write data onto the multiplexed bus until the start of the next bus cycle. If the CPU executes idle cycles following a read cycle, the CPU will not drive the lower 16 bus lines until the next bus cycle is required.

Since the CPU prefetches up to six bytes of the instruction stream for storage and execution from an internal instruction queue, the relationship of instruction fetch and associated operand transfers may be skewed in time and separated by additional instruction fetch bus cycles. In general, if an instruction is fetched into the 8086's internal instruction queue, several additional instructions may be fetched before the instruction is removed from the queue and executed. If the instruction being executed from the queue is a jump or other control transfer instruction, any instructions remaining in the queue are not executed and are discarded with no effect on the CPU's operation. The bus activity observed during execution of a specific instruction is dependent on the preceding instructions but is always deterministic within the specific sequence.

Table 2A1

S3	S4	
0	0	Alternate (relative to the ES segment)
1	0	Stack (relative to the SS segment)
0	1	Code/None (relative to the CS segment or a default of zero)
1	1	Data (relative to the DS segment)

S5 = IF (interrupt enable flag)
S6 = 0 (indicates the 8086 is on the bus)

2B. 8086 Address and Data Bus Concepts

Since the majority of system memories and peripherals require a stable address for the duration of the bus cycle, the address on the multiplexed address/data bus during T1 should be latched and the latched address used to select the desired peripheral or memory location. Since the 8086 has a 16-bit data bus, the multiplexed bus components of the 8085 family are not applicable to the 8086 (a device on address/data bus lines 8-15 will not be able to receive the byte selection address on lines 0-7). To demultiplex the bus (Fig. 2B1a), the 8086 system provides an Address Latch Enable signal (ALE) to capture the address in either the 8282 or 8283 8-bit bi-stable latches (Diag. 2B1). The latches are either inverting (8283) or non-inverting (8282) and have outputs driven by three-state buffers that supply 32 mA drive capability and can switch a 300 pF capacitive load in 22 ns (inverting) or 30 ns (non-inverting). They propagate the address through to the outputs while ALE is high and latch the address on the falling edge of ALE. This only delays address access and chip select decoding by the propagation delay of the latch. The outputs are enabled through the low active $\overline{OE}$ input. The demultiplexing of the multiplexed address/data bus (latchings of the address from the multiplexed bus), can be done locally at appropriate points in the system or at the CPU with a separate address bus distributing the address throughout the system (Fig. 2B1b). For optimum system performance and compatibility with multiprocessor and MULTIBUS™ configurations, the latter technique is strongly recommended over the first. The remainder of this note will assume the bus is demultiplexed at the CPU.

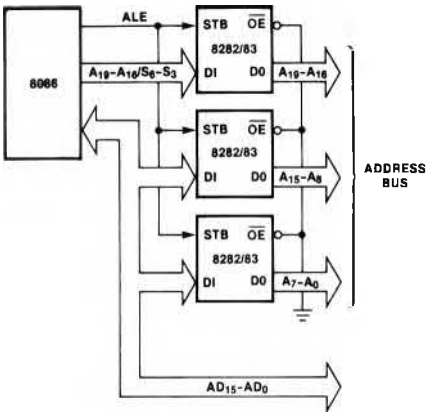


Figure 2B1a. Demultiplexing the 8086 Bus

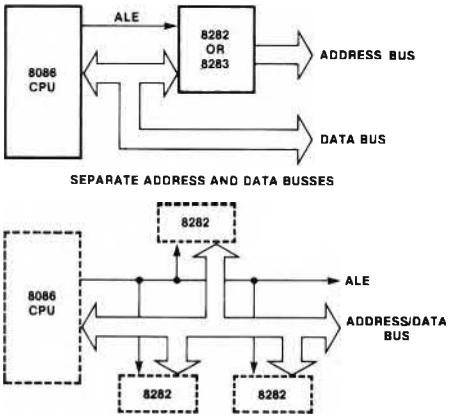


Figure 2B1b.

The programmer views the 8086 memory address space as a sequence of one million bytes in which any byte may contain an eight bit data element and any two consecutive bytes may contain a 16-bit data element. There is no constraint on byte or word addresses (boundaries). The address space is physically implemented on a sixteen bit data bus by dividing the address space into two banks of up to 512K bytes (Fig. 2B2). One bank is connected to the lower half of the sixteen-bit data bus (D7-0) and contains even addressed bytes (A0 = 0). The other bank is connected to the upper half of the data bus (D15-8) and contains odd addressed bytes (A0 = 1). A specific byte within each bank is selected by address lines A19-A1. To perform byte transfers to even addresses (Fig. 2B3a), the information is transferred over the lower half of the data bus (D7-0). A0 (active low) is used to enable the bank connected to the lower half of the data bus to participate in the transfer. Another signal provided by the 8086, Bus High Enable ($\overline{BHE}$), is used to disable the bank on the upper half of the data bus from participating in the transfer. This is necessary to prevent a write operation to the lower bank from destroying data in the upper bank. Since $\overline{BHE}$ is a multiplexed signal with timing identical to the A19-A16 address lines, it also should be latched with ALE to provide a stable signal during the bus cycle. During T2 through T4, the $\overline{BHE}$ output is multiplexed with status line S7 which is equal to $\overline{BHE}$. To perform byte transfers to odd addresses (Fig. 2B3b), the information is transferred over the upper half of the data bus (D15-D8) while $\overline{BHE}$ (active low) enables the upper bank and A0 disables the lower bank. Directing the data transfer to the appropriate half of the data bus and activation of $\overline{BHE}$ and A0 is performed by the 8086, transparent to the programmer. As an example, consider loading a byte of data into the CL register (lower half of the CX register) from an odd addressed memory location (referenced over the upper half of the 16-bit data bus). The data is transferred into the 8086 over the upper 8 bits of the data bus, automatically redirected to the lower half of the 8086 internal 16-bit data path and stored into the CL register. This capability also allows byte I/O transfers with the AL register to be directed to I/O devices connected to either the upper or lower half of the 16-bit data bus.

To access even addressed sixteen bit words (two consecutive bytes with the least significant byte at an even

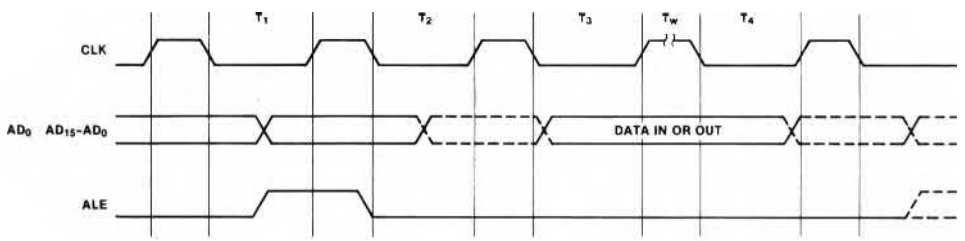


Diagram 2B1. ALE Timing

dressed 16-bit word (Fig. 2B3d), the least significant byte (addressed by A19-A1) is first transferred over the upper half of the bus (odd addressed byte, upper bank, $\overline{BHE}$ low active and A0 = 1). The most significant byte is accessed by incrementing the address (A19-A0) which allows A19-A1 to address the next physical word location (remember, A0 was equal to one which indicated a word referenced from an odd byte boundary). A second bus cycle is then executed to perform the transfer of the most significant byte with the lower bank (A0 is now active low and $\overline{BHE}$ is high). The sequence is automatically executed by the 8086 whenever a word transfer is executed to an odd address. Directing the upper and lower bytes of the 8086's internal sixteen-bit registers to the appropriate halves of the data bus is also performed automatically by the 8086 and is transparent to the programmer.

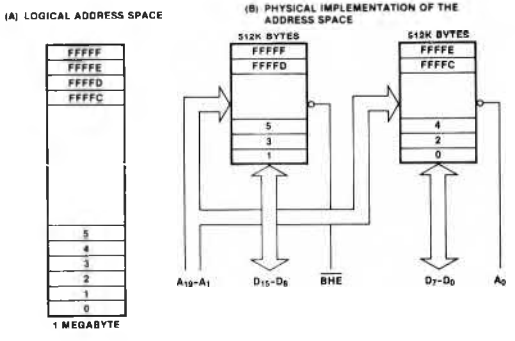


Figure 2B2. 8086 Memory

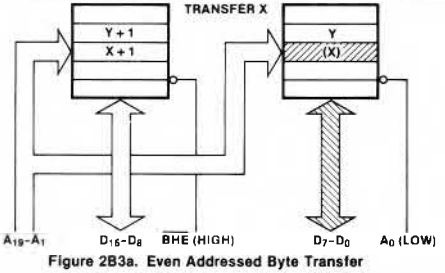


Figure 2B3a. Even Addressed Byte Transfer

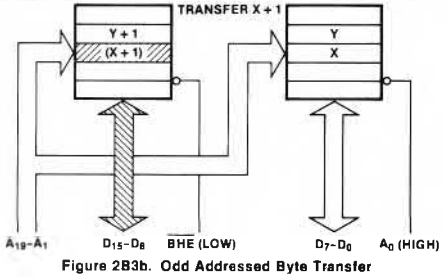


Figure 2B3b. Odd Addressed Byte Transfer

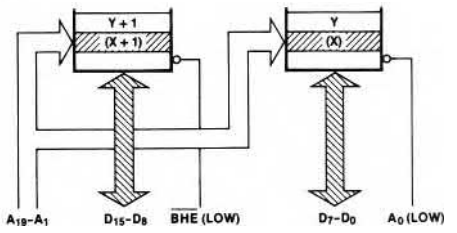


Figure 2B3c. Even Addressed Word Transfer

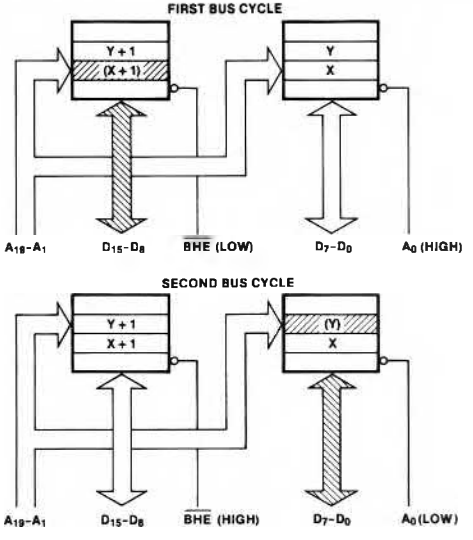


Figure 2B3d. Odd Addressed Word Transfer

During a byte read, the CPU floats the entire sixteen-bit data bus even though data is only expected on the upper or lower half of the data bus. As will be demonstrated later, this action simplifies the chip select decoding requirements for read only devices (ROM, EPROM). During a byte write operation, the 8086 will drive the entire sixteen-bit data bus. The information on the half of the data bus not transferring data is indeterminate. These concepts also apply to the I/O address space. Specific examples of I/O and memory interfacing are considered in the corresponding sections.

2C. System Data Bus Concepts

When referring to the system data bus, two implementation alternatives must be considered; (a) the multiplexed address/data bus (Fig. 2C1a) and a data bus buffered from the multiplexed bus by transceivers (Fig. 2C1b).

If memory or I/O devices are connected directly to the multiplexed bus, the designer must guarantee the devices do not corrupt the address on the bus during T1.

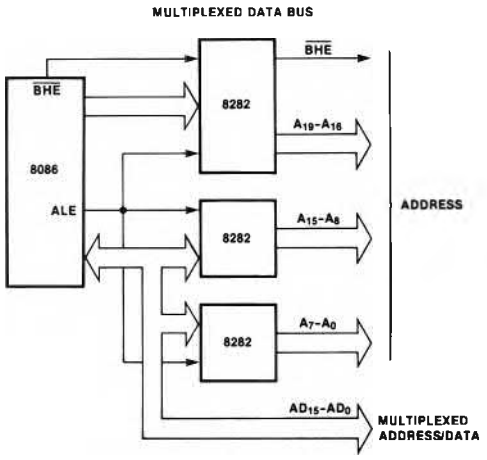


Figure 2C1a. Multiplexed Data Bus

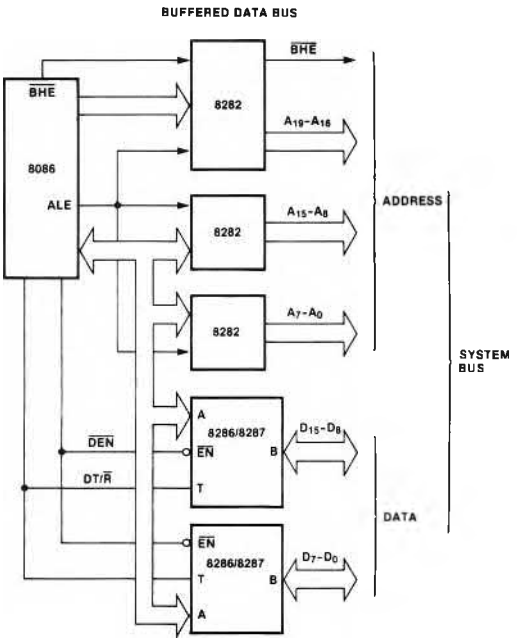


Figure 2C1b. Buffered Data Bus

To avoid this, device output drivers should not be enabled by the device chip select, but should have an output enable controlled by the system read signal (Fig. 2C2). The 8086 timing guarantees that read is not valid until after the address is latched by ALE (Diag. 2C1). All Intel peripherals, EPROM products and RAM's for microprocessors provide output enable or read inputs to allow connection to the multiplexed bus.

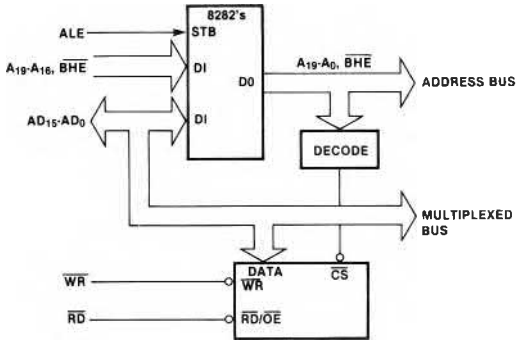


Figure 2C2. Devices with Output Enables on the Multiplexed Bus

Several techniques are available for interfacing devices without output enables to the multiplexed bus but each introduces other restrictions or limitations. Consider Figure 2C3 which has chip select gated with read and write. Two problems exist with this technique. First, the chip select access time is reduced to the read access time, and may require a faster device if maximum system performance (no wait states) is to be achieved (Diag. 2C2). Second, the designer must verify that chip select to write setup and hold times for the device are not violated (Diag. 2C3). Alternate techniques can be extracted from the bus interfacing techniques given later in this section but are subject to the associated restrictions. In general, the best solution is obtained with devices having output enables.

A subsequent limitation on the multiplexed bus is the 8086's drive capability of 2.0 mA and capacitive loading of 100 pF to guarantee the specified A.C. characteristics. Assuming capacitive loads of 20 pF per I/O device, 12 pF per address latch and 5-12 pF per memory device, a system mix of three peripherals and two to four memory devices (per bus line) are close to the loading limit.

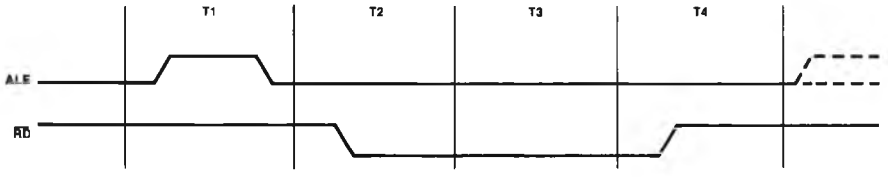


Diagram 2C1. Relationship of ALE to READ

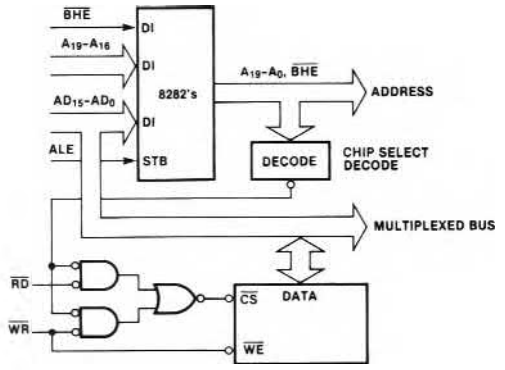
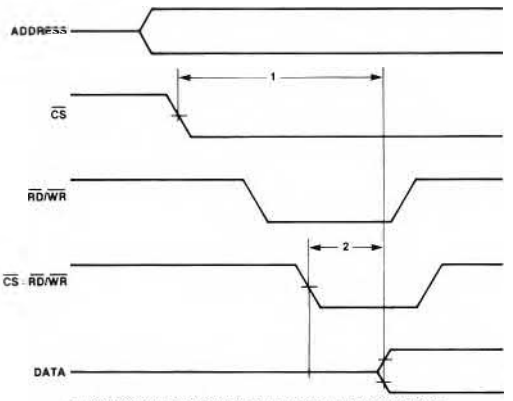


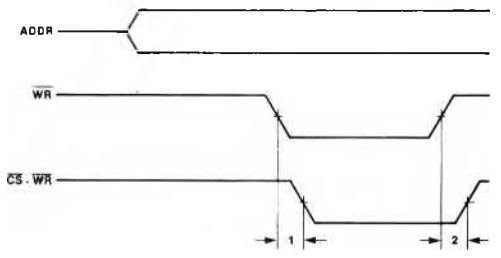
Figure 2C3. Devices without Output Enables on the Multiplexed Bus



1 ACCESS TIME FOR CS GENERATED FROM ADDRESS DECODE.

2 ACCESS TIME IF CS IS GATED WITH RD/WR.

Diagram 2C2. Access Time: CS Gated with RD/WR



1 CS IS NOT VALID PRIOR TO WRITE AND BECOMES ACTIVE ONE OR TWO GATE DELAYS LATER.

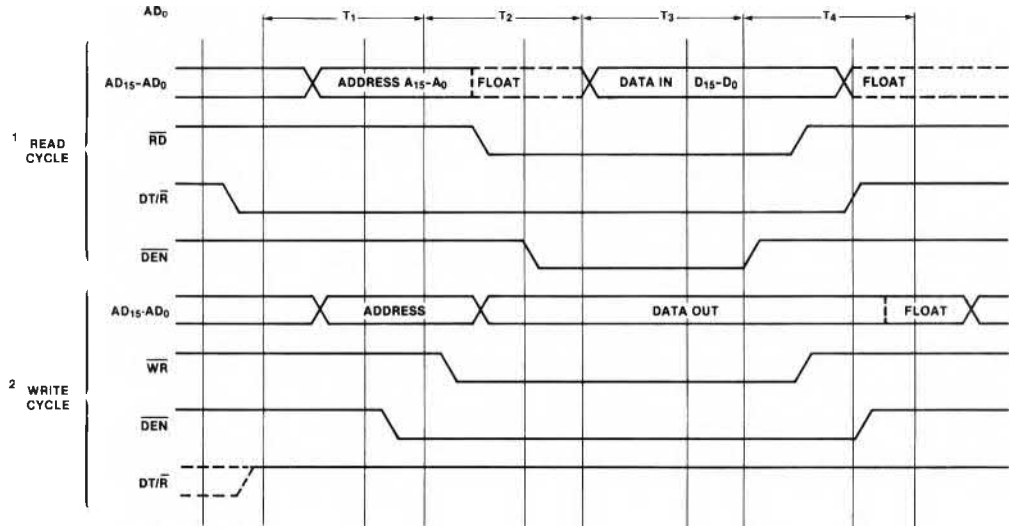
2 CS REMAINS VALID AFTER WRITE ONE OR TWO GATE DELAYS.

Diagram 2C3. CS to WR Set-Up and Hold

To satisfy the capacitive loading and drive requirements of larger systems, the data bus must be buffered. The 8286 non-inverting and 8287 inverting octal transceivers are offered as part of the 8086 family to satisfy this requirement. They have three-state output buffers that drive 32 mA on the bus interface and 10 mA on the CPU interface and can switch capacitive loads of 300 pF at the bus interface and 100 pF on the CPU interface in 22 ns (8287) or 30 ns (8286). To enable and control the direction of the transceivers, the 8086 system provides Data ENable (DEN) and Data Transmit/Receive (DT/$\overline{R}$) signals (Fig. 2C1b). These signals provide the appropriate timing to guarantee isolation of the multiplexed bus from the system during T1 and elimination of bus contention with the CPU during read and write (Diag. 2C4). Although the memory and peripheral devices are isolated from the CPU (Fig. 2C4), bus contention may still exist in the system if the devices do not have an output enable control other than chip select. As an example, bus contention will exist during transition from one chip select to another (the newly selected device begins driving the bus before the previous device has disabled its drivers). Another, more severe case exists during a write cycle. From chip select to write active, a device whose outputs are controlled only by chip select, will drive the bus simultaneously with write data being driven through the transceivers by the CPU (Diag. 2C5). The same technique given for circumventing these problems on the multiplexed bus can be applied here with the same limitations.

One last extension to the bus implementation is a second level of buffering to reduce the total load seen by devices on the system bus (Fig. 2C5). This is typically done for multiboard systems and isolation of memory arrays. The concerns with this configuration are the additional delay for access and more important, control of the second transceiver in relationship to the system bus and the device being interfaced to the system bus. Several techniques for controlling the transceiver are given in Figure 2C6. This first technique (Fig. 2C6a) simply distributes DEN and DT/$\overline{R}$ throughout the system. DT/$\overline{R}$ is inverted to provide proper direction control for the second level transceivers. The second example (Fig. 2C6b) provides control for devices with output enables. $\overline{RD}$ is used to normally direct data from the system bus to the peripheral. The buffer is selected whenever a device on the local bus is chip selected. Bus contention is possible on the device's local bus during a read as the read simultaneously enables the device output and changes the transceiver direction. The contention may also occur as the read is terminated.

For devices without output enables, the same technique can be applied (Fig. 2C6c) if the chip select to the device is conditioned by read or write. Controlling the chip select with read/write prevents the device from driving against the transceiver prior to the command being received. The limitations with this technique are access limited to read/write time and limited CS to write setup and hold times.



1 $\overline{DEN}$ IS ENABLED AFTER THE 8086 HAS FLOATED THE MULTIPLEXED BUS

2 DEN ENABLES THE TRANSCEIVERS EARLY IN THE CYCLE, BUT DT/R GUARANTEES THE TRANSCEIVERS ARE IN TRANSMIT RATHER THAN RECEIVE MODE AND WILL NOT DRIVE AGAINST THE CPU.

Diagram 2C4. Bus Transceiver Control

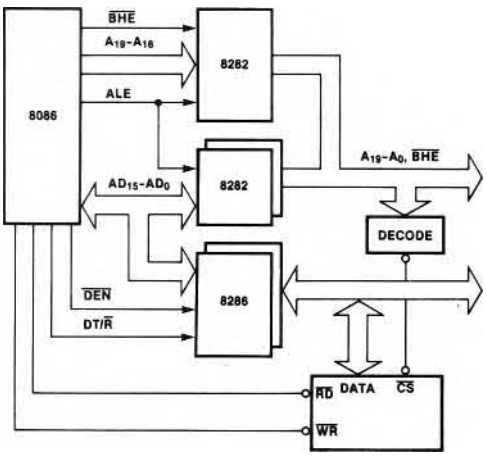


Figure 2C4. Devices with Output Enables on the System Bus

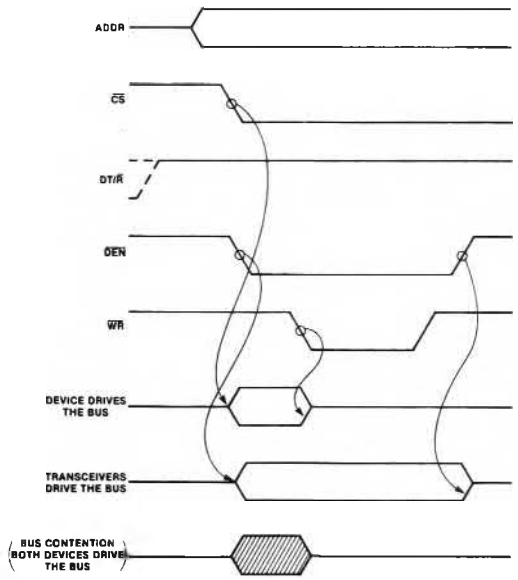


Diagram 2C5.

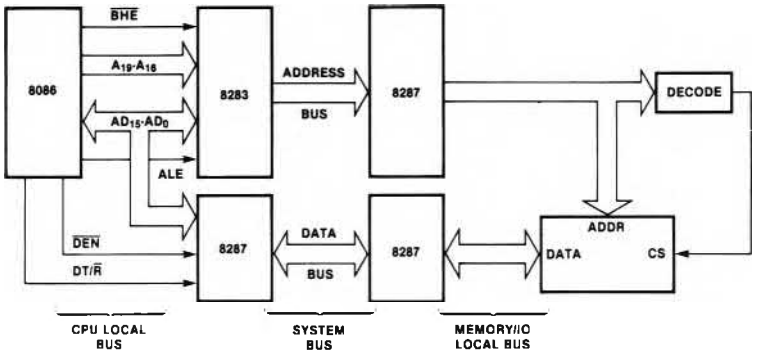


Figure 2C5. Fully Buffered System

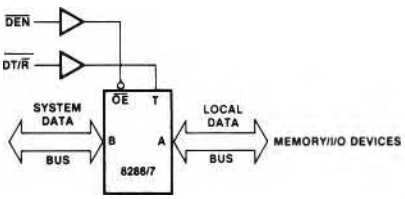


Figure 2C6a. Controlling System Transceivers with DEN and DT/R

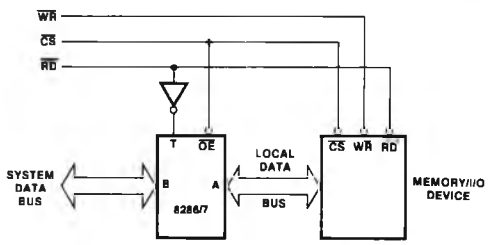


Figure 2C6b. Buffering Devices with OE/RD

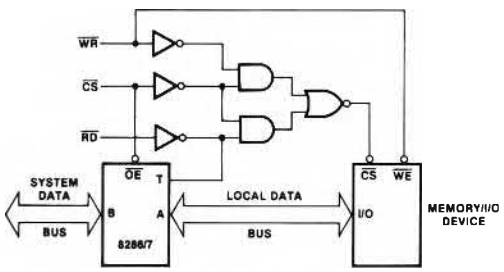


Figure 2C6c. Buffering Devices without OE/RD and with Common or Separate Input/Output

An alternate technique applicable to devices with and without output enables is shown in Figure 2C6d. RD again controls the direction of the transceiver but it is not enabled until a command and chip select are active. The possibility for bus contention still exists but is reduced to variations in output enable vs. direction change time for the transceiver. Full access time from chip select is now available, but data will not be valid prior to write and will only be held valid after write by the delay to disable the transceiver.

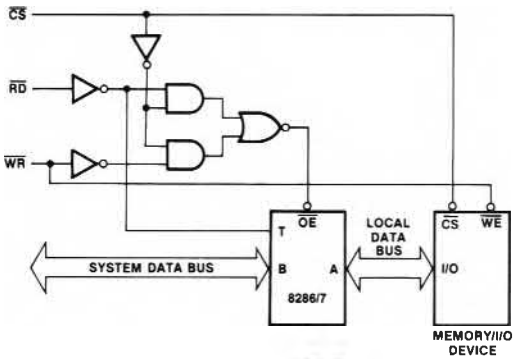


Figure 2C6d. Buffering Devices without OE/RD and with Common or Separate Input/Output

One last technique is given for devices with separate inputs and outputs (Fig. 2C6e). Separate bus receivers and drivers are provided rather than a single transceiver. The receiver is always enabled while the bus driver is controlled by RD and chip select. The only possibility for bus contention in this system occurs as multiple devices on each line of the local read bus are enabled and disabled during chip selection changes.

Throughout this note, the multiplexed bus will be considered the local CPU bus and the demultiplexed address and buffered data bus will be the system bus. For additional information on bus contention and the system problems associated with it, refer to Appendix 1.

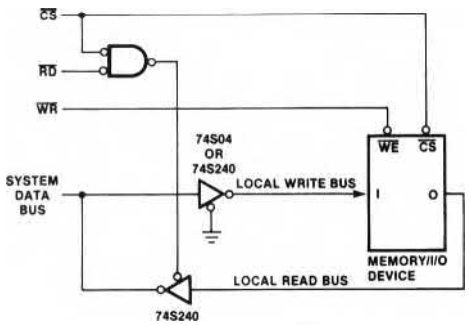


Figure 2C6e. Buffering Devices without $\overline{OE}/\overline{RD}$ and with Separate Input/Output

2D. Multiprocessor Environment

The 8086 architecture supports multiprocessor systems based on the concept of a shared system bus (Fig. 2D1). All CPU's in the system communicate with each other and share resources via the system bus. The bus may be either the Intel Multibus™ system bus or an extension of the system bus defined in the previous section. The major addition required to the demultiplexed system bus is arbitration logic to control access to the system bus. As each CPU asynchronously requests access to the shared bus, the arbitration logic resolves priorities and grants bus access to the highest priority CPU. Having gained access to the bus, the CPU completes its transfer and will either relinquish the bus or wait to be forced to relinquish the bus. For a discussion on Multibus™ arbitration techniques, refer to AP-28A, Intel Multibus™ Interfacing.

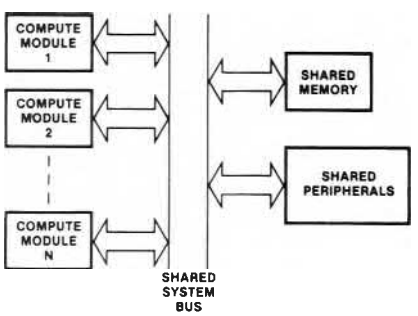


Figure 2D1. 8086 Family Multiprocessor System

To support a multimaster interface to the Multibus system bus for the 8086 family, the 8289 bus arbiter is included as part of the family. The 8289 is compatible with the 8086's local bus and in conjunction with the 8288 bus controller, implements the Multibus protocol for bus arbitration. The 8289 provides a variety of arbitration and prioritization techniques to allow optimization of bus availability, throughput and utilization of shared resources. Additional features (implemented through strapping options) extend the configuration options beyond a pure CPU interface to the multimaster system bus for access to shared resources to include concurrent support of a local CPU bus for private resources. For specific configurations and additional information on the 8289, refer to application note AP-51.

3. 8086 SYSTEM DETAILS

3A. Operating Modes

Possibly the most unique feature of the 8086 is the ability to select the base machine configuration most suited to the application. The MN/$\overline{MX}$ input to the 8086 is a strapping option which allows the designer to select between two functional definitions of a subset of the 8086 outputs.

MINIMUM MODE

The minimum mode 8086 (Fig. 3A1) is optimized for small to medium (one or two boards), single CPU systems. Its system architecture is directed at satisfying the requirements of the lower to middle segment of high performance 16-bit applications. The CPU maintains the full megabyte memory space, 64K byte I/O space and 16-bit data path. The CPU directly provides all bus control (DT/$\overline{R}$, $\overline{DEN}$, ALE, M/$\overline{IO}$), commands ($\overline{RD}$,$\overline{WR}$,$\overline{INTA}$) and a simple CPU preemption mechanism (HOLD, HLDA) compatible with existing DMA controllers.

MAXIMUM MODE

The maximum mode (Fig. 3A2) extends the system architecture to support multiprocessor configurations, and local instruction set extension processors (co-processors). Through addition of the 8288 bipolar bus controller, the 8086 outputs assigned to bus control and commands in the minimum mode are redefined to allow these extensions and enhance general system performance. Specifically, (1) two prioritized levels of processor preemption ($\overline{RQ}/\overline{GT0}$, $\overline{RQ}/\overline{GT1}$) allow multiple processors to reside on the 8086's local bus and share its interface to the system bus, (2) Queue status (QS0,QS1) is available to allow external devices like ICE™-86 or special instruction set extension co-processors to track the CPU instruction execution, (3) access control to shared resources in multiprocessor systems is supported by a hardware bus lock mechanism and (4) system command and configuration options are expanded via ancillary devices like the 8288 bus controller and 8289 bus arbiter.

The queue status indicates what information is being removed from the internal queue and when the queue is being reset due to a transfer of control (Table 3A1). By monitoring the $\overline{S0}$,$\overline{S1}$,$\overline{S2}$ status lines for instructions entering the 8086 (1,0,0 indicates code access while A0 and $\overline{BHE}$ indicate word or byte) and QS0, QS1 for instructions leaving the 8086's internal queue, it is possible to track the instruction execution. Since instructions are executed from the 8086's internal queue, the queue status is presented each CPU clock cycle and is not related to the bus cycle activity. This mechanism (1) allows a co-processor to detect execution of an

ecution of a specific memory location. An example of a circuit used by ICE is given in Figure 3A3. The first up down counter tracks the depth of the queue while the second captures the queue depth on a match. The second counter decrements on further fetches from the queue until the queue is flushed or the count goes to zero indicating execution of the match address. The first counter decrements on fetch from the queue (QS0 = 1) and increments on code fetches into the to the counter (T201 and T301) unless a single byte is loaded over the upper half of the bus (A0-P is high). Since the execution unit (EU) is not synchronized to the bus interface unit (BIU), a fetch from the queue can occur simultaneously with a transfer into the queue. The exclusive-or gate driving the ENP input of the first counter allows these simultaneous operations to cancel each other and not modify the queue depth.

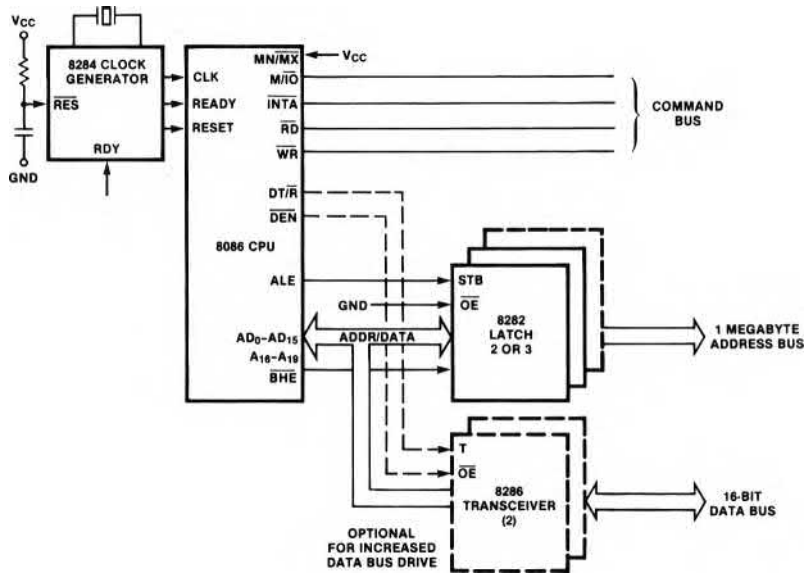


Figure 3A1. Minimum Mode 8086

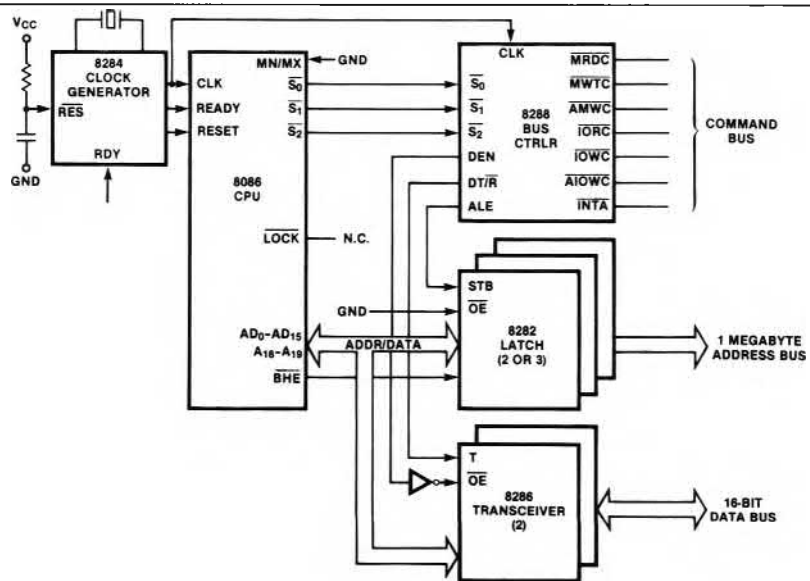


Figure 3A2. Maximum Mode 8086

TABLE 3A1. QUEUE STATUS

QS_1	QS_0	
0 (LOW)	0	No Operation
0	1	First Byte of Op Code from Queue
1 (HIGH)	0	Empty the Queue
1	1	Subsequent Byte from Queue

The queue status is valid during the CLK cycle after which the queue operation is performed.

To address the problem of controlling access to shared resources, the maximum mode 8086 provides a hardware $\overline{LOCK}$ output. The $\overline{LOCK}$ output is activated through the instruction stream by execution of the LOCK prefix instruction. The $\overline{LOCK}$ output goes active in the first CPU clock cycle following execution of the prefix and remains active until the clock following the completion of the instruction following the LOCK prefix. To provide bus access control in multiprocessor systems, the $\overline{LOCK}$ signal should be incorporated into the system bus arbitration logic resident to the CPU.

During normal multiprocessor system operation, priority of the shared system bus is determined by the arbitration circuitry on a cycle by cycle basis. As each CPU requires a transfer over the system bus, it requests access to the bus via its resident bus arbitration logic. When the CPU gains priority (determined by the system bus arbitration scheme and any associated logic), it takes control of the bus, performs its bus cycle and either maintains bus control, voluntarily releases the bus or is forced off the bus by the loss of priority. The lock mechanism prevents the CPU from losing bus control (either voluntarily or by force) and guarantees a CPU the ability to execute multiple bus cycles (during execution of the locked instruction) without intervention and possible corruption of the data by another CPU. A classic use of the mechanism is the 'TEST and SET semaphore' during which a CPU must read from a shared memory location and return data to the location without allowing another CPU to reference the same location between the TEST operation (read) and the SET operation (write). In the 8086 this is accomplished with a locked exchange instruction.

```
LOCK XCHG reg, MEMORY ; reg is any register
                      ;MEMORY is the address of the
                      ;semaphore
```

The activity of the $\overline{LOCK}$ output is shown in Diagram 3A1. Another interesting use of the $\overline{LOCK}$ for multiprocessor systems is a locked block move which allows high speed message transfer from one CPU's message buffer to another.

During the locked instruction, a request for processor preemption ($\overline{RQ}/\overline{GT}$) is recorded but not acknowledged until completion of the locked instruction. The $\overline{LOCK}$ has no direct affect on interrupts. As an example, a locked HALT instruction will cause HOLD (or $\overline{RQ}/\overline{GT}$) requests to be ignored but will allow the CPU to exit the HALT state on an interrupt. In general, prefix bytes are considered extensions of the instructions they precede. Therefore, interrupts that occur during execution of a prefix are not acknowledged (assuming interrupts are enabled) until completion of the instruction following the prefixes (except for instructions which allow servicing interrupts during their execution, i.e., HALT, WAIT and repeated string primitives). Note that multiple prefix bytes may precede an instruction. As another example, consider a 'string primitive' preceded by the repetition

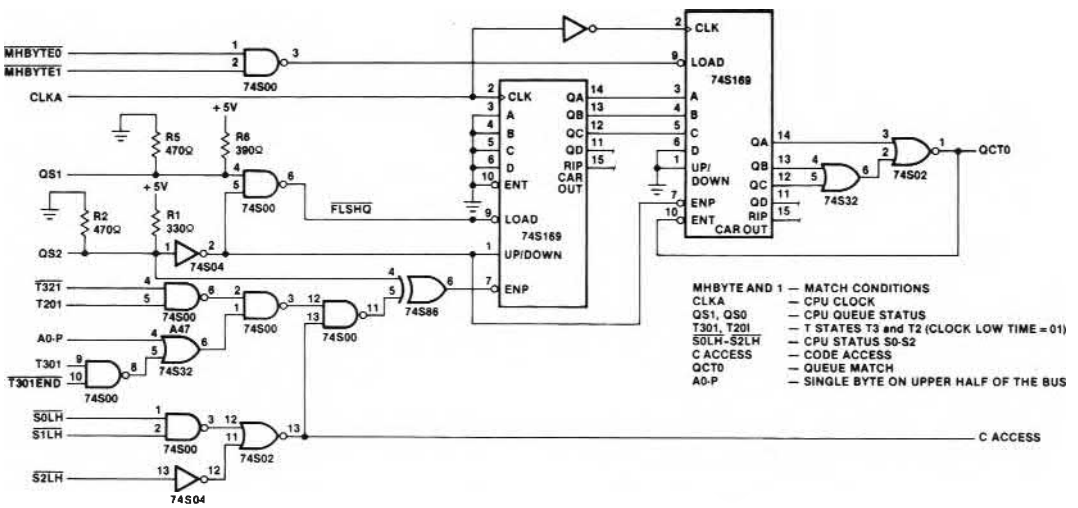


Figure 3A3. Example Circuit to Track the 8086 Queue

prefix (REP) which is interruptible after each execution of the string primitive. This holds even if the REP prefix is combined with the LOCK prefix and prevents interrupts from being locked out during a block move or other repeated string operation. As long as the operation is not interrupted, $\overline{LOCK}$ remains active. Further information on the operation of an interrupted string operation with multiple prefixes is presented in the section dealing with the 8086 interrupt structure.

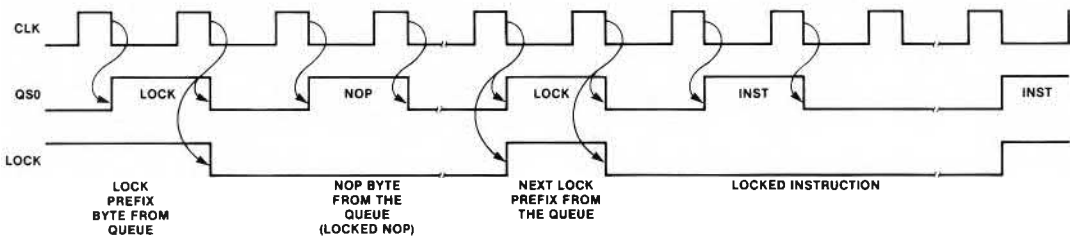
Three additional status lines ($\overline{S0}$, $\overline{S1}$, $\overline{S2}$) are defined to provide communications with the 8288 and 8289. The status lines tell the 8288 when to initiate a bus cycle, what type of command to issue and when to terminate the bus cycle. The 8288 samples the status lines at the beginning of each CPU clock (CLK). To initiate a bus cycle, the CPU drives the status lines from the passive state ($\overline{S0}$, $\overline{S1}$, $\overline{S2}$ = 1) to one of seven possible command codes (Table 3A2). This occurs on the rising edge of the clock during T4 of the previous bus cycle or a TI (idle cycle, no current bus activity). The 8288 detects the status change by sampling the status lines on the high to low transition of each clock cycle. The 8288 starts a bus cycle by generating ALE and appropriate buffer direction control in the clock cycle immediately following detection of the status change (T1). The bus transceivers and the selected command are enabled in the next clock cycle (T2) (or T3 for normal write commands). When the status returns to the passive state, the 8288 will terminate the command as shown in Diagram 3A2. Since the CPU will not return the status to the passive state until the 'ready' indication is received, the 8288 will maintain active command and bus control for any number of wait cycles. The status lines may also be used by other processors on the 8086's local bus to monitor bus activity and control the 8288 if they gain control of the local bus.

TABLE 3A2. STATUS LINE DECODES

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1 (HIGH)	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

The 8288 provides the bus control (DEN, DT/$\overline{R}$, ALE) and commands ($\overline{INTA}$, $\overline{MRDC}$, $\overline{IORC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IOWC}$, $\overline{AIOWC}$) removed from the CPU. The command structure has separate read and write commands for memory and I/O to provide compatibility with the Multibus command structure.

The advanced write commands are enabled one clock period earlier than the normal write to accommodate the wider write pulse widths often required by peripherals and static RAMs. The normal write provides data setup prior to write to accommodate dynamic RAM memories and I/O devices which strobe data on the leading edge of write. The advanced write commands do not guarantee that data is valid prior to the leading edge of the command. The DEN signal in the maximum mode is inverted from the minimum mode to extend transceiver control by allowing logical conjunction of DEN with other signals. While not appearing to be a significant benefit in the basic maximum mode configuration, introduction of interrupt control and various system configurations will demonstrate the usefulness of qualifying DEN. Diagram 3A3 compares the timing of the minimum and maximum mode bus transfer commands. Although the



1 QUEUE STATUS INDICATES FIRST BYTE OF OPCODE FROM THE QUEUE.

2 THE $\overline{LOCK}$ OUTPUT WILL GO INACTIVE BETWEEN SEPARATE LOCKED INSTRUCTIONS.

3 TWO CLOCKS ARE REQUIRED FOR DECODE OF THE LOCK PREFIX AND ACTIVATION OF THE $\overline{LOCK}$ SIGNAL.

4 SINCE QUEUE STATUS REFLECTS THE QUEUE OPERATION IN THE PREVIOUS CLOCK CYCLE, THE $\overline{LOCK}$ OUTPUT ACTUALLY GOES ACTIVE COINCIDENT WITH THE START OF THE NEXT INSTRUCTION AND REMAINS ACTIVE FOR ONE CLOCK CYCLE FOLLOWING THE INSTRUCTION.

5 IF THE INSTRUCTION FOLLOWING THE LOCK PREFIX IS NOT IN THE QUEUE, THE $\overline{LOCK}$ OUTPUT STILL GOES ACTIVE AS SHOWN WHILE THE INSTRUCTION IS BEING FETCHED.

6 THE BIU WILL STILL PERFORM INSTRUCTION FETCH CYCLES DURING EXECUTION OF A LOCKED INSTRUCTION. THE $\overline{LOCK}$ MERELY LOCKS THE BUS TO THIS CPU FOR WHATEVER BUS CYCLES THE CPU PERFORMS DURING THE LOCKED INSTRUCTION.

Diagram 3A1. 8086 Lock Activity

maximum mode configuration is designed for multiprocessor environments, large single CPU designs (either Multibus systems or greater than two PC boards) should also use the maximum mode. Since the 8288 is a bipolar dedicated controller device, its output drive for the commands (32 mA) and tolerances on AC characteristics (timing parameters and worse case delays) provide better large system performance than the minimum mode 8086.

In addition to assuming the functions removed from the CPU, the 8288 provides additional strapping options and controls to support multiprocessor configurations and peripheral devices on the CPU local bus. These capabilities allow assigning resources (memory or I/O) as shared (available on the Multibus system bus) or private (accessible only by this CPU) to reduce contention for access to the Multibus system bus and improve multi-CPU system performance. Specific configuration possibilities are discussed in AP-51.

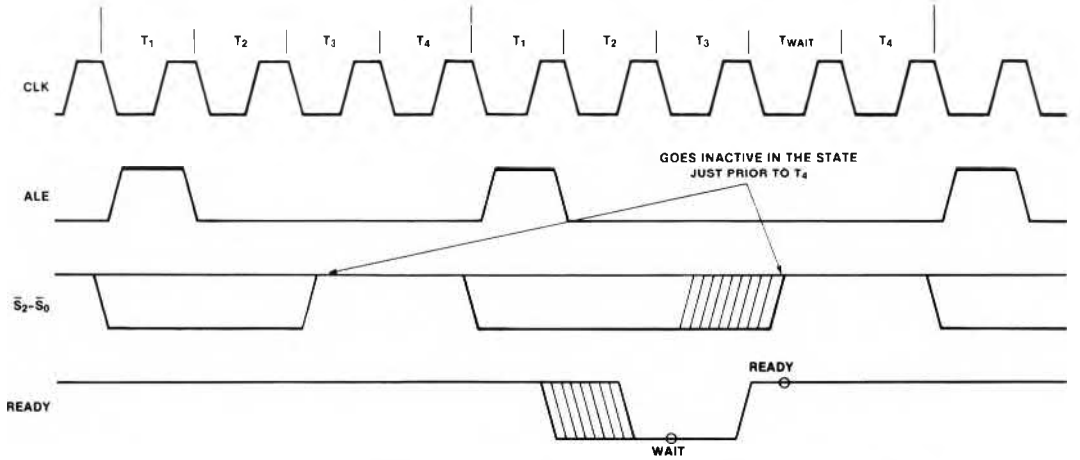


Diagram 3A2. Status Line Activation and Termination

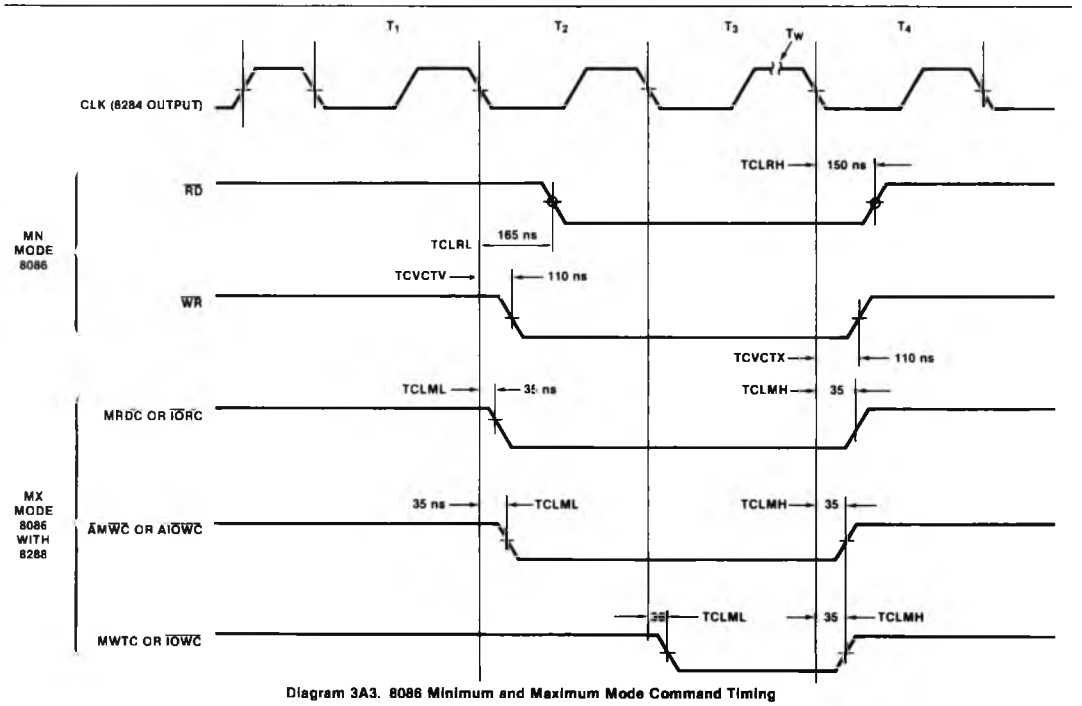


Diagram 3A3. 8086 Minimum and Maximum Mode Command Timing

3B. Clock Generation

The 8086 requires a clock signal with fast rise and fall times (10 ns max) between low and high voltages of −0.5 to +0.6 low and 3.9 to VCC+1.0 high. The maximum clock frequency of the 8086 is 5 MHz and 8 MHz for the 8086-2. Since the design of the 8086 incorporates dynamic cells, a minimum frequency of 2 MHz is required to retain the state of the machine. Due to the minimum frequency requirement, single stepping or cycling of the CPU may not be accomplished by disabling the clock. The timing and voltage requirements of the CPU clock are shown in Figure 3B1. In general, for frequencies below the maximum, the CPU clock need not satisfy the frequency dependent pulse width limitations stated in the 8086 data sheet. The values specified only reflect the minimum values which must be satisfied and are stated in terms of the maximum clock frequency. As the clock frequency approaches the maximum frequency of the CPU, the clock must conform to a 33% duty cycle to satisfy the CPU minimum clock low and high time specifications.

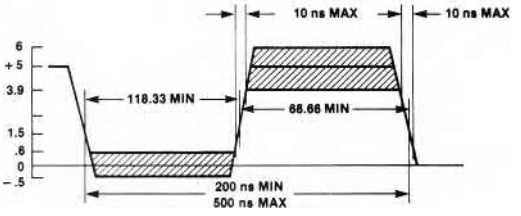


Figure 3B1. 8086 Clock

An optimum 33% duty cycle clock with the required voltage levels and transition times can be obtained with the 8284 clock generator (Fig. 3B2). Either an external frequency source or a series resonant crystal may drive the 8284. The selected source must oscillate at 3X the desired CPU frequency. To select the crystal inputs of the 8284 as the frequency source for clock generation, the F/$\overline{C}$ input to the 8284 must be strapped to ground. The strapping option allows selecting either the crystal or the external frequency input as the source for clock generation. Although the 8284 provides an input for a tank circuit to accommodate overtone mode crystals, fundamental mode crystals are recommended for more accurate and stable frequency generation. When selecting a crystal for use with the 8284, the series resistance should be as low as possible. Since other circuit components will tend to shift the operating frequency from resonance, the operating impedance will typically be higher than the specified series resistance. If the attenuation of the oscillator's feedback circuit reduces the loop gain to less than one, the oscillator will fail. Since the oscillator delays in the 8284 appear as inductive elements to the crystal, causing it to run at a frequency below that of the pure series resonance, a capacitor should be placed in series with the crystal and the X2 input of the 8284. This capacitor serves to cancel this inductive element. The value of the capacitor (CL) must not cause the impedance of the feedback circuit to reduce the loop gain below one. The impedance of the capacitor is a function of the operating frequency and can be determined from the following equation:

$$XCL = 1/2\pi * F * CL$$

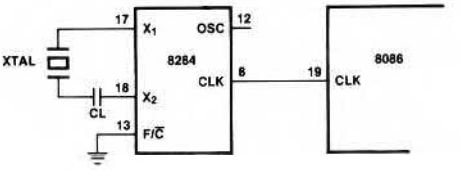


Figure 3B2. 8284 Clock Generator

It is recommended that the crystal series resistance plus XCL be kept less than 1K ohms. This capacitor also serves to debias the crystal and prevent a DC voltage bias from straining and perhaps damaging the crystalline structure. As the crystal frequency increases, the amount of capacitance should be decreased. For example, a 12 MHz crystal may require CL ~ 24 pF while 22 MHz may require CL ~ 8 pF. If very close correlation with the pure series resonance is not necessary, a nominal CL value of 12-15 pF may be used with a 15 MHz crystal (5 MHz 8086 operation). Board layout and component variances will affect the actual amount of inductance and therefore the series capacitance required to cancel it out (this is especially true for wire-wrapped layouts).

Two of the many vendors which supply crystals for Intel microprocessors are listed in Table 3B1 along with a list of crystal part numbers for various frequencies which may be of interest. For additional information on specifying crystals for Intel components refer to application note AP-35.

TABLE 3B1. CRYSTAL VENDORS

f	Parallel/ Series	Crystek[1] Corp.	CTS Knight,[2] Inc.
15.0 MHz	S	CY15A	MP150
18.432	S	CY19B*	MP184*
24.0 MHz	S	CY24A	MP240

*Intel also supplies a crystal numbered 8801 for this application.

Notes: 1. Address: 1000 Crystal Drive, Fort Meyers, Florida 33901
2. Address: 400 Reimann Ave., Sandwich, Illinois

If a high accuracy frequency source, externally variable frequency source or a common source for driving multiple 8284's is desired, the External Frequency Input (EFI) of the 8284 can be selected by strapping the F/$\overline{C}$ input to 5 volts through ~1K ohms (Fig. 3B3). The external frequency source should be TTL compatible, have a 50% duty cycle and oscillate at three times the desired CPU operating frequency. The maximum EFI frequency the 8284 can accept is slightly above 24 MHz with minimum clock low and high times of 13 ns. Although

no minimum EFI frequency is specified, it should not violate the CPU minimum clock rate. If a common frequency source is used to drive multiple 8284's distributed throughout the system, each 8284 should be driven by its own line from the source. To minimize noise in the system, each line should be a twisted pair driven by a buffer like the 74LS04 with the ground of the twisted pair connecting the grounds of the source and receiver. To minimize clock skew, the lines to all 8284's should be of equal length. A simple technique for generating a master frequency source for additional 8284's is shown in Figure 3B4. One 8284 with a crystal is used to generate the desired frequency. The oscillator output of the 8284 (OSC) equals the crystal frequency and is used to drive the external frequency to all other 8284's in the system.

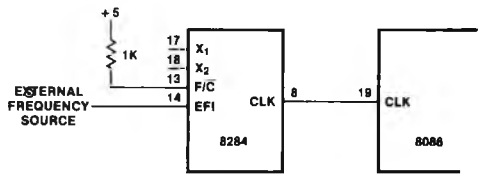


Figure 3B3. 8284 with External Frequency Source

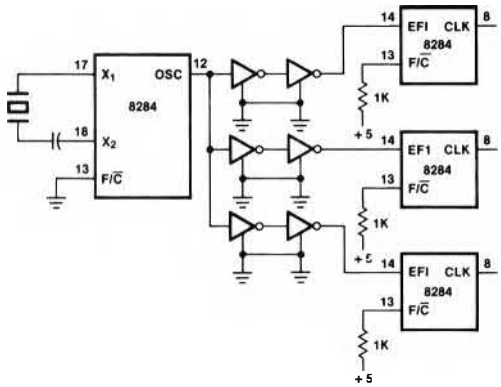


Figure 3B4. External Frequency for Multiple 8284s

The oscillator output is inverted from the oscillator signal used to drive the CPU clock generator circuit. Therefore, the oscillator output of one 8284 should not drive the EFI input of a second 8284 if both are driving clock inputs of separate CPU's that are to be synchronized. The variation on EFI to CLK delay over a range of 8284's may approach 35 to 45 ns. If, however, all 8284's are of the same package type, have the same relative supply voltage and operate in the same temperature environment, the variation will be reduced to between 15 and 25 ns.

There are three frequency outputs from the 8284, the oscillator (OSC) mentioned above, the system clock (CLK) which drives the CPU, and a peripheral clock (PCLK) that runs at one half the CPU clock frequency. The oscillator output is only driven by the crystal and is not affected by the F/$\overline{C}$ strapping option. If a crystal is not connected to the 8284 when the external frequency input is used, the oscillator output is indeterminate. The CPU clock is derived from the selected frequency source by an internal divide by three counter. The counter generates the 33% duty cycle clock which is optimum for the CPU at maximum frequency. The peripheral clock has a 50% duty cycle and is derived from the CPU clock. Diagram 3B0 shows the relationship of CLK to OSC and PCLK to CLK. The maximum skew is 20 ns between OSC and CLK, and 22 ns between CLK and PCLK.

Since the state of the 8284 divide by three counter is indeterminate at system initialization (power on), an external sync to the counter (CSYNC) is provided to allow synchronization of the CPU clock to an external event. When CSYNC is brought high, the CLK and PCLK outputs are forced high. When CSYNC returns low, the next positive clock from the frequency source starts clock generation. CSYNC must be active for a minimum of two periods of the frequency source. If CSYNC is asynchronous to the frequency source, the circuit in Figure 3B5 should be used for synchronization. The two latches minimize the probability of a meta-stable state in the latch driving CSYNC. The latches are clocked with the inverse of the frequency source to guarantee the 8284 setup and hold time of CSYNC to the frequency source (Diag. 3B1). If a single 8284 is to be synchronized to an external event and an external frequency source is not used, the oscillator output of the 8284 may be used to

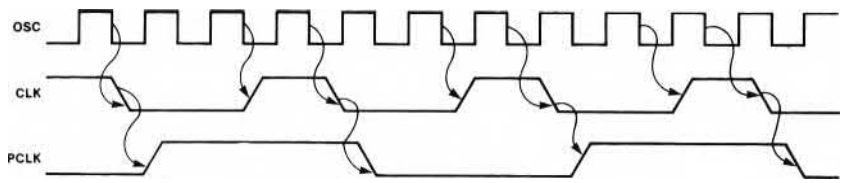


Diagram 3B0. OSC → CLK and CLK → PCLK Relationships

source must drive all 8284's and a single CSYNC synchronization circuit must drive the CSYNC input of all 8284's (Fig. 3B7). Since activation of CSYNC may cause violation of CPU minimum clock low time, it should only be enabled during reset or CPU clock high. CSYNC must also be disabled a minimum of four CPU clocks before the end of reset to guarantee proper CPU reset.

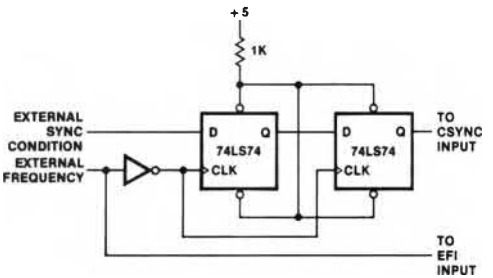


Figure 3B5. Synchronizing CSYNC with EFI

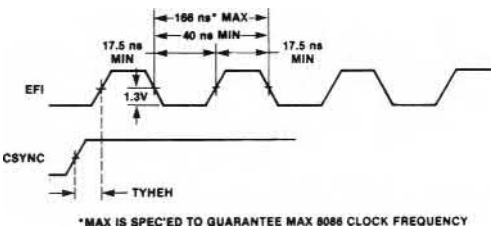


Diagram 3B1. CSYNC Setup and Hold to EFI

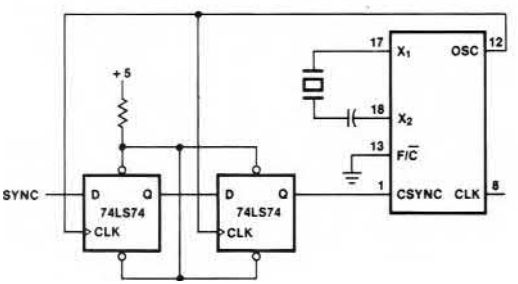


Figure 3B6. EFI from 8284 Oscillator

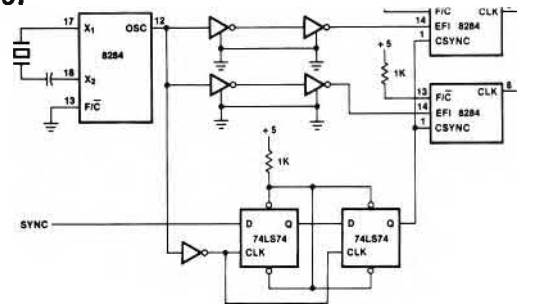


Figure 3B7. Synchronizing Multiple 8284s

Due to the fast transitions and high drive (5 mA) of the 8284 CLK output, it may be necessary to put a 10 to 100 ohm resistor in series with the clock line to eliminate ringing (resistor value depending on the amount of drive required). If multiple sources of CLK are needed with minimum skew, CLK can be buffered by a high drive device (74S241) with outputs tied to 5 volts through 100 ohms to guarantee VOH = 3.9 min (8086 minimum clock input high voltage) (Fig. 3B8). A single 8284 should not be used to generate the CLK for multiple CPU's that do not share a common local (multiplexed) bus since the 8284 synchronizes ready to the CPU and can only accommodate ready for a single CPU. If multiple CPU's share a local bus, they should be driven with the same clock to optimize transfer of bus control. Under these circumstances, only one CPU will be using the bus for a particular bus cycle which allows sharing a common READY signal (Fig. 3B9).

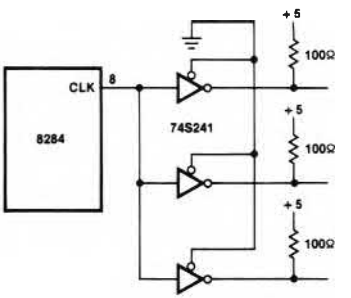


Figure 3B8. Buffering the 8284 CLK Output

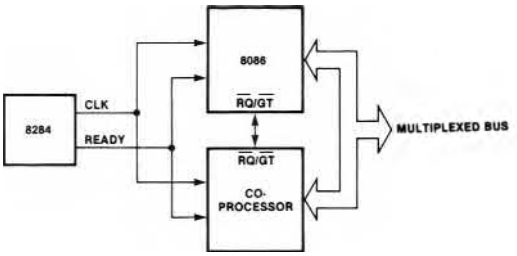


Figure 3B9. 8086 and Co-Processor on the Local Bus Share a Common 8284

3C. Reset

The 8086 requires a high active reset with minimum pulse width of four CPU clocks except after power on which requires a 50 μs reset pulse. Since the CPU internally synchronizes reset with the clock, the reset is internally active for up to one clock period after the external reset. Non-Maskable Interrupts (NMI) or hold requests on $\overline{RQ}/\overline{GT}$ which occur during the internal reset, are not acknowledged. A minimum mode hold request or maximum mode $\overline{RQ}$ pulses active immediately after the internal reset will be honored before the first instruction fetch.

From reset, the 8086 will condition the bus as shown in Table 3C1. The multiplexed bus will three-state upon detection of reset by the CPU. Other signals which three-state will be driven to the inactive state for one clock low interval prior to entering three-state (Fig. 3C1). In the minimum mode, ALE and HLDA are driven inactive and are not three-stated. In the maximum mode, $\overline{RQ}/\overline{GT}$ lines are held inactive and the queue status indicates no activity. The queue status will not indicate a reset of the queue so any user defined external circuits monitoring the queue should also be reset by the system reset. 22K ohm pull-up resistors should be connected to the CPU command and bus control lines to guarantee the inactive state of these lines in systems where leakage currents or bus capacitance may cause the voltage levels to settle below the minimum high voltage of devices in the system. In maximum mode systems, the 8288 contains internal pull-ups on the $\overline{S0}$-$\overline{S2}$ inputs to maintain the inactive state for these lines when the CPU floats the bus. The high state of the status lines during reset causes the 8288 to treat the reset sequence as a passive state. The condition of the 8288 outputs for the passive state are shown in Table 3C2. If the reset occurs during a bus cycle, the return of the status lines to the passive state will terminate the bus cycle and return the command lines to the inactive state. Note that the 8288 does not three-state the command outputs based on the passive state of the status lines. If the designer needs to three-state the CPU off the bus during reset in a single CPU system, the reset signal should also be connected to the 8288's $\overline{AEN}$ input and the output enable of the address latches (Fig. 3C2). This forces the command and address bus interface to three-state while the inactive state of DEN from the 8288 three-states the transceivers on the data bus.

Table 3C1. 8086 Bus During Reset

Signals	Condition
$AD_{15\text{-}0}$	Three-State
$A_{19\text{-}16}/S_{6\text{-}3}$	Three-State
BHE/S_7	Three-State
$\overline{S2}/(M/\overline{IO})$	Driven to "1" then three-state
$\overline{S1}/(DT/\overline{R})$	Driven to "1" then three-state
$\overline{S0}/\overline{DEN}$	Driven to "1" then three-state
$LOCK/\overline{WR}$	Driven to "1" then three-state
$\overline{RD}$	Driven to "1" then three-state
$\overline{INTA}$	Driven to "1" then three-state
ALE	0
HLDA	0
$\overline{RQ}/\overline{GT0}$	1
$\overline{RQ}/\overline{GT1}$	1
QS0	0
QS1	0

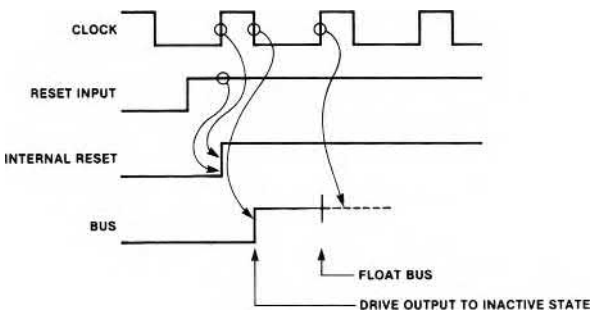


Figure 3C1. 8086 Bus Conditioning on Reset

TABLE 3C2. 8288 OUTPUTS DURING PASSIVE MODE

ALE	0
DEN	0
DT/$\overline{R}$	1
MCE/PDEN	0/1
COMMANDS	1

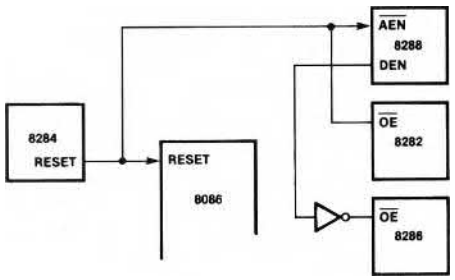


Figure 3C2. Reset Disable for Max Mode 8086 Bus Interface

For multiple processor systems using arbitration of a multimaster bus, the system reset should be connected to the $\overline{INIT}$ input of the 8289 bus arbiter in addition to the 8284 reset input (Fig. 3C3). The low active $\overline{INIT}$ input forces all 8289 outputs to their inactive state. The inactive state of the 8289 $\overline{AEN}$ output will force the 8288 to three-state the command outputs and the address latches to three-state the address bus interface. DEN inactive from the 8288 will three-state the data bus interface. For the multimaster CPU configuration, the reset should be common to all CPU's (8289's and 8284's) and satisfy the maximum of either the CPU reset requirements or 3 TBLBL (3 8289 bus clock times) + 3 TCLCL (3 8086 clock cycle times) to satisfy 8289 reset requirements.

If the 8288 command outputs are three-stated during reset, the command lines should be pulled up to V_{CC} through 2.2K ohm resistors.

The reset signal to the 8086 can be generated by the 8284. The 8284 has a schmitt trigger input ($\overline{RES}$) for generating reset from a low active external reset. The hysteresis specified in the 8284 data sheet implies that at least .25 volts will separate the 0 and 1 switching point of the 8284 reset input. Inputs without hysteresis will switch from low to high and high to low at approximately the same voltage threshold. The inputs are guaranteed to switch at specified low and high voltages (VIL and VIH) but the actual switching point is anywhere in-between. Since VIL min is specified at .8 volts, the hysteresis guarantees that the reset will be active until the input reaches at least 1.05 volts. A reset will not be recognized until the input drops at least .25 volts below the reset inputs VIH of 2.6 volts.

To guarantee reset from power up, the reset input must remain below 1.05 volts for 50 microseconds after V_{CC} has reached the minimum supply voltage of 4.5 volts. The hysteresis allows the reset input to be driven by a simple RC circuit as shown in Figure 3C4. The calculated RC value does not include time for the power supply to reach 4.5 volts or the charge accumulated during this interval. Without the hysteresis, the reset output might oscillate as the input voltage passes through the switching voltage of the input. The calculated RC value provides the minimum required reset period of 50 microseconds for 8284's that switch at the 1.05 volt level and a reset period of approximately 162 microseconds for 8284's that switch at the 2.6 volt level. If tighter tolerance between the minimum and maximum reset times is necessary, the reset circuit shown in Figure 3C5 might be used rather than the simple RC circuit. This circuit provides a constant current source and a linear charge rate on the capacitor rather than the inverse exponential charge rate of the RC circuit. The maximum reset period for this implementation is 124 microseconds.

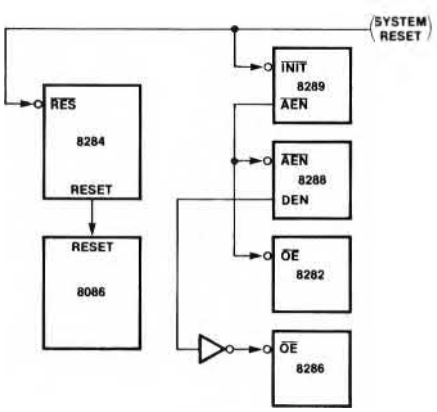


Figure 3C3. Reset Disable of for Max Mode 8086 Bus Interface in Multi CPU System

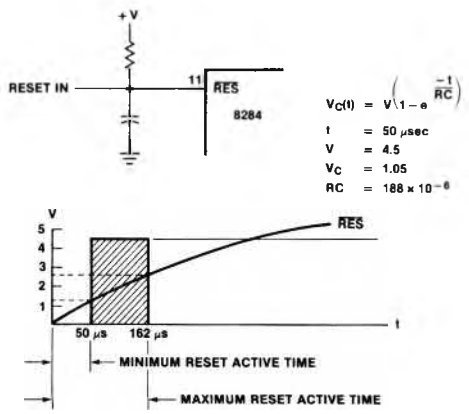


Figure 3C4. 8284 Reset Circuit

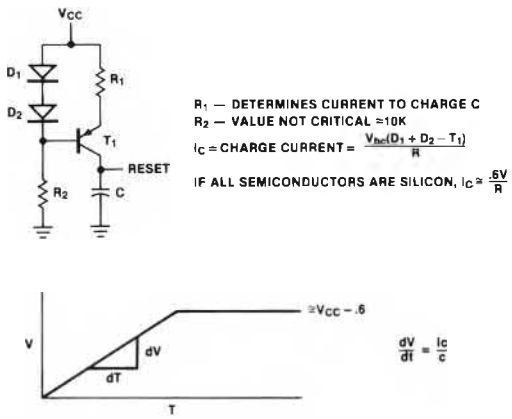


Figure 3C5. Constant Current Power-On Reset Circuit

The 8284 synchronizes the reset input with the CPU clock to generate the RESET signal to the CPU (Fig. 3C6). The output is also available as a general reset to the entire system. The reset has no effect on any clock circuits in the 8284.

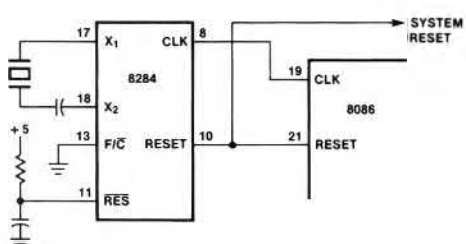


Figure 3C6. 8086 Reset and System Reset

3D. Ready Implementation and Timing

As discussed previously, the ready signal is used in the system to accommodate memory and I/O devices that cannot transfer information at the maximum CPU bus bandwidth. Ready is also used in multiprocessor systems to force the CPU to wait for access to the system bus or Multibus system bus. To insert a wait state in the bus cycle, the READY signal to the CPU must be inactive (low) by the end of T2. To avoid insertion of a wait state, READY must be active (high) within a specified setup time prior to the positive transition during T3. Depending on the size and characteristics of the system, ready implementation may take one of two approaches.

The classical ready implementation is to have the system 'normally not ready.' When the selected device receives the command ($\overline{RD}/\overline{WR}/\overline{INTA}$) and has had sufficient time to complete the command, it activates READY to the CPU, allowing the CPU to terminate the bus cycle. This implementation is characteristic of large multiprocessor, Multibus systems or systems where propagation delays, bus access delays and device characteristics inherently slow down the system. For maximum system performance, devices that can run with no wait states must return 'READY' within the previously described limit. Failure to respond in time will only result in the insertion of one or more wait cycles.

An alternate technique is to have the system 'normally ready.' All devices are assumed to operate at the maximum CPU bus bandwidth. Devices that do not meet the requirement must disable READY by the end of T2 to guarantee the insertion of wait cycles. This implementation is typically applied to small single CPU systems and reduces the logic required to control the ready signal. Since the failure of a device requiring wait states to disable READY by the end of T2 will result in premature termination of the bus cycle, the system timing must be carefully analyzed when using this approach.

The 8086 has two different timing requirements on READY depending on the system implementation. For a 'normally ready' system to insert a wait state, the READY must be disabled within 8 ns (TRYLCL) after the end of T2 (start of T3) (Diag. 3D1). To guarantee proper

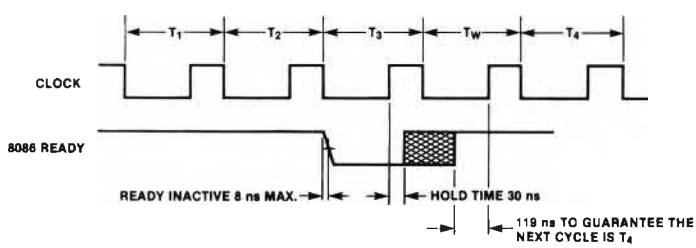


Diagram 3D1. Normally Ready System Inserting a Wait State

operation of the 8086, the READY input must not change from ready to not ready during the clock low time of T3. For a 'normally not ready' system to avoid wait states, READY must be active within 119 ns (TRYHCH) of the positive clock transition during T3 (Diag. 3D2). For both cases, READY must satisfy a hold time of 30 ns (TCHRYX) from the T3 or TW positive clock transition.

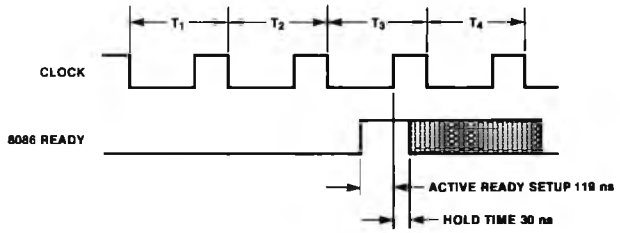


Diagram 3D2. Normally Not Ready System Avoiding a Wait State

To generate a stable READY signal which satisfies the previous setup and hold times, the 8284 provides two separate system ready inputs (RDY1, RDY2) and a single synchronized ready output (READY) for the CPU. The RDY inputs are qualified with separate access enables ($\overline{AEN1}$,$\overline{AEN2}$, low active) to allow selecting one of the two ready signals (Fig. 3D1). The gated signals are logically OR'ed and sampled at the beginning of each CLK cycle to generate READY to the CPU (Diag. 3D3). The sampled READY signal is valid within 8 ns (TRYLCL) after CLK to satisfy the CPU timing requirements on 'not ready' and ready. Since READY cannot change until the next CLK, the hold time requirements are also satisfied. The system ready inputs to the 8284 (RDY1,RDY2) must be valid 35 ns (TRIVCL) before T3 and $\overline{AEN}$ must be valid 60 ns before T3. For a system using only one RDY input, the associated $\overline{AEN}$ is tied to ground while the other $\overline{AEN}$ is connected to 5 volts through ~1K ohms (Fig. 3D2a). If the system generates a low active ready signal, it can be connected to the 8284 $\overline{AEN}$ input if the additional setup time required by the 8284 $\overline{AEN}$ input is satisfied. In this case, the associated RDY input would be tied high (Fig. 3D2b).

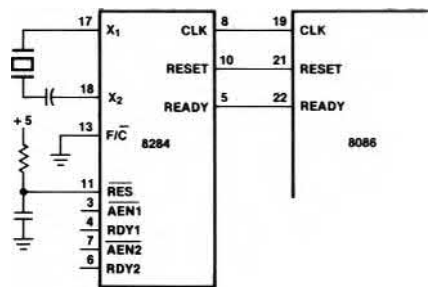
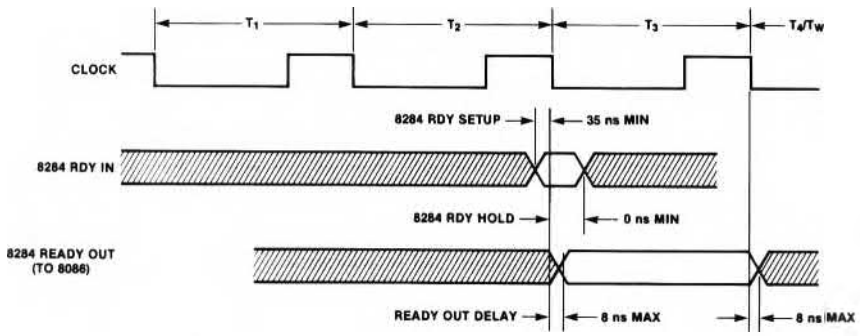


Figure 3D1. Ready Inputs to the 8284 and Output to the 8086



NOTE: THE 8284 DATA SHEET SPECIFIES READY OUT DELAY (TRYLCL) AS −8 ns 'BEFORE' THE END OF T_2 WHICH IMPLIES THE TIMING SHOWN.

Diagram 3D3. 8284 with 8086 Ready Timing

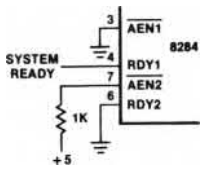


Figure 3D2a. Using RDY1/RDY2 to Generate Ready

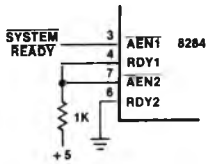


Figure 3D2b. Using $\overline{AEN1}$/$\overline{AEN2}$ to Generate Ready

The majority of memory and peripheral devices which fail to operate at the maximum CPU frequency typically do not require more than one wait state. The circuit given in Figure 3D3 is an example of a simple wait state generator. The system ready line is driven low whenever a device requiring one wait state is selected. The flip flop is cleared by ALE, enabling RDY to the 8284. If no wait states are required, the flip flop does not change. If the system ready is driven low, the flip flop toggles on the low to high clock transition of T2 to force one wait state. The next low to high clock transition toggles the flip flop again to indicate ready and allow completion of the bus cycle. Further changes in the state of the flip flop will not affect the bus cycle. The circuit allows approximately 100 ns for chip select decode and conditioning of the system ready (Diag. 3D4).

If the system is 'normally not ready,' the programmer should not assign executable code to the last six bytes of physical memory. Since the 8086 prefetches instructions, the CPU may attempt to access non-existent memory when executing code at the end of physical memory. If the access to non-existent memory fails to enable READY, the system will be caught in an indefinite wait.

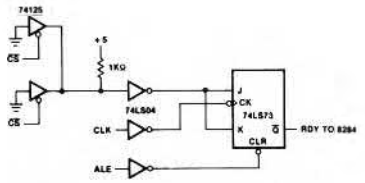


Figure 3D3. Single Wait State Generator

3E. Interrupt Structure

The 8086 interrupt structure is based on a table of interrupt vectors stored in memory locations 0H through 003FFH. Each vector consists of two bytes for the instruction pointer and two bytes for the code segment. These two values combine to form the address of the interrupt service routine. This allows the table to contain up to 256 interrupt vectors which specify the starting address of the service routines anywhere in the one megabyte address space of the 8086. If fewer than 256 different interrupts are defined in the system, the user need only allocate enough memory for the interrupt vector table to provide the vectors for the defined interrupts. During initial system debug, however, it may be desirable to assign all undefined interrupt types to a trap routine to detect erroneous interrupts.

Each vector is associated with an interrupt type number which points to the vector's location in the interrupt vector table. The interrupt type number multiplied by four gives the displacement of the first byte of the associated interrupt vector from the beginning of the table. As an example, interrupt type number 5 points to the sixth entry in the interrupt vector table. The contents of this entry in the table points to the interrupt service routine for type 5 (Fig. 3E1). This structure allows the user to specify the memory address of each service routine by placing the address (instruction pointer and code segment values) in the table location provided for that type interrupt.

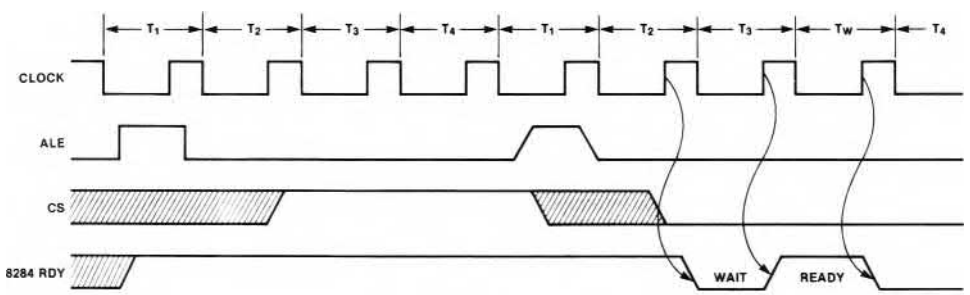


Diagram 3D4.

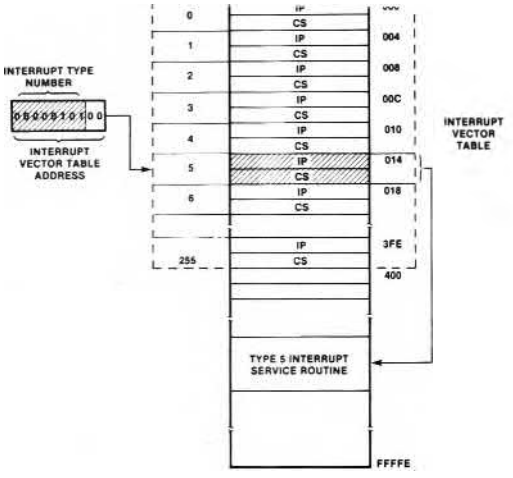


Figure 3E1. Direction to Interrupt Service Routine through the Interrupt Vector Table

All interrupts in the 8086 must be assigned an interrupt type which uniquely identifies each interrupt. There are three classes of interrupt types in the 8086; predefined interrupt types which are issued by specific functions within the 8086 and user defined hardware and software interrupts. Note that any interrupt type including the predefined interrupts can be issued by the user's hardware and/or software.

PREDEFINED INTERRUPTS

The predefined interrupt types in the 8086 are listed below with a brief description of how each is invoked. When invoked, the CPU will transfer control to the memory location specified by the vector associated with the specific type. The user must provide the interrupt service routine and initialize the interrupt vector table with the appropriate service routine address. The user may additionally invoke these interrupts through hardware or software. If the preassigned function is not used in the system, the user may assign some other function to the associated type. However, for compatibility with future Intel hardware and software products for the 8086 family, interrupt types 0-31 should not be assigned as user defined interrupts.

TYPE 0 — DIVIDE ERROR

This interrupt type is invoked whenever a division operation is attempted during which the quotient exceeds the maximum value (ex. division by zero). The interrupt is non-maskable and is entered as part of the execution of the divide instruction. If interrupts are not reenabled by the divide error interrupt service routine, the service routine execution time should be included in the worst case divide instruction execution time (primarily when considering the longest instruction execution time and its effect on latency to servicing hardware interrupts).

This interrupt type occurs one instruction (Trap Flag) is set in the flag register. It is used to allow software single stepping through a sequence of code. Single stepping is initiated by copying the flags onto the stack, setting the TF bit on the stack and popping the flags. The interrupt routine should be the single step routine. The interrupt sequence saves the flags and program counter, then resets the TF flag to allow the single step routine to execute normally. To return to the routine under test, an interrupt return restores the IP, CS and flags with TF set. This allows the execution of the next instruction in the program under test before trapping back to the single step routine. Single Step is not masked by the IF (Interrupt Flag) bit in the flag register.

TYPE 2 — NMI (Non-Maskable Interrupt)

This is the highest priority hardware interrupt and is non-maskable. The input is edge triggered but is synchronized with the CPU clock and must be active for two clock cycles to guarantee recognition. The interrupt signal may be removed prior to entry to the service routine. Since the input must make a low to high transition to generate an interrupt, spurious transitions on the input should be suppressed. If the input is normally high, the NMI low time to guarantee triggering is two CPU clock times. This input is typically reserved for catastrophic failures like power failure or timeout of a system watchdog timer.

TYPE 3 — ONE BYTE INTERRUPT

This is invoked by a special form of the software interrupt instruction which requires a single byte of code space. Its primary use is as a breakpoint interrupt for software debug. With full representation within a single byte, the instruction can map into the smallest instruction for absolute resolution in setting breakpoints. The interrupt is not maskable.

TYPE 4 — INTERRUPT ON OVERFLOW

This interrupt occurs if the overflow flag (OF) is set in the flag register and the INTO instruction is executed. The instruction allows trapping to an overflow error service routine. The interrupt is non-maskable.

Interrupt types 0 and 2 can occur without specific action by the programmer (except for performing a divide for Type 0) while types 1, 3, and 4 require a conscious act by the programmer to generate these interrupt types. All but type 2 are invoked through software activity and are directly associated with a specific instruction.

USER DEFINED SOFTWARE INTERRUPTS

The user can generate an interrupt through the software with a two byte interrupt instruction INT nn. The first byte is the INT opcode while the second byte (nn) contains the type number of the interrupt to be performed. The INT instruction is not maskable by the interrupt enable flag. This instruction can be used to transfer control to routines that are dynamically relocatable and whose location in memory is not known by the calling

program. This technique also saves the flags of the calling program on the stack prior to transferring control. The called procedure must return control with an interrupt return (IRET) instruction to remove the flags from the stack and fully restore the state of the calling program.

All interrupts invoked through software (all interrupts discussed thus far with the exception of NMI) are not maskable with the IF flag and initiate the transfer of control at the end of the instruction in which they occur. They do not initiate interrupt acknowledge bus cycles and will disable subsequent maskable interrupts by resetting the IF and TF flags. The interrupt vector for these interrupt types is either implied or specified in the instruction. Since the NMI is an asynchronous event to the CPU, the point of recognition and initiation of the transfer of control is similar to the maskable hardware interrupts.

USER DEFINED HARDWARE INTERRUPTS

The maskable interrupts initiated by the system hardware are activated through the INTR pin of the 8086 and are masked by the IF bit of the status register (interrupt flag). During the last clock cycle of each instruction, the state of the INTR pin is sampled. The 8086 deviates from this rule when the instruction is a MOV or POP to a segment register. For this case, the interrupts are not sampled until completion of the following instruction. This allows a 32-bit pointer to be loaded to the stack pointer registers SS and SP without the danger of an interrupt occurring between the two loads. Another exception is the WAIT instruction which waits for a low active input on the $\overline{\text{TEST}}$ pin. This instruction also continuously samples the interrupt request during its execution and allows servicing interrupts during the wait. When an interrupt is detected, the WAIT instruction is again fetched prior to servicing the interrupt to guarantee the interrupt routine will return to the WAIT instruction.

UNINTERRUPTABLE INSTRUCTION SEQUENCE

```
MOV SS, NEW$STACK$SEGMENT
MOV SP, NEW$STACK$POINTER
```

Also, since prefixes are considered part of the instruction they precede, the 8086 will not sample the interrupt line until completion of the instruction the prefix(es) precede(s). An exception to this (other than HALT or WAIT) is the string primatives preceded by the repeat (REP) prefix. The repeated string operations will sample the interrupt line at the completion of each repetition. This includes repeat string operations which include the lock prefix. If multiple prefixes precede a repeated string operation, and the instruction is interrupted, only the prefix immediately preceding the string primative is restored. To allow correct resumption of the operation, the following programming technique may be used:

```
LOCKED$BLOCK$MOVE:  LOCK REP MOVS DEST, CS:SOURCE
                    AND CX,    CX
                    JNZ LOCKED$BLOCK$MOVE
```

The code bytes generated by the 8086 assembler for the MOVS instruction are (in descending order): LOCK prefix, REP prefix, Segment Override prefix and MOVS. Upon return from the interrupt, the segment override prefix is restored to guarantee one additional transfer is performed between the correct memory locations. The instructions following the move operation test the repetition count value to determine if the move was completed and return if not.

If the INTR pin is high when sampled and the IF bit is set to enable interrupts, the 8086 executes an interrupt acknowledge sequence. To guarantee the interrupt will be acknowledged, the INTR input must be held active until the interrupt acknowledge is issued by the CPU. If the BIU is running a bus cycle when the interrupt condition is detected (as would occur if the BIU is fetching an instruction when the current instruction completes), the

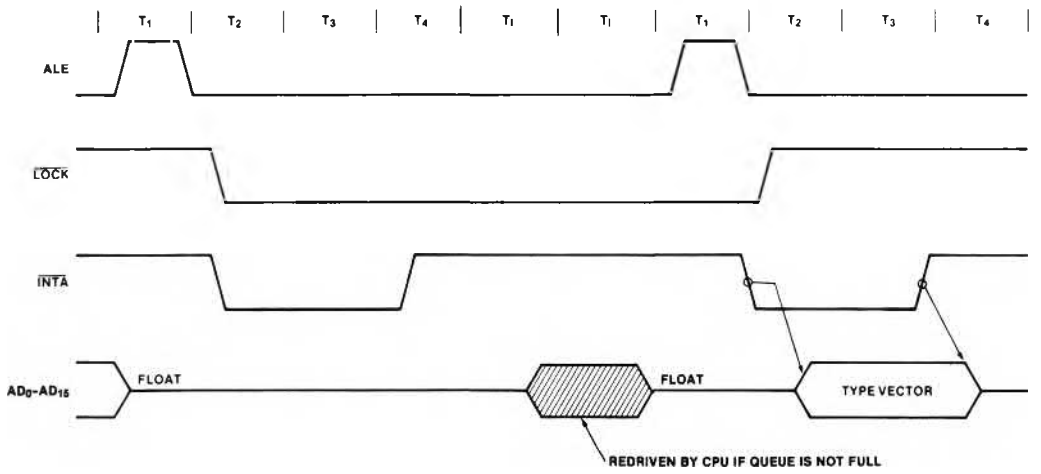


Figure 3E2. Interrupt Acknowledge Sequence

interrupt must be valid at the 8086 2 clock cycles prior to T4 of the bus cycle if the next cycle is to be an interrupt acknowledge cycle. If the 2 clock setup is not satisfied, another pending bus cycle will be executed before the interrupt acknowledge is issued. If a hold request is also pending (this might occur if an interrupt and hold request are made during execution of a locked instruction), the interrupt is serviced after the hold request is serviced.

The interrupt acknowledge sequence is only generated in response to an interrupt on the 8086 INTR input. The associated bus activity is shown in Figure 3E2. The cycle consists of two $\overline{\text{INTA}}$ bus cycles separated by two idle clock cycles. During the bus cycles the $\overline{\text{INTA}}$ command is issued rather than read. No address is provided by the 8086 during either bus cycle ($\overline{\text{BHE}}$ and status are valid), however, ALE is still generated and will load the address latches with indeterminate information. This condition requires that devices in the system do not drive their outputs without being qualified by the Read Command. As will be shown later, the ALE is useful in maximum mode systems with multiple 8259A priority interrupt controllers. During the $\overline{\text{INTA}}$ bus cycles, DT/$\overline{\text{R}}$ and DEN are conditioned to allow the 8086 to receive a one byte interrupt type number from the interrupt system. The first $\overline{\text{INTA}}$ bus cycle signals an interrupt acknowledge cycle is in progress and allows the system to prepare to present the interrupt type number on the next $\overline{\text{INTA}}$ bus cycle. The CPU does not capture information on the bus during the first cycle. The type number must be transferred to the 8086 on the lower half of the 16-bit data bus during the second cycle. This implies that devices which present interrupt type numbers to the 8086 must be located on the lower half of the 16-bit data bus. The timing of the $\overline{\text{INTA}}$ bus cycles (with exception of address timing) is similar to read cycle timing. The 8086 interrupt acknowledge sequence deviates from the form used on 8080 and 8085 in that no instruction is issued as part of the sequence. The 8080 and 8085 required either a restart or call instruction be issued to affect the transfer of control.

In the minimum mode system, the M/$\overline{\text{IO}}$ signal will be low indicating I/O during the $\overline{\text{INTA}}$ bus cycles. The 8086 internal $\overline{\text{LOCK}}$ signal will be active from T2 of the first bus cycle until T2 of the second to prevent the BIU from honoring a hold request between the two $\overline{\text{INTA}}$ cycles.

In the maximum mode, the status lines $\overline{\text{S0}}$-$\overline{\text{S2}}$ will request the 8288 to activate the $\overline{\text{INTA}}$ output for each cycle. The $\overline{\text{LOCK}}$ output of the 8086 will be active from T2 of the first cycle until T2 of the second to prevent the 8086 from honoring a hold request on either $\overline{\text{RQ}}$/$\overline{\text{GT}}$ input and to prevent bus arbitration logic from relinquishing the bus between $\overline{\text{INTA}}$'s in multi-master systems. The consequences of READY are identical to those for READ and WRITE cycles.

Once the 8086 has the interrupt type number (from the bus for hardware interrupts, from the instruction stream for software interrupts or from the predefined condition), the type number is multiplied by four to form the displacement to the corresponding interrupt vector in the interrupt vector table. The four bytes of the interrupt vector are: least significant byte of the instruction pointer, most significant byte of the instruction pointer, least significant byte of the code segment register, most significant byte of the code segment register. During the transfer of control, the CPU pushes the flags and current code segment register and instruction pointer onto the stack. The new code segment and instruction pointer values are loaded and the single step and interrupt flags are reset. Resetting the interrupt flag disables response to further hardware interrupts in the service routine unless the flags are specifically re-enabled by the service routine. The CS and IP values are read from the interrupt vector table with data read cycles. No segment registers are used when referencing the vector table during the interrupt context switch. The vector displacement is added to zero to form the 20-bit address and S4, S3 = 10 indicating no segment register selection.

The actual bus activity associated with the hardware interrupt acknowledge sequence is as follows: Two interrupt acknowledge bus cycles, read new IP from the interrupt vector table, read new CS from the interrupt vector table, Push flags, Push old CS, Opcode fetch of the first instruction of the interrupt service routine, and Push old IP. After saving the old IP, the BIU will resume normal operation of prefetching instructions into the queue and servicing EU requests for operands. S5 (interrupt enable flag status) will go inactive in the second clock cycle following reading the new CS.

The number of clock cycles from the end of the instruction during which the interrupt occurred to the start of interrupt routine execution is 61 clock cycles. For software generated interrupts, the sequence of bus cycles is the same except no interrupt acknowledge bus cycles are executed. This reduces the delay to service routine execution to 51 clocks for INT nn and single step, 52 clocks for INT3 and 53 clocks for INTO. The same interrupt setup requirements with respect to the BIU that were stated for the hardware interrupts also apply to the software interrupts. If wait states are inserted by either the memories or the device supplying the interrupt type number, the given clock times will increase accordingly.

When considering the precedence of interrupts for multiple simultaneous interrupts, the following guidelines apply: 1. INTR is the only maskable interrupt and if detected simultaneously with other interrupts, resetting of IF by the other interrupts will mask INTR. This causes INTR to be the lowest priority interrupt serviced after all other interrupts unless the other interrupt service routines reenable interrupts. 2. Of the nonmaskable interrupts (NMI, Single Step and software generated), in general, Single Step has highest priority (will be serviced first) followed by NMI, followed by the software interrupts. This implies that a simultaneous NMI and Single Step trap will cause the NMI service routine to follow single step; a simultaneous software trap and Single Step trap will cause the software interrupt service routine to follow single step and a simultaneous NMI and software trap will cause the NMI service routine to be executed followed by the software interrupt service routine. An exception to this priority structure occurs if all three interrupts are pending. For this case, transfer of control to the software interrupt ser-

vice routine followed by the NMI trap will cause both the NMI and software interrupt service routines to be executed without single stepping. Single stepping resumes upon execution of the instruction following the instruction causing the software interrupt (the next instruction in the routine being single stepped).

If the user does not wish to single step before INTR service routines, the single step routine need only disable interrupts during execution of the program being single stepped and reenable interrupts on entry to the single step routine. Disabling the interrupts during the program under test prevents entry into the interrupt service routine while single step (TF = 1) is active. To prevent single stepping before NMI service routines, the single step routine must check the return address on the stack for the NMI service routine address and return control to that routine without single step enabled. As examples, consider Figures 3E3a and 3E3b. In 3E3a Single Step and NMI occur simultaneously while in 3E3b, NMI, INTR and a divide error all occur during a divide instruction being single stepped.

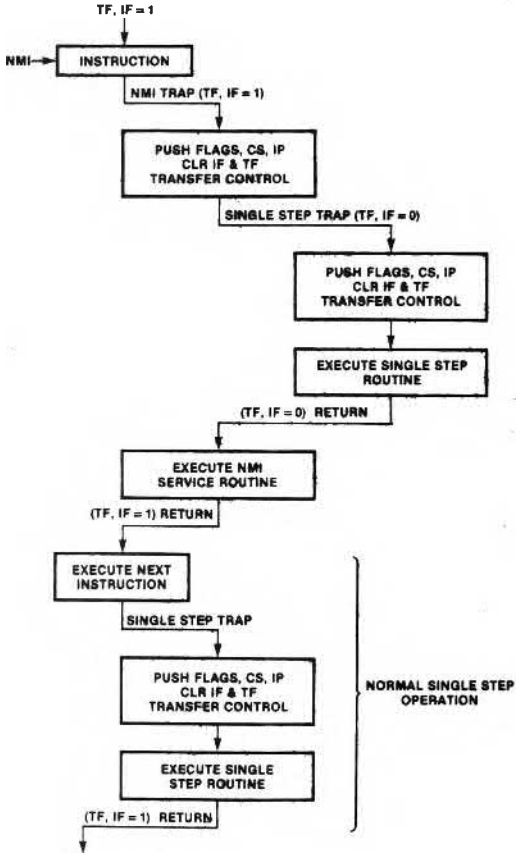


Figure 3E3a. NMI During Single Stepping and Normal Single Step Operation

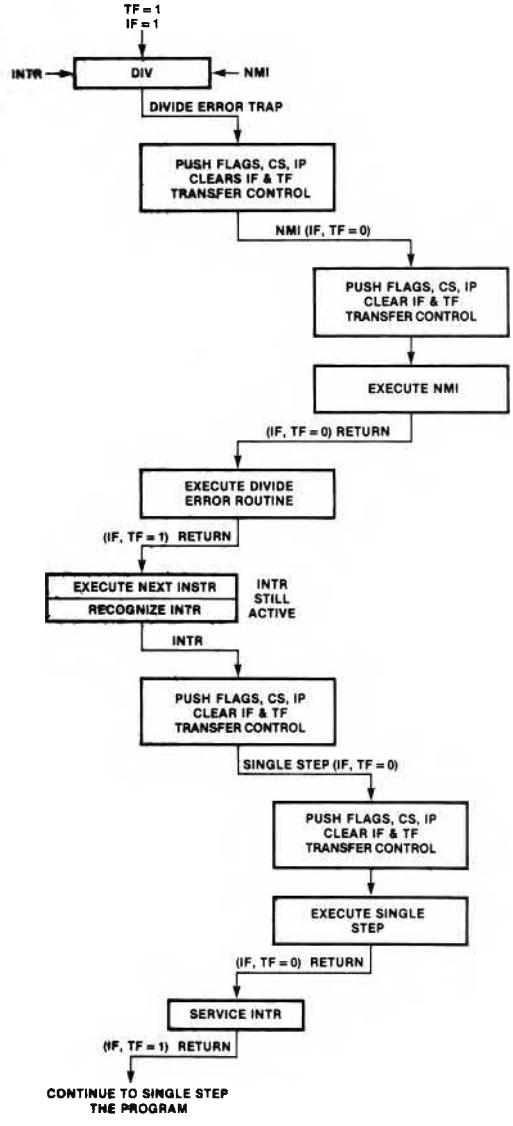
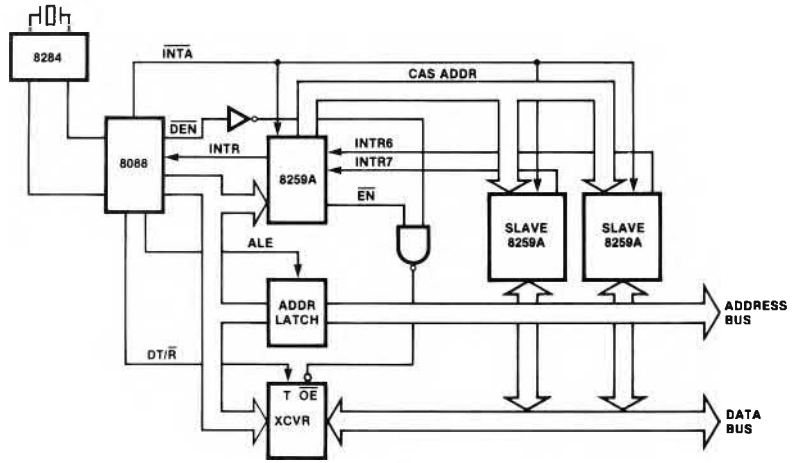


Figure 3E3b. NMI, INTR, Single Step and Divide Error Simultaneous Interrupts

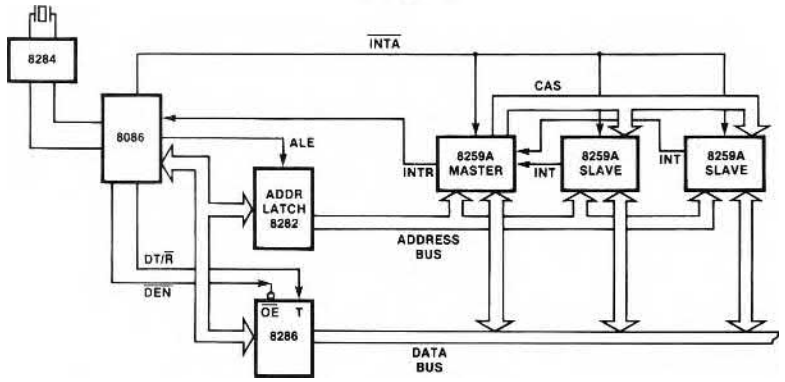
SYSTEM CONFIGURATIONS

To accommodate the $\overline{\text{INTA}}$ protocol of the maskable hardware interrupts, the 8259A is provided as part of the 8086 family. This component is programmable to operate in both 8080/8085 systems and 8086 systems. The devices are cascadable in master/slave arrangements to allow up to 64 interrupts in the system. Figures 3E4 and 3E5 are examples of 8259A's in minimum and maximum mode 8086 systems. The minimum mode configuration (a) shows an 8259A connected to the CPU's

connected to a demultiplexed bus system. These interconnects are also applicable to maximum mode systems. The configuration given for a maximum mode system shows a master 8259A on the CPU's multiplexed bus with additional slave 8259A's out on the buffered system bus. This configuration demonstrates several unique features of the maximum mode system interface. If the master 8259A receives interrupts from a mix of slave 8259A's and regular interrupting devices, the slaves must provide the type number for devices connected to them while the master provides the type number for devices directly attached to its interrupt inputs. The master 8259A is programmable to determine if an interrupt is from a direct input or a slave 8259A and will use this information to enable or disable the data bus transceivers (via the 'nand' function of DEN and $\overline{EN}$). If the master must provide the type number, it will disable the data bus transceivers. If the slave provides the type number, the master will enable the data bus transceivers. The $\overline{EN}$ output is normally high to allow the proper slave when servicing a slave interrupt, the master must provide a cascade address to the slave. If the 8288 is not strapped in the I/O bus mode (the 8288 IOB input connected to ground), the MCE/$\overline{PDEN}$ output becomes a MCE or Master Cascade Enable output. This signal is only active during $\overline{INTA}$ cycles as shown in Figure 3E6 and enables the master 8259A's cascade address onto the 8086's local bus during ALE. This allows the address latches to capture the cascade address with ALE and allows use of the system address bus for selecting the proper slave 8259A. The MCE is gated with $\overline{LOCK}$ to minimize local bus contention between the 8086 three-stating its bus outputs and the cascade address being enabled onto the bus. The first $\overline{INTA}$ bus cycle allows the master to resolve internal priorities and output a cascade address to be transmitted to the slaves on the subsequent $\overline{INTA}$ bus cycle. For additional information on the 8259A, reference application note AP-59.



a.



b.

Figure 3E4. Min Mode 8086 with Master 8259A on the Local Bus and Slave 8259As on the System Bus

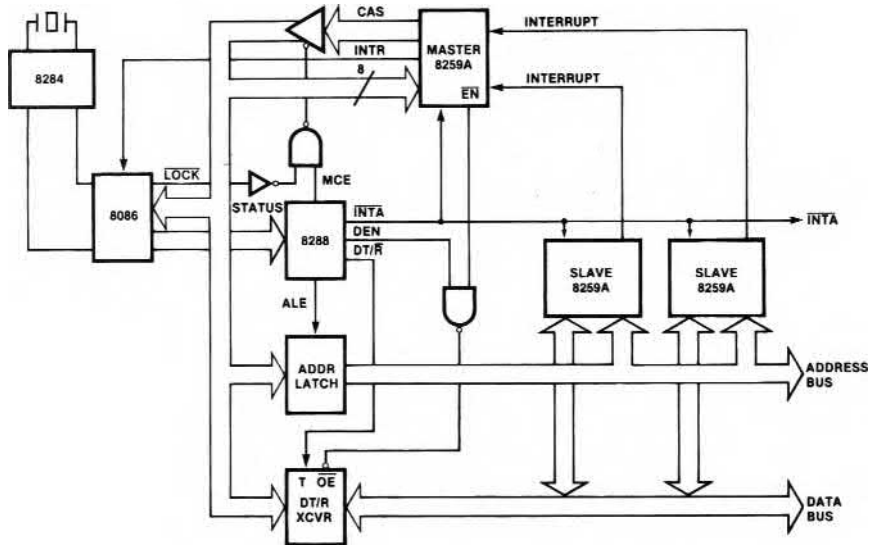


Figure 3E5. Max Mode 8086 with Master 8259A on the Local Bus and Slave 8259As on the System Bus

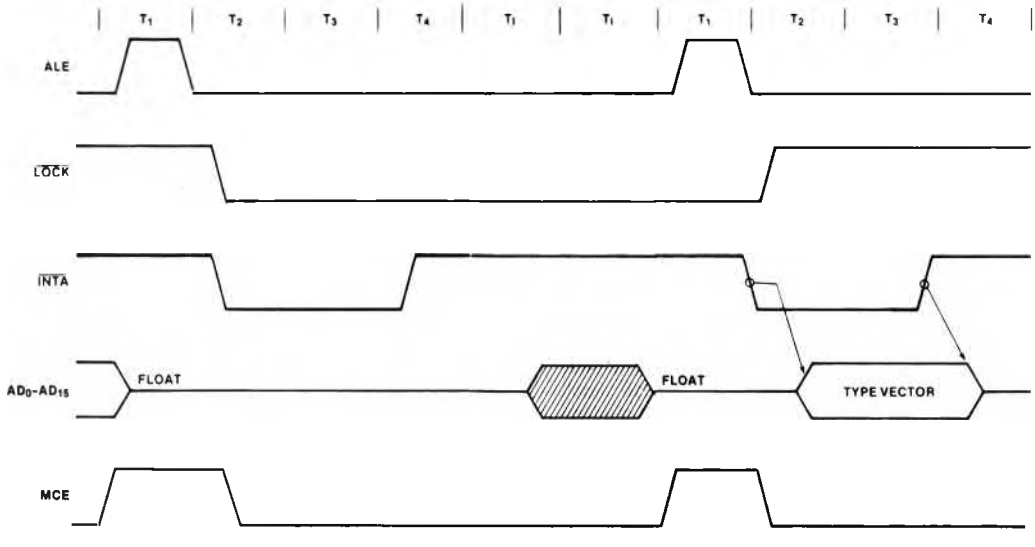


Figure 3E6. MCE Timing to Gate 8259A CAS Address onto the 8086 Local Bus

3F. Interpreting the 8086 Bus Timing Diagrams

At first glance, the 8086 bus timing diagrams (Diag. 3F1 min mode and Diag. 3F2 max mode) appear rather complex. However, with a few words of explanation on how to interpret them, they become a powerful tool in determining system requirements. The timing diagrams for both the minimum and maximum modes may be divided into six sections: (1) address and ALE timing; (2) read cycle timing; (3) write cycle timing; (4) interrupt acknowledge timing; (5) ready timing; and (6) HOLD/HLDA or $\overline{RQ}/\overline{GT}$ timing. Since the A.C. characteristics of the signals are specified relative to the CPU clock, the relationship between the majority of signals can be deduced by simply determining the clock cycles between the clock edges the signals are relative to and adding or subtracting the appropriate minimum or maximum parameter values. One aspect of system timing not compensated for in this approach is the worst case relationship between minimum and maximum parameter values (also known as tracking relationships). As an example, consider a signal which has specified minimum and maximum turn on and turn off delays. Depending on device characteristics, it may not be possible for the component to simultaneously demonstrate a maximum turn-on and minimum turn-off delay even though worst case analysis might imply the possibility. This argument is characteristic of MOS devices and is therefore applicable to the 8086 A.C. characteristics. The message is: worst case analysis mixing minimum and maximum delay parameters will typically exceed the worst case obtainable and therefore should not be subjected to further subjective degradation to obtain worst-worst case values. This section will provide guidelines for specific areas of 8086 timing sensitive to tracking relationships.

A. MINIMUM MODE BUS TIMING

1. ADDRESS and ALE

The address/ALE timing relationship is important to determine the ability to capture a valid address from the multiplexed bus. Since the 8282 and 8283 latches capture the address on the trailing edge of ALE, the critical timing involves the state of the address lines when ALE terminates. If the address valid delay is assumed to be maximum TCLAV and ALE terminates at its earliest point, TCHLLmin (assuming zero minimum delay), the address would be valid only TCLCHmin-TCLAVmax = 8 ns prior to ALE termination. This result is unrealistic in the assumption of maximum TCLAV and minimum TCHLL. To provide an accurate measure of the true worst case, a separate parameter specifies the minimum time for address valid prior to the end of ALE (TAVAL). TAVAL = TCLCH-60 ns overrides the clock related timings and guarantees 58 ns of address setup to ALE termination for a 5 MHz 8086. The address is guaranteed to remain valid beyond the end of ALE by the TLLAX parameter. This specification overrides the relationship between TCHLL and TCLAX which might seem to imply the address may not be valid by the end of the latest possible ALE. TLLAX holds for the entire address bus. The TCLAXmin spec on the address indicates the earliest the bus will go invalid if not restrained by a slow ALE. TLLAX and TCLAX apply to the entire multiplexed bus for both read and write cycles. AD15-0 is three-stated for read cycles and immediately switched to write data during write cycles. AD19-16 immediately switch from address to status for both read and write cycles. The minimum ALE pulse width is guaranteed by TLHLLmin which takes precedence over the value obtained by relating TCLLHmax and TCHLLmin.

To determine the worst case delay to valid address on a demultiplexed address bus, two paths must be considered: (1) delay of valid address and (2) delay to ALE. Since the 8282 and 8283 are flow through latches, a valid address is not transmitted to the address bus until ALE is active. A comparison of address valid delay TCLAVmax with ALE active delay TCLLHmax indicates TCLAVmax is the worst case. Subtracting the latch propagation delay gives the worst case address bus valid delay from the start of the bus cycle.

2. Read Cycle Timing

Read timing consists of conditioning the bus, activating the read command and establishing the data transceiver enable and direction controls. DT/$\overline{R}$ is established early in the bus cycle and requires no further consideration. During read, the $\overline{DEN}$ signal must allow the transceivers to propagate data to the CPU with the appropriate data setup time and continue to do so until the required data hold time. The $\overline{DEN}$ turn on delay allows TCLCL + TCHCLmin – TCVCTVmax – TDVCL = 127 ns transceiver enable time prior to valid data required by the CPU. Since the CPU data hold time TCLDXmin and minimum $\overline{DEN}$ turnoff delay TCVCTXmin are both 10 ns relative to the same clock edge, the hold time is guaranteed. Additionally, $\overline{DEN}$ must disable the transceivers prior to the CPU redriving the bus with the address for the next bus cycle. The maximum $\overline{DEN}$ turn off delay (TCVCTXmax) compared with the minimum delay for addresses out of the 8086 (TCLCL + TCLAVmin) indicates the transceivers are disabled at least 105 ns before the CPU drives the address onto the multiplexed bus.

If memory or I/O devices are connected directly to the multiplexed address and data bus, the TAZRL parameter guarantees the CPU will float the bus before activating read and allowing the selected device to drive the bus. At the end of the bus cycle, the TRHAV parameter specifies the bus float delay the device being deselected must satisfy to avoid contention with the CPU driving the address for the next bus cycle. The next bus cycle may start as soon as the cycle following T4 or any number of clock cycles later.

The minimum delay from read active to valid data at the CPU is 2TCLCL – TCLRLmax – TDVCL = 205 ns. The minimum pulse width is 2TCLCL – 75 ns = 325 ns. This specification (TRLRH) overrides the result which could be derived from clock relative delays (2TCLCL – TCLRLmax + TCLRHmin).

3. Write Cycle Timing

The write cycle involves providing write data to the system, generating the write command and controlling data bus transceivers. The transceiver direction control signal DT/$\overline{R}$ is conditioned to transmit at the end of each read cycle and does not change during a write cycle.

This allows the transceiver enable signal $\overline{DEN}$ to be active early in the cycle (while addresses are valid) without corrupting the address on the multiplexed bus. The write data and write command are both enabled from the leading edge of T2. Comparing minimum $\overline{WR}$ active delay TCVCTVmin with the maximum write data delay TCLDV indicates that write data may be not valid until 100 ns after write is active. The devices in the system should capture data on the trailing edge of the write command rather than the leading edge to guarantee valid data. The data from the 8086 is valid a minimum of 2TCLCL – TCLDVmax + TCVCTXmin = 300 ns before the trailing edge of write. The minimum write pulse width is TWLWH = 2TCLCL – 60 ns = 340 ns. The CPU maintains valid write data TWHDX ns after write. The TWHDZ specification overrides the result derived by relating TCLCHmin and TCHDZmin which implies write data may only be valid 18 ns after $\overline{WR}$. The 8086 floats the bus after write only if being forced off the bus by a HOLD or $\overline{RQ}$ input. Otherwise, the CPU simply switches the output drivers from data to address at the beginning of the next bus cycle. As with the read cycle, the next bus cycle may start in the clock cycle following T4 or any clock cycle later.

$\overline{DEN}$ is disabled a minimum of TCLCHmin + TCVCTXmin – TCVCTXmax = 18 ns after write to guarantee data hold time to the selected device. Since we are again evaluating a minimum TCVCTX with a maximum TCVCTX, the real minimum delay from the end of write to transceiver disable is approximately 60 ns.

4. Interrupt Acknowledge Timing

The interrupt acknowledge sequence consists of two interrupt acknowledge bus cycles as previously described. The detailed timing of each cycle is identical to the read cycle timing with two exceptions: command timing and address/data bus timing.

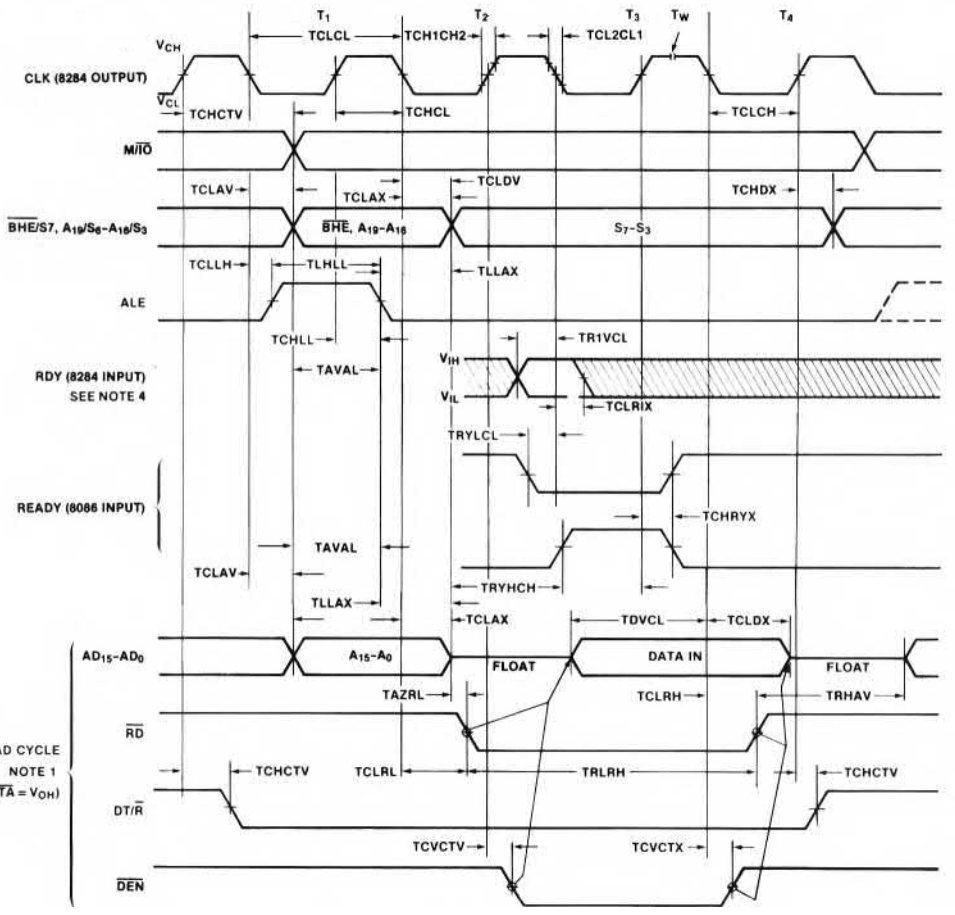
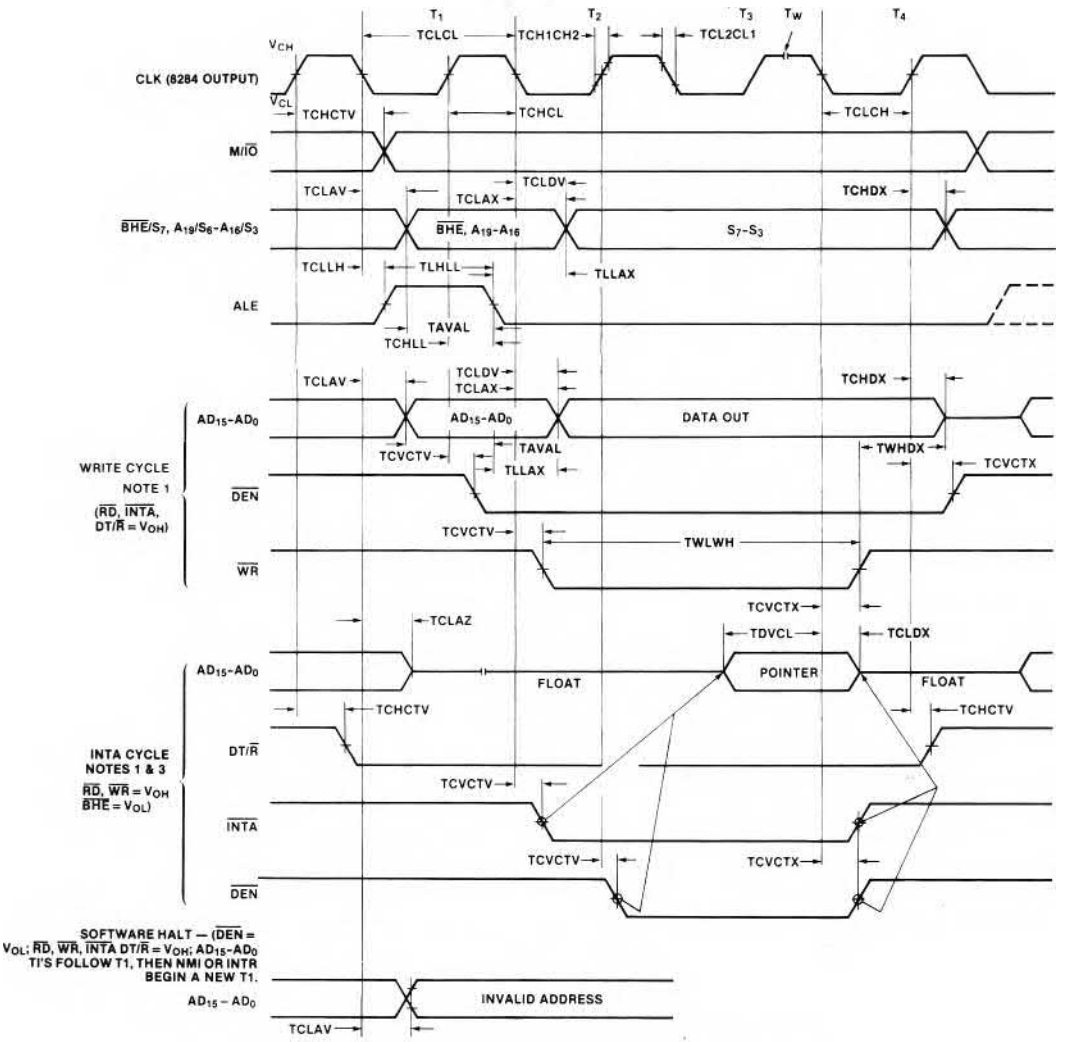


Figure 3F1. 8086 Bus Timing — Minimum Mode System



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. BOTH INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING THE SECOND INTA CYCLE. CONTROL SIGNALS SHOWN FOR SECOND INTA CYCLE.
4. SIGNALS AT 8284 ARE SHOWN FOR REFERENCE ONLY.
5. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

Figure 3F1. 8086 Bus Timing — Minimum Mode System (Con't)

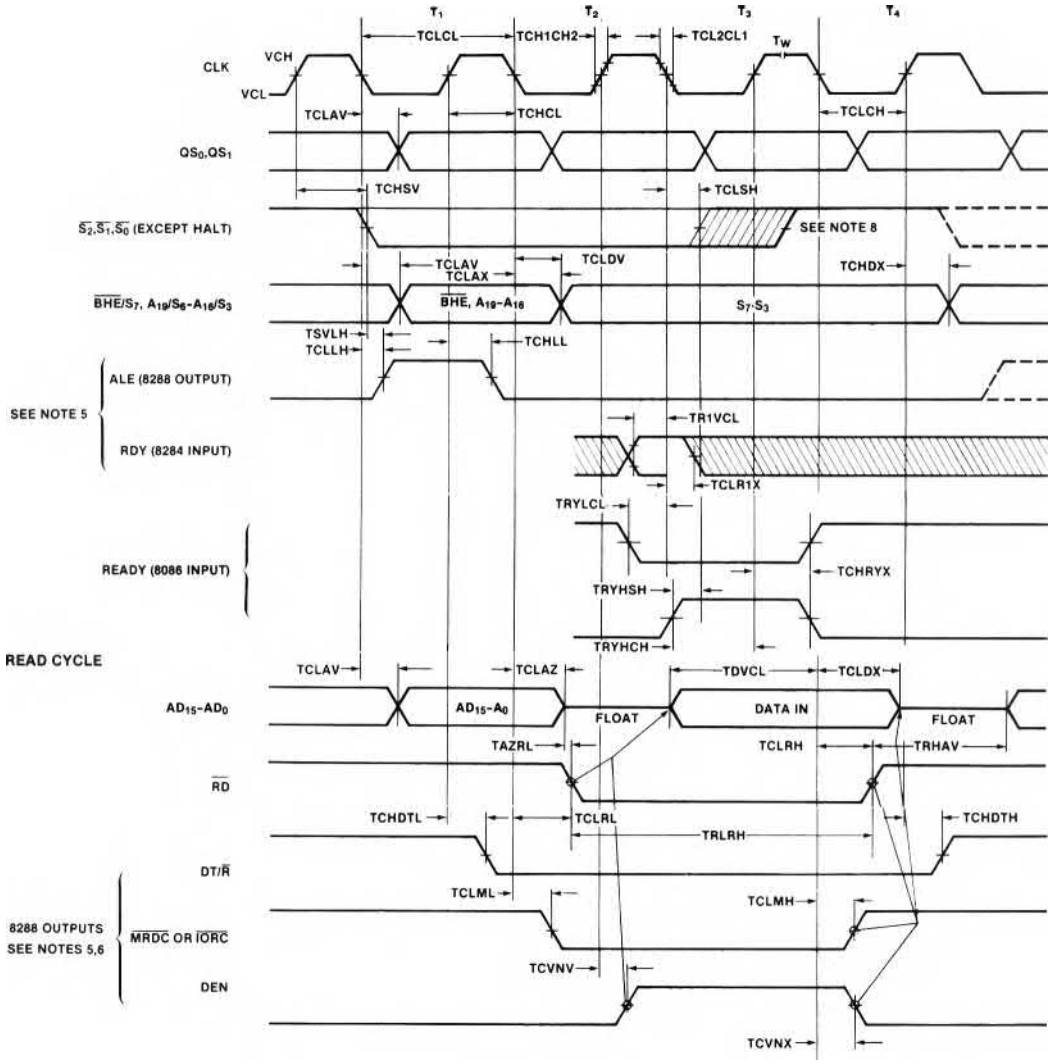
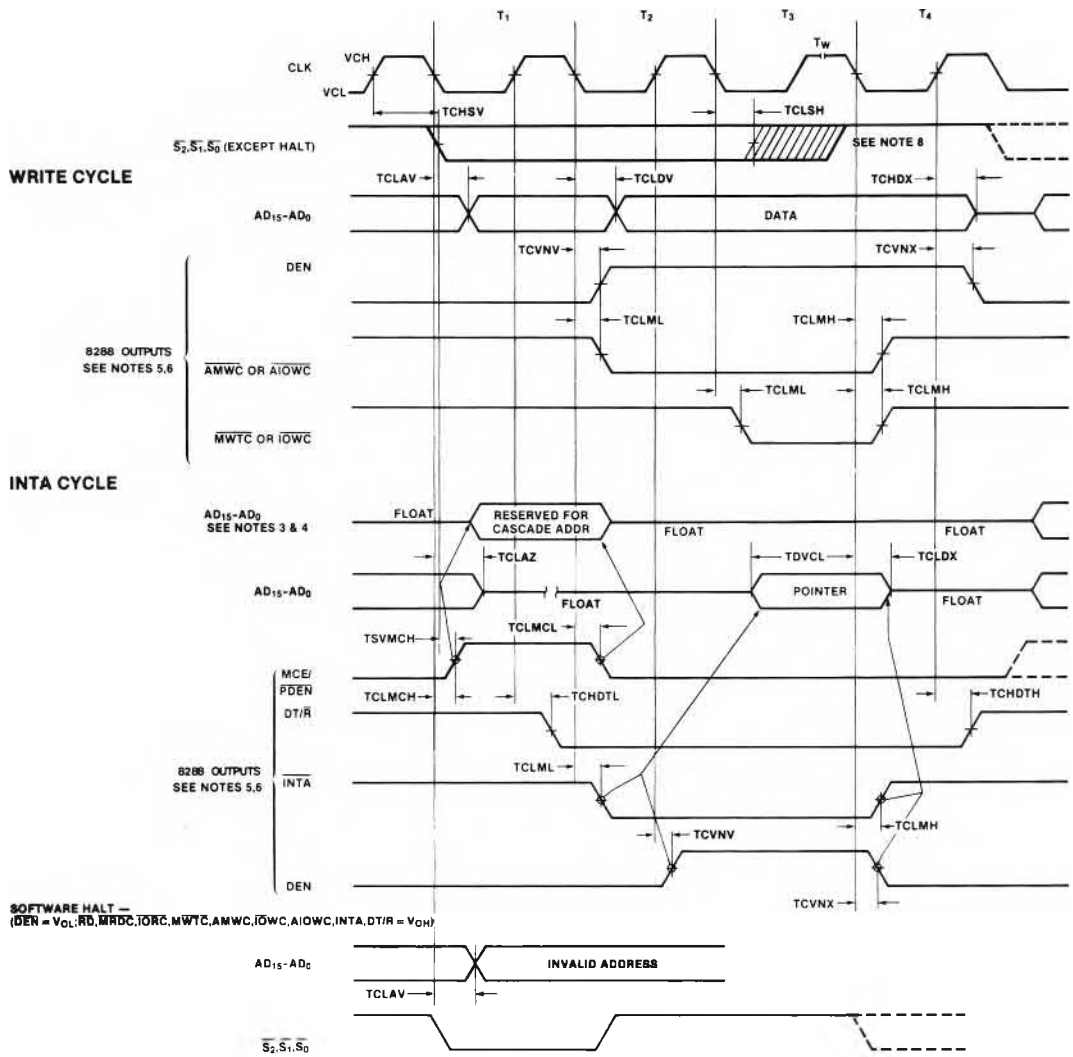


Figure 3F2a. 8086 Bus Timing — Maximum Mode System (Using 8288)



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. CASCADE ADDRESS IS VALID BETWEEN FIRST AND SECOND INTA CYCLES.
4. BOTH INTA CYCLES RUN BACK-TO-BACK. THE 8086 LOCAL ADDR/DATA BUS IS FLOATING DURING THE SECOND INTA CYCLE. CONTROL FOR POINTER ADDRESS IS SHOWN FOR SECOND INTA CYCLE.
5. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
6. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS (MRDC, MWTC, AMWC, IORC, IOWC, AIOWC, INTA AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
7. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
8. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.

Figure 3F2b. 8086 Bus Timing — Maximum Mode System (Using 8288) (Con't)

The multiplexed address/data bus floats from the beginning (T1) of the $\overline{INTA}$ cycle (within TCLAZ ns). The upper four multiplexed address/status lines do not three-state. The address value on A19-A16 is indeterminate but the status information will be valid (S3 = 0, S4 = 0, S5 = IF, S6 = 0, S7 = $\overline{BHE}$ = 0). The multiplexed address/data lines will remain in three-state until the cycle after T4 of the $\overline{INTA}$ cycle. This sequence occurs for each of the $\overline{INTA}$ bus cycles. The interrupt type number read by the 8086 on the second $\overline{INTA}$ bus cycle must satisfy the same setup and hold times required for data during a read cycle.

The $\overline{DEN}$ and DT/$\overline{R}$ signals are enabled for each $\overline{INTA}$ cycle and do not remain active between the two cycles. Their timing for each cycle is identical to the read cycle.

The $\overline{INTA}$ command has the same timing as the write command. It is active within 110 ns of the start of T2 providing 260 ns of access time from command to data valid at the 8086. The command is active a minimum of TCVCTXmin = 10 ns into T4 to satisfy the data hold time of the 8086. This provides minimum $\overline{INTA}$ pulse width of 300 ns, however taking signal delay tracking into consideration gives a minimum pulse width of 340 ns. Since the maximum inactive delay of $\overline{INTA}$ is TCVCTXmax = 110 ns and the CPU will not drive the bus until 15 ns (TCLAVmin) into the next clock cycle, 105 ns are available for interrupt devices on the local bus to float their outputs. If the data bus is buffered, $\overline{DEN}$ provides the same amount of time for local bus transceivers to three-state their outputs.

5. Ready Timing

The detailed timing requirements of the 8086 ready signal and the system ready signal into the 8284 are described in Section 3D. The system ready signal is typically generated from either the address decode of the selected device or the address decode and the command ($\overline{RD}$, $\overline{WR}$, $\overline{INTA}$). For a system which is normally not ready, the time to generate ready from a valid address and not insert a wait state, is 2TCLCL – TCLAVmax – TR1VCLmax = 255 ns. This time is available for buffer delays and address decoding to determine if the selected device does not require a wait state and drive the RDY line high. If wait cycles are required, the user hardware must provide the appropriate ready delay. Since the address will not change until the next ALE, the RDY will remain valid throughout the cycle. If the system is normally ready, selected devices requiring wait states also have 255 ns to disable the RDY line. The user circuitry must delay re-enabling RDY by the appropriate number of wait states.

If the $\overline{RD}$ command is used to enable the RDY signal, TCLCL – TCLRLmax – TRIVCLmax = 15 ns are available for external logic. If the $\overline{WR}$ command is used, TCLCL – TCVCTVmax – TRIVCLmax = 55 ns are available. Comparison of RDY control by address or command indicates that address decoding provides the best timing. If the system is normally not ready, address decode alone could be used to provide RDY for devices not requiring wait states while devices requiring wait states may use a combination of address decode and command to activate a wait state generator. If the system is normally ready, devices not requiring wait states do nothing to RDY while devices needing wait states should disable RDY via the address decode and use a combination of address decode and command to activate a delay to re-enable RDY.

If the system requires no wait states for memory and a fixed number of wait states for $\overline{RD}$ and $\overline{WR}$ to all I/O devices, the M/$\overline{IO}$ signal can be used as an early indication of the need for wait cycles. This allows a common circuit to control ready timing for the entire system without feedback of address decodes.

6. Other Considerations

Detailed HOLD/HLDA timing is covered in the next section and is not examined here. One last signal consideration needs to be mentioned for the minimum mode system. The $\overline{TEST}$ input is sampled by the 8086 only during execution of the WAIT instruction. The $\overline{TEST}$ signal should be active for a minimum of 6 clock cycles during the WAIT instruction to guarantee detection.

B. MAXIMUM MODE BUS TIMING

The maximum mode 8086 bus operations are logically equivalent to the minimum mode operation. Detailed timing analysis now involves signals generated by the CPU and the 8288 bus controller. The 8288 also provides additional control and command signals which expand the flexibility of the system.

1. ADDRESS and ALE

In the maximum mode, the address information continues to come from the CPU while the ALE strobe is generated by the 8288. To determine the worst case relationships between ALE and the address, we first must determine 8288 ALE activation relative to the $\overline{S0}$-$\overline{S2}$ status from the CPU. The maximum mode timing diagram specifies two possible delay paths to generate ALE. The first is TCHSV + TSVLH measured from the rising edge of the clock cycle preceding T1. The second path is TCLLH measured from the start of T1. Since the 8288 initiates a bus cycle from the status lines leaving the passive state ($\overline{S0}$-$\overline{S2}$ = 1), if the 8086 is late in issuing the status (TCHSVmax) while the clock high time is a minimum (TCHCLmin), the status will not have changed by the start of T1 and ALE is issued TSVLH ns after the status changes. If the status changes prior to the beginning of T1, the 8288 will not issue the ALE until TCLLH ns after the start of T1. The resulting worst case delay to enable ALE (relative to the start of T1) is TCHSVmax + TSVLHmax – TCHCLmin = 58 ns. Note, when calculating signal relationships, be sure to use the proper maximum mode values rather than equivalent minimum mode values.

The trailing edge of ALE is triggered in the 8288 by the positive clock edge in T1 regardless of the delay to enable ALE. The resulting minimum ALE pulse width is TCLCHmax – 58 ns = 75 ns assuming TCHLL = 0. TCLCHmax must be used since TCHCLmin was assumed to derive the 58 ns ALE enable delay. The address is guaranteed to be valid TCLCHmin + TCHLLmin – TCLAVmax = 8 ns prior to the trailing edge

by separate devices, no tracking of A.C. characteristics can be assumed.

The address hold time to the latches is guaranteed by the address remaining valid until the end of T1 while ALE is disabled a maximum of 15 ns from the positive clock transition in T1 (TCHCLmin – TCHLLmax = 52 ns address hold time). The multiplexed bus transitions from address to status and write data or three-state (for read) are identical to the minimum mode timing. Also, since the address valid delay (TCLAV) remains the critical path in establishing a valid address, the address access times to valid data and ready are the same as the minimum mode system.

2. Read Cycle Timing

The maximum mode system offers read signals generated by both the 8086 and the 8288. The 8086 $\overline{RD}$ output signal timing is identical to the minimum mode system. Since the A.C. characteristics of the read commands generated by the 8288 are significantly better than the 8086 output, access to devices on the demultiplexed buffered system bus should use the 8288 commands. The 8086 $\overline{RD}$ signal is available for devices which reside directly on the multiplexed bus. The following evaluations for read, write and interrupt acknowledge only consider the 8288 command timing.

The 8288 provides separate memory and I/O read signals which conform to the same A.C. characteristics. The commands are issued TCLML ns after the start of T2 and terminate TCLMH ns after the start of T4. The minimum command length is 2TCLCL – TCLMLmax + TCLMLmin = 375 ns. The access time to valid data at the CPU is 2TCLCL – TCLMLmax – TDVCLmax = 335 ns. Since the 8288 was designed for systems with buffered data busses, the commands are enabled before the CPU has three-stated the multiplexed bus and should not be used with devices which reside directly on the multiplexed bus (to do so could result in bus contention during 8086 bus float and device turn-on).

The direction control for data bus transceivers is established in T1 while the transceivers are not enabled by DEN until the positive clock transition of T2. This provides TCLCH + TCVNVmin = 123 ns for 8086 bus float delay and TCHCLmin + TCLCL – TCVNVmax – TDVCLmax = 187 ns of transceiver active to data valid at the CPU. Since both DEN and command are valid a minimum of 10 ns into T4, the CPU data hold time TCLDX is guaranteed. A maximum DEN disable of 45 ns (TCVNX max) guarantees the transceivers are disabled by the start of the next 8086 bus cycle (215 ns minimum from the same clock edge). On the positive clock transition of T4, DT/$\overline{R}$ is returned to transmit in preparation for a possible write operation on the next bus cycle. Since the system memory and I/O devices reside on a buffered system bus, they must three-state their outputs before the device for the next bus cycle is selected (approximately 2TCLCL) or the transceivers drive write data onto the bus (approximately 2TCLCL).

3. Advanced Write Commands

The 8288 provides advanced write commands for memory and I/O. The advanced write commands are active a full clock cycle ahead of the normal write commands and have timing identical to the read commands. The advanced write pulse width is 2TCLCL – TCLMLmax + TCLMHmin = 375 ns while the normal write pulse width is TCLCL – TCLMLmax + TCLMHmin = 175 ns. Write data setup time to the selected device is a function of either the data valid delay from the 8086 (TCLDV) or the transceiver enable delay TCVNV. The worst case delay to valid write data is TCLDV = 110 ns minus transceiver propagation delays. This implies the data may not be valid until 100 ns after the advanced write command but will be valid approximately TCLCL – TCLDVmax + TCLMLmin = 100 ns prior to the leading edge of the normal write command. Data will be valid 2TCLCL – TCLDVmax + TCLMHmin = 300 ns before the trailing edge of either write command. The data and command overlap for the advanced command is 300 ns while the overlap with the normal write command is 175 ns. The transceivers are disabled a minimum of TCLCHmin – TCLMHmax + TCVNXmin = 85 ns after the write command while the CPU provides valid data a minimum of TCLCHmin – TCLMHmax + TCHDZmin = 85 ns. This guarantees write data hold of 85 ns after the write command. The transceivers are disabled TCLCL – TCVNXmax + TCHDTLmin = 155 ns (assuming TCHDTL = 0) prior to transceiver direction change for a subsequent read cycle.

4. Interrupt Acknowledge Timing

The maximum mode $\overline{INTA}$ sequence is logically identical to the minimum mode sequence. The transceiver control (DEN and DT/$\overline{R}$) and $\overline{INTA}$ command timing of each interrupt acknowledge cycle is identical to the read cycle. As in the minimum mode system, the multiplexed address/data bus will float from the leading edge of T1 for each $\overline{INTA}$ bus cycle and not be driven by the CPU until after T4 of each $\overline{INTA}$ cycle. The setup and hold times on the vector number for the second cycle are the same as data setup and hold for the read. If the device providing the interrupt vector number is connected to the local bus, TCLCL – TCLAZmax + TCLMLmin = 130 ns are available from 8086 bus float to $\overline{INTA}$ command active. The selected device on the local bus must disable the system data bus transceivers since DEN is still generated by the 8288.

If the 8288 is not in the IOB (I/O Bus) mode, the 8288 MCE/$\overline{PDEN}$ output becomes the MCE output. This output is active during each $\overline{INTA}$ cycle and overlaps the ALE signal during T1. The MCE is available for gating cascade addresses from a master 8259A onto three of the upper AD15-AD8 lines and allowing ALE to latch the cascade address into the address latches. The address lines may then be used to provide CAS address selection to slave 8259A's located on the system bus (reference Figure 3E5). MCE is active within 15 ns of status or the start of T1 for each $\overline{INTA}$ cycle. MCE should not enable the CAS lines onto the multiplexed bus during the first cycle since the CPU does not guarantee to float

the bus until 80 ns into the first $\overline{INTA}$ cycle. The first MCE can be inhibited by gating MCE with $\overline{LOCK}$. The 8086 $\overline{LOCK}$ output is activated during T2 of the first cycle and disabled during T2 of the second cycle. The overlap of $\overline{LOCK}$ with MCE allows the first MCE to be masked and the second MCE to gate the cascade address onto the local bus. Since the 8259A will not provide a cascade address until the second cycle, no information is lost. As with ALE, MCE is guaranteed valid within 58 ns of the start of T1 to allow 75 ns CAS address setup to the trailing edge of ALE. MCE remains active TCHCLmin – TCHLLmax + TCLMCLmin = 52 ns after ALE to provide data hold time to the latches.

If the 8288 is strapped in the IOB mode, the MCE output becomes $\overline{PDEN}$ and all I/O references are assumed to be devices on the local bus rather than the demultiplexed system bus. Since $\overline{INTA}$ cycles are considered I/O cycles, all interrupts are assumed to come from the local system and cascade addresses are not gated onto the system address bus. Additionally, the DEN signal is not enabled since no I/O transfers occur on the system bus. If the local I/O bus is also buffered by transceivers, the $\overline{PDEN}$ signal is used to enable those transceivers. $\overline{PDEN}$ A.C. characteristics are identical to DEN with $\overline{PDEN}$ enabled for I/O references and DEN enabled for instruction or memory data references.

5. Ready Timing

Ready timing based on address valid timing is the same for maximum and minimum mode systems. The delay from 8288 command valid to RDY valid at the 8284 is TCLCL – TCLMLmax – TRIVCLmin = 130 ns. This time is available for external circuits to determine the need to insert wait states and disable RDY or enable RDY to avoid wait states. $\overline{INTA}$, all read commands and advanced write commands provide this timing. The normal write command is not valid until after the RDY signal must be valid. Since both normal and advanced write commands are generated by the 8288 for all write cycles, the advanced write may be used to generate a RDY indication even though the selected device uses the normal write command.

Since separate commands are provided for memory and I/O, no M/$\overline{IO}$ signal is specifically available as in the minimum mode to allow an early 'wait state required' indication for I/O devices. The $\overline{S2}$ status line, however is logically equivalent to the M/$\overline{IO}$ signal and can be used for this purpose.

6. Other Considerations

The $\overline{RQ}$/$\overline{GT}$ timing is covered in the next section and will not be duplicated here. The only additional signals to be considered in the maximum mode are the queue status lines QS0, QS1. These signals are changed on the leading edge of each clock cycle (high to low transition) including idle and wait cycles (the queue status is independent of the bus activity). External logic may sample the lines on the low to high transition of each clock cycle. When sampled, the signals indicate the queue activity in the previous clock cycle and therefore lag the CPU's activity by one cycle. The $\overline{TEST}$ input requirements are identical to those stated for the minimum mode.

To inform the 8288 of HALT status when a HALT instruction is executed, the 8086 will initiate a status transition from passive to HALT status. The status change will cause the 8288 to emit an ALE pulse with an indeterminate address. Since no bus cycle is initiated (no command is issued), the results of this address will not affect CPU operation (i.e., no response such as READY is expected from the system). This allows external hardware to latch and decode all transitions in system status.

3G. Bus Control Transfer (HOLD/HLDA and $\overline{RQ}$/$\overline{GT}$)

The 8086 supports protocols for transferring control of the local bus between itself and other devices capable of acting as bus masters. The minimum mode configuration offers a signal level handshake similar to the 8080 and 8085 systems. The maximum mode provides an enhanced pulse sequence protocol designed to optimize utilization of CPU pins while extending the system configurations to two prioritized levels of alternate bus masters. These protocols are simply techniques for arbitration of control of the CPU's local bus and should not be confused with the need for arbitration of a system bus.

1. MINIMUM MODE

The minimum mode 8086 system uses a hold request input (HOLD) to the CPU and a hold acknowledge (HLDA) output from the CPU. To gain control of the bus, a device must assert HOLD to the CPU and wait for the HLDA before driving the bus. When the 8086 can relinquish the bus, it floats the $\overline{RD}$, $\overline{WR}$, $\overline{INTA}$ and M/$\overline{IO}$ command lines, the $\overline{DEN}$ and DT/$\overline{R}$ bus control lines and the multiplexed address/data/status lines. The ALE signal is not three-stated. The CPU acknowledges the request with HLDA to allow the requestor to take control of the bus. The requestor must maintain the HOLD request active until it no longer requires the bus. The HOLD request to the 8086 directly affects the bus interface unit and only indirectly affects the execution unit. The CPU will continue to execute from its internal queue until either more instructions are needed or an operand transfer is required. This allows a high degree of overlap between CPU and auxiliary bus master operation. When the requestor drops the HOLD signal, the 8086 will respond by dropping HLDA. The CPU will not re-drive the bus, command and control signals from three-state until it needs to perform a bus transfer. Since the 8086 may still be executing from its internal queue when HOLD drops, there may exist a period of time during which no device is driving the bus. To prevent the command lines from drifting below the minimum VIH level during the transition of bus control, 22K ohm pull up resistors should be connected to the bus command lines. The timing diagram in Figure 3G1 shows the handshake sequence and 8086 timing to sample HOLD, float the bus, and enable/disable HLDA relative to the CPU clock.

To guarantee valid system operation, the designer must assure that the requesting device does not assert con-

trol of the bus prior to the 8086 relinquishing control and that the device relinquishes control of the bus prior to the 8086 driving the bus. The HOLD request into the 8086 must be stable THVCH ns prior to the CPU's low to high clock transition. Since this input is not synchronized by the CPU, signals driving the HOLD input should be synchronized with the CPU clock to guarantee the setup time is not violated. Either clock edge may be used. The maximum delay between HLDA and the 8086 floating the bus is TCLAZmax – TCLHAVmin = 70 ns. If the system cannot tolerate the 70 ns overlap, HLDA active from the 8086 should be delayed to the device. The minimum delay for the CPU to drive the control bus from HOLD inactive is THVCHmin + 3TCLCL = 635 ns and THVCHmin + 3TCLCL + TCHCL = 701 ns to drive the multiplexed bus. If the device does not satisfy these requirements, HOLD inactive to the 8086 should be delayed. The delay from HLDA inactive to driving the busses is TCLCL + TCLCHmin – TCLHAVmax = 158 ns for the control bus and 2TCLCL – TCLHAVmax = 240 ns for the data bus.

1.1 Latency of HLDA to HOLD

The decision to respond to a HOLD request is made in the bus interface unit. The major factors that influence the decision are the current bus activity, the state of the $\overline{\text{LOCK}}$ signal internal to the CPU (activated by the software LOCK prefix) and interrupts.

If the $\overline{\text{LOCK}}$ is not active, an interrupt acknowledge cycle is not in progress and the BIU (Bus Interface Unit) is executing a T4 or TI when the HOLD request is received, the minimum latency to HLDA is:

35 ns	THVCH min (Hold setup)
65 ns	TCHCL min
200 ns	TCLCL (bus float delay)
10 ns	TCLHAV min (HLDA delay)
310 ns	@ 5 MHz

The maximum delay under these conditions is:

34 ns	(just missed setup time)
200 ns	delay to next sample
82 ns	TCHCL max
200 ns	TCLCL (bus float delay)
160 ns	TCLHAV max (HLDA delay)
677 ns	@ 5 MHz

If the BIU just initiated a bus cycle when the HOLD Request was received, the worst case response time is:

34 ns	THVCH (just missed)
82 ns	TCHCL max
7*200	bus cycle execution
N*200	N wait states/bus cycle
160 ns	TCLHAV max (HLDA delay)
1.676 μs	@ 5 MHz, no wait states

Note, the 200 ns delay for just missing is included in the delay for bus cycle execution. If the operand transfer is a word transfer to an odd byte boundary, two bus cycles are executed to perform the transfer. The BIU will not acknowledge a HOLD request between the two bus cycles. This type of transfer would extend the above maximum latency by four additional clocks plus N additional wait states. With no wait states in the bus cycle, the maximum would be 2.476 microseconds.

Although the minimum mode 8086 does not have a hardware $\overline{\text{LOCK}}$ output, the software LOCK prefix may still be included in the instruction stream. The CPU internally reacts to the LOCK prefix as would the maximum mode 8086. Therefore, the LOCK does not allow a HOLD request to be honored until completion of the instruction following the prefix. This allows an instruction which performs more than one memory reference (ex. ADD [BX], CX; which adds CX to [BX]) to execute without another bus master gaining control of the bus between memory references. Since the LOCK signal is active for one clock longer than the instruction execution, the maximum latency to HLDA is:

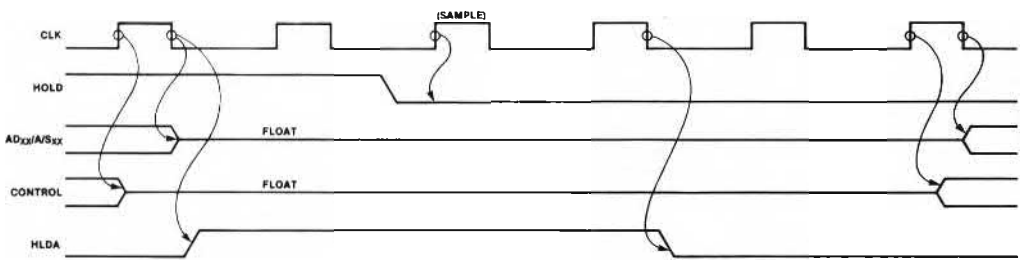


Figure 3G1. HOLD/HLDA Sequence

34 ns	THVCH (just miss)
200 ns	delay to next sample
82 ns	TCHCL max
(M + 1)*200 ns	LOCK instruction execution
200 ns	set up HLDA (internal)
160 ns	TCLHAV max (HLDA delay)
(M*200 ns) + 876 ns	@ 5 MHz

If the HOLD request is made at the beginning of an interrupt acknowledge sequence, the maximum latency to HLDA is:

34 ns	THVCH (just missed)
82 ns	TCHCL max
2600 ns	13 clock cycles for INTA
160 ns	TCLHAV max
2.876 μs	@ 5 MHz

1.2 Minimum Mode DMA Configuration

A typical use of the HOLD/HLDA signals in the minimum mode 8086 system is bus control exchange with DMA devices like the Intel 8257-5 or 8237 DMA controllers. Figure 3G2 gives a general interconnect for this type of configuration using the 8237-2. The DMA controller resides on the upper half of the 8086's local bus and shares the A8-A15 demultiplexing address latch of the 8086. All registers in the 8237-2 must be assigned odd addresses to allow initialization and interrogation by the CPU over the upper half of the data bus. The 8086 $\overline{RD}/\overline{WR}$ commands must be demultiplexed to provide separate I/O and memory commands which are compatible with the 8237-2 commands. The AEN control from the 8237-2 must disable the 8086 commands from the command bus, disable the address latches from the lower (A0-A7) and upper (A19-A16) address bus and select the 8237-2 address strobe (ADSTB) to the A8-A15 address latch. If the data bus is buffered, a pull-up resistor on the DEN line will keep the buffers disabled. The DMA controller will only transfer bytes between

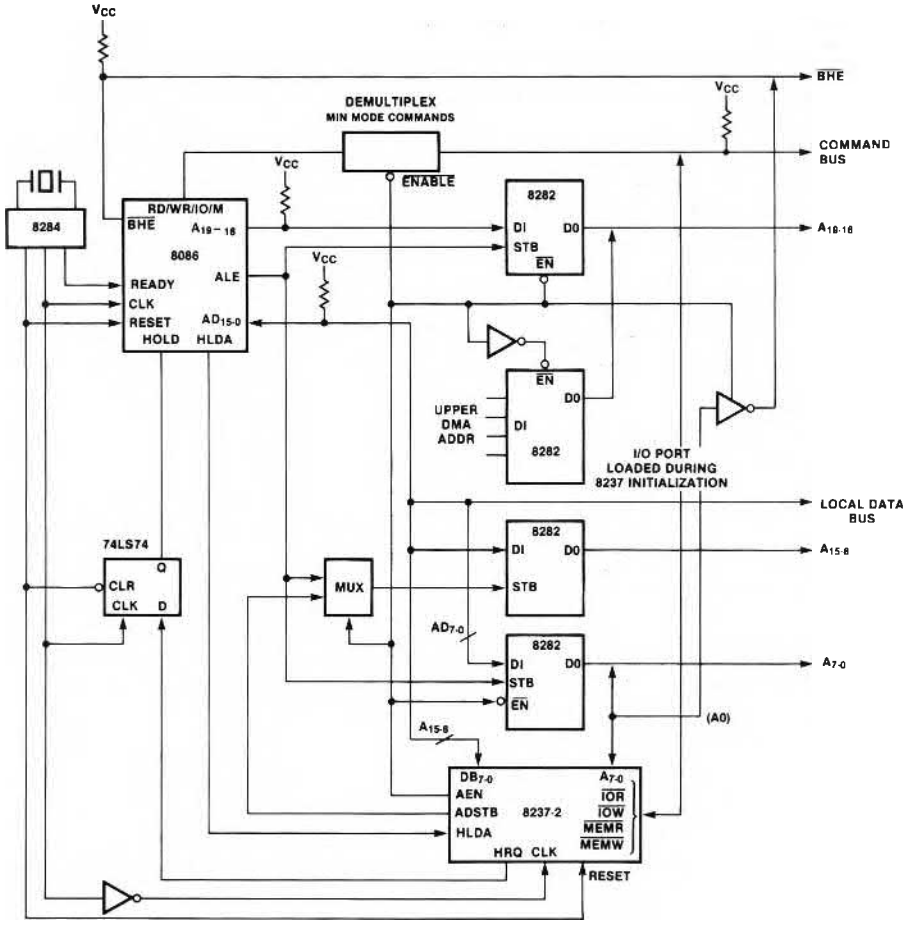


Figure 3G2. DMA Using the 8237-2

circuit given in Section 4. Address lines A7-A0 are driven directly by the 8237 and $\overline{BHE}$ is generated by inverting A0. If A19-A16 are used, they must be provided by an additional port with either a fixed value or initialized by software and enabled onto the address bus by AEN.

Figure 3G3 gives an interconnection for placing the 8257 on the system bus. By using a separate latch to hold the upper address from the 8257-5 and connecting the outputs to the address bus as shown, 16-bit DMA transfers are provided. In this configuration, AEN simultaneously enables A0 and $\overline{BHE}$ to allow word transfers. AEN still disables the CPU interface to the command and address busses.

2. MAXIMUM MODE ($\overline{RQ}/\overline{GT}$)

The maximum mode 8086 configuration supports a significantly different protocol for transferring bus control. When viewed with respect to the HOLD/HLDA sequence of the minimum mode, the protocol appears difficult to implement externally. However, it is necessary to understand the intent of the protocol and its purpose within the system architecture.

The maximum mode $\overline{RQ}/\overline{GT}$ sequence is intended to transfer control of the CPU local bus between the CPU and alternate bus masters which reside totally on the local bus and share the complete CPU interface to the system bus. The complete interface includes the address latches, data transceivers, 8288 bus controller and 8289 multi master bus arbiter. If the alternate bus masters in the system do not reside directly on the 8086 local bus, system bus arbitration is required rather than local CPU bus arbitration. To satisfy the need for multimaster system bus arbitration at each CPU's system interface, the 8289 bus arbiter should be used rather than the CPU $\overline{RQ}/\overline{GT}$ logic.

To allow a device with a simple HOLD/HLDA protocol to gain control of a single CPU system bus, the circuit in Figure 3G4 could be used. The design is effectively a simple bus arbiter which isolates the CPU from the system bus when an alternate bus master issues a HOLD request. The output of the circuit, $\overline{AEN}$ (Address ENable), disables the 8288 and 8284 when the 8086 indicates idle status ($\overline{S0}$,$\overline{S1}$,$\overline{S2}$ = 1), $\overline{LOCK}$ is not active and a HOLD request is active. With $\overline{AEN}$ inactive, the 8288 three-states the command outputs and disables DEN

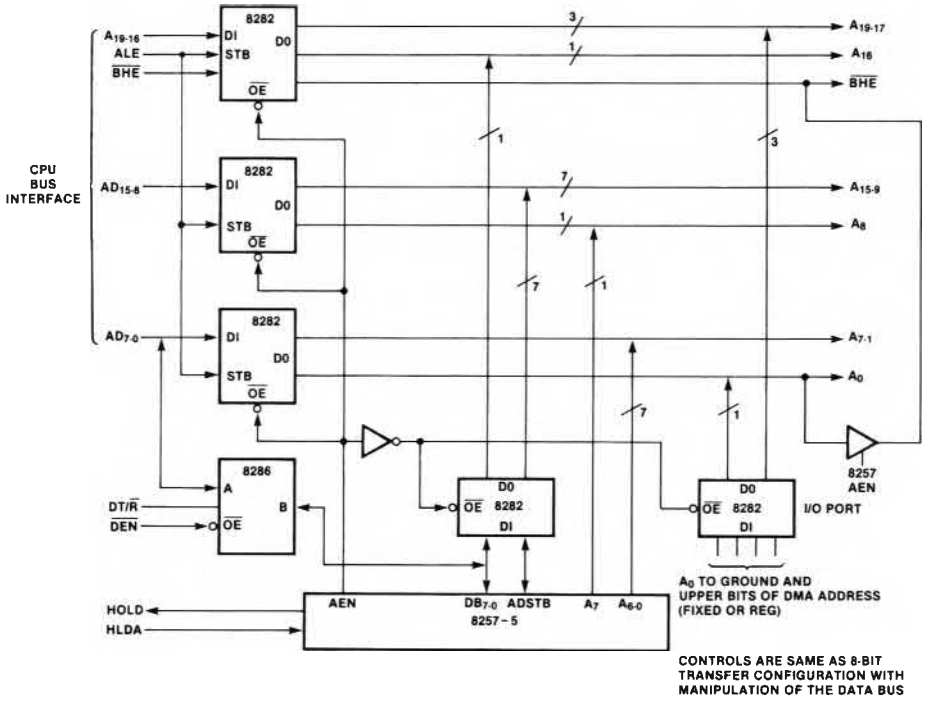


Figure 3G3. 8086 Min System, 8257 on System Bus 16-Bit Transfers

which three-states the data bus transceivers. $\overline{AEN}$ must also three-state the address latch (8282 or 8283) outputs. These actions remove the 8086 from the system bus and allow the requesting device to drive the system bus. The $\overline{AEN}$ signal to the 8284 disables the ready input and forces a bus cycle initiated by the 8086 to wait until the 8086 regains control of the system bus. The CPU may actively drive its local bus during this interval.

The requesting device will not gain control of the bus during an 8086 initiated bus cycle, a locked instruction or an interrupt acknowledge cycle. The $\overline{LOCK}$ signal from the 8086 is active between $\overline{INTA}$ cycles to guarantee the CPU maintains control of the bus. Unlike the minimum mode 8086 HOLD response, this arbitration circuit allows the requestor to gain control of the bus between consecutive bus cycles which transfer a word operand on an odd address boundary and are not locked. Depending on the characteristics of the requesting device, any of the 74LS74 outputs can be used to generate a HLDA to the device.

Upon completion of its bus operations, the alternate bus master must relinquish control of the system bus and drop the HOLD request. After $\overline{AEN}$ goes inactive, the address latches and data transceivers are enabled but, if a CPU initiated bus cycle is pending, the 8288 will not drive the command bus until a minimum of 105 ns or maximum of 275 ns later. If the system is normally not ready, the 8284 $\overline{AEN}$ input may immediately be enabled with ready returning to the CPU when the selected device completes the transfer. If the system is normally ready, the 8284 $\overline{AEN}$ input must be delayed long enough to provide access time equivalent to a normal bus cycle. The 74LS74 latches in the design provide a minimum of TCLCHmin for the alternate device to float the system bus after releasing HOLD. They also provide 2TCLCL ns address access and 2TCLCL – TAEVCHmax ns (8288 command enable delay) command access prior to enabling 8284 ready detection. If HLDA is generated as shown in Figure 3G4, TCLCL ns are available for the 8086 to release the bus prior to issuing HLDA while HLDA is dropped almost immediately upon loss of HOLD.

A circuit configuration for an 8257-5 using this technique to interface with a maximum mode 8086 can be derived from Figure 3G3. The 8257-5 has its own address latch for buffering the address lines A15-A8 and uses its $\overline{AEN}$ output to enable the latch onto the address bus. The maximum latency from HOLD to HLDA for this circuit is dependent on the state of the system when the HOLD is issued. For an idle system the maximum delay is the propagation delay through the nand gate and R/S flip-flop (TD1) plus 2TCLCL plus TCLCHmax plus propagation delay of the 74LS74 and 74LS02 (TD2). For a locked instruction it becomes: TD1 + TD2 + (M + 2) *TCLCL + TCLCHmax where M is the number of clocks required for execution of the locked instruction. For the interrupt acknowledge cycle the latency is TD1 + TD2 + 9 *TCLCL + TCLCHmax.

2.2 Shared Local Bus ($\overline{RQ}/\overline{GT}$ Usage)

The $\overline{RQ}/\overline{GT}$ protocol was developed to allow up to two instruction set extension processors (co-processors) or other special function processors (like the 8089 I/O processor in local mode) to reside directly on the 8086 local bus. Each $\overline{RQ}/\overline{GT}$ pin of the 8086 supports the full protocol for exchange of bus control (Fig. 3G5). The sequence consists of a request from the alternate bus master to gain control of the system bus, a grant from the CPU to indicate the bus has been relinquished and a release pulse from the alternate master when done. The two $\overline{RQ}/\overline{GT}$ pins ($\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$) are prioritized with $\overline{RQ}/\overline{GT0}$ having the highest priority. The prioritization only occurs if requests have been received on both pins before a response has been given to either. For example, if a request is received on $\overline{RQ}/\overline{GT1}$ followed by a request on $\overline{RQ}/\overline{GT0}$ prior to a grant on $\overline{RQ}/\overline{GT1}$, $\overline{RQ}/\overline{GT0}$ will gain priority over $\overline{RQ}/\overline{GT1}$. However, if $\overline{RQ}/\overline{GT1}$ had already received a grant, a request on $\overline{RQ}/\overline{GT0}$ must wait until a release pulse is received on $\overline{RQ}/\overline{GT1}$.

The request/grant sequence interaction with the bus interface unit is similar to HOLD/HLDA. The CPU continues to execute until a bus transfer for additional instructions or data is required. If the release pulse is

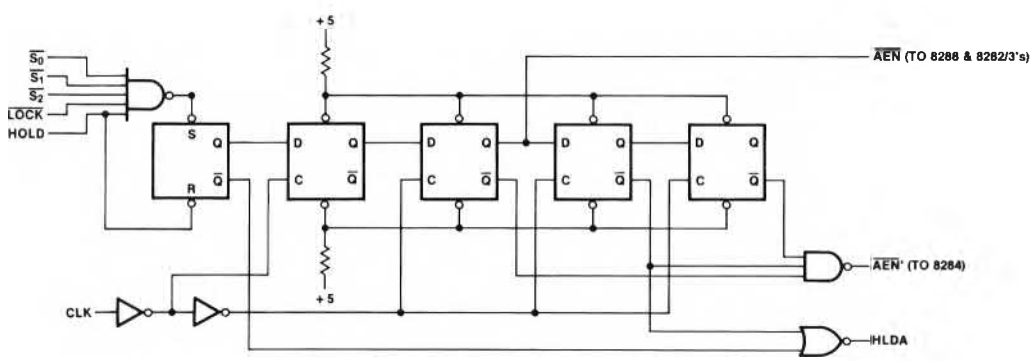


Figure 3G4. Circuit to Translate HOLD into AEN Disable for Max Mode 8086

received before the CPU needs the bus, it will not drive the bus until a transfer is required.

Upon receipt of a request pulse, the 8086 floats the multiplexed address, data and status bus, the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status lines, the $\overline{LOCK}$ pin and $\overline{RD}$. This action does not disable the 8288 command outputs from driving the command bus and does not disable the address latches from driving the address bus. The 8288 contains internal pull-up resistors on the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status lines to maintain the passive state while the 8086 outputs are three-state. The passive state prevents the 8288 from initiating any commands or activating DEN to enable the transceivers buffering the data bus. If the device issuing the $\overline{RQ}$ does not use the 8288, it must disable the 8288 command outputs by disabling the 8288 $\overline{AEN}$ input. Also, address latches not used by the requesting device must be disabled.

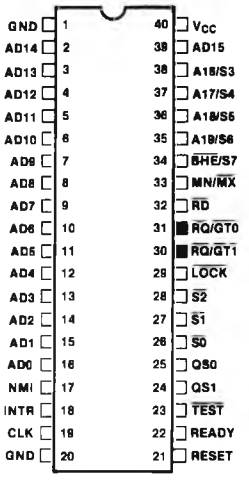


Figure 3G5. 8086 RQ/GT Connections

2.3 $\overline{RQ}/\overline{GT}$ Operation

Detailed timing of the $\overline{RQ}/\overline{GT}$ sequence is given in Figure 3G6. To request a transfer of bus control via the $\overline{RQ}/\overline{GT}$ lines, the device must drive the line low for no more than one CPU clock interval to generate a request pulse. The pulse must be synchronized with the CPU clock to guarantee the appropriate setup and hold times to the clock edge which samples the $\overline{RQ}/\overline{GT}$ lines in the CPU. After issuing a request pulse, the device must begin sampling for a grant pulse with the next low to high clock edge. Since the 8086 can respond with a grant pulse in the clock cycle immediately following the request, the $\overline{RQ}/\overline{GT}$ line may not return to the positive level between the request and grant pulses. Therefore edge triggered logic is not valid for capturing a grant pulse. It also implies the circuitry which generates the request pulse must guarantee the request is removed in time to detect a grant from the CPU. After receiving the grant pulse, the requesting device may drive the local bus. Since the 8086 does not float the address and data bus, $\overline{LOCK}$ or $\overline{RD}$ until the high to low clock transition following the low to high clock transition the requestor uses to sample for the grant, the requestor should wait the float delay of the 8086 (TCLAZ) before driving the local bus. This precaution prevents bus contention during the access of bus control by the requestor.

To return control of the bus to the 8086, the alternate bus master relinquishes bus control and issues a release pulse on the same $\overline{RQ}/\overline{GT}$ line. The 8086 may drive the $\overline{S0}$-$\overline{S2}$ status lines, $\overline{RD}$ and $\overline{LOCK}$, three clock cycles after detecting the release pulse and the address/data bus TCHCLmin ns (clock high time) after the status lines. The alternate bus master should be three-stated off the local bus and have other 8086 interface circuits (8288 and address latches) re-enabled within the 8086 delay to regain control of the bus.

2.4 $\overline{RQ}/\overline{GT}$ Latency

The $\overline{RQ}$ to $\overline{GT}$ latency for a single $\overline{RQ}/\overline{GT}$ line is similar to the HOLD to HLDA latency. The cases given for the minimum mode 8086 also apply to the maximum mode. For each case the delay from $\overline{RQ}$ detection by the CPU to $\overline{GT}$ detection by the requestor is:

(HOLD to HLDA delay) – (THVCH + TCHCL + TCLHAV)

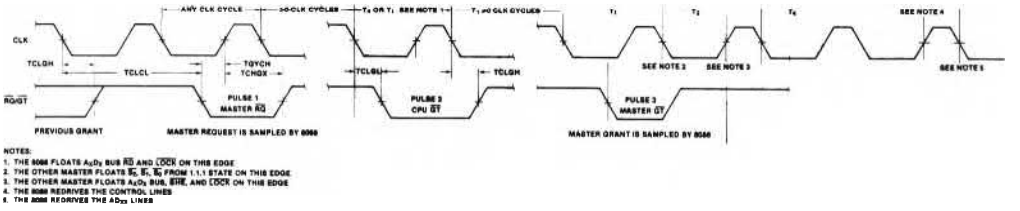


Figure 3G6. Request/Grant Sequence

This gives a clock cycle maximum delay for an idle bus interface. All other cases are the minimum mode result minus 476 ns. If the 8086 has previously issued a grant on one of the $\overline{RQ}/\overline{GT}$ lines, a request on the other $\overline{RQ}/\overline{GT}$ line will not receive a grant until the first device releases the interface with a release pulse on its $\overline{RQ}/\overline{GT}$ line. The delay from release on one $\overline{RQ}/\overline{GT}$ line to a grant on the other is typically one clock period as shown in Figure 3G7. Occasionally the delay from a release on $\overline{RQ}/\overline{GT1}$ to a grant on $\overline{RQ}/\overline{GT0}$ will take two clock cycles and is a function of a pending request for transfer of control from the execution unit. The latency from request to grant when the interface is under control of a bus master on the other $\overline{RQ}/\overline{GT}$ line is a function of the other bus master. The protocol embodies no mechanism for the CPU to force an alternate bus master off the bus. A watchdog timer should be used to prevent an errant alternate bus master from 'hanging' the system.

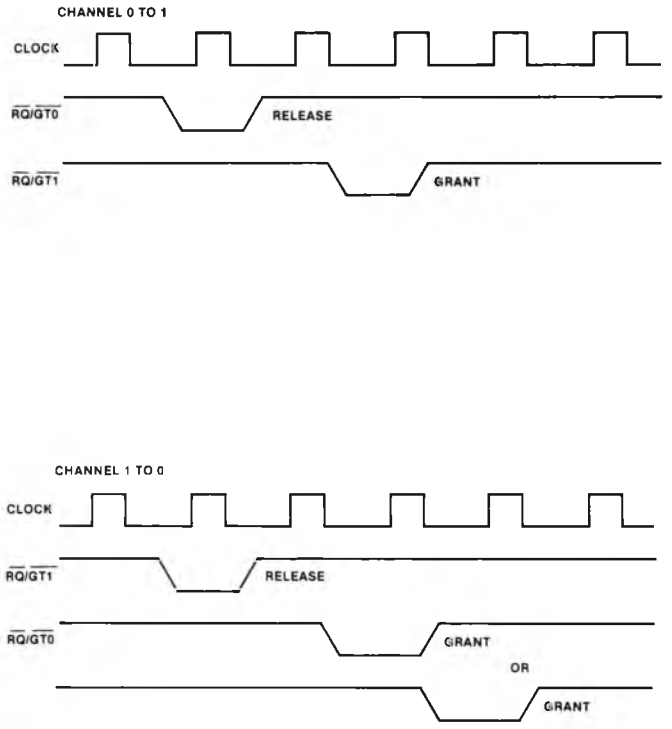


Figure 3G7. Channel Transfer Delay

quence into a RQ/GT pulse sequence is given in Figure 3G8. After receiving the grant pulse, the HLDA is enabled TCHCLmin ns before the CPU has three-stated the bus. If the requesting circuit drives the bus within 20 ns drops at the beginning of the release pulse to provide 2TCLCL + TCLCH for the requestor to relinquish control of the status lines and 3TCLCL to float the remaining signals.

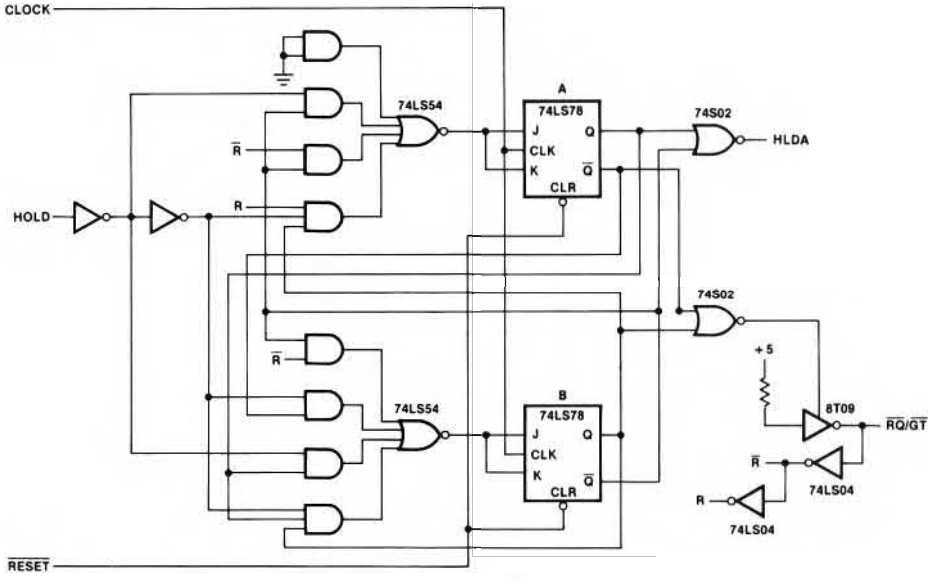


Figure 3G8a. HOLD/HLDA ⇔ $\overline{RQ}/\overline{GT}$ Conversion Circuit

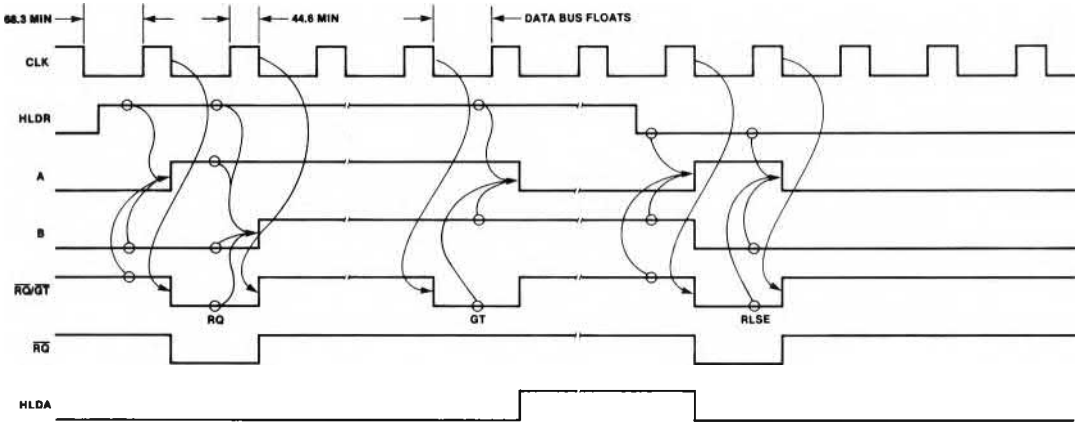


Figure 3G8b. HOLD/HLDA ⇔ $\overline{RQ}/\overline{GT}$ Conversion Timing

4. INTERFACING WITH I/O

The 8086 is capable of interfacing with 8- and 16-bit I/O devices using either I/O instructions or memory mapped I/O. The I/O instructions allow the I/O devices to reside in a separate I/O address space while memory mapped I/O allows the full power of the instruction set to be used for I/O operations. Up to 64K bytes of I/O mapped I/O may be defined in an 8086 system. To the programmer, the separate I/O address space is only accessible with INPUT and OUTPUT commands which transfer data between I/O devices and the AX (for 16-bit data transfers) or AL (for 8-bit data transfers) register. The first 256 bytes of the I/O space (0 to 255) are directly addressable by the I/O instructions while the entire 64K is accessible via register indirect addressing through the DX register. The later technique is particularly desirable for service procedures that handle more than one device by allowing the desired device address to be passed to the procedure as a parameter. I/O devices may be connected to the local CPU bus or the buffered system bus.

4A. Eight-Bit I/O

Eight-bit I/O devices may be connected to either the upper or lower half of the data bus. Assigning an equal number of devices to the upper and lower halves of the bus will distribute the bus loading. If a device is connected to the upper half of the data bus, all I/O addresses assigned to the device must be odd (A0 = 1). If the device is on the lower half of the bus, its addresses must be even (A0 = 0). The address assignment directs the eight-bit transfer to the upper (odd byte address) or lower (even byte address) half of the sixteen-bit data bus. Since A0 will always be a one or zero for a specific device, A0 cannot be used as an address input to select registers within a specific device. If a device on the upper half of the bus and one on the lower half are assigned addresses that differ only in A0 (adjacent odd and even addresses), A0 and $\overline{BHE}$ must be conditions of chip select decode to prevent a write to one device from erroneously performing a write to the other. Several techniques for generating I/O device chip selects are given in Figure 4A1.

The first technique (a) uses separate 8205's to generate chip selects for odd and even addressed byte peripherals. If a word transfer is performed to an even addressed device, the adjacent odd addressed I/O device is also selected. This allows accessing the devices individually with byte transfers or simultaneously as a 16-bit device with word transfers. Figure 4A1(b) restricts the chip selects to byte transfers, however a word transfer to an odd address will cause the 8086 to run two byte transfers that the decode technique will not detect. The third technique simply uses a single 8205 to generate odd and even device selects for byte transfers and will only select the even addressed eight-bit device on a word transfer to an even address.

If greater than 256 bytes of the I/O space or memory mapped I/O is used, additional decoding beyond what is shown in the examples may be necessary. This can be done with additional TTL, 8205's or bipolar PROMs (Intel's 3605A). The bipolar PROMs are slightly slower than multiple levels of TTL (50 ns vs 30 to 40 ns for TTL) but provide full decoding in a single package and allow inserting a new PROM to reconfigure the system I/O map without circuit board or wiring modifications (Fig. 4A2).

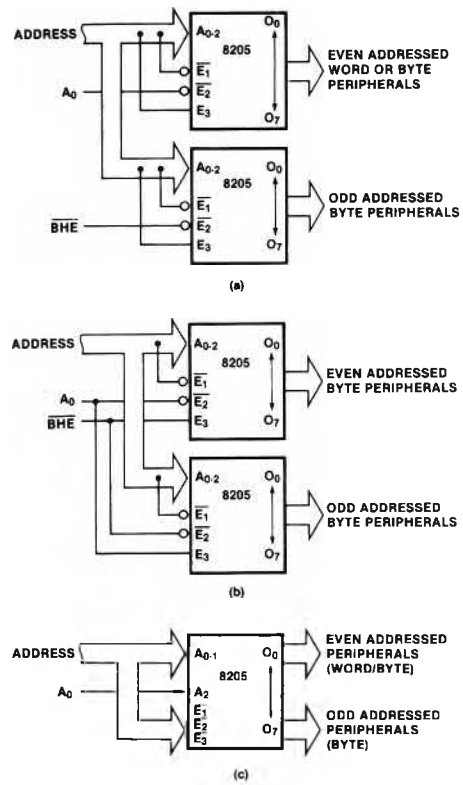


Figure 4A1. Techniques for I/O Device Chip Selects

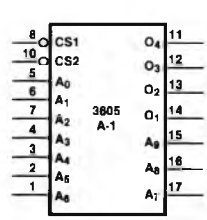


Figure 4A2. Bipolar PROM Decoder

One last technique for interfacing with eight-bit peripherals is considered in Figure 4A3. The sixteen-bit data bus is multiplexed onto an eight-bit bus to accommodate byte oriented DMA or block transfers to memory mapped eight-bit I/O. Devices connected to this interface may be assigned a sequence of odd and even addresses rather than all odd or even.

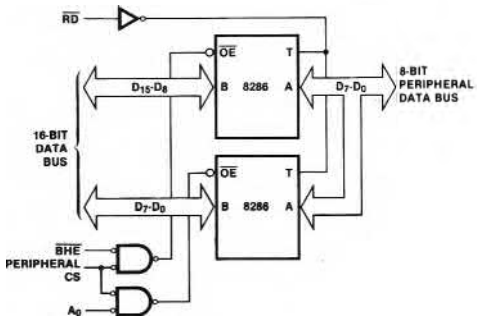
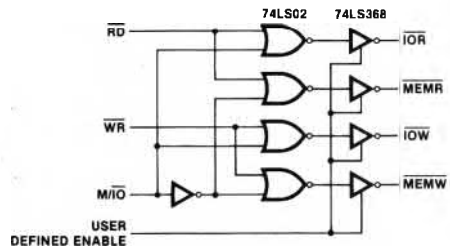


Figure 4A3. 16- to 8-Bit Bus Conversion



NOTE: IF IT IS NOT NECESSARY TO THREE-STATE THE COMMAND LINES, A DECODER (8205 OR 74S138) COULD BE USED. THE 74LS257 IS NOT RECOMMENDED SINCE THE OUTPUTS MAY EXPERIENCE VOLTAGE SPIKES WHEN ENTERING OR LEAVING THREE-STATE.

Figure 4C1. Decoding Memory and I/O $\overline{RD}$ and $\overline{WR}$ Commands for Minimum Mode 8086 Systems

4B. Sixteen-Bit I/O

For obvious reasons of efficient bus utilization and simplicity of device selection, sixteen-bit I/O devices should be assigned even addresses. To guarantee the device is selected only for word operations, A0 and $\overline{BHE}$ should be conditions of chip select code (Fig. 4B1).

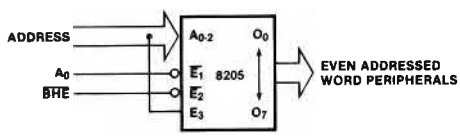


Figure 4B1. Sixteen-Bit I/O Decode

4C. General Design Considerations

MIN/MAX, MEMORY I/O MAPPED AND LINEAR SELECT

Since the minimum mode 8086 has common read and write commands for memory and I/O, if the memory and I/O address spaces overlap, the chip selects must be qualified by M/$\overline{IO}$ to determine which address space the devices are assigned to. This restriction on chip select decoding can be removed if the I/O and memory addresses in the system do not overlap and are properly decoded; all I/O is memory mapped; or $\overline{RD}$, $\overline{WR}$ and M/$\overline{IO}$ are decoded to provide separate memory and I/O read/write commands (Fig. 4C1). The 8288 bus controller in the maximum mode 8086 system generates separate I/O and memory commands in place of a M/$\overline{IO}$ signal. An I/O device is assigned to the I/O space or memory space (memory mapped I/O) by connection of either I/O or memory command lines to the command inputs of the device. To allow overlap of the memory and I/O address space, the device must not respond to chip select alone but must require a combination of chip select and a read or write command.

Linear select techniques (Fig. 4C2) for I/O devices can only be used with devices that either reside in the I/O address space or require more than one active chip select (at least one low active and one high active). Devices with a single chip select input cannot use linear select if they are memory mapped. This is due to the assignment of memory address space FFFFF0H-FFFFFFH to reset startup and memory space 00000H-003FFH to interrupt vectors.

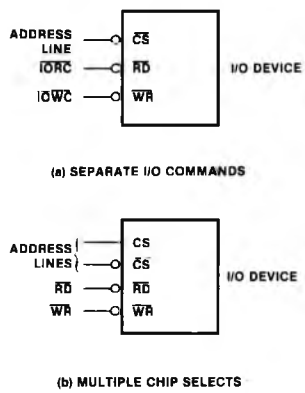


Figure 4C2. Linear Select for I/O

4D. Determining I/O Device Compatibility

This section presents a set of A.C. characteristics which represent the timing of the asynchronous bus interface of the 8086. The equations are expressed in terms of the CPU clock (when applicable) and are derived for minimum and maximum modes of the 8086. They represent the bus characteristics at the CPU.

The results can be used to determine I/O device requirements for operation on a single CPU local bus or buffered system bus. These values are not applicable to

a Multibus system bus interface. The requirements for a Multibus system bus are available in the Multibus Interface specification.

A list of bus parameters, their definition and how they relate to the A.C. characteristics of Intel peripherals are given in Table 4D1. Cycle dependent values of the parameters are given in Table 4D2. For each equation, if more than one signal path is involved, the equation reflects the worst case path.

ex. TAVRL(address valid before read active) =
(1) Address from CPU to $\overline{RD}$ active
(or)
(2) ALE (to enable the address through the address latches) to $\overline{RD}$ active

The worst case delay path is (1).

For the maximum mode 8086 configurations, TAVWLA, TWLWHA and TWLCLA are relative to the advanced write signal while TAVWL, TWLWH and TWLCL are relative to the normal write signal.

TABLE 4D1. PARAMETERS FOR PERIPHERAL COMPATIBILITY

TAVRL — Address stable before RD leading edge	(TAR)
TRHAX — Address hold after RD trailing edge	(TRA)
TRLRH — Read pulse width	(TRR)
TRLDV — Read to data valid delay	(TRD)
TRHDZ — Read trailing edge to data floating	(TDF)
TAVDV — Address to valid data delay	(TAD)
TRLRL — Read cycle time	(TRCYC)
TAVWL — Address valid before write leading edge	(TAW)
TAVWLA — Address valid before advanced write	(TAW)
TWHAX — Address hold after write trailing edge	(TWA)
TWLWH — Write pulse width	(TWW)
TWLWHA — Advanced write pulse width	(TWW)
TDVWH — Data set up to write trailing edge	(TDW)
TWHDX — Data hold from write trailing edge	(TWD)
TWLCL — Write recovery time	(TRV)
TWLCLA — Advanced write recovery time	(TRV)
TSVRL — Chip select stable before RD leading edge	(TAR)
TRHSX — Chip select hold after RD trailing edge	(TRA)
TSLDV — Chip select to data valid delay	(TRD)
TSVWL — Chip select stable before WR leading edge	(TAW)
TWHSX — Chip select hold after WR trailing edge	(TWA)
TSVWLA — Chip select stable before advanced write	(TAW)

Symbols in parentheses are equivalent parameters specified for Intel peripherals.

In the given list of equations, TWHDXB is the data hold time from the trailing edge of write for the minimum mode with a buffered data bus. For this equation, TCVCTX cannot be a minimum for data hold and a maximum for write inactive. The maximum difference is 50 ns giving the result TCLCH-50. If the reader wishes to verify the equations or derive others, refer to Section 3F for assistance with interpreting the 8086 bus timing diagrams.

Figure 4D1 shows four representative configurations and the compatible Intel peripherals (including wait states if required) for each configuration are given in Table 4D3. Configuration 1 and 2 are minimum mode demultiplexed bus 8086 systems without (1) and with (2) data bus transceivers. Configurations 3 and 4 are maximum mode systems with one (3) and two (4) levels of address and data buffering. The last configuration is characteristic of a multi-board system with bus buffers on each board. The 5 MHz parameter values for these configurations are given in Table 4D4 and demonstrate the relaxed device requirements for even a large complex configuration. The analysis assumes all components are exhibiting the specified worst case parameter values and are under the corresponding temperature, voltage and capacitive load conditions. If the capacitive loading on the 8282/83 or 8286/87 is less than the maximum, graphs of delay vs. capacitive loading in the respective data sheets should be used to determine the appropriate delay values.

TABLE 4D2. CYCLE DEPENDENT PARAMETER REQUIREMENTS FOR PERIPHERALS

(a) Minimum Mode

TAVRL = TCLCL + TCLRLmin − TCLAVmax = TCLCL − 100
TRHAX = TCLCL − TCLRHmax + TCLLHmin = TCLCL − 150
TRLRH = 2TCLCL − 60 = 2TCLCL − 60
TRLDV = 2TCLCL − TCLRLmax − TDVCLmin = 2TCLCL − 195
TRHDZ = TRHAVmin = 155 ns
TAVDV = 3TCLCL − TDVCLmin − TCLAVmax = 3TCLCL − 140
TRLRL = 4TCLCL = 4TCLCL
TAVWL = TCLCL + TCVCTVmin − TCLAVmax = TCLCL − 100
TWHAX = TCLCL + TCLLHmin − TCVCTXmax = TCLCL − 110
TWLWH = 2TCLCL − 40 = 2TCLCL − 40
TDVWH = 2TCLCL + TCVCTXmin − TCLDVmax = 2TCLCL − 100
TWHDX = TWHDZmin = 89
TWLCL = 4TCLCL = 4TCLCL
TWHDXB = TCLCHmin + (− TCVCTXmax + TCVCTXmin) = TCLCHmin − 50

Note: Delays relative to chip select are a function of the chip select decode technique used and are equal to: equivalent delay from address − chip select decode delay.

(b) Maximum Mode

TAVRL = TCLCL + TCLMLmin − TCLAVmax = TCLCL − 100
TRHAX = TCLCL − TCLMHmax + TCLLHmin = TCLCL − 40
TRLRH = 2TCLCL − TCLMLmax + TCLMHmin = 2TCLCL − 25
TRLDV = 2TCLCL − TCLMLmax − TDVCLmin = 2TCLCL − 65
TRHDZ = TRHAVmin = 155
TAVDV = 3TCLCL − TDVCLmin − TCLAVmax = 3TCLCL − 140
TRLRL = 4TCLCL = 4TCLCL
TAVWLA = TAVRL = TCLCL − 100
TAVWL = TAVRL + TCLCL = 2TCLCL − 100
TWHAX = TRHAX = TCLCL − 40
TWLWHA = TRLRH = 2TCLCL − 25
TWLWH = TRLRH − TCLCL = TCLCL − 25
TDVWH = 2TCLCL + TCLMHmin − TCLDVmax = 2TCLCL − 100
TWHDX = TCLCHmin − TCLMHmax + TCHDZmin = TCLCHmin − 30
TWLCL = 3TCLCL = 3TCLCL
TWLCLA = 4TCLCL = 4TCLCL

TABLE 4D3. COMPATIBLE PERIPHERALS (5 MHz 8086)

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
8251A	✓	1W	✓	✓
8253-5	✓	1W	✓	✓
8255A-5	✓	1W	✓	✓
8257-5	✓	1W	✓	✓
8259A	✓	✓	✓	✓
8271	✓	1W	✓	✓
8273	✓	1W	✓	✓
8275	✓	1W	✓	✓
8279-5	✓	1W	✓	✓
8041A*	✓	1W	✓	✓
8741A	✓	1W	✓	✓
8291	✓	✓	✓	✓

*Includes other Intel peripherals based on the 8041A (i.e., 8292, 8294, 8295).

✓ implies full operation with no wait states.

W implies the number of wait states required.

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
TAVRL	70	72	70	58
TRHAX	57	27	169	141
TRLRH	340	320	375	347
TRLDV	205	150	305	261
TRHDZ	155	158	382	360
TAVDV	430	400	400	372
TRLRL	800	770	800	772
TAVWL	70	72	270	258
TAVWLA	—	—	70	58
TWHAX	97	67	169	141
TWLWH	360	340	175	147
TWLWHA	—	—	375	347
TDVWH	300	339	270	258
TWHDX	88	15	95	13
TWLCL	800	772	600	572
TWLCLA	—	—	800	772
TSVRL	52	54	52	40
TRHSX	50	50	171	143
TSLDV	412	382	382	354
TSVWL	52	54	252	240
TWHSX	90	90	171	143
TSVWLA	—	—	52	40

— Not applicable.

for additional delays from address latches and data transceivers in the configuration. Once the system configuration is selected, the system requirements can be determined at the peripheral interface and used to evaluate compatibility of the peripheral to the system. During this process, two areas must be considered. First, can the device operate at maximum bus bandwidth and if not, how many wait states are required. Second, are there any problems that cannot be resolved by wait states.

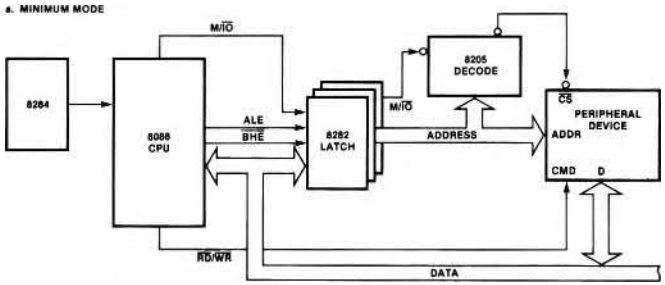
Examples of the first are TRLRH (read pulse width) and TRLDV (read access or $\overline{RD}$ active to output data valid). Consider address access time (valid address to valid data) for the maximum mode fully buffered configuration.

TAVDV = 3TCYC − 140 ns — address latch delay — address buffer delay — chip select decode delay — 2 transceiver delays

Assuming inverting latches, buffers and transceivers with 22 ns max delays (8283, 8287) and a bipolar PROM decode with 50 ns delay, the result is:

TAVDV = 322 ns @ 5 MHz

a. MINIMUM MODE



b. MINIMUM MODE BUFFERED DATA AND COMMAND BUSSES

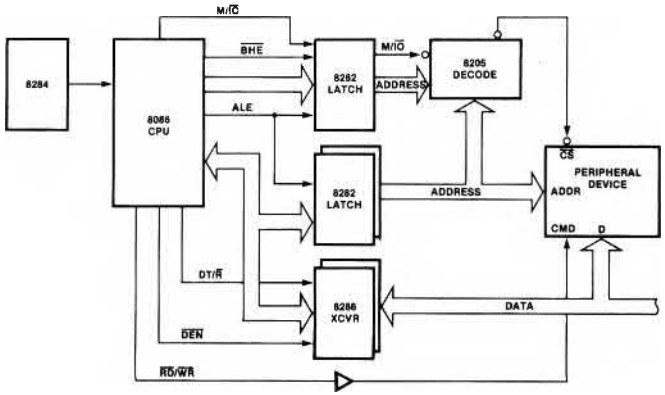
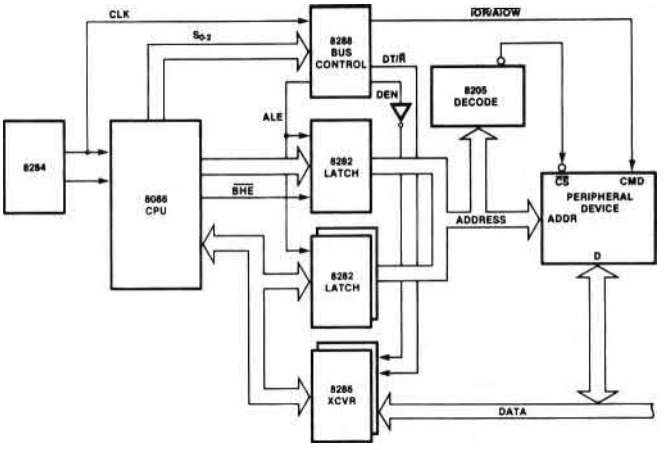


Figure 4D1. 8086 System Configurations

c. MAXIMUM MODE BUFFERED DATA BUS



NOTE: FOR OPTIMUM PERFORMANCE WITH INTEL PERIPHERALS, $\overline{AIOW}$ (ADVANCED WRITE) SHOULD BE USED.

d. MAXIMUM MODE DOUBLE BUFFERED SYSTEM

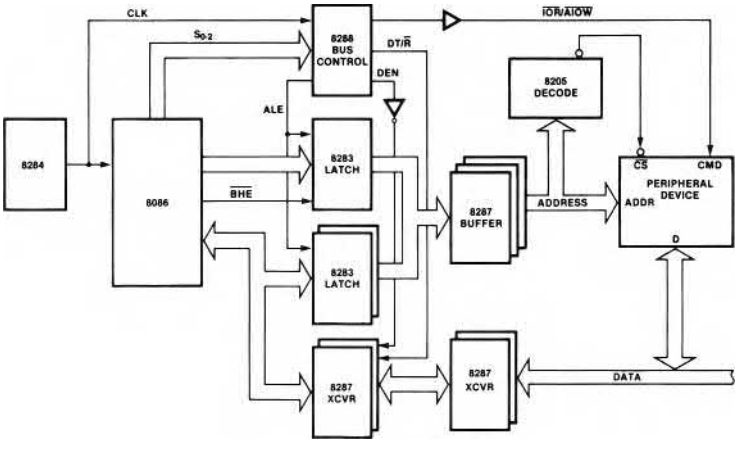


Figure 4D1. 8086 System Configurations (Con't)

The result gives the address to data valid delay required at the peripheral (in this configuration) to satisfy zero wait state CPU access time. If the maximum delay specified for the peripheral is less than the result, this parameter is compatible with zero wait state CPU operation. If not, wait states must be inserted until TAVDV + n * TCYC (n is the number of wait states) is greater than the peripherals maximum delay. If several parameters require wait states, either the largest number required should always be used or different transfer cycles can insert the maximum number required for that cycle.

The second area of concern includes TAVRL (address set up to read) and TWHDX (data hold after write). Incompatibilities in this area cannot be resolved by the insertion of wait states and may require either additional hardware, slowing down the CPU (if the parameter is related to the clock) or not using the device.

As an example consider address valid prior to advanced write low (TAVWLA) for the maximum mode fully buffered system.

TAVWLA = TCYC − 100 ns — address latch delay — address buffer delay — chip select decode delay + write buffer delay (minimum)

Assuming inverting latches and buffers with 22 ns delay (8283, 8287) and an 8205 address decoder with 18 ns delay

TAVWLA = 38 ns which is the time a 5 MHz 8086 system provides

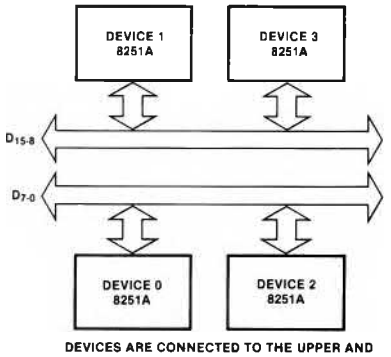
4E. I/O Examples

1. Consider an interrupt driven procedure for handling multiple communication lines. On receiving an interrupt from one of the lines, the invoked procedure polls the lines (reading the status of each) to determine which line to service. The procedure does not enable lines but simply services input and output requests until the associated output buffer is empty (for output requests) or until an input line is terminated (for the example only EOT is considered). On detection of the terminate condition, the routine will disable the line. It is assumed that other routines will fill a lines output buffer and enable the device to request output or empty the input buffer and enable the device to input additional characters.

The routine begins operation by loading CX with a count of the number of lines in the system and DX with the I/O address of the first line. The I/O addresses are assigned as shown in Figure 4E1 with 8251A's as the I/O devices. The status of each line is read to determine if it needs service. If yes, the appropriate routine is called to input or output a character. After servicing the line or if no service is needed, CX is decremented and DX is incremented to test the next line. After all lines have been tested and serviced, the routine terminates. If all interrupts from the lines are OR'd together, only one interrupt is used for all lines. If the interrupt is input to the CPU through an 8259A interrupt controller, the 8259A should be programmed in the level triggered mode to guarantee all line interrupts are serviced.

To service either an input or output request, the called routine transfers DX to BX, and shifts BX to form the offset for this device into the table of input or output buffers. The first entry in the buffer is an index to the next character position in the buffer and is loaded into the SI register. By specifying the base address of the table of buffers as a displacement into the data segment, the base + index + displacement addressing mode allows direct access to the appropriate memory location. 8086 code for part of this example is shown in Figure 4E2.

2. As a second example, consider using memory mapped I/O and the 8086 string primative instructions to perform block transfers between memory and I/O. By assigning a block of the memory address space (equivalent in size to the maximum block to be transferred to the I/O device) and decoding this address space to generate the I/O device's chip select, the block transfer capability is easily implemented. Figure 4E3 gives an interconnect for 16-bit I/O devices while Figure 4E4 incorporates the 16-bit bus to 8-bit bus multiplexing scheme to support 8-bit I/O devices. A code example to perform such a transfer is shown in Figure 4E5.



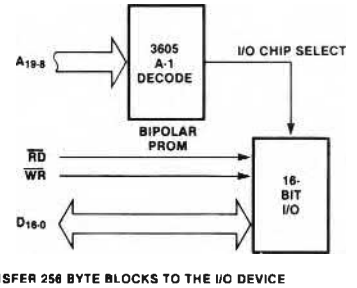
DEVICES ARE CONNECTED TO THE UPPER AND LOWER HALVES OF THE DATA BUS.

ADDRESS		
0	DEVICE 0	DATA
1	DEVICE 1	DATA
2	DEVICE 0	CONTROL/STATUS
3	DEVICE 1	CONTROL/STATUS
4	DEVICE 2	DATA
5	DEVICE 3	DATA
6	DEVICE 2	CONTROL/STATUS
7	DEVICE 3	CONTROL/STATUS
ETC.	"	"

Figure 4E1. Device Assignment

```
; THIS CODE DEMONSTRATES TESTING DEVICE
; STATUS FOR SERVICE, CONSTRUCTING THE
; APPROPRIATE LINE BUFFER ADDRESS FOR INPUT
; AND OUTPUT AND SERVICING AN INPUT
; REQUEST

            MASK EQU OFFFDH
CHECK_STATUS: INPUT  AL, DX          ; GET 8251A STATUS.
              MOV    AH, AL
              TEST   AH, READ_OR_WRITE_STATUS
              JZ     NEXT_IO
              CALL   ADDRESS
              TEST   AH, READ STATUS
              JZ     WRITE_SERVICE
              CALL   READ
              TEST   AH, WRITE STATUS
              JZ     NEXT_IO
WRITE_SERVICE: CALL  WRITE
NEXT_IO:      DEC    CX              ; TEST IF DONE.
              JNC    EXIT            ; YES, RESTORE & RETURN.
              AND    DX, MASK        ; REMOVE A1 AND
              ADD    DX, 3           ; INCREMENT ADDRESS.
              OR     DX, 2           ; SELECT STATUS FOR
              JMP    CHECK_STATUS    ; NEXT INPUT.

ADDRESS:      AND    DX, MASK        ; SELECT DATA.
              MOV    BH, DL          ; CONSTRUCT BUFFER
              INC    BH              ; DISPLACEMENT FOR
              SHR    BH              ; THIS DEVICE.
              XOR    BL, BL          ; BX IS THE DISPLACEMENT.
              RET

READ:         INPUT AL, DX                      ; READ CHARACTER.
              MOV SI, READ_BUFFERS [BX]         ; GET CHARACTER POINTER.
              MOV READ_BUFFERS [BX + SI], AL    ; STORE CHARACTER.
              INC READ_BUFFERS [BX]             ; INCR CHARACTER POINTER.
              CMP AL, EOT                       ; END OF TRANSMISSION?
              JNZ CONT_READ
              CALL DISABLE READ                 ; YES, DISABLE RECEIVER.
              CONT_READ:  RET                   ; SEND MESSAGE THAT INPUT
                                                ; IS READY.
```

Figure 4E2.

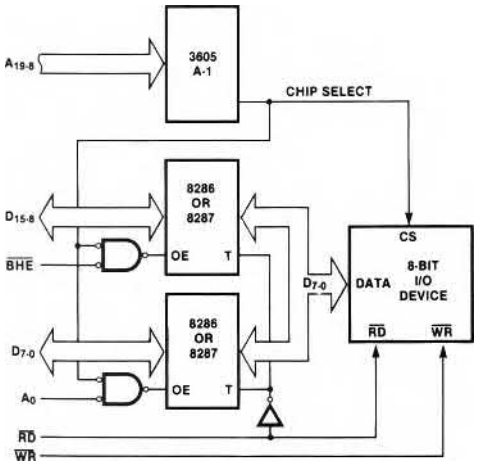


TRANSFER 256 BYTE BLOCKS TO THE I/O DEVICE

THE ADDRESS SPACE ASSIGNED TO THE I/O DEVICE IS

MEMORY DATA NEED NOT BE ALIGNED TO EVEN ADDRESS BOUNDARIES
I/O TRANSFERS MUST BE WORD TRANSFERS TO EVEN ADDRESS BOUNDARIES

Figure 4E3. Block Transfer to 16-Bit I/O Using 8086 String Primatives



ADDRESS ASSIGNMENT SAME AS PREVIOUS EXAMPLE. 16-BIT BUS IS MULTIPLEXED ONTO AN 8-BIT PERIPHERAL BUS.

Figure 4E4. Block Transfer to 8-Bit I/O Using 8086 String Primatives

```
            ; DEFINE THE I/O ADDRESS SPACE
             I/O SEGMENT
             ORG BLOCK_ADDRESS
I/O_BLOCK:   DW 128 DUP (?)
             I/O ENDS

            ; ASSUME THE DATA IS FROM THE CURRENT
            ; DATA SEGMENT
             CLD                        ; DF = FORWARD
             LES DI, I/O_BLOCK_ADDRESS  ; I/O BLOCK ADDRESS
                                        ; CONTAINS THE ADDRESS
                                        ; OF I/O BLOCK
             MOV CX, BLOCK_LENGTH
             MOV SI, SOURCE_ADDRESS
             MOVS I/O BLOCK             ; PERFORM WORD TRANSFERS

            ; END CODE EXAMPLE
```

NOTE THE CODE IS CAPABLE OF PERFORMING BYTE TRANSFERS BY CHANGING THE I/O BLOCK DEFINITION FROM 128 WORD TO 256 BYTES

Figure 4E5. Code for Block Transfers

5. INTERFACING WITH MEMORIES

Figure 5.1 is a general block diagram of an 8086 memory. The basic characteristics of the diagram are the partitioning of the 16-bit word memory into high and low 8-bit banks on the upper and lower halves of the data bus and inclusion of $\overline{BHE}$ and A0 in the selection of the banks. Specific implementations depend on the type of memory and the system configuration.

5A. ROM and EPROM

The easiest devices to interface to the system are ROM and EPROM. Their byte format provides a simple bus interface and since they are read only devices, A0 and $\overline{BHE}$ need not be included in their chip enable/select decoding (chip enable is similar to chip select but additionally determines if the device is in active or standby power mode). The address lines connected to the devices start with A1 and continue up to the maximum number the device can accept, leaving the remaining address lines for chip enable/select decoding. To connect the devices directly to the multiplexed bus, they must have output enables. The output enable is also necessary to avoid bus contention in other configurations. Figure 5A1 shows the bus connections for ROM and EPROM memories. No special decode techniques are required for generating chip enables/selects. Each valid decode selects one device on the upper and lower halves of bus to allow byte and word access. Byte access is achieved by reading the full word onto the bus with the 8086 only accepting the desired byte. For the minimum mode 8086, if $\overline{RD}$, $\overline{WR}$ and M/$\overline{IO}$ are not decoded to form separate commands for memory and I/O, and the I/O space overlaps the memory space assigned to the EPROM/ROM then M/$\overline{IO}$ (high active) must be a condition of chip enable/select decode. The output enable is controlled by the system memory read signal.

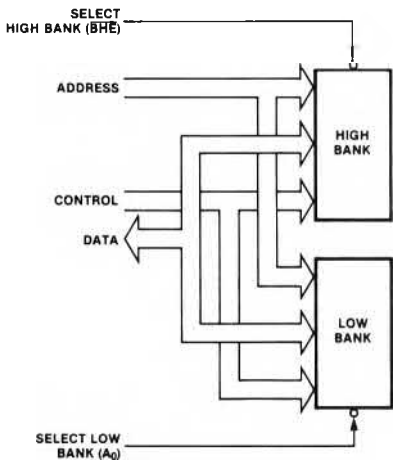
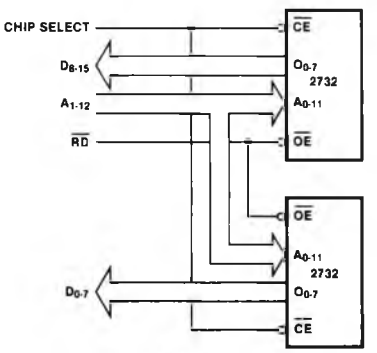


Figure 5.1. 8086 Memory Array



NOTE A_0 AND $\overline{BHE}$ ARE NOT USED.

Figure 5A1. EPROM/ROM Bus Interface

Static ROM's and EPROM's have only four parameters to evaluate when determining their compatibility to the system. The parameters, equations and evaluation techniques given in the I/O section are also applicable to these devices. The relationship of parameters is given in Table 5A1. TACC and TCE are related to the same equation and differ only by the delay associated with the chip enable/select decoder. As an example, consider a 2716 EPROM memory residing on the multiplexed bus of a minimum mode configuration:

TACC = 3TCLCL – 140 – address buffer delay = 430 ns
(8282 = 30 ns max delay)

TCE = TACC – decoder delay = 412 ns
(8205 decoder delay = 18 ns)

TOE = 2TCLCL – 195 = 205 ns

TDF = = 155 ns

TABLE 5A1. EPROM/ROM PARAMETERS

TOE — Output Enable to Valid Data = TRLDV
TACC — Address to Valid Data = TAVDV
TCE — Chip Enable to Valid Data = TSLDV
TDF — Output Enable High to Output Float = TRHDZ

The results are the times the system configuration requires of the component for full speed compatibility with the system. Comparing these times with 2716 parameter limits indicates the 2716-2 will work with no wait states while the 2716 will require one wait state. Table 5A2 demonstrates EPROM/ROM compatibility for the configurations presented in the I/O section. Before designing a ROM or EPROM memory system, refer to AP-30 for additional information on design techniques that give the system an upgrade path from 16K to 32K and 64K devices.

TABLE 5A2. COMPATIBLE EPROM/ROM (5 MHz 8086)

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
2716-1	✓	✓	✓	✓
2716-2	✓	1W	1W	1W
2732	1W	1W	1W	1W
2332	✓	✓	✓	✓
2364	✓	✓	✓	✓

5B. Static RAM

Interfacing static RAM to the system introduces several new requirements to the memory design. A0 and $\overline{BHE}$ must be included in the chip select/chip enable decoding of the devices and write timing must be considered in the compatibility analysis.

For each device, the data bus connections must be restricted to either the upper or lower half of the data bus. Devices like the 2114 or 2142 must not straddle the upper and lower halves of the data bus (Fig. 5B1). To allow selecting either the upper byte, lower byte or full 16-bit word for a write operation, $\overline{BHE}$ must be a condition of decode for selecting the upper byte and A0 must be a condition of decode for selecting the lower byte. Figure 5B2 gives several selection techniques for devices with single chip selects and no output enables (2114, 2141, 2147). Figure 5B3 gives selection techniques for devices with chip selects and output enables.

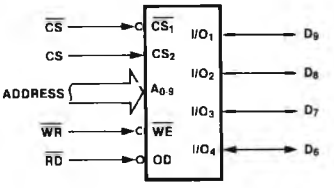


Figure 5B1. Incorrect Connection of 2142 Across Byte Boundaries

The first group requires inclusion of A0 and $\overline{BHE}$ to decode or enable the chip selects. Since these memories do not have output enables, read and write are used as enables for chip select generation to prevent bus contention. If read and write are not used to enable the chip selects, devices with common input/output pins (like the 2114) will be subjected to severe bus contention between chip select and write active. For devices with separate input/output lines (like 2141, 2147), the outputs can be externally buffered with the buffer enable controlled by read. This solution will only allow bus contention between memory devices in the array during chip select transition periods. These techniques are considered in more detail in Section 2C.

For devices with output enables (2142), write may be gated with $\overline{BHE}$ and A0 to provide upper and lower bank write strobes. This simplifies chip select decoding by eliminating $\overline{BHE}$ and A0 as a condition of decode. Although both devices are selected during a byte write operation, only one will receive a write strobe. No bus contention will exist during the write since a read command must be issued to enable the memory output drivers.

If multiple chip selects are available at the device, $\overline{BHE}$ and A0 may directly control device selection. This allows normal chip select decoding of the address space and direct connection of the read and write commands to the devices. Alternately, the multiple chip select inputs of the device could directly decode the address space (linear select) and be combined with the separate write strobe technique to minimize the control circuitry needed to generate chip selects.

As with the EPROM's and ROM's, if separate commands are not provided for memory and I/O in the minimum mode 8086 and the address spaces overlap, M/$\overline{IO}$ (high active) must be a condition of chip select decode. Also, the address lines connected to the memory devices must start with A1 rather than A0.

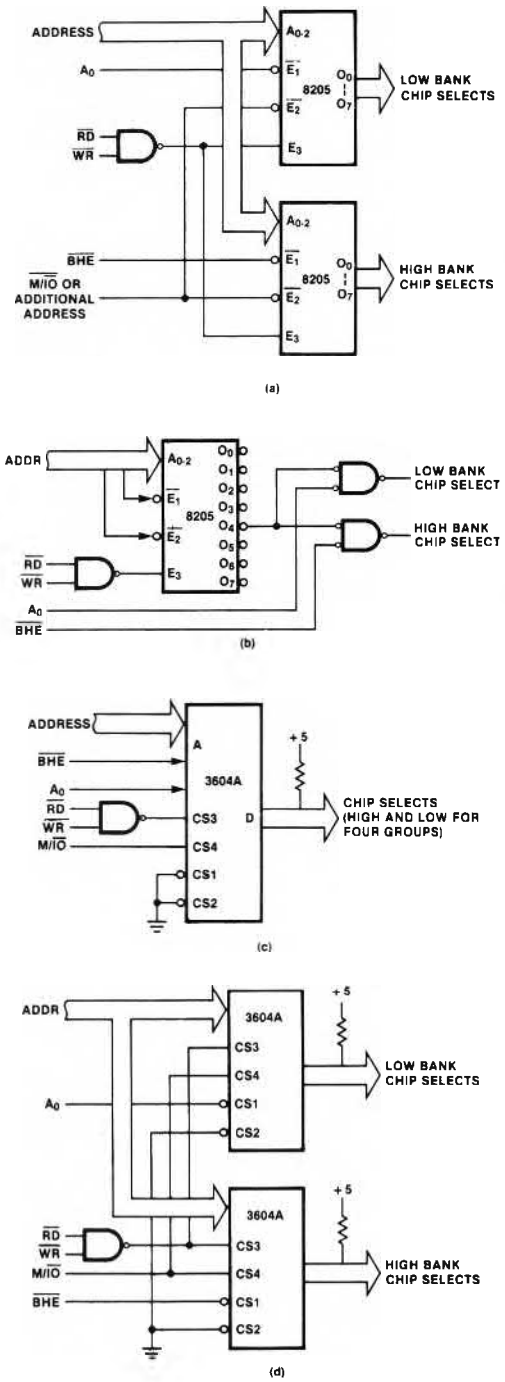


Figure 5B2. Generating Chip Selects for Devices without Output Enables

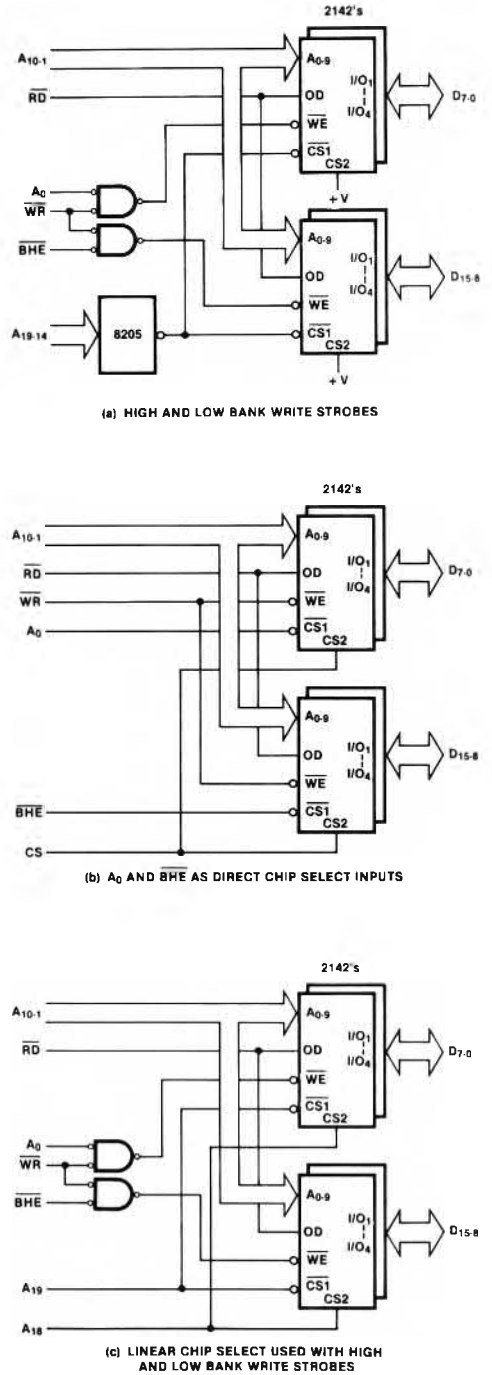


Figure 5B3. Chip Selection for Devices with Output Enables

For analysis of RAM compatibility, the write timing parameters listed in Table 5B1 may also need to be considered (depending on the RAM device being considered). The CPU clock relative timing is given in Table 5B2. The equations specify the device requirements at the CPU and provide a base for determining device requirements in other configurations. As an example consider the write timing requirements of a 2142 in a maximum mode buffered 8086 system (Figure 5B4). The 2142 write parameters that must be analyzed are TWA advanced write pulse width, TWR write release time, TDWA data to write time overlap and TDH data hold from write time.

TWA = 2TCLCL – TCLMLmax + TCLMHmin = 375 ns.
TWR = 2TCLCL – TCLMHmax + TCLLHmin + TSHOVmin = 170 ns.
TDWA = 2TCLCL – TCLDVmax + TCLMHmin – TIVOVmax = 265 ns.
TDH = TCLCH – TCLMHmax + TCHDXmin + TIVOVmin = 95 ns.

TABLE 5B1. TYPICAL WRITE TIMING PARAMETERS

TW — Write Pulse Width
TWR — Write Release (Address Hold From End of Write)
TDW — Data and Write Pulse Overlap
TDH — Data Hold From End of Write
TAW — Address Valid to End of Write
TCW — Chip Select to End of Write
TASW — Address Valid to Beginning of Write

TABLE 5B2. CYCLE DEPENDENT WRITE PARAMETERS FOR RAM MEMORIES

(a) Minimum Mode
TW = TWLWH = 2TCLCL – 60 = 340 ns
TWR = TCLCL – TCVCTXmax + TCLLHmin = 90 ns
TDW = 2TCLCL – TCLDVmax + TCVCTXmin = 300 ns
TDH = TWHDX = 88 ns
TAW = 3TCLCL – TCLAVmax + TCVCTXmin = 500 ns
TCW = TAW – Chip Select Decode
TASW = TCLCL – TCLAVmax + TCVCTXmin = 100 ns
(b) Maximum Mode
TW = TCLCL – TCLMLmax + TCLMHmin = 175 ns
TWR = TCLCL – TCLMHmax + TCLLHmin = 165 ns
TDW = TW = 175 ns
TDH = TCLCHmin – TCLMHmax + TCHDXmin = 93 ns
TAW = 3TCLCL – TCLAVmax + TCLMHmin = 500 ns
TCW = TAW – Chip Select Decode
TASW = 2TCLCL – TCLAVmax + TCLMLmin = 300 ns
TWA* = TW + TCLCL = 375 ns
TDWA* = 2TCLCL – TCLDVmax + TCLMHmin = 300 ns
TASWA* = TASW – TCLCL = 100 ns
*Relative to Advanced Write.

Comparing these results with the 2142 family indicates the standard 2142 write timing is fully compatible with this 8086 configuration. Read timing analysis is also necessary to completely determine compatibility of the devices.

5C. Dynamic RAM

Dynamic RAM is perhaps the most complex device to design into a system. To relieve the engineer of most of this burden, Intel provides the 8202 dynamic RAM controller as part of the 8086 family of peripheral devices. This section will discuss using the 8202 with the 8086 to build a dynamic memory system for an 8086 system. For additional information on the 8202, refer to the 8202 data sheet (9800873) and application note AP-45 Using the 8202 Dynamic RAM Controller (9800809A).

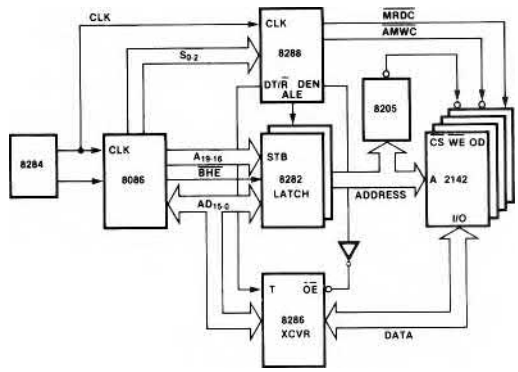


Figure 5B4. Sample Configuration for Compatibility Analysis Example

5.C.1 Standard 8086-8202 Interconnect

Figure 5.C.1.1 shows a standard interconnection for an 8202 into an 8086 system. The configuration accommodates 64K words (128K bytes) of dynamic RAM addressable as words or bytes. To access the RAM, the 8086 initiates a bus cycle with an address that selects the 8202 (via $\overline{PCS}$) and the appropriate transfer command ($\overline{MRDC}$ or $\overline{MWTC}$). If the 8202 is not performing a refresh cycle, the access starts immediately, otherwise, the 8086 must wait for completion of the refresh. $\overline{XACK}$ from the 8202 is connected to the 8284 RDY input to force the CPU to wait until the RAM cycle is completed before the CPU can terminate the bus cycle. This effectively synchronizes the asynchronous events of refresh and CPU bus cycles. The normal write command ($\overline{MWTC}$) is used rather than the advanced command ($\overline{AMWC}$) to guarantee the data is valid at the dynamic RAMS before the write command is issued. The gating of $\overline{WE}$ with A0 and $\overline{BHE}$ provides selective write strobes to the upper and lower banks of memory to allow byte and word write operations. The logic which generates the strobe for the data latches allows read data to propagate to the system as soon as the data is available and latches the data on the trailing edge of $\overline{CAS}$.

DETAILED TIMING

Read Cycle

For no wait state operation, the 8086 requires data to be valid from $\overline{MRDC}$ in:

2TCLCL – TCLML – TDVCL – buffer delays = 291 ns.

Since the 8202 is $\overline{CAS}$ access limited, we need only examine $\overline{CAS}$ access time. The 8202/2118 guarantees data valid from 8202 $\overline{RD}$ low to be:

(tph + 3tp + 100 ns) 8202 TCC delay + TCAC for the 2118

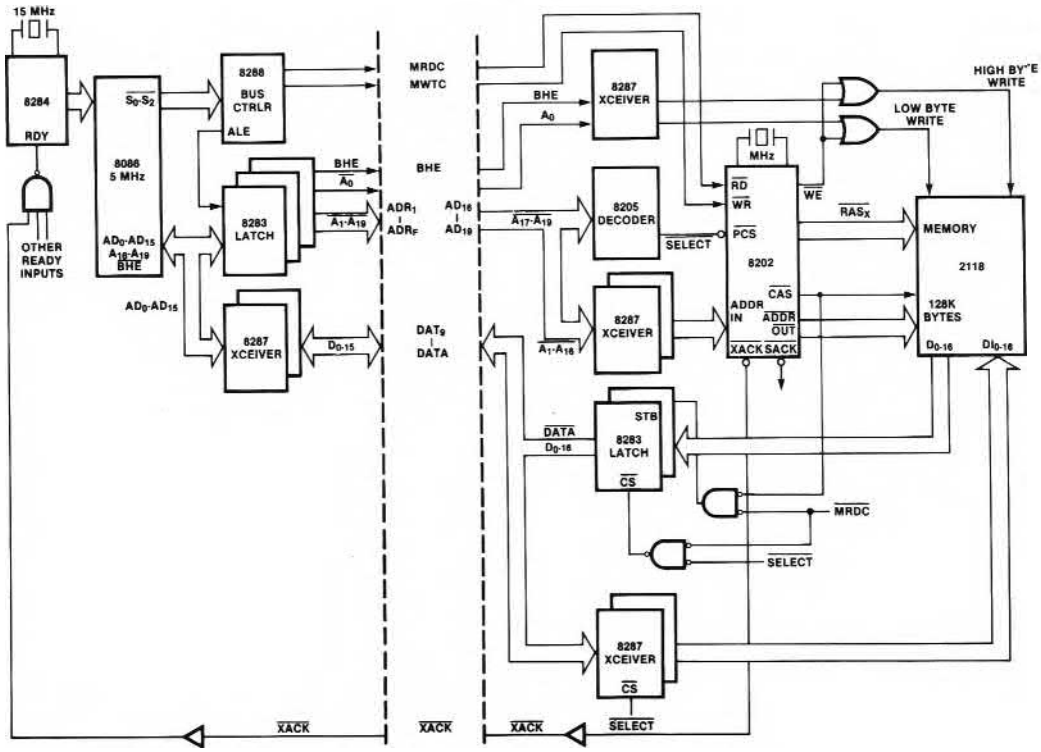


Figure 5C1.1. 5 MHz 8086/8202/128K Byte System — Double Data, Control and Address Buffering (Note: Bus driver on 8202 is not needed if less than 64K bytes are used)

For a 25 MHz 8202 and 2118-3, we get 297 ns which is insufficient for no wait state operation. If only 64K bytes are accessed, the 8202 requires only (tph + 3tp + 85 ns) giving 282 ns access and no wait states required. Refer to Figure 5.C.1.2 and 5C.1.3 for timing information on the 8202 and 2118.

Write Cycle

An important consideration for dynamic RAM write cycles is to guarantee data to the RAM is valid when both $\overline{CAS}$ and $\overline{WE}$ are active. For the 2118, if $\overline{WE}$ is valid prior to $\overline{CAS}$, the data setup is to $\overline{CAS}$ and if $\overline{CAS}$ is valid before $\overline{WE}$ (as would occur during a read modify write cycle) the data setup time is to $\overline{WE}$. For the 8202, the $\overline{WR}$ to $\overline{CAS}$ delay is analyzed to determine the data setup time to $\overline{CAS}$ inherently provided by the 8202 command to $\overline{RAS}$/$\overline{CAS}$ timing. The minimum delay from $\overline{WR}$ to $\overline{CAS}$ is:

TCCmin = tph + 2tp + 25 = 127 ns @ 25 MHz

Subtracting buffer delays and data setup at the 2118, we have 83 ns to generate valid data after the write command is issued by the CPU (in this case the 8288). Since the 8086 will not guarantee valid data until TCLAVmax – TCLMLmin = 100 ns from the advanced write signal, the normal write signal is used. The normal write $\overline{MWTC}$ guarantees data is valid 100 ns before it is active. The worst case write pulse width is approximately 175 ns which is sufficient for all 2118's.

Synchronization

To force the 8086 to wait during refresh the $\overline{XACK}$ or $\overline{SACK}$ lines must be returned to the 8284 ready input. The maximum delay from $\overline{RD}$ to $\overline{SACK}$ (if the 8202 is not performing refresh) is TAC = tp + 40 = 80 ns. To prevent a wait state at the 8086, RDY must be valid at the 8284 TCLCHmin – TCLMLmax – TR1VCLmax = 48 ns after the command is active. This implies that under worst case conditions, one wait state will be inserted for every read cycle. Since $\overline{MWTC}$ does not occur until one clock later, two wait states may be inserted for writes.

The $\overline{XACK}$ from command delay will assert RDY TCC + TCX = (tph + 3tp + 100) + (5tp + 20) = 460 ns after the command. This will typically insert one or two wait states.

Unless 2118-3's are used in 64K byte or less memories, $\overline{SACK}$ must not be used since it does not guarantee a wait state. From the previous access time analysis we saw that other configurations required a wait state.

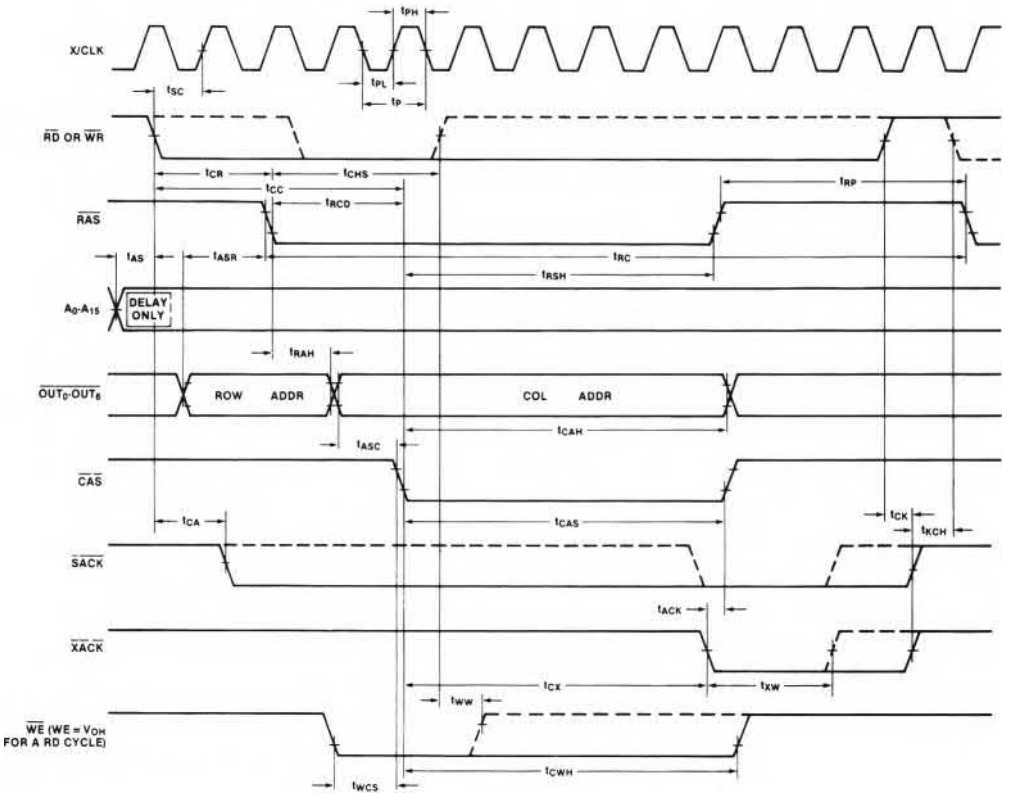


Figure 5C1.2. 8202 Timing Information

A.C. CHARACTERISTICS

$T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

Measurements made with respect to $RAS_1 - RAS_4$, CAS, WE, $OUT_0 - OUT_6$ are at 2.4V and 0.8V. All other pins are measured at 1.5V.

Loading:
64 Devices

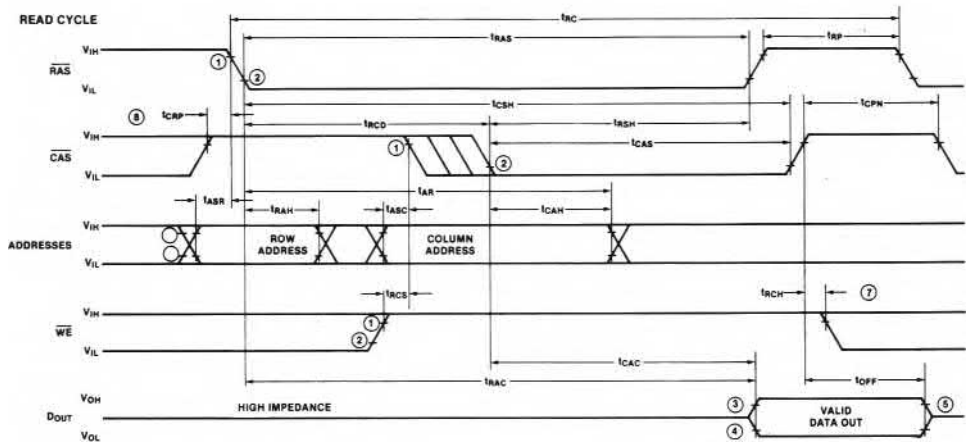
$\overline{SACK}$, $\overline{XACK}$	CL = 30 pF
$\overline{OUT_0} - \overline{OUT_6}$	CL = 320 pF
$\overline{RAS_1} - \overline{RAS_4}$	CL = 230 pF
$\overline{WE}$	CL = 450 pF
$\overline{CAS}$	CL = 640 pF

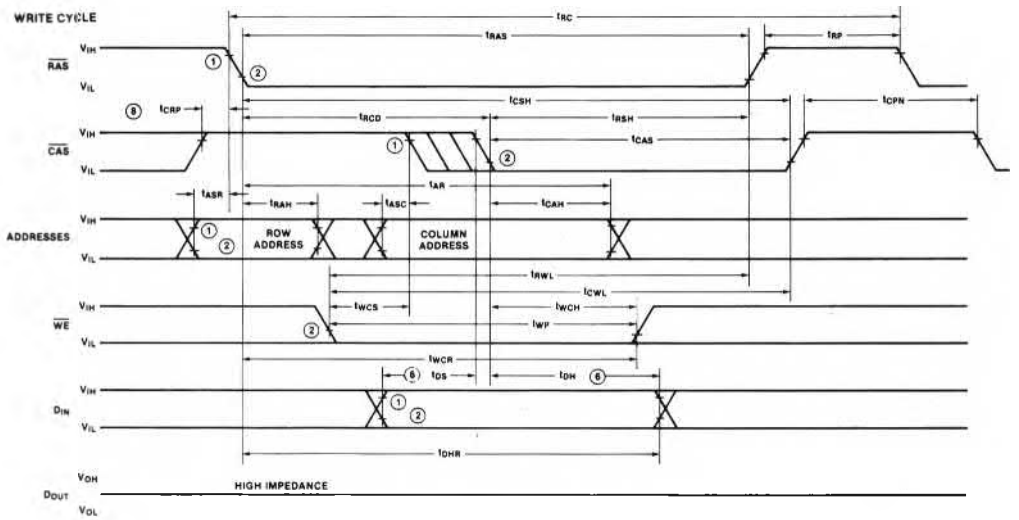
Symbol	Parameter	Min	Max	Units
t_P	Clock (Internal/External) Period (See Note 1)	40	54	ns
t_{RC}	Memory Cycle Time	$10\ t_P - 30$	$12\ t_P$	ns
t_{RAH}	Row Address Hold Time	$t_P - 10$		ns
t_{ASR}	Row Address Setup Time	t_{PH}		ns
t_{CAH}	Column Address Hold Time	$5\ t_P$		ns
t_{ASC}	Column Address Setup Time	$t_P - 35$		ns
t_{RCD}	$\overline{RAS}$ to $\overline{CAS}$ Delay Time	$2\ t_P - 10$	$2\ t_P + 45$	ns
t_{WCS}	$\overline{WE}$ Setup to $\overline{CAS}$	$t_P - 40$		ns
t_{RSH}	$\overline{RAS}$ Hold Time	$5\ t_P - 30$		ns
t_{CAS}	$\overline{CAS}$ Pulse Width	$5\ t_P - 30$		ns
t_{RP}	$\overline{RAS}$ Precharge Time (See Note 2)	$4\ t_P - 30$		ns
t_{WCH}	$\overline{WE}$ Hold Time to $\overline{CAS}$	$5\ t_P - 35$		ns
t_{REF}	Internally Generated Refresh to Refresh Time 64 Cycle 128 Cycle	 $548\ t_P$ $264\ t_P$	 $576\ t_P$ $288\ t_P$	 ns ns
t_{CR}	$\overline{RD}$, $\overline{WR}$ to $\overline{RAS}$ Delay	$t_{PH} + 30$	$t_{PH} + t_P + 75$	ns
t_{CC}	$\overline{RD}$, $\overline{WR}$ to $\overline{CAS}$ Delay	$t_{PH} + 2\ t_P + 25$	$t_{PH} + 3\ t_P + 100$	ns
t_{RFR}	REFRQ to $\overline{RAS}$ Delay	$1.5\ t_P + 30$	$2.5\ t_P + 100$	ns
t_{AS}	A_0—A_{15} to $\overline{RD}$, $\overline{WR}$ Setup Time (See Note 4)	0		ns
t_{CA}	$\overline{RD}$, $\overline{WR}$ to $\overline{SACK}$ Leading Edge		$t_P + 40$	ns
t_{CK}	$\overline{RD}$, $\overline{WR}$ to $\overline{XACK}$, $\overline{SACK}$ Trailing Edge Delay		30	ns
t_{KCH}	$\overline{RD}$, $\overline{WR}$ Inactive Hold to $\overline{SACK}$ Trailing Edge	10		ns
t_{SC}	$\overline{RD}$, $\overline{WR}$, $\overline{PCS}$ to X/CLK Setup Time (See Note 3)	15		ns
t_{CX}	$\overline{CAS}$ to $\overline{XACK}$ Time	$5\ t_P - 40$	$5\ t_P + 20$	ns
t_{ACK}	$\overline{XACK}$ Leading Edge to $\overline{CAS}$ Trailing Edge Time	10		ns
t_{XW}	$\overline{XACK}$ Pulse Width	$2\ t_P - 25$		ns
t_{LL}	REFRQ Pulse Width	20		ns
t_{CHS}	$\overline{RD}$, $\overline{WR}$, $\overline{PCS}$ Active Hold to $\overline{RAS}$	0		ns
t_{WW}	$\overline{WR}$ to $\overline{WE}$ Propagation Delay	8	50	ns
t_{AL}	S_1 to ALE Setup Time	40		ns
t_{LA}	S_1 to ALE Hold Time	$2\ t_P + 40$		ns
t_{PL}	External Clock Low Time	15		ns
t_{PH}	External Clock High Time	22		ns
t_{PH}	External Clock High Time for $V_{CC} = 5V \pm 5\%$	18		ns

Notes:

1. t_P minimum determines maximum oscillator frequency.
   t_P maximum determines minimum frequency to maintain 2 ms refresh rate and t_{RP} minimum.
2. To achieve the minimum time between the $\overline{RAS}$ of a memory cycle and the $\overline{RAS}$ of a refresh cycle, such as a transparent refresh, REFRQ should be pulsed in the previous memory cycle.
3. t_{SC} is not required for proper operation which is in agreement with the other specs, but can be used to synchronize external signals with X/CLK if it is desired.
4. If t_{AS} is less than 0 then the only impact is that t_{ASR} decreases by a corresponding amount.

Figure 5C1.2. 8202 Timing Information (Con't)





NOTES: 1,2. $V_{IH\ MIN}$ AND $V_{IL\ MAX}$ ARE REFERENCE LEVELS FOR MEASURING TIMING OF INPUT SIGNALS.
3,4. $V_{OH\ MIN}$ AND $V_{OL\ MAX}$ ARE REFERENCE LEVELS FOR MEASURING TIMING OF D_{OUT}.
5. t_{OFF} IS MEASURED TO $I_{OUT} < I_{OL}$.
6. t_{DS} AND t_{DH} ARE REFERENCED TO $\overline{CAS}$ OR $\overline{WE}$, WHICHEVER OCCURS LAST.
7. t_{RCH} IS REFERENCED TO THE TRAILING EDGE OF $\overline{CAS}$ OR $\overline{RAS}$, WHICHEVER OCCURS FIRST.
8. t_{CRP} REQUIREMENT IS ONLY APPLICABLE FOR $\overline{RAS}$/$\overline{CAS}$ CYCLES PRECEDED BY A $\overline{CAS}$ ONLY CYCLE (i.e., FOR SYSTEMS WHERE $\overline{CAS}$ HAS NOT BEEN DECODED WITH $\overline{RAS}$).

Figure 5C1.3. 2118 Family Timing

A.C. CHARACTERISTICS[1,2,3]

$T_A = 0°C$ to $70°C$, $V_{DD} = 5V \pm 10\%$, $V_{SS} = 0V$, unless otherwise noted.

READ, WRITE, READ-MODIFY-WRITE AND REFRESH CYCLES

Symbol	Parameter	2118-3		2118-4		2118-7		Unit	Notes
		Min.	Max.	Min.	Max.	Min.	Max.		
t_{RAC}	Access Time From $\overline{RAS}$		100		120		150	ns	4,5
t_{CAC}	Access Time from $\overline{CAS}$		55		65		80	ns	4,5,6
t_{REF}	Time Between Refresh		2		2		2	ms	
t_{RP}	$\overline{RAS}$ Precharge Time	110		120		135		ns	
t_{CPN}	$\overline{CAS}$ Precharge Time (non-page cycles)	50		55		70		ns	
t_{CRP}	$\overline{CAS}$ to $\overline{RAS}$ Precharge Time	0		0		0		ns	
t_{RCD}	$\overline{RAS}$ to $\overline{CAS}$ Delay Time	25	45	25	55	25	70	ns	7
t_{RSH}	$\overline{RAS}$ Hold Time	70		85		105		ns	
t_{CSH}	$\overline{CAS}$ Hold Time	100		120		165		ns	
t_{ASR}	Row Address Set-Up Time	0		0		0		ns	
t_{RAH}	Row Address Hold Time	15		15		15		ns	
t_{ASC}	Column Address Set-Up Time	0		0		0		ns	
t_{CAH}	Column Address Hold Time	15		15		20		ns	
t_{AR}	Column Address Hold Time to $\overline{RAS}$	60		70		90		ns	
t_T	Transition Time (Rise and Fall)	3	50	3	50	3	50	ns	8
t_{OFF}	Output Buffer Turn Off Delay	0	45	0	50	0	60	ns	
READ AND REFRESH CYCLES									
T_{RC}	Random Read Cycle Time	235		270		320		ns	
t_{RAS}	$\overline{RAS}$ Pulse Width	115	10000	140	10000	175	10000	ns	
t_{CAS}	$\overline{CAS}$ Pulse Width	55	10000	65	10000	95	10000	ns	
t_{RCS}	Read Command Set-Up Time	0		0		0		ns	
t_{RCH}	Read Command Hold Time	0		0		0		ns	
WRITE CYCLE									
t_{RC}	Random Write Cycle Time	235		270		320		ns	
t_{RAS}	$\overline{RAS}$ Pulse Width	115	10000	140	10000	175	10000	ns	
t_{CAS}	$\overline{CAS}$ Pulse Width	55	10000	65	10000	95	10000	ns	
t_{WCS}	Write Command Set-Up Time	0		0		0		ns	9
t_{WCH}	Write Command Hold Time	25		30		45		ns	
t_{WCR}	Write Command Hold Time, to $\overline{RAS}$	70		85		115		ns	
t_{WP}	Write Command Pulse Width	25		30		50		ns	
t_{RWL}	Write Command to $\overline{RAS}$ Lead Time	60		65		110		ns	
t_{CWL}	Write Command to $\overline{CAS}$ Lead Time	45		50		100		ns	
t_{DS}	Data-In Set-Up Time	0		0		0		ns	
t_{DH}	Data-In Hold Time	25		30		45		ns	
t_{DHR}	Data-In Hold Time, to $\overline{RAS}$	70		85		115		ns	
READ-MODIFY-WRITE CYCLE									
t_{RWC}	Read-Modify-Write Cycle Time	285		320		410		ns	
t_{RRW}	RMW Cycle $\overline{RAS}$ Pulse Width	165	10000	190	10000	265	10000	ns	
t_{CRW}	RMW Cycle $\overline{CAS}$ Pulse Width	105	10000	120	10000	185	10000	ns	
t_{RWD}	$\overline{RAS}$ to $\overline{WE}$ Delay	100		120		150		ns	9
t_{CWD}	$\overline{CAS}$ to $\overline{WE}$ Delay	55		65		80		ns	9

NOTES:
1. All voltages referenced to V_{SS}.
2. Eight cycles are required after power-up or prolonged periods (greater than 2 ms) of $\overline{RAS}$ inactivity before proper device operation is achieved. Any 8 cycles which perform refresh are adequate for this purpose.
3. A.C. Characteristics assume $t_T = 5$ ns.
4. Assume that $t_{RCD} \leq t_{RCD}$ (max.). If t_{RCD} is greater than t_{RCD} (max.) then t_{RAC} will increase by the amount that t_{RCD} exceeds t_{RCD} (max.).
5. Load = 2 TTL loads and 100 pF.
6. Assumes $t_{RCD} \geq t_{RCD}$ (max.).
7. t_{RCD} (max.) is specified as a reference point only; if t_{RCD} is less than t_{RCD} (max.) access time is t_{RAC}, if t_{RCD} is greater than t_{RCD} (max.) access time is $t_{RCD} + t_{CAC}$.
8. t_T is measured between V_{IH} (min.) and V_{IL} (max.).
9. t_{WCS}, t_{CWD} and t_{RWD} are specified as reference points only. If $t_{WCS} \geq t_{WCS}$ (min.) the cycle is an early write cycle and the data out pin will remain high impedance throughout the entire cycle. If $t_{CWD} \geq t_{CWD}$ (min.) and $t_{RWD} \geq t_{RWD}$ (min.), the cycle is a read-modify-write cycle and the data out will contain the data read from the selected address. If neither of the above conditions is satisfied, the condition of the data out is indeterminate.

Figure 5C1.3. 2118 Family Timing (Con't)

5.C.2 Enhanced Operation

Two problems are evident from the previous investigation:

1) $\overline{SACK}$ timing from command will not allow reliable operation while $\overline{XACK}$ is not active early enough to prevent wait states.

2) The normal write command required to guarantee data setup is not enabled until the CPU has sampled READY thereby forcing multiple wait states during write operations.

The first problem could be resolved if an early command could be generated that would guarantee $\overline{SACK}$ was valid when READY was sampled and $\overline{SACK}$ to data valid satisfied the CPU requirements. Figure 5.C.2.1 is a circuit which provides an early read command derived from the maximum mode status. The early command is enabled from the trailing edge of ALE and disabled on the trailing edge of the normal command. The command provides an additional TCHCLmin – TCHLLmax + TCLMLmax – circuit delays = 53 ns of access time and time to generate RDY from the early command. If we go back to our previous equations, early command to valid data at the CPU is now:

TCHCLmin – TCHLLmax + 2TCLCL – TDVCLmax – buffer and circuit delays = 333 ns

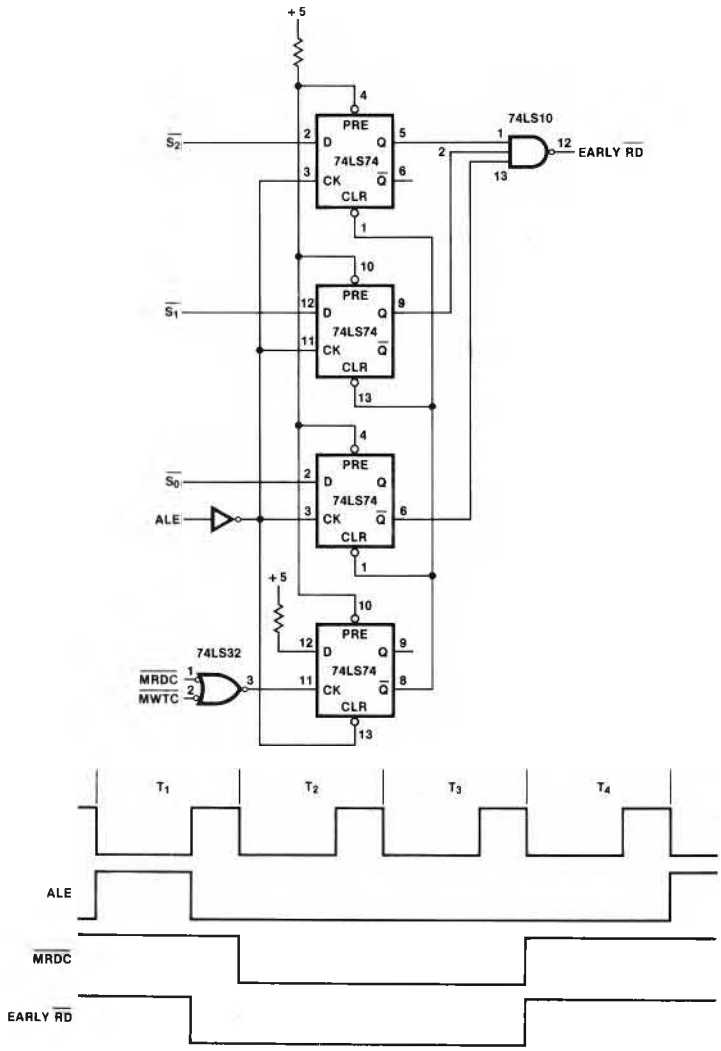


Figure 5C2.1. Early Read and Write Command Generation

We can now use the slowest 2118 which gives 8202 and 2118 access of 320 ns. Early command to RDY timing is TCLCL – TCHLLmax – circuit delays – TR1VCLmax = 115 ns and provides 35 ns of margin beyond the 8202 command to $\overline{SACK}$ delay.

The write timing of the 8202 and write data valid timing of the 8086 do not allow use of an early write command. However, if the 8202 clock is reduced from 25 MHz to 20 MHz and $\overline{WE}$ to the RAM's is gated with $\overline{CAS}$, the advanced write command ($\overline{AMWC}$) may be used. At 20 MHz the minimum command to $\overline{CAS}$ delay is 148 ns while the maximum data valid delay is 144 ns.

The reduced 8202 clock frequency still satisfies no wait state read operation from early read and will insert no more than one wait state for write (assuming no conflict with refresh). 20 MHz 8202 operation will however require using the 2118-4 to satisfy read access time.

Note that slowing the 8202 to 22.2 MHz guarantees valid data within 10 ns after $\overline{CAS}$ and allows using the 2118-7. Since this analysis is totally based on worst case minimum and maximum delays, the designer should evaluate the timing requirements of his specific implementation.

It should be noted that the 8202 $\overline{SACK}$ is equivalent to $\overline{XACK}$ timing if the cycle being executed was delayed by refresh. Delaying $\overline{SACK}$ until $\overline{XACK}$ time causes the CPU to enter wait states until the cycle is completed. If the cycle is a read cycle, the $\overline{XACK}$ timing guarantees data is valid at the CPU before RDY is issued to the CPU.

The use of the early command signals also solves a problem not mentioned previously. The cycle rate of the 8202 @ 20 MHz requires that commands (from leading edge to leading edge) be separated by a minimum of 695 ns. The maximum mode 8086 however may issue a read command 600 ns after the normal write command. For the early read command and advanced write command, 725 ns are guaranteed between commands.

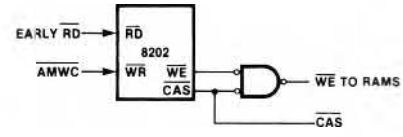


Figure 5C2.2. Delayed Write to Dynamic RAMs

APPENDIX I
BUS CONTENTION AND ITS EFFECT ON SYSTEM INTEGRITY

SYSTEM ARCHITECTURE

As higher performance microprocessors have become available, the architecture of microprocessor systems has been evolving, again placing demands on memory. For many years, system designers have been plagued with the problem of bus contention when connecting multiple memories to a common data bus. There have been various schemes for avoiding the problem, but device manufacturers have been unable to design internal circuits that would guarantee that one memory device would be "off" the bus before another device was selected. With small memories (512x8 and 1Kx8), it has been traditional to connect all the system address lines together and utilize the difference between t_{ACC} and t_{CO} to perform a decode to select the correct device (as shown in Figure 1).

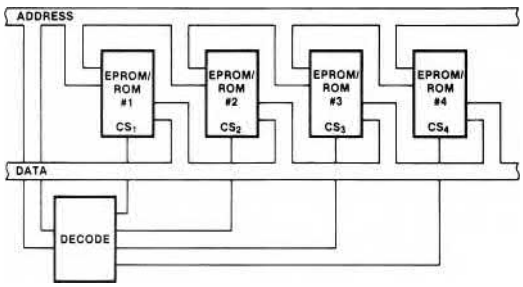


Figure 1. Single Control Line Architecture

With the 1702A, the chip select to output delay was only 100 ns shorter than the address access time; or to state it another way, the t_{ACC} time was 1000 ns while the t_{CO} time was 900 ns. The 1702A t_{ACC} performance of 1000 ns was suitable for the 4004 series microprocessors, but the 8080 processor required that the corresponding numbers be reduced to $t_{ACC} = 450$ ns and $t_{CO} = 120$ ns. This allowed a substantial improvement in performance over the 4004 series of microprocessors, but placed a substantial burden on the memory. The 2708 was developed to be compatible with the 8080 both in access time and power supply requirements. A portion of each 8080 machine cycle time had to be devoted to the architecture of the system decoding scheme used. This devoted portion of the machine cycle included the time required for the system controller (8224) to perform its function before the actual decode process could begin.

Let's pause here and examine the actual decode scheme that was used so we can understand how the control functions that a memory device requires are related to system architecture.

The 2708 can be used to illustrate the problem of having a single control line. The 2708 has only one read control function, chip select ($\overline{CS}$), which is very fast ($t_{CO} = 120$ ns) with respect to the overall access time ($t_{ACC} = 450$ ns) of the 2708. It is this time difference (330 ns) that is used to perform the decode function, as illustrated in Figure 2. The scheme works well and does not limit system performance, but it does lead to the possibility of bus contention.

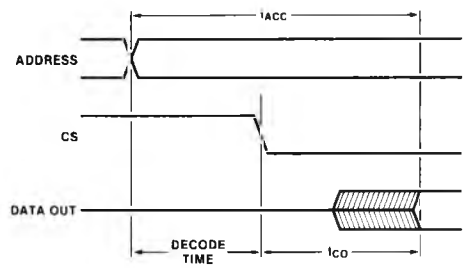


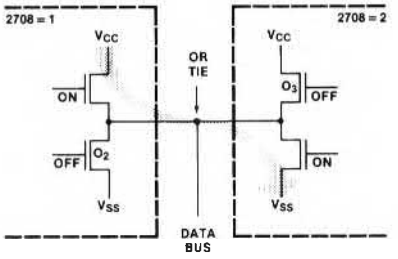
Figure 2. Single Line Control Architecture

BUS CONTENTION

There are actually two problems with the scheme described in the previous section. First, if one device in a multiple memory system has a relatively long deselect time, and a relatively fast decoder is used, it would be possible to have another device selected at the same time. If the two devices thus selected were reading opposite data; that is, device number one reading a HIGH and device number two reading a LOW, the output transistors of the two memory devices would effectively produce a short circuit, as Figure 3 illustrates. In this case, the current path is from V_{CC} on device number one to GND on device number two. This current is limited only by the "on" impedance of the MOS output transistors and can reach levels in excess of 200 mA per device. If the MOS transistors have a lot of "extra" margin, the current is usually not destructive; however, an instantaneous load of 400 mA can produce "glitches" on the V_{CC} supply—glitches large enough to cause standard TTL devices to drop bits or otherwise malfunction, thus causing incorrect address decode or generation.

The second problem with a single control line scheme is more subtle. As previously mentioned, there is only one control function available on the 2708 and any decoding scheme must use it out of necessity. In addition, any inadvertent changes in the state of the high order address lines that are inputs to the decoder will cause a change in the device that is selected. The result is the same as before—bus contention, only from a different source. The deselected device cannot get "off" the bus before the selected one is "on" the bus as the addresses rapidly change state. One approach to solving this problem would be to design (and specify as a maximum) devices

with t_{DF} time less than t_{CO} time, thereby assuring that if one device is selected while another is simultaneously being deselected, there would be some small (20 ns) margin. Even with this solution, the user would not be protected from devices which have very fast t_{CO} times (t_{CO} is specified as a maximum).



RESULTS OF IMPROPER TIMING WHEN OR TYING MULTIPLE MEMORIES.

Figure 3. Results of Improper Timing when OR Tying Multiple Memories

The only sure solution appears to be the use of an external bus driver/transceiver that has an independent enable function. Then that function, not the "device selecting function," or addresses, could control the flow of data "on" and "off" the bus, and any contention problems would be confined to a particular card or area of a large card. In fact, many systems are implemented that way—the use of bus drivers is not at all uncommon in large systems where the drive requirements of long, highly capacitive interconnecting lines must be taken into consideration—it also may be the reason why more system designers were not aware of the bus contention problem until they took a previously large (multicard) system and, using an advanced micorprocessor and higher density memory devices, combined them all on one card, thereby eliminating the requirement for the bus drivers, but experiencing the problem of bus contention as described above.

THE MICROPROCESSOR/MEMORY INTERFACE

From the foregoing discussion, it becomes clear that some new concepts, both with regard to architecture and performance are required. A new generation of two control line devices is called for with general requirements as listed below:

1. Capability to control the data "on" and "off" the system bus, independent of the device selecting function identified above.

2. Access time compatible with the high performance microprocessors that are currently available..

Now let's examine the system architecture that is required to implement the two line control and prevent bus contention. This is shown in the form of a timing diagram (Figure 4). As before, addresses are used to generate the unique device selecting function, but a separate and independent Output Enable (OE) control is now used to gate data "on" and "off" the system data bus. With this scheme, bus contention is completely eliminated as the processor determines the time during which data must be present on the bus and then releases the bus by way of the Output Enable line, thus freeing the bus for use by other devices, either memories or peripheral devices. This type of architecture can be easily accomplished if the memory devices have two control functions, and the system is implemented according to the block diagram shown in Figure 5. It differs from the previous block diagram (shown in Figure 1) in that the control bus, which is connected to all memory Output Enable pins, provides separate and independent control over the data bus. In this way, the microprocessor is always in control of the system; while in the previous system, the microprocessor passed control to the particular memory device and then waited for data to become available. Another way to look at it is, with a single control line the sytem is always asynchronous with respect to microprocessor/memory communications. By using two control lines, the memory is synchronized to the processor.

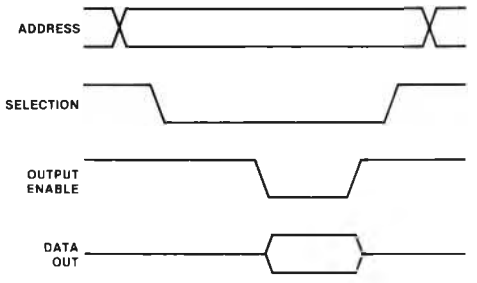


Figure 4. Two Control Line Architecture

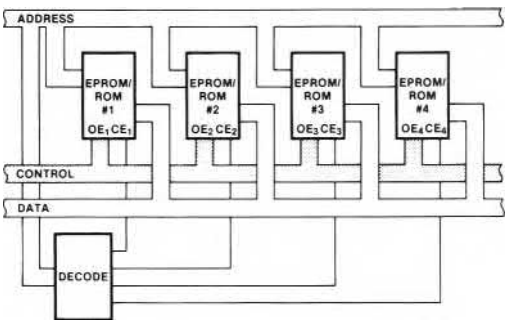


Figure 5. Two Control Line Architecture

intel®

APPLICATION NOTE

AP-61

July 1979

Multitasking for the 8086

Cecil Moore
Applications Engineer
Microprocessor Products

© Intel Corporation, 1979

Multitasking For the 8086

Contents

INTRODUCTION

ANATOMY OF THE TASK MULTIPLEXER

DEFINITIONS

STATE DIAGRAM

LINKED LISTS

DELAY STRUCTURE

PROCEDURES

- ACTIVATE$TASK Procedure
- ACTIVATE$DELAY Procedure
- DECREMENT$DELAY Procedure
- CASE$TASK Procedure
- PREEMPT Procedure
- DISPATCH Procedure

PL/M-86 PROCEDURES

- Initialization and the Main Loop
- Additional Ideas
- Source Code

REFERENCES

INTRODUCTION

Real-time software systems differ markedly from batch processing systems. An external signal indicating that it is time for an hourly log or an interrupt caused by an emergency condition is an event usually not encountered in batch processing. Because real-time control systems of all types share a number of characteristics, it is possible to develop flexible operating systems which will meet the needs of a great majority of real-time applications. Intel Corporation has developed such a system, the RMX/80™ system, for the iSBC™ line of 8080/85 based single board computers. Thus, the user is released from the chore of designing an operating system and is free to concentrate his efforts on the applications software for the individual tasks and merely integrate them into a pre-existing system.

But what if a user does not need all the capabilities of an RMX/80™ system or wants a different hardware configuration than an iSBC™ computer? This application note contains a set of PL/M-86 procedures designed to be used in medium-complexity 8086 real-time systems.

A normal control system can be broken down into a number of concurrently executable tasks. The CPU can be running only one task at any instant of time but the speed of the processor often makes concurrent tasks appear to be running simultaneously. Breaking the software functions into separate concurrent tasks is the job of the designer/programmer. Once this is done there remains the problem of integrating these tasks with a supervisory program which acts as a traffic cop in the scheduling and execution of the separate tasks. This note discusses a set of PL/M-86 procedures to implement the supervisory program function.

A minimum operating system might (like its batch processing cousin) have only a queue for ready tasks (tasks waiting to be executed). Any task that becomes ready is put on the bottom of the queue and when a running task is finished, the task on the top of the queue is started. Any interrupt causes the state of the system to be saved, an interrupt routine to be executed, the state of the system to be restored, and execution of the interrupted program to continue. The interrupt routine might (or might not) put a new task on the ready queue. This approach has worked well for many simple control systems, especially in the single-chip computer area. But what features are lacking in this approach that are necessary (or at least nice)?

1. A system of priorities is often needed. All waiting ready tasks must be executed sooner or later but some tasks need immediate attention while others can be run when there is nothing else to do. If a midnight monthly report, due for completion by 8 a.m. the next day, is in the process of printing at 1 a.m. and a fire alarm occurs, it is reasonable to assume that the fire alarm has higher priority since the fire could conceivably render the monthly report irrelevant.

There are a number of ways in which to assign priorities. Tasks are usually numbered and may be assigned priorities according to their ascending (or descending) numbers. They could instead be grouped into a number of priority levels, with tasks on the same level having equal priorities. The latter approach is taken in this application note.

Assume that a monthly report is being printed and an alarm occurs in the external world that, because of its importance, must be attended to immediately. The interrupt routine, executed as a result of the alarm input, should not automatically return to the interrupted logging routine but instead should call a preempt routine which checks to see if a higher priority task is ready for execution. The reason for this is that the monthly report routine, if returned to, has no way of "knowing" that a higher priority task is waiting to be executed. The alarm output task has been readied by the interrupt routine and since it is known to be higher priority than the logging task, it is executed first, thereby immediately signaling the system operator that there has been an alarm. It then returns to the logging task provided that there are no further high priority tasks waiting to be executed. The logging printer may not have even paused during the alarm output task. The computer appears to human beings to be executing concurrent tasks simultaneously.

Of course, the alarm output function could be performed inside the interrupt procedure. But sooner or later, the designer will encounter a worst case situation in which there is not enough time to execute all required tasks between interrupts, and the system will fall behind in real-time. It is much cleaner to make the interrupt procedures as short as possible and stack up tasks to be executed than to stack up interrupt procedures.

2. Another feature that might be necessary is a capability to put a task to sleep for a known period of real time. Assume a relay output must remain closed for one second. Most real-time systems cannot tolerate the dedication of the CPU to such a trivial task for that length of time so a system of programmable dynamic delays could be implemented. This application note implements such a system.

Although the PL/M-86 procedures here have been debugged and tested, it is assumed that the user will want to change, add, or delete features as needed. This application note is intended to present ideas for a logical structure of procedures that, because they are written in PL/M-86, can be easily modified to user requirements. Each procedure will be discussed in detail and integration and optional features will be presented.

PL/M-86

PLM-86 is a block structured high level language that allows direct design of software modules. Using PL/M-86, designers can forget their assembly level

coding problems and design directly in a subset of the English language. The 8086 architecture was designed to accommodate highly structured languages and the PLM-86 compiler is quite efficient in the generation of machine code.

PLM-86 STRUCTURE

PL/M-86 automatically keeps track of the level of the different software blocks. (See Chapter 10, "PL/M-86 Programming Manual"). There are methods of writing PL/M-86 which contribute to the understandability of the source code without adding to the amount of object code generated. For instance, the following three IF/THEN/ELSE blocks generate identical object code but are compiled from different source statements.

Line	Level	Statement
3	1	IF A = B THEN C = D; ELSE E = F; G = H;
7	1	IF A = B THEN
8	1	C = D;
		ELSE
9	1	E = F;
10	1	G = H;
11	1	IF A = B THEN DO;
13	2	C = D ;
14	2	END;
15	1	ELSE DO;
16	2	E = F;
17	2	END;
18	1	G = H;

It is not instantly apparent from the code on line 3 or the code starting at line 7 which statements will be executed. However, adding the DO; and END; statements (starting at line 11) remove any doubt. Either the statements starting at line 11 or the statements starting at line 15 will be executed and the statement on line 18 will be executed in either case. Why? Because all these lines are at level 1 in the block structure. The other lines are at level 2 because of the DO;/END; combinations. When one refers to the relatively complex structures of the task multiplexer procedures, the usefulness of such an approach is obvious, as the procedures have been indented according to the level numbers generated by PL/M-86. In particular, if the designer is not careful, nested IF/THEN/ELSE statements can generate improper results. Using a proper number of DO;/END; combinations avoids the possible ambiguity in nested IF/THEN/ELSE statements as can be seen in the ACTIVATE$TASK procedure listed in the PL/M-86 source code later in this note. The DO;/END; construct naturally must be used when multiple statements are required within the IF/THEN/ELSE blocks. Following are examples of the possible primary structures of PL/M-86:

```
DO;
    A = B;
    C = D;
    END;

DO WHILE A = B:
    C = D;
    E = F;
    END;

DO I = 1 TO 5;
    A = I;
    C = D + I;
    END;

DO CASE A;
    A = B;
    A = C;
    A = D;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE DO;
    E = F;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE IF A = C THEN DO;
    D = E;
    END;
ELSE IF A = D THEN DO;
    E = F;
    END;
ELSE DO;
    F = G;
    END;
```

A complete tutorial on structured programming is beyond the scope and intent of this application note and the reader is referred to the appropriate references appearing in the bibliography.

ANATOMY OF THE TASK MULTIPLEXER

Once a decision is made on the details of the kind of data structure that is needed to implement the task multiplexer, the procedures that manipulate the structure are relatively simple to write. The following characteristics are assumed for the task multiplexer appearing in this application note.

There are two levels of priority, high and low. All high priority tasks that are ready to run will be dispatched, executed, and completed, on a FIFO basis, before any low priority task is dispatched.

Any task can be interrupted. No task multiplexer procedure can be interrupted.

If a high priority task is interrupted, it will be completed before any other task is dispatched. If a low priority task is interrupted, all ready high priority tasks will be dispatched, executed, and completed before program control is returned to the low priority task.

There are two ready queues, one for high priority tasks and one for low priority tasks. Each queue has a head (top) pointer and a tail (bottom) pointer and tasks on any queue are link-listed from head to tail. Tasks are "dispatched" (taken off the queue) at the head and "activated" (put on the queue) at the tail on a FIFO basis.

Link-listed queues are chosen for simplicity. All dispatch and activate information is contained in the head and tail pointers. Tasks located in the middle of these link-lists are of no concern for activating and dispatching. This means, of course, that tasks are executed in the order that they appear on the queue, i.e., first-in, first-out.

There is a pointer byte associated with each task. If a task is on either the low priority or high priority ready queue, its associated pointer byte will point to the next task number on the list. These pointer bytes enable the task ready lists to be linked. Note that the pointer byte is 0 for the last task on a list.

There is a status (flag) byte associated with each task. If a task is on a ready list or a delay list, bit 7 will be a "1" indicating that that particular task is busy. If a task is on either high priority or low priority ready queues, bit 6 will be a "1" indicating that the task is on one of the ready queues. If the task is listed on the delay list, (see next item), bit 5 will be a "1" indicating that this particular task has a delay in progress. If a task is unlisted, bits 5-7 will be "0." Bits 0-4 are not used by the task multiplexer procedures and are available to the user, giving 5 user defined flags per task.

There is a delay byte associated with each task. This feature allows tasks to be "put to sleep" for a variable length of time, from 1 to 255 "ticks" of the interrupt clock. If a task does not need an associated delay then this byte is available to the user as a utility byte to be used for any purpose. These delays will be discussed in detail later in the application note.

The following diagram is a representation of the task multiplexer data structure:

TASK NUMBER	POINTER BYTE	STATUS BYTE	DELAY BYTE
0	n	n + 1	n + 2
1	n + 3	n + 4	n + 5
2	n + 6	n + 7	n + 8
3	n + 9	n + 10	n + 11
4	n + 12	n + 13	n + 14
5	n + 15	n + 16	n + 17
m − 1	n + 3m − 6	n + 3m − 5	n + 3m − 4
m	n + 3m − 2	n + 3m − 1	n + 3m

3m + 3 TOTAL RAM BYTES
n = FIRST RAM ADDRESS OF ARRAY

Following is a chart of what a task multiplexer data structure might look like at a given moment in time:

HIGH$PRIORITY$HEAD = 5
HIGH$PRIORITY$TAIL = 3
LOW$PRIORITY$HEAD = 8
LOW$PRIORITY$TAIL = 10
DELAY$HEAD = 4

TASK NUMBER	TASK(n).PNTR	TASK(n).STATUS	TASK(n).DELAY
0	*	*	*
1	3	1100 0000	0
2	0	1010 0000	3
3	0	1100 0000	0
4	7	1010 0000	4
5	1	1100 0000	0
6	0	0000 0000	0
7	2	1010 0000	6
8	10	1100 0000	0
9	0	0000 0000	0
10	0	1100 0000	0

*See text.

What information can one ascertain from observation of the above chart? The ready-to-run high priority tasks, in order, are 5,1,3. This can be seen by following the high priority ready linked list from head to tail. The ready-to-run low priority tasks, in order are 8, 10. The TASK(n).PNTR byte = 0 for the last listed task. Tasks 4, 7, 2 are listed, in order, on the delay list and have associated delays of 4, 10, 13 ticks respectively. Tasks 6 and 9 are not listed and therefore idle. The * for the TASK (0) bytes indicate a special condition. There is no TASK00 allowed and a zero condition is treated as an error condition. TASK(0).PNTR byte is used for the DELAY$HEAD byte to minimize code in the ACTIVATE$DELAY procedure. TASK(0).STATUS and TASK(0).DELAY are unused bytes.

DEFINITIONS

NEW$TASK is the number of the task that will be installed on a ready list or the delay list when ACTIVATE$TASK or ACTIVATE$DELAY is called.

NEW$DELAY is the value of the delay that will be installed on the delay list when ACTIVATE$DELAY is called.

A task is defined as RUNNING if it is in the act of execution or if an interrupt routine is executing which interrupted a RUNNING task.

A task is defined as PREEMPTED if it has been interrupted and a higher priority task is being executed.

A task is defined as READY if it is contained within one of the ready queues.

A task is defined as IDLE if its BUSY$BIT (bit 7) is not set, i.e., it is not listed anywhere else. Note that it is possible to completely disable an IDLE task simply by setting its BUSY$BIT. In that case, it is not and cannot be listed anywhere else. This feature is useful during system integration.

STATE DIAGRAM

The state diagram indicates the relationships among the possible task states and the procedures involved in changing states.

The state diagram looks somewhat complicated and a discussion of the possible change of states is in order. Assuming a certain existing state, future possible states will be discussed including the procedures which can cause the change of state.

From the unlisted (idle) state, the ACTIVATE$TASK procedure will put the NEW$TASK on either the high priority ready queue or the low priority ready queue at the tail end of the queue. The number of the task automatically assigns the priority and therefore the proper queue. All task numbers below FIRSTLOWPRIORITY$TASK are assumed to be high priority tasks. Also, from the unlisted state the ACTIVATE$DELAY procedure will put the NEW$TASK and NEW$DELAY at the proper position on the delay list.

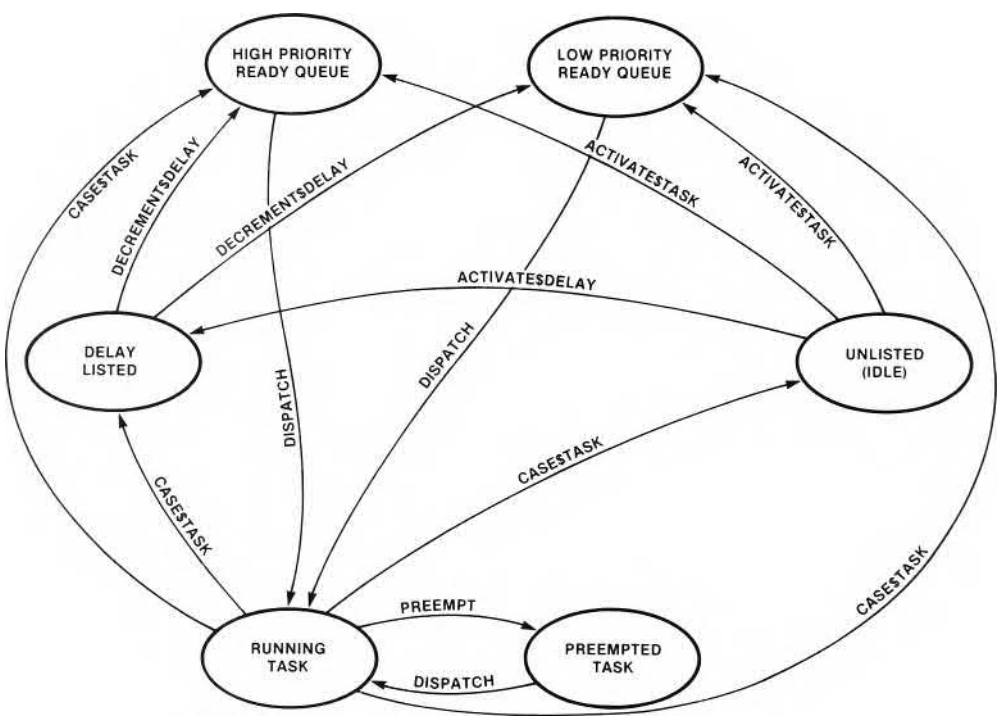
After a task has been put on either high priority ready queue or low priority ready queue it eventually will go to the RUNNING$TASK state. The DISPATCH procedure accomplishes this action.

From the delay list a task can only go to one of the ready queues. When a task's associated delay goes to zero the DECREMENT$DELAY procedure calls the ACTIVATE$TASK procedure and installs the NEW$TASK on the proper ready queue.

From the RUNNING$TASK state a task may use the CASE$TASK procedure to put itself on the ready list tail by setting NEW$TASK = RUNNING$TASK. It may instead put itself on the delay list by setting NEW$TASK = RUNNING$TASK and also setting NEW$DELAY equal to something other than zero. Otherwise, it will progress to the unlisted state upon completion.

The CASE$TASK procedure unlists tasks when they have completed execution. A low priority RUNNING$TASK will go to the preempted state if a high priority task is on the ready list following an interrupt during execution of the low priority task if the PREEMPT procedure is called.

And finally, a PREEMPTED$TASK will return to a RUNNING$TASK state when all high priority ready tasks have completed execution. This is accomplished by the DISPATCH procedure which then returns to the PREEMPT procedure.



STATE DIAGRAM

Some lockouts are necessary to avoid chaos in the task multiplexer. These are as follows:

The BUSY$BIT = 1 in the TASK(n).STATUS byte will abort the ACTIVATE$TASK and the ACTIVATE$DELAY procedures and return an indication of the aborting by setting the STATUS byte equal zero. A task must be unlisted to be able to be installed on a list.

A RUNNING$TASK may put itself on a list after it has executed but it is not allowed to re-list any listed tasks (i.e., no task may ever be listed twice at the same time!). A task that tries to activate another task that is already busy can wait (via the delay feature) for the required task to complete execution, become idle, and therefore be available to be activated. A PREEMPTED$TASK may not be listed. If the ACTIVATE$TASK or ACTIVATE$DELAY procedure is called and NEW$TASK = PREEMPTED$TASK, the procedure will be aborted and return with STATUS = 0. Otherwise, the STATUS byte is returned with the new task status.

Only one task may be preempted as there are only two levels of priority. The user may desire to implement many levels of priority in which case a linked-list of preempted tasks could be declared in a structure which includes the number of the first task in each priority level group of tasks. This obviously complicates the PREEMPT and DISPATCH procedures.

The tasks themselves are made into reentrant procedures because of the necessary forward references of the CASE$TASK procedure.

PL/M-86 allows structures and arrays of structures. The structure needed for the task multiplexer is a link-list pointer byte, a task status byte, and a task delay byte. Each task has an associated pointer byte, status byte, and delay byte. These are combined into an array of up to 255 tasks. For purposes of this discussion, the number of tasks is chosen as an arbitrary 10, leading to the following array declaration.

```
DECLARE TASK(10)STRUCTURE
(PNTR BYTE,STATUS BYTE,DELAY BYTE);
```

Thus the delay byte associated with task number 7 can be accessed by using the variable TASK(7).DELAY and the status of task number 5 can be examined through the use of TASK(5).STATUS. The TASK(n).PNTR byte contains the task number of the next listed task on the same list as TASK(n), i.e., if TASK(n) is on the delay list, then TASK(n).PNTR will contain the number of the next task on the delay list or 0 indicating the end of the list.

TASK(n).STATUS is a byte with the following reserved flags:

BIT 7 BUSY$BIT, "1" IF TASK IS BUSY
BIT 6 READY$BIT, "1" IF ON READY LIST
BIT 5 DELAY$BIT, "1" IF ON DELAY LIST
BIT 4 — BIT 0 UNUSED

The unused bits in the STATUS byte are available to the user.

The TASK(n).DELAY byte is a number which can put TASK(n) to sleep for up to 255 system clock ticks. The system clock tick is interrupt driven from the user's timer and its period is chosen for the particular application. A one millisecond timer is popular and assuming such a time, delays of up to 255 ms are available in the task multiplexer as it is written. If this delay range is not wide enough, the user may want to define his TASK(n).DELAY as a word instead of a byte in the PL/M-86 declare statement, giving delays of up to 65 seconds from the basic one millisecond clock tick.

LINKED LISTS

Linked lists are useful for a number of reasons. However, a treatise on linked lists would defeat the purpose of this application note and the reader is referred to the references listed in the bibliography.

The linked lists used in this application note have a head byte associated with each list, i.e., the head byte contains the number of the first task on the list. The first task pointer byte points to the second task on the list, etc. The pointer of the last task on the list is set at zero to indicate that it is the last task. Two of the linked lists are ready queues and require a tail byte as well as a head byte. The tail byte points to the last entry on the list. Tasks are put on the bottom, or tail, of the ready lists and are taken off the top, or head, of the ready lists. The delay list has no tail but does have a head, called a DELAY$HEAD. The delay list is not a queue, as delays are installed on the list in order of delay magnitude for reasons to be explained later.

There are two ready lists, one for high priority tasks and one for low priority tasks. The head and tail pointers associated with these two lists are: HIGH$PRIORITY$HEAD, HIGH$PRIORITY$TAIL, LOW$PRIORITY$HEAD, and LOW$PRIORITY$TAIL. Obviously, the structure can be expanded to any number of priority levels by expanding the head and tail pointers and the historical record of the preempted tasks.

DELAY STRUCTURE

A task multiplexer can have a number of simultaneous delays active and it would be efficient if there were a way to keep from decrementing all delays on every clock tick, which is most time consuming. One way to accomplish this feat is to move the problem from the DECREMENT$DELAY routine to the ACTIVATE$DELAY routine. The delays are arranged in a linked-list of ascending sizes such that the value of each delay includes the sum of all previous delays. This allows the decrementing of only one delay during each clock tick interrupt routine. An example will further illuminate this approach. Suppose the following conditions exist:

Task 7 has a 5 millisecond delay

Task 3 has an 8 millisecond delay

Task 9 has a 14 millisecond delay

The delay structure is arranged so that:

DELAY$HEAD = 07
TASK(7).PNTR = 03
TASK(3).PNTR = 09
TASK(9).PNTR = 00
TASK(7).DELAY = 05 (FIRST DELAY = 5)
TASK(3).DELAY = 03 (5 + 3 = 8)
TASK(9).DELAY = 06 (5 + 3 + 6 = 14)

The linked-list is arranged so that the delays are in ascending order and each delay is equal to the sum of all previous delays up through that point. Since this is true, all delays are effectively decremented merely by decrementing the first delay. Of course, something for nothing is impossible and the speed gained by arranging the delays in the above manner is paid for by the complexity of the ACTIVATE$DELAY routine. But since the ACTIVATE$DELAY routine is executed less frequently than the DECREMENT$DELAY routine, the savings in real time is worth the added complexity.

Suppose a new delay is to be activated in the above scheme. Task 5 with a delay of 10 milliseconds is to be added. A before and after chart will indicate what the ACTIVATE$DELAY procedure must accomplish.

BEFORE

TASK NUMBER		07	03	09	
POINTER	07	03	09	00	
DELAY		05	03	06	

AFTER

TASK NUMBER		07	03	05	09
POINTER	07	03	05*	09@	00
DELAY		05	03	02@	04*

FIRST POINTER IS THE DELAY$HEAD
CHANGES ARE MARKED WITH AN *
ADDITIONS ARE MARKED WITH AN @

Note that the pointer before the added task has changed and the delay after the added task has changed. The function of the ACTIVATE$DELAY procedure is to accomplish these changes and additions.

PROCEDURES

The following procedure explanations reference the PL/M-86 source code listing which follows the application note text.

ACTIVATE$TASK Procedure

This procedure is initiated by a call instruction with the byte NEW$TASK containing the number of the task to be put on the proper ready queue.

Interrupts must be disabled whenever the link-lists are being changed. If interrupts are enabled when this procedure is called, they should be re-enabled upon returning.

The assignment of priority is a simple matter. A declare statement, DECLARE FIRSTLOWPRIORITY$TASK LITERALLY 'N,' (where N is the actual number of the first low priority task) indicates to the procedures that tasks 1 to N are high priority tasks and tasks N or higher are low priority tasks.

This procedure checks the busy bit in the status byte to see if this particular task is already busy and if so, returns a STATUS of zero. Otherwise, it returns the new STATUS of the task. It then checks the priority to see if this particular task is a high or low priority. If it is high priority, then the task pointer pointed to by the HIGH$ PRIORITY$TAIL pointer is changed from zero to the number of the NEW$TASK. The HIGH$PRIORITY$TAIL pointer is then changed to the number of the NEW$TASK and the pointer associated with NEW$ TASK is made equal to zero. This completes the ACTIVATE$TASK functions. If the new task is a low priority task, then the same functions are performed using the LOW$PRIORITY$TAIL pointer.

ACTIVATE$DELAY Procedure

This procedure is initiated by a call with the byte NEW$ TASK containing the number of the task to be put on the delay list and the byte NEW$DELAY containing the value of the associated delay.

Interrupts are disabled and the busy bit of this particular task is checked. If the busy bit is set the STATUS byte is set to zero and the procedure returns without activating the delay. If the busy bit is not set the integer value DIFFERENCE is set equal to the NEW$DELAY value. POINTER$0 is set equal to the DELAY$HEAD. POINTER$1 is set to zero. The DO WHILE loop executes until POINTER$0 equals zero or DIFFERENCE is less than zero. Remember that the proper place to insert the new delay is being searched for, and that will be either at the end of the list (POINTER$0 = 0) or when the sum of the previous delays do not exceed the new delay value. The DO WHILE loop has POINTER$0, POINTER$1, OLD$DIFFERENCE, and DIFFERENCE keeping track of where the procedure is in the loop, while searching for the proper place to insert the new delay. The existing delays are sequentially subtracted from the remains of NEW$ DELAY according to the link-listed order until the end of the list or a negative result is encountered indicating that the proper delay insertion point has been reached. At this point POINTER$0 contains the task number to be assigned to TASK(NEW$TASK).PNTR. POINTER$1 contains the task number immediately preceding the NEW$TASK such that TASK(POINTER$1). PNTR = NEW$ TASK and our link list is fully updated, with the actual delays yet to go. If POINTER$0 = 0 it means that the new delay is larger than any of the other delays and therefore should go on the end of the list so TASK(NEW$ TASK).DELAY is set equal to the DIFFERENCE. If

POINTER$0 is not equal to zero then if POINTER$0 equals POINTER$1 (indicating that there were not any delays previously listed), then TASK(POINTER$1).PNTR is set equal to zero. TASK(NEW$TASK).DELAY is set equal to the OLD$DIFFERENCE and TASK (POINTER$0).DELAY is set equal to the negative of DIFFERENCE which at this point is negative, thereby resulting in a positive unsigned number. The reader is encouraged to implement an example (see Delay Structure section) to prove that the above approach is valid. Particular attention should be paid to the contents of the two pointers, as they are the key to the procedure. The final function of this procedure is to set the BUSY$BIT and DELAY$BIT in the TASK(NEW$ TASK).STATUS byte. The byte named STATUS which is returned by this procedure is set equal to the status of the new task. If it is desired to have interrupts enabled, they must be enabled after the procedure return instruction. The reason for such a complex method of activating a delay will become apparent in the following section.

DECREMENT$DELAY Procedure

The first delay on the linked-list is decremented and, if it is zero, the associated task is put on the appropriate ready queue. The next delay (if any) is checked to see if it is zero and if so, that task is put on the appropriate ready queue, etc. A loop is performed until either no delay or a non-zero delay is found. The procedure then returns.

It is assumed that this procedure is part of an interrupt routine and that the interrupts are disabled during its execution. Interrupts cannot be enabled during changes to any of the linked-lists or else recovery may not be possible.

This procedure begins by checking to see if there are any active delays. If DELAY$HEAD = 0 then this procedure returns immediately. Otherwise it decrements the first delay. If this delay goes to zero then the associated task number is passed to the ACTIVATE$ TASK procedure as the OFF$DELAY byte. A new DELAY$HEAD is chosen from the next link-listed delay and that delay checked for a value of zero which will happen if the first two or more delays are equal. This loop is accomplished by the DO WHILE DELAY$ HEAD <> 0 AND TASK(DELAY$HEAD).DELAY = 0; This procedure is designed to require very little CPU time unless a delay times out. The DO WHILE loop is bypassed if the resulting delay value is not zero. A certain amount of care should be exercised to insure that many delays do not all time out at the same time. One method would be to modify the ACTIVATE$DELAY procedure to insure that there are no zero entries in the delay bytes. The basic procedure, however, assumes that the clock "tick" timing will be chosen to minimize the above potential problem.

CASE$TASK Procedure

This procedure performs the function of calling the task indicated by the contents of the RUNNING$TASK byte. All listed tasks are called in this manner. The CASE$TASK procedure is called by the DISPATCH procedure. When a particular task has completed execution it returns to the CASE$TASK procedure which then resets the BUSY$BIT and the READY$BIT and returns to the DISPATCH procedure after setting RUNNING$TASK equal to zero. This procedure allows a task to relist itself immediately upon returning from execution.

PREEMPT PROCEDURE

The PREEMPT procedure is called whenever it is possible that a high priority task has been put on the ready queue while a low priority task was in the process of execution. An example will illustrate:

Assume that the control system is being interrupted by the 60 Hz line frequency and a register is being incremented each time this 16.67 ms edge occurs. When the register gets to 60 (indicating that one second has passed), the register is zeroed and the high priority time-keeping task is put on the ready queue. Assume also that a low priority data logging task was running when this interrupt occurred. The interrupt routine calls PREEMPT. If a high priority task is running, PREEMPT simply returns. But in our example, a low priority task is running so PREEMPT transfers RUNNING$TASK to PREEMPTED$TASK and calls DISPATCH, which calls CASE$TASK, which calls the time-keeping task. When the time-keeping task has completed, it returns to CASE$TASK which returns to DISPATCH which returns to the PREEMPT procedure which returns to the interrupt routine which returns to the interrupted low priority data logging task if no other high priority tasks are on the ready queue. If the high priority ready queue is not empty, any and all high priority tasks will be completed before the interrupted routine is returned to. PREEMPT refuses to return to the interrupt routine until HIGH$ PRIORITY$HEAD is equal to zero. It is important to note that a low priority task will not be preempted unless the PREEMPT procedure is called. As noted above, it is normally called from the interrupt routine which interrupted the low priority task, but there is nothing to prohibit PREEMPT from being called from inside a low priority task procedure.

DISPATCH PROCEDURE

This procedure calls a high priority task if HIGH$ PRIORITY$HEAD is not equal to zero, restores a preempted task if PREEMPTED$TASK is not equal to zero, calls a low priority task if LOW$PRIORITY$HEAD is not equal to zero, and simply returns if there is nothing to do, all in order of priority. The DISPATCH procedure is called from the main program loop which must enable interrupts as DISPATCH disables interrupts as soon as

it is called. It is also called by the PREEMPT procedure. RUNNING$TASK must be 0 when this procedure is called.

PL/M-86 PROCEDURES

Because the block structure and levels are so important to the understanding of the following procedures, they have been indented according to level. This was a simple task accomplished by no indenting for level one, indenting once for level two, etc. The resulting attractive, easy to follow format was worth the effort to increase the initial level of understanding for readers of this application note who are not intimately familiar with PL/M.

Everything except the very simple main program loop has been made into procedures. Interrupt routines and tasks are also procedures. Keeping track of interrupts, calls, and returns is easy for PL/M and a violation of the block structure through such devices as GOTO targets outside the procedure body is the best way the author knows to crash and burn. Honor the power of the structure, accept the limitations involved, and checkout and debugging will be a pleasure.

Since CASE$TASK references the individual tasks, the task procedure structure was included in the PL/M-86 compilation. All the user has to do is insert the particular task code in place of the /*TASKnn CODE*/ comment, define the interrupt procedures and the system should be ready to run. Obviously, the user will desire to change the total number of tasks and the number of the FIRSTLOWPRIORITY$TASK.

INITIALIZATION AND THE MAIN LOOP

The last entry in the PL/M-86 program is the initialization process which essentially zeros the task multiplexer data and the main loop which loops until TRUE = FALSE, i.e. forever, with interrupts enabled. The STATUS = STATUS instruction simply insures that the loop can be interrupted as the instruction following an ENABLE instruction is not interruptible.

These few instructions are included for information only and will need to be expanded considerably for use in a real-world system. The task multiplexer procedures were checked out on an iSBC 86/12™ computer running under random interrupt control and these instructions were the minimum necessary to cause the system to run. As was stated earlier, the following source code does not include any interrupt procedures and these will have to be generated following the format explained in the PL/M-86 programming manual.

ADDITIONAL IDEAS

Resource allocation is a feature that could be added to the task multiplexer. To keep it simple and yet avoid the deadlock problem (two tasks each grab a resource that the other needs), an extra array can be added to the TASK(n).XXX structure in which each bit in the byte (or word), represents a resource necessary for the execution of a task. A RESOURCES$STATUS byte can then keep the dynamic busy status of the system resources (printers, terminals, floating point math packages, etc.). When the CASE$TASK procedure is called, the resources required by the next RUNNING$ TASK can be compared to the RESOURCES$STATUS byte to see if the required resources are available. If they are, the following PL/M-86 statement will update the new status of the resources:

```
RESOURCES$STATUS = RESOURCES$STATUS OR
        TASK(RUNNING$TASK).RESOURCES;
```

However, if the resources are not available, the CASE$ TASK procedure can return the task to the ready or delay list and try again later. When the task has completed, the following PL/M-86 statement will update the resources status byte:

```
RESOURCES$STATUS = RESOURCES$STATUS AND NOT
        TASK(RUNNING$TASK).RESOURCES;
```

Message passing from task to task may also be necessary. Assuming that a task will have only one message at a time to deliver or receive, another byte could be added to the task structure such that TASK(RUNNING$TASK).MESSAGE could represent a byte containing the number of the task wishing to deliver a message to the RUNNING$TASK. Since a task can call CASE$TASK which in turn will call another task, message block parameters can be passed directly from one task to another. The task that calls CASE$TASK must handle the necessary housekeeping involved in recovering after the message has been passed. Of course, the data structure would have to be expanded to accommodate the message parameters and blocks. For further ideas involving message handling refer to the RMX/80™ user's guide.

Two additional relatively simple procedures could be added to obtain the SUSPEND and RESUME features of the RMX/80™ system. Remember that if the BUSY$BIT is set in a TASK(n).STATUS byte and the task is unlisted, then it cannot be listed. If it is desired to dynamically enable and disable a task, this bit could be set by a SUSPEND procedure and reset by the RESUME procedure.

SOURCE CODE

```
TM86:DO;

DECLARE TOTAL$TASKS LITERALLY '10';
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '0';
DECLARE BUSY$BIT LITERALLY '10000000B';
DECLARE READY$BIT LITERALLY '01000000B';
DECLARE DELAY$BIT LITERALLY '00100000B';
DECLARE FIRST$LOW$PRIORITY$TASK LITERALLY '6';

DECLARE TASK(TOTAL$TASKS) STRUCTURE(PNTR BYTE, STATUS BYTE, DELAY BYTE);
DECLARE HIGH$PRIORITY$HEAD BYTE, HIGH$PRIORITY$TAIL BYTE;
DECLARE LOW$PRIORITY$HEAD BYTE, LOW$PRIORITY$TAIL BYTE;
DECLARE RUNNING$TASK BYTE, PREEMPTED$TASK BYTE;
DECLARE STATUS BYTE, NEW$TASK BYTE, NEW$DELAY BYTE;
DECLARE DELAY$HEAD BYTE AT (@TASK(0).PNTR);

ACTIVATE$TASK: PROCEDURE; /* ASSUMES NEW$TASK<>0 */
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      IF NEW$TASK < FIRST$LOW$PRIORITY$TASK THEN DO;
         IF HIGH$PRIORITY$TAIL<>0 THEN DO;
            TASK(HIGH$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE HIGH$PRIORITY$TAIL=0 THEN */ DO;
            HIGH$PRIORITY$HEAD=NEW$TASK;
            END;
         HIGH$PRIORITY$TAIL=NEW$TASK;
         END;
      ELSE /* SINCE TASK IS LOW PRIORITY THEN */ DO;
         IF LOW$PRIORITY$TAIL<>0 THEN DO;
            TASK(LOW$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE LOW$PRIORITY$TAIL=0 THEN */ DO;
            LOW$PRIORITY$HEAD=NEW$TASK;
            END;
         LOW$PRIORITY$TAIL=NEW$TASK;
         END;
      TASK(NEW$TASK).PNTR=0;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
             BUSY$BIT OR READY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   RETURN;
   END ACTIVATE$TASK;
```

```
ACTIVATE$DELAY: PROCEDURE;/*ASSUMES NEW$TASK, NEW$DELAY<>0*/
   DECLARE POINTER$0 BYTE, POINTER$1 BYTE;
   DECLARE OLD$DIFFERENCE INTEGER, DIFFERENCE INTEGER;
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      DIFFERENCE=INT(NEW$DELAY);
      POINTER$0=DELAY$HEAD;
      POINTER$1=0;
      DO WHILE POINTER$0<>0 AND DIFFERENCE>0;
         OLD$DIFFERENCE=DIFFERENCE;
         DIFFERENCE=DIFFERENCE-INT(TASK(POINTER$0).DELAY);
         IF DIFFERENCE>0 THEN DO;
            POINTER$1=POINTER$0;
            POINTER$0=TASK(POINTER$1).PNTR;
            END;
         END;
      TASK(NEW$TASK).PNTR=POINTER$0;
      TASK(POINTER$1).PNTR=NEW$TASK;
      IF POINTER$0=0 THEN TASK(NEW$TASK).DELAY=LOW(UNSIGN(DIFFERENCE));
      ELSE /* SINCE DIFFERENCE<0 THEN */ DO;
         IF POINTER$0=POINTER$1 THEN TASK(POINTER$1).PNTR=0;
         TASK(NEW$TASK).DELAY=LOW(UNSIGN(OLD$DIFFERENCE));
         TASK(POINTER$0).DELAY=LOW(UNSIGN(-DIFFERENCE));
         END;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
            BUSY$BIT OR DELAY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   NEW$DELAY=0;
   RETURN;
   END ACTIVATE$DELAY;

DECREMENT$DELAY: PROCEDURE;     /* ASSUMES INTERRUPTS DISABLED */
   DECLARE OFF$DELAY BYTE;
   IF DELAY$HEAD<>0 THEN DO;
      TASK(DELAY$HEAD).DELAY=TASK(DELAY$HEAD).DELAY-1;
      DO WHILE DELAY$HEAD<>0 AND TASK(DELAY$HEAD).DELAY=0;
         OFF$DELAY=DELAY$HEAD;
         DELAY$HEAD=TASK(DELAY$HEAD).PNTR;
         TASK(OFF$DELAY).STATUS=TASK(OFF$DELAY).STATUS
             AND NOT(BUSY$BIT OR DELAY$BIT);
         NEW$TASK=OFF$DELAY;
         CALL ACTIVATE$TASK;
         END;
      END;
   RETURN;
   END DECREMENT$DELAY;
```

```
CASE$TASK: PROCEDURE REENTRANT;
   DO CASE RUNNING$TASK;
      CALL TASK00;
      CALL TASK01;
      CALL TASK02;
      CALL TASK03;
      CALL TASK04;
      CALL TASK05;
      CALL TASK06;
      CALL TASK07;
      CALL TASK08;
      CALL TASK09;
      END;
   TASK(RUNNING$TASK).STATUS=TASK(RUNNING$TASK).STATUS AND
             NOT (BUSY$BIT OR READY$BIT);
   TASK(RUNNING$TASK).PNTR=0;
   IF RUNNING$TASK=NEW$TASK THEN DO;
      IF NEW$DELAY<>0 THEN DO;
         CALL ACTIVATE$DELAY;
         END;
      ELSE /* SINCE NEW$DELAY=0 */ DO;
         CALL ACTIVATE$TASK;
         END;
      END;
   RUNNING$TASK=0;
   RETURN;
   END CASE$TASK;


PREEMPT:PROCEDURE REENTRANT; /* ASSUMES INTERRUPTS DISABLED */
   IF PREEMPTED$TASK=0 THEN DO;
      IF (HIGH$PRIORITY$HEAD<>0) AND (RUNNING$TASK>=
             FIRST$LOW$PRIORITY$TASK) THEN DO;
         PREEMPTED$TASK=RUNNING$TASK;
         RUNNING$TASK=0;
         DO WHILE PREEMPTED$TASK<>0;
            CALL DISPATCH;
            END;
         END;
      END;
   RETURN;
   END PREEMPT;
```

```
DISPATCH:PROCEDURE REENTRANT; /* ASSUMES RUNNING$TASK=0 */
   DISABLE;
   IF HIGH$PRIORITY$HEAD<>0 THEN DO;
      RUNNING$TASK=HIGH$PRIORITY$HEAD;
      HIGH$PRIORITY$HEAD=TASK(RUNNING$TASK).PNTR;
      IF HIGH$PRIORITY$HEAD = 0 THEN HIGH$PRIORITY$TAIL = 0;
      CALL CASE$TASK;
      END;
   ELSE IF PREEMPTED$TASK<>0 THEN DO;
      RUNNING$TASK=PREEMPTED$TASK;
      PREEMPTED$TASK=0;
      END;
   ELSE IF LOW$PRIORITY$HEAD<>0 THEN DO;
      RUNNING$TASK=LOW$PRIORITY$HEAD;
      LOW$PRIORITY$HEAD=TASK(RUNNING$TASK).PNTR;
      IF LOW$PRIORITY$HEAD = 0 THEN LOW$PRIORITY$TAIL = 0;
      CALL CASE$TASK;
      END;
   ELSE RETURN;
   RETURN;
   END DISPATCH;
```

```
TASKØØ: PROCEDURE REENTRANT;/*ERROR CODE*/RETURN;END TASKØØ;

TASKØ1: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ1 CODE*/
   DISABLE;
   RETURN;
   END TASKØ1;

TASKØ2: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ2 CODE*/
   DISABLE;
   RETURN;
   END TASKØ2;

TASKØ3: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ3 CODE*/
   DISABLE;
   RETURN;
   END TASKØ3;

TASKØ4: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ4 CODE*/
   DISABLE;
   RETURN;
   END TASKØ4;

TASKØ5: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ5 CODE*/
   DISABLE;
   RETURN;
   END TASKØ5;

TASKØ6: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ6 CODE*/
   DISABLE;
   RETURN;
   END TASKØ6;

TASKØ7: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ7 CODE*/
   DISABLE;
   RETURN;
   END TASKØ7;
```

```
TASKØ8: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ8 CODE*/
   DISABLE;
   RETURN;
   END TASKØ8;

TASKØ9: PROCEDURE REENTRANT;
   ENABLE;
              /*TASKØ9 CODE*/
   DISABLE;
   RETURN;
   END TASKØ9;

              /*INITIALIZE*/

DISABLE;
DO STATUS=Ø TO 9;
   TASK(STATUS).PNTR=Ø;
   TASK(STATUS).STATUS=Ø;
   TASK(STATUS).DELAY=Ø;
   NEW$TASK,NEW$DELAY=Ø;
   HIGH$PRIORITY$HEAD,HIGH$PRIORITY$TAIL=Ø;
   LOW$PRIORITY$HEAD,LOW$PRIORITY$TAIL=Ø;
   RUNNING$TASK,PREEMPTED$TASK=Ø;
   END;

              /* MAIN LOOP */

DO WHILE TRUE<>FALSE;
   CALL DISPATCH;
   ENABLE;
   STATUS=STATUS;
   END;


END TM86;
```

REFERENCES

1. Hansen, Brinch, *Operating System Principles*, Prentice-Hall, Englewood, N.J., 1973.

2. Knuth, D. E., *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., 1969.

3. Wirth, Nicklaus, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood, N.J., 1976.

4. "PL/M-86 Programming Manual," Intel Corporation, 1978, manual order number 9800466A.

5. "RMX/80 User's Guide," Intel Corporation, 1977, manual order number 9800522B.

intel®

APPLICATION NOTE

AP-50

September 1979

Debugging Strategies and Considerations for 8089 Systems

© Intel Corporation, 1979

Debugging Stragegies and Considerations for 8089 Systems

Contents

INTRODUCTION

STATIC (OR FUNCTIONAL) DEBUGGING

Hardware Testing
External Processor Interface
Software Testing

REAL-TIME TESTING

Logic Analyzer Techniques

A REVIEW OF IOP OPERATION

Task Execution
Going from Instruction Execution into DMA
DMA Termination
Priorities/Dual Channel Operation

SUMMARY

Appendix I. CHECKLIST OF POSSIBLE PROBLEMS

Appendix II. BREAKPOINT ROUTINE AND CONTROL PROGRAM

Our thanks to John Atwood and Dave Ferguson, the authors of this note. Both John and Dave are members of Intel's 8089 design engineering group. Please direct any questions you may have to your local Intel FAE (field application engineer) or to MPO Marketing at Intel, Santa Clara.

INTRODUCTION

The Intel 8089 is the first integrated I/O processor available. This I/O processor (IOP) makes available the power of I/O channels, as used in mainframes and minicomputers, in a microcomputer form. Designed as part of the MCS-86™ family, the IOP can be interfaced with the MCS-80™ and MCS-85™ families as well.

An I/O channel is basically a processor remote from the main CPU, which independently runs I/O operations upon command of the CPU. To relate the 8089 to existing LSI components, it is similar to a microprocessor that is time-multiplexed with a DMA controller, but with two channels available. However, since the 8089 processor is optimized for I/O and multiprocessor operations, and the DMA has been made much more flexible than existing DMA controllers, a truly general purpose and powerful I/O control system is available on one chip.

Due to the uniqueness of the 8089, this application note was written to review debugging strategies and point out possible pitfalls when developing an IOP system. Debugging an IOP system is very similar to debugging microprocessor/DMA controller systems, and many of the techniques described here are standard microprocessor techniques. However, several factors are present which can complicate the debugging process:

1. Multiprocessor Operation

Although usable by itself, the IOP is designed to be used with other processors. All factors normally encountered with multiprocessor operation, including bus arbitration, processor communication, critical code sections, etc., must be addressed in the design and debug of an IOP system.

2. DMA Tie-In to IOP Program Execution

The relationship between IOP program execution and DMA transfers and termination is different from earlier DMA controllers and should be fully understood to properly run the system.

3. Dependency of Programs on Real-Time I/O Operations

Requirements by I/O devices for maximum data rates and minimum latency times force the software programmer to be aware of hardware timing constraints and can complicate program debugging.

4. Dual Channel Operation

Related to multiprocessor operation and real-time dependencies, the two independent channels available on the 8089 may have to be coordinated with each other to make the whole system function. Dependence of one channel on the other can also complicate debugging.

Due to the complexities of running in a real-time environment, as many steps as possible should be taken to facilitate debugging. A major help here is to make sure as much of the hardware and software as possible is working before running real-time tasks. This is a good practice anyway, but it should be reemphasized that a complex multichannel system can quickly get out of hand if more than a few things are not right.

An aid to debugging any system is a clean, well organized system design. The 8089 lends itself to structured, modular software interfaces to the host CPU, via the linked-list initialization structure, and parameter communication through the parameter block (PB) area. Some of the aspects of structured programming that aid debugging are:

- *Top Down Programming* — The functions done by low-level routines are well understood, and the number of program fixes, which can cause more errors, is minimized.
- *Program Modularity* — Small, easy to manage subprograms can be debugged independently, increasing the chance that the entire system will work the first time.
- *Modular Remoteness* — By having all program modules communicate only through a well-defined interface, one module's knowledge of the "inner workings" of another is minimized. System software complexity is reduced. Updates to program modules are more reliable, too.

Two major areas of debugging will be outlined here — static (or functional) debugging in which the hardware and software are not tested in a real-time environment, and real-time debugging. Applying a logic analyzer to IOP debugging will also be explained, and a review of IOP operation and potential problems will be done.

STATIC (OR FUNCTIONAL) DEBUGGING

The predominant errors in a system, when first tried out, are either errors in implementation (i.e., wrong hookups or coding errors), or an incorrect implementation (a wrong assumption somewhere). Most of these bugs can be found through static debugging techniques that are usually easier to work with than real-time testing.

Hardware Testing

Static hardware testing is done mainly to see if all individual parts of the system work. so the whole system will "play" when run. The level of testing can run from checking for continuity and shorts (which finds only hookup errors) to trying to move data around and running I/O devices from a monitor or special test programs (which can also find incorrect circuit design). In all but the simplest systems, the latter approach is recommended since it is a step towards software debugging.

Several approaches to hardware testing will be covered. Running diagnostic programs (such as a monitor) out of the IOP's host system, in both the LOCAL and REMOTE modes, will be covered. The case where the host system cannot support diagnostic software and must have an external processor to exercise the IOP and its peripherals will also be explained.

The case where the host system can run diagnostics or test programs that have interactive user I/O, such as a CRT terminal or teletype, provides the most straightforward way to test the IOP. Naturally, before these programs can be run, the basic hardware must be correct enough to run programs. When this point is reached, a monitor program can be used to exercise memory and I/O controllers on the system bus.

It should be mentioned that aids, other than just testing with software, are helpful for hardware debugging. While a necessity for real-time debugging, a logic analyzer is also a definite help for static hardware debugging. Its main use in hardware debugging is showing timing relationships between address or data paths and other signals. It is especially useful for functional software debugging, to be described shortly. The last debugging section outlines the use of an analyzer with the IOP. Of course, an oscilloscope, logic probes and pulsers, etc., can be used to trace out specific logic or timing problems.

LOCAL Mode

When the IOP is running in the LOCAL Mode, all I/O controllers and memory are accessible by the host or controlling CPU. Thus a standard monitor, such as the one supplied with the SDK-86 or available for the iSBC-86/12™ development kit, can exercise all hardware on the bus.* The breakpoint routines, however, will not work due to the different instruction set. The 8086 or 8088 is best suited for running the IOP in the LOCAL mode due to identical status lines and bus timing, as well as the Request/Grant line, which eliminates bus arbitration hardware. Figure 1 shows the general LOCAL mode configuration.

*The SDK-86 serial monitor is a good basis for a general 8086 monitor. The IOP cannot be used directly with the SDK-86, since the 8086 is running in the minimum mode. The SDK-86 can be converted to run in the maximum mode, if desired.

REMOTE Mode

From a system design standpoint, running the IOP in the REMOTE Mode is advantageous in that it removes the I/O bus cycles from the system bus. Normally, the remote I/O is not accessible to the host CPU. Until the IOP is able to run its own test programs to transfer data from the REMOTE bus to the system bus, I/O controllers and memory on the REMOTE bus will be invisible to the host. To get around this problem during prototyping, either an external processor interface can be used (see next section), or a temporary bypass can be made to access the REMOTE bus from the system bus.

Bypassing the normal REMOTE/SYSTEM interface is a handy technique for doing preliminary debugging on the REMOTE bus. This can be done by memory-mapping the IOP's I/O space into an unused portion of the host CPU's system memory space. When accessing this space, the IOP access to its own I/O space is disabled, and a separate set of address buffers, transceivers and bus control signal buffers are enabled. Reads and writes can then be done to the formerly inaccessible REMOTE bus by the host CPU.

A simple system (Figure 2) implements this bypassing scheme. It was designed for just forcing or examining devices on the REMOTE bus and may not read or write correctly if the IOP is simultaneously trying to do bus cycles. A more sophisticated arbitration system would permit reliable run-time checking also.

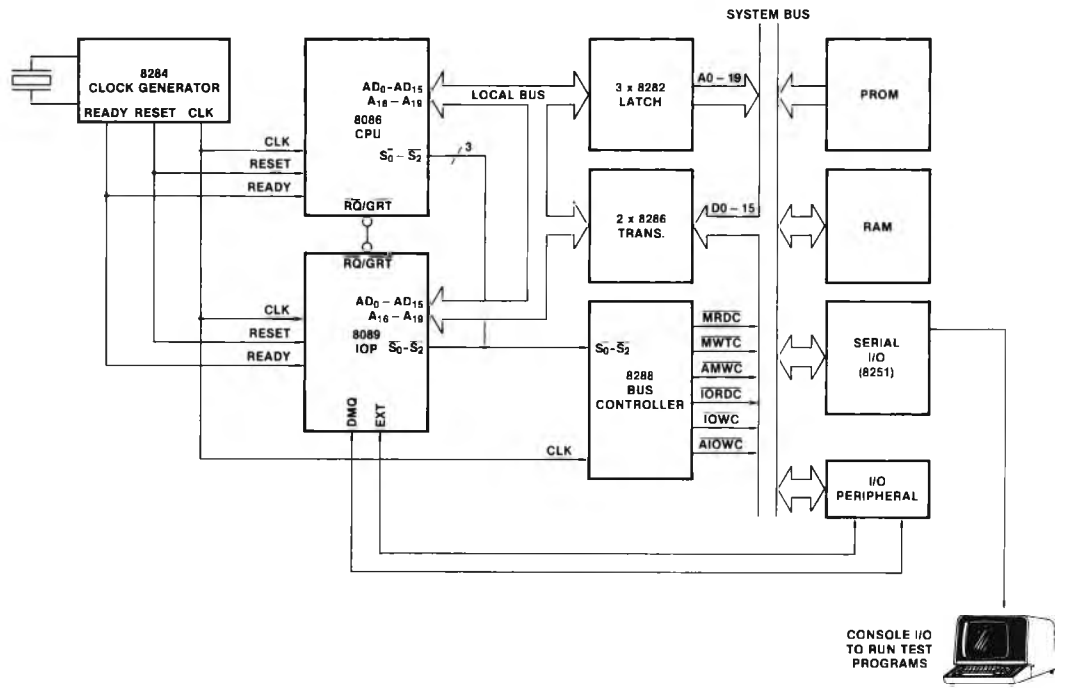


Figure 1. Generalized LOCAL Configuration—8086 in Max Mode

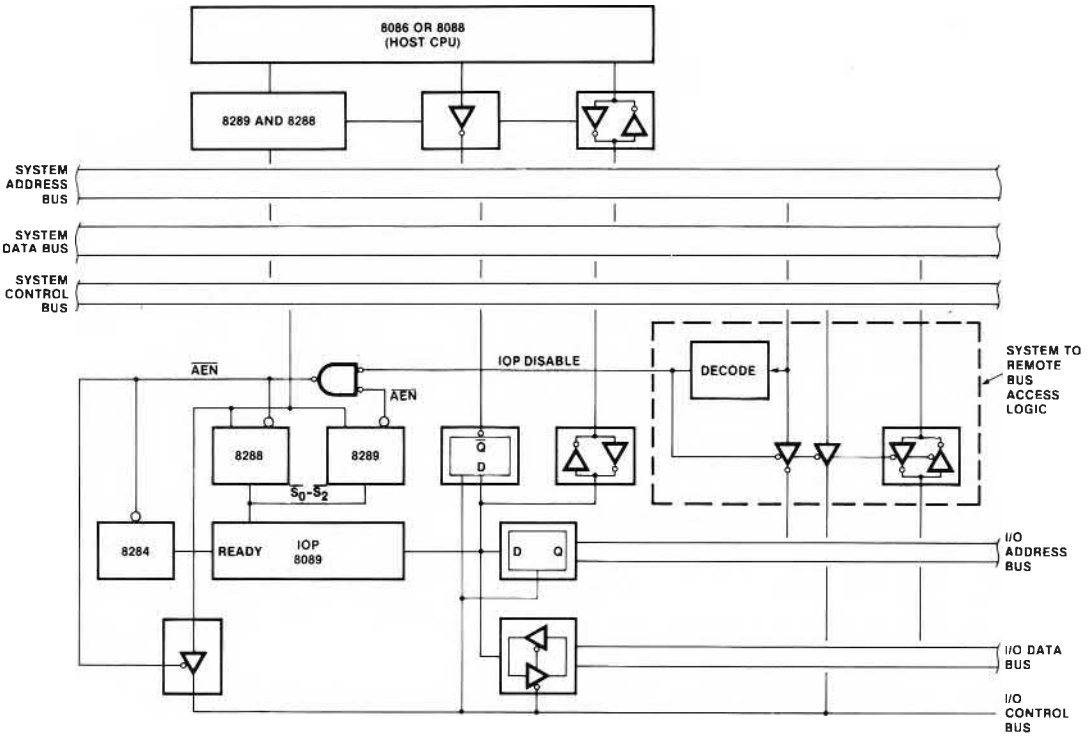


Figure 2. Remote Mode Bypass for Debugging

Running the IOP in the REMOTE mode, particularly if the MULTIBUS™ protocol is adhered to, has the advantage that the IOP can be exercised with any MULTIBUS-compatible processor. If the main processor is not amenable to being used as a debugging tool, another processor could be used to debug the hardware interface. If the microprocessor is of the same type as the intended host processor, software debugging can be done as well. A generalized REMOTE mode configuration using the MULTIBUS is shown in Figure 3.

External Processor Interface

A technique that can be used if the host processor cannot run any debugging or monitor routines is to have an external processor tie into the host processor's bus. This is useful if the main system CPU cannot run an interactive monitor or other debugging programs. If a MULTIBUS interface is being used, an 8289 bus arbiter and a set of address/data/control buffers can be used. A somewhat simpler system, similar to the remote bus access system mentioned above, could be used for static debugging of non-MULTIBUS systems. Again, if true bus arbitration is added (which brings us nearly to a MULTIBUS interface), it could also be used for run-time testing. Intel processors that have the MULTIBUS interface include the iSBC-80/20™, iSBC-86/12™, iSBC-80/10™, iSBC-80/05™, the Intellec® development systems, among others.

In the previously described systems, the external processor would disable the host CPU's access to the bus, either by some form of bus request or by a "brute force" disabling of the CPU's buffers. In the latter case, the external processor could only control the bus during a time that the CPU is halted, without destroying the program flow. Mapping the processor's memory space into the external processor memory space is the simplest method, but can impact programs being run on the external processor. If the processor under test utilizes the MULTIBUS interface (with bus arbitration), then a processor like the iSBC-80/30™ or iSBC-86/12™ could be used as the debug vehicle with no special hardware. A more flexible interface that would have less impact on the system memory space would have the addresses for the system under test generated from latches loaded by the I/O instructions from the external processor. This case must have software routines to interface to the I/O ports and handle the desired debugging routines (see Figure 4).

Software Testing

It is desirable to check as much of the IOP program as possible statically, since various tools and techniques are available which may not be usable during real-time

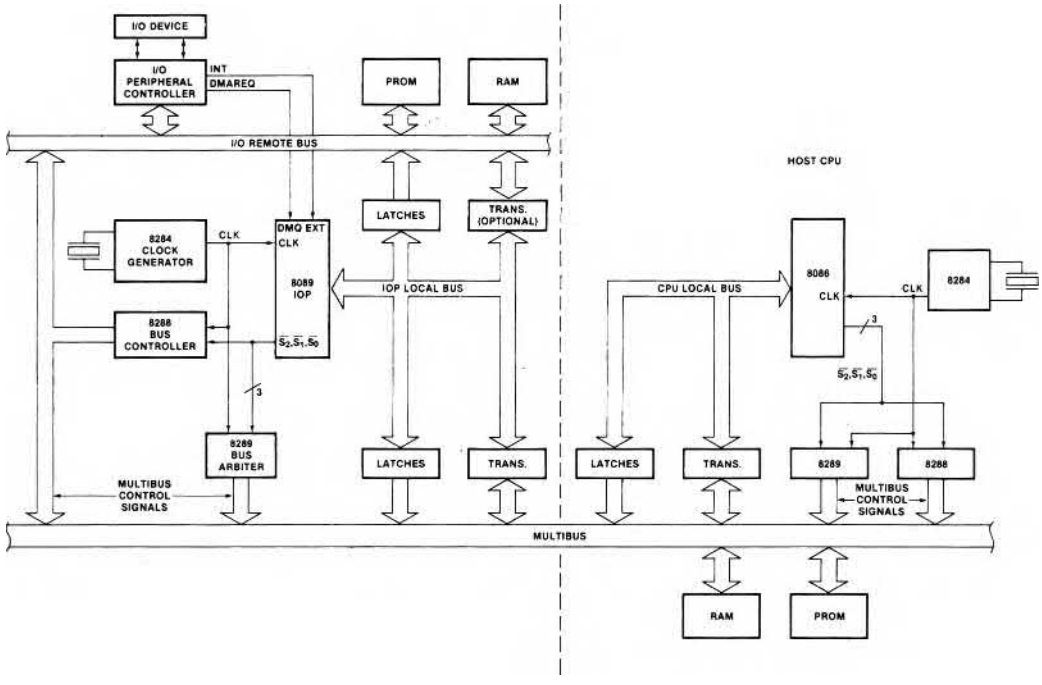


Figure 3. Generalized Remote Bus Using MULTIBUS Interface

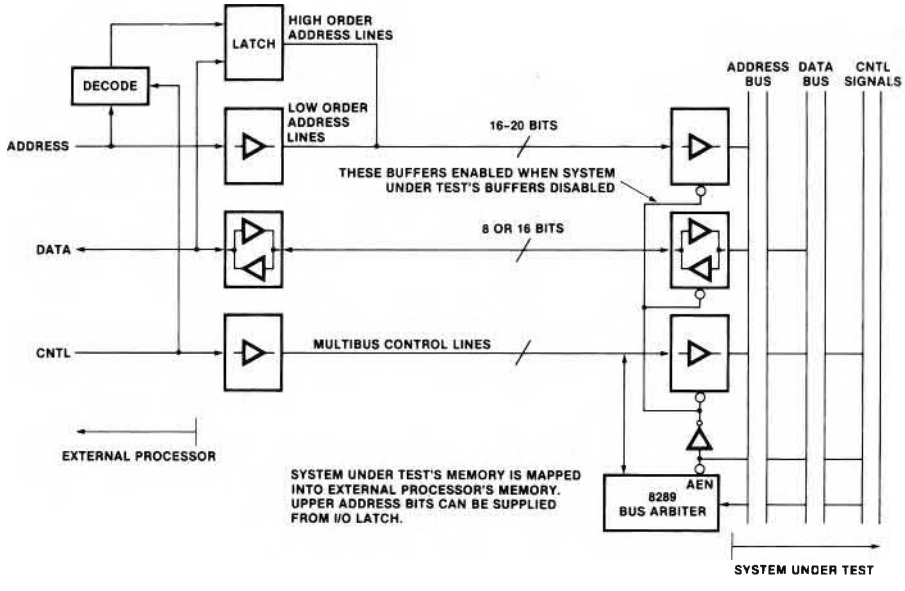


Figure 4. External Processor Interface

testing. This "static" software testing is not applicable to heavily I/O-dependent or DMA-dependent routines, but is best suited to longer computational or data handling routines. The idea is to test the correctness of algorithms, rather than seeing if the whole system runs.

There are two main approaches to functional software testing. One is to essentially run the program in real time and monitor program flow on a logic analyzer. The difference between this and real-time testing is that program subsections can be tested separately by using different TP (Task Pointer) starting addresses. If it is necessary to set up certain registers or parameters in memory, a small "setup" program can be run after initialization, which can load up registers or memory, then jump to the program section desired.

Another technique is to run the programs with breakpoint routines so that one can step through code segments and follow program execution. Software breakpoints are usually implemented by inserting a jump or restart to a monitor routine at the breakpoint location. This jump or restart is machine language dependent so, unfortunately, the existing breakpoint routines within monitors for the 8080 or 8086 are not applicable.

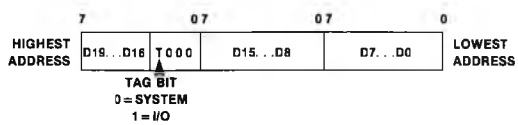
New routines tailored to the 8089 can be used, and, if done properly, can even be used to examine programs running on a REMOTE bus. Using breakpoints is somewhat complicated on the 8089 because the minimum instruction length is two bytes. There is no absolute CALL instruction, only a relative one (which would have to have its displacement recalculated each time it was used). But, with a several-byte absolute jump inserted at each place a breakpoint is desired, full breakpoint capabilities can be obtained.

There are many ways the breakpoints can be implemented. When a breakpoint is reached, the 8089 itself could output the machine state to a console through its own routines. Better suited to debugging, though, is a system that has the 8089 place its machine state in memory, alert the host processor, and then halt. The host then picks up the 8089's state and can treat it in the same way it runs its own breakpoint routines. Since the host processor is more likely to be running a monitor or some other kind of debugging routine (and most likely has at least temporary console I/O), it is the logical system to initiate and examine 8089 breakpoints. If the IOP is running in the REMOTE mode, and the host processor has access to the I/O bus via the scheme mentioned in the hardware debugging section, then IOP programs running on the REMOTE bus can be examined.

The breakpoint itself can consist of an escape sequence that is used to save the TP value and jump to the save routine, or just a jump to the save routine. This routine saves all register contents for the channel the breakpoint is in, signals the host processor, and stops the IOP. All user programmable registers (GA, GB, GC, IX, MC, BC, TP), as well as the pointer tags, are accessible. The PP (Parameter Pointer) and PSW are not normally accessible, but if the generation of the CA is such that the IOP can send itself a CA, then by sending a CA HALT, the PSW will appear at PP + 3. Remember that since the IOP doesn't have arithmetic or logical condition codes, the PSW is not as important as in other machines.

The most straightforward way to pass data from the IOP to the host processor is through the PB (Parameter Block) area since the PP will normally remain relatively fixed throughout the IOP program. In order not to infringe on the PB areas used by the programs, an area 18 bytes long should be allocated at the end of the PB block to hold the register contents. Using other areas to store the register data requires saving and reloading a pointer register as part of the breakpoint escape sequence.

The data returned from the breakpoint save routine will appear to the host processor as a sequential block of data in the PB area. Sixteen-bit data can easily be extracted, but 20-bit pointer data will have to be reconstructed from the move pointer (MOVP) format:



Several means are available to signal the host processor that a breakpoint has been reached. A bit could be set in memory or an interrupt sent to the CPU. The best way, though, is to use the BUSY flag (at CP + 1 or CP + 9). After starting the IOP, the BUSY flag is set to FF. When a breakpoint is reached, the IOP performs its save routine and does either a software or CA HALT. These result in clearing the BUSY flag, which then signals the CPU to obtain valid breakpoint data. The CPU can then restart the IOP by either a CA START or CA CONTINUE.

The breakpoint routine outlined above will work for a "one-shot" test. However, to be more useful as a general purpose debugging tool, some refinements must be added. To keep from destroying the program whenever a breakpoint is placed, the supervisory program running from the host processor must save the IOP code that is occupied by the escape sequence. When the breakpoint is completed and IOP execution is to resume, the host program restores the IOP code, sets the TP in the CB area back to where the breakpoint was placed, and sends a CA START. Since the length of each instruction can be easily found from bits 1–4 of the opcode, a single stepping function can also be done.* By the time this is implemented, the host program is becoming a full-fledged debugging routine. Appendix 3 describes a debugging program that makes use of the ideas presented here.

Breakpoint routines can be quite useful, but some restrictions and limitations should be mentioned. The processor examining the breakpoints must have access to the IOP program memory, either directly, or through IOP programs that simulate direct access. The program memory must be in RAM. The breakpoint must be

*The formula for length of instructions is: length (in bytes) = 2 + 1 (if bits 1,0 = 01) + 1 (if bits 3,2 = 01) + 2 (if bit 3 = 1) + 2 (if LPDI).

placed on an instruction boundary, and multiple breakpoints must not be placed so that they overlap. There may be some impact on the PB area. CA generation may have to be different than usual. But, despite these limitations, the breakpoints offer a useful and more conventional software debugging tool than analyzers.

REAL-TIME TESTING

Running an IOP program in its final environment with real I/O devices is the true test of dynamic operation. The program is no longer in a static, isolated environment. The demands of DMA and multiprocessing may reveal unplanned timing dependencies or critical section problems. There may also be sections of hardware or software, which couldn't be tested statically, that may have bugs. The whole purpose of static or functional testing is to dig these problems out while convenient debugging tools can be used. Since there are no simple techniques for real-time debugging, the use of a logic state analyzer and techniques to fully understand the IOP's real-time operation will be emphasized.

Multiprocessing operations and real-time asynchronous I/O requests can cause the timing complexity of the system as a whole to rise beyond the point of complete comprehension by an individual. It is then essential that techniques to ensure correctness are used. These include good design methods, especially a clean, well-structured design, as well as good testing. A thorough test requires the attitude that the system should be tested for failures, rather than tested for correctness. In other words, one should try to make the system fail, tests should be chosen that will put the worst stress on critical timing areas.

The best way to do this is to write a diagnostic program that puts the CPU, IOP, and I/O devices through the worst conceivable timing and program combinations. Ideally, the program should be self-checking so that it can be run without supervision, printing any data or program errors that occur, much like a memory test.

The two main real-time problem areas are insufficient data rates or latency, and critical section problems. To test for data rate problems, run the system clock at its lowest expected frequency and use memory and I/O with maximum expected wait states. Identify the tightest program timings and try to have these sections coincide with worst case DMA or other heavy bus utilization (see dual channel operation later). Critical section problems can occur when two independent processors communicate with each other with improper "handshaking." This can result in one processor missing another's message, or even having both processors hang up, waiting for each other to go ahead. The 8089 provides aids to these problems, including the TSL instruction (to implement semaphores) and the BUSY flag. However, any interprocessor communication (including one channel of the IOP to the other) should be checked. Beware of cases when one processor is running considerably slower than the other (due to DMA overhead or chained instruction sequences).

The techniques for real-time debugging evolve from functional testing using a logic analyzer. For all but the simplest systems, an analyzer is essential, since it can graphically show program execution and timing relationships during real-time execution. Another aid is a delayed oscilloscope. Triggering the scope from the logic analyzer, the delay can be adjusted so that any signal in the system can be monitored.

To facilitate the use of the logic analyzer, especially if its memory is not very deep or when using it to trigger an oscilloscope, a repetitive system can be used to continually update the display. Using a repetitive reset helps to debug the software-hardware interface, since oscilloscope or logic analyzer probes can be readily moved around the circuit to observe new signals without manually retriggering the display. At its simplest, the reset to the host processor can be strobed, say every 10 ms. The processor will then provide the two channel attentions (CAs) that are needed to initialize the IOP. Where this isn't feasible, the CAs can be externally forced by either a string of one-shots or a simple processor with timing loops (such as a SDK-85 or SDK-86). See Figure 5 for initialization timing.

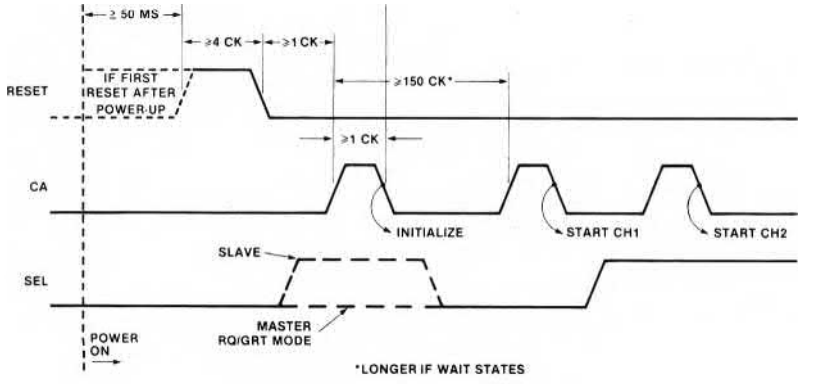


Figure 5. Initialization Input Sequence

Memory protection of the IOP and system programs is helpful when debugging DMA operation. It is quite easy for runaway DMA to wipe out memory. Another precaution to avoid this problem is to set an upper limit on the number of bytes transferred by always specifying a byte count termination.

Logic Analyzer Techniques

In the absence of other powerful debugging systems, the logic analyzer has shown to be an extremely useful tool. Because of its importance in debugging an IOP system, some basic techniques and observations that relate to monitoring IOP operation will be reviewed here. The particular brand or type of analyzer used is not too important, but would be desirable to have the following features:

- At least a 24-bit data width
- Flexible triggering and qualification control
- Display after triggering on a sequence of states
- Capability for hexadecimal data display

It is best to hook up to the address/data lines at the IOP, as opposed to looking at the separate address and data lines, since 39 lines would be required just to look at address, data and status lines. The three lower status lines should be monitored to show the type of bus cycle being run. Other lines can be connected where needed, at places like the DRQ lines, the EXT lines or other lines related to the system.

For general purpose debugging, triggering the analyzer on the rising edge of the IOP clock shows the most useful data concerning bus cycles. Of course, using the falling edge may be necessary to check certain signals, particularly ones that are active only while the clock is low. The following discussion is based on sampling data on the clock's rising edge.

One should be careful when setting up the triggering for the analyzer that the desired event is what is displayed and not a later event with the same trigger word. This can happen when the logic analyzer is in the repetitive trigger mode. It may retrigger before the system actually resets. A sequence restart feature is helpful.

The basis of following program execution and DMA on a logic analyzer is to follow an 8089 bus cycle, which is identical to a 8086 and 8088 bus cycle. The following diagram shows a typical 8089 bus cycle.

For general purpose debugging, displaying every clock is useful, but for quickly finding one's way around a program, the analyzer can be qualified so that only instruction fetches (status = 100 or 000), with ALE active, are trapped. A much more compact display of execution flow results.

BUS CYCLE	A16-19	AD0-15	S0-2	
				PREVIOUS ADDRESS, UPPER STATUS
	X	X X X X	1 1 1	IDLE STATUS
T1	F	F 0 1 0	1 0 1	20-BIT ADDRESS = FF010,
T2	E	F F F F	1 0 1	LOWER STATUS = MEMORY DATA READ
T3	E	A A 5 0	1 1 1	16-BIT DATA RETURNED = AA50
T4	E	F 0 1 0	1 1 1	ADDRESS REMAINS IN CHIP OUTPUT LATCH AFTER END OF BUS CYCLE

DATA NOT READY YET

UPPER STATUS INDICATES: NON-DMA, CH1

As mentioned earlier, on a 16-bit bus, most instructions starting on odd addresses won't show the first fetch, since the internal queue is in use. It is a good idea in that case to use only even instruction boundaries as trigger words. When following dual channel operation, one should keep an eye on the upper status bits (S3-S6), since S3 indicates which channel is running (0 = CH1, 1 = CH2), and S4 indicates DMA/non-DMA transfer (0 = DMA, 1 = non-DMA).

A REVIEW OF IOP OPERATION
(With things to look out for)

When trying to get an unfamiliar system going for the first time, it is too easy to stumble on apparent problems that are really just unexpected operation modes or peculiarities of the machine. For this reason the basic principles of IOP operation will be reviewed here with special emphasis on possible problem areas or pitfalls that a user might encounter when debugging a 8089 system. The topics are covered generally in the order encountered when bringing up a system. For complete details of operation and some design examples, see the 8086 Family User's Manual.

RESET

RESET must be active (HIGH) for at least four clocks in order to fully initialize all internal circuitry. On power up, RESET should be held high for at least 50 microseconds. The chip is only ready to accept a Channel Attention (CA) one clock after RESET goes inactive.

Note that the SEL pin is sampled on the falling edge of the first CA after RESET to tell the 8089 whether it is a master (0) or a slave (1) for its request/grant circuitry. If a master, it will assume it has the bus from the beginning. If a slave, it will strobe the $\overline{RQ}/\overline{GT}$ Line to request the bus back and will not start any bus transfers until it has been granted the bus. If the $\overline{RQ}/\overline{GT}$ line is not being used, make sure the IOP comes up in the master mode.

Initialization

Upon the first CA after reset, a sequence of instructions is executed from an internal ROM. These instructions pick up parameters and load data from the linked list sequence (Figure 6). The instruction sequence is essentially:

MOVB SYSBUS from FFFF6
LPD System Configuration Block (SCB) from FFFF8
MOVB SOC from (SCB)
LPD Control Pointer (CP) from (SCB) + 2
MOVBI "00" to CP + 1 (clears BUSY flag)

Remember that four bytes must be fetched during an LPD. If on a 16-bit bus, with even addressed boundaries, only two fetches are needed. Otherwise (8-bit bus or odd boundaries), four fetches are needed.

Even though no bus cycles are run to fetch these instructions, the CH1 Task Pointer (TP) appears on the address latches during the short internal fetch periods. On power up, this value is meaningless, but if a repetitive RESET is used, the TP remains unchanged from the end of the last program run. See Figure 6 for the start of a typical initialization sequence as viewed on a logic analyzer.

Bit 0 in the SYSBUS field sets the actual (or physical) system bus width that the IOP expects. In the 8-bit mode, only byte accesses are made, and all 8-bit data should appear on the lower eight data lines. In the 16-bit mode, word accesses can be made (if the address is even), all data on even addresses appears on the lower eight data lines, and all data at odd addresses appears on the upper eight.

Bit 0 in the SOC field sets the physical width for the I/O bus. The same rules for the system bus apply here. Note that these bits should reflect the actual hardware implementation and are not to be confused with the DMA logical widths set by the WID instruction.

The R bit (bit 1) in the SOC field is used to change the mode of the $\overline{RQ}/\overline{GT}$ circuitry. When the IOP is on the same bus as an 8086, it is required to have the R bit be 0, with the 8086 as the master and the 8089 as the slave.

CA	A_{19}-A_0	S_3-S_0	T	COMMENTS
1	FF F F F	111		Trigger CLK 1
1	FF F F F	111		
	FF F F F	111		
	FF F F F	111		
	E0 0 0 0	111		Bus un-tristated
	E0 0 0 0	111		
	FF C 6 D	111		
10				TP to latch
CK	FF C 6 D	111		
	FF F F 6	101	T1	Address loaded to latch
	EF F F F	101	T2	Data not ready yet (nothing on bus)
	EF F 0 1	111	T3	SYSBUS loaded into chip (01)
	EF F F F	111	T4	Nothing on bus
	EF F F 6	111		After bus cycle, address remains in latch
	EF F F 6	111		
	FF C 6 D	111		TP is loaded to latch, even though fetches are from internal ROM
14				
CK	FF C 6 D	111		
	FF F F 8	101	T1	Address to latch
	EF F F F	101	T2	
	EF F F 0	111	T3	1st 2 bytes of LPD data fetched (FFF0)
	EF F F F	111	T4	
	EF F F 8	111		
	EF F F 8	111		
	EF F F 8	111		
	EF F F 8	111		
	FF F F A	101		
	EF F F F	101		
	EF F F A	111		2nd 2 bytes of LPD data fetched (FFFA)
6				
CK	EF F F A	111		
	FF C 6 D	111		

Figure 6. Start of Initialization Sequence On a 16-Bit Bus

The master (8086 or 8088) can never take the bus away from the slave (8089); only the slave can give back the bus. In other words, during DMA transfers, the 8089 would not have the bus taken away. This is the only mode compatible with the 8086 or 8088.

When two IOPs are being used on the same bus, the $\overline{RQ}/\overline{GT}$ circuity can be put into an equal priority mode by setting the R bit to one. A slave can only be granted the bus if the master is doing unchained instructions or running idle cycles. The master can request the bus back from the slave at any time. The slave grants it if doing unchained instructions or if it is idling. The master and slave are put on essentially the same priority.

At the end of initialization, the "BUSY" flag of CH1 is cleared. For systems where the 8086 is waiting for the initialization sequence to end before giving another CA, it can set the BUSY flag high prior to initialization. The BUSY flag going low is a sign that the IOP is ready for another CA. It is important to remember that the IOP will not respond to, nor latch, a CA during an initialization sequence.

Channel Attentions

The main system processor initiates communications with the IOP through the Channel Attention (CA) line. As mentioned earlier, the first CA after system RESET initializes the IOP. All subsequent CAs cause the IOP to do a two-step process. It first fetches the Channel Control Word (CCW) from the appropriate channel at (PP) for channel 1 or (PP + 8) for channel 2. (SEL at the time of CA falling determines the channel for all following actions.) The lower three bits of the CCW Command Field (CF) are examined and then cause the IOP to execute the desired function.

Command Field (CF)

Control of task block programs is accomplished through the command field. The various CF functions are:

CF

000 — Examine other field only and set BUSY flag

001 — Start task program in I/O space

011 — Start task program in system memory

The start command causes the following instructions to be executed out of the internal ROM:

LDP CP from (CP) + 2 (CH1) or + 10 (CH2)

LDP TP from (PP) (for TP in system) or

MOVB TBP from (PP) (for TBP in I/O)

MOVBI "FF" to (CP) + 1 or + 9 (set BUSY flag)

111 — HALT channel. BUSY flag cleared to "00"

110 — HALT channel. Save state of machine and clear BUSY flag by executing:

MOVP TP to (PP)

MOVB PSW to (PP) + 3

MOVBI "00" to (PP) + 1 or + 9

The channel will HALT and the machine will continue execution on the other channel or go to idle if the other channel is idle.

101 — Continue channel. The channel is revived after a HALT by executing:

MOVP TP from (PP)
MOVB PSW from (PP) + 3
MOVBI "FF" to (CP) + 1 or + 9
(set BUSY flag)

Do not do a CONTINUE after initialization without doing a CA START first since the (PP) register in CH1 is used as a temporary register (to hold SCB) and is only correctly loaded by a CA START.

The upper 5 bits in the CCW will have affect if CF = 000 or upon a CA START. Some things to note about these upper fields are:

- *Priority Bit* — If both channels are doing tasks of the same overall priority, the tasks with the higher priority bit will run. If the priority bits are the same, execution will alternate between the two channels.
- *BLL Bit (Bus Load Limit)* — Keeps nonchained instructions from occurring more often than once every 128 clocks. However, channel attention or termination cycles, even on the other channel, may disrupt the exact time interval to the next instruction.

It should be noted that the setting or clearing of the BUSY flag occurs after the loading or storing of registers, so that in a system where the main CPU uses the BUSY flag as a form of semaphore to tell when the IOP is truly finished, there is no danger that the SCB, CP, PP or TP could be changed before the IOP loads them.

Also since DMA termination cycles and chained instruction execution have a higher priority than CA, it is possible for CA to be "shut-out" by these higher priorities running on the other channel. However, since CA is always latched (except during initialization), it won't be forgotten.

How Can a Channel be Halted?

Sometimes a channel may stop its operation unexpectedly. To see what could cause this, and to show the impact of halting a channel, the various ways of stopping a channel are explained:

HALTED CHANNEL — If the channel has never started after initialization, if it has received a CA HALT command or a software HALT, channel operation is suspended. If the other channel can run, it will, otherwise idle cycles will run. Only a CA START or CONTINUE can resume operation.

WAITING FOR A DMA REQUEST — If the channel is in a source or destination synchronized DMA transfer mode, it will wait until DRQ is active before running its synchronized transfer. To minimize the impact on the overall throughput of the chip, the other channel can run during these DRQ wait periods.

WAITING TO GET THE BUS BY $\overline{RQ}/\overline{GT}$ — If the IOP has given the bus away via $\overline{RQ}/\overline{GT}$, it won't initiate any bus transfers until it has the bus back. The machine will run up to just before T1 of a bus clock cycle and will three-state its address/data and status pins until it has been granted the bus.

WAITING FOR READY — When running bus transfers, READY is sampled at T3 of a busy cycle. If inactive, the whole chip will wait until READY goes active.

The last two cases of waiting (or "wait" states) stop the whole chip and do not permit the other channel to run. However, with READY inactive or with the bus not acquired, there is not much that can be done on the other channel anyway. These two cases only stop the chip when running bus cycles. Any internal operations can proceed without having the bus or with the system not READY.

Note the difference between when the chip is HALTed when using $\overline{RQ}/\overline{GT}$ and an external arbiter (8289) for bus arbitration. Not having the bus due to $\overline{RQ}/\overline{GT}$ will inhibit the bus cycle from even starting. Since the 8289 stops the chip by forcing $\overline{AEN}$ inactive, which goes through the 8284 clock generator to force READY inactive to the IOP (or 8086/8088), a bus cycle has already been started, with ALE asserted, and the address on the address/data lines. When the bus is obtained, operation proceeds at T3 of the bus cycle.

As will be mentioned later, many invalid opcodes will cause the machine to hang up. In these cases the address/data lines will point to where the bad opcode was fetched.

Task Execution

Although optimized for fast and flexible DMA operation, the IOP is also a full-fledged microprocessor. The 8086 Family User's Manual deals with programming strategies and other details. Some of the things to be noted during debugging will be mentioned here.

Instruction Fetching

Unlike the 8085 (but like the 8086), the 8089 labels all fetches from the instruction stream, whether OPCODE, offset, displacement, or literal data, as an instruction fetch on the status lines. In some cases, such as MOV R,I and ADD R,I, the instruction fetch time greatly exceeds execution time because literals are treated as instruction fetches. When following programs on a logic analyzer, triggering on status = 100 or 000 (instruction fetch) and a known program address is the handiest way to trace the flow of the program.

When running programs on a 16-bit bus, a 1-byte queue register comes into play, saving the upper byte fetched from the last instruction fetch, if not used by the previous instruction. This reduces fetch time and bus utilization since the odd byte doesn't need to be fetched again. An internal four-clock cycle fetches data from the queue. Like the internal ROM fetches, the task pointer is put out on the address/data lines, but no bus cycle is run.

The queue can have some possible unexpected affects that have to be taken into account during debugging. These apply only to 16-bit systems and are:

1. Instructions that start on odd boundaries will not likely have bus cycles run to fetch the odd byte unless jumped to, unless preceded by LPDI (which clears the queue), or an instruction that modifies the task pointer is executed. The latter causes the queue to be cleared so that part of an old instruction won't become part of the new one.
2. There is a queue register for each channel so loading or clearing the queue on one channel has no affect on the other channel's queue.
3. The second word of immediate data fetched by a LPDI is done during a pseudo-instruction fetch cycle that cannot make use of the queue or already fetched data. Thus, if on an odd boundary, fetching an LPDI will be byte, word, byte, byte, byte, and the queue will not be loaded.

When Can the Other Channel Interrupt Instruction Execution?

This will be explained more in the "dual channel" operation section, but a few points will be mentioned here. All instructions are made up of internal cycles, with each cycle composed of two to eight clocks. Each bus cycle is one internal cycle, but there can be internal cycles with no comunications to outside the chip. Internal cycles will be extended by the number of wait states in each bus cycle. Between any of these cycles, DMA from the other channel can intervene if the priorities permit it. Instruction fetching and execution can only interrupt instructions on the other channel when the instruction has been completed, not between internal cycles.

Registers

All the registers have some special purpose use in the Instruction Execution or DMA, but all except the CC register can be used as general purpose registers during instruction sequences. A few are loaded specially:

- *CP* — Is only loaded during an initialization sequence. There is one CP register that handles both channels. (All others are duplicated, one set for each channel.)
- *PP* — Is only properly loaded during a CA START command. It holds the SCB value after the initialization sequence.
- *TP* — This is included as part of the registers in the RRR field, but cannot be operated on unless you plan on having your program execution jump around. Everytime this is operated on, the queue is cleared. The TP is loaded from two words (address and displacement) on a CA START, LPD, or LPDI, and loaded from 3-byte MOVP format (see illustration on page 5) on a CA CONTINUE, and can be operated on using any register oriented instructions.

The following registers are loaded during program execution, but can have special effects:

- *CC* — The only thing that affects instructions in the CC register is the chaining bit. If chaining doesn't matter (if only one channel is being used without channel attentions, for example), then the CC register can be general purpose. However, for portability of programs, it is strongly suggested not to use the CC register except for altering DMA parameters and chaining.
- *MC* — Is a general purpose 16-bit register, but is also used to do a masked comparison either for DMA search/match termination or for the JMCE and JMCNE instructions.
- *BC, IX* — Both general purpose 16-bit registers. In instructions that reference memory using the AA field, if AA = 11, the IX register is incremented by the number of bytes fetched or stored.
- *Pointer Registers (GA, GB, GC and TP)* — Are 20-bit registers, but can also be used as 16-bit registers. Adds will carry into the upper 4 bits, but other operations (COMP, OR, AND) are done only on the lower 16 bits. Note that when used as pointers to system memory, it is possible to add a large 16-bit number to the pointer and to put the pointer into another 64K block of memory.

Sign Extension

All program data brought into the chip, either literals or displacements in opcodes, or program data fetched from memory, is sign-extended. Offsets used for calculating addresses are not sign extended. Any 8-bit data brought in has bit 7 sign-extended up to bit 19. Sixteen-bit data is sign-extended from bit 15 to bit 19. It is important to note this, because it can affect logical operations. For example, if one wanted to OR 0084H with 1234H in register GC, you couldn't do ORBI GC, 84H, because bit 7 would sign-extend into the upper byte. Instead, you should code ORI, 0084H to do this properly (note that this has a word for the immediate data). The non-ADD operations will cause the upper four bits of the pointer registers to be invalid since the upper four bits of the ALU come only from the adder.

Tags

It should be noted that the way the IOP knows which bus to access (system or I/O) is via the Tag bit associated with the pointer register used. The TAG can only be set in these ways: loading as a 16-bit register (MOV R,M, MOV R,I) sets TAG to I/O space, loading as a pointer (LPD, LPDI) sets TAG to a system space), or bringing the TAG in from memory by a MOVP instruction.

Effects of Invalid Opcodes

The upper 6 bits of the 2-btye opcode actually determine which opcode will be executed. If these bits are a valid opcode, but lower bits are invalid, the chances are good that the bad bits will be ignored. But if the upper six bits are invalid, there is a very good chance that the chip will hang up and stop execution in that channel. The only way to get out of this mode is to reset the chip. If this hang-up occurs, it can usually be traced because the last address of the instruction fetch will still be on the

address/data lines, showing where the program went astray.

Going from Instruction Execution into DMA

The XFER instruction places the current channel into the DMA mode after the next instruction. This permits one last instruction to start up an I/O device (start CRT display on an 8275, for example). However, in order for the IOP to get setup for DMA, the GA, GB, and CC registers should not be altered during this last instruction. Failure to observe this will probably result in an improper first DMA fetch. The WID instruction can be placed after XFER.

DMA Transfers

Incrementing/Non-Incrementing pointers

A memory or I/O pointer can be made to increment for each byte transferred during DMA or it can remain fixed. Incrementing is used primarily for memory block transfers, and non-incrementing is used to access I/O ports.

B/W Mode

Each DMA transfer is composed of separate fetch and store cycles so that 8/16-bit data can be assembled and disassembled, and translation and termination may also be easily handled. There are four possible transfers or B/W modes. They are:

B – B — 1 byte fetched, 1 byte stored
B/B – W — 2 bytes fetched, 1 word stored
W – B/B — 1 word fetched, 2 bytes stored
W – W — 1 word fetched, 1 word stored

The B/W mode used depends on the logical bus width (selected by the WID instruction), address boundary, and incrementing mode.

All systems with 8-bit physical buses will run in the B/B mode. On 16-bit physical buses the other modes are possible, depending on the logical widths selected. Note that the logical bus width can be different than the physical bus width since there are cases where an 8-bit peripheral may be used on a 16-bit bus. The selection of the logical width, and not the physical width, is what determines the B/W mode. Thus it is the responsibility of the programmer not to program an invalid combination (i.e., don't specify a 16-bit logical width on an 8-bit physical bus).

Any transfer on an odd boundary will be B/B but if the pointer is incrementing and on a 16-bit logical bus, after the first transfer, the pointer will be on an even boundary. The IOP will then try to maintain word transfers in order to transfer data as effeciently as possible. See the user's manual for details. The change in B/W mode occurs only after the first transfer or, as explained in the termination section, upon certain byte count terminations.

Synchronization

In the unsynchronzied mode, transfers occur as fast as priorities will allow. This is the IOP's "block-move" mode. Most I/O peripherals only want a DMA transfer on demand; the DRQ lines, along with synchronization specified, will handle this need. Source synchronization is used for I/O reads and destination synchronization is used for I/O writes.

If the IOP is waiting for a DMA request, it will run programs or DMA on the other channel, or execute idle cycles if nothing is pending. If running idle cycles when the DRQ comes, the transfer starts five clocks after DRQ is recognized. If running DMA or instructions on the other channel, the DRQ cannot be serviced until the current internal cycle is done, and may require a maximum of 12 clocks (without bus arbitration or wait states).

Consecutive DRQ-synchronized DMA transfers on the same channel are separated by four idle clocks (assuming no other delays) by an internal sampling mechanism. This happens between the 2-byte fetches on source-synchronized B/B-W cycles, and between the two stores on destination-synchronized W-B/B cycles. This delay between consecutive DMA cycles allows adequate time for proper acknowledgement of the current DMA request before the next request is processed. On destination-synchronized DMA, this isn't a problem, but on source-synchronized DMA, there will be four extra clocks per transfer. Unless one is running right at the speed limit, this won't be a problem. Near the maximum data rate, unsynchronized transfers can be used, with synchronization done by manipulating the READY line.

Translate Mode

When the translate bit is set, the data fetched during DMA will be added to the GC register. This new pointer will in turn be used to fetch, via a seven clock extra fetch cycle, new data, which will then be stored. Translate is only defined for byte transfers. The bytes are added to GC as a positive offset, so a lookup table for translating data can be a maximum of 256 bytes long. Even if the data to be translated falls within a smaller range (such as ASCII code), a full 256-byte lookup table is recommended so that erroneous data can be flagged and controlled.

Translate can be run on any of the B/B transfer modes, so it is useful for doing block translation within program execution as well as translation directly to or from an I/O port.

DMA Termination

One of the powerful features of the IOP is its varied DMA termination conditions and their close tie-in with resuming Instruction Block programs. However, because of the multitude of DMA modes, care must be taken in predicting the exact termination parameters. Various things to be careful about will be outlined here.

Byte Count (BC) Termination

The BC register is decremented for every byte transferred whether or not BC termination is set. If BC termination is set, the last transfer done is the one that results in BC being zero. To avoid the problem of missing BC = 0 on word transfers, if BC is odd between every transfer, the IOP detects when BC is 1, and forces the last transfer to be in the B/B mode. Since both the fetch and store cycles are complete, the source and destination pointers point exactly to the next byte or word that would have been fetched.

Masked Compare (MC) Termination

An MC termination occurs when a pattern matches (or doesn't match, depending on mode selected) the lower half of the MC register (the match pattern) with only the bits that are enabled by the upper half of MC (the mask pattern) contributing to a match. Thus the masked bits can be "don't cares" in both the data byte and the match byte.

The masked comparison is only done on store (deposit) cycles. Any bytes transferred (in B/B or W-B/B mode) will be compared. But, since the MC comparison is done on only one byte, any words stored (W-W or B-B/W) have only their lower byte compared. This may be fine, but if not, make the destination logical width 8 bits.

Just like BC termination, the pointers will point to the next data to be transferred. The BC will also be decremented correctly, except if the termination occurs on the first byte of a W-B/B transfer. In this case the BC will be decremented as if the entire transfer (both bytes) had taken place.

The store cycle that causes an MC termination will be lengthened by two extra clocks (or by one extra clock if there are wait states), to allow time to set up the termination cycle.

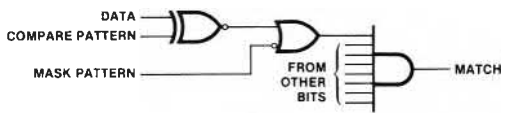


Figure 7. Masked Compare Logic for 1-Bit

External (EXT) Termination

External termination allows the I/O device or controller to use its own conditions to generate a termination. Basically, the IOP will halt DMA as soon as it recognizes an EXT terminate, even if a transfer is only partially complete. There might be concern that multibyte cycles (W-B/B or B/B-W) might have data lost if an EXT terminate stopped the store cycle. In unsynchronized DMA this would happen, but this mode is typically not used with I/O controllers that could generate external terminations. In synchronized DMA modes, it is assumed that the I/O controller will only do a DRQ for valid data transferred, and that it won't give an EXT terminate with its DRQ active. In destination synchronization, the possible problem occurs in the W-B/B mode, where EXT terminate comes after the first store but before the second. This is fine, since even though data was overfetched, the proper amount was actually transferred. In source synchronization, the B/B-W mode raises problems since if an EXT terminate came after the first byte fetched and before the second byte fetched, normally no store cycles would be done at all, thus losing the first byte fetched. In this case (i.e., source synced, DRQ inactive, and 1 byte already fetched), a single byte store cycle is run before the termination cycle, ensuring data integrity.

In order to prevent an invalid signal level from becoming trapped from the asynchronous EXT term lines, two clocks of delay and signal conditioning are done on these lines. In addition, a termination cycle can only be started at certain times during DMA (or TB on the other channel — see dual channel operation section). The EXT terminate lines should be valid eight clocks before the start of the DMA cycle to be stopped.

EXT is sampled even when the IOP is running something on the other channel. Remember though, that despite the high priority of termination, the current instruction on the other channel has to finish before the termination cycle is run. Simultaneous EXTs on both channels result in CH1 termination being done first.

In order to have enough time to process a byte count termination, the BC register is always decremented during DMA fetch cycles. Because of this, external or MC terminations that occur during W-B/B cycles will result in the byte count always being decremented by two, even if only one byte is stored. This also occurs in the block-to-block or block-to-port B/B-W modes. To find the exact number of bytes transferred, the source pointer address can be checked in the block-to-port and block-to-block modes during B/B-W cycles and in the block-to-port W-B/B mode. The destination pointer address can be used to find the number of bytes transferred in the port-to-block and block-to-block modes during W-B/B cycles.

Termination Cycles and Multiple Terminations

Upon termination, the user can run different task block programs, depending on which type of termination has occurred, by specifying an appropriate termination offset. That is, instruction fetching will begin after a termination cycle starting at either the TP value before the DMA started, TP + 4 or TP + 8. These offsets permit long or short jumps to termination routines.

The termination cycle is an add immediate instruction that runs from the internal ROM and adds the proper offset to the TP. It is 15 clocks long for TP + 4 and TP + 8 termination and 12 clocks long for TP + 0 termination.

As mentioned earlier, EXT terminate must come a certain time before the end of a transfer to ensure that the next transfer doesn't start. If it comes in time and MC termination also occurs on the current transfer, then the termination cycle with the largest offset is run. A simultaneous BC terminate cycle will have priority over MC and will result in the running the BC termination program.

Priorities/Dual Channel Operation

The IOP can share its internal and external hardware between two separate channels. The user sees two identical IOP channels with all registers, machine flags, etc., independent of the other channel. The only register in common is the CP register, loaded by the initialization sequence. The mechanism for achieving dual channel operation is time multiplexing between the two channels.

Since interleaving two channels affects their response time to external events and since interfacing to these events is the prime purpose of the IOP, several means of adjusting the priorities of the channels are provided.

Before going into the priority algorithms in detail the four types of cycles that are affected by the priorities will be outlined:

1. *DMA Cycles* — Any type of DMA transfer cycle, including single transfers and translate cycles. DMA can be interrupted after any bus transfer by the other channel.
2. *Instruction Cycles* — Any instructions that have been fetched out of I/O or system memory. Instruction cycles are made up of internal cycles, each two to eight clocks long (assuming no wait states). Some cycles may not run bus transfers. Instructions can be interrupted by DMA after any one of the internal cycles, but can only be interrupted by instructions on the other channel (normal ones or ones from internal ROM) after the current instruction is completed.
3. *Termination Cycle* — Performed when DMA transfers end and instructions resume (except on single transfers).
4. *Channel Attention Cycles* — Performed when channel attention is given, performs actions specified in the CCW field. Both termination and CA cycles can be interrupted by DMA after any internal cycle, but can only be interrupted by instruction cycles after the complete sequence of internal cycles is done.

Termination and channel attention cycles as well as the initialization cycle (which never runs concurrently with other operations) are sequences of instructions fetched from an internal ROM.

Recognizing the higher importance in doing DMA, termination and (to a lesser extent) CA cycles, the following priority scheme is built into the IOP. Any channel that has a higher-priority operation will run continuously until done. If both channels are running the same priority, execution will alternate between them.

Highest Priority

1. DMA transfers, termination, chained instructions
2. Channel attention cycles
3. Instruction cycles
4. Idle cycles

Lowest Priority

Two ways exist to alter the priority scheme. One way is to utilize the priority bits for each channel. If one is greater than the other, that channel will run at the expense of the other if both channels are otherwise running at the same priority. Thus the P bit only has effect on channels running at the same priority level.

If one wants to run instructions along with or in place of DMA on the other channel, the other technique is to set the chaining bit (in the CC register) which brings the instruction priority up to the level of DMA. Care should be taken with this since now CAs are at a lower priority than instructions and will not be serviced unless that channel goes idle. Chaining will also lock out normal instructions on the other channel. Chaining should thus be used with care.

In order to reduce the possibility of shutting out channel attentions, an exception is made to the above priority scheme. After every DMA transfer, whether synchronized or unsynchronized, the IOP will service any pending CA. However, chained task block execution will still shut out CAs on the other channel.

What is the importance of priorities? Well, as an example, let's say that we are running long periods of non-time-critical block moves (via DMA) on one channel and running short bursts of DMA that must be serviced promptly on the other channel. With the default priorities, the short DMA channel bursts would be interleaved with the longer DMA, reducing the maximum transfer rate for both channels. If, however, the priority bit was one on the burst mode DMA and zero on the other, the bursts would be serviced continuously at the fastest possible data rate.

An even more critical case would be the same low priority, long DMA transfers on one channel with DMA on the other channel that must terminate, run a short instruction sequence, and resume DMA again within a short, fixed time. (This might be the case in running a CRT display with linked list processing between lines.) Normally, the low priority, long DMA could indefinitely block the short TB sequence. By setting the high-priority channel's priority bit to one and putting it into the chained instruction mode, the low priority channel would stop its DMA entirely so that the termination/instruction sequence could run.

When establishing the priorities to be run, care should be taken that both channels will run successfully under a worst case combination. This can be tricky when the channels are running asynchronously with fast data rates and/or short latencies, but must be taken into account. Of course, running only one channel on the IOP is an easy solution, but if more than one IOP is being used in the system, the priorities and delays of the bus arbitration used (either $\overline{RQ}/\overline{GT}$ or an 8289 bus arbiter) must be taken into account. It may be found that the on-chip arbitration between the two channels is faster and more powerful than external arbitration.

SUMMARY

It is hoped that the material presented here will aid those who are putting together and debugging an 8089 IOP system, and help them in understanding the operation of the IOP. Many of the debugging techniques should be familiar to those who have worked with micro- and minicomputer systems before. Other debugging techniques not mentioned here, which work well with microprocessor systems, could be just as applicable to the 8089. The unique nature of the IOP among LSI devices warrants special consideration for its I/O functions and multiprocessor capabilities.

Appendix I

CHECKLIST OF POSSIBLE PROBLEMS

HARDWARE PROBLEMS

- Is RESET at least four clocks long?
- Are both V_{SS} lines connected to ground?
- Does the first CA falling edge come at least two clocks after RESET goes away?
- Does the second CA come at least 150 clocks (16-bit system, no wait states) after the first CA?
- Is READY correctly synchronized and gated by local/system bus lines?
- Is SEL correct for first CA so that IOP comes up correctly as master or slave?
- If two IOPs are local to each other, is a 2.7K pull-up resistor used on RQ/GT?

SOFTWARE PROBLEMS

- Are the initialization parameters in the initialization linked-list correct?
- Is BUSY flag being properly tested by host CPU software before modifying PB or providing a new command?
- Has the chaining, translate, or lock bit in the CC register been erroneously set?
- Have DMA termination conditions been met? The IOP could be trying to do endless DMA.

Appendix II

BREAKPOINT ROUTINE AND CONTROL PROGRAM

The debugging program described here is an example of the kind of software development tool that can be developed for the 8089 IOP. It was written to try out various breakpoint schemes, and has been used to debug an engineering application test system. The program is not meant to be the ultimate debugging tool, but is an example of what can be put together to utilize the breakpoint routine described earlier in the application note.

The debugging program was tested on a 8086-based system that emulates the SDK-86 I/O structure, and uses the SDK-86 serial monitor. This enables it to use the SDK-86 Serial Downloader to interface to an Intellec® development system on which the software was created. The 8086 system is interfaced via a MULTIBUS™ interface to an IOP running in the REMOTE mode. The remote bus access technique, mentioned earlier in this note, is implemented on this system, but was not used in the software debugging program.

The breakpoint routine uses a simple jump to a save routine. The PL/M-86 supervisory or control program handles the placement of the jump within the users program. Since it can not normally access the remote bus, all IOP programs to be tested must run out of system memory.

When the control program starts, it assumes the IOP has just been reset. It then prompts the user for the CP and PP values. After this, it sends the first (initialization) channel attention. It then asks the user for the channel to be run, and the starting and stopping addresses. After the stopping address has been entered, a Channel Attention Start is given. If the breakpoint is reached, a HALT is executed, and the control program prints the register contents. If the breakpoint hasn't been reached, the user can type any character, and a Channel Attention Halt will be sent to the IOP. If the IOP responds within 50 ms, the TP where it was halted is printed. Otherwise, the control program issues an error message. If, at any time, the user wants to get out of the program, typing an ESC will pass control back to the SDK-86 monitor. Figure 9 shows the flow of the control program.

Note that, unlike a single CPU debugging routine, having the 8086 supervise the 8089 enables a clean exit from crashed IOP programs. The program code where jumps had been placed are always restored. The control program is a good example of how the power of dual processors can be put to good advantage.

Comments within the control program indicate parameters that need to be changed to run on different systems. It should be noted that channel attentions are invoked by the recommended method of using an I/O write to a port to generate CA and using A0 for SEL.

Source and object files of this program are available through Intel's INSITE™ User's Program Library as program 8089 Break. 89 (number AD6).

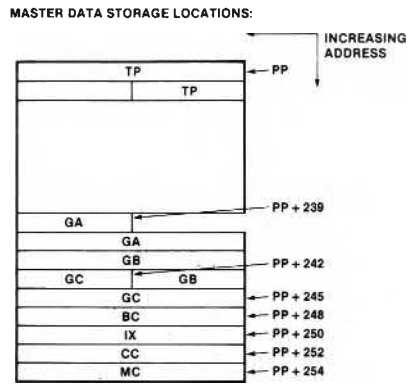


Figure 8. Breakpoint Routine to Run 8089 Program out of System Memory

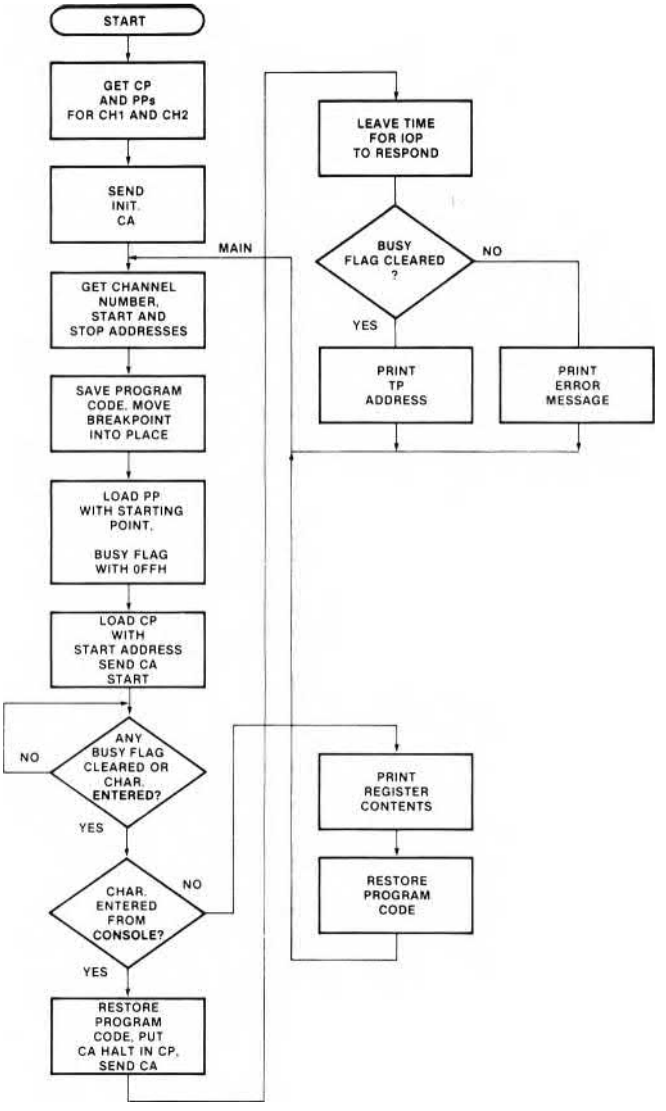


Figure 9. Breakpoint Routine to Run 8089 Program out of System Memory

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   1

ISIS-II PL/M-86 X103 COMPILATION OF MODULE BREAKPOINT
OBJECT MODULE PLACED IN BREAK.OBJ
COMPILER INVOKED BY    F1 PLM86 BREAK.SRC PAGEWIDTH (100)


            $TITLE ('8089 BREAKPOINT ROUTINE')
            /*
                8089 BREAK POINT PROCEDURE
                WRITTEN BY DAVE FERGUSON 2/2/79   REV 2  8/14/79
                INTEL CORPORATION

                                                                          */

  1         BREAK$POINT:
                DO;
  2   1         DECLARE I BYTE;
  3   1         DECLARE SAVECODE (4) WORD; /*BUFFER FOR STORAGE*/
  4   1         DECLARE ONEPP POINTER; /* CHAN ONE PP */
  5   1         DECLARE TWOPP POINTER; /* CHAN TWO PP */
  6   1         DECLARE STARTBYTES (4) BYTE, /* BUFFER FOR START ADDRESS */

  7   1         DECLARE STARTPOINTER POINTER; /* POINTER FOR START ADDR. */
  8   1         DECLARE ENDPOINTER POINTER; /* POINTER FOR END ADDR. */
  9   1         DECLARE PRESENT POINTER AT (@INPNTR); /* POINTER BUFFER */
 10   1         DECLARE TRUE LITERALLY 'OFFH',FALSE LITERALLY '000H';


                /*  YOU MUST CONFIGURE YOUR I/O STRUCTURE AND
                   SYSTEM TO MATCH THE PROGRAM OR VISA VERSA */
 11   1         DECLARE CRTSTATUS LITERALLY 'OFFF2H', /* 8251 STATUS PORT */
                CRTDATA LITERALLY 'OFFFOH', /* 8251 DATA PORTS */
                CHANATTEN LITERALLY 'OFAH', /* CHANNEL ONE CHANNEL ATTENTION PORT */
                 /* CHANNEL TWO CHANNEL ATTENTION PORT = CHANATTEN + 1 */
                CHANNELONE LITERALLY '00H',
                CHANNELTWO LITERALLY '01H',

                /* ASCII IS A STRING OF HEX CHARACHTERS IN ASCII FORM */
                ASCII (*) BYTE DATA ('0123456789ABCDEF'),
                TITLE$STRING (*) BYTE DATA (0AH,0DH,'8089 BREAKPOINT VER 1.0',
                                 0AH,0DH,'TYPE ESCAPE TO RETURN TO MONITOR.',
                                 0AH,0DH,0),
                CHANGIVEN (*) BYTE DATA ('CHANNEL ATTENTION GIVEN TYPE ANY KEY TO ABORT.'
                                ,0AH,0DH,0),
                BKREACHED (*) BYTE DATA (0AH,0DH,'BREAKPOINT REACHED',0AH,0DH,0),
                GETCP (*) BYTE DATA ('INPUT CP IN HEX',0AH,0DH,00),
                GET$PP (*) BYTE DATA ('INPUT PP IN HEX FOR ',00H),
                GETSTART (*) BYTE DATA (0AH,0DH,'INPUT STARTING ADDRESS IN HEX',0AH,0DH,00H),
                STOPADDR (*) BYTE DATA ('INPUT END ADDRESS IN HEX',0AH,0DH,00H),
                CHANNUMBER (*) BYTE DATA (0AH,0DH,'CHANNEL ONE OR TWO? ',00H),
                ABORT (*) BYTE DATA (' FATAL ERROR - IOP DOES NOT RESPOND TO CHANNEL',
                ' ATTENTION  RE-INITIALIZE SYSTEM ',0),
                ABORTAT (*) BYTE DATA (' TP WAS ',0),
                ONE (*) BYTE DATA (' CHANNEL ONE',0AH,0DH,00H),
                TWO (*) BYTE DATA (' CHANNEL TWO',0AH,0DH,00H),
                GASTRING (*) BYTE DATA  ('GA = ',00H),
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   2

                GBSTRING (*) BYTE DATA ('GB = ',00H),
                GCSTRING (*) BYTE DATA ('GC = ',0H),
                BCSTRING (*) BYTE DATA  (0AH,0DH,'BC = ',00H),
                IXSTRING (*) BYTE DATA (0AH,0DH,'IX = ',00H),
                CCSTRING (*) BYTE DATA (0AH,0DH,'CC = ',00H),
                MCSTRING (*) BYTE DATA (0AH,0DH,'MC = ',00H)

 12   1         DECLARE CHAR BYTE,
 13   1         DECLARE ONETWO BYTE;

                /* SDKMON IS A PLM TECHNIQUE USED TO FORCE THE CPU INTO AN
                   INTERUPT LEVEL 3.  IN ORDER TO USE THIS THE PROGRAM MUST
                   BE COMPILED (LARGE).  */
 14   1         SDKMON:
                PROCEDURE;
 15   2          DECLARE HERE (*) BYTE DATA (0CCH),
                /* THIS IS AN INT.  3  */
                   WHERE WORD  DATA( HERE);
 16   2          CALL WHERE;
 17   2         END;


                /* CO SENDS A CHAR TO THE CONSOLE WHEN READY */
                /* THIS ROUTINE IS WRITTEN TO RUN VIA THE SERIAL
                   PORT OF AN SDK86 */
 18   1         CO:
                PROCEDURE (C);
 19   2          DECLARE C BYTE;
 20   2          DO WHILE (INPUT(CRTSTATUS) AND 01H) = 0; END;
 22   2          OUTPUT (CRTDATA) = C;
 23   2         END;

                /* CI GETS A CHARACHTER FROM THE USER VIA THE SERIAL PORT */
                /* CI AUTOMATICALLY ECHOS THE CHARACHTER TO THE USER CONSOLE */
 24   1         DECLARE ESCAPE LITERALLY '1BH';

 25   1         CI: PROCEDURE BYTE;
 26   2           DO WHILE (INPUT(CRT$STATUS) AND 02H) = 0; END;
 28   2           CHAR = INPUT (CRTDATA) AND 07FH;
 29   2           CALL CO(CHAR);
 30   2           IF CHAR = ESCAPE THEN CALL SDKMON; /* GO TO SDK MONITOR */
 32   2           RETURN CHAR;
 33   2         END;

                /* VALIDHEX CHECKS THE VALIDITY OF A BYTE AS A HEX CHARACHTER*/
                /* THE PROCEDURE RETURNS TRUE IF VALID FALSE IF NOT */

 34   1         VALIDHEX:
                PROCEDURE (H) BYTE,
 35   2          DECLARE H BYTE;
 36   2          DO I=0 TO LAST(ASCII);
 37   3           IF H=ASCII(I) THEN RETURN TRUE;
 39   3          END;
 40   2          RETURN FALSE;
 41   2         END;
```

```
                  /* HEXCONV CONVERTS A HEX CHARACTER TO BINARY FOR MACHINE USE.
                     IF THE CHARACTER IS NOT A VALID HEX CHAR, THE PROCEDURE RETURNS
                     THE VALUE OFFH */
 42   1           HEXCONV
                  PROCEDURE (DAT) BYTE;
 43   2             DECLARE DAT BYTE;
 44   2             IF VALIDHEX(DAT) <> OFFH THEN RETURN TRUE;
 46   2             DO I=0 TO LAST(ASCII);
 47   3               IF DAT = ASCII(I) THEN RETURN I;
 49   3             END;
 50   2             END;

                  /* HEXOUT WILL CONVERT A VALUE OF TYPE BYTE TO AN ASCII STRING
                     AND SEND IT TO THE CONSOLE */

 51   1           HEXOUT
                  PROCEDURE(C);
 52   2             DECLARE C BYTE;
 53   2             CALL CO(ASCII(SHR(C,4) AND OFH));
 54   2             CALL CO(ASCII(C AND OFH));
 55   2           END;


                  /* WORDOUT CONVERTS A VALUE OF TYPE WORD TO AN ASCII STRING
                     AND SENDS IT TO THE CONSOLE */
 56   1           WORDOUT
                  PROCEDURE (W);
 57   2             DECLARE W WORD;
 58   2             CALL HEXOUT(HIGH(W));
 59   2             CALL HEXOUT(LOW(W));
 60   2           END;

                  /* GETADDRESS IS A PROCEDURE TO GET AN ADDRESS FROM THE CONSOLE.
                     THIS PROCEDURE WILL ONLY CONSIDER THE LAST 5 CHARACHTERS ENTERED
                     */
 61   1           DECLARE INPNTR (4) BYTE;

 62   1           GET$ADDRESS
                  PROCEDURE POINTER;
 63   2             DECLARE BUFF BYTE;
                  /*CLEAR ALL VALUES TO ZERO */
 64   2             INPNTR(0) = 0;
 65   2             INPNTR(1) = 0;
 66   2             INPNTR(2) = 0;
 67   2             INPNTR(3) = 0;

 68   2             BUFF = 0;
 69   2             DO WHILE BUFF <> TRUE;
                  /* THIS SEQUENCE OF SHIFTS ALLOW THE USER TO TYPE IN FIVE
                     OR MORE CHARACHTERS TO BECOME THE ACTUAL POINTER FOR 8089
                     OR 8086. THIS PROCEDURE RETURNS THE LAST FIVE IN PROPER
                     SEQUENCE  STORED IN INPNTR(0-3).  THE STORAGE
                     IS AS FOLLOWS
                          1 THE LAST CHARACTER INPUT GOES INTO
                            THE LOW FOUR BITS OF INPNTR(0).
                          2 THE NEXT TO LAST CHARACTER GOES INTO
                              THE LOW FOUR BITS OF INPNTR(2).
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   4

                          3. THE THIRD CHARACTER INPUT GOES INTO
                             THE HIGH FOUR BITS OF INPNTR(2)
                          4. THE SECOND CHARACHTER INPUT GOES INTO
                            THE LOW FOUR BITS OF INPNTR(3)
                          5. THE FIRST CHARACTER INPUT GOES INTO
                            THE UPPER FOUR BITS OF INPNTR(3).
                       THE 86 SHIFTS INPNTR (2,AND3) LEFT FOUR BITS AND ADDS THIS TO
                       INPNTR(0) RESULTING IN THE ADDRESS THE USER TYPED IN.  */

 70   3                INPNTR(3) = (SHL(INPNTR(3),4) OR (SHR( INPNTR(2),4) AND 0FH));
 71   3                INPNTR(2) = (SHL(INPNTR(2),4) OR (INPNTR(0) AND 0FH));
 72   3                INPNTR(0) = BUFF;
 73   3                BUFF = CI;
 74   3                BUFF = HEXCONV(BUFF);
 75   3              END;
 76   2              CALL CO(0AH);  /*LINE FEED TO CRT*/
 77   2              CALL CO(0DH);  /*CARRIAGE RET TO CRT*/
 78   2              RETURN PRESENT;  /* PRESENT IS A POINTER TO THE ARRAY INPNTR.  */
 79   2            END;

                  /* STRINGOUT IS A PROCEDURE TO SEND THE CONSOLE AN ASCII STRING
                     ENDING IN THE VALUE 00.  STRINGOUT NEEDS A VALUE OF TYPE POINTER
                  */

 80   1            STRING$OUT:
                   PROCEDURE(PTR);
 81   2              DECLARE PTR POINTER,STR BASED PTR (1) BYTE;
 82   2              I = 0;
 83   2              DO WHILE STR(I) <> 0;
 84   3               CALL CO(STR(I));
 85   3               I = I + 1;
 86   3              END;
 87   2            END;


 88   1            DECLARE TAGIS (*) BYTE DATA (' OPERATING IN ',0),
                           TAGISONE (*) BYTE DATA ('IO SPACE',0AH,0DH,0),
                           TAGISZERO (*) BYTE DATA ('SYSTEM SPACE',0AH,0DH,0);
                  /* TAGTEST TESTS THE TAG BIT AND SENDS A MESSAGE TO THE CONSOLE
                     THE TAG IS LOCATED IN BIT THREE.  A TAG BIT OF ONE MEANS THE
                     POINTER IS TO I/O SPACE, AND A TAG BIT OF ZERO MEANS THE
                     POINTER IS TO SYSTEM SPACE */
                  /* THE CALLER MUST DECIDE WHICH BYTE HAS THE TAG AND PASS IT TO TAGTEST */

 89   1            TAGTEST:
                   PROCEDURE(TEST);
 90   2              DECLARE TEST BYTE;
 91   2              CALL STRINGOUT(@TAGIS);
 92   2              IF (TEST AND 01000B) <> 0
                     THEN
 93   2              DO;
 94   3                CALL STRINGOUT(@TAGISONE);
 95   3              END;
                     ELSE
 96   2              DO;
 97   3                CALL STRINGOUT(@TAGISZERO);
 98   3              END;
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   5

 99   2          END;
100   1          DECLARE SAVE$ADDR LITERALLY '2000H',
                     SAVE$SEG LITERALLY '00C0H';

101   1          DECLARE BREAK89 (4) WORD DATA (9B81H,0891H,SAVE$ADDR,SAVE$SEG);
                 /* BREAK89 IS AN 4 WORD ESCAPE SEQUENCE TO ADDRESS 2000H
                    CONSISTING OF AN  LPDI TP,SAVE$ADDR WITH SEGMENT
                        LOCATED AT 0C00H.    */

                 /* BRKRTN IS 33 BYTES OF CODE THAT STORES ALL REGISTERS
                    AS FOLLOWS
                     GA STORED    AT PP + 239
                     GB STORED    AT PP + 242
                     GC STORED    AT PP + 245
                     BC STORED    AT PP + 248
                     IX STORED    AT PP + 250
                     CC STORED    AT PP + 252
                     MC STORED    AT PP + 254
                       */

102   1          DECLARE BRKRTN (33) BYTE AT (02C00H)
                * 02C00H IS ACTUALLY (SAVE$ADDR + (SHL(SAVE$SEG),4)), AND SHOULD
                    MATCH ADDRESS AND SEGMENT WHERE BREAK ROUTINE IS WANTED    */
                  INITIAL
              (03H,09BH,0EFH,023H,09BH,0F2H,043H,09BH,0F5H,063H,087H,0F8H,0A3H,087H,
              0FAH,0C3H,087H,0FCH,0E3H,087H,0FEH,020H,048H) ,
103   1          DECLARE PP POINTER,
104   1          DECLARE PPP BASED PP (1) BYTE;

105   1          START$PRGM
                   PROCEDURE(ONE$TWO,PPP),
106   2            DECLARE ONE$TWO BYTE,PPP POINTER,
                   WHERE BASED PPP (1) BYTE,

107   2                WHERE(0) = START$BYTES(0);
108   2            WHERE(1) = 0,
109   2            WHERE(2) = START$BYTES(2);
110   2            WHERE(3) = START$BYTES(3),
111   2            CPDAT((ONE$TWO) * 8) = 3,
                   /* IF ONETWO = 1 THEN OUTPUT TO PORT 0F8H, IF ONETWO
                      IS 0 THEN OUTPUT TO PORT 0FAH */
112   2            OUTPUT(CHANATTEN + (ONETWO )) = 0,
113   2            CALL STRINGOUT(@CHANGIVEN);
114   2          END;

                 /* THIS PART OF THE PROGRAM ALLOWS THE USER TO DEFINE THE
                   CP,PP OF EACH CHANNEL */
115   1          DECLARE BREAKOUT BASED ENDPOINTER (1) WORD;

116   1          DECLARE CP POINTER,
117   1          DECLARE CPDAT BASED CP (1) BYTE;

118   1          DECLARE ONEPPDAT BASED ONEPP (1) BYTE,
119   1          DECLARE TWOPPDAT BASED TWOPP (1) BYTE;


120   1          CALL STRINGOUT (@TITLESTRING);
```

```
121   1          CALL STRINGOUT(@GETCP);
122   1          CP = GETADDRESS;
123   1          CALL STRINGOUT(@GETPP);
124   1          CALL STRINGOUT(@ONE),
125   1          ONEPP = GETADDRESS;
126   1          CALL STRINGOUT(@GETPP);
127   1          CALL STRINGOUT(@TWO);
128   1          TWOPP = GETADDRESS;
129   1          OUTPUT (CHANATTEN) = 0; /* INITIALIZATION CA */

130   1      MAIN:
                 CALL STRINGOUT(@CHANNUMBER);
131   1          CHAR = CI; /* GET CHANNEL NUMBER */
132   1          IF (CHAR AND 01H) <> 0 /* CHECK BIT ZERO TO DEFINE
                                  CHANNEL NUMBER */
                 THEN DO;
134   2            CALL STRINGOUT(@ONE);
135   2            ONETWO = CHANNEL$ONE;
136   2          END;
                 ELSE
137   1           DO;
138   2             CALL STRINGOUT(@TWO),
139   2             ONETWO = CHANNEL$TWO;
140   2          END;

141   1          CALL STRINGOUT(@GET$START); /* GET STARTING ADDRESS
                                  FROM USER */

142   1          STARTPOINTER = GETADDRESS;
143   1          DO I = 0 TO 3; /* MOVE STARTING ADDRESS INTO CP AREA */
144   2            STARTBYTES(I) = INPNTR(I),
145   2          END;
146   1          CALL STRINGOUT(@STOPADDR); /* GET STOP ADDRESS
                                  FROM USER */

147   1          ENDPOINTER = GETADDRESS;
148   1          DO I = 0 TO 3; /* MOVE CODE TO SAFE AREA  */
149   2            SAVECODE(I) = BREAKOUT(I);
150   2          END;
151   1          DO I = 0 TO 3;
152   2            BREAKOUT(I) = BREAK89(I), /* MOVE ESCAPE SEQUENCE INTO PLACE */
153   2          END;
154   1          CPDAT(1) = OFFH; /* SET  CHANNEL ONE BUSY FLAG */
155   1          CPDAT(9) = OFFH; /* SET CHANNEL TWO BUSY FLAG */
156   1          DO CASE ONETWO;
157   2          PP = ONEPP;
158   2          PP = TWOPP;
159   2          END;
160   1          CALL START$PRGM(ONE$TWO,PP);
                 /* WAIT FOR ONE OF THE FOLLOWING
                    1.CPDAT(1) = 0  CH1 NOT BUSY
                    2.CPDAT(9) = 0 CH2 NOT BUSY
                    3.THE 8251 REC. BUFFER IS FULL BECAUSE USER HAS DEPRESSED A KEY
                 */
161   1          DO WHILE (  (CPDAT(1) AND CPDAT(9)) AND (NOT (INPUT(CRT$STATUS) AND 02H))) = OFFH;
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   7

 162   2            END;
 163   1          IF (INPUT(CRT$STATUS) AND 02H) <> 0
                     THEN
 164   1            DO;
 165   2              CHAR = CI,
 166   2              DO I = 0 TO 3;
 167   3                BREAKOUT(I) = SAVECODE(I);
 168   3              END;
                  /* IF ONETWO = 0 THEN PUT CHA HLT IN CPDAT(0)
                     IF ONETWO = 1 THEN PUT CHA HLT IN CPDAT(8)
                  */
 169   2              CPDAT(ONE$TWO *8) = 06H;
                  /* IF ONETWO = 0 THEN OUTPUT TO PORT 0FAH, IF ONETWO
                     IS 1 THEN OUTPUT TO PORT 0FBH.
                  */
 170   2              OUTPUT(CHANATTEN + ONETWO) = 0;
 171   2              DO I = 0 TO 5;
 172   3               CALL TIME(100);
 173   3              END;

                  /* IF BUSY FLAG HAS BEEN CLEARED, THEN A CA HALT&SAVE
                     WAS EXECUTED.   IF SO, PRINT SAVED TP; IF NOT, ABORT  */

 174   2              IF CPDAT(SHL(ONETWO,3) + 1) <> 0   /* CHECK BUSY FLAG */
                      THEN
 175   2              DO,
 176   3               CALL STRINGOUT(@ABORT),
 177   3              END;
                      ELSE
 178   2              DO;
 179   3               CALL STRINGOUT(@ABORTAT);
 180   3               CALL CO(ASCII(SHR(PPP(2),4))); /* UPPER NIBBLE OF ADDR
                                    STORED BY HALT */

 181   3               CALL HEXOUT(PPP(1)); /* MIDDLE BYTE OF ADDR
                                 STORED BY HALT */

 182   3               CALL HEXOUT(PPP(0)); /* LEAST SIG BYTE OF ADDR
                                 STORED BY HALT */

 183   3              END;
 184   2              CPDAT(ONETWO * 8) = 3H; /* CA START IN CPDAT(0) OR CPDAT(8) */
 185   2              GO TO MAIN,
 186   2            END;
 187   1          DO;

 188   2           CALL STRINGOUT(@BKREACHED),

 189   2           CALL STRINGOUT(@GASTRING);
 190   2           CALL CO(ASCII(SHR(PPP(241),4)));
 191   2           CALL HEXOUT(PPP(240));
 192   2           CALL HEXOUT(PPP(239));
 193   2           CALL TAGTEST(PPP(241)),

 194   2           CALL STRINGOUT(@GBSTRING);
 195   2           CALL CO(ASCII(SHR(PPP(244),4)));
 196   2           CALL HEXOUT(PPP(243));
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   8

 197   2          CALL HEXOUT(PPP(242));
 198   2          CALL TAGTEST(PPP(244));

 199   2          CALL STRINGOUT(@GCSTRING);
 200   2          CALL CO(ASCII(SHR(PPP(247),4)));
 201   2          CALL HEXOUT(PPP(246));
 202   2          CALL HEXOUT(PPP(245));
 203   2          CALL TAGTEST(PPP(247));

 204   2          CALL STRINGOUT(@BCSTRING);
 205   2          CALL HEXOUT(PPP(249)),
 206   2          CALL HEXOUT(PPP(248));

 207   2          CALL STRINGOUT(@IXSTRING);
 208   2          CALL HEXOUT(PPP(251));
 209   2          CALL HEXOUT(PPP(250));

 210   2          CALL STRINGOUT(@CCSTRING),
 211   2          CALL HEXOUT(PPP(253));
 212   2          CALL HEXOUT(PPP(252));

 213   2          CALL STRINGOUT(@MCSTRING);
 214   2          CALL HEXOUT(PPP(255));
 215   2          CALL HEXOUT(PPP(254)),

 216   2         END;
                 /* RESTORE CODE TO ORIGINAL LOCATION  */
 217   1         DO I = 0 TO 3;
 218   2           BREAKOUT(I) = SAVECODE(I);
 219   2         END.

 220   1         GO TO MAIN,

 221   1      END.


MODULE INFORMATION

     CODE AREA SIZE.     = 0619H   1561D
     CONSTANT AREA SIZE = 01EFH    495D
     VARIABLE AREA SIZE = 0020H     32D
     MAXIMUM STACK SIZE = 0014H     20D
     427 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
8089 ASSEMBLER

ISIS-II 8089 ASSEMBLER X004 ASSEMBLY OF MODULE AP50_BREAKPOINT_ROUTINE
OBJECT MODULE PLACED IN :F0:BRKASM.OBJ
ASSEMBLER INVOKED BY ASM89.4 BRKASM.SRC

                                  1 NAME     AP50_BREAKPOINT_ROUTINE
0000                              2 BRKPNT   SEGMENT
                                  3 ;*************************************
                                  4 ;   BASIC 8089 BREAKPOINT ROUTINE
                                  5 ;     BY JOHN ATWOOD    REV 3  8/13/79
                                  6 ;     INTEL CORPORATION
                                  7 ;*************************************
                                  8
                                  9 ;   THE FOLLOWING CODE IS CONTAINED IN THE PL/M-86
                                 10 ;   CONTROL PROGRAM(BREAK.89) AND IS ASSEMBLED HERE
                                 11 ;   TO ILLUSTRATE HOW THE ESCAPE SEQUENCE AND SAVE
                                 12 ;   ROUTINE CODE WAS GENERATED.   TO USE THE 8089 BREAK-
                                 13 ;   POINT PROGRAM, THIS ASM89 PROGRAM WOULD NOT BE
                                 14 ;   NEEDED.   SAVE_ADDR IS THE SAME AS SAVE$ADDR IN THE
                                 15 ;   BREAK.89 PROGRAM.
                                 16
  2000                           17 SAVE_ADDR        EQU      2000H    ;SAVE ROUTINE ADDRESS
                                 18
0000   9108 00200000             19          LPDI TP,SAVE_ADDR         ;JUMP TO SAVE ROUTINE
                                 20
                                 21 ;*************************************
                                 22
                                 23 ;   REGISTER SAVE LOCATIONS WITHIN PB:
                                 24
                                 25 REGS     STRUC
0000                             26 PBLOCK:  DS       239      ;PARAMETER BLOCK
00EF                             27 GASAV:   DS       3        ;GA AREA
00F2                             28 GBSAV:   DS       3        ;GB AREA
00F5                             29 GCSAV:   DS       3        ;GC AREA
00F8                             30 BCSAV:   DS       2        ;BC AREA
00FA                             31 IXSAV:   DS       2        ;IX AREA
00FC                             32 CCSAV:   DS       2        ;CC AREA
00FE                             33 MCSAV:   DS       2        ;MC AREA
0100                             34 REGS     ENDS
                                 35
                                 36 ;   REGISTER SAVE ROUTINE:
                                 37
                                 38 ORG      SAVE_ADDR
                                 39
2000   039B EF                   40          MOVP [PP].GASAV,GA        ;SAVE GA
2003   239B F2                   41          MOVP [PP].GBSAV,GB        ;SAVE GB
2006   439B F5                   42          MOVP [PP].GCSAV,GC        ;SAVE GC
2009   6387 F8                   43          MOV [PP].BCSAV,BC         ;SAVE BC
200C   A387 FA                   44          MOV [PP].IXSAV,IX         ;SAVE IX
200F   C387 FC                   45          MOV [PP].CCSAV,CC         ;SAVE CC
2012   E387 FE                   46          MOV [PP].MCSAV,MC         ;SAVE MC
                                 47
2015   2048                      48          HLT                       ;STOP THIS CHANNEL,
                                 49                                    ;CLEAR BUSY FLAG.
                                 50 ;*************************************
2017                             51 BRKPNT   ENDS
                                 52
                                 53 END
```

intel

APPLICATION NOTE

AP-51

March 1979

Designing 8086, 8088, 8089 Multiprocessing Systems with the 8289 Bus Arbiter

Designing 8086, 8088, 8089 Multiprocessor Systems with the 8289 Bus Arbiter

Contents

INTRODUCTION

BUS ARBITER OPERATING CHARACTERISTICS

MULTI-MASTER SYSTEM BUS SURRENDER AND REQUEST

8289 BUS ARBITER INTERFACING TO THE 8288 BUS CONTROLLER

8289 BUS ARBITER INTERNAL ARCHITECTURE

8086 FAMILY PROCESSOR TYPES AND SYSTEM CONFIGURATIONS

8289 SINGLE BUS INTERFACE

IOB INTERFACE

RESB INTERFACE

INTERFACE TO TWO MULTI-MASTER BUSES

WHEN TO USE THE DIFFERENT MODES
- Single Bus Multi-master Interface
- IOB Mode
- Resident Bus Mode

CONCLUSION

Our thanks to Jim Nadir, the author of this application note. Jim is a design engineer in the microprocessors and peripherals operation division. Please direct any technical questions you may have to your local Intel FAE (Field Application Engineer).

INTRODUCTION

Over the past several years, microprocessors have been increasing in popularity. The performance improvements and cost reductions afforded by LSI technology have spurred on the design motivation of using multiple processors to meet system real-time performance requirements. The desire for improved system real-time response, system reliability and modularity has made multiprocessing techniques an increasingly attractive alternative to the system design engineer; techniques that are characterized as having more than one microprocessor share common resources, such as memory and I/O, over a common multiple processor bus.

This type of design concept allows the system designer to partition overall system functions into tasks that each of several processors can handle individually to increase system performance and throughput. But, how should a designer proceed to implement a multiprocessing system? Should he design his own? If so, how are the microprocessors synchronized to avoid contention problems? The designer could put them all in phase using one clock for all the microprocessors. This may work, until the physical dimensions of the system become large. When this occurs, the designer is faced with many problems, like clock skew (resulting in bus spec violations) and duty cycle variations.

A better approach to implementing a multiprocessor system is not to have a common processor clock, but allow each processor to work asynchronously with respect to each other. The microprocessor requests to use the multiple processor bus could then be synchronized to a high frequency external clock which will permit duty cycle and phase shift variations. This type of approach has the benefit of allowing modularity of hardware. When new system functions are desired, more processing power can be added without impacting existing processor task partitioning.

One approach to implement this asynchronous processing structure would be to have all the bus requests enter a priority encoder which samples its inputs as a function of the higher frequency "bus clock". The inputs would arrive asynchronously to the priority encoder and would be resolved by the priority encoder structure as to which microprocessor would be granted the bus. Another approach, that used by Intel, is rather than allowing the requests to arrive asynchronously with respect to one another at the priority encoder, the bus requests are synchronized first to an external high frequency bus clock and then sent to the priority encoder to be resolved. In this way, the resolving circuitry common to all microprocessors is kept at a minimum. Overall system reliability is improved in the sense that should a circuit which serves to synchronize the processor's request (which is now located on the same card as the microprocessor itself) fail, it is only necessary to remove that card from the system and the rest of the system will continue to function. Whereas in the other approach, should the synchronizing mechanism fail, the whole system goes down, as the synchronizing mechanism is located at the shared resource. In addition to the improved system reliability, moving the synchronization mechanism to the processor permits processor control over that mechanism, thereby permitting system flexibility (as will be shown) which could not be reasonably obtained by any other approach.

This synchronizing or arbitrating function was integrated into the 8289, a custom arbitration unit for the 8086, 8088, and 8089 processors. This note basically describes the 8289 arbitration unit, illustrates its different modes of operation and hardware connect in a multiprocessor system. Related and useful documents are: 8086 user's manual, 8289 data sheet, Article Reprint -55: Design Motivations for Multiple Processor Microcomputer Systems (which discusses implementing a semaphore with the MULTIBUS™) and Application Note 28A, Intel MULTIBUS™ interfacing.

BUS ARBITER OPERATING CHARACTERISTICS

The 8289 Bus Arbiter operates in conjunction with the 8288 Bus Controller to interface an 8086, 8088, or 8089 processor to a multi-master system bus (the 8289 is used as a general bus arbitration unit). The processor is unaware of the arbiter's existence and issues commands as though it has exclusive use of the system bus. If the processor does not have the use of the multi-master system bus, the bus arbiter prevents the bus controller, the data transceivers and the address latches from accessing the system bus (i.e., all bus driver outputs are forced into the high impedance state). Since the command was not issued, a transfer acknowledge (XACK) will not be returned and the processor will enter into wait states. Transfer acknowledges are signals returned from the addressed resource to indicate to the processor that the transfer is complete. This signal is typically used to control the ready inputs of the clock generator. The processor will remain in wait until the bus arbiter acquires the use of the multi-master system bus, whereupon the bus arbiter will allow the bus controller, the data transceivers and the address latches to access the system bus. Once the command has been issued and a data transfer has taken place, a transfer acknowledge (XACK) is returned to the processor. The processor then completes its transfer cycle. Thus, the arbiter serves to multiplex a processor (or bus master) onto a multi-master system bus and avoid contention problems between bus masters.

Since there can be many bus masters on a multi-master system bus, some means of resolving priority between bus masters simultaneously requesting the bus must be provided. The 8289 Bus Arbiter provides for several resolving techniques. All the techniques are based on a priority concept that at a given time one bus master will have priority above all the rest. These techniques include the parallel priority resolving techniques, serial priority resolving and rotating priority techniques.

A parallel priority resolving technique has a separate bus request ($\overline{BREQ}$) line for each arbiter on the multi-master bus (see Figure 1). Each $\overline{BREQ}$ line enters into a priority encoder which generates the binary address of the highest priority $\overline{BREQ}$ line which is active at the inputs. The output binary address is decoded by a decoder to select the corresponding $\overline{BPRN}$ (bus priority in) line to be returned to the highest priority requesting arbiter. The arbiter receiving priority ($\overline{BPRN}$ active low) then allows its associated bus master onto the multi-master system bus as soon as it becomes available (i.e., it is no longer busy). When one bus arbiter gains priority over another arbiter, it cannot immediately seize the bus, it must wait until the present bus occupant completes its transfer cycle. Upon completing its transfer cycle, the present bus occupant recognizes that it no longer has priority and surrenders the bus, releasing $\overline{BUSY}$. $\overline{BUSY}$ is an active low OR-tied signal line which goes to every bus arbiter on the system bus. When $\overline{BUSY}$ goes high, the arbiter which presently has bus priority ($\overline{BPRN}$ active low) then seizes the bus and pulls $\overline{BUSY}$ low to keep other arbiters off the bus. (See waveform timing diagram, Figure 2.) Note that all multi-master system bus transactions are synchronized to the bus clock ($\overline{BCLK}$). This allows for the parallel priority resolving circuitry or, any other priority resolving scheme employed, time to settle and make a correct decision.

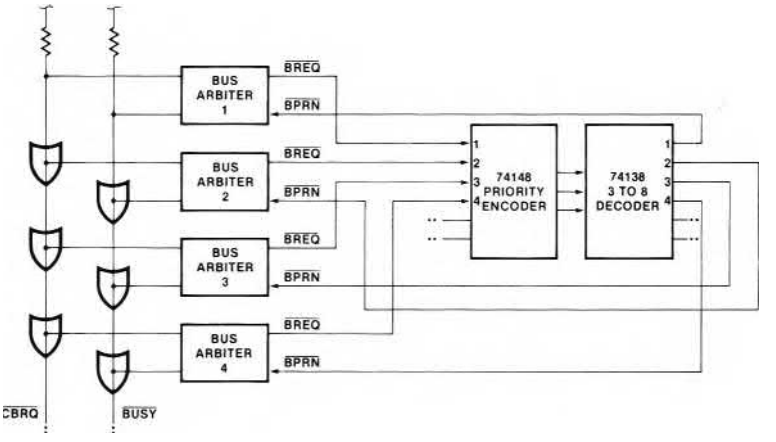
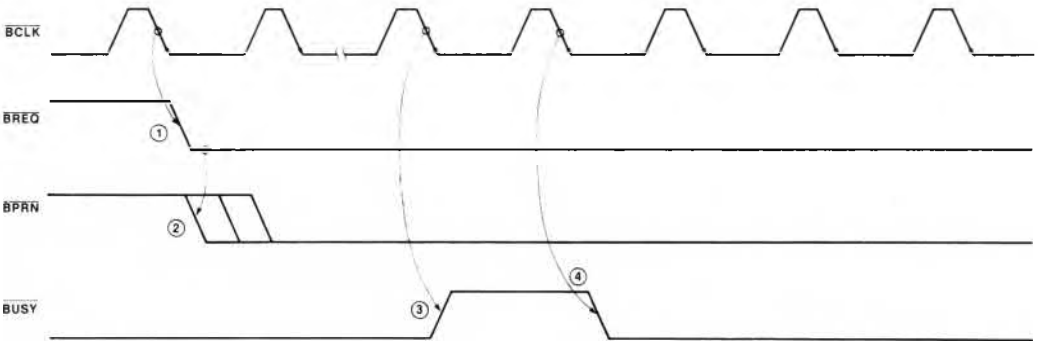


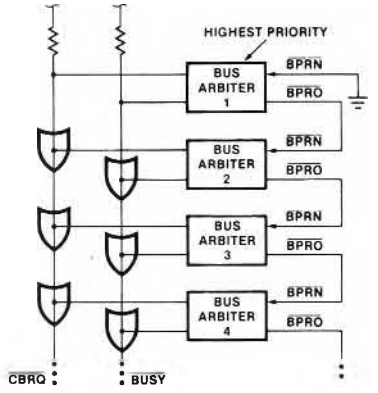
Figure 1. Parallel Priority Resolving Technique



① HIGHER PRIORITY BUS ARBITER REQUESTS THE MULTI-MASTER SYSTEM BUS.
② ATTAINS PRIORITY.
③ LOWER PRIORITY BUS ARBITER RELEASES BUSY.
④ HIGHER PRIORITY BUS ARBITER THEN ACQUIRES THE BUS AND PULLS BUSY DOWN.

Figure 2. Higher Priority Arbiter Obtaining The Bus From A Lower Priority Arbiter

A serial priority resolving technique eliminates the need for the priority encoder-decoder arrangement by daisy-chaining the bus arbiters together. This is accomplished by connecting the higher priority bus arbiter's $\overline{BPRO}$ (bus priority out) output to the $\overline{BPRN}$ of the next lower priority (see Figure 3). The highest priority bus arbiter would have its $\overline{BPRN}$ line grounded, signifying to the arbiter that it always has highest priority when requesting the bus.



THE NUMBER OF ARBITERS THAT MAY BE DAISY-CHAINED TOGETHER IN THE SERIAL PRIORITY RESOLVING TECHNIQUE IS A FUNCTION OF BCLK AND THE PROPAGATION DELAY FROM ARBITER TO ARBITER. NORMALLY, AT 10 MHz ONLY 3 ARBITERS MAY BE DAISY-CHAINED. SEE TEXT.

Figure 3. Serial Priority Resolving

A rotating priority resolving technique arrangement is similar to that of the parallel priority resolving technique except that priority is dynamically reassigned. The priority encoder is replaced by a more complex circuit which rotates priority between requesting arbiters, thus guaranteeing each arbiter equal time on the multi-master system bus.

There are advantages and disadvantages for each of the techniques described above. The rotating priority resolving technique requires an extensive amount of logic to implement, while the serial technique can accommodate only a limited number of bus arbiters before the daisy-chain propagation delay exceeds the multi-master system bus clock ($\overline{BCLK}$). The parallel priority resolving technique is, in general, the best compromise. It allows for many arbiters to be present on the bus while not requiring much logic to implement.

Whatever resolving technique is chosen, it is the highest priority bus arbiter requesting use of the multi-master system bus which obtains the bus. Exceptions do exist with the 8289 Bus Arbiter where a lower priority arbiter may take away the bus from a higher priority arbiter without the need for any additional external logic. This is accomplished through the use of the $\overline{CBRQ}$ pin, discussed in a later section.

MULTI-MASTER SYSTEM BUS SURRENDER AND REQUEST

The 8289 Bus Arbiter provides an intelligent interface to allow a processor or bus master of the 8086 family to access a multi-master system bus. The arbiter directs the processor onto the bus and allows both higher and lower priority bus masters to acquire the bus. Higher priority masters obtain the bus when the present bus master utilizing the bus completes its transfer cycle (including hold time). Lower priority bus masters obtain the bus when a higher priority bus master is not accessing the system bus and a lower priority arbiter has pulled $\overline{CBRQ}$ low. This signifies to the arbiter presently holding the multi-processor bus that a lower priority arbiter would like to acquire the bus when it is not being used. A strapping option (ANYRQST) allows the multi-master system bus to be surrendered to any bus master requesting the bus, regardless of its priority. If there are no other bus masters requesting the bus, the arbiter maintains the bus as long as its associated bus master has not entered the HALT state. *The 8289 Bus Arbiter will not voluntarily surrender the system bus and has to be forced off by another bus master.* An exception to this can be obtained by strapping $\overline{CBRQ}$ low and ANYRQST high. In this configuration the 8289 will release the bus after each transfer cycle.

How the 8289 Bus Arbiter is configured determines the manner in which the arbiter requests and surrenders the system bus. If the arbiter is configured to operate with a processor which has access to both a multi-master system bus and a resident bus, the arbiter requests the use of the multi-master system bus only for system bus accesses (i.e., it is a function of the SYSB/$\overline{RESB}$ input pin). While the processor is accessing the resident bus, the arbiter permits a lower priority bus master to seize the system bus via $\overline{CBRQ}$, since it is not being used. A processor configuration with both an I/O peripheral bus and a system bus behaves similarly. If the processor is accessing the peripheral bus, the arbiter permits the surrendering of the multi-master system bus to a lower priority bus master. To request the use of the multi-master system bus, the processor must perform a system memory access (as opposed to an I/O access).

The arbiter decodes the processor status lines to determine what type of access is being performed and behaves correspondingly. For simpler system configurations, such as a processor which accesses only a multi-master system bus, the arbiter requests the use of the system bus when it detects the status lines initiating a transfer cycle. The decoding of these status lines can be referenced in the 8086, 8088 (non-I/O processor) data sheets or the 8089 (I/O processor) data sheet.

There is one condition common to all system configurations where the multi-master system bus is surrendered to a lower priority bus master requesting the bus by pulling $\overline{CBRQ}$ low. This is the idle or inactive state (TI) which is unique to the 8086 and 8088 processor family. This TI state comes about due to the processor's ability to fetch instructions in advance and store them internally for quick access. The size of the internal queue was optimized so that the processor would make the most ef-

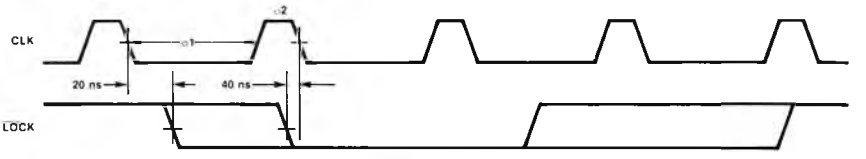
fective use of its resources and be slightly execution bound. Since the processor can fetch code faster than it can execute it, it will fill to capacity its internal storage queue. When this occurs, the processor will enter into idle or inactive states (TI) until the processor has executed some of the code in the storage queue. Once this occurs, the processor will exit the TI state and again start code fetching. Between entering into and exiting from the TI state an indeterminate number of TI states can occur during which the bus arbiter permits the surrendering of the multi-master system bus to a lower priority bus master. As noted earlier and worth repeating here, once the 8289 Bus Arbiter acquires the use of the multi-master system it will not voluntarily surrender the bus and has to be forced off by another bus master. This will be discussed in more detail later.

Two other signals, $\overline{\text{LOCK}}$ and $\overline{\text{CRQLCK}}$ (Figure 4), lend to the flexibility of the 8289 Bus Arbiter within system configurations. $\overline{\text{LOCK}}$ is a signal generated by the processor to prevent the bus arbiter from surrendering the multi-master system bus to any other bus master, either higher or lower priority. $\overline{\text{CRQLCK}}$ (common request lock) serves to prevent the bus arbiter from surrendering the bus to a lower priority bus master when conditions warrant it. $\overline{\text{LOCK}}$ is used for implementing software semaphores for critical code sections and real time critical events (such as refreshing or hard disk transfers).

8289 BUS ARBITER INTERFACING TO THE 8288 BUS CONTROLLER

Once the 8289 Bus Arbiter determines to either allow its associated processor onto the multi-master system bus or to surrender the bus, it must guarantee that command setup and hold times are not violated. This is a two part problem. One, guaranteeing hold time and two, guaranteeing setup time. The 8288 Bus Controller performs the actual task of establishing setup time, while the 8289 Bus Arbiter establishes hold time (see Figure 5).

The 8289 Bus Arbiter communicates with the 8288 Bus Controller via the $\overline{\text{AEN}}$ line. When the arbiter allows its associated processor access to the multi-master system bus, it activates $\overline{\text{AEN}}$. $\overline{\text{AEN}}$ immediately enables the address latches and data transceivers. The bus controller responds to $\overline{\text{AEN}}$ by bringing its command output buffers out of high impedance state but keeping all commands disqualified until command setup time is established. Once established, the appropriate command is then issued. $\overline{\text{AEN}}$ is brought to the false state after the command hold time has been established by the arbiter when surrendering the bus.

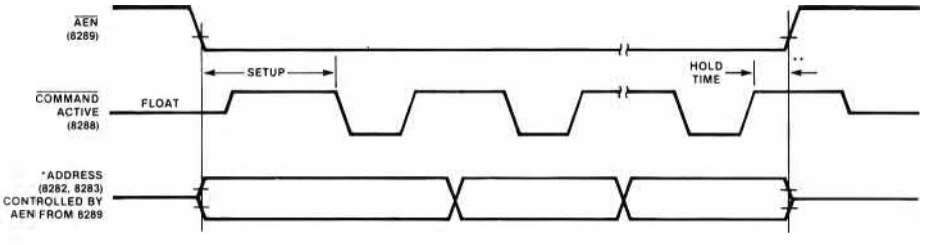


LOCK TIMING

THE ONLY CRITICAL LOCK TIMING IS THAT SHOWN ABOVE. $\overline{\text{LOCK}}$ MUST BE ACTIVATED NO SOONER THAN 20 ns INTO φ1 AND NO LATER THAN 40 ns PRIOR TO THE END OF φ2. LOCK INACTIVE HAS NO CRITICAL TIMING AND CAN BE ASYNCHRONOUS.

$\overline{\text{CRQLCK}}$ HAS NO CRITICAL TIMING AND IS CONSIDERED AS AN ASYNCHRONOUS INPUT SIGNAL.

Figure 4. Lock Timing



*ADDRESSES ARE ACTIVATED IMMEDIATELY WHILE COMMAND IS DELAY TO ESTABLISH SETUP TIME REQUIREMENTS.

**THE 8289 ARBITER INTERNALLY TRACKS THE PROCESSOR CYCLE TO ESTABLISH THE PROPER AMOUNT OF HOLD TIME AFTER THE COMMAND HAS GONE INACTIVE.

Figure 5. Single Bus Interface Timing

8289 BUS ARBITER INTERNAL ARCHITECTURE

A block diagram of the internal architecture of the 8289 Bus Arbiter is shown in Figure 6. It is useful to understand this block diagram when discussing the different modes of the 8289 and their impact on processor bus operations; however, you may want to skip this section to "8086 family processor types and system configurations" and return to it afterwards, as this section addresses the very involved reader. The front end state generator (FETG) and the back end state generator (BETG) allow the arbiter to track the processor cycle. An examination of an 8086 family processor state timings show that all command and control signals are issued in states T1 and T2 while being terminated in states T3 and T4, with an indeterminate number of wait states (Tw) occurring in between. Note further, that an indeterminate number of idle or inactive states can occur immediately proceeding and following a given transfer cycle. Since an indeterminate number of wait states can occur, two state generators are required; one to generate control signals (the FETG) and one to terminate control signals (the BETG). The FETG is triggered into operation when the processor activates the status lines. The FETG is reset and the BETG is triggered into operation by the status lines going to the passive condition. The BETG is reset when the status lines again go active.

It is necessary for the 8289 Bus Arbiter to track the processor in order that it is properly able to determine where and when to request or surrender the use of the multi-master system bus. In system configurations which access a resident bus, the use of the multi-master system bus is requested later in order to allow time for the SYSB/$\overline{\text{RESB}}$ input to become valid. For systems which access a peripheral bus, the arbiter issues a request for the system bus only for memory transfer cycles which it decodes from the status lines (and time must be allowed for the status lines to become valid and then decoded). In a system which accesses only a multi-master system bus, a request is made as soon as the arbiter detects an active-going transition on the processor's status lines. Thus, when the processor initiates a transfer cycle, the FETG is triggered into operation and, depending upon what mode the arbiter is configured in, the STATUS & MODE DECODE circuitry initiates a request for the system bus at the appropriate time. The request enters the BREQ SET circuitry where it is then synchronized to the multi-master system bus clock ($\overline{\text{BCLK}}$) by the PROCESSOR SYNCHRONIZATION circuitry.* Once synchronized, the multi-master system bus interface circuitry issues a $\overline{\text{BREQ}}$. When the priority resolving circuitry returns a $\overline{\text{BPRN}}$ (bus priority in), the PROCESSOR SYNCHRONIZATION circuitry seizes the bus the next time it becomes available (i.e., $\overline{\text{BUSY}}$ goes high) by pulling $\overline{\text{BUSY}}$ low one $\overline{\text{BCLK}}$ after it goes high and enables $\overline{\text{AEN}}$. (See waveform timing diagram in Figure 2). Once the arbiter acquires the use of the system bus and a data exchange has taken place (a transfer acknowledge, XACK, was returned to the processor), the processor status lines go passive and the

*Due to the asynchronous nature of processor trasnsfer request to the multi-master system bus clock, it is necessary to synchronize the processor's transfer request to BCLK.

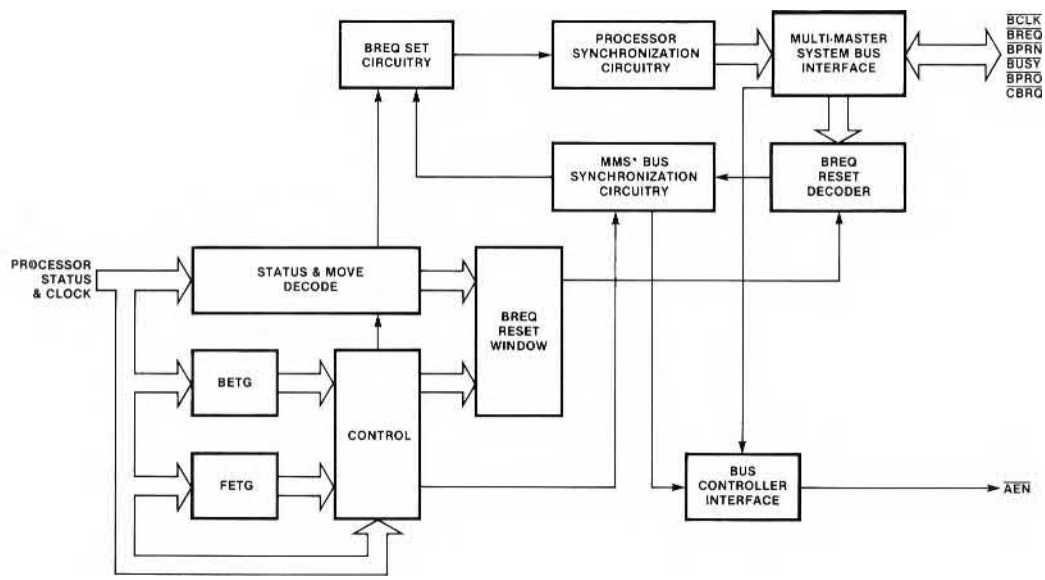


Figure 6. 8289 Bus Arbiter Block Diagram

BETG is triggered into operation. The BETG provides the timing for the bus surrender circuitries in the event that conditions warrant the surrender of the multi-master bus, i.e., the bus arbiter lost priority to a higher bus master or the processor has entered into TI states and $\overline{CBRQ}$ is pulled low, etc. If such is the case, the BREQ RESET DECODER initiates a bus surrender request. The bus surrender request is synchronized by the MMS BUS SYNCHRONIZATION CIRCUITRY to the processor clock. The MMS BUS SYNCHRONIZATION CIRCUITRY instructs the bus controller interface circuitry to make $\overline{AEN}$ go false and resets the BREQ SET circuitry. Resetting the BREQ SET circuitry will cause its output to go false and be synchronized by the processor synchronization, eventually instructing the MULTI-MASTER SYSTEM BUS INTERFACE circuitry to reset $\overline{BREQ}$. In the event that a lower priority arbiter has caused the arbiter to surrender the bus, it is necessary that $\overline{BREQ}$ be reset. Resetting $\overline{BREQ}$ allows the priority resolving circuitry to generate $\overline{BPRN}$ to the next highest priority bus master requesting the bus. The BREQ RESET WINDOW circuitry provides a 'window' wherein the arbiter allows the multi-master system bus to be surrendered and serves as part of the MMS bus-processor synchronization circuitry.

8086 FAMILY PROCESSOR TYPES AND SYSTEM CONFIGURATIONS

There are two types of processors in the 8086 family — an I/O processor (the 8089 IOP) and a non-I/O processor (the 8086 and 8088 CPUs). Consequently, there are two basic operating modes in the 8289 Bus Arbiter. One, the IOB (I/O peripheral bus) mode, permits the processor access to both an I/O peripheral bus and a multi-master system bus. The second, the RESB (resident bus) mode, permits the processor to communicate over both a resident bus and a multi-master system bus. Even though it is intended for the arbiter to be configured in the IOB mode when interfacing to an I/O processor and for it to be in the RESB mode when interfacing to a non-I/O processor, it is quite possible for the reverse to be true. That is, it is possible for a non-I/O processor to have access to an I/O peripheral bus or for an I/O processor to have access to a resident bus as well as access to a multi-master system bus. The IOB strapping option configures the 8289 Bus Arbiter into the IOB mode and RESB strapping option configures it into the resident bus mode. If both strapping options are strapped false, a third mode of operation is created, the single bus mode, in which the arbiter interfaces the processor to a multi-master system bus only. With both options strapped true, the arbiter interfaces the processor to a multi-master system bus, a resident bus and an I/O bus.

To better understand the 8289 Bus Arbiter, each of the operating modes, along with their respective timings, are examined by means of examples. The simplest configuration, the Single Bus Configuration, (both IOB and RESB strapped inactive) will be considered first, followed by the I/O bus Configuration and the Resident Bus Configuration. Finally, brief mention is made of a configuration that allows the processor to interface to two multi-master system buses. This particular configuration is briefly mentioned because, as will be seen, it is simply an extension of the resident bus configuration. When discussing the Single Bus Configuration, processor/arbiter, arbiter/system bus and internal arbiter, considerations are made resulting in a table that illustrates overhead in requesting the system bus. As this applies to the other 8289 configurations, only additional considerations will be given. A summary of when to use the different configurations is given at the end.

8289 SINGLE BUS INTERFACE

Figure 7 shows a block diagram of a bus master which has to interface only to a system bus — preferably the MULTIBUS — where there exists more than one bus master. In later configurations, it will be shown how the processor can be made to interface with more than one bus. Since the processor has only to interface with one bus, this configuration is called "single".

Connecting the 8289 Bus Arbiter to the processor is as simple as it was to connect the 8288 Bus Controller. Namely, the three status lines, $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ are directly connected from the processor to the arbiter. The clock line from the 8284 Clock Generator is brought down and connected. (Note that both the 8288 Bus Controller and the 8289 Bus Arbiter are connected to the same clock, CLK and not the peripheral clock, PCLK as the 8086 processor.) From the arbiter, $\overline{AEN}$ is connected to the bus controller and to the clock generator. The $\overline{IOB}$ pin on the arbiter is strapped high and on the controller the IOB pin is strapped low. In addition, the RESB pin on the arbiter is strapped low, finishing the processor interface.

Some flexibility exists with the MULTIBUS or multi-master system bus interface. The system designer must first decide upon the type of priority resolving scheme to be employed, whether it is to be the serial, parallel, or rotating priority scheme. A rotating priority scheme would be employed where the system designer would want to guarantee that every bus master on the bus would be given time on the bus. In the serial and parallel schemes, the possibility exists that the lowest assigned priority bus master may not acquire the bus for long periods of time. This occurs because priority is permanently assigned and if bus demand is high by the higher assigned priorities, then the lower priorities must wait. In most cases, this situation is acceptable because the highest priority is assigned to the bus master that cannot wait. Highest priority is usually assigned to DMA type devices where service requirements occur in real time. CPUs are assigned the lower priorities. For the purpose of this discussion, the parallel priority scheme will be used with brief reference to the serial priority scheme.

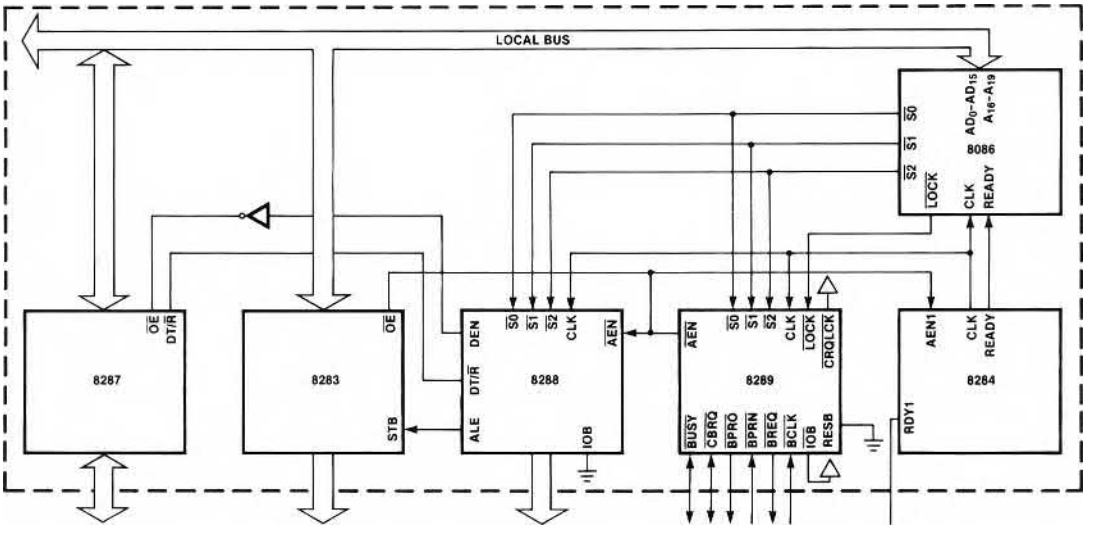


Figure 7. Single Multimaster Bus Interface

Figure 8 shows how a typical multi-processing system might be configured with the 8289 in the Single Bus mode. In the system there are three bus masters, each having the assigned priority as indicated—priority 1 being the highest and priority 3 being the lowest. Priority is established using the parallel priority scheme (ignore the dotted signal interconnect for the moment). Each bus arbiter monitors its associated processor and issues a bus request ($\overline{BREQ}$) whenever its processor wants the bus. A common clocking signal ($\overline{BCLK}$) runs to each of the arbiters in the system. It is from the falling edge of this clock that all bus requests are issued. Since all bus requests are made on the same clock edge, a valid priority can be established by the priority resolving circuitry by the next falling $\overline{BCLK}$ edge. Note that all multi-master system bus (MULTIBUS) input signals are considered to be valid at the falling edge of $\overline{BCLK}$. And that all multi-master system bus output signals are issued from the falling edge of $\overline{BCLK}$. With the parallel resolving module, arbiters 2 and 3 would issue their respective $\overline{BREQ}$s (Figure 9) on the falling edge of $\overline{BCLK}$ 1, as shown. The outputs ($\overline{BPRN}$ 1, $\overline{BPRN}$ 2, and $\overline{BPRN}$ 3) of the priority encoder-decoder arrangement change to reflect their new input conditions and need to be valid early enough in front of $\overline{BCLK}$ 2 to guarantee the arbiter's setup time requirements. Since arbiter 2 at the time is the highest priority arbiter requesting the bus, bus priority is given to arbiter 2 ($\overline{BPRN}$ 2 goes low), and since the bus was not busy ($\overline{BUSY}$ is high) at the time priority was granted to arbiter 2, arbiter 2 pulls $\overline{BUSY}$ inactive on $\overline{BCLK}$ 2, thereby seizing the bus and excluding all other arbiters access to the bus. Once the bus is seized, arbiter 2 activates its $\overline{AEN}$. $\overline{AEN}$ going low directly enables the 8283 address latches and wakes up the 8288 Bus Controller. The bus controller enables the 8287 transceivers, waits until the address to command setup time has been established, and then enables its command drivers onto the bus.

If the serial priority resolving mode was used instead, much of the events that happened for the parallel priority resolving mode would be the same except, of course, there would be no parallel priority resolving module. Instead, the system would be connected as indicated in Figure 8 by the dotted signal lines connecting the $\overline{BPRO}$ of one arbiter to $\overline{BPRN}$ of the next lower priority arbiter.

The $\overline{BREQ}$ lines would be disconnected and the priority encoder-decoder arrangement removed. This arrangement is simpler than the parallel priority arrangement except that the daisy-chain propagation delay of the highest priority bus arbiter's $\overline{BPRO}$ to the lowest priority bus arbiter's $\overline{BPRN}$, including setup time requirement ($\overline{BPRN}$ to $\overline{BLCK}$), cannot exceed the $\overline{BCLK}$ period. In short, this means there are only so many arbiters that can be daisy-chained for a given $\overline{BCLK}$ frequency. Of course, the lower the $\overline{BCLK}$ frequency, the more arbiters can be daisy-chained. The maximum $\overline{BCLK}$ frequency is specified at 10 MHz, which would allow for three 8289 arbiters to be daisy-chained. In general, the number of arbiters that can be connected in the serial daisy-chain configuration can be determined from the following equation:

$$\overline{BCLK} \text{ period} \geq TBLPOH + TPNPO\,(N-1) + TPNBL$$

where N = # of arbiters in system

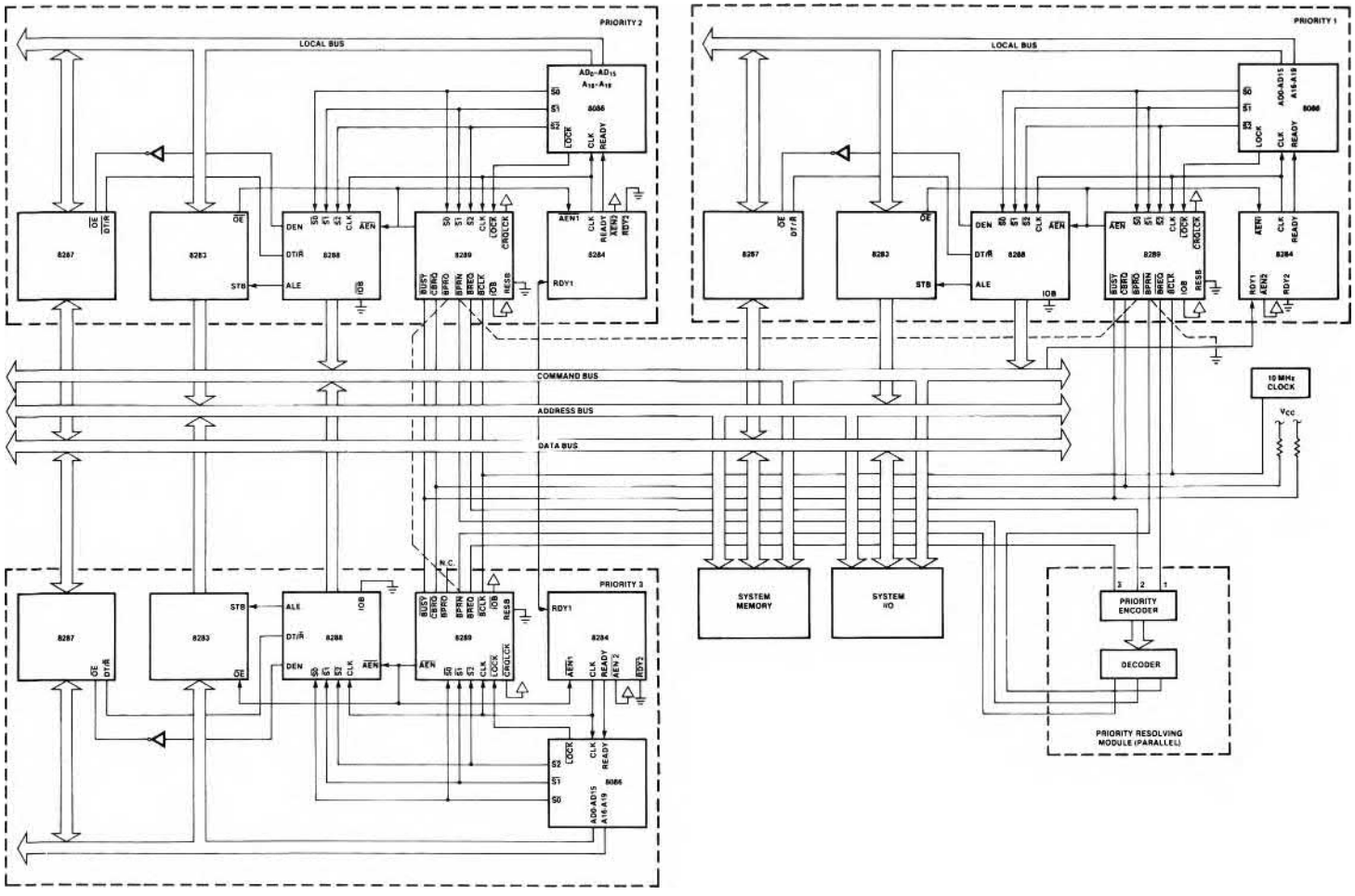
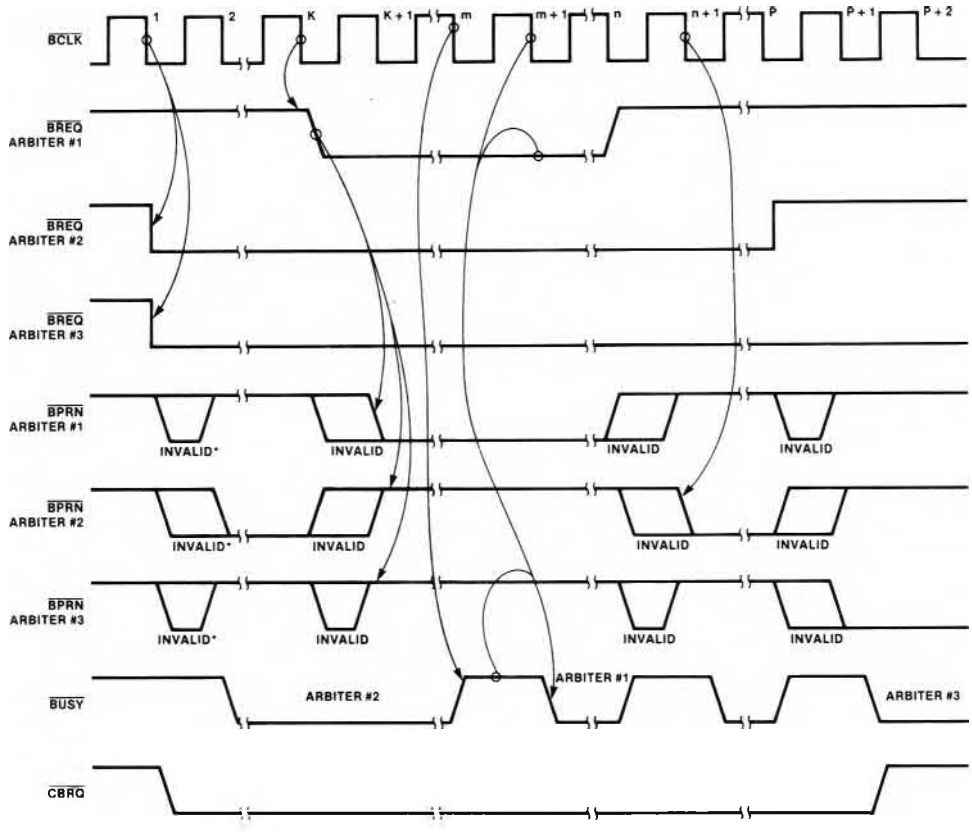


Figure 8. Multiprocessing System With 8289 In Single Bus Mode



*DECODING GLITCHES ARE PERMITTED

Figure 9. Example Timing For Figure 8

Returning to Figure 9, it can be seen that K $\overline{\text{BCLK}}$s later, arbiter 1 has decided to request the bus and its $\overline{\text{BREQ}}$, $\overline{\text{BREQ 1}}$, has gone low. Since arbiter 1 is of higher priority than arbiter 2, which presently has the bus, bus priority is reassigned by the priority module (or the daisy-chain approach in the serial priority) to arbiter 1. $\overline{\text{BPRN 1}}$ goes low and $\overline{\text{BPRN 2}}$ now goes high ($\overline{\text{BPRN 3}}$ remains high, even though decoding can cause it to glitch momentarily). The loss of priority instructs arbiter 2 that a higher priority arbiter wants the bus and that it is to release the bus as soon as its present transfer cycle is done. Since arbiter 2 cannot immediately release the bus, arbiter 1 must wait. In the particular case illustrated in Figure 9, arbiter 2 releases the bus (allows $\overline{\text{BUSY}}$ to go high) on clock edge M, and on clock edge M + 1, arbiter 1 now seizes the bus, pulling $\overline{\text{BUSY}}$ low. Arbiter 1 is the highest priority arbiter in the system and it now has the bus. Arbiters 2 and 3 still want the bus (their $\overline{\text{BREQ}}$s are both low).

How quickly arbiter 1 can acquire the bus is dependent upon the configuration and strapping options of the arbiter it is trying to acquire it from. For example, if the $\overline{\text{LOCK}}$ input to arbiter 2 was active (low) at the time, then arbiter 1, even though it was of higher priority, would not have acquired the bus until after $\overline{\text{LOCK}}$ was released (goes high). Effectively, $\overline{\text{LOCK}}$ locks the arbiter onto the bus once the bus has been acquired. $\overline{\text{LOCK}}$ will not force another arbiter to release the bus any sooner, it just prevents the bus from being given away no matter what the priority of the other arbiter. Another factor to be considered is where in the transfer cycle is the processor when the arbiter is instructed to give up the bus. Obviously, if the cycle had just started, it will take longer for the bus to be released than if the cycle was just ending. Another factor to be included in this consideration is the phase relationship of the processor's clock (CLK) to the bus clock ($\overline{\text{BCLK}}$). This relationship is examined in more detail later on. Table 1 lists the time

requirements for various arbiter actions such as bus acquisition and bus release (under $\overline{LOCK}$ and other circumstances) taking into account the phase relationships between CLK and $\overline{BCLK}$.

Bus Request ($\overline{BREQ}$↓)	Mode	Delay (Max)	Delay (Min)
Status→$\overline{BREQ}$↓	Single	2 $\overline{BCLK}$s	1 $\overline{BCLK}$
Status→$\overline{BREQ}$↓	IOB	2 $\overline{BCLK}$s + ~1 CLK*	1 $\overline{BCLK}$ + ~½ CLK*
Status→$\overline{BREQ}$↓	RESB	2 $\overline{BCLK}$s + ~2 CLKs†	1 $\overline{BCLK}$ + ~1½ CLKs†
Status→$\overline{BREQ}$↓	IOB•RESB	2 $\overline{BCLK}$s + ~2 CLKs†	1 $\overline{BCLK}$ + 1½ CLKs†

*Request originates off of $\phi 2$ of T1 and $\overline{BREQ}$↓ occurs 1 $\overline{BCLK}$ (min) to 2 $\overline{BCLK}$s (max) thereafter. Depending upon where status occurs with respect to clock determines how long a time exists between status and $\phi 2$ of T1, and is anywhere from ½ CLK (min) to 1 CLK (max).

†Request originates off of T2•$\phi 1$ and $\overline{BREQ}$↓ occurs 1 $\overline{BCLK}$ (min) to 2 $\overline{BCLK}$s (max) thereafter. The same reasoning as used in the IOB mode is valid here.

Bus Release ($\overline{BREQ}$↑)	Mode	Delay (Max)	Delay (Min)
Higher Priority ($\overline{BPRN}$↑)	All	2 CLKs + 2 $\overline{BCLK}$s	1 CLK + 1 $\overline{BCLK}$
Lower Priority ($\overline{CBRQ}$↑)	All	2 CLKs + 2 $\overline{BCLK}$s	1 CLK + 1 $\overline{BCLK}$
Surrender occurs once the proper surrender conditions exist.			

Table 1. Surrender and Request Time Delays

One signal which has been basically ignored to this point is $\overline{CBRQ}$. $\overline{CBRQ}$, like $\overline{BUSY}$, is an open-collector signal from the arbiter which is tied to the $\overline{CBRQ}$ signals of the other arbiters and to a pull-up resistor (see Figure 8). $\overline{CBRQ}$ is both an input and an output. As an output, $\overline{CBRQ}$ serves to instruct the arbiter presently on the bus that another arbiter wishes to acquire the bus. As an input, $\overline{CBRQ}$ serves to instruct the arbiter presently on the bus that another arbiter wants the bus. $\overline{CBRQ}$ is an input or output, dependent on whether the arbiter is on the bus or not (respectively), and is issued as a function of $\overline{BREQ}$. Thus, a lower priority arbiter requesting the bus already controlled by a higher priority arbiter will pull $\overline{CBRQ}$ low, as well as $\overline{BREQ}$. Even a higher priority arbiter will pull $\overline{CBRQ}$ low until it acquires the bus. Note, however, that the higher priority arbiter will acquire the bus through the reassignment of priorities — it being given priority and the other arbiter presently on the bus losing it. In effect, $\overline{CBRQ}$ serves to notify the arbiter that an arbiter of lower priority wants the bus.

If the arbiter presently on the bus is configured to react to $\overline{CBRQ}$ and the proper surrender conditions exist, the bus is released. When releasing the bus, the arbiter also turns off its $\overline{BREQ}$ ($\overline{BREQ}$ goes high) in order to allow priority to be established to the next lower arbiter requesting the bus. Such is the case shown in Figure 9. Whereas it was assumed that the proper surrender conditions did not exist for arbiter 2 when it had the bus, it is assumed that the proper conditions do exist during the time that arbiter 1 has the bus. Arbiter 2 had to give up the bus because an arbiter of higher priority was requesting it. Arbiter 1 surrenders the bus because the proper surrender conditions exist and a lower priority arbiter requested the bus by pulling $\overline{CBRQ}$ low. This is an assumed condition which is not otherwise shown in Figure 9. This is not an unrealistic condition. Normally, a higher priority arbiter will acquire the bus through the reassignment of priorities, while lower priority arbiters acquire the bus through $\overline{CBRQ}$.

Digressing for a moment, the 8289 Bus Arbiter will not voluntarily surrender the bus (except when the processor halts execution). As a result, it has to be forced off the bus. The 8289 Bus Arbiter does not generate a $\overline{BREQ}$ for each cycle. It generates a $\overline{BREQ}$ once and then hangs onto the bus. To do otherwise would require that $\overline{BREQ}$ be dropped (go high) after each transfer cycle so that if it did need to do another transfer cycle, another arbiter would automatically be assigned priority. This approach, however, entails certain overhead. Command to address setup and hold time must be prefixed and appended to each transfer cycle. Each transfer cycle would be characterized by first acquiring the bus, then establishing the setup time requirements, finally performing the transfer cycle, establishing the hold time requirements, and then releasing the bus (see Figure 10). If another transfer cycle was to immediately follow and if the arbiter still had priority, then the whole above procedure would be repeated. The end result would be wasted time as hold times following setup times (see Figure 10A). The approach taken by the 8289 Bus Arbiter of having to be forced off the bus, even when it is not using the bus (i.e., forced off by a lower priority arbiter), provides for greater bus efficiency. A lower priority arbiter having to force off another arbiter that is not using the bus but just hanging on to it, may not seem very efficient. In actuality it is a good trade-off. In many multimaster systems some bus masters occasionally demand the bus, while others demand the bus constantly. The bus master which constantly demands the bus may momentarily need not to access the bus. Why should that arbiter surrender the bus when chances are that the other bus masters which occasionally access the bus don't want it at the time? If it doesn't give up the bus, then it can momentarily cease access to the bus and then continue, without any performance penalty of having to reestablish control of the bus. The greater bus efficiency that it affords is well worth the added complexity (Figure 10B).

Returning to Figure 9, the combination of the proper surrender conditions existing and $\overline{CBRQ}$ being low, forced the higher priority arbiter, arbiter 1, off the bus. Arbiter 2, being of next higher priority and wanting the bus, acquired the bus on clock edge N + 1. If arbiter 1 decides to re-access the bus, it would reacquire the bus through the reassignment of priorities. This is not the case shown in Figure 9. Arbiter 1 has decided that it does not need the bus and does not renew its $\overline{BREQ}$. Arbiter 2, having acquired the bus through $\overline{CBRQ}$, is now the highest priority arbiter requesting the bus. As can be seen it is not the only arbiter requesting the bus. Arbiter 3 is still patiently waiting for the bus and $\overline{CBRQ}$ remains low. The same conditions that forced arbiter 1 off the

bus for arbiter 2 now forces arbiter 2 off the bus for arbiter 3. When the proper surrender conditions exist, arbiter 2 releases its $\overline{BREQ}$ and surrenders the bus to arbiter 3. Arbiter 3 acquires the bus on clock edge P + 1 and releases its $\overline{CBRQ}$. Since no other arbiter wants the bus (i.e., there is no other arbiter holding $\overline{CBRQ}$ low), $\overline{CBRQ}$ goes high (inactive). This would have also been true when arbiter 2 acquired the bus and released its $\overline{CBRQ}$ if arbiter 3 didn't want the bus.

In the Single interface, the arbiter monitors the processor's status lines, which are activated whenever the processor performs a transfer cycle. The arbiter, on detecting the status lines going active, will issue a $\overline{BREQ}$ if the status is not the HALT status. If the processor issues the HALT status, the arbiter will not request the bus, and if it has the bus, will release it.

This effectively concludes how arbiters interact to one another on the bus. Having examined the processor-to-arbiter interface, and arbiter-to-MULTIBUS (arbiter-to-arbiter) interaction, one interface is left, the internal interface of processor-related signals to that of MULTIBUS-related signals.

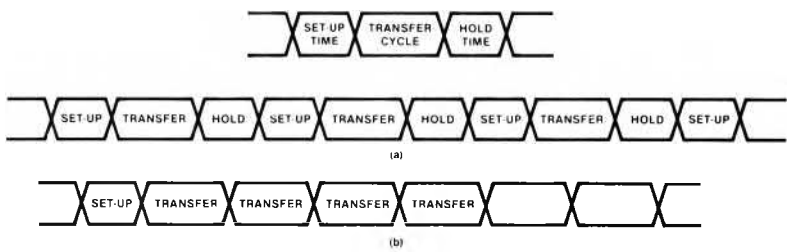
An important point to remember is that the processor has its own clock (CLK) and the multi-master system bus has its own ($\overline{BCLK}$). These two clocks are usually out of phase and of different frequencies. Thus, the arbiter must synchronize events occurring on one interface to events occurring on another interface. As a result of this back and forth synchronization, ambiguity can arise as to when events actually do take place.

Very simply, the 8289 arbiter operation can be represented as two events, requesting and surrendering. Figure 11 is a representation of the timing relationships involved. The request input is a function of the processor's clock and the surrender input is a function of either the bus clock or the processor's clock. To request the bus, the processor activates its status lines which in turn enables the request input. Depending upon the phase relationship between the occurrence of status (request active) and $\overline{BCLK}$, $\overline{BREQ}$ appears one to two $\overline{BCLK}$s later. As shown in Figure 12, the phase relationship between request and $\overline{BCLK}$ is such that the BRQ1 flip-flop may or may not catch request on the first $\overline{BCLK}$.*

If BRQ1 flip-flop does catch the request, then one $\overline{BCLK}$ later, $\overline{BREQ}$ goes low and one $\overline{BCLK}$ after that, $\overline{BUSY}$ goes low (it is assumed that priority is immediately granted and that the bus is available). If BRQ1 flip-flop does not catch the request, then request is caught on the next $\overline{BCLK}$ and $\overline{BREQ}$ goes low one $\overline{BCLK}$ later, followed by $\overline{BUSY}$ which also goes low one $\overline{BCLK}$ later. Note that $\overline{BREQ}$ and $\overline{BUSY}$ track, as $\overline{BREQ}$ is an input term for $\overline{BUSY}$. During bus acquisition, the surrender flip-flop is false (SURNDR Q = low) and $\overline{AEN}$ follows $\overline{BUSY}$.

Once the bus is acquired, the surrender circuitry is enabled so that when a valid surrender condition exists, the bus can be surrendered. The surrender circuitry synchronizes the surrender request to the processor's clock and drives SURNDR low. Like the acquisition circuitry, it takes from one to two processor clocks to generate SURNDR and depends upon the phase relationship between the surrender request and the processor's clock.

*The two bus request flip-flops, BRQ1 and BRQ2, are edge-triggered, high resolution flip-flops and serve to reduce the probability of walkout down to an acceptable level. Walkout occurs because $\overline{BCLK}$ is asynchronous with respect to request. If walkout does occur on BRQ1 flip-flop, the probability is high that the BRQ1 flip-flop will resolve itself prior to BRQ2 flip-flop being triggered. Even if BRQ1 flip-flop did not quite resolve itself, the probability of BRQ2 flip-flop walking out to an unacceptable point in time is itself low.



a) BUS UTILIZATION AS A RESULT OF HAVING TO REQUEST AND RELEASE THE BUS FOR EACH TRANSFER CYCLE. THIS PERMITS LOWER PRIORITY ARBITERS EASY ACCESS TO THE BUS SHOULD THE HIGHER PRIORITY ARBITER NO LONGER NEED THE BUS. HOWEVER, BUS EFFICIENCY IS POOR DUE TO THE ARBITER THRASING ON AND OFF OF THE BUS FOR EACH TRANSFER CYCLE.

b) 8289 BUS UTILIZATION IS MORE EFFICIENT IN THAT THE ARBITER HAS ONLY TO ACQUIRE THE BUS ONCE. THE 8289 HANGS ONTO THE BUS UNTIL FORCED OFF. THIS APPROACH ADDS A LITTLE MORE COMPLEXITY TO THE SYSTEM INASMUCH AS SOME MEANS MUST BE PROVIDED FOR LOWER PRIORITY ARBITERS TO FORCE THE HIGHER PRIORITY ARBITER OFF OF THE BUS WHEN IT IS NOT USING IT. THE ADDED COMPLEXITY IS WELL WORTH THE BUS EFFICIENCY AND SYSTEM FLEXIBILITY IT AFFORDS. THE 8289 ARBITER CAN BE CONFIGURED TO HAVE THE TRANSFER TIMING AS SHOWN IN (a) (IMITATING THE METHOD 8218 AND 8219 USES, BUS ARBITERS FOR 8080 AND 8085 RESPECTIVELY) BY STRAPPING ANYROST HIGH AND CBREQ LOW.

Figure 10. Two Techniques For Doing Multibus Transfer Cycles

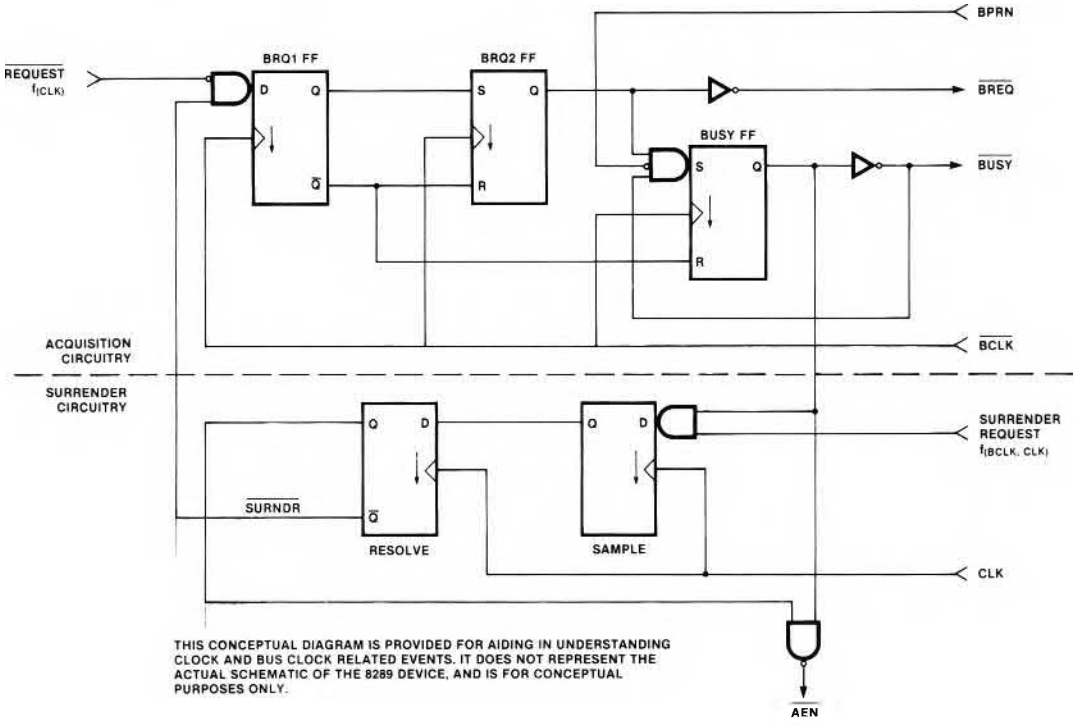
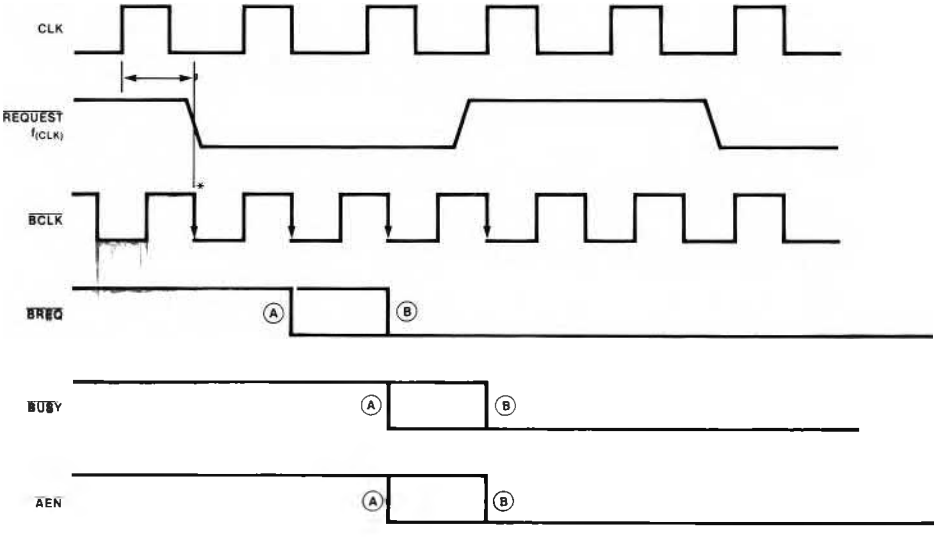


Figure 11. Symbolic Representation of Internal 8289 Timing



* WHEN THE REQUEST OCCURS SIMULTANEOUSLY WITH BCLK, BCLK MAY OR MAY NOT CATCH THE REQUEST. IF IT DOES, THE WAVEFORMS FOLLOW THOSE SHOWN DESIGNATED BY (A) . IF NOT, THE REQUEST IS PICKED UP ON THE NEXT EDGE OF BCLK AND THE WAVEFORMS FOLLOW THOSE SHOWN DESIGNATED BY (B) .

Figure 12. Results Of An Asynchronous Event

Having synchronized the surrender request to the processor's clock to generate SURNDR, SURNDR is then synchronized to $\overline{BCLK}$ to reset the BUSY and BRQ flip-flops. When BUSY-Q goes low, the surrender circuitry is reset which in turn re-enables the request input. The timing in Figure 13 shows the surrender request input going high on the falling edge of the clock. If the Sample flip-flop was able to catch the surrender request on the edge of clock 1, then SURNDR would be generated (go low) on clock edge 2. If not, SURNDR would be generated on clock edge 3. SURNDR going low on clock edge 2 will be, for ease of discussion, referred to as SURNDR a and SURNDR going low on clock edge 3 will be referred to as SURNDR b. As can be seen from Figure 13, SURNDR a just happens to go low on $\overline{BCLK}$ edge 2. Since SURNDR is used to reset the BRQ flip-flops, which are clocked by the falling edge of $\overline{BCLK}$, the BRQ1 flip-flop may or may not catch SURNDR a on $\overline{BCLK}$ edge 2. If it does, then BRQ and $\overline{BUSY}$ go high on BCLK edge 3 which, for convenience, will be called $\overline{BREQ}$ a or $\overline{BUSY}$ a. If not, then $\overline{BREQ}$ and $\overline{BUSY}$ will go high on $\overline{BCLK}$ edge 4, which will be referred to as $\overline{BREQ}$ b or $\overline{BUSY}$ b, respectively. SURNDR b occurs early enough to assure that $\overline{BUSY}$ and $\overline{BREQ}$ are reset on $\overline{BCLK}$ edge 5, which will be referred to as $\overline{BUSY}$ b1 and $\overline{BREQ}$ b1. Depending upon when $\overline{BUSY}$ goes high, determines when the surrender circuitry is reset and how soon the next $\overline{BREQ}$ can be generated. $\overline{BUSY}$ a1 causes SURNDR c to occur where shown and SURNDR c in turn would allow the earliest bus request to occur at $\overline{BREQ}$ c1. At the other extreme, $\overline{BUSY}$ b1 allows the earliest bus request to occur at $\overline{BREQ}$ e1.

Table 1 summarizes the maximum and minimum delays for bus request, once the proper request and surrender conditions exist. Table 2 lists the proper surrender conditions.

Mode	Surrender Conditions
Single	HALT state, loss of BPRN, TI•CBREQ
IOB	HALT state, loss of BPRN, TI•CBREQ, I/O Command•CBRQ
RESB	HALT state, loss of BPRN, TI•CBREQ, (SYSB/RESB = 0)•CBRQ
IOB•RESB	HALT state, loss of BPRN, TI•CBREQ, (SYSB/RESB = 0)•CBREQ, I/O Command•CBRQ

Table 2. Surrender Conditions

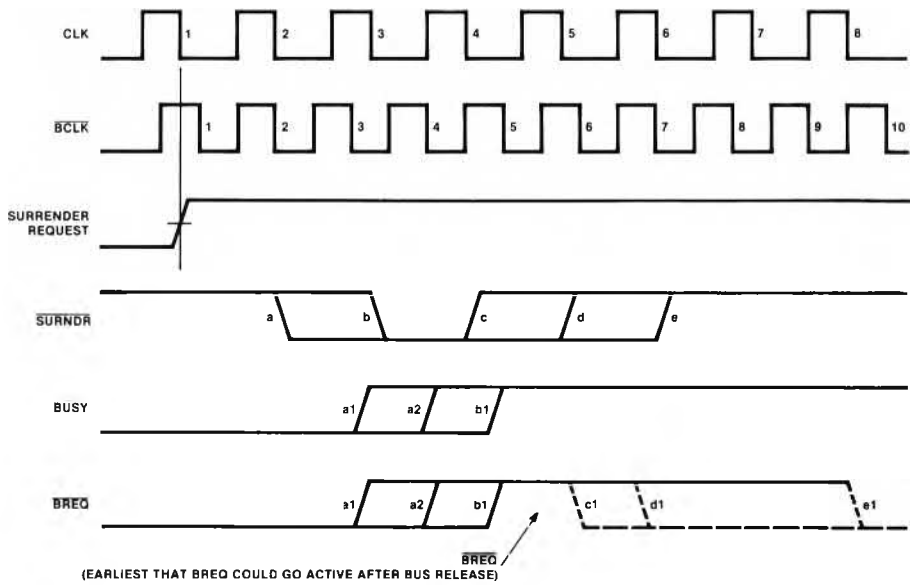


Figure 13. Asynchronous Bus Release

IOB INTERFACE

Now that the processor-arbiter, arbiter-system bus and internal arbiter timings have been discussed, it is appropriate to consider the other interfaces that the 8289 Bus Arbiter provides.

In the IOB mode, the processor communicates and controls a host of peripherals over the peripheral bus. When the I/O processor needs to communicate with system memory, it is done so over the system memory bus. Figure 14 shows a possible I/O processor system configuration, utilizing the 8089 I/O processor in its REMOTE mode. Resident memory exists on the peripheral bus in order that canned I/O routines and buffer storage can be provided. Resident memory is treated as an I/O peripheral. When a peripheral device needs servicing, the I/O processor accesses resident memory for the proper I/O driver routine and services the device, transmitting or storing peripheral data in buffer storage area of resident memory. The resident memory's buffer storage area could then be emptied or replenished from system memory via the system bus. Using the IOB interface allows an I/O processor the capability of executing from local memory (on the peripheral bus) concurrently with the host processor.

Timing in this mode is no different from timing in the SINGLE BUS mode. The only difference lies in the request and surrender conditions. The arbiter extends the single bus mode conditions to qualify when the system bus is requested and adds on additional surrender conditions. The system bus is only requested during system bus commands (the arbiter decodes the processor's status lines) and, in addition to the other surrender terms, the arbiter permits surrender to occur during I/O bus (or local bus) commands, when the I/O processor is using its own local bus.

Like the arbiter, the bus controller must also be informed of the mode it is operating in. In the IOB mode, the 8288 bus controller issues I/O bus commands independently of the state of $\overline{AEN}$ from the arbiter. It is assumed that all I/O bus commands are intended for the I/O bus and hence there is a separate I/O command bus from the controller. All I/O bus commands are sent directly to the I/O bus and are not influenced by $\overline{AEN}$. System bus commands are assumed as going to the system bus. Since system bus commands are directed to the system bus, they must still be influenced by $\overline{AEN}$ and the arbitration mechanism provided by the 8289.

As an example, suppose the processor issues an I/O bus command. The 8288 Bus Controller generates the necessary control signal to latch the I/O address and configure the transceivers in the correct direction. In the IOB mode, the multiplexed MCE/$\overline{PDEN}$ pin of the 8288 becomes $\overline{PDEN}$ (peripheral data enable) and serves to enable the I/O bus's data transceivers during I/O bus commands. DEN similarily serves to enable the system bus's data transceivers during memory commands. $\overline{PDEN}$ and DEN are mutually exclusive, so it is not possible for both sets of transceivers to be on, thereby avoiding contention between the two sets. Since the I/O bus commands are generated independently of $\overline{AEN}$ In the IOB mode, the I/O bus has no delay effects due to the arbiter. During this time in which the processor is accessing memory the arbiter, if it already has the bus, will permit it to be surrendered to either a higher or lower priority independently of where the processor is in

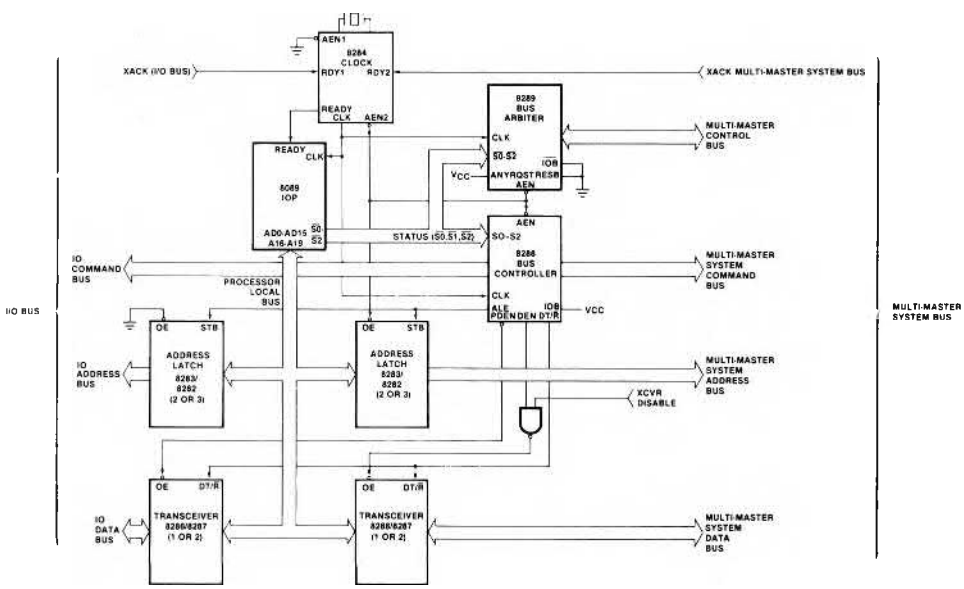


Figure 14. 8289 Configured In I/O Bus Mode With 8089 I/O Processor

its transfer cycle (i.e., independent of the machine state).* If the arbiter does not already have the bus, it will make no effort to acquire the bus.

If the processor issues a memory command instead, the same set of events take place, except that 1) the system bus's data transceivers are enabled instead of the peripherals bus's data transceivers, and 2) when the command is issued depends upon the state of the arbiter. In both cases of I/O bus commands and system bus commands, the address generated for that command is latched into both sets of address latches, the system bus's address latches, and the peripherals bus's address latches. For each command (regardless of command type), an address is put out on the I/O bus and on the system bus if the arbiter has the bus at that particular time. However, the bus controller only issues a command to one of the buses and hence, no ill effects are suffered by addressing both buses.

If the arbiter already has the system bus when a system bus command is issued, no delays due to the arbiter will be noticed by the processor. If the arbiter doesn't have the bus and must acquire it, then the processor will be delayed (via the system bus command being delayed by the bus controller through $\overline{AEN}$ from the arbiter) until the arbiter has acquired the bus. The arbiter will then permit the bus controller to issue the command and the transfer cycle continues.

RESB INTERFACE

The non-I/O processors in the 8086 family can communicate with both a resident bus and a multi-master system bus. Two bus controllers would be needed in such a configuration as shown in Figure 15. In such a system configuration the processor would have to access to memory and peripherals of both buses. Address mapping techniques can be applied to select which bus is to be accessed. The SYSB/$\overline{RESB}$ (system bus/resident bus) input on the arbiter serves to instruct the arbiter as to whether or not the system bus is to be accessed. It also enables or disables commands from one of the bus controllers.

In such a system configuration, it is possible to issue both memory and I/O commands to either bus and as a result, two bus controllers are needed, one for each bus. Since the controllers have to issue both memory and I/O commands to their respective buses, the IOB options on the controllers are strapped off (IOB is low). The arbiter, too, has to be informed of the system configuration in order to respond appropriately to system inputs and has its RESB option strapped on (RESB is high). The arbiter's IOB option is strapped inactive ($\overline{IOB}$ is high). Strapping the arbiter into the resident bus mode enables the arbiter to respond to the state of the SYSB/$\overline{RESB}$ input. Depending upon the state of this input, the arbiter either requests and acquires the system bus or permits the surrendering of that bus.

*Under other circumstances, bus surrendering would only be permitted during the period from where address to command hold time has been established just prior to where the next command would be issued.

In the system shown in Figure 15, memory mapping techniques are applied on the resident bus side of the system rather than on the multiprocessor or system bus side. As mentioned earlier in the IOB interface, both sets of address latches (the resident bus's address latches and the system bus's address latches) are latched with the same address; in this case, by their respective bus controllers.* The system bus's address latches, however, may or may not be enabled depending upon the state of the arbiter. The resident bus's address latches are always enabled, hence the address mapping technique is applied to the resident bus.

Address mapping techniques can range in complexity from a single bit of the address bus (usually the most significant bit of the address), to a decoder, to a PROM. The more elaborate mapping technique, such as PROM, provides segment mapping, system flexibility, and easy mapping modifications (simply make a new PROM).

In actual operation, both bus controllers respond to the processor's status lines and both will simultaneously issue an address latch strobe (ALE) to their respective address latches. Both bus controllers will issue command and control signals unless inhibited. The purpose of the address mapping circuitry is to inhibit one of the bus controllers before contention or erroneous commands can occur. The transceivers are enabled off the same clock edge the commands are issued, namely $\phi 1$ of T2 (Figure 16). The address is strobed into the address latches by ALE. ALE is activated as soon as the processor issues status, and is terminated on $\phi 2$ of of T1. From when ALE is issued, plus the propagation delay of the address latches, determines where the address is valid. The time from which the address is valid to where control and commands are issued determines how much settling time is available for the address mapping circuitry. The mapping circuitry must inhibit (via CEN) one of the bus controllers prior to where controls and commands are issued. Part of the settling time (see Figure 16) is consumed as a setup time requirement to the bus controllers. As it turns out, CEN (command enable) can be disqualified as late as on the falling edge of clock (the leading edge of $\phi 1$ of T2) without fear of the bus controller issuing any commands or transceiver control signals. In systems (8 MHz) where less time is available for the address mapping circuitry, the address latches can be bypassed, hooking the mapping circuitry straight onto the processor's multiplexed address/data bus (the local bus) and using ALE to strobe the mapping circuitry. This would avoid the propagation delay time of the transceivers. Besides needing to inhibit one of the bus controllers, the arbiter needs to be informed of the address mapping circuitry's decision. Depending upon that decision, the arbiter acquires or permits the release of the system bus.

*A simpler system with an 8086 or 8088 can exist, if it is desirable to only have PROM, ROM, or a read only peripheral interface on the resident bus. The 8086 and 8088 additionally generate a read signal in conjunction with the 8288 control signals. By using this read signal and memory mapping, the 8086 or 8088 could operate from local program store without having the contention of using the system bus.

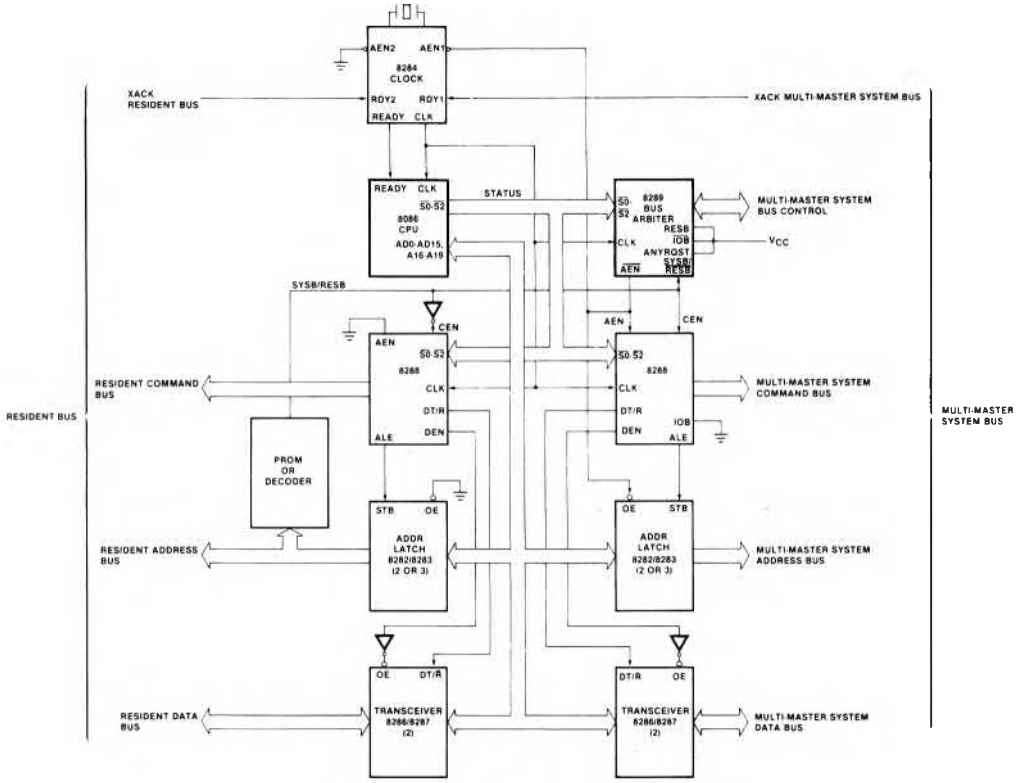


Figure 15. 8289 Configured In Resident Bus Mode

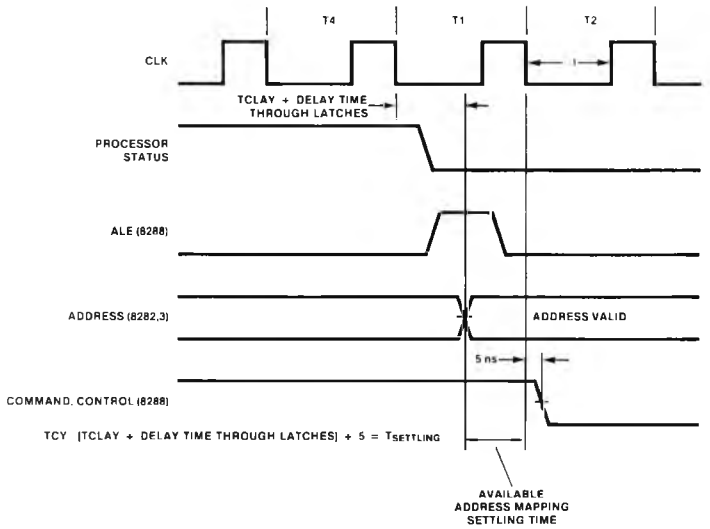


Figure 16. Time Available For Address Mapping Prom

The arbiter is informed of this decision via its SYSB/$\overline{\text{RESB}}$ input. If the memory mapping circuitry selects the resident bus, then SYSB/$\overline{\text{RESB}}$ input to the arbiter and CEN input of the system bus controller are brought low; and the CEN Input of the resident bus controller is brought high. The commands and control signals of the resident bus are now enabled and those of the system bus are disabled. In addition, with the arbiter being informed that the transfer cycle is occurring on the resident bus, the system bus is permitted to be surrendered. Glitching is permitted on the SYSB/$\overline{\text{RESB}}$ input of the arbiter up until $\phi 1$ of T2. Thereafter, only clean transitions can occur on the input.* So, if mapping circuitry can settle prior to $\phi 1$ of T2, there is no need to be concerned over glitching. If the mapping circuitry is unable to settle prior to this time, then the designer must guarantee a clean transition on the SYSB/$\overline{\text{RESB}}$ input.

INTERFACE TO TWO MULTI-MASTER BUSES

The interface of an 8086 family processor to two multi-system buses is simply an extension of the resident bus interface. The only difference is that now two arbiters are needed, one for each multi-master bus, and the address mapping circuitry must acquire its input straight off the processor's multiplexed address/data bus (the local bus), using ALE as an address strobe input. Figure 17 depicts how such a system might be configured.

Figure 17 illustrates the use of the 8289 in a system environment in three of its four modes. The host 8086 CPU (priority 3) is using the 8289 in its single bus multi-master mode, while an 8089 I/O processor is using the 8289 in its IOB mode. A work station based on an 8088 processor uses the 8289 in it system/resident bus mode. This diagram represents a hypothetical system wherein there can exist more than one work station (only one shown). Each work station shares system resources and I/O. The lowest priority processor (8086) would provide supervisory functions and system control, i.e., allow operator intervention into the system resources. A work station would call in assemblers and compilers or application programs as needed. When compiled or assembled, the results are transferred to the I/O station for output, thus freeing up a work station for another user.

*In certain memory mapping techniques, the CENs of the bus controllers are controlled differently from the SYSB/$\overline{\text{RESB}}$ input of the arbiter. In short, CEN is brought low automatically to both bus controllers, thereby disabling their command and control outputs. This permits a longer settling time for the memory mapping circuitry, since both controllers are disabled. When the mapping circuitry settles, sometime after $\phi 1$ of T2, one of the bus controllers and its associated bus arbiter (if one exists) is enabled. After $\phi 1$ of T2, the arbiter can only permit clean transitions on the SYSB/$\overline{\text{RESB}}$ input line.

If one work station is used, the serial priority resolving technique could be used between the 8289 Bus Arbiters (shown in dotted lines). If more than one work station is desired, it would be necessary to either slow down the system bus clock to accommodate the additional arbiters, or resort to the parallel resolving technique (as shown).

WHEN TO USE THE DIFFERENT MODES

Single Bus Multi-Master Interface

This mode is the simplest and is sufficient for systems where a multiprocessing environment exists and the system bus bandwidth is sufficient to handle the peak concurrent requirements of a multi-master environment. This solution can provide an inexpensive solution for multi-masters to access an expensive I/O device. If, however, the system bus bandwidth is exceeded, the IOB or system/resident modes should be considered.

IOB Mode

The IOB mode is ideal when the bus can be separated into an I/O bus and memory or system bus. This mode is commonly used with the 8089 I/O processor in its REMOTE configuration to separate the I/O space from memory space. With the 8089, all instructions operate on either system or I/O address space. 64K bytes of I/O space can be accessed by the processors in the 8086 family.

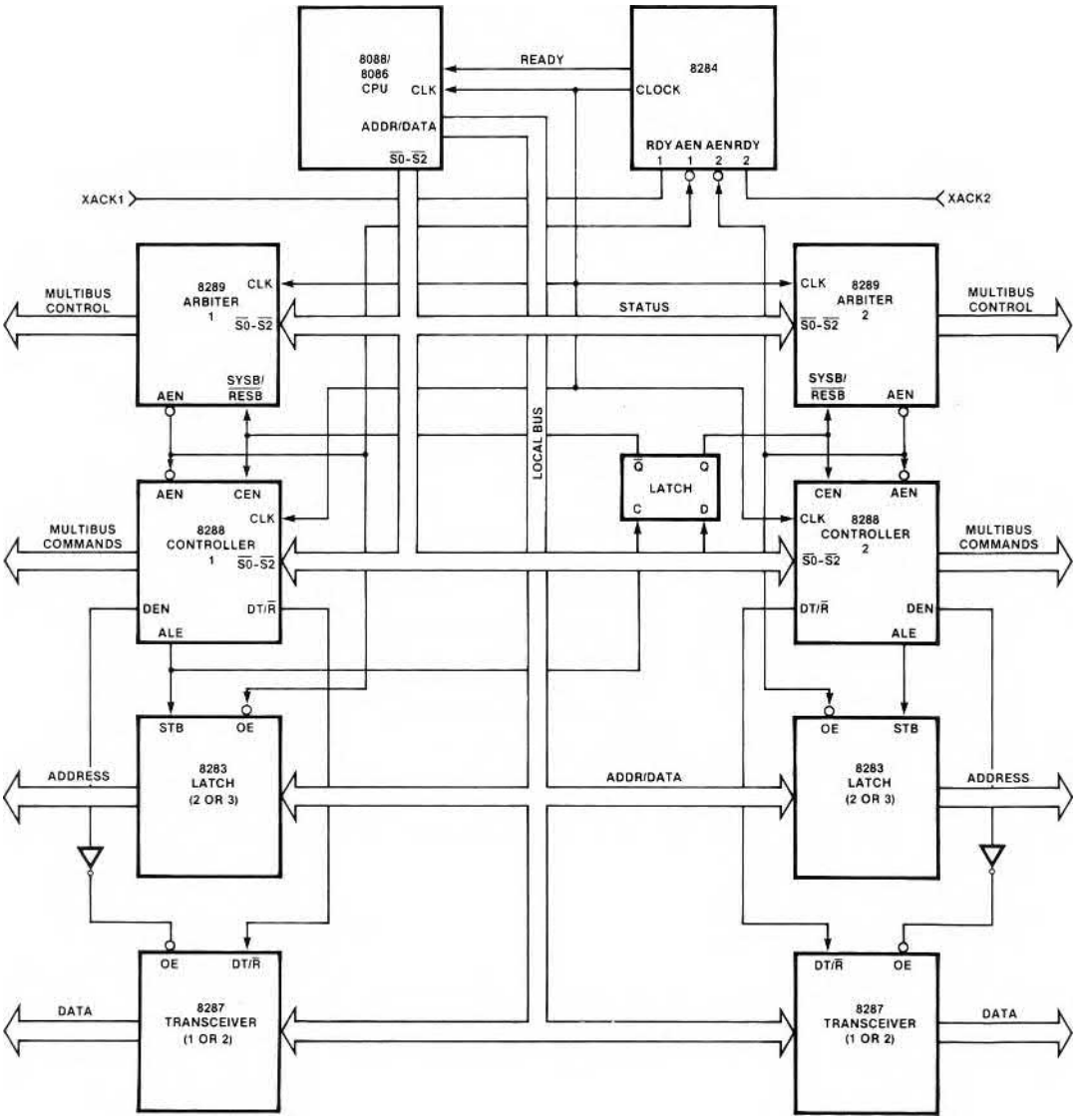
The remaining processors in the 8086 family are constrained to using only I/O instructions when referencing I/O space. If this is a limitation, and it is desirable to remove some of the processor functions to its private resources, the resident bus mode should be considered.

Resident Bus Mode

The resident bus mode allows for maximum flexibility for a CPU device, giving it both access to its own local resources with full instruction set capability, and the system resources. The CPU can work from its own local resources without contention on the system bus. By using a PROM for memory mapping, memory space can be easily altered in this mode. This mode requires the use of a second 8288 bus controller chip.

CONCLUSION

The 8289 brings a new dimension to microcomputer architecture by allowing the advanced 8/16-bit microprocessors to play easily in a multi-master, multiprocessing environment. With the flexible modes of the 8289, a user can define one of several bus architectures to meet his cost/performance needs. Modularity, improved system reliability and increased performance are just a few of the benefits that designing a multiprocessing system provides.



MEMORY MAPPING DECODING IS SHOWN TAKING PLACE DIRECTLY OFF OF THE PROCESSOR'S LOCAL MULTIPLEXED ADDRESS/DATA BUS.

Figure 17. Using 8289s To Interface To Two Multimaster System Buses.

THIS PAGE LEFT INTENTIONALLY BLANK

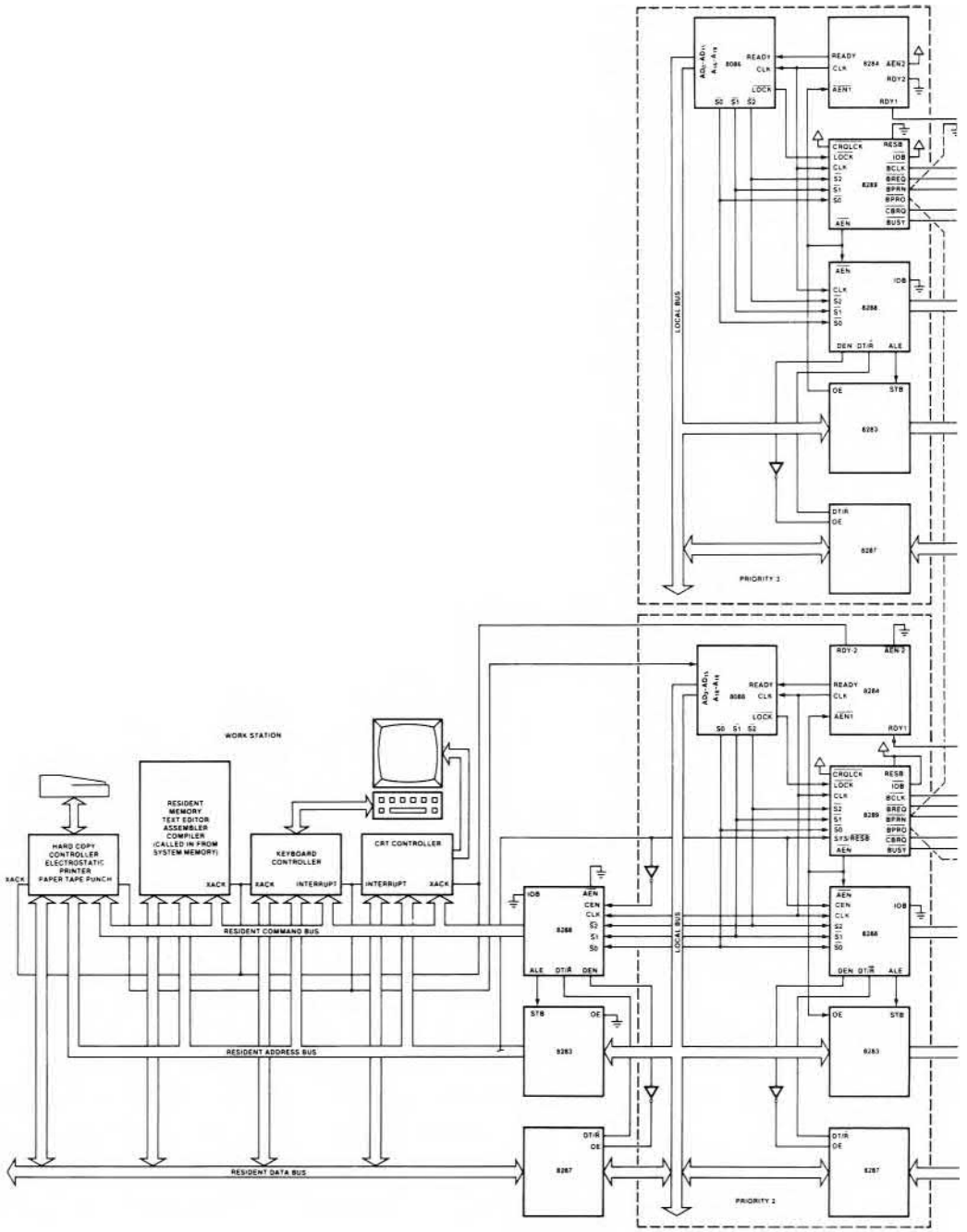


Figure 18. 8289 Used In Each Of 3 Modes, Single Bus, I/O Bus, and Resident Bus Modes Implementing A Hypothetical Multimaster Bus System

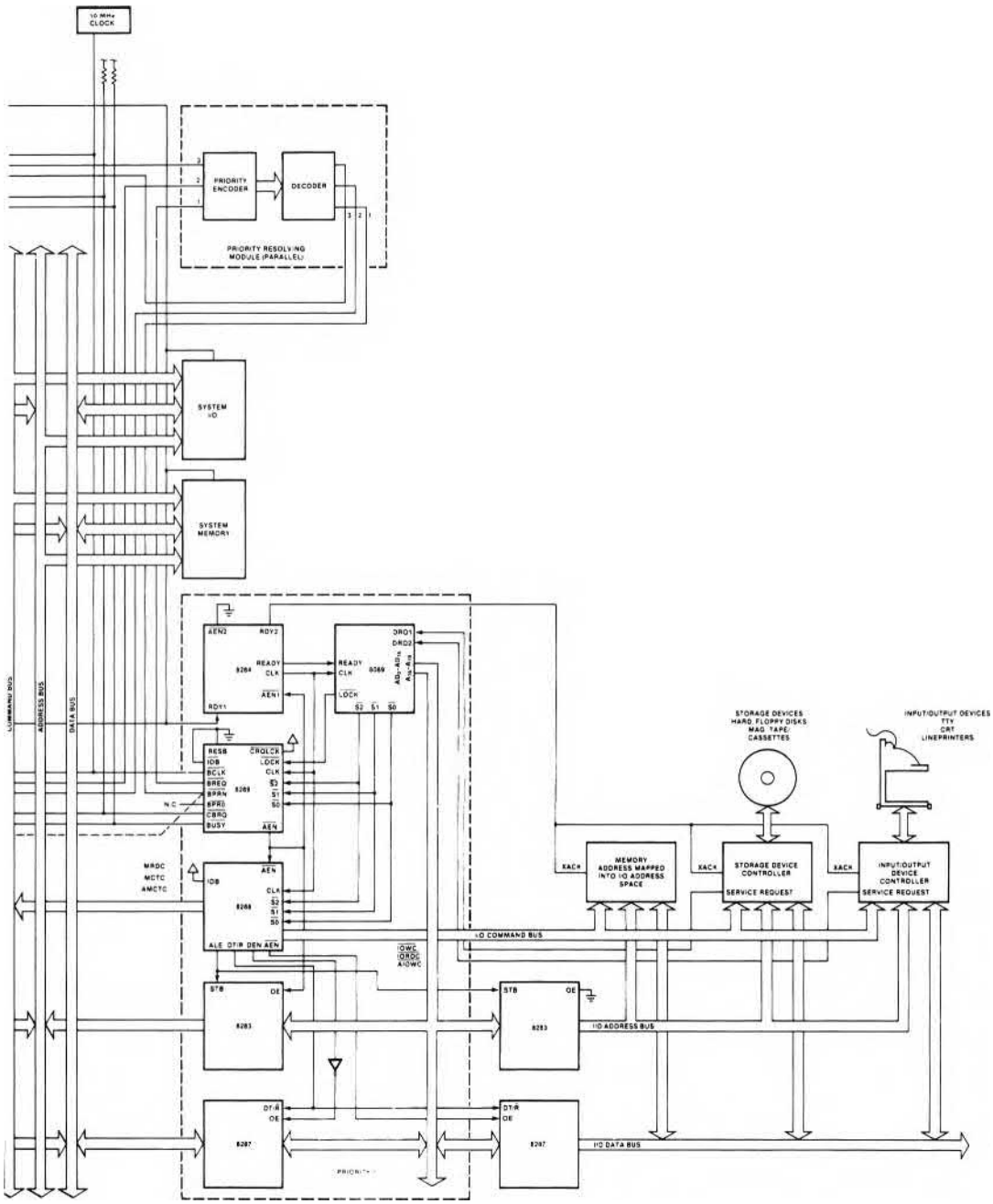


Figure 18. 8289 Used In Each Of 3 Modes, Single Bus, I/O Bus, and Resident Bus Modes Implementing A Hypothetical Multimaster Bus System

intel

APPLICATION NOTE

AP-59

September 1979

Using the 8259A Programmable Interrupt Controller

Robin Jigour
Microcomputer Applications

© Intel Corporation, 1979

121500-001

Using the 8259A Programmable Interrupt Controller

Contents

INTRODUCTION

CONCEPTS

MCS80™-8259A Overview
MCS85™-8259A Overview
MCS86/88™-8259A Overview

FUNCTIONAL BLOCK DIAGRAM

Interrupt Registers and Control Logic
Other Functional Blocks
Pin Functions

OPERATION OF THE 8259A

Interrupt Vectoring
- MCS80/85 Mode
- MCS86/88 Mode

Interrupt Priorities
- Fully Nested Mode
- End of Interrupt
- Automatic Rotation
- Specific Rotation
- Interrupt Masking

Interrupt Triggering
- Level Triggered Mode
- Edge Triggered Mode

Interrupt Status
- Reading Interrupt Registers
- Poll Command

Interrupt Cascading
- Cascade Mode
- Special Fully Nested Mode
- Buffered Mode

PROGRAMMING THE 8259A

Initialization Command Words (ICWs)
Operational Command Words (OCWs)

APPLICATION EXAMPLES

Power Fail/Auto Start with Battery Back-Up RAM
78 Level Interrupt Structure
Timer Controlled Interrupts

CONCLUSIONS

APPENDIX A

APPENDIX B

INTRODUCTION

The Intel 8259A is a Programmable Interrupt Controller (PIC) designed for use in real-time interrupt driven microcomputer systems. The 8259A manages eight levels of interrupts and has built-in features for expansion up to 64 levels with additional 8259A's. Its versatile design allows it to be used within MCS-80, MCS-85, MCS-86, and MCS-88 microcomputer systems. Being fully programmable, the 8259A provides a wide variety of modes and commands to tailor 8259A interrupt processing for the specific needs of the user. These modes and commands control a number of interrupt oriented functions such as interrupt priority selection and masking of interrupts. The 8259A programming may be dynamically changed by the software at any time, thus allowing complete interrupt control throughout program execution.

The 8259A is an enhanced, fully compatible revision of its predecessor, the 8259. This means the 8259A can use all hardware and software originally designed for the 8259 without any changes. Furthermore, it provides additional modes that increase its flexibility in MCS-80 and MCS-85 systems and allow it to work in MCS-86 and MCS-88 systems. These modes are:

- MCS-86/88 Mode
- Automatic End of Interrupt Mode
- Level Triggered Mode
- Special Fully Nested Mode
- Buffered Mode

Each of these are covered in depth further in this application note.

This application note was written to explain completely how to use the 8259A within MCS-80, MCS-85, MCS-86, and MCS-88 microcomputer systems. It is divided into five sections. The first section, "Concepts", explains the concepts of interrupts and presents an overview of how the 8259A works with each microcomputer system mentioned above. The second section, "Functional Block Diagram", describes the internal functions of the 8259A in block diagram form and provides a detailed functional description of each device pin. "Operation of the 8259A", the third section, explains in depth the operation and use of each of the 8259A modes and commands. For clarity of explanation, this section doesn't make reference to the actual programming of the 8259A. Instead, all programming is covered in the fourth section, "Programming the 8259A". This section explains how to program the 8259A with the modes and commands mentioned in the previous section. These two sections are referenced in Appendix A. The fifth and final section "Application Examples", shows the 8259A in three typical applications. These applications are fully explained with reference to both hardware and software.

The reader should note that some of the terminology used throughout this application note may differ slightly from existing data sheets. This is done to better clarify and explain the operation and programming of the 8259A.

1. CONCEPTS

In microcomputer systems there is usually a need for the processor to communicate with various Input/Output (I/O) devices such as keyboards, displays, sensors, and other peripherals. From the system viewpoint, the processor should spend as little time as possible servicing the peripherals since the time required for these I/O chores directly affects the amount of time available for other tasks. In other words, the system should be designed so that I/O servicing has little or no effect on the total system throughput. There are two basic methods of handling the I/O chores in a system: status polling and interrupt servicing.

The status poll method of I/O servicing essentially involves having the processor "ask" each peripheral if it needs servicing by testing the peripheral's status line. If the peripheral requires service, the processor branches to the appropriate service routine; if not, the processor continues with the main program. Clearly, there are several problems in implementing such an approach. First, how often a peripheral is polled is an important constraint. Some idea of the "frequency-of-service" required by each peripheral must be known and any software written for the system must accommodate this time dependence by "scheduling" when a device is polled. Second, there will obviously be times when a device is polled that is not ready for service, wasting the processor time that it took to do the poll. And other times, a ready device would have to wait until the processor "makes its rounds" before it could be serviced, slowing down the peripheral.

Other problems arise when certain peripherals are more important than others. The only way to implement the "priority" of devices is to poll the high priority devices more frequently than lower priority ones. It may even be necessary to poll the high priority devices while in a low priority device service routine. It is easy to see that the polled approach can be inefficient both time-wise and software-wise. Overall, the polled method of I/O servicing can have a detrimental effect on system throughput, thus limiting the tasks that can be performed by the processor.

A more desirable approach in most systems would allow the processor to be executing its main program and only stop to service the I/O when told to do so by the I/O itself. This is called the interrupt service method. In effect, the device would asynchronously signal the processor when it required service. The processor would finish its current instruction and then vector to the service routine for the device requesting service. Once the service routine is complete, the processor would resume exactly where it left off. Using the interrupt service method, no processor time is spent testing devices, scheduling is not needed, and priority schemes are readily implemented. It is easy to see that, using the interrupt service approach, system throughput would increase, allowing more tasks to be handled by the processor.

However, to implement the interrupt service method between processor and peripherals, additional hardware is usually required. This is because, after interrupting the processor, the device must supply information for vectoring program execution. Depending on the processor used, this can be accomplished by the device taking control of the data bus and "jamming" an instruction(s) onto it. The instruction(s) then vectors the pro-

gram to the proper service routine. This of course requires additional control logic for each interrupt requesting device. Yet the implementation so far is only in the most basic form. What if certain peripherals are to be of higher priority than others? What if certain interrupts must be disabled while others are to be enabled? The possible variations go on, but they all add up to one theme; to provide greater flexibility using the interrupt service method, hardware requirements increase.

So, we're caught in the middle. The status poll method is a less desirable way of servicing I/O in terms of throughput, but its hardware requirements are minimal. On the other hand, the interrupt service method is most desirable in terms of flexibility and throughput, but additional hardware is required.

The perfect situation would be to have the flexibility and throughput of the interrupt method in an implementation with minimal hardware requirements. The 8259A Programmable Interrupt Controller (PIC) makes this all possible.

The 8259A Programmable Interrupt Controller (PIC) was designed to function as an overall manager of an interrupt driven system. No additional hardware is required. The 8259A alone can handle eight prioritized interrupt levels, controlling the complete interface between peripherals and processor. Additional 8259A's can be "cascaded" to increase the number of interrupt levels processed. A wide variety of modes and commands for programming the 8259A give it enough flexibility for almost any interrupt controlled structure. Thus, the 8259A is the feasible answer to handling I/O servicing in microcomputer systems.

Now, before explaining exactly how to use the 8259A, let's go over interrupt structures of the MCS-80, MCS-85, MCS-86, and MCS-88 systems, and how they interact with the 8259A. Figure 1 shows a block diagram of the 8259A interfacing with a standard system bus. This may prove useful as reference throughout the rest of the "Concepts" section.

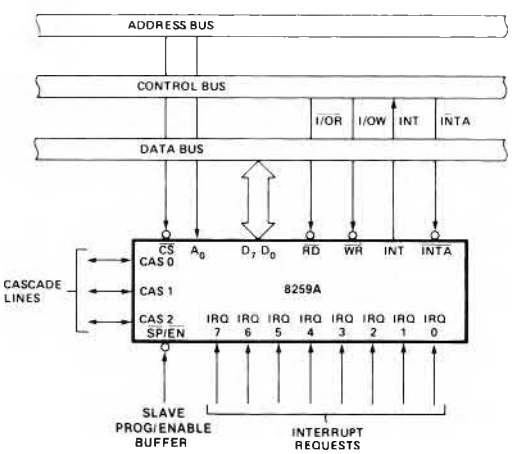


Figure 1. 8259A Interface to Standard System Bus

1.1 MCS-80™—8259A OVERVIEW

In an MCS-80—8259A interrupt configuration, as in Figure 2, a device may cause an interrupt by pulling one of the 8259A's interrupt request pins (IR0-IR7) high. If the 8259A accepts the interrupt request (this depends on its programmed condition), the 8259A's INT (interrupt) pin will go high, driving the 8080A's INT pin high.

The 8080A can receive an interrupt request any time, since its INT input is asynchronous. The 8080A, however, doesn't always have to acknowledge an interrupt request immediately. It can accept or disregard requests under software control using the EI (Enable Interrupt) or DI (Disable Interrupt) instructions. These instructions either set or reset an internal interrupt enable flip-flop. The output of this flip-flop controls the state of the INTE (Interrupt Enabled) pin. Upon reset, the 8080A interrupts are disabled, making INTE low.

At the end of each instruction cycle, the 8080A examines the state of its INT pin. If an interrupt request is present and interrupts are enabled, the 8080A enters an interrupt machine cycle. During the interrupt machine cycle the 8080A resets the internal interrupt enable flip-flop, disabling further interrupts until an EI instruction is executed. Unlike normal machine cycles, the interrupt machine cycle doesn't increment the program counter. This ensures that the 8080A can return to the pre-interrupt program location after the interrupt is completed. The 8080A then issues an $\overline{\text{INTA}}$ (Interrupt Acknowledge) pulse via the 8228 System Controller Bus Driver. This $\overline{\text{INTA}}$ pulse signals the 8259A that the 8080A is honoring the request and is ready to process the interrupt.

The 8259A can now vector program execution to the corresponding service routine. This is done during a sequence of the three $\overline{\text{INTA}}$ pulses from the 8080A via the 8228. Upon receiving the first $\overline{\text{INTA}}$ pulse the 8259A places the opcode for a CALL instruction on the data bus. This causes the contents of the program counter to be pushed onto the stack. In addition, the CALL instruction causes two more $\overline{\text{INTA}}$ pulses to be issued, allowing the 8259A to place onto the data bus the starting address of the corresponding service routine. This address is called the interrupt-vector address. The lower 8 bits (LSB) of the interrupt-vector address are released during the second $\overline{\text{INTA}}$ pulse and the upper 8 bits (MSB) during the third $\overline{\text{INTA}}$ pulse. Once this sequence is completed, program execution then vectors to the service routine at the interrupt-vector address.

If the same registers are used by both the main program and the interrupt service routine, their contents should be saved when entering the service routine. This includes the Program Status Word (PSW) which consists of the accumulator and flags. The best way to do this is to "PUSH" each register used onto the stack. The service routine can then "POP" each register off the stack in the reverse order when it is completed. This prevents any ambiguous operation when returning to the main program.

Once the service routine is completed, the main program may be re-entered by using a normal RET (Return) instruction. This will "POP" the original con-

tents of the program counter back off the stack to resume program execution where it left off. Note, that because interrupts are disabled during the interrupt acknowledge sequence, the EI instruction must be executed either during the service routine or the main program before further interrupts can be processed.

For additional information on the 8080A interrupt structure and operation, refer to the MCS-80 User's Manual.

1.2 MCS-85™—8259A OVERVIEW

An MCS-85—8259A configuration processes interrupts in much the same format as an MCS-80—8259A configuration. When an interrupt occurs, a sequence of three $\overline{INTA}$ pulses causes the 8259A to release onto the data bus a CALL instruction and an interrupt-vector address for the corresponding service routine. Other events that occur during the 8080A interrupt machine cycle, such as disabling interrupts and not incrementing the program counter, also occur in the 8085A interrupt acknowledge machine cycle. Additionally, the instructions for saving registers, enabling or disabling of interrupts, and returning from service routines are literally the same.

The 8085A, however, has a different interrupt hardware scheme as shown in Figure 3. For one, the 8085A supplies its own $\overline{INTA}$ output pin rather than using an addi-

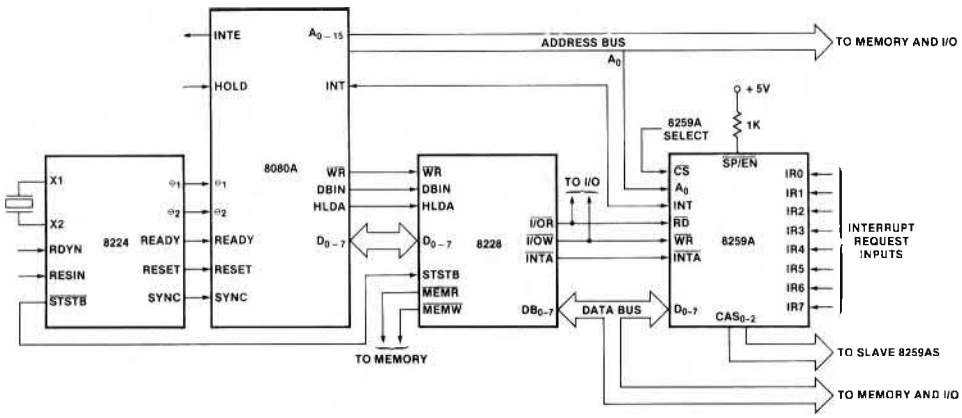


Figure 2. MCS-80 8259A Basic Configuration Example

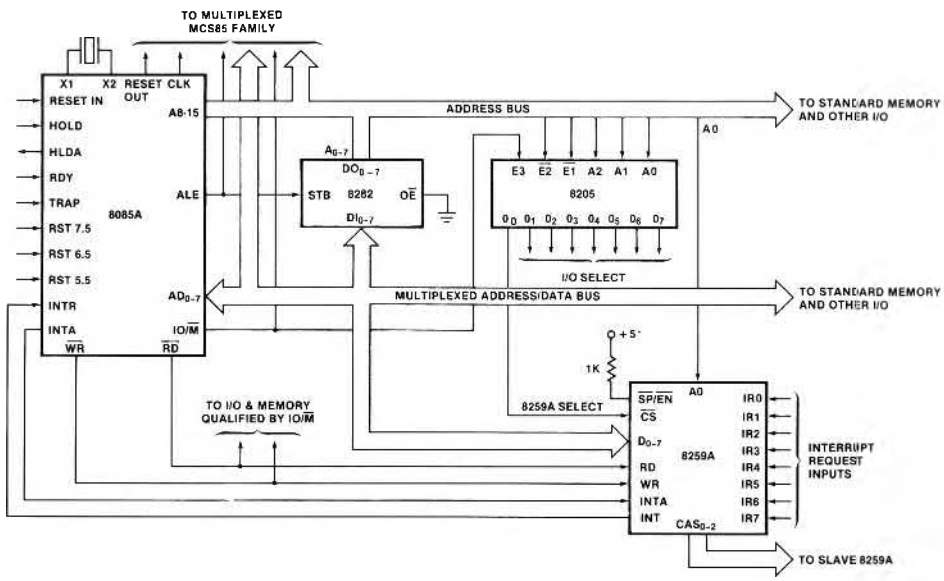


Figure 3. MCS-85™ 8259A Basic Configuration Example

tional chip, as the 8080A uses the 8228 System Controller Bus Driver. Another hardware difference is the 8085A has five hardware interrupt pins: INTR, RST 7.5, RST 6.5, RST 5.5, and TRAP. The INTR (Interrupt Request) pin is the equivalent to the 8080A's INT pin. The RST (Restart) pins and TRAP pin are all restart interrupts which vector program execution to an individual dedicated address when asserted. The important factor associating these interrupts is their relative priority, as shown below:

TRAP	Highest Priority
RST 7.5	
RST 6.5	
RST 5.5	
INTR	Lowest Priority

The INTR pin has lowest priority among the other 8085A hardware interrupts. Thus, precautions to prevent interrupting 8259A service routines may be necessary. This, of course, depends on how the 8085A interrupts are being used in a particular application. Such precautions can be implemented, however, by masking the RST pins using the SIM instruction. The TRAP pin on the other hand is non-maskable; all interrupt pins but TRAP can be controlled by the EI (Enable Interrupt) and DI (Disable Interrupt) instructions.

For a complete description of the 8085A interrupt structure, refer to the MCS-85 User's Manual.

1.3 MCS-86/88™—8259A OVERVIEW

Operation of an MCS-86/88—8259A configuration has basic similarities of the MCS-80/85—8259A configurations. That is, a device can cause an interrupt by pulling one of the 8259A's interrupt request pins (IR0-IR7) high. If the 8259A honors the request, its INT pin will go high, driving the 8086/8088's INTR pin high. Like the 8080A and 8085A, the INTR pin of the 8086/8088 is asynchronous, thus it can receive an interrupt any time. The 8086/8088 can also accept or disregard requests on INTR under software control using the STI (Set Interrupt) or CLI (Clear Interrupt) instructions. These instructions set or clear the interrupt-enabled flag IF. Upon 8086/8088 reset the IF flag is cleared, disabling external interrupts on INTR. Beside the INTR pin, the 8086/8088 provides an NMI (Non-Maskable Interrupt) pin. The NMI functions similar to the 8085A's TRAP; it can't be disabled or masked. NMI has higher priority than INTR.

Figure 4 shows an MCS-86 MAX Mode system interfacing with an 8259A on the local bus. This MCS-86—8259A configuration is also representative of an MCS-88—8259A configuration except for the data bus which is 16 bits for 8086 and 8 bits for 8088. In the MCS-86 system the 8259A must be on the lower 8 bits of the data bus. Note that the 8259A could also be interfaced on the system bus.

Although there are some basic similarities, the actual processing of interrupts with an 8086/8088 is different than an 8080A or 8085A. When an interrupt request is present and interrupts are enabled, the 8086/8088 enters its interrupt acknowledge machine cycle. The interrupt acknowledge machine cycle pushes the flag registers onto the stack (as in a PUSHF instruction). It then clears the IF flag which disables interrupts. The contents of

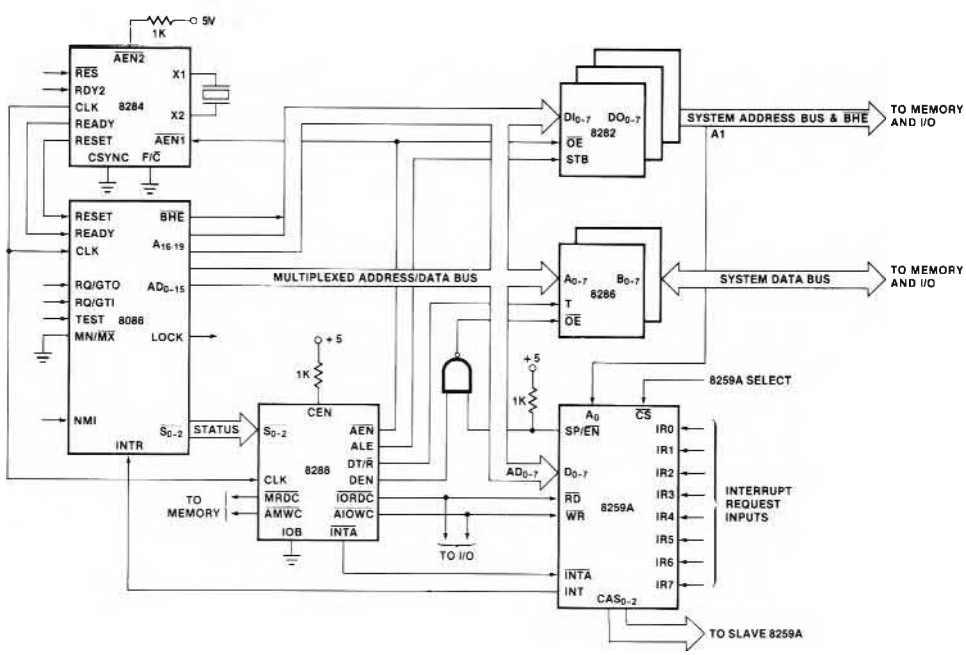


Figure 4. MSC-86™ 8259A Basic Configuration Example (8086 in Max. Mode)

both the code segment and the instruction pointer are then also pushed onto the stack. Thus, the stack retains the pre-interrupt flag status and pre-interrupt program location which are used to return from the service routine. The 8086/8088 then issues the first of two $\overline{INTA}$ pulses which signal the 8259A that the 8086/8088 has honored its interrupt request. If the 8086/8088 is used in its "MIN Mode" the $\overline{INTA}$ signal is available from the 8086/8088 on its $\overline{INTA}$ pin. If the 8086/8088 is used in the "MAX Mode" the $\overline{INTA}$ signal is available via the 8288 Bus Controller $\overline{INTA}$ pin. Additionally, in the "MAX Mode" the 8086/8088 LOCK pin goes low during the interrupt acknowledge sequence. The LOCK signal can be used to indicate to other system bus masters not to gain control of the system bus during the interrupt acknowledge sequence. A "HOLD" request won't be honored while LOCK is low.

The 8259A is now ready to vector program execution to the corresponding service routine. This is done during the sequence of the two $\overline{INTA}$ pulses issued by the 8086/8088. Unlike operation with the 8080A or 8085A, the 8259A doesn't place a CALL instruction and the starting address of the service routine on the data bus. Instead, the first $\overline{INTA}$ pulse is used only to signal the 8259A of the honored request. The second $\overline{INTA}$ pulse causes the 8259A to place a single interrupt-vector byte onto the data bus. Not used as a direct address, this interrupt-vector byte pertains to one of 256 interrupt "types" supported by the 8086/8088 memory. Program execution is vectored to the corresponding service routine by the contents of a specified interrupt type.

All 256 interrupt types are located in absolute memory locations 0 through 3FFH which make up the 8086/8088's interrupt-vector table. Each type in the interrupt-vector table requires 4 bytes of memory and stores a code segment address and an instruction pointer address. Figure 5 shows a block diagram of the interrupt-vector table. Locations 0 through 3FFH should be reserved for the interrupt-vector table alone. Furthermore, memory locations 00 through 7FH (types 0–31) are reserved for use by Intel Corporation for Intel hardware and software products. To maintain compatibility with present and future Intel products, these locations should not be used.

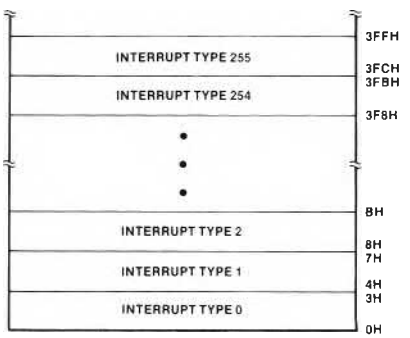


Figure 5. 8086/8088 Interrupt Vector Table

When the 8086/8088 receives an interrupt-vector byte from the 8259A, it multiplies its value by four to acquire the address of the interrupt type. For example, if the interrupt-vector byte specifies type 128 (80H), the vectored address in 8086/8088 memory is 4 × 80H, which equals 200H. Program execution is then vectored to the service routine whose address is specified by the code segment and instruction pointer values within type 128 located at 200H. To show how this is done, let's assume interrupt type 128 is to vector data to 8086/8088 memory location 2FF5FH. Figure 6 shows two possible ways to set values of the code segment and instruction pointer for vectoring to location 2FF5FH. Address generation by the code segment and instruction pointer is accomplished by an offset (they overlap). Of the total 20-bit address capability, the code segment can designate the upper 16 bits, the instruction pointer can designate the lower 16 bits.

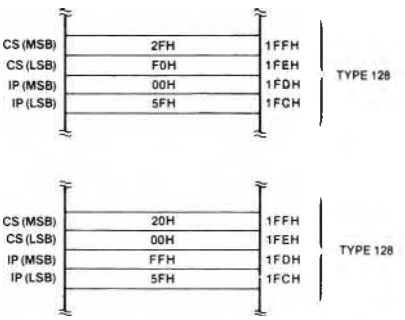


Figure 6. Two Examples of 8086/8088 Interrupt Type 128 Vectoring to Location 2FF5FH

When entering an interrupt service routine, those registers that are mutually used between the main program and service routine should be saved. The best way to do this is to "PUSH" each register used onto the stack immediately. The service routine can then "POP" each register off the stack in the same order when it is completed.

Once the service routine is completed the main program may be re-entered by using a IRET (Interrupt Return) instruction. The IRET instruction will pop the pre-interrupt instruction pointer, code segment and flags off the stack. Thus the main program will resume where it was interrupted with the same flag status regardless of changes in the service routine. Note especially that this includes the state of the IF flag, thus interrupts are re-enabled automatically when returning from the service routine.

Beside external interrupt generation from the INTR pin, the 8086/8088 is also able to invoke interrupts by software. Three interrupt instructions are provided: INT, INT (Type 3), and INTO. INT is a two byte instruction, the second byte selects the interrupt type. INT (Type 3) is a one byte instruction which selects interrupt Type 3. INTO is a conditional one byte interrupt instruction which selects interrupt Type 4 if the OF flag (trap on overflow) is set. All the software interrupts vector program execution as the hardware interrupts do.

For further information on 8086/8088 interrupt operation and internal interrupt structure refer to the MCS-86 User's Manual and the 8086 System Design application note.

2. 8259A FUNCTIONAL BLOCK DIAGRAM

A block diagram of the 8259A is shown in Figure 7. As can be seen from this figure, the 8259A consists of eight major blocks: the Interrupt Request Register (IRR), the In-Service Register (ISR), the Interrupt Mask Register (IMR), the Priority Resolver (PR), the cascade buffer/comparator, the data bus buffer, and logic blocks for control and read/write. We'll first go over the blocks directly related to interrupt handling, the IRR, ISR, IMR, PR, and the control logic. The remaining functional blocks are then discussed.

2.1 INTERRUPT REGISTERS AND CONTROL LOGIC

Basically, interrupt requests are handled by three "cascaded" registers: the Interrupt Request Register (IRR) is use to store all the interrupt levels requesting service; the In-Service Register (ISR) stores all the levels which are being serviced; and the Interrupt Mask Register (IMR) stores the bits of the interrupt lines to be masked. The Priority Resolver (PR) looks at the IRR, ISR and IMR, and determines whether an INT should be issued by the the control logic to the processor.

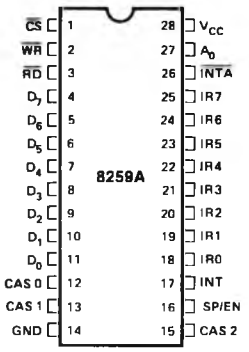
Figure 8 shows conceptually how the Interrupt Request (IR) input handles an interrupt request and how the various interrupt registers interact. The figure represents one of eight "daisy-chained" priority cells, one for each IR input.

The best way to explain the operation of the priority cell is to go through the sequence of internal events that happen when an interrupt request occurs. However, first, notice that the input circuitry of the priority cell allows for both level sensitive and edge sensitive IR inputs. Deciding which method to use is dependent on the particular application and will be discussed in more detail later.

When the IR input is in an inactive state (LOW), the edge sense latch is set. If edge sensitive triggering is selected, the "Q" output of the edge sense latch will arm the input gate to the request latch. This input gate will be disarmed after the IR input goes active (HIGH) and the interrupt request has been acknowledged. This disables the input from generating any further interrupts until it has returned low to re-arm the edge sense latch. If level sensitive triggering is selected, the "Q" output of the edge sense latch is rendered useless. This means the level of the IR input is in complete control of interrupt generation; the input won't be disarmed once acknowledged.

When an interrupt occurs on the IR input, it propagates through the request latch and to the PR (assuming the input isn't masked). The PR looks at the incoming requests and the currently in-service interrupts to ascertain whether an interrupt should be issued to the processor. Let's assume that the request is the only one incoming and no requests are presently in service. The PR then causes the control logic to pull the INT line to the processor high.

PIN CONFIGURATION



PIN NAMES

D_7-D_0	DATA BUS (BI-DIRECTIONAL)
$\overline{RD}$	READ INPUT
$\overline{WR}$	WRITE INPUT
A_0	COMMAND SELECT ADDRESS
$\overline{CS}$	CHIP SELECT
CAS1-CAS0	CASCADE LINES
SP/EN	SLAVE PROGRAM/ENABLE BUFFER
INT	INTERRUPT OUTPUT
$\overline{INTA}$	INTERRUPT ACKNOWLEDGE INPUT
IR0-IR7	INTERRUPT REQUEST INPUTS

BLOCK DIAGRAM

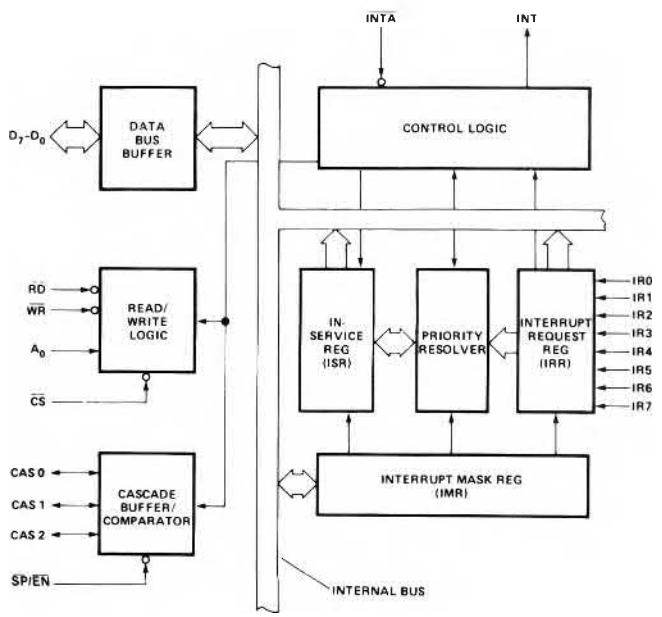


Figure 7. 8259A Block Diagram and Pin Configuration

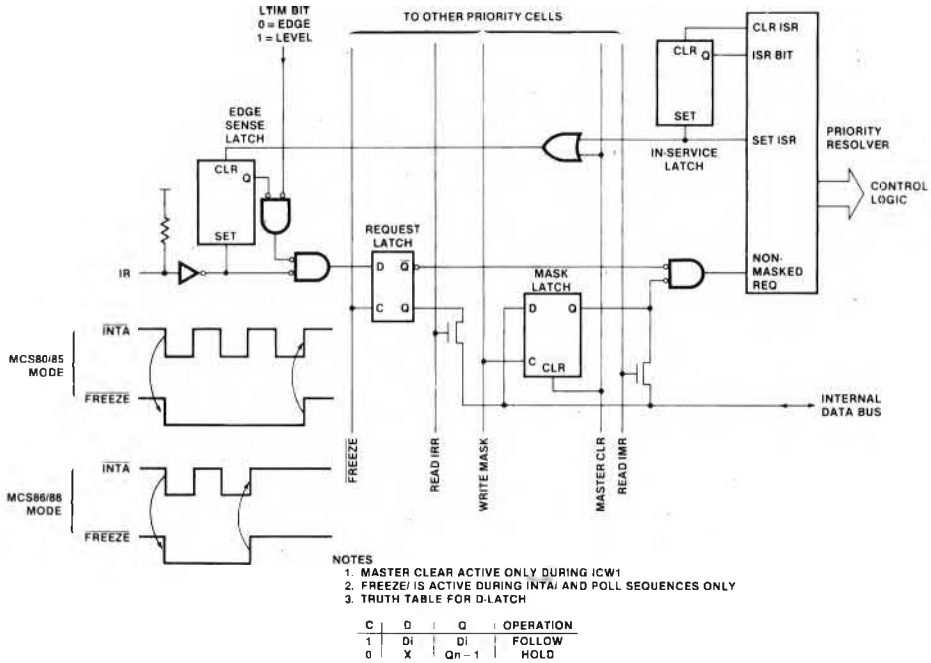


Figure 8. Priority Cell

When the processor honors the INT pulse, it sends a sequence of INTA pulses to the 8259A (three for 8080A/8085A, two for 8086/8088). During this sequence the state of the request latch is frozen (note the INTA-freeze request timing diagram). Priority is again resolved by the PR to determine the appropriate interrupt vectoring which is conveyed to the processor via the data bus.

Immediately after the interrupt acknowledge sequence, the PR sets the corresponding bit in the ISR which simultaneously clears the edge sense latch. if edge sensitive triggering is used, clearing the edge sense latch also disarms the request latch. This inhibits the possibility of a still active IR input from propagating through the priority cell. The IR input must return to an inactive state, setting the edge sense latch, before another interrupt request can be recognized. If level sensitive triggering is used, however, clearing the edge sense latch has no affect on the request latch. The state of the request latch is entirely dependent upon the IR input level. Another interrupt will be generated immediately if the IR level is left active after its ISR bit has been reset. An ISR bit gets reset with an End-of-Interrupt (EOI) command issued in the service routine. End-of-interrupts will be covered in more detail later.

2.2 OTHER FUNCTIONAL BLOCKS

Data Bus Buffer

This three-state, bidirectional 8-bit buffer is used to interface the 8259A to the processor system data bus (via DB0–DB7). Control words, status information, and interrupt-vector data are transferred through the data bus buffer.

Read/Write Control Logic

The function of this block is to control the programming of the 8259A by accepting OUTput commands from the processor. It also controls the releasing of status onto the data bus by accepting INput commands from the processor. The initialization and operation command word registers which store the various control formats are located in this block. The $\overline{RD}$, $\overline{WR}$, $\overline{A0}$, and $\overline{CS}$ pins are used to control access to this block by the processor.

Cascade Buffer/Comparator

As mentioned earlier, multiple 8259A's can be combined to expand the number of interrupt levels. A master-slave relationship of cascaded 8259A's is used for the expansion. The $\overline{SP}/\overline{EN}$ and the CAS0–2 pins are used for operation of this block. The cascading of 8259A's is covered in depth in the "Operation of the 8259A" section of this application note.

2.3 PIN FUNCTIONS

Name	Pin #	I/O	Function
V_{CC}	28	I	+5V supply
GND	14	I	Ground

Name	Pin #	I/O	Function
$\overline{CS}$	1	I	*Chip Select:* A low on this pin enables $\overline{RD}$ and $\overline{WR}$ communication between the CPU and the 8259A. $\overline{INTA}$ functions are independent of $\overline{CS}$.
$\overline{WR}$	2	I	*Write:* A low on this pin when $\overline{CS}$ is low enables the 8259A to accept command words from the CPU.
$\overline{RD}$	3	I	*Read:* A low on this pin when $\overline{CS}$ is low enables the 8259A to release status onto the data bus for the CPU.
D7-D0	4-11	I/O	*Bidirectional Data Bus:* Control, status and interrupt-vector information is transferred via this bus.
CAS0-CAS2	12,13, 15	I/O	*Cascade Lines:* The CAS lines form a private 8259A bus to control a multiple 8259A structure. These pins are outputs for a master 8259A and inputs for a slave 8259A.
$\overline{SP}/\overline{EN}$	16	I/O	*Slave Program/Enable Buffer:* This is a dual function pin. When in the buffered mode it can be used as an output to control buffer transceivers ($\overline{EN}$). When not in the buffered mode it is used as an input to designate a master ($\overline{SP}$ = 1) or slave ($\overline{SP}$ = 0).
INT	17	O	*Interrupt:* This pin goes high whenever a valid interrupt request is asserted. It is used to interrupt the CPU, thus it is connected to the CPU's interrupt pin.
IR0-IR7	18-25	I	*Interrupt Requests:* Asynchronous inputs. An interrupt request can be generated by raising an IR input (low to high) and holding it high until it is acknowledged (edge triggered mode), or just by a high level on an IR input (level triggered mode).
$\overline{INTA}$	26	I	*Interrupt Acknowledge:* This pin is used to enable 8259A interrupt-vector data onto the data bus. This is done by a sequence of interrupt acknowledge pulses issued by the CPU.
A0	27	I	*A0 Address Line:* This pin acts in conjunction with the $\overline{CS}$, $\overline{WR}$, and $\overline{RD}$ pins. It is used by the 8259A to decipher between various command words the CPU writes and status the CPU wishes to read. It is typically connected to the CPU A0 address line (A1 for 8086/8088).

3. OPERATION OF THE 8259A

Interrupt operation of the 8259A falls under five main categories: vectoring, priorities, triggering, status, and cascading. Each of these categories use various modes and commands. This section will explain the operation of these modes and commands. For clarity of explanation, however, the actual programming of the 8259A isn't covered in this section but in "Programming the 8259A". Appendix A is provided as a cross reference between these two sections.

3.1 INTERRUPT VECTORING

Each IR input of the 8259A has an individual interrupt-vector address in memory associated with it. Designation of each address depends upon the initial programming of the 8259A. As stated earlier, the interrupt sequence and addressing of an MCS-80 and MCS-85 system differs from that of an MCS-86 and MCS-88 system. Thus, the 8259A must be initially programmed in either a MCS-80/85 or MCS-86/88 mode of operation to insure the correct interrupt vectoring.

MCS-80/85™ Mode

When programmed in the MCS-80/85 mode, the 8259A should only be used within an 8080A or an 8085A system. In this mode the 8080A/8085A will handle interrupts in the format described in the "MCS-80—8259A or MCS-85—8259A Overviews."

Upon interrupt request in the MCS-80/85 mode, the 8259A will output to the data bus the opcode for a CALL instruction and the address of the desired routine. This is in response to a sequence of three $\overline{INTA}$ pulses issued by the 8080A/8085A after the 8259A has raised INT high.

The first INTA pulse to the 8259A enables the CALL opcode "CD_H" onto the data bus. It also resolves IR priorities and effects operation in the cascade mode, which will be covered later. Contents of the first interrupt-vector byte are shown in Figure 9A.

During the second and third $\overline{INTA}$ pulses, the 8259A conveys a 16-bit interrupt-vector address to the 8080A/8085A. The interrupt-vector addresses for all eight levels are selected when initially programming the 8259A. However, only one address is needed for programming. Interrupt-vector addresses of IR0-IR7 are automatically set at equally spaced intervals based on the one programmed address. Address intervals are user definable to 4 or 8 bytes apart. If the service routine for a device is short it may be possible to fit the entire routine within an 8-byte interval. Usually, though, the service routines require more than 8 bytes. So, a 4-byte interval is used to store a Jump (JMP) instruction which directs the 8080A/8085A to the appropriate routine. The 8-byte interval maintains compatibility with current 8080A/8085A Restart (RST) instruction software, while the 4-byte interval is best for a compact jump table. If the 4-byte interval is selected, then the 8259A will automatically insert bits A0-A4. This leaves A5-A15 to be programmed by the user. If the 8-byte interval is selected, the 8259A will automatically insert bits A0-A5. This leaves only A6-A15 to be programmed by the user.

The LSB of the interrupt-vector address is placed on the data bus during the second $\overline{INTA}$ pulse. Figure 9B shows the contents of the second interrupt-vector byte for both 4 and 8-byte intervals.

The MSB of the interrupt-vector address is placed on the data bus during the third $\overline{INTA}$ pulse. Contents of the third interrupt-vector byte is shown in Figure 9C.

	D7	D6	D5	D4	D3	D2	D1	D0
CALL CODE	1	1	0	0	1	1	0	1

A. FIRST INTERRUPT VECTOR BYTE, MCS80/85 MODE

IR	Interval = 4							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0
4	A7	A6	A5	1	0	0	0	0
3	A7	A6	A5	0	1	1	0	0
2	A7	A6	A5	0	1	0	0	0
1	A7	A6	A5	0	0	1	0	0
0	A7	A6	A5	0	0	0	0	0

IR	Interval = 8							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	1	1	1	0	0	0
6	A7	A6	1	1	0	0	0	0
5	A7	A6	1	0	1	0	0	0
4	A7	A6	1	0	0	0	0	0
3	A7	A6	0	1	1	0	0	0
2	A7	A6	0	1	0	0	0	0
1	A7	A6	0	0	1	0	0	0
0	A7	A6	0	0	0	0	0	0

B. SECOND INTERRUPT VECTOR BYTE, MCS80/85 MODE

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

C. THIRD INTERRUPT VECTOR BYTE, MCS80/85 MODE

Figure 9. 9A-C. Interrupt-Vector Bytes for 8259A, MCS 80/85 Mode

MCS-86/88™ Mode

When programmed in the MCS-86/88 mode, the 8259A should only be used within an MCS-86 or MCS-88 system. In this mode, the 8086/8088 will handle interrupts in the format described earlier in the "8259A—8086/8088 Overview".

Upon interrupt in the MCS-86/88 mode, the 8259A will output a single interrupt-vector byte to the data bus. This is in response to only two $\overline{\text{INTA}}$ pulses issued by the 8086/8088 after the 8259A has raised INT high.

The first $\overline{\text{INTA}}$ pulse is used only for set-up purposes internal to the 8259A. As in the MCS-80/85 mode, this set-up includes priority resolution and cascade mode operations which will be covered later. Unlike the MCS-80/85 mode, no CALL opcode is placed on the data bus.

The second $\overline{\text{INTA}}$ pulse is used to enable the single interrupt-vector byte onto the data bus. The 8086/8088 uses this interrupt-vector byte to select one of 256 interrupt "types" in 8086/8088 memory. Interrupt type selection for all eight IR levels is made when initially programming the 8259A. However, reference to only one interrupt type is needed for programming. The upper 5 bits of the interrupt vector byte are user definable. The lower 3 bits are automatically inserted by the 8259A depending upon the IR level.

Contents of the interrupt-vector byte for 8086/8088 type selection is put on the data bus during the second $\overline{\text{INTA}}$ pulse and is shown in Figure 10.

IR	D7	D6	D5	D4	D3	D2	D1	D0
7	T7	T6	T5	T4	T3	1	1	1
6	T7	T6	T5	T4	T3	1	1	0
5	T7	T6	T5	T4	T3	1	0	1
4	T7	T6	T5	T4	T3	1	0	0
3	T7	T6	T5	T4	T3	0	1	1
2	T7	T6	T5	T4	T3	0	1	0
1	T7	T6	T5	T4	T3	0	0	1
0	T7	T6	T5	T4	T3	0	0	0

Figure 10. Interrupt Vector Byte, MCS 86/88™ Mode

3.2 INTERRUPT PRIORITIES

A variety of modes and commands are available for controlling interrupt priorities of the 8259A. All of them are programmable, that is, they may be changed dynamically under software control. With these modes and commands, many possibilities are conceivable, giving the user enough versatility for almost any interrupt controlled application.

Fully Nested Mode

The fully nested mode of operation is a general purpose priority mode. This mode supports a multilevel-interrupt structure in which priority order of all eight IR inputs are arranged from highest to lowest.

Unless otherwise programmed, the fully nested mode is entered by default upon initialization. At this time, IR0 is assigned the highest priority through IR7 the lowest. The fully nested mode, however, is not confined to this IR structure alone. Once past initialization, other IR inputs can be assigned highest priority also, keeping the multilevel-interrupt structure of the fully nested mode. Figure 11A-C shows some variations of the priority structures in the fully nested mode.

IR LEVELS	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
PRIORITY	7	6	5	4	3	2	1	0

A

IR LEVELS	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
PRIORITY	4	3	2	1	0	7	6	5

B

IR LEVELS	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
PRIORITY	1	0	7	6	5	4	3	2

C

Figure 11. A-C. Some Variations of Priority Structure in the Fully Nested Mode

Further explanation of the fully nested mode, in this section, is linked with information of general 8259A interrupt operations. This is done to ease explanation to the user in both areas.

In general, when an interrupt is acknowledged, the highest priority request is determined from the IRR (Interrupt Request Register). The interrupt vector is then placed on the data bus. In addition, the corresponding bit in the ISR (In-Service Register) is set to designate the routine in service. This ISR bit remains set until an EOI (End-Of-Interrupt) command is issued to the 8259A. EOI's will be explained in greater detail shortly.

In the fully nested mode, while an ISR bit is set, all further requests of the same or lower priority are inhibited from generating an interrupt to the microprocessor. A higher priority request, though, can generate an interrupt, thus vectoring program execution to its service routine. Interrupts are only acknowledged, however, if the microprocessor has previously executed an "Enable Interrupts" instruction. This is because the interrupt request pin on the microprocessor gets disabled automatically after acknowledgement of any interrupt. The assembly language instructions used to enable interrupts are "EI" for 8080A/8085A and "STI" for 8086/8088. Interrupts can be disabled by using the instruction "DI" for 8080A/ 8085A and "CLI" for 8086/8088. When a routine is completed a "return" instruction is executed, "RET" for 8080A/8085A and "IRET" for 8086/8088.

Figure 12 illustrates the correct usage of interrupt related instructions and the interaction of interrupt levels in the fully nested mode.

Assuming the IR priority assignment for the example in Figure 12 is IR0 the highest through IR7 the lowest, the sequence is as follows. During the main program, IR3 makes a request. Since interrupts are enabled, the microprocessor is vectored to the IR3 service routine. During the IR3 routine, IR1 asserts a request. Since IR1 has higher priority than IR3, an interrupt is generated. However, it is not acknowledged because the microprocessor disabled interrupts in response to the IR3 interrupt. The IR1 interrupt is not acknowledged until the "Enable Interrupts" instruction is executed. Thus the IR3 routine has a "protected" section of code over which no interrupts (except non-maskable) are allowed. The IR1 routine has no such "protected" section since an "Enable Interrupts" instruction is the first one in its service routine. Note that in this example the IR1 request must stay high until it is acknowledged. This is covered in more depth in the "Interrupt Triggering" section.

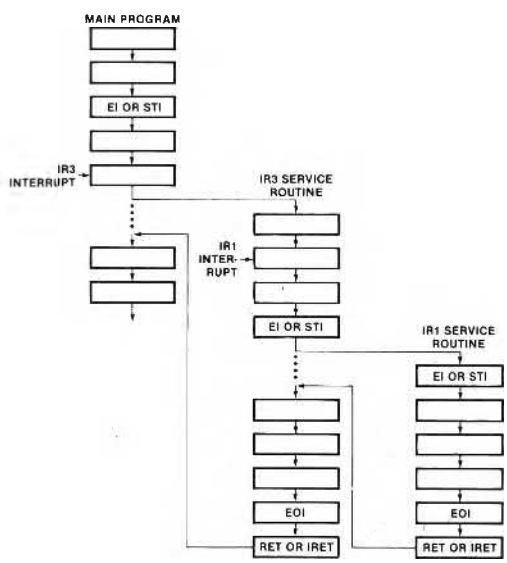


Figure 12. Fully Nested Mode Example (MCS 80/85™ or MCS 86/88™)

What is happening to the ISR register? While in the main program, no ISR bits are set since there aren't any interrupts in service. When the IR3 interrupt is acknowledged, the ISR3 bit is set. When the IR1 interrupt is acknowledged, both the ISR1 and the ISR3 bits are set, indicating that neither routine is complete. At this time, only IR0 could generate an interrupt since it is the only input with a higher priority than those previously in service. To terminate the IR1 routine, the routine must inform the 8259A that it is complete by resetting its ISR bit. It does this by executing an EOI command. A "return" instruction then transfers execution back to the IR3 routine. This allows IR0–IR2 to interrupt the IR3 routine again, since ISR3 is the highest ISR bit set. No further interrupts occur in the example so the EOI command resets ISR3 and the "return" instruction causes the main program to resume at its pre-interrupt location, ending the example.

A single 8259A is essentially always in the fully nested mode unless certain programming conditions disturb it. The following programming conditions can cause the 8259A to go out of the high to low priority structure of the fully nested mode.

- The automatic EOI mode
- The special mask mode
- A slave with a master not in the special fully nested mode

These modes will be covered in more detail later, however, they are mentioned now so the user can be aware of them. As long as these program conditions aren't inacted, the fully nested mode remains undisturbed.

End of Interrupt

Upon completion of an interrupt service routine the 8259A needs to be notified so its ISR can be updated. This is done to keep track of which interrupt levels are in the process of being serviced and their relative priorities. Three different End-Of-Interrupt (EOI) formats are available for the user. These are: the non-specific EOI command, the specific EOI command, and the automatic EOI Mode. Selection of which EOI to use is dependent upon the interrupt operations the user wishes to perform.

Non-Specific EOI Command

A non-specific EOI command sent from the microprocessor lets the 8259A know when a service routine has been completed, without specification of its exact interrupt level. The 8259A automatically determines the interrupt level and resets the correct bit in the ISR.

To take advantage of the non-specific EOI the 8259A must be in a mode of operation in which it can predetermine in-service routine levels. For this reason the non-specific EOI command should only be used when the most recent level acknowledged and serviced is always the highest priority level. When the 8259A receives a non-specific EOI command, it simply resets the highest priority ISR bit, thus confirming to the 8259A that the highest priority routine of the routines in service is finished.

The main advantage of using the non-specific EOI command is that IR level specification isn't necessary as in the "Specific EOI Command", covered shortly. However, special consideration should be taken when deciding to use the non-specific EOI. Here are two program conditions in which it is best not used:

- Using the set priority command within an interrupt service routine.
- Using a special mask mode.

These conditions are covered in more detail in their own sections, but are listed here for the users reference.

Specific EOI Command

A specific EOI command sent from the microprocessor lets the 8259A know when a service routine of a particular interrupt level is completed. Unlike a non-specific EOI command, which automatically resets the highest priority ISR bit, a specific EOI command specifies an exact ISR bit to be reset. One of the eight IR levels of the 8259A can be specified in the command.

The reason the specific EOI command is needed, is to reset the ISR bit of a completed service routine whenever the 8259A isn't able to automatically determine it. An example of this type of situation might be if the priorities of the interrupt levels were changed during an interrupt routine ("Specific Rotation"). In this case, if any other routines were in service at the same time, a non-specific EOI might reset the wrong ISR bit. Thus the specific EOI command is the best bet in this case, or for that matter, any time in which confusion of interrupt priorities may exist. The specific EOI command can be used in all conditions of 8259A operation, including those that prohibit non-specific EOI command usage.

Automatic EOI Mode

When programmed in the automatic EOI mode, the microprocessor no longer needs to issue a command to notify the 8259A it has completed an interrupt routine. The 8259A accomplishes this by performing a non-specific EOI automatically at the trailing edge of the last $\overline{\text{INTA}}$ pulse (third pulse in MCS-80/85, second in MCS-86).

The obvious advantage of the automatic EOI mode over the other EOI command is no command has to be issued. In general, this simplifies programming and lowers code requirements within interrupt routines.

However, special consideration should be taken when deciding to use the automatic EOI mode because it disturbs the fully nested mode. In the automatic EOI mode the ISR bit of a routine in service is reset right after it's acknowledged, thus leaving no designation in the ISR that a sevice routine is being executed. If any interrupt request occurs during this time (and interrupts are enabled) it will get serviced regardless of its priority, low or high. The problem of "over nesting" may also happen in this situation. "Over nesting" is when an IR input keeps interrupting its own routine, resulting in unnecessary stack pushes which could fill the stack in a worst case condition. This is not usually a desired form of operation!

So what good is the automatic EOI mode with problems like those just covered? Well, again, like the other EOIs, selection is dependent upon the application. If interrupts are controlled at a predetermined rate, so as not to cause the problems mentioned above, the automatic EOI mode works perfect just the way it is. However, if interrupts happen sporadically at an indeterminate rate, the automatic EOI mode should only be used under the following guideline:

- When using the automatic EOI mode with an indeterminate interrupt rate, the microprocessor should keep its interrupt request input disabled during execution of service routines.

By doing this, higher priority interrupt levels will be serviced only after the completion of a routine in service. This guideline restores the fully nested structure in regards to the IRR; however, a routine in-service can't be interrupted.

Automatic Rotation — Equal Priority

Automatic rotation of priorities serves in applications where the interrupting devices are of equal priority, such as communications channels. The concept is that once a peripheral is serviced, all other equal priority peripherals should be given a chance to be serviced before the original peripheral is serviced again. This is accomplished by automatically assigning a peripheral the lowest priority after being serviced Thus, in worst case, the device would have to wait until all other devices are serviced before being serviced again.

There are two methods of accomplishing automatic rotation. One is used in conjunction with the non-specific EOI, "rotate on non-specific EOI command". The other is used with the automatic EOI mode, "rotate in automatic EOI mode".

Rotate on Non-Specific EOI Command

When the rotate on non-specific EOI command is issued, the highest ISR bit is reset as in a normal non-specific EOI command. After it's reset though, the corresponding IR level is assigned lowest priority. Other IR priorities rotate to conform to the fully nested mode based on the newly assigned low priority

Figures 13A and B show how the rotate on non-specific EOI command effects the interrupt priorities. Let's assume the IR priorities were assigned with IR0 the highest and IR7 the lowest, as in 13A. IR6 and IR4 are already in service but neither is completed. Being the higher priority routine, IR4 is necessarily the routine being executed. During the IR4 routine a rotate on non-specific EOI command is executed. When this happens, bit 4 in the ISR is reset. IR4 then becomes the lowest priority and IR5 becomes the highest as in 13B.

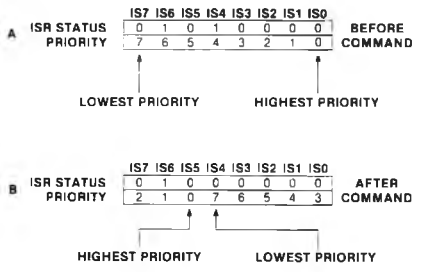


Figure 13. A-B. Rotate on Non-specific EOI Command Example

Rotate in Automatic EOI Mode

The rotate in automatic EOI mode works much like the rotate on non-specific EOI command. The main difference is that priority rotation is done automatically after

the last $\overline{INTA}$ pulse of an interrupt request. To enter or exit this mode a rotate-in-automatic-EOI set command and rotate-in-automatic-EOI clear command is provided. After that, no commands are needed as with the normal automatic EOI mode. However, it must be remembered, when using any form of the automatic EOI mode, special consideration should be taken. Thus, the guideline for the automatic EOI mode also stands for the rotate in automatic EOI mode.

Specific Rotation — Specific Priority

Specific rotation gives the user versatile capabilities in interrupt controlled operations. It serves in those applications in which a specific device's interrupt priority must be altered. As opposed to automatic rotation which automatically sets priorities, specific rotation is completely user controlled. That is, the user selects which interrupt level is to receive lowest or highest priority. This can be done during the main program or within interrupt routines. Two specific rotation commands are available to the user, the "set priority command" and the "rotate on specific EOI command."

Set Priority Command

The set priority command allows the programmer to assign an IR level the lowest priority. All other interrupt levels will conform to the fully nested mode based on the newly assigned low priority.

An example of how the set priority command works is shown in Figures 14A and 14B. These figures show the status of the ISR and the relative priorities of the interrupt levels before and after the set priority command. Two interrupt routines are shown to be in service in Figure 14A. Since IR2 is the highest priority, it is necessarily the routine being executed. During the IR2 routine, priorities are altered so that IR5 is the highest. This is done simply by issuing the set priority command to the 8259A. In this case, the command specifies IR4 as being the lowest priority. The result of this set priority command is shown in Figure 14B. Even though IR7 now has higher priority than IR2, it won't be acknowledged until the IR2 routine is finished (via EOI). This is because priorities are only resolved upon an interrupt request or an interrupt acknowledge sequence. If a higher priority request occurs during the IR2 routine, then priorities are resolved and the highest will be acknowledged.

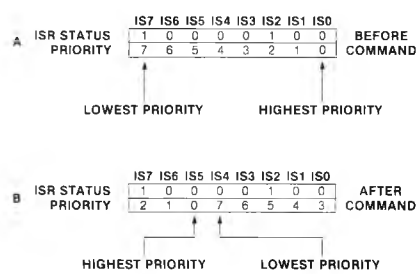


Figure 14. A-B. Set Priority Command Example

When completing a service routine in which the set priority command is used, the correct EOI must be issued. The non-specific EOI command shouldn't be used in the same routine as a set priority command. This is because the non-specific EOI command resets the highest ISR bit, which, when using the set priority command, is not always the most recent routine in service. The automatic EOI mode, on the other hand, can be used with the set priority command. This is because it automatically performs a non-specific EOI before the set priority command can be issued. The specific EOI command is the best bet in most cases when using the set priority command within a routine. By resetting the specific ISR bit of a routine being completed, confusion is eliminated.

Rotate on Specific EOI Command

The rotate on specific EOI command is literally a combination of the set priority command and the specific EOI command. Like the set priority command, a specified IR level is assigned lowest priority. Like the specific EOI command, a specified level will be reset in the ISR. Thus the rotate on specific EOI command accomplishes both tasks in only one command.

If it is not necessary to change IR priorities prior to the end of an interrupt routine, then this command is advantageous. For an EOI command must be executed anyway (unless in the automatic EOI mode), so why not do both at the same time?

Interrupt Masking

Disabling or enabling interrupts can be done by other means than just controlling the microprocessor's interrupt request pin. The 8259A has an IMR (Interrupt Mask Register) which enhances interrupt control capabilities. Rather than all interrupts being disabled or enabled at the same time, the IMR allows individual IR masking. The IMR is an 8-bit register, bits 0–7 directly correspond to IR0–IR7. Any IR input can be masked by writing to the IMR and setting the appropriate bit. Likewise, any IR input can be enabled by clearing the correct IMR bit.

There are various uses for masking off individual IR inputs. One example is when a portion of a main routine wishes only to be interrupted by specific interrupts. Another might be disabling higher priority interrupts for a portion of a lower priority service routine. The possibilities are many.

When an interrupt occurs while its IMR bit is set, it isn't necessarily forgotten. For, as stated earlier, the IMR acts only on the output of the IRR. Even with an IR input masked it is still possible to set the IRR. Thus, when resetting an IMR, if its IRR bit is set it will then generate an interrupt. This is providing, of course, that other priority factors are taken into consideration and the IR request remains active. If the IR request is removed before the IMR is reset, no interrupt will be acknowledged.

Special Mask Mode

In various cases, it may be desirable to enable interrupts of a lower priority than the routine in service. Or, in other words, allow lower priority devices to generate interrupts. However, in the fully nested mode, all IR levels of

priority below the routine in service are inhibited. So what can be done to enable them?

Well, one method could be using an EOI command before the actual completion of a routine in service. But beware, doing this may cause an "over nesting" problem, similar to in the automatic EOI mode. In addition, resetting an ISR bit is irreversible by software control, so lower priority IR levels could only be later disabled by setting the IMR.

A much better solution is the special mask mode. Working in conjunction with the IMR, the special mask mode enables interrupts from all levels except the level in service. This is done by masking the level that is in service and then issuing the special mask mode command. Once the special mask mode is set, it remains in effect until reset.

Figure 15 shows how to enable lower priority interrupts by using the Special Mask Mode (SMM). Assume that IR0 has highest priority when the main program is interrupted by IR4. In the IR4 service routine an enable interrupt instruction is executed. This only allows higher priority interrupt requests to interrupt IR4 in the normal fully nested mode. Further in the IR4 routine, bit 4 of the IMR is masked and the special mask mode is entered. Priority operation is no longer in the fully nested mode. All interrupt levels are enabled except for IR4. To leave the special mask mode, the sequence is executed in reverse.

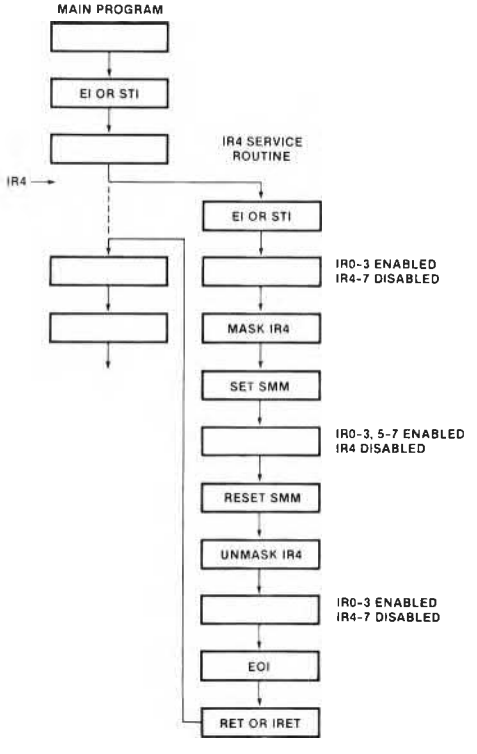


Figure 15. Special Mask Mode Example (MCS 80/85™ or MCS 86/88™)

Precautions must be taken when exiting an interrupt service routine which has used the special mask mode. A non-specific EOI command can't be used when in the special mask mode. This is because a non-specific won't clear an ISR bit of an interrupt which is masked when in the special mask mode. In fact, the bit will appear invisible. If the special mask mode is cleared before an EOI command is issued a non-specific EOI command can be used. This could be the case in the example shown in Figure 15, but, to avoid any confusion it's best to use the specific EOI whenever using the special mask mode.

It must be remembered that the special mask mode applies to all masked levels when set. Take, for instance, IR1 interrupting IR4 in the previous example. If this happened while in the special mask mode, and the IR1 routine masked itself, all interrupts would be enabled except IR1 and IR4 which are masked.

3.3 INTERRUPT TRIGGERING

There are two classical ways of sensing an active interrupt request: a level sensitive input or an edge sensitive input. The 8259A gives the user the capability for either method with the edge triggered mode and the level triggered mode. Selection of one of these interrupt triggering methods is done during the programmed initialization of the 8259A.

Level Triggered Mode

When in the level triggered mode the 8259A will recognize any active (high) level on an IR input as an interrupt request. If the IR input remains active after an EOI command has been issued (resetting its ISR bit), another interrupt will be generated. This is providing of course, the processor INT pin is enabled. Unless repetitious interrupt generation is desired, the IR input must be brought to an inactive state before an EOI command is issued in its service routine. However, it must not go inactive so soon that it disobeys the necessary timing requirements shown in Figure 16. Note that the request on the IR input must remain until after the falling edge of the first $\overline{\text{INTA}}$ pulse. If on any IR input, the request goes inactive before the first $\overline{\text{INTA}}$ pulse, the 8259A will respond as if IR7 was active. In any design in which there's a possibility of this happening, the IR7 default feature can be used as a safeguard. This can be accomplished by using the IR7 routine as a "clean-up routine" which might recheck the 8259A status or merely return program execution to its pre-interrupt location.

Depending upon the particular design and application, the level triggered mode has a number of uses. For one, it provides for repetitious interrupt generation. This is useful in cases when a service routine needs to be continually executed until the interrupt request goes inactive. Another possible advantage of the level triggered mode is it allows for "wire-OR'ed" interrupt requests. That is, a number of interrupt requests using the same IR input. This can't be done in the edge triggered mode, for if a device makes an interrupt request while the IR input is high (from another request), its transition will be "shadowed". Thus the 8259A won't recognize further interrupt requests because its IR input is already high. Note that when a "wire-OR'ed" scheme is used, the ac-

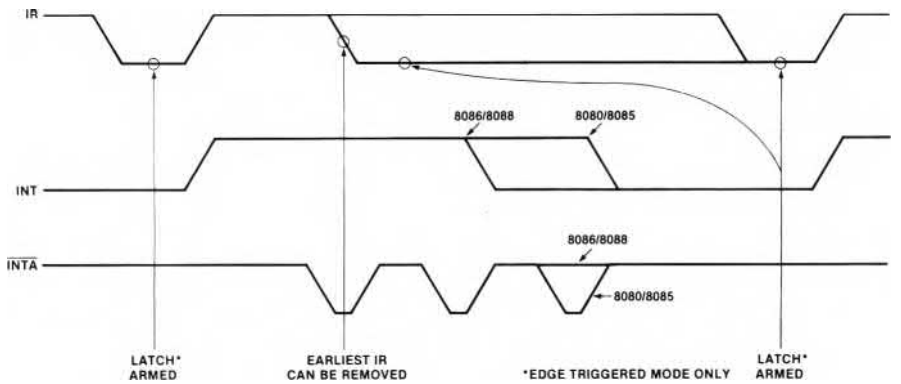


Figure 16. IR Triggering Timing Requirements

tual requesting device has to be determined by the software in the service routine.

Caution should be taken when using the automatic EOI mode and the level triggered mode together. Since in the automatic EOI mode an EOI is automatically performed at the end of the interrupt acknowledge sequence, if the processor enables interrupts while an IR input is still high, an interrupt will occur immediately. To avoid this situation interrupts should be kept disabled until the end of the service routine or until the IR input returns low.

Edge Triggered Mode

When in the edge triggered mode, the 8259A will only recognize interrupts if generated by an inactive (low) to active (high) transition on an IR input. The edge triggered mode incorporates an edge lockout method of operation. This means that after the rising edge of an interrupt request and the acknowledgement of the request, the positive level of the IR input won't generate further interrupts on this level. The user needn't worry about quickly removing the request after acknowledgement in fear of generating further interrupts as might be the case in the level triggered mode. Before another interrupt can be generated the IR input must return to the inactive state.

Referring back to Figure 16, the timing requirements for interrupt triggering is shown. Like the level triggered mode, in the edge triggered mode the request on the IR input must remain active until after the falling edge of the first $\overline{\text{INTA}}$ pulse for that particular interrupt. Unlike the level triggered mode, though, after the interrupt request is acknowledged its IRR latch is disarmed. Only after the IR input goes inactive will the IRR latch again become armed, making it ready to receive another interrupt request (in the level triggered mode, the IRR latch is always armed). Because of the way the edge triggered mode functions, it is best to use a positive level with a negative pulse to trigger the IR requests. With this type of input, the trailing edge of the pulse causes the interrupt and the maintained positive level meets the necessary timing requirements (remaining high until after the interrupt acknowledge occurs). Note that the IR7 default feature mentioned in the "level triggered mode" section also works for the edge triggered mode.

Depending upon the particular design and application, the edge triggered mode has various uses. Because of its edge lockout operation, it is best used in those applications where repetitious interrupt generation isn't desired. It is also very useful in systems where the interrupt request is a pulse (this should be in the form of a negative pulse to the 8259A). Another possible advantage is that it can be used with the automatic EOI mode without the cautions in the level triggered mode. Overall, in most cases, the edge triggered mode simplifies operation for the user, since the duration of the interrupt request at a positive level is not usually a factor.

3.4 INTERRUPT STATUS

By means of software control, the user can interrogate the status of the 8259A. This allows the reading of the internal interrupt registers, which may prove useful for interrupt control during service routines. It also provides for a modified status poll method of device monitoring, by using the poll command. This makes the status of the internal IR inputs available to the user via software control. The poll command offers an alternative to the interrupt vector method, especially for those cases when more than 64 interrupts are needed.

Reading Interrupt Registers

The contents of each 8-bit interrupt register, IRR, ISR, and IMR, can be read to update the user's program on the present status of the 8259A. This can be a versatile tool in the decision making process of a service routine, giving the user more control over interrupt operations. Before delving into the actual process of reading the registers, let's briefly review their general descriptions:

IRR (Interrupt Request Register)	Specifies all interrupt levels requesting service.
ISR (In-Service Register)	Specifies all interrupt levels which are being serviced.
IMR (Interrupt Mask Register)	Specifies all interrupt levels that are masked.

To read the contents of the IRR or ISR, the user must first issue the appropriate read register command (read IRR or read ISR) to the 8259A. Then by applying a $\overline{RD}$ pulse to the 8259A (an INput instruction), the contents of the desired register can be acquired. There is no need to issue a read register command every time the IRR or ISR is to be read. Once a read register command is received by the 8259A, it "remembers" which register has been selected. Thus, all that is necessary to read the contents of the same register more than once is the $\overline{RD}$ pulse and the correct addressing (A0 = 0, explained in "Programming the 8259A"). Upon initialization, the selection of registers defaults to the IRR. Some caution should be taken when using the read register command in a system that supports several levels of interrupts. If the higher priority routine causes an interrupt between the read register command and the actual input of the register contents, there's no guarantee that the same register will be selected when it returns. Thus it is best in such cases to disable interrupts during the operation.

Reading the contents of the IMR is different than reading the IRR or ISR. A read register command is not necessary when reading the IMR. This is because the IMR can be addressed directly for both reading and writing. Thus all that the 8259A requires for reading the IMR is a $\overline{RD}$ pulse and the correct addressing (A0 = 1, explained in "Programming the 8259A").

Poll Command

As mentioned towards the beginning of this application note, there are two methods of servicing peripherals: status polling and interrupt servicing. For most applications the interrupt service method is best. This is because it requires the least amount of CPU time, thus increasing system throughput. However, for certain applications, the status poll method may be desirable.

For this reason, the 8259A supports polling operations with the poll command. As opposed to the conventional method of polling, the poll command offers improved device servicing and increased throughput. Rather than having the processor poll each peripheral in order to find the actual device requiring service, the processor polls the 8259A. This allows the use of all the previously mentioned priority modes and commands. Additionally, both polled and interrupt methods can be used within the same program.

To use the poll command the processor must first have its interrupt request pin disabled. Once the poll command is issued, the 8259A will treat the next ($\overline{CS}$ qualified) $\overline{RD}$ pulse issued to it (an INput instruction) as an interrupt acknowledge. It will then set the appropriate bit in the ISR, if there was an interrupt request, and enable a special word onto the data bus. This word shows whether an interrupt request has occurred and the highest priority level requesting service. Figure 17 shows the contents of the "poll word" which is read by the processor. Bits W0-W2 convey the binary code of the highest priority level requesting service. Bit I designates whether or not an interrupt request is present. If an interrupt request is present, bit I will equal 1. If there isn't an interrupt request at all, bit I will equal 0 and bits W0-W2 will be set to ones. Service to the requesting device is achieved by software decoding the poll word and branching to the appropriate service routine. Each time the 8259A is to be polled, the poll command must be written before reading the poll word.

The poll command is useful in various situations. For instance, it's a good alternative when memory is very limited, because an interrupt-vector table isn't needed. Another use for the poll command is when more than 64 interrupt levels are needed (64 is the limit when cascading 8259's). The only limit of interrupts using the poll command is the number of 8259's that can be addressed in a particular system. Still another application of the poll command might be when the INT or $\overline{INTA}$ signals are not available. This might be the case in a large system where a processor on one card needs to use an 8259A on a different card. In this instance, the poll command is the only way to monitor the interrupt devices and still take advantage of the 8259A's prioritizing features. For those cases when the 8259A is using the poll command only and not the interrupt method, each 8259A must receive an initialization sequence (interrupt vector). This must be done even though the interrupt vector features of the 8259A are not used. In this case, the interrupt vector specified in the initialization sequence could be a "fake".

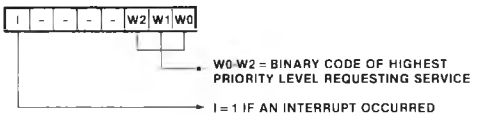


Figure 17. Poll Word

3.5 INTERRUPT CASCADING

As mentioned earlier, more than one 8259A can be used to expand the priority interrupt scheme to up to 64 levels without additional hardware. This method for expanded interrupt capability is called "cascading". The 8259A supports cascading operations with the cascade mode. Additionally, the special fully nested mode and the buffered mode are available for increased flexibility when cascading 8259A's in certain applications.

Cascade Mode

When programmed in the cascade mode, basic operation consists of one 8259A acting as a master to the others which are serving as slaves. Figure 18 shows a system containing a master and two slaves, providing a total of 22 interrupt levels.

A specific hardware set-up is required to establish operation in the cascade mode. With Figure 18 as a reference, note that the master is designated by a high on the $\overline{SP}/\overline{EN}$ pin, while the $\overline{SP}/\overline{EN}$ pins of the slaves are grounded (this can also be done by software, see buffered mode). Additionally, the INT output pin of each slave is connected to an IR input pin of the master. The CAS0-2 pins for all 8259A's are paralleled. These pins act as outputs when the 8259A is a master and as inputs for the slaves. Serving as a private 8259A bus, they control which slave has control of the system bus for interrupt vectoring operation with the processor. All other pins are connected as in normal operation (each 8259A receives an INTA pulse).

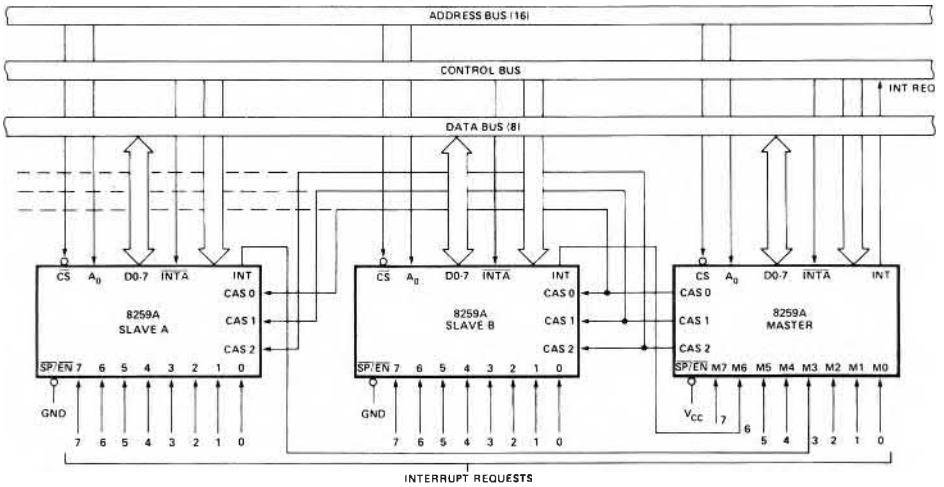


Figure 18. Cascaded 8259A'S 22 Interrupt Levels

Besides hardware set-up requirements, all 8259A's must be software programmed to work in the cascade mode. Programming the cascade mode is done during the initialization of each 8259A. The 8259A that is selected as master must receive specification during its initialization as to which of its IR inputs are connected to a slave's INT pin. Each slave 8259A, on the other hand, must be designated during its initialization with an ID (0 through 7) corresponding to which of the master's IR inputs its INT pin is connected to. This is all necessary so the CAS0-2 pins of the masters will be able to address each individual slave. Note that as in normal operation, each 8259A must also be initialized to give its IR inputs a unique interrupt vector. More detail on the necessary programming of the cascade mode is explained in "Programming the 8259A".

Now, with background information on both hardware and software for the cascade mode, let's go over the sequence of events that occur during a valid interrupt request from a slave. Suppose a slave IR input has received an interrupt request. Assuming this request is higher priority than other requests and in-service levels on the slave, the slave's INT pin is driven high. This signals the master of the request by causing an interrupt request on a designated IR pin of the master. Again, assuming that this request to the master is higher priority than other master requests and in-service levels (possibly from other slaves), the master's INT pin is pulled high, interrupting the processor.

The interrupt acknowledge sequence appears to the processor the same as the non-cascading interrupt acknowledge sequence; however, it's different among the 8259A's. The first $\overline{\text{INTA}}$ pulse is used by all the 8259A's for internal set-up purposes and, if in the 8080/8085 mode, the master will place the CALL opcode on the data bus. The first $\overline{\text{INTA}}$ pulse also signals the master to place the requesting slave's ID code on the CAS lines. This turns control over to the slave for the rest of the interrupt acknowledge sequence, placing the appropriate pre-programmed interrupt vector on the data bus, completing the interrupt request.

During the interrupt acknowledge sequence, the corresponding ISR bit of both the master and the slave get set. This means two EOI commands must be issued (if not in the automatic EOI mode), one for the master and one for the slave.

Special consideration should be taken when mixed interrupt requests are assigned to a master 8259A; that is, when some of the master's IR inputs are used for slave interrupt requests and some are used for individual interrupt requests. In this type of structure, the master's IR0 must not be used for a slave. This is because when an IR input that isn't initialized as a slave receives an interrupt request, the CAS0-2 lines won't be activated, thus staying in the default condition addressing for IR0 (slave IR0). If a slave is connected to the master's IR0 when a non-slave interrupt occurs on another master IR input, erroneous conditions may result. Thus IR0 should be the last choice when assigning slaves to IR inputs.

Special Fully Nested Mode

Depending on the application, changes in the nested structure of the cascade mode may be desired. This is because the nested structure of a slave 8259A differs from that of the normal fully nested mode. In the cascade mode, if a slave receives a higher priority interrupt request than one which is in service (through the same slave), it won't be recognized by the master. This is because the master's ISR bit is set, ignoring all requests of equal or lower priority. Thus, in this case, the higher priority slave interrupt won't be serviced until after the master's ISR bit is reset by an EOI command. This is most likely after the completion of the lower priority routine.

If the user wishes to have a truly fully nested structure within a slave 8259A, the special fully nested mode should be used. The special fully nested mode is pro-

grammed in the master only. This is done during the master's initialization. In this mode the master will ignore only those interrupt requests of lower priority than the set ISR bit and will respond to all requests of equal or higher priority. Thus if a slave receives a higher priority request than one in service, it will be recognized. To insure proper interrupt operation when using the special fully nested mode, the software must determine if any other slave interrupts are still in service before issuing an EOI command to the master. This is done by resetting the appropriate slave ISR bit with an EOI and then reading its ISR. If the ISR contains all zeros, there aren't any other interrupts from the slave in service and an EOI command can be sent to the master. If the ISR isn't all zeros, an EOI command shouldn't be sent to the master. Clearing the master's ISR bit with an EOI command while there are still slave interrupts in service would allow lower priority interrupts to be recognized at the master. An example of this process is shown in the second application in the "Applications Examples" section.

Buffered Mode

The buffered mode is useful in large systems where buffering is required on the data bus. Although not limited to only 8259A cascading, it's most pertinent in this use. In the buffered mode, whenever the 8259A's data bus output is enabled, its $\overline{SP}/\overline{EN}$ pin will go low. This signal can be used to enable data transfer through a buffer transceiver in the required direction.

Figure 19 shows a conceptual diagram of three 8259A's in cascade, each slave is controlling an individual 8286 8-bit bidirectional bus driver by means of the buffered mode. Note the pull-up on the $\overline{SP}/\overline{EN}$. It is used to enable data transfer to the 8259A for its initial programming. When data transfer is to go from the 8259A to the processor, $\overline{SP}/\overline{EN}$ will go low; otherwise, it will be high.

A question should arise, however, from the fact that the $\overline{SP}/\overline{EN}$ pin is used to designate a master from a slave; how can it be used for both master-slave selection and buffer control? The answer to this is the provision for software programmable master-slave selection when in the buffer mode. The buffered mode is selected during each 8259A's initialization. At the same time, the user can assign each individual 8259A as a master or slave (see "Programming the 8259A").

4. PROGRAMMING THE 8259A

Programming the 8259A is accomplished by using two types of command words: Initialization Command Words (ICWs) and Operational Command Words (OCWs). All the modes and commands explained in the previous section, "Operation of the 8259A", are programmable using the ICWs and OCWs (see Appendix A for cross reference). The ICWs are issued from the processor in a sequential format and are used to set-up the 8259A in an initial state of operation. The OCWs are issued as needed to vary and control 8259A operation.

Both ICWs and OCWs are sent by the processor to the 8259A via the data bus (8259A $\overline{CS} = 0$, $\overline{WR} = 0$). The 8259A distinguishes between the different ICWs and OCWs by the state of its A0 pin (controlled by processor addressing), the sequence they're issued in (ICWs only), and some dedicated bits among the ICWs and OCWs. Those bits which are dedicated are indicated so by fixed values (0 or 1) in the corresponding ICW or OCW programming formats which are covered shortly. Note, when issuing either ICWs or OCWs, the interrupt request pin of the processor should be disabled.

4.1 INITIALIZATION COMMAND WORDS (ICWs)

Before normal operation can begin, each 8259A in a system must be initialized by a sequence of two to four programming bytes called ICWs (Initialization Command Words). The ICWs are used to set-up the necessary conditions and modes for proper 8259A operation.

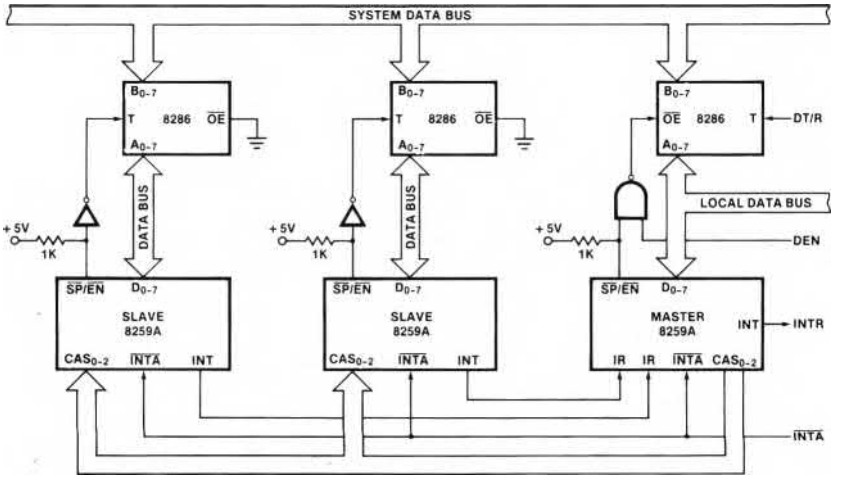


Figure 19. Cascade-Buffered Mode Example

Figure 20 shows the initialization flow of the 8259A. Both ICW1 and ICW2 must be issued for any form of 8259A operation. However, ICW3 and ICW4 are used only if designated so in ICW1. Determining the necessity and use of each ICW is covered shortly in individual groupings. Note that, once intialized, if any programming changes within the ICWs are to be made, the entire ICW sequence must be reprogrammed, not just an individual ICW.

Certain internal set-up conditions occur automatically within the 8259A after the first ICW has been issued. These are:

A. Sequencer logic is set to accept the remain ng ICWs as designated in ICW1.

B. The ISR (In-Service Register) and IMR (Interrupt Mask Register) are both cleared.

C. The special mask mode is reset.

D. The rotate in automatic EOI mode flip-flop is cleared.

E. The IRR (Interrupt Request Register) is selected for the read register command.

F. If the IC4 bit equals 0 in ICW1, all functions in ICW4 are cleared; 8080/8085 mode is selected by default.

G. The fully nested mode is entered with an initial priority assignment of IR0 highest through IR7 lowest.

H. The edge sense latch of each IR priority cell is cleared, thus requiring a low to high transition to generate an interrupt (edge triggered mode effected only).

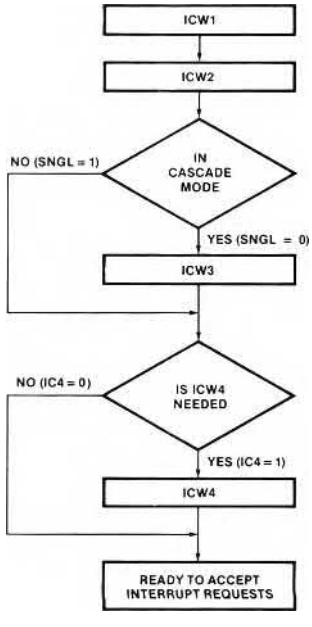
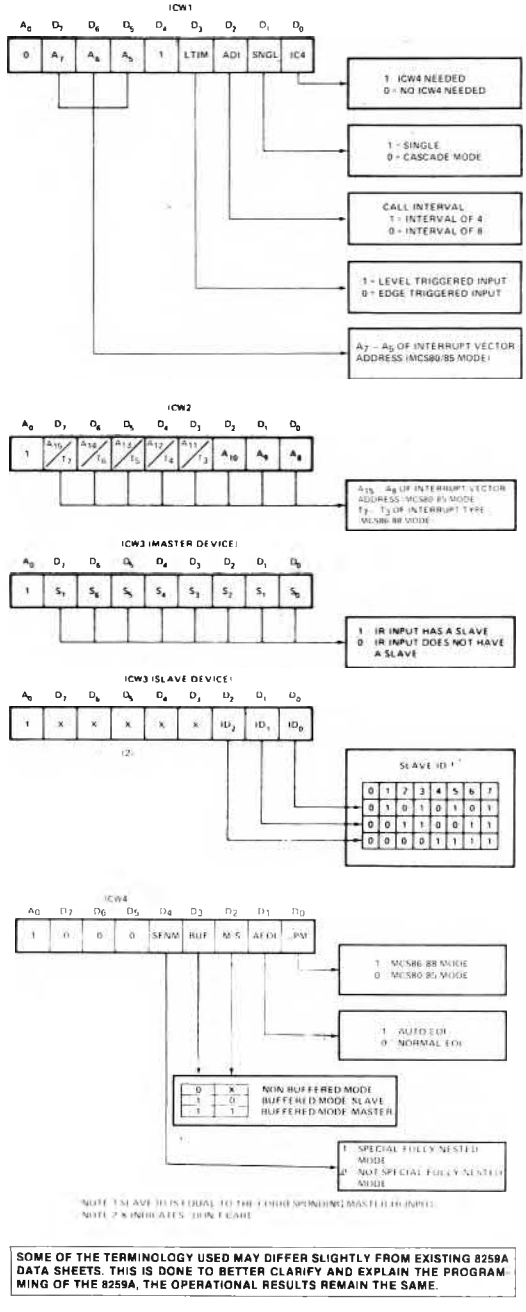


Figure 20. Initialization Flow

The ICW programming format, Figure 21, shows bit designation and a short definition of each ICW. With the ICW format as reference, the functions of each ICW will now be explained individually.



SOME OF THE TERMINOLOGY USED MAY DIFFER SLIGHTLY FROM EXISTING 8259A DATA SHEETS. THIS IS DONE TO BETTER CLARIFY AND EXPLAIN THE PROGRAMMING OF THE 8259A, THE OPERATIONAL RESULTS REMAIN THE SAME.

Figure 21. Initialization Command Words (ICWS) Programming Format

ICW1 and ICW2

Issuing ICW1 and ICW2 is the minimum amount of programming needed for any type of 8259A operation. The majority of bits within these two ICWs are used to designate the interrupt vector starting address. The remaining bits serve various purposes. Description of the ICW1 and ICW2 bits is as follows:

IC4: The IC4 bit is used to designate to the 8259A whether or not ICW4 will be issued. If any of the ICW4 operations are to be used, ICW4 must equal 1. If they aren't used, then ICW4 needn't be issued and IC4 can equal 0. Note that if IC4 = 0, the 8259A will assume operation in the MCS-80/85 mode.

SNGL: The SNGL bit is used to designate whether or not the 8259A is to be used alone or in the cascade mode. If the cascade mode is desired, SNGL must equal 0. In doing this, the 8259A will accept ICW3 for further cascade mode programming. If the 8259A is to be used as the single 8259A within a system, the SNGL bit must equal 1; ICW3 won't be accepted.

ADI: The ADI bit is used to specify the address interval for the MCS-80/85 mode. If a 4-byte address interval is to be used, ADI must equal 1. For an 8-byte address interval, ADI must equal 0. The state of ADI is ignored when the 8259A is in the MCS-86/88 mode.

LTIM: The LTIM bit is used to select between the two IR input triggering modes. If LTIM = 1, the level triggered mode is selected. If LTIM = 0, the edge triggered mode is selected.

A5-A15: The A5-A15 bits are used to select the interrupt vector address when in the MCS-80/85 mode. There are two programming formats that can be used to do this. Which one is implemented depends upon the selected address interval (ADI). If ADI is set for the 4-byte interval, then the 8259A will automatically insert A0-A4 (A0, A1 = 0 and A2, A3, A4 = IR0-7). Thus A5-A15 must be user selected by programming the A5-A15 bits with the desired address. If ADI is set for the 8-byte interval, then A0-A5 are automatically inserted (A0, A1, A2 = 0 and A3, A4, A5 = IR0-7). This leaves A6-A15 to be selected by programming the A6-A15 bits with the desired address. The state of bit 5 is ignored in the latter format.

T3-T7: The T3-T7 bits are used to select the interrupt type when the MCS-86/88 mode is used. The programming of T3-T7 selects the upper 5 bits. The lower 3 bits are automatically inserted, corresponding to the IR level causing the interrupt. The state of bits A5-A10 will be ignored when in the MCS-86/88 mode. Establishing the actual memory address of the interrupt is shown in Figure 22.

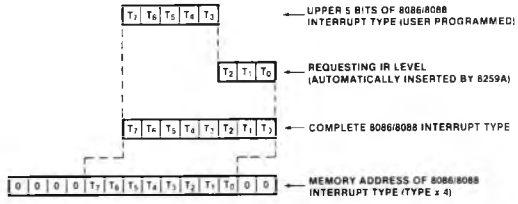


Figure 22. Establishing Memory Address of 8086/8088 Interrupt Type

ICW3

The 8259A will only accept ICW3 if programmed in the cascade mode (ICW1, SNGL = 0). ICW3 is used for specific programming within the cascade mode. Bit definition of ICW3 differs depending on whether the 8259A is a master or a slave. Definition of the ICW3 bits is as follows:

S0-7 (Master): If the 8259A is a master (either when the $\overline{SP}/\overline{EN}$ pin is tied high or in the buffered mode when M/S = 1 in ICW4), ICW3 bit definition is S0-7, corresponding to "slave 0-7". These bits are used to establish which IR inputs have slaves connected to them. A 1 designates a slave, a 0 no slave. For example, if a slave was connected to IR3, the S3 bit should be set to a 1. (S0) should be last choice for slave designation.

ID0-ID2 (Slave): If the 8259A is a slave (either when the $\overline{SP}/\overline{EN}$ pin is low or in the buffered mode when M/S = 0 in ICW4), ICW3 bit definition is used to establish its individual identity. The ID code of a particular slave must correspond to the number of the masters IR input it is connected to. For example, if a slave was connected to IR6 of the master, the slaves ID0-2 bits should be set to ID0 = 0, ID1 = 1, and ID2 = 1.

ICW4

The 8259A will only accept ICW4 if it was selected in ICW1 (bit IC4 = 1). Various modes are offered by using ICW4. Bit definition of ICW4 is as follows:

μPM: The μPM bit allows for selection of either the MCS-80/85 or MCS-86/88 mode. If set as a 1 the MCS-86/88 mode is selected, if a 0, the MCS-80/85 mode is selected.

AEOI: The AEOI bit is used to select the automatic end of interrupt mode. If AEOI = 1, the automatic end of interrupt mode is selected. If AEOI = 0, it isn't selected; thus an EOI command must be used during a service routine.

M/S: The M/S bit is used in conjunction with the buffered mode. If in the buffered mode, M/S defines whether the 8259A is a master or a slave. When M/S is set to a 1, the 8259A operates as the master; when M/S is 0, it operates as a slave. If not programmed in the buffered mode, the state of the M/S bit is ignored.

BUF: The BUF bit is used to designate operation in the buffered mode, thus controlling the use of the $\overline{SP}/\overline{EN}$ pin. If BUF is set to a 1, the buffered mode is programmed and $\overline{SP}/\overline{EN}$ is used as a transceiver enable output. If BUF is 0, the buffered mode isn't programmed and $\overline{SP}/\overline{EN}$ is used for master/slave selection. Note if ICW4 isn't programmed, $\overline{SP}/\overline{EN}$ is used for master/slave selection.

SFNM: The SFNM bit designates selection of the special fully nested mode which is used in conjunction with the cascade mode. Only the master should be programmed in the special fully nested mode to assure a truly fully nested structure among the slave IR inputs. If SFNM is set to a 1, the special fully nested mode is selected; if SFNM is 0, it is not selected.

4.2 OPERATIONAL COMMAND WORD (OCWs)

Once initialized by the ICWs, the 8259A will most likely be operating in the fully nested mode. At this point, operation can be further controlled or modified by the use of OCWs (Operation Command Words). Three OCWs are available for programming various modes and commands. Unlike the ICWs, the OCWs needn't be in any type of sequential order. Rather, they are issued by the processor as needed within a program.

Figure 23, the OCW programming format, shows the bit designation and short definition of each OCW. With the OCW format as reference, the functions of each OCW will be explained individually.

OCW1

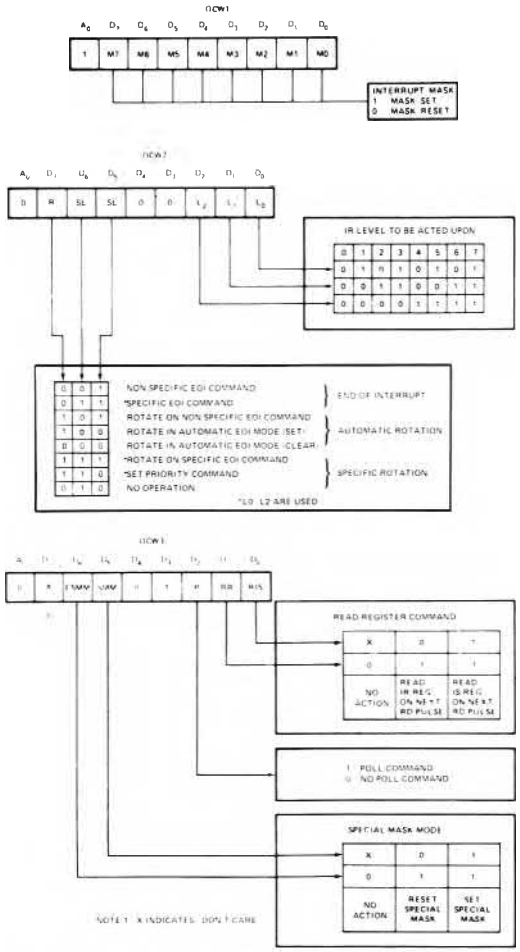
OCW1 is used solely for 8259A masking operations. It provides a direct link to the IMR (Interrupt Mask Register). The processor can write to or read from the IMR via OCW1. The OCW1 bit definition is as follows:

M0-M7: The M0-M7 bits are used to control the masking of IR inputs. If an M bit is set to a 1, it will mask the corresponding IR input. A 0 clears the mask, thus enabling the IR input. These bits convey the same meaning when being read by the processor for status update.

OCW2

OCW2 is used for end of interrupt, automatic rotation, and specific rotation operations. Associated commands and modes of these operations (with the exception of AEOI initialization), are selected using the bits of OCW2 in a combined fashion. Selection of a command or mode should be made with the corresponding table for OCW2 in the OCW programming format (Figure 20), rather than on a bit by bit basis. However, for completeness of explanation, bit definition of OCW2 is as follows:

L0-L2: The L0-L2 bits are used to designate an interrupt level (0-7) to be acted upon for the operation selected by the EOI, SL, and R bits of OCW2. The level designated will either be used to reset a specific ISR bit or to set a specific priority. The L0-L2 bits are enabled or disabled by the SL bit.



SOME OF THE TERMINOLOGY USED MAY DIFFER SLIGHTLY FROM EXISTING 8259A DATA SHEETS. THIS IS DONE TO BETTER CLARIFY AND EXPLAIN THE PROGRAMMING OF THE 8259A. THE OPERATIONAL RESULTS REMAIN THE SAME.

Figure 23. Operational Command Words (OCWs) Programming Format

EOI: The EOI bit is used for all end of interrupt commands (not automatic end of interrupt mode). If set to a 1, a form of an end of interrupt command will be executed depending on the state of the SL and R bits. If EOI is 0, an end of interrupt command won't be executed.

SL: The SL bit is used to select a specific level for a given operation. If SL is set to a 1, the L0-L2 bits are enabled. The operation selected by the EOI and R bits will be executed on the specified interrupt level. If SL is 0, the L0-L2 bits are disabled.

R: The R bit is used to control all 8259A rotation operations. If the R bit is set to a 1, a form of priority rotation will be executed depending on the state of SL and EOI bits. If R is 0, rotation won't be executed.

OCW3

OCW3 is used to issue various modes and commands to the 8259A. There are two main categories of operation associated with OCW3: interrupt status and interrupt masking. Bit definition of OCW3 is as follows:

RIS: The RIS bit is used to select the ISR or IRR for the read register command. If RIS is set to 1, ISR is selected. If RIS is 0, IRR is selected. The state of the RIS is only honored if the RR bit is a 1.

RR: The RR bit is used to execute the read register command. If RR is set to a 1, the read register command is issued and the state of RIS determines the register to be read. If RR is 0, the read register command isn't issued.

P: The P bit is used to issue the poll command. If P is set to a 1, the poll command is issued. If it is 0, the poll command isn't issued. The poll command will override a read register command if set simultaneously.

SMM: The SMM bit is used to set the special mask mode. If SMM is set to a 1, the special mask mode is selected. If it is 0, it is not selected. The state of the SMM bit is only honored if it is enabled by the ESMM bit.

ESMM: The ESMM bit is used to enable or disable the effect of the SMM bit. If ESMM is set to a 1, SMM is enabled. If ESMM is 0, SMM is disabled. This bit is useful to prevent interference of mode and command selections in OCW3.

5. APPLICATION EXAMPLES

In this section, the 8259A is shown in three different application examples. The first is an actual design implementation supporting an 8080A microprocessor system, "Power Fail/Auto Start with Battery Back-Up RAM". The second is a conceptual example of incorporating more than 64 interrupt levels in an 8080A or 8085A system, "78 Level Interrupt System". The third application is a conceptual design using an 8086 system, "Timer Controlled Interrupts". Although specific microprocessor systems are used in each example, these applications can be applied to either MCS-80, MCS-85, MCS-86, or MCS-88 systems, providing the necessary hardware and software changes are made. Overall, these applications should serve as a useful guide, illustrating the various procedures in using the 8259A.

5.1 POWER FAIL/AUTO-START WITH BATTERY BACK-UP RAM

The first application illustrates the 8259A used in an 8080A system, supporting a battery back-up scheme for the RAM (Random Access Memory) in a microcomputer system. Such a scheme is important in numerical and process control applications. The entire microcomputer system could be supported by a battery back-up scheme, however, due to the large amount of current usually required and the fact that most machinery is not supported by an auxiliary power source, only the state of calculations and variables usually need to be saved. In the event of a loss of power, if these items are not already stored in RAM, they can be transferred there and saved using a simple battery back-up system.

The vehicle used in this application is the Intel® SBC-80/20 Single Board Computer. An 8259A is used in the SBC-80/20 along with control lines helpful in implementing the power-down and automatic restart sequence used in a battery back-up system. The SBC-80/20 also contains user-selectable jumpers which allow the on-board RAM to be powered by a supply separate from the supply used for the non-RAM components. Also, the output of an undedicated latch is available to be connected to the IR inputs of the 8259A (the latch is cleared via an output port). In addition, an undedicated, buffered input line is provided, along with an input to the RAM decoder that will protect memory when asserted.

The additional circuitry to be described was constructed on an SBC-905 prototyping board. An SBC-635 power supply was used to power the non-RAM section of the SBC-80/20 while an external DC supply was used to simulate the back-up battery supplying power to the RAM. The SBC-635 was used since it provides an open collector ACLO output which indicates that the AC input line voltage is below 103/206 VAC (RMS).

The following is an example of a power-down and restart sequence that introduces the various power fail signals.

1. An AC power failure occurs and the ACLO goes high (ACLO is pulled up by the battery supply). This indicates that DC power will be reliable for at most 7.5 ms. The power fail circutry generates a Power Fail Interrupt ($\overline{PFI}$) signal. This signal sets the $\overline{PFI}$ latch, which is connected to the IR0 input of the 8259A, and sets the Power Fail Sense (PFS) latch. The state of this latch will indicate to the processor, upon reset, whether it is coming up from a power failure (warm start) or if it is coming up initially (cold start).
2. The processor is interrupted by the 8259A when the PFI latch is set. This pushes the pre-power-down program counter onto the stack and calls the service routine for the IR0 input. The IR0 service routine saves the processor status and any other needed variables. The routine should end with a HALT instruction to minimize bus transitions.
3. After a predetermined length of time (5 ms in this example) the power fail circuitry generates a Memory Protect ($\overline{MPRO}$) signal. All processing for the power failure (including the interrupt response delays) must be completed within this 5 ms window. The $\overline{MPRO}$ signal ensures that spurious transitions on the system control bus caused by power going down do not alter the contents of the RAM.
4. DC power goes down.
5. AC power returns. The power-on reset circuitry on the SBC-80/20 generates a system RESET.
6. The processor reads the state of the $\overline{PFS}$ line to determine the appropriate start-up sequence. The PFS latch is cleared, the MPRO signal is removed, and the PFI latch driving IR0 is cleared by the Power Fail Sense Reset ($\overline{PFSR}$) signal. The system then continues from the pre-power-down location for a warm start by restoring the processor status and popping the pre-power-down program counter off the stack.

Figure 24 illustrates this timing.

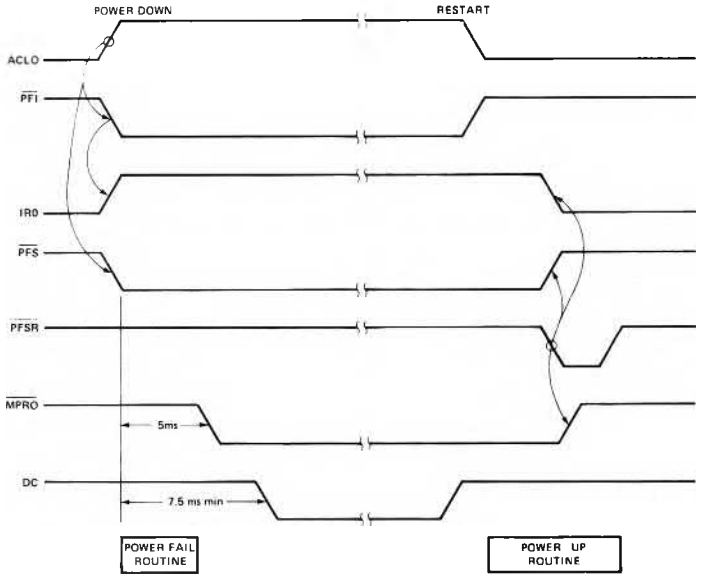


Figure 24. Power Down Restart Timing

Figure 25 shows the block diagram for the system. Notice that the RAM, the RAM decoder, and the power-down circuitry are powered by the battery supply.

The schematic of the power-down circuitry and the SBC-80/20 interface is shown in Figure 26. The design is very straightforward and uses CMOS logic to minimize the battery current requirements. The cold start switch is necessary to ensure that during a cold start, the $\overline{PFS}$ line is indicating "cold start" sense ($\overline{PFS}$ high). Thus, for a cold start, the cold start switch is depressed during power on. After that, no further action is needed. Notice that the $\overline{PFI}$ signal sets the on-board PFI latch. The output of this latch drives the 8259A IR0 input. This latch is cleared during the restart routine by executing an OUTput D4H instruction. The state of the $\overline{PFS}$ line may be read on the least significant data bus line (DB0) by executing an INput D4H instruction. An 8255 port (8255 #1, port C, bit 0) is used to control the $\overline{PFSR}$ line.

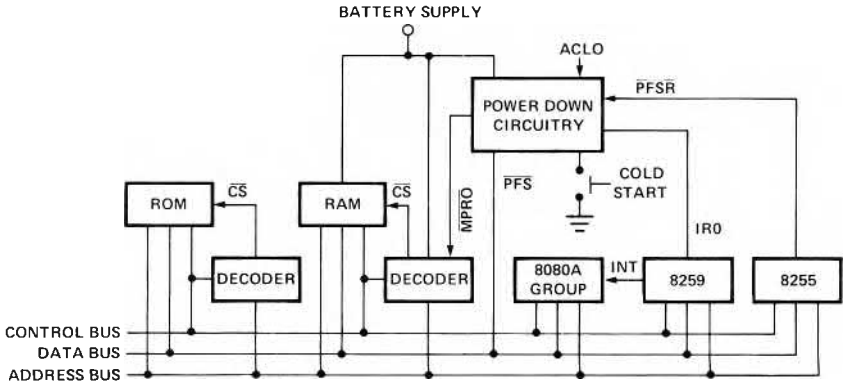


Figure 25. Block Diagram of SBC 80/20 with Power Down Circuit

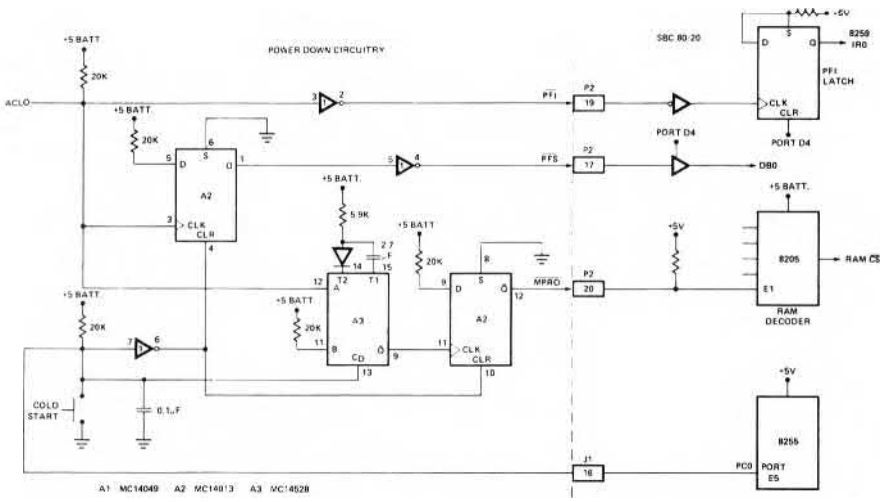


Figure 26. Power Down Circuit - SBC 80/20 Interface

The fully nested mode for the 8259A is used in its initial state to ensure the IR0 always has the highest priority. The remaining IR inputs can be used for any other purpose in the system. The only constraint is that the service routines must enable interrupts as early as possible. Obviously, this is to ensure that the power-down interrupt does not have to wait for service. If a rotating priority scheme is desired, another 8259A could be added as a slave and be programmed to operate in a rotating mode. The master would remain in the initial state of the fully nested mode so that the IR0 still remains the highest priority input.

The software to support the power-down circuitry is shown in Figure 27. The flow for each label will be discussed.

After any system reset, the processor starts execution at location 0000H (START). The $\overline{\text{PFS}}$ status is read and execution is transferred to CSTART if $\overline{\text{PFS}}$ indicates a cold start (i.e., someone is depressing the cold start switch) or WSTART if a warm start is indicated ($\overline{\text{PFS}}$ LOW). CSTART is the start of the user's program. The Stack Pointers (SP) and device initialization were included just to remind the reader that these must occur. The first EI instruction must appear after the 8259A has received its initialization sequence. The 8259A (and other devices) are initialized in the INIT subroutine.

When a power failure occurs, execution is vectored by the 8259A to REGSAV by way of the jump table at JSTART. The pre-power-down program counter is placed on the stack. REGSAV saves the processor registers and flags in the usual manner by pushing them onto the stack. Other items, such as output port status, programmable peripheral states, etc., are pushed onto the stack at this time. The Stack Pointer (SP) could be pushed onto the stack by way of the register pair HL but the top of the stack can exist anywhere in memory and there is no way then of knowing where that is when in the power-up routine. Thus, the SP is saved at a dedicated location in RAM. It isn't really necessary to send an EOI command to the 8259A in REGSAV since power will be removed from the 8259A, but one is included for completeness. The final instruction before actually losing power is a HALT. This minimizes somewhat spurious transitions on the various busses and lets the processor die gracefully.

On reset, when a warm start is detected, execution is transferred to WSTART. WSTART activates $\overline{\text{PFSR}}$ by way of the 8255 (all outputs go low then the 8255 is initialized). In the power-down circuitry, $\overline{\text{PFSR}}$ clears the PFS latch and removes the $\overline{\text{MPRO}}$ signal which then allows access to the RAM. WSTART also clears the PFI latch which arms the 8259A IR0 input. Then the 8259A is re-initialized along with any other devices. The SP is retrieved from RAM and the processor registers and flags are restored by popping them off the stack. Interrupts are then enabled. Now the power-down program counter is on top of the stack, so executing a RETurn instruction transfers the processor to exactly where it left off before the power failure.

Aside from illustrating the usefulness of the 8259A (and the SBC-80/20) in implementing a power failure protected microcomputer system, this application should also point out a way of preserving the processor status when using interrupts.

```
LOC  OBJ     SLO     SOURCE STATEMENT

                  8 ;
                  1 ;
                  2 ;POWER DOWN AND RESTART FOR THE SBC 80/20
                  3 ;
                  4 ;
                  5 ;SYSTEM EQUATES
0000              6 PTS9A   EQU   0C9H        ;8259 PORT WITH A0=0
0000              7 PTS9B   EQU   0D9H        ;8259 PORT WITH A0=1
00E7              8 PPI1CT  EQU   0E7H        ;8255 #1 CONTROL PORT
00E6              9 PPI1C   EQU   0E6H        ;8255 #1 PORT C
C380             10 SPSAVE  EQU   C380H       ;SP STORAGE IN RAM
0001             11 JPT     EQU   01H        ;MSB OF 8259 JUMP TABLE
                 12 ;
                 13 ;
                 14 ;STARTING POINT AFTER SYSTEM RESET
                 15 ;
                 16 ;
0000             17         ORG   0H
0000 DB04        18 START   IN    04H         ;READ PFS STATUS
0002 1F          19         RRA                ;PFS/ ON D0 PUT IN CARRY
0003 DA2001      20         JC    CSTART       ;PFS=1  THEN COLD START
                 21 ;
                 22 ;
                 23 ;WSTART LOCATION  PFS=0  THEN WARM START
                 24 ;
                 25 ;
0006 3E80        26 WSTART  MVI   A,80H        ;SET 8255 #1 TO OUTPUT MODE
0008 D3E7        27         OUT   PPI1CT       ;8255 CONTROL PORT. PFS/ GOES LOW
                 28
                 29         ;OUTPUT COMMAND WHICH PFS/ GO LOW WHICH REMOVES PWR  NMI
                 30         ;CLEARS PFS LATCH
                 31
000A 3E01        32         MVI   A,01H        ;RETURN PFS/ HIGH
000C D3E6        33         OUT   PPI1C        ;8255 #1 PORT C
000E DB04        34         OUT   04H          ;RESET PFI LATCH
0010 CD1000      35         CALL  INIT         ;GO INITIALIZE EVERYTHING
0013 2A8003      36         LHLD  SPSAVE       ;RETRIVE SP FROM RAM
0016 F9          37         SPHL               ;PUT BACK INTO SP
0017 C1          38         POP   B            ;RESTORE BC
0018 D1          39         POP   D            ;RESTORE DE
0019 E1          40         POP   H            ;RESTORE HL
001A F1          41         POP   PSW          ;RESTORE A PLUS FLAGS
001B FB          42         EI                 ;ENABLE INTERRUPTS
001C C9          43         RET                ;ARE POWER-DOWN PC ON TOP OF STACK
                 44                            ;RETURN TO IT
                 45
                 46
                 47 ;INITIALIZATION ROUTINE  AT LEAST DO 8259 BUT OTHERS CAN BE ADDED
                 48
                 49 ;
001C 3E1C        50 INIT    MVI   A,16H        ;A=1,SI,A7-A5=0  ICW1
001E D3C9        51         OUT   PTS9A        ;8259 PORT WITH A0=0
0021 3E01        52         MVI   A,JPT        ;MSB OF JUMP TABLE ICW2
0023 D3D8        53         OUT   PTS9B        ;8259 PORT WITH A0=1
                 54
                 55         ;ADD ANY OTHER INITIALIZATIONS HERE
                 56
0025 C9          57         RET                ;RETURN
                 58 ;
                 59 ;
                 60 ;POWER DOWN ROUTINE  PC SAVE -DOWN DTS AND STATUS
                 61 ;
                 62 ;
0026 F5          63 PEDOWN  PUSH  PSW          ;SAVE A PLUS FLAGS
0027 E5          64         PUSH  H            ;SAVE HL
0028 D5          65         PUSH  D            ;SAVE DE
0029 C5          66         PUSH  B            ;SAVE BC
002A 210000      67         LXI   H,0000H      ;ADD SP TO GET SP
002D 39          68         DAD   SP           ;SP NOW IN HL
002E 228003      69         SHLD  SPSAVE       ;SAVE SP IN RAM
                 70
                 71         ;DON NO POLL  NEEDED BUT INCLUDED IF ...
                 72
002E 3E20        73         MVI   A,20H        ;NON-SPECIFIC EOI
0030 D3C9        74         OUT   PTS9A        ;8259 PORT WITH A0=0
0032 76          75         HLT                ;HALT  GO DOWN SPECIFICALLY
                 76
                 77
                 78 ;8259 JUMP TABLE  ONLY  IR0 IS USED  OTHERS UNPOSTED
                 79 ;
                 80
0100             81         ORG   100H
0100 C32600      82 JSTART  JMP   PEDOWN       ;IR0
0103 00          83         NOP
0104 C31028      84         JMP   2810H        ;IR1
0107 00          85         NOP
0108 C32028      86         JMP   2820H        ;IR2
010B 00          87         NOP
010C C33028      88         JMP   2830H        ;IR3
010F 00          89         NOP
0110 C34028      90         JMP   2840H        ;IR4
0113 00          91         NOP
0114 C35028      92         JMP   2850H        ;IR5
0117 00          93         NOP
0118 C36028      94         JMP   2860H        ;IR6
011B 00          95         NOP
011C C37028      96         JMP   2870H        ;IR7
011F 00          97         NOP
                 98 ;
                 99 ;
                100 ;COLD START LOCATION  USER'S PROGRAM ENTERS HERE
                101 ;
                102 ;
0120 31803F     103 CSTART  LXI   SP,3F80H     ;INITIALIZE SP
0123 CD1000     104         CALL  INIT         ;INITIALIZE EVERYTHING ELSE
0126 DB04       105         OUT   04H          ;RESET PFI LATCH
0128 FB         106         EI                 ;ENABLE INTERRUPTS
                107
                108         ;USER PROGRAM STARTS HERE
                109
                110         END                ;DONE
```

Figure 27. Power Down and Restart Software

5.2 78 LEVEL INTERRUPT SYSTEM

The second application illustrates an interrupt structure with greater than 64 levels for an 8080A or 8085A system. In the cascade mode, the 8259A supports up to 64 levels with direct vectoring to the service routine. Extending the structure to greater than 64 levels requires polling, using the poll command. A 78 level interrupt structure is used as an illustration; however, the principles apply to systems with up to 512 levels.

To implement the 78 level structure, 3 tiers of 8259A's are used. Nine 8259A's are cascaded in the master-slave scheme, giving 64 levels at tier 2. Two additional 8259A's are connected, by way of the INT outputs, to two of the 64 inputs. The 16 inputs at tier 3, combined with the 62 remaining tier 2 inputs, give 78 total levels. The fully nested structure is preserved over all levels, although direct vectoring is supplied for only the tier 2 inputs. Software is required to vector any tier 3 requests. Figure 28 shows the tiered structure used in this example. Notice that the tier 3 8259A's are connected to the bottom level slave (SA7). The master-slaves are interconnected as shown in "Interrupt Cascading", while the tier 3 8259A's are connected as "masters"; that is, the SP/EN pins are pulled high and the CAS pins are left unconnected. Since these 8259A's are only going to be used with the poll command, no INTA is required, therefore the INTA pins are pulled high.

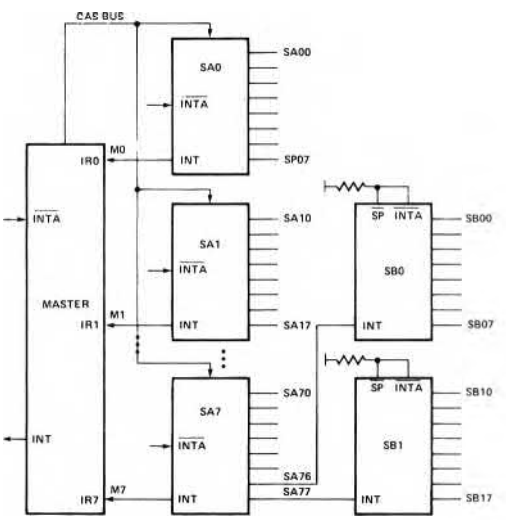


Figure 28. 78 Level Interrupt Structure

The concept used to implement the 78 levels is to directly vector to all tier 2 input service routines. If a tier 2 input contains a tier 3 8259A, the service routine for that input will poll the tier 3 8259A and branch to the tier 3 input service routine based on the poll word read after the poll command. Figure 29 shows how the jump table is organized assuming a starting location of 1000H and contiguous tables for all the tier 2 8259A's. Note that "SA35" denotes the IR5 input of the slave connected to the master IR3 input. Also note that for the normal tier 2 inputs, the jump table vectors the processor directly to the service routine for that input, while for the tier 2 inputs with 8259A's connected to their IR inputs, the processor is vectored to a service routine (i.e., SB0) which will poll to determine the actual tier 3 input requesting service. The polling routine utilizes the jump table starting at 1200H to vector the processor to the correct tier 3 service routine.

Each 8259A must receive an initialization sequence regardless of the mode. Since the tier 1 and 2 8259A's are in cascade and the special fully nested mode is used (covered shortly), all ICWs are required. The tier 3 8259A's don't require ICW3 or ICW4 since only polling will be used on them and they are connected as masters not in the cascade mode. The initialization sequence for each tier is shown in Figure 30. Notice that the master is initialized with a "dummy" jump table starting at 00H since all vectoring is done by the slaves. The tier 3 devices also receive "dummy" tables since only polling is used on tier 3.

As explained in "Interrupt Cascading", to preserve a truly fully nested mode within a slave, the master 8259A should be programmed in the special fully nested mode. This allows the master to acknowledge all interrupts at and above the level in service disregarding only those of lower priority. The special fully nested mode is programmed in the master only, so it only affects the immediate slaves (tier 2 not tier 3). To implement a fully nested structure among tier 3 slaves some special housekeeping software is required in all the tier-2-with-tier-3-slave routines. The software should simply save the state of the tier 2 IMR, mask all the lower tier 2 interrupts, then issue a specific EOI, resetting the ISR of the tier 2 interrupt level. On completion of the routine the IMR is restored.

Figure 31 shows an example flow and program for any tier 2 service routine without a tier 3 8259A. Figure 32 shows an example flow and program for any tier 2 service routine with a tier 3 8259A. Notice the reading of the ISR in both examples; this is done to determine whether or not to issue an EOI command to the master (refer to the section on "Special Fully Nested Mode" for further details).

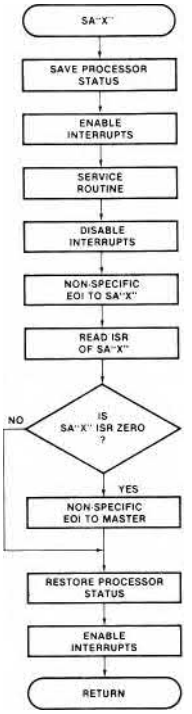
LOCATION	8259	CODE		COMMENTS
1000 H	SA0	JMP	SA00	SA00 SERVICE ROUTINE
.				
.				
101C H		JMP	SA07	. SA07 SERVICE ROUTINE
1020 H	SA1	JMP	SA10	SA10 SERVICE ROUTINE
.				
.				
103C H		JMP	SA17	SA17 SERVICE ROUTINE
.	.		.	
.	.		.	SA20–SA67 SERVICE ROUTINES
.	.		.	
10E0 H	SA7	JMP	SA70	. SA70 SERVICE ROUTINE
.				
.				
10F8 H		JMP	SB0	. SB0 POLL ROUTINE
10FC H		JMP	SB1	. SB1 POLL ROUTINE
1200 H	SB0	JMP	SB00	. SB00 SERVICE ROUTINE
.				
.				
121C H		JMP	SB07	. SB07 SERVICE ROUTINE
1220 H	SB1	JMP	SB10	. SB10 SERVICE ROUTINE
.				
.				
123C H		JMP	SB17	. SB17 SERVICE ROUTINE

Figure 29. Jump Table Organization

```
; INITIALIZATION SEQUENCE FOR 78 LEVEL INTERRUPT STRUCTURE
;
; INITIALIZE MASTER
;
MINT:   MVI     A,15H       ; ICW1, LTM = 0, ADI = 1, S = 0, IC4 = 1
        OUT     MPTA        ; MASTER PORT A0 = 0
        MVI     A,00H       ; ICW2, DUMMY ADDRESS
        OUT     MPTB        ; MASTER PORT A0 = 1
        MVI     A,0FFH      ; ICW3, S7-S0 = 1
        OUT     MPTB        ; MASTER PORT A0 = 1
        MVI     A,10H       ; ICW4, SFNM = 1
        OUT     MPTB        ; MASTER PORT A0 = 1
;
; INITIALIZE SA SLAVES - X DENOTES SLAVE ID (SEE KEY)
;
SAXINT: MVI     A,..        ; SEE KEY FOR ICW1, LTM = 0, ADI = 1, S = 0, IC4 = 1
        OUT     SAXPTA      ; SA"X" PORT A0 = 0
        MVI     A, 10H      ; ICW2, ADDRESS MSB
        OUT     SAXPTB      ; SA"X" PORT A0 = 1
        MVI     A0XH        ; ICW3, SA ID
        OUT     SAXPTB      ; SA"X" PORT A0 = 1
        MVI     A10H        ; ICW4, SFNM = 1
        OUT     SAXPTB      ; SA"X" PORT A0 = 1
;
; REPEAT ABOVE FOR EACH SA SLAVE
;
; INITIALIZE SB SLAVES - X DENOTES 0 or 1 (DO SB0, REPEAT FOR SB1)
;
SBXINT  MVI     A,16H       ; ICW1, LTM = 0, ADI = 1, S = 1, IC4 = 0
        OUT     SBXPTA      ; SB"X" PORT A0 = 0
        MVI     A,00H       ; ICW2, DUMMY ADDRESS
        OUT     SBXPTB      ; SB"X" PORT A0 = 1
```

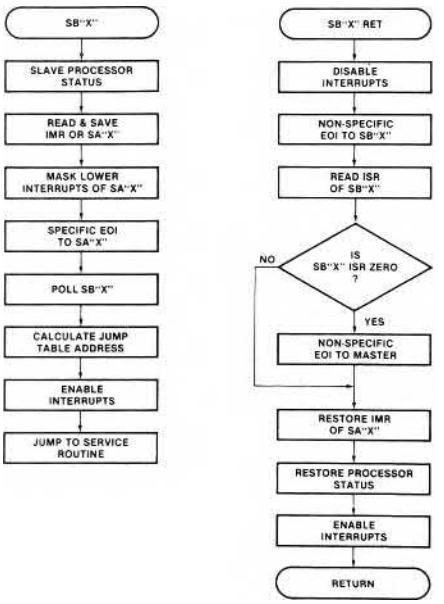
SA INITIALIZATION KEY		
SA"X"	σ (ICW1)	JUMP TABLE START (H)
0	15	1000
1	35	1020
2	55	1040
3	75	1060
4	95	1080
5	B5	10A0
6	D5	10C0
7	F5	10E0

Figure 30. Initialization Sequence for 78 Level Interrupt Structure




```
; SA"X" ROUTINE - GENERAL INTERRUPT SERVICE ROUTINE
; FOR TIER 2 INTERRUPTS WITHOUT TIER 3 8259A
;
SAX:    PUSH D          ; SAVE DE
        PUSH B          ; SAVE BC
        PUSH H          ; SAVE HL
        PUSH PSW        ; SAVE A, FLAGS
        EI              ; ENABLE INTERRUPTS
;
; SERVICE ROUTINE GOES HERE
;
        DI              ; DISABLE INTERRUPTS
        MVI     20H     ; OCW2, NON-SPECIFIC EOI
        OUT     SAXPTA  ; SA"X" PORT A0 = 0
        MUI     A.0BH   ; OCW3, READ REGISTER, ISR
        OUT     SAXPTA  ; SA"X" PORT A0 = 0
        IN      SAXPTA  ; SA"X" PORT A0 = 0. SA"X" ISR
        ANI     0FFH    ; TEST FOR ZERO
        JZN     SAXRSR  ; IF NOT ZERO, RESTORE STATUS
        MVI     A.0BH   ; OCW2, NON-SPECIFIC EOI
        OUT     MASPTA  ; MASTER PORT A0 = 0
SAXRSR: POP     PSW     ; RESTORE A, FLAGS
        POP     H       ; RESTORE HL
        POP     B       ; RESTORE BC
        POP     D       ; RESTORE DE
        EI              ; ENABLE INTERRUPTS
        RET             ; RETURN
```

Figure 31. Example Service Routine for Tier 2 Interrupt (SA"X") without Tier 3 8259A (SB"X")




```
; SB"X" ROUTINE - SERVICE ROUTINE FOR TIER 2
; INTERRUPTS WITH TIER 3 8259AS
SBX:    PUSH D                  ; SAVE DE
        PUSH B                  ; SAVE BC
        PUSH H                  ; SAVE HL
        PUSH PSW                ; SAVE A, FLAGS
        IN      SAXPTB          ; READ SA"X" IMR
        MOV     D,A             ; SAVE
        MVI     A,XXH           ; MASK SA"X" LOWER IR
        OUT     SAXPTB          ; SA"X" PORT A0 = 1
        MVI     A,6XH           ; OCW2 SPECIFIC EOI SA"X"
        OUT     SAXPTA          ; SA"X" PORT A0 = 1
        LXI     H,1200H         ; JUMP TABLE START
        MVI     B,00H           ; CLEAR B
        MVI     A,0CH           ; OCW3, POLL COMMAND
        OUT     SBXPTA          ; SB"X" PORT A0 = 0
        IN      SBXPTA          ; GET POLL WORD
        ANI     07H             ; LIMIT TO 3 BITS
        ADD     A               ; GET TABLE OFFSET
        ADD     A
        MOV     C,A             ; OFFSET TO C
        DAD     B               ; HL HAS TABLE ADDRESS
        EI                      ; ENABLE INTERRUPTS
;
; SB"X"RET ROUTINE - FOR EOI AND MASK RESTORE
; AFTER SB"X" ROUTINE
;
SBXRET  DI                      ; DISABLE INTERRUPTS
        MVI     A,20H           ; OCW2, NON SPECIFIC EOI
        OUT     SBXPTA          ; SA"X" PORT A0 = 0
        MVI     A,0BH           ; OCW3, READ REGISTER ISR
        OUT     SAXPTA          ; SA"X" PORT A0 = 0
        IN      SBXPTA          ; SA"X" PORT A0 = 0, ISR
        ANI     0FFH            ; TEST FOR ZERO
        JNZ     SBXRSR          ; IF ≠ 0 RESTORE IMR
        MVI     A,20H           ; OCW2, NON-SPECIFIC EOI
        OUT     MASPTA          ; MASTER PORT A0 = 0
SBXRSR: MOV     A,D             ; RESTORE SA"X" IMR
        OUT     SAXPTB          ; SA"X" PORT A0 = 1
        POP     PSW             ; RESTORE A, FLAGS
        POP     H               ; RESTORE HL
        POP     B               ; RESTORE BC
        POP     D               ; RESTORE DE
        EI                      ; RESTORE DE
        RET                     ; RETURN
```

Figure 32. Example Service Routine for Tier 2 Interrupt (SA"X") with Tier 3 8259A (SB"X")

5.3 TIMER CONTROLLED INTERRUPTS

In a large number of controller type microprocessor designs, certain timing requirements must be implemented throughout program execution. Such time dependent applications include control of keyboards, displays, CRTs, printers, and various facets of industrial control. These examples, however, are just a few of many designs which require device servicing at specific rates or generation of time delays. Trying to maintain these timing requirements by processor control alone can be costly in throughput and software complexity. So, what can be done to alleviate this problem? The answer, use the 8259A Programmable Interrupt Controller and external timing to interrupt the processor for time dependent device servicing.

This application example uses the 8259A for timer controlled interrupts in an 8086 system. External timing is done by two 8253 Programmable Interval Timers. Figure 33 shows a block diagram of the timer controlled interrupt circuitry which was built on the breadboard area of an SDK-86 (system design kit). Besides the 8259A and the 8253's, the necessary I/O decoding is also shown. The timer controlled interrupt circuitry interfaces with the SDK-86 which serves as the vehicle of operation for this design.

A short overview of how this application operates is as follows. The 8253's are programmed to generate interrupt requests at specific rates to a number of the 8259A IR inputs. The 8259A processes these requests by interrupting the 8086 and vectoring program execution to the appropriate service routine. In this example, the routines use the SDK-86 display panel to display the number of the interrupt level being serviced. These routines are merely for demonstration purposes to show the necessary procedures to establish the user's own routines in a timer controlled interrupt scheme.

Let's go over the operation starting with the actual interrupt timing generation which is done by two 8253 Programmable Interval Timers (8253 #1 and 8253 #2). Each 8253 provides three individual 16-bit counters (counters 0-2) which are software programmable by the processor. Each counter has a clock input (CLK), gate input (GATE), and an output (OUT). The output signal is based on divisions of the clock input signal. Just how or when the output occurs is determined by one of the 8253's six programmable modes, a programmable 16-bit count, and the state of the gate input.

Figure 34 shows the 8253 timing configuration used for generating interrupts to the 8259A. The SDK-86's PCLK (peripheral clock) signal provides a 400 ns period clock to CLK0 of 8253 #1. Counter 0 is used in mode 3 (square wave rate generator), and acts as a prescaler to provide the clock inputs of the other counters with a 10 ms period square wave. This 10 ms clock period made it easy to calculate exact timings for the other counters. Counter 2 of the 8253 #1 is used in mode 2 (rate generator), it is programmed to output a 10 ms pulse for every 200 pulses it receives (every 2 sec). The output of counter 2 causes an interrupt on IR1 of the 8259A. All the 8253 #2 counters are used in mode 5 (hardware triggered strobe) in which the gate input initiates counter operations. In this case the output of 8253 #1 counter 2 controls the gate of each 8253 #2 counter. When one of the 8253 #2 counters receive the 8253 #1 counter 2 output pulse on its gate, it will output a pulse (10 ms in duration) after a certain preprogrammed number of clock pulses have occurred. The programmed number of clock pulses for the 8253 #2 counters is as follows. 50 pulses (0.5 sec) for counter 0, 100 pulses (1 sec) for counter 1, and 150 pulses (1.5 sec) for counter 2. The outputs of these counters cause interrupt requests on IR2 through IR4 of the 8259A. Counter 1 of 8253 #1 is used in mode 0 (interrupt on terminal count). Unlike the other modes used which initialize operation automatically or by gate triggering, mode 0 allows software controlled counter initialization. When counter 1 of 8253 #1 is set during program execution, it will count 25 clocks (250 ms) and then pull its output high, causing an interrupt request on IR0 of the 8259A. Figure 35 shows the timing generated by the 8253's which cause interrupt request on the 8259A IR inputs.

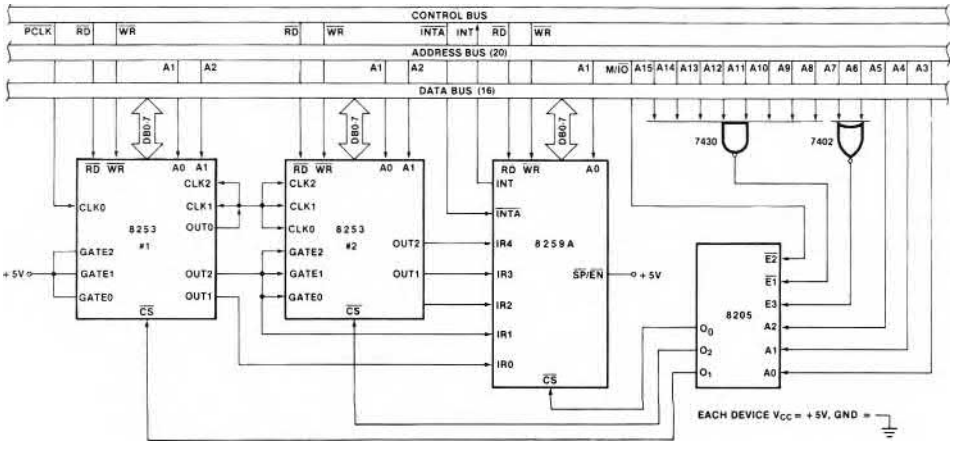


Figure 33. Timer Controlled Interrupt Circuit on SDK 86 Breadboard Area

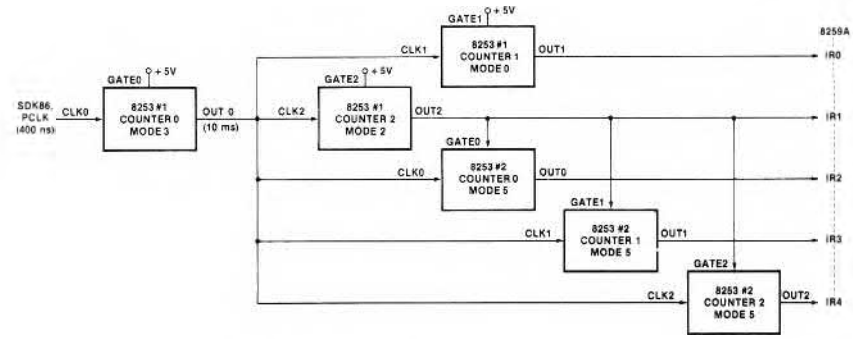


Figure 34. 8253 Timing Configuration for Timer Controlled Interrupts

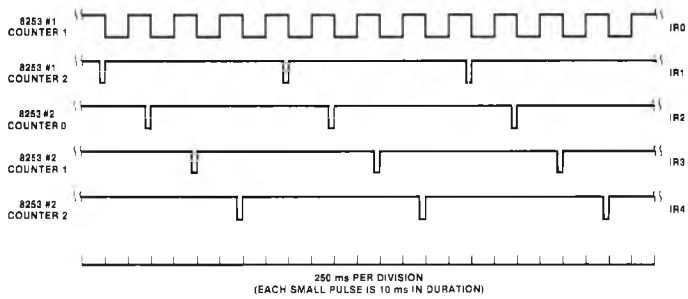


Figure 35. 8259A IR Input Signal From 8253S

There are basically two methods of timing generation that can be used in a timer controlled interrupt structure: dependent timing and independent timing. Dependent timing uses a single timing occurrence as a reference to base other timing occurrences on. On the other hand, independent timing has no mutual reference between occurrences. Industrial controller type applications are more apt to use dependent timing, whereas independent timing is prone to individual device control.

Although this application uses primarily dependent timing, independent timing is also incorporated as an example. The use of dependent timing can be seen back in Figure 34, where timing for IR2 through IR4 uses the IR1 pulse as reference. Each one of the 8253 #2 counters will generate an interrupt request a specific amount of times after the IR1 interrupt request occurs. When using the dependent method, as in this case, the IR2 through IR4 requests must occur before the next IR1 request. Independent timing is used to control the IR0 interrupt request. Note that its timing isn't controlled by any of the other IR requests. In this timer controlled interrupt configuration the dependent timing is initially set to be self running and the independent timing is software initialized. However, both methods can work either way by using the various 8253 modes to generate the same interrupt timing.

The 8259A processes the interrupts generated by the 8253's according to how it is programmed. In this application it is programmed to operate in the edge triggered mode, MCS-86/88 mode, and automatic EOI mode. In the edge triggered mode an interrupt request on an 8259A IR input becomes active on the rising edge. With this in mind, Figure 35 shows that IR0 will generate an interrupt every half second and IR1 through IR4 will each generate an interrupt every 2 seconds spaced apart at half second intervals. Interrupt vectoring in the MCS-86/88 mode is programmed so IR0, when activated, will select interrupt type 72. This means IR1 will select interrupt type 73, IR2 interrupt type 74, and so on through IR4. Since IR5 through IR7 aren't used, they are masked off. This prevents the possibility of any accidental interrupts and rids the necessity to tie the unused IR inputs to a steady level. Figure 36 shows the 8259A IR levels (IR0–IR4) with their corresponding interrupt type in the 8086 interrupt-vector table. Type 77 in the table is selected by a software "INT" instruction during program execution. Each type is programmed with the necessary code segment and instruction pointer values for vectoring to the appropriate service routine. Since the 8259A is programmed in the automatic EOI Mode, it doesn't require an EOI command to designate the completion of the service routine.

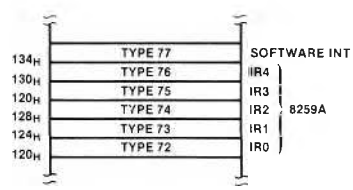


Figure 36. Interrupt "Type" Designation

As mentioned earlier, the interrupt service routines in this application are used merely to demonstrate the timer controlled interrupt scheme, not to implement a particular design. Thus a service routine simply displays the number of its interrupting level on the SDK-86 display panel. The display panel is controlled by the 8279 Keyboard and Display Controller. It is initialized to display "Ir" in its two left-most digits during the entire display sequence. When an interrupt from IR1 through IR4 occurs the corresponding routine will display its IR number via the 8279. During each IR1 through IR4 service routine a software "INT77" instruction is executed. This instruction vectors program execution to the service routine designated by type 77, which sets the 8253 counter controlling IR0 so it will cause an interrupt in 250 ms. When the IR0 interrupt occurs its routine will turn off the digit displayed by the IR1 through IR4 routines. Thus each IR level (IR1-IR4) will be displayed for 250 ms followed by a 250 ms off time caused by IR0. Figure 37 shows the entire display sequence of the timer controlled interrupt application.

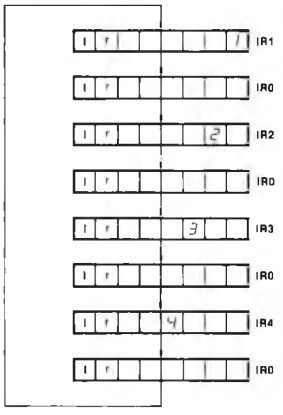


Figure 37. SDK Display Sequence for Timer Controlled Interrupts Program (Each Display Block Shown is 250 msec in Duration)

Now that we've covered the operation, let's move on to the program flow and structure of the timer controlled interrupt program. The program flow is made up of an initialization section and six interrupt service routines. The initialization program flow is shown in Figure 38. It starts by initializing some of the 8086's registers for program operation; this includes the extra segment, data segment, stack segment, and stack pointer. Next, by using the extra segement as reference, interrupt types 72 through 77 are set to vector interrupts to the appropriate routines. This is done by moving the code segment and instruction pointer values of each service routine into the corresponding type location. The 8253 counters are then programmed with the proper mode and count to provide the interrupt timing mentioned earlier. All counters with the exception of the 8253 #1, counter 1 are fully initialized at this point and will start counting. Counter 1 of 8253 #1 starts counting when its counter is loaded during the "INTR77" service routine, which will be covered shortly. Next, the 8259A is issued ICW1, ICW2, ICW4, and OCW1. The ICWs program the 8259A for the edge triggered mode, automatic EOI mode, and the proper interrupt vectoring (IR0, type 72). OCW1 is used to mask off the unused IR inputs (IR5-IR7). The 8279 is then set to display "IR" on its two left-most digits. After that the 8086 enables interrupts and a "dummy" main program is executed to wait for interrupt requests.

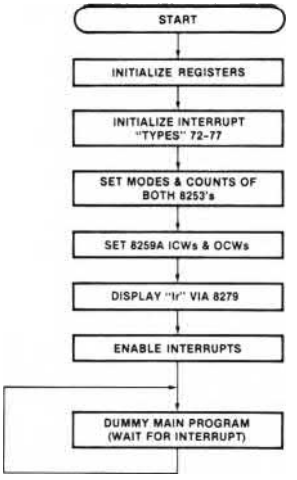


Figure 38. Initialization Program Flow for Timer Controlled Interrupts

There are six different interrupt service routines used in the program. Five of these routines, "INTR72" through "INTR76", are vectored to via the 8259A. Figure 39A-C shows the program flow for all six service routines. Note that "INTR73" through "INTR76" (IR1-IR4) basically use the same flow. These four similar routines display the number of its interrupting IR level on the SDK-86 display panel. The "INTR77" routine is vectored to by software during each of the previously mentioned routines and sets up interrupt timing to cause the "INTR72" (IR0) routine to be executed. The "INTR72" routine turns off the number on the SDK-86 display panel.

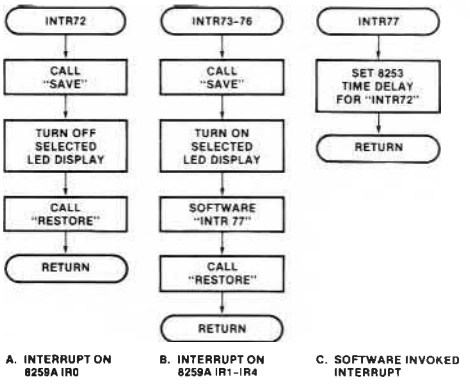


Figure 39. A-C. Interrupts Service Routine Flow for Timer Controlled Interrupts.

To best explain how these service routines work, let's assume an interrupt occurred on IR1 of the 8259A. The associated service routine for IR1 is "INTR73". Entering "INTR73", the first thing done is saving the pre-interrupt program status. This isn't really necessary in this program since a "dummy" main program is being executed; however, it is done as an example to show the operation. Rather than having code for saving the registers in each separate routine, a mutual call routine, "SAVE", is used. This routine will save the register status by pushing it on the stack. The next portion of "INTR73" will display the number of its IR level, "1", in the first digit of the SDK-86 display panel. After that, a software INT instruction is executed to vector program execution to the "INTR77" service routine. The "INTR77" service routine simply sets the 8253 #1 counter 1 to cause an interrupt on IR0 in 250 ms and then returns to "INTR73". Once back in "INTR73", the pre-interrupt status is restored by a call routine, "RESTORE". It does the opposite of "SAVE", returning the register status by popping it off the stack. The "INTR73" routine then returns to the "dummy " main program. The flow for the "INTR74" through "INTR76" routines are the same except for the digit location and the IR level displayed.

After 250 ms have elapsed, counter 1 of 8253 #1 makes an interrupt request on IR0 of the 8259A. This causes the "INTR72" service routine to be executed. Since this routine interrupts the main program, it also uses the "SAVE" routine to save pre-interrupt program status. It then turns off the digit displaying the IR level. In the case of the "INTR73" routine, the "1" is blanked out. The pre-interrupt status is then restored using the "RESTORE" routine and program execution returns to the "dummy" main program.

The complete program for the timer controlled interrupts application is shown in Appendix B. The program was executed in SDK-86 RAM starting at location 0500H (code segment = 0050, instruction pointer = 0).

CONCLUSION

This application note has explained the 8259A in detail and gives three applications illustrating the use of some of the numerous programmable features available. It should be evident from these discussions that the 8259A is an extremely flexible and easily programmable member of the Intel® MCS-80, MCS-85, MCS-86, and MCS-88 families.

This table is provided merely for reference information between the "Operation of the 8259A" and "Programming the 8259A" sections of this application note. It shouldn't be used as a programming reference guide (see "Programming the 8259A").

Operational Description	Command Words	Bits
MCS-80/85™ Mode	ICW1, ICW4*	IC4, μPM*
Address Interval for MCS-80/85 Mode	ICW1	ADI
Interrupt Vector Address for MCS-80/85 Mode	ICW1, ICW2	A5-A15
MCS-86/88 Mode	ICW1, ICW4	IC4, μPM
Interrupt Vector Byte for MCS-86/88 Mode	ICW2	T3-T7
Fully Nested Mode	OCW-Default	—
Non-Specific EOI Command	OCW2	EOI
Specific EOI Command	OCW2	SEOI, EOI, L0-L2
Automatic EOI Mode	ICW1, ICW4	IC4, AEOI
Rotate On Non-Specific EOI Command	OCW2	EOI
Rotate In Automatic EOI Mode	OCW2	R, SEOI, EOI
Set Priority Command	OCW2	L0-L2
Rotate on Specific EOI Command	OCW2	R, SEOI, EOI
Interrupt Mask Register	OCW1	M0-M7
Special Mask Mode	OCW3	ESMM-SMM
Level Triggered Mode	ICW1	LTIM
Edge Triggered Mode	ICW1	LTIM
Read Register Command, IRR	OCW3	ERIS, RIS
Read Register Command, ISR	OCW3	ERIS, RIS
Read IMR	OCW1	M0-M7
Poll Command	OCW3	P
Cascade Mode	ICW1, ICW3	SNGL, S0-7, ID0-2
Special Fully Nested Mode	ICW1, ICW4	IC4, SFNM
Buffered Mode	ICW1, ICW4	IC4, BUF, M/S

*Only needed if ICW4 is used for purposes other than μP mode set.

```
MCS-86 ASSEMBLER    TCI59A


ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE TCI59A
OBJECT MODULE PLACED IN  F1:TCI59A.OBJ
ASSEMBLER INVOKED BY   F1:ASM86  F1:TCI59A.SRC

LOC  OBJ                LINE   SOURCE

                         1   ;******************* TIMER CONTROLLED INTERRUPTS ******************
                         2   ;
                         3   ;
                         4   ;
                         5   ;               EXTRA SEGMENT DECLARATIONS
                         6   ;
----                     7   EXTRA SEGMENT
                         8   ;
0120                     9           ORG     120H
0120 0401               10   TP72IP  DW      INTR72          ;TYPE 72 INSTRUCTION POINTER
0122 ????               11   TP72CS  DW      ?               ;TYPE 72 CODE SEGMENT
0124 1801               12   TP73IP  DW      INTR73          ;TYPE 73 INSTRUCTION POINTER
0126 ????               13   TP73CS  DW      ?               ;TYPE 73 CODE SEGMENT
0128 3001               14   TP74IP  DW      INTR74          ;TYPE 74 INSTRUCTION POINTER
012A ????               15   TP74CS  DW      ?               ;TYPE 74 CODE SEGMENT
012C 4801               16   TP75IP  DW      INTR75          ;TYPE 75 INSTRUCTION POINTER
012E ????               17   TP75CS  DW      ?               ;TYPE 75 CODE SEGMENT
0130 6001               18   TP76IP  DW      INTR76          ;TYPE 76 INSTRUCTION POINTER
0132 ????               19   TP76CS  DW      ?               ;TYPE 76 CODE SEGMENT
0134 7801               20   TP77IP  DW      INTR77          ;TYPE 77 INSTRUCTION POINTER
0136 ????               21   TP77CS  DW      ?                TYPE 77 CODE SEGMENT
                        22   ;
----                    23   EXTRA ENDS
                        24 * ;
                        25   ;               DATA SEGMENT DECLARATIONS
                        26   ;
----                    27   DATA    SEGMENT
                        28   ;
0000 ????               29   STACK1  DW      ?                       ;VARIABLE TO SAVE CALL ADDRESS
0002 ????               30   AXTEMP  DW      ?                       ;VARIABLE TO SAVE AX REGISTER
0004 ??                 31   DIGIT   DB      ?                       ;VARIABLE TO SAVE SELECTED DIGIT
                        32   ;
----                    33   DATA    ENDS
                        34   ;
                        35   ;               CODE SEGMENT DECLARATION
                        36   ;
----                    37   CODE    SEGMENT
                        38   ;
                        39   ASSUME  ES:EXTRA,DS:DATA,CS:CODE
                        40   ;
                        41   ;               INITIALIZE REGISTERS
                        42   ;
0000 B80000             43   START:  MOV     AX,0H                   ;EXTRA SEGMENT AT 0H
0003 8EC0               44           MOV     ES,AX
0005 B87000             45           MOV     AX,70H                  ;DATA SEGMENT AT 700H
0008 8ED8               46           MOV     DS,AX
000A B87800             47           MOV     AX,78H                  ;STACK SEGMENT AT 780H
000D 8ED0               48           MOV     SS,AX
000F BC8000             49           MOV     SP,80H                  ;STACK POINTER AT 80H (STACK=800H)
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ                LINE   SOURCE

                         50   ;
                         51   ;              LOAD INTERRUPT VECTOR TABLE
                         52   ;
0012 B80401              53   TYPES:  MOV    AX,OFFSET (INTR72)      ;LOAD TYPE 72
0015 26A32001            54          MOV    TP72IP,AX
0019 268C0E2201          55          MOV    TP72CS,CS
001E B81801              56          MOV    AX,OFFSET (INTR73)      ;LOAD TYPE 73
0021 26A32401            57          MOV    TP73IP,AX
0025 268C0E2601          58          MOV    TP73CS,CS
002A B83001              59          MOV    AX,OFFSET (INTR74)      ;LOAD TYPE 74
002D 26A32801            60          MOV    TP74IP,AX
0031 268C0E2A01          61          MOV    TP74CS,CS
0036 B84801              62          MOV    AX,OFFSET (INTR75)      ;LOAD TYPE 75
0039 26A32C01            63          MOV    TP75IP,AX
003D 268C0E2E01          64          MOV    TP75CS,CS
0042 B86001              65          MOV    AX,OFFSET (INTR76)      ;LOAD TYPE 76
0045 26A33001            66          MOV    TP76IP,AX
0049 268C0E3201          67          MOV    TP76CS,CS
004E B87801              68          MOV    AX,OFFSET (INTR77)      ;LOAD TYPE 77
0051 26A33401            69          MOV    TP77IP,AX
0055 268C0E3601          70          MOV    TP77CS,CS
                         71   ;
                         72   ;              8253 INITIALIZATION
                         73   ;
005A BA0EFF              74   SET531: MOV    DX,0FF0EH               ;8253 #1 CONTROL WORD
005D B036                75          MOV    AL,36H                  ;COUNTER 0, MODE 3, BINARY
005F EE                  76          OUT    DX,AL
0060 B071                77          MOV    AL,71H                  ;COUNTER 1, MODE 0, BCD
0062 EE                  78          OUT    DX,AL
0063 B0B5                79          MOV    AL,0B5H                 ;COUNTER 2, MODE 2, BCD
0065 EE                  80          OUT    DX,AL
0066 BH08FF              81          MOV    DX,0FF08H               ;LOAD COUNTER 0 (10MS)
0069 B0A8                82          MOV    AL,0A8H                 ;LSB
006B EE                  83          OUT    DX,AL
006C B061                84          MOV    AL,61H                  ;MSB
006E EE                  85          OUT    DX,AL
006F BA0CFF              86          MOV    DX,0FF0CH               ;LOAD COUNTER 2 (2SEC)
0072 B000                87          MOV    AL,00H                  ;LSB
0074 EE                  88          OUT    DX,AL
0075 B002                89          MOV    AL,02H                  ;MSB
0077 EE                  90          OUT    DX,AL
0078 BA16FF              91   SET532: MOV    DX,0FF16H               ;8253 #2 CONTROL WORD
007B B03B                92          MOV    AL,3BH                  ;COUNTER 0, MODE 5,BCD
007D EE                  93          OUT    DX,AL
007E B07B                94          MOV    AL,7BH                  ;COUNTER 1, MODE 5, BCD
0080 EE                  95          OUT    DX,AL
0081 B0BB                96          MOV    AL,0BBH                 ;COUNTER 2, MODE 5, BCD
0083 EE                  97          OUT    DX,AL
0084 BA10FF              98          MOV    DX,0FF10H               ;LOAD COUNTER 0 (.5SEC)
0087 B050                99          MOV    AL,50H                  ;LSB
0089 EE                 100          OUT    DX,AL
008A B000               101          MOV    AL,00H                  ;MSB
008C EE                 102          OUT    DX,AL
008D BA12FF             103          MOV    DX,0FF12H               ;LOAD COUNTER 1 (1SEC)
0090 B000               104          MOV    AL,00H                  ;LSB
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ                LINE   SOURCE

0092 EE                 105            OUT     DX,AL
0093 B001               106            MOV     AL,01H                  ;MSB
0095 EE                 107            OUT     DX,AL
0096 BA14FF             108            MOV     DX,0FF14H               ;LOAD COUNTER 2 (1.5SEC)
0099 B050               109            MOV     AL,50H                  ;LSB
009B EE                 110            OUT     DX,AL
009C B001               111            MOV     AL,01H                  ;MSB
009E EE                 112            OUT     DX,AL
                        113    ;
                        114    ;               8259A INITIALIZATION
                        115    ;
009F BA00FF             116    SET59A  MOV     DX,0FF00H               ;8259A A0=0
00A2 B013               117            MOV     AL,13H                  ;ICW1-LTIM=0,S=1,IC4=1
00A4 EE                 118            OUT     DX,AL
00A5 BA02FF             119            MOV     DX,0FF02H               ;8259A A0=1
00A8 B048               120            MOV     AL, 48H                 ;ICW2-INTERRUPT TYPE 72 (120H)
00AA EE                 121            OUT     DX,AL
00AB B003               122            MOV     AL, 03H                 ;ICW4-SFNM=0,BUF=0,AEOI=1,MPM=1
00AD EE                 123            OUT     DX,AL
00AE B0E0               124            MOV     AL,0E0H                 ;OCW1-MASK IR5,6,7 (NOT USED)
00B0 EE                 125            OUT     DX,AL
                        126    ;
                        127    ;               8279 INITIALIZATION
                        128    ;
00B1 BAEAFF             129    SET79:  MOV     DX,0FFEAH               ;8279 COMMAND WORDS AND STATUS
00B4 B0D0               130            MOV     AL,0D0H                 ;CLEAR DISPLAY
00B6 EE                 131            OUT     DX,AL
00B7 EC                 132    WAIT79  IN      AL,DX                   ;READ STATUS
00B8 D0C0               133            ROL     AL,1                    ;"DU" BIT TO CARRY
00BA 72FB               134            JB      WAIT79                  ;JUMP IF DISPLAY IS UNAVAILABLE
00BC B087               135            MOV     AL, 87H                 ;DIGIT 8
00BE EE                 136            OUT     DX,AL
00BF BAE8FF             137            MOV     DX,0FFE8H               ;8279 DATA WORD
00C2 B006               138            MOV     AL,06H                  ;CHARACTER "I"
00C4 EE                 139            OUT     DX,AL
00C5 BAEAFF             140            MOV     DX,0FFEAH               ;8279 COMMAND WORD
00C8 B086               141            MOV     AL,86H                  ;DIGIT 7
00CA EE                 142            OUT     DX,AL
00CB BAE8FF             143            MOV     DX,0FFE8H               ;8279 DATA WORD
00CE B050               144            MOV     AL,50H                  ;CHARACTER "R"
00D0 EE                 145            OUT     DX,AL
00D1 FB                 146            STI                             ;ENABLE INTERRUPTS
                        147    ;
                        148    ;
                        149    ;               DUMMY PROGRAM
                        150    ;
00D2 EBFE               151    DUMMY   JMP     DUMMY                   ;WAIT FOR INTERRUPT
                        152    ;
                        153    ;
00D4 A30200             154    SAVE:   MOV     AXTEMP,AX               ;SAVE AX
00D7 58                 155            POP     AX                      ;POP CALL RETURN ADDRESS
00D8 A30000             156            MOV     STACK1,AX               ;SAVE CALL RETURN ADDRESS
00DB A10200             157            MOV     AX,AXTEMP               ;RESTORE AX
00DE 50                 158            PUSH    AX                      ;SAVE PROCESSOR STATUS
00DF 53                 159            PUSH    BX
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ                LINE   SOURCE

00E0 51                  160          PUSH    CX
00E1 52                  161          PUSH    DX
00E2 55                  162          PUSH    BP
00E3 56                  163          PUSH    SI
00E4 57                  164          PUSH    DI
00E5 1E                  165          PUSH    DS
00E6 06                  166          PUSH    ES
00E7 A10000              167          MOV     AX,STACK1              ;RESTORE CALL RETURN ADDRESS
00EA 50                  168          PUSH    AX                     ;PUSH CALL RETURN ADDRESS
00EB C3                  169          RET
                         170   ;
00EC 58                  171   RESTOR: POP    AX                     ;POP CALL RETURN ADDRESS
00ED A30000              172          MOV     STACK1,AX              ;SAVE CALL RETURN ADDRESS
00F0 07                  173          POP     ES                     ;RESTORE PROCESSOR STATUS
00F1 1F                  174          POP     DS
00F2 5F                  175          POP     DI
00F3 5E                  176          POP     SI
00F4 5D                  177          POP     BP
00F5 5A                  178          POP     DX
00F6 59                  179          POP     CX
00F7 5B                  180          POP     BX
00F8 58                  181          POP     AX
00F9 A30200              182          MOV     AXTEMP,AX              ;SAVE AX
00FC A10000              183          MOV     AX,STACK1              ;RESTORE CALL RETURN ADDRESS
00FF 50                  184          PUSH    AX                     ;PUSH CALL RETURN ADDRESS
0100 A10200              185          MOV     AX,AXTEMP              ;RESTORE AX
0103 C3                  186          RET
                         187   ;
                         188   ;
                         189   ;              INTERRUPT 72, CLEAR DISPLAY,IR0 8259A
                         190   ;
0104 E8CDFF              191   INTR72: CALL   SAVE                   ;ROUTINE TO SAVE PROCESSOR STATUS
0107 BAEAFF              192          MOV     DX,0FFEAH              ;8279 COMMAND WORD
010A A00400              193          MOV     AL,DIGIT               ;SELECTED LED DIGIT
010D EE                  194          OUT     DX,AL
010E BAE8FF              195          MOV     DX,0FFE8H              ;8279 DATA
0111 B000                196          MOV     AL,00H                 ;BLANK OUT DIGIT
0113 EE                  197          OUT     DX,AL
0114 E8D5FF              198          CALL    RESTOR                 ;ROUTINE TO RESTORE PROCESSOR STATUS
0117 CF                  199          IRET                           ;RETURN FROM INTERRUPT
                         200   ;
                         201   ;
                         202   ;              INTERRUPT 73, IR1 8259A
                         203   ;
0118 E8B9FF              204   INTR73: CALL   SAVE                   ;ROUTINE TO SAVE PROCESSOR STATUS
011B BAEAFF              205          MOV     DX,0FFEAH              ;8279 COMMAND WORD
011E B080                206          MOV     AL,80H                 ;LED DISPLAY DIGIT 1
0120 A20400              207          MOV     DIGIT,AL
0123 EE                  208          OUT     DX,AL
0124 BAE8FF              209          MOV     DX,0FFE8H              ;8279 DATA
0127 B006                210          MOV     AL,06H                 ;CHARACTER "1"
0129 EE                  211          OUT     DX,AL
012A CD4D                212          INT     77                     ;TIMER DELAY FOR LED ON TIME
012C E8BDFF              213          CALL    RESTOR                 ;ROUTINE TO RESTORE PROCESSOR STATUS
012F CF                  214          IRET                           ;RETURN FROM INTERRUPT
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ                LINE   SOURCE

                        215   ;
                        216   ;
                        217   ;              INTERRUPT 74, IR2 8259A
                        218   ;
0130 E8A1FF             219   INTR74: CALL   SAVE                    ;ROUTINE TO SAVE PROCESSOR STATUS
0133 BAEAFF             220          MOV    DX,0FFEAH               ;8279 COMMAND WORD
0136 B081               221          MOV    AL,81H                  ;LED DISPLAY DIGIT 2
0138 A20400             222          MOV    DIGIT,AL
013B EE                 223          OUT    DX,AL
013C BAE8FF             224          MOV    DX,0FFE8H               ;8279 DATA
013F B05B               225          MOV    AL,5BH                  ;CHARACTER "2"
0141 EE                 226          OUT    DX,AL
0142 CD4D               227          INT    77                      ;TIMER DELAY FOR LED ON TIME
0144 E8A5FF             228          CALL   RESTOR                  ;ROUTINE TO RESTORE PROCESSOR STATUS
0147 CF                 229          IRET                           ;RETURN FROM INTERRUPT
                        230   ;
                        231   ;
                        232   ;              INTERRUPT 75, IR3 8259A
                        233   ;
0148 E889FF             234   INTR75: CALL   SAVE                    ;ROUTINE TO SAVE PROCESSOR STATUS
014B BAEAFF             235          MOV    DX,0FFEAH               ;8279 COMMAND WORD
014E B082               236          MOV    AL,82H                  ;LED DISPLAY DIGIT 3
0150 A20400             237          MOV    DIGIT,AL
0153 EE                 238          OUT    DX,AL
0154 BAE8FF             239          MOV    DX,0FFE8H               ;8279 DATA
0157 B04F               240          MOV    AL,4FH                  ;CHARACTER "3"
0159 EE                 241          OUT    DX,AL
015A CD4D               242          INT    77                      ;TIMER DELAY FOR LED ON TIME
015C E8CDFF             243          CALL   RESTOR                  ;ROUTINE TO RESTORE PROCESSOR STATUS
015F CF                 244          IRET                           ;RETURN FROM INTERRUPT
                        245   ;
                        246   ;
                        247   ;              INTERRUPT 76, IR4 8259A
                        248   ;
0160 E871FF             249   INTR76: CALL   SAVE                    ;ROUTINE TO SAVE PROCESSOR STATUS
0163 BAEAFF             250          MOV    DX,0FFEAH               ;8279 COMMAND WORD
0166 B083               251          MOV    AL,83H                  ;LED DISPLAY DIGIT 4
0168 A20400             252          MOV    DIGIT,AL
016B EE                 253          OUT    DX,AL
016C BAE8FF             254          MOV    DX,0FFE8H               ;8279 DATA
016F B066               255          MOV    AL,66H                  ;CHARACTER "4"
0171 EE                 256          OUT    DX,AL
0172 CD4D               257          INT    77                      ;TIMER DELAY FOR LED ON TIME
0174 E875FF             258          CALL   RESTOR                  ;ROUTINE TO RESTORE PROCESSOR STATUS
0177 CF                 259          IRET                           ;RETURN FROM INTERRUPT
                        260   ;
                        261   ;
                        262   ;              INTERRUPT 77, TIMER DELAY, SOFTWARE CONTROLLED
                        263   ;
0178 BA0AFF             264   INTR77: MOV    DX,0FF0AH               ;LOAD COUNTER 1 8253 #1 (250 MSEC)
017B B025               265          MOV    AL,25H                  ;LSB
017D EE                 266          OUT    DX,AL
017E B000               267          MOV    AL,00H                  ;MSB
0180 EE                 268          OUT    DX,AL
0181 CF                 269          IRET                           ;RETURN FROM INTERRUPT
```

```
MCS-86 ASSEMBLER    TC159A


LOC  OBJ                LINE   SOURCE

                         270   ;
                         271   ;
----                     272   CODE    ENDS.
                         273   ;
                         274   ;
  0000                   275           END     START

SYMBOL TABLE LISTING
------ ----- -------


NAME    TYPE     VALUE  ATTRIBUTES

??SEG   SEGMENT         SIZE=0400H PARA PUBLIC
AXTEMP  V WORD   0002H  DATA
CODE.   SEGMENT         SIZE=0182H PARA
DATA.   SEGMENT         SIZE=0005H PARA
DIGIT   V BYTE   0004H  DATA
DUMMY   L NEAR   00D2H  CODE
EXTRA   SEGMENT         SIZE=0138H PARA
INTR72  L NEAR   0104H  CODE
INTP72  L NEAR   0118H  CODE
INTR74  L NEAR   0130H  CODE
INTR75  L NEAR   0148H  CODE
INTR76  L NEAR   0160H  CODE
INTR77  L NEAR   0178H  CODE
RESTOR  L NEAR   00ECH  CODE
SAVE.   L NEAR   0004H  CODE
SET531  L NEAR   005AH  CODE
SET532  L NEAR   007SH  CODE
SET59A  L NEAR   009FH  CODE
SET79   L NEAR   00B1H  CODE
STACK1  V WORD   0000H  DATA
STRRT   L NEAR   0000H  CODE
TP72CS  V WORD   0122H  EXTRA
TP72IP  V WORD   0120H  EXTRA
TP73CS  V WORD   0126H  EXTRA
TP73IP  V WORD   0124H  EXTRA
TP74CS  V WORD   012AH  EXTRA
TP74IP  V WORD   0128H  EXTRA
TP75CS  V WORD   012EH  EXTRA
TP75IP  V WORD   012CH  EXTRA
TP76CS  V WORD   0132H  EXTRA
TP76IP  V WORD   0130H  EXTRA
TP77CS  V WORD   0136H  EXTRA
TP77IP  V WORD   0134H  EXTRA
TYPES   L NEAR   0012H  CODE
WAIT79  L NEAR   00B7H  CODE


ASSEMBLY COMPLETE. NO ERRORS FOUND
```

intel®

APPLICATION NOTE

AP-28A

January 1979

Intel® MULTIBUS™ Interfacing

Joe Barthmaier
OEM Microcomputer
Systems Applications

© Intel Corporation, 1979.

9800587C

Related Intel Publications

MCS-80™ User's Manual, 98-153D

MCS-85™ User's Manual, 98-366C.

MCS-86™ User's Manual, 9800722A.

iSBC 80/20 and iSBC 80/20-4 Single Board Computer Hardware Reference Manual, 98-317C.

iSBC™ 86/12 Single Board Computer Hardware Reference Manual, 9800645A.

Intel® Multibus™ Specification, 9800683.

Intel® MULTIBUS™ Interfacing

Contents

I. INTRODUCTION

II. MULTIBUS™ SYSTEM BUS DESCRIPTION

OVERVIEW

MULTIBUS™ SIGNAL DESCRIPTIONS

OPERATING CHARACTERISTICS

MULTIBUS™ SLAVE INTERFACE CIRCUIT ELEMENTS

III. MULTIBUS™ SLAVE DESIGN EXAMPLE

FUNCTIONAL/PROGRAMMING CHARACTERISTICS

THEORY OF OPERATION

IV. SUMMARY

APPENDIX A
MULTIBUS™ PIN ASSIGNMENTS

APPENDIX B
BUS TIMING SPECIFICATIONS

APPENDIX C
BUS DRIVERS, RECEIVERS, AND TERMINATIONS

APPENDIX D
BUS POWER SUPPLY SPECIFICATIONS

APPENDIX E
MECHANICAL SPECIFICATIONS

APPENDIX F
MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC, 8/16-BIT VERSION

APPENDIX G
MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC, 8-BIT VERSION

I. INTRODUCTION

A significant measure of the power and flexibility of the Intel OEM Computer Product Line can be attributed to the design of the Intel MULTIBUS system bus. The bus structure provides a common element for communication between a wide variety of system modules which include: Single Board Computers, memory, digital, and analog I/O expansion boards, and peripheral controllers.

The purpose of this application note is to help you develop a working knowledge of the Intel MULTIBUS specification. This knowledge is essential for configuring a system containing multiple modules. Another purpose is to provide you with the information necessary to design a bus interface for a slave module. One of the tools that will be used to achieve this goal is the complete description of a MULTIBUS slave design example. Other portions of this application note provide an in depth examination of the bus signals, operating characteristics, and bus interface circuits.

This application note was originally written in 1977. Since 1977, the MULTIBUS specification has been significantly expanded to cover operation with both 8 and 16-bit system modules and with an auxiliary power bus. This application note now contains information on these new MULTIBUS specification features.

In addition, a detailed MULTIBUS specification has also been published which provides the user with further information concerning MULTIBUS interfacing. The MULTIBUS specification and other useful documents are listed in the overleaf of this note under Related Intel Publications.

II. MULTIBUS™ SYSTEM BUS DESCRIPTION

Overview

The Intel MULTIBUS signal lines can be grouped in the following categories: 20 address lines, 16 bidirectional data lines, 8 multilevel interrupt lines, and several bus control, timing and power supply lines. The address and data lines are driven by three-state devices, while the interrupt and some other control lines are open-collector driven.

Modules that use the MULTIBUS system bus have a master-slave relationship. A bus master module can drive the command and address lines: it can control the bus. A Single Board Computer is an example of a bus master. A bus slave cannot control the bus. Memory and I/O expansion boards are examples of bus slaves. The MULTIBUS architecture provides for both 8 and 16-bit bus masters and slaves.

Notice that a system may have a number of bus masters. Bus arbitration results when more than one master requests control of the bus at the same time. A bus clock is usually provided by one of the bus masters and may be derived independently from the processor clock. The bus clock provides a timing reference for resolving bus contention among multiple requests from bus masters. For example, a processor and a DMA (direct memory access) module may both request control of the bus. This feature allows different speed masters to share resources on the same bus. Actual transfers via the bus, however, proceed asynchronously with respect to the bus clock. Thus, the transfer speed is dependent on the transmitting and receiving devices only. The bus design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. Once a bus request is granted, single or multiple read/write transfers can proceed. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations and high-speed direct-memory-access (DMA) operations. However, the master-slave capabilities of the bus are by no means limited to these two applications.

MULTIBUS™ Signal Descriptions

This section defines the signal lines that comprise the Intel MULTIBUS system bus. These signals are contained on either the P1 or P2 connector of boards compatible with the MULTIBUS specification. The P1 signal lines contain the address, data, bus control, bus exchange, interrupt and power supply lines. The P2 signal lines contain the optional auxiliary signal lines. Most signals on the bus are active-low. For example, a low level on a control signal on the bus indicates active, while a low level on an address or data signal on the bus represents logic "1" value.

NOTE

In this application note, a signal will be designated active-low by placing a slash (/) after the mnemonic for the signal.

Appendix A contains a pin assignment list of the following signals:

MULTIBUS P1 Signal Lines —

Initialization Signal Line

INIT/

Initialization signal; resets the entire system to a known internal state. INIT/ may be driven by one of the bus masters or by an external source such as a front panel reset switch.

Address and Inhibit Lines

ADR0/ - ADR13/

20 address lines; used to transmit the address of the memory location or I/O port to be accessed. The lines are labeled ADR0/ through ADR9/, ADRA/ through ADRF/ and ADR10/ through ADR13/. ADR13/ is the most significant bit. 8-bit masters use 16 address lines (ADR0/ - ADRF/) for memory addressing and 8 address lines (ADR0/ - ADR7/) for I/O port selection. 16-bit masters use all twenty address lines for memory addressing and 12 address lines (ADR0/ - ADRB/) for I/O port selection. Thus, 8-bit masters may address 64K bytes of memory and 256 I/O devices while 16-bit masters may address 1 megabyte of memory and 4096 I/O devices. (The 8086 CPU actually permits 16 address bits to be used to specify I/O devices, the MULTIBUS specification, however, states that only the low order 12 address bits can be used to specify I/O ports.) In a 16-bit system, the ADR0/ line is used to indicate whether a low (even) byte or a high (odd) byte of memory or I/O space is being accessed in a word oriented memory or I/O device.

BHEN/

Byte High Enable; the address control line which is used to specify that data will be transferred on the high byte (DAT8/ - DATF/) of the MULTIBUS data lines. With current iSBC boards, this signal effectively specifies that a word (two byte) transfer is to be performed. This signal is used only in systems which incorporate sixteen bit memory or I/O modules.

INH1/

Inhibit RAM signal; prevents RAM memory devices from responding to the memory address on the system address bus. INH1/ effectively allows ROM memory devices to override RAM devices when ROM and RAM memory are assigned the same memory addresses. INH1/ may also be used to allow memory mapped I/O devices to override RAM memory.

INH2/

Inhibit ROM signal; prevents ROM memory devices from responding to the memory address on the system address bus. INH2/ effectively allows auxiliary ROM (e.g., a bootstrap program) to override ROM devices when ROM and auxiliary ROM memory are assigned the same memory addresses. INH2/ may also be used to allow memory mapped I/O devices to override ROM memory.

Data Lines

DAT0/ - DATF/

16 bidirectional data lines; used to transmit or receive information to or from a memory location or I/O port. DATF/ being the most significant bit. In 8-bit systems, only lines DAT0/ - DAT7/ are used (DAT7/ being the most significant bit). In 16-bit systems, either 8 or 16 lines may be used for data transmission.

Bus Priority Resolution Lines

BCLK/

Bus clock; the negative edge (high to low) of BCLK/ is used to synchronize bus priority resolution circuits. BCLK/ is asynchronous to the CPU clock. It has a 100 ns minimum period and a 35% to 65% duty cycle. BCLK/ may be slowed, stopped, or single stepped for debugging.

CCLK/

Constant clock; a bus signal which provides a clock signal of constant frequency for unspecified general use by modules on the system bus. CCLK/ has a minimum period of 100 ns and a 35% to 65% duty cycle.

BPRN/

Bus priority in signal; indicates to a particular master module that no higher priority module is requesting use of the system bus. BPRN/ is synchronized with BCLK/. This signal is not bused on the backplane.

chain/ bus priority resolution schemes. BPRO/ is passed to the BPRN/ input of the master module with the next lower bus priority. BPRO/ is synchronized with BCLK/. This signal is not bused on the backplane.

BUSY/

Bus busy signal; an open collector line driven by the bus master currently in control to indicate that the bus is currently in use. BUSY/ prevents all other master modules from gaining control of the bus. BUSY/ is synchronized with BCLK/.

BREQ/

Bus request signal; used with a parallel bus priority network to indicate that a particular master module requires use of the bus for one or more data transfers. BREQ/ is synchronized with BCLK/. This signal is not bused on the backplane.

CBRQ/

Common bus request; an open-collector line which is driven by all potential bus masters and is used to inform the current bus master that another master wishes to use the bus. If CBRQ/ is high, it indicates to the bus master that no other master is requesting the bus, and therefore, the present bus master can retain the bus. This saves the bus exchange overhead for the current master.

Information Transfer Protocol Lines

A bus master provides separate read/write command signals for memory and I/O devices: MRDC/, MWTC/, IORC/ and IOWC/, as explained below. When a read/write command is active, the address signals must be stabilized at all slaves on the bus. For this reason, the protocol requires that a bus master must issue address signals (and data signals for a write operation) at least 50 ns ahead of issuing a read/write command to the bus, initiating the data transfer. The bus master must keep address signals unchanged until at least 50 ns after the read/write command is turned off, terminating the data transfer.

A bus slave must provide an acknowledge signal to

MRDC/

Memory read command; indicates that the address of a memory location has been placed on the system address lines and specifies that the contents (8 or 16 bits) of the addressed location are to be read and placed on the system data bus. MRDC/ is asynchronous with respect to BCLK/.

MWTC/

Memory write command; indicates that the address of a memory location has been placed on the system address lines and that data (8 or 16 bits) has been placed on the system data bus. MWTC/ specifies that the data is to be written into the addressed memory location. MWTC/ is asynchronous with respect to BCLK/.

IORC/

I/O read command; indicates that the address of an input port has been placed on the system address bus and that the data (8 or 16 bits) at that input port is to be read and placed on the system data bus. IORC/ is asynchronous with respect to BCLK/.

IOWC/

I/O write command; indicates that the address of an output port has been placed on the system address bus and that the contents of the system data bus (8 or 16 bits) are to be output to the address port. IOWC/ is asynchronous with respect to BCLK/.

XACK/

Transfer acknowledge signal; the required response of a slave board which indicates that the specified read/write operation has been completed. That is, data has been placed on, or accepted from, the system data bus lines. XACK/ is asynchronous with respect to BCLK/.

Asynchronous Interrupt Lines

INT0/ - INT7/

8 Multi-level, parallel interrupt request lines;

used with a parallel interrupt resolution network. INT0 has the highest priority, while INT7 has lowest priority. Interrupt lines should be driven with open collector drivers.

INTA

Interrupt acknowledge; an interrupt acknowledge line (INTA), driven by the bus master, requests the transfer of interrupt information onto the bus from slave priority interrupt controllers (8259s or 8259As). The specific information timed onto the bus depends upon the implementation of the interrupt scheme. In general, the leading edge of INTA indicates that the address bus is active while the trailing edge indicates that data is present on the data lines.

MULTIBUS P2 Signal Lines — The signals contained on the MULTIBUS P2 auxiliary connector are used primarily by optional power back-up circuitry for memory protection. P2 signals are not bused on the backplane, and therefore, require a separate connector for each board using the P2 signals. Present iSBC boards have a slot in the card edge and should be used with a keyed P2 edge connector. Use of the P2 signal lines is optional.

ACLO

AC Low; this signal generated by the power supply goes high when the AC line voltage drops below a certain voltage (e.g., 103v AC in 115v AC line voltage systems) indicating D.C. power will fail in 3 msec. ACLO goes low when all D.C. voltages return to approximately 95% of the regulated value. This line must be pulled up by the optional standby power source, if one is used.

PFIN

Power fail interrupt; this signal interrupts the processor when a power failure occurs, it is driven by external power fail circuitry.

PFSN/

Power fail sense; this line is the output of a latch which indicates that a power failure has occurred. It is reset by PFSR . The power fail sense latch is part of external power fail circuitry and must be powered by the standby power source.

PFSR

Power fail sense reset; this line is used to reset the power fail sense latch (PFSN).

MPRO

Memory protect; prevents memory operation during period of uncertain DC power, by inhibiting memory requests. MPRO is driven by external power fail circuitry.

ALE

Address latch enable; generated by the CPU (8085 or 8086) to provide an auxiliary address latch.

HALT/

Halt; indicates that the master CPU is halted.

AUX RESET/

Auxiliary Reset; this externally generated signal initiates a power-up sequence.

WAIT/

Bus master wait state; this signal indicates that the processor is in a wait state.

Reserved — Several P1 and P2 connector bus pins are unused. However, they should be regarded as reserved for dedicated use in future Intel products.

Power Supplies — The power supply bus pins are detailed in Appendix A which contains the pin assignment of signals on the MULTIBUS backplane.

It is the designer's responsibility to provide adequate bulk decoupling on the board to avoid current surges on the power supply lines. It is also recommended that you provide high frequency

decoupling for the logic on your board. Values of 22uF for +5v and +12v pins and 10uF for -5v and -12v pins are typical on iSBC boards.

Operating Characteristics

Beyond the definition of the MULTIBUS signals themselves, it is important to examine the operating characteristics of the bus. The AC requirements outline the timing of the bus signals and in particular, define the relationships between the various bus signals. On the other hand, the DC requirements specify the bus driver characteristics, maximum bus loading per board, and the pull-up/down resistors.

The AC requirements are best presented by a discussion of the relevant timing diagrams. Appendix B contains a list of the MULTIBUS timing specifications. The following sections will discuss data transfers, inhibit operations, interrupt operations, MULTIBUS multi-master operation and power fail considerations.

Data Transfers — Data transfers on the MULTIBUS system bus occur with a maximum bandwidth of 5 MHz for single or multiple read/write transfers. Due to bus arbitration and memory access time, a typical maximum transfer rate is often on the order of 2 MHz.

Read Data

Figure 1 shows the read operation AC timing diagram. The address must be stable (t_{AS}) for a minimum of 50 ns before command (IORC/ or MRDC/). This time is typically used by the bus interface to decode the address and thus provide the required device selects. The device selects establish the data paths on the user system in anticipation of the strobe signal (command) which will follow. The minimum command pulse width is 100 ns. The address must remain stable for at least 50 ns following the command (t_{AH}). Valid data should not be driven onto the bus prior to command, and must not be removed until the command is cleared. The XACK/ signal, which is a response indicating the specified read/write operation has been completed, must coincide or follow both the read access and valid data (t_{DXL}). XACK/ must be held until the command is cleared (t_{XAH}).

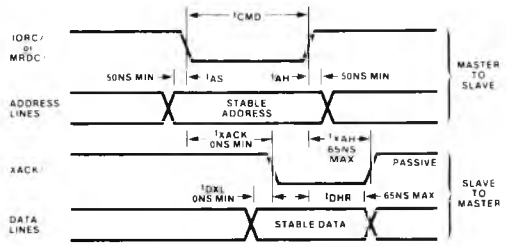


Figure 1. Read AC Timing

Write Data

The write operation AC timing diagram is shown in Figure 2. During a write data transfer, valid data must be presented simultaneously with a stable address. Thus, the write data setup time (t_{DS}) has the same requirement as the address setup time (t_{AS}). The requirement for stable data both before and after command (IOWC/ or MWTC/) enables the bus interface circuitry to latch data on either the leading or trailing edge of command.

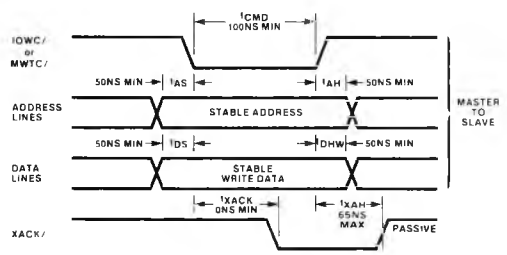


Figure 2. Write AC Timing

Data Byte Swapping in 16-bit Systems

A 16-bit master may transfer data on the MULTIBUS data lines using 8-bit or 16-bit paths depending on whether a byte or word (2 byte) operation has been specified. (A word transfer specified with an odd I/O or memory address will actually be executed as two single byte transfers.) An 8-bit master may only perform byte transfers on the MULTIBUS data lines DAT0/ - DAT7/.

In order to maintain compatibility with older 8-bit masters and slaves, a byte swapping buffer is included in all new 16-bit masters and 16-bit slaves. In the iSBC product line, all byte transfers will take place on the low 8 data lines DAT0/ - DAT7/. Figure 3 contains a example of 8/16-bit

data driver logic for 16-bit master and slave systems. In the 8/16-bit system, there are three sets of buffers; the lower byte buffer which accesses DAT0/ - DAT7/, the upper byte buffer which accesses DAT8/ - DATF/, and the swap byte buffer which accesses the MULTIBUS data lines DAT0/ - DAT7/ and transfers the data to/from the on-board data bus lines D8 - DF.

Figure 4 summarizes the 8 and 16-bit data paths used for three types of MULTIBUS transfers. Two signals control the data transfers.

Byte High Enable (BHEN/) active indicates that the bus is operating in sixteen bit mode, and Address Bit 0 (ADR0/) defines an even or odd byte transfer address.

On the first type of transfer, BHEN/ is inactive, and ADR0/ is inactive indicating the transfer of an even eight bit byte. The transfer takes place across data lines DAT0/ - DAT7/.

On the second type of transfer, BHEN/ is inactive, and ADR0/ is active indicating the transfer of a high (odd) byte. On this type of transfer, the odd (high) byte is transferred through the Swap Byte Buffer to DAT0/ - DAT7/. This makes eight bit and sixteen bit systems compatible.

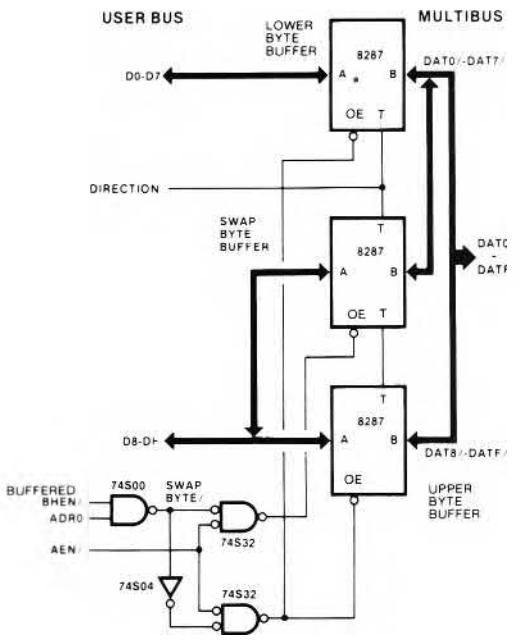


Figure 3. 8/16-Bit Data Drivers

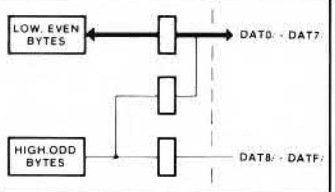
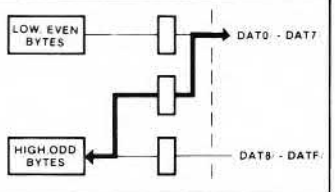
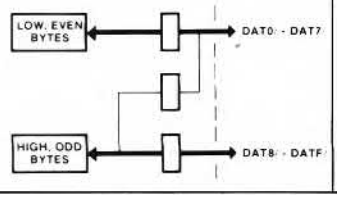
16-BIT DEVICE	MULTIBUS	BHEN/	ADR0/	MULTIBUS TRANSFER DATA PATH	DEVICE BYTE TRANSFERRED
		H	H	8-BIT, DAT0/ - DAT7/	EVEN
		H	L	8-BIT, DAT0/ - DAT7/	ODD
		L	H	16-BIT, DAT0/ - DATF/	EVEN AND ODD

Figure 4. 8/16-Bit Device Transfer Operation

transfer, the low (even) byte is transferred on DAT0/ - DAT7/ and the high (odd) byte is transferred on DAT8/ - DATF/.

Note that the condition when both BHEN and ADR0 are active is not used with present iSBC boards. This condition could be used to transfer a high odd byte of data on DAT8 - DATF , thus eliminating the need for the swap byte buffer. However, this is not a recommended transfer type, because it eliminates the capability of communicating with 8-bit modules.

Inhibit Operations — Bus inhibit operations are required by certain bootstrap and memory mapped I/O configurations. The purpose of the inhibit operation is to allow a combination of RAM, ROM, or memory mapped I/O to occupy the same memory address space. In the case of a bootstrap, it may be desirable to have both ROM and RAM memory occupy the same address space, selecting ROM instead of RAM for low order memory only when the system is reset. A system designed to use space, may need to inhibit RAM or ROM memory to perform its functions.

There are two essential requirements for a successful inhibit operation. The first is that the inhibit signal must be asserted as soon as possible, within a maximum of 100 ns (t_{CI}), after stable address. The second requirement for a successful inhibit operation is that the acknowledge must be delayed (t_{XACKB}) to allow the inhibited slave to terminate any irreversible timing operations initiated by detection of a valid command prior to its inhibit.

This situation may arise because a command can be asserted within 50 ns after stable address (t_{AS}) and yet inhibit is not required until 100 ns (t_{ID}) after stable address. The acknowledge delay time (t_{XACKB}) is a function of the cycle time of the inhibited slave memory. Inhibiting the iSBC 016 RAM board, for example, requires a minimum of 1.5 usec. Less time is typically needed to inhibit other memory modules. For example, the iSBC 104 board requires 475 ns.

Figure 5 depicts a situation in which both RAM

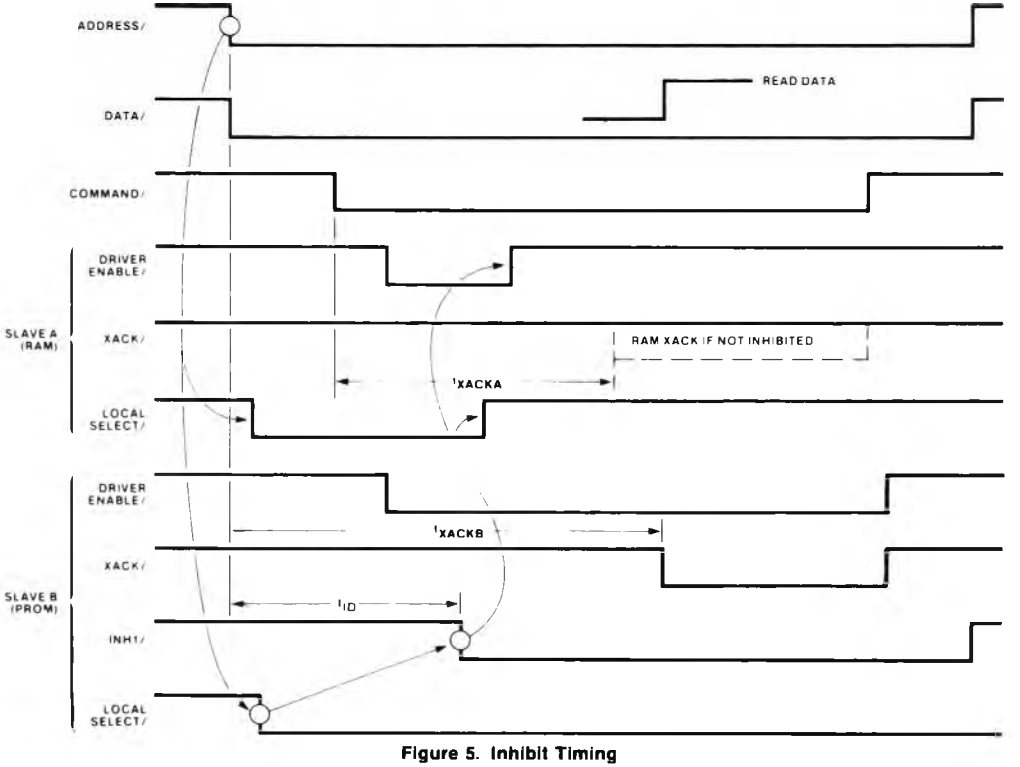


Figure 5. Inhibit Timing

and PROM memory have the same memory addresses. In this case, PROM inhibits RAM, producing the effect of PROM overriding RAM. After address is stable, local selects are generated for both the PROM and the RAM. The PROM local select produces the INH1/ signal which then removes the RAM local select and its driver enable. Because the slave RAM has been inhibited after it had already begun its cycle, the PROM XACK/ must be delayed (t_{XACKB}) until after the latest possible acknowledgement from the RAM (t_{XACKA}).

Interrupt Operations — The MULTIBUS interrupt lines INT0/ - INT7/ are used by a MULTIBUS master to receive interrupts from bus slaves, other bus masters or external logic such as power fail logic. A bus master may also contain internal interrupt sources which do not require the bus interrupt lines to interrupt the master. There are two interrupt implementation schemes used by bus interrupts, Non Bus Vectored Interrupts and Bus Vectored Interrupts. Non Bus Vectored Interrupts do not convey interrupt vector address information on the bus. Bus Vectored Interrupts are interrupts from slave Priority Interrupt Controllers (PICs) which do convey interrupt vector address information on the bus.

Non Bus Vectored Interrupts

Non Bus Vectored Interrupts are those interrupts whose interrupt vector address is generated by the bus master and do not require the MULTIBUS address lines for transfer of the interrupt vector address. The interrupt vector address is generated by the interrupt controller on the master and transferred to the processor over the local bus. The source of the interrupt can be on the master module or on other bus modules, in which case the bus modules use the MULTIBUS interrupt request lines (INT0/ - INT7/) to generate their interrupt requests to the bus master. When an interrupt request line is activated, the bus master performs it own interrupt operation and processes the interrupt. Figure 6 shows an example of Non Bus Vectored Interrupt implementation.

Bus Vectored Interrupts

Bus Vectored Interrupts (Figure 7) are those interrupts which transfer the interrupt vector address along the MULTIBUS address lines from the slave to the bus master using the INTA/ command signal for synchronization.

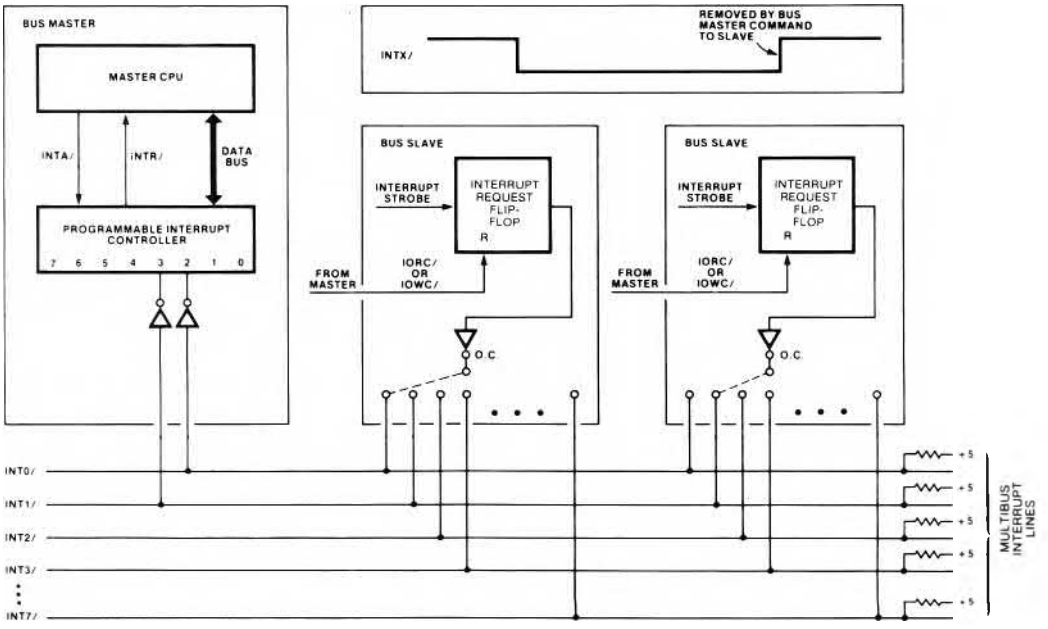


Figure 6. Non Bus Vectored Interrupt Implementation

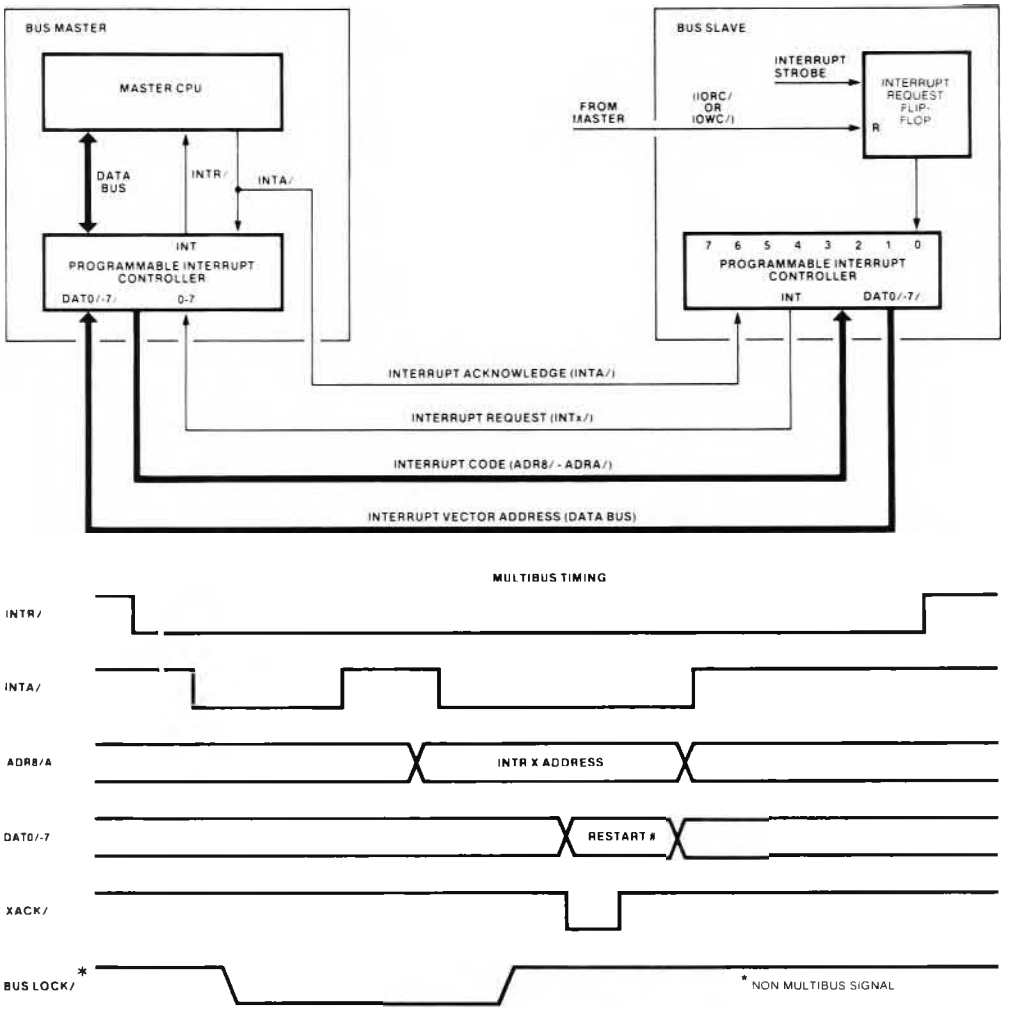


Figure 7. Bus Vectored Interrupt Logic (With 2 INTA/ Timing Diagram)

When an interrupt request from the MULTIBUS interrupt lines INT0/ - INT7/ occurs, the interrupt control logic on the bus master interrupts its processor. The processor on the bus master generates an INTA/ command which freezes the state of the interrupt logic on the MULTIBUS slaves for priority resolution. The bus master also locks (retains the bus between bus cycles) the MULTIBUS control lines to guarantee itself consecutive bus cycles. After the first INTA/ command, the bus master's interrupt control logic puts an interrupt code on to the MULTIBUS address lines ADR8/ - ADRA/. The interrupt code is the address of the highest priority active interrupt request line. At this point in the Bus Vectored Interrupt procedure, two different sequences could take place. The difference occurs, because the MULTIBUS specification can support masters which generate one additional INTA/ (8086 masters) or two additional INTA/s (8080A and 8085 masters).

If the bus master generates one additional INTA/, this second INTA/ causes the bus slave interrupt control logic to transmit an interrupt vector 8-bit pointer on the MULTIBUS data lines. The vector pointer is used by the bus master to determine the memory address of the interrupt service routine.

If the bus master generates two additional INTA/s, these two INTA/ commands allow the

bus slave to put a two byte interrupt vector address on to the MULTIBUS data lines (one byte for each INTA/). The interrupt vector address is used by the bus master to service the interrupt.

The MULTIBUS specification provides for only one type of Bus Vectored Interrupt operation in a given system. Slave boards which have an 8259 interrupt controller are only capable of 3 INTA/ operation (2 additional INTA/s after the first INTA/). Slave boards with the 8259A interrupt controller are capable of either 2 INTA/ or 3 INTA/ operation. All slave boards in a given system must operate in the same way (2 INTA/s or 3 INTA/s) if Bus Vectored Interrupts are to be used. However, the MULTIBUS specification does provide for Bus Vectored Interrupts and Non Bus Vectored Interrupts in the same system.

MULTIBUS Multi-Master Operation — The MULTIBUS system bus can accommodate several bus masters on the same system, each one taking control of the bus as it needs to affect data transfers. The bus masters request bus control through a bus exchange sequence.

Two bus exchange priority resolution techniques are discussed, a serial technique and a parallel technique. Figures 8 and 9 illustrate these two techniques. The bus exchange operation discussed later is the same for both techniques.

Serial Priority Technique

Serial priority resolution is accomplished with a daisy chain technique (see Figure 8). The priority input (BPRN/) of the highest priority master is tied to ground. The priority output (BPRO/) of the highest priority master is then connected to the priority input (BPRN/) of the next lower priority master, and so on. Any master generating a bus request will set its BPRO/ signal high to the next lower priority master. Any master seeing a high signal on its BPRN/ line will sets its BPRO/ line high, thus passing down priority information to lower priority masters. In this implementation, the bus request line (BREQ/) is not used outside of the individual masters. A limited number of masters can be accommodated by this technique, due to gate delays through the daisy chain. Using the current Intel MULTIBUS controller chip on the master boards up to 3 masters may be accommodated if a BCLK/ period of 100 ns is used. If more bus masters are required, either BCLK/ must be slowed or a parallel priority technique used.

Parallel Priority Technique

In the parallel priority technique, the priority is resolved in a priority resolution circuit in which the highest priority BREQ/ input is encoded with a priority encoder chip (74148). This coded value is then decoded with a priority decoder chip (74S138) to activate the appropriate BPRN/ line. The BPRO/ lines are not used in the parallel priority scheme. However, since the MULTIBUS backplane contains a trace from the BPRN/ signal of one card slot to the BPRO/ signal of the adjacent lower card slot, the BPRO/ must be disconnected from the bus on the board or the backplane trace must be cut. A practical limit of sixteen masters can be accommodated using the parallel priority technique due to physical bus length limitations. Figure 9 contains the schematic for a typical parallel resolution network. Note that the parallel priority resolution network must be externally supplied.

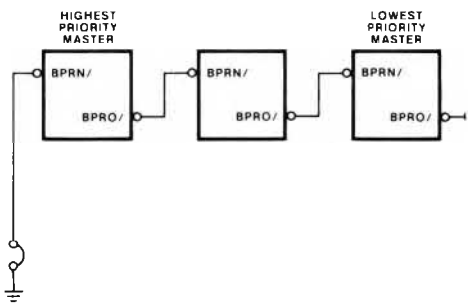


Figure 8. Serial Priority Technique

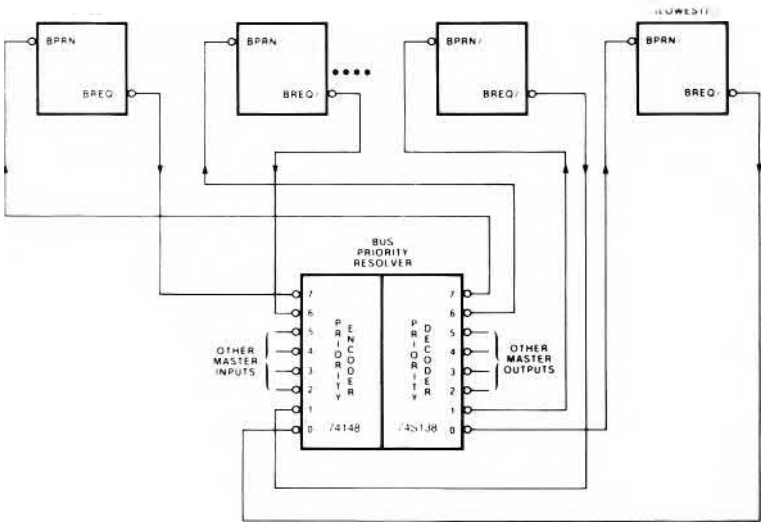


Figure 9. Parallel Priority Technique

MULTIBUS Exchange Operation — A timing diagram for the MULTIBUS exchange operation is shown in Figure 10. This implementation example uses a parallel resolution scheme, however, the timing would be basically the same for the serial resolution scheme.

In this example, master A has been assigned a lower priority than master B. The bus exchange occurs because master B generates a bus request during a time when master A has control of the bus.

The exchange process begins when master B requires the bus to access some resource such as an I/O or memory module while master A controls the bus. This internal request is synchronized with the trailing edge (high to low) of BCLK/ to generate a bus request (BREQ/). The bus priority resolution circuit changes the BPRN/ signal from active (low) to inactive (high) for master A and from inactive to active for master B. Master A must first complete the current bus command if one is in operation. After master A completes the command, it sets BUSY/ inactive on the next trailing edge of BCLK/. This allows the actual bus exchange to occur, because master A has relinquished control of the bus, and master B has been granted its BPRN/. During this time, the drivers for master A are disabled. Master B must take control of the bus with the next trailing edge of BCLK/ to complete the bus exchange. Master B takes control by activating BUSY/ and enabling its drivers.

It is possible for master A to retain control of the bus and prevent master B from getting control. Master A activates the Bus Override (or Bus Lock) signal which keeps BUSY/ active allowing control of the bus to stay with master A. This guarantees a master consecutive bus cycles for software or hardware functions which require exclusive, continuous access to the bus.

Note that in systems with only a single master it is necessary to ground the BPRN/ pin of the master, if slave boards are to be accessed. In single board systems which use a CPU board capable of Bus Vectored Interrupt operation, the BPRN/ pin must also be grounded.

In a single master system bus transfer efficiency may be gained if the BUS OVERRIDE signal is kept active continuously. This permits the master to maintain control of the bus at all times, therefore saving the overhead of the master reacquiring the bus each time it is needed.

The CBRQ/ line may be used by a master in control of the bus to determine if another master

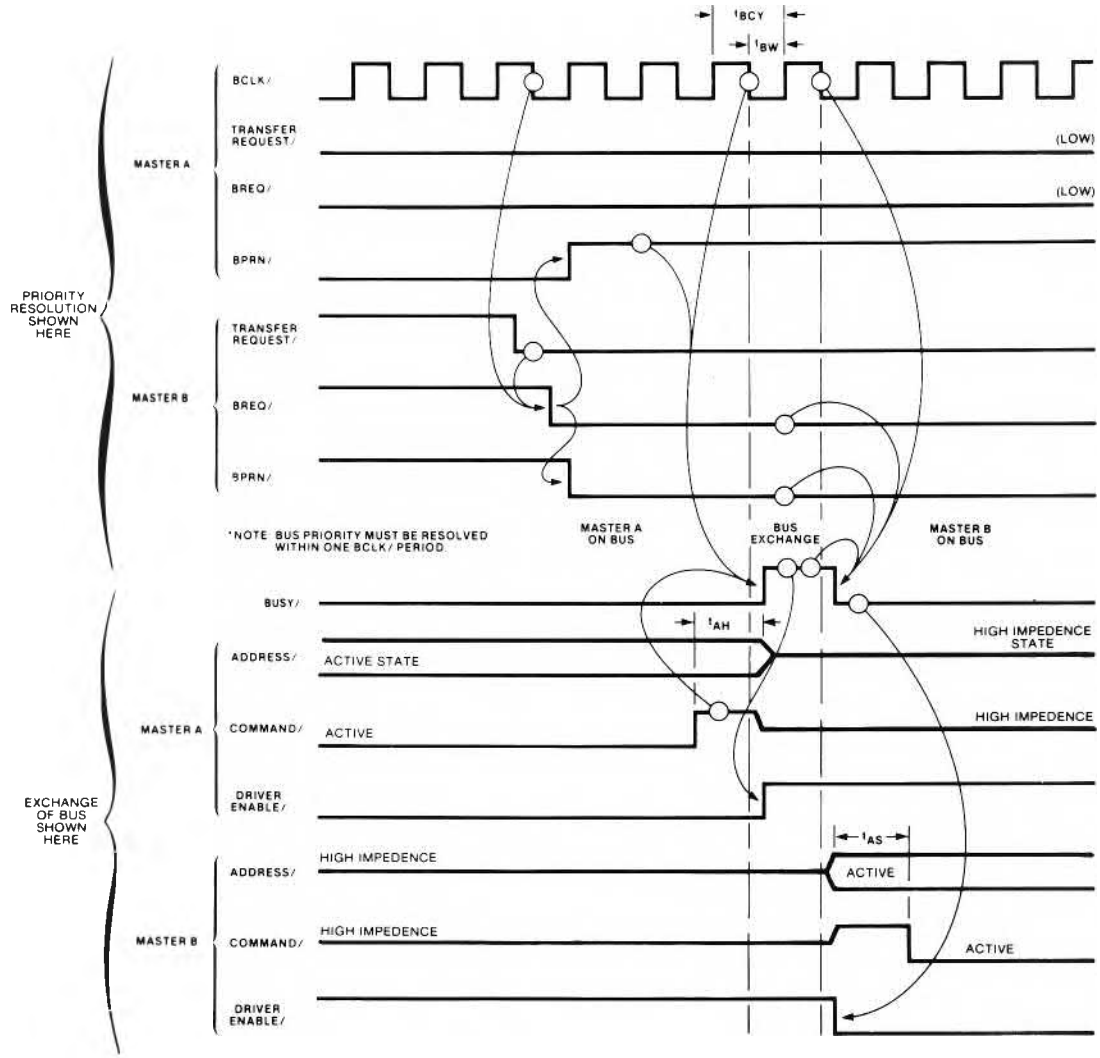


Figure 10. Bus Control Exchange Operation

requires the bus. If a master currently in control of the bus sees the CBRQ/ line inactive, it will maintain control of the bus between adjacent bus accesses. Therefore, when a bus access is required, the master saves the overhead of reacquiring the bus. If a current bus master sees the CBRQ/ line active, it will then relinquish control of the bus after the current bus access and will contend for the bus with the other master(s) requiring the bus. The relative priorities of the masters will determine which master receives the bus.

Note that except for the BUS OVERRIDE state, no single master may keep exclusive control of the bus. This is true because it is impossible for the CPU on a master to require continuous access to the bus. Other lower priority masters will always be able to gain access to the bus between accesses of a higher priority master.

Power Fail Considerations — The MULTIBUS P2 connector signals provide a means of handling power failures. The circuits required for power

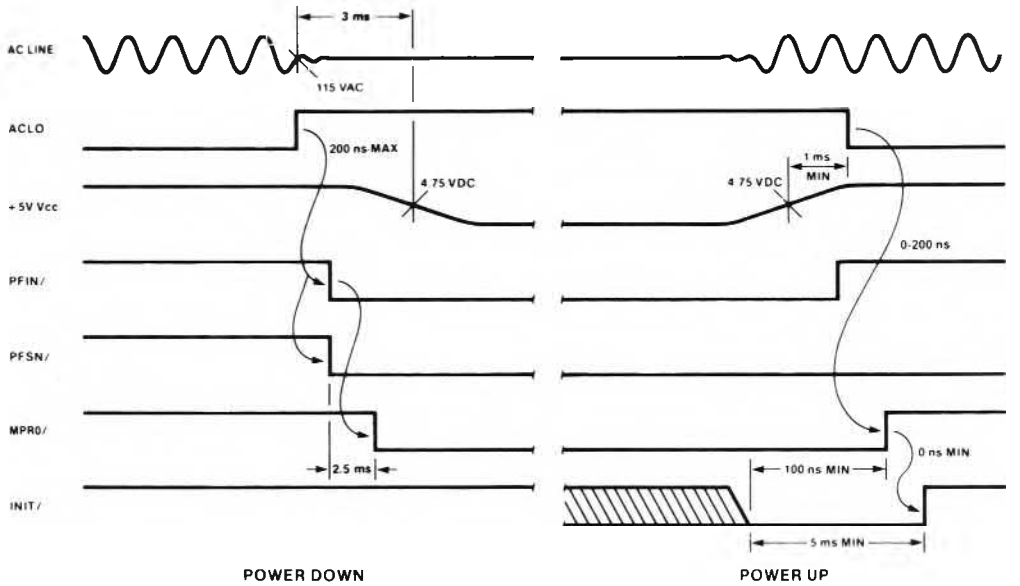


Figure 11. Power Fail Timing Sequence

failure detection and handling are optional and must be supplied by the user. Figure 11 shows the timing of a power fail sequence.

The power supply monitors the AC power level. When power drops below an acceptable value, the power supply raises ACLO which tells the power fail logic that a minimum of three milliseconds will elapse before DC power will fall below regulated voltage levels. The power fail logic sets a sense latch (PFSN/) and generates an interrupt (PFIN/) to the processor so the processor can store its environment. After a 2.5 millisecond timeout, the memory protect signal (MPRO/) is asserted by the power fail logic preventing any memory activity. As power falls, the memory goes on standby power. Note that the power fail logic must be powered from the standby source.

As the AC line revives, the logic voltage level is monitored by the power supply. After power has been at its operating level for one millisecond minimum, the power supply sets the signal ACLO low, beginning the restart sequence. First, the memory protect line (MPRO/) then the initialize line (INIT/) become inactive. The bus master now starts running. The bus master checks the power fail latch (PFSN/) and, if it finds it set, branches to a power up routine which resets the latch (PFSR/), restores the environment, and resumes execution.

Note that INIT/ is activated only after DC power has risen to the regulated voltage levels and must stay low for five milliseconds minimum before the system is allowed to restart. Alternatively, INIT/ may be held low through an open collector device by MPRO/.

How the power failure equipment is configured is left to the system designer. The backup power source may be batteries located on the memory boards or more elaborate facilities located off-board. The location of the power fail logic determines which MULTIBUS power fail lines are used. Pins on the P2 connector have been specified for the power failure functions for use as needed.

To further clarify the location and use of the power fail circuitry, an example of a typical power fail system block diagram is shown in Figure 12. A single board computer and a slave memory board are contained in the system. It is desired to power the memory circuit elements of the memory board from auxiliary power. The single board computer will remain on the main power supply. To accomplish this, user supplied power fail logic and

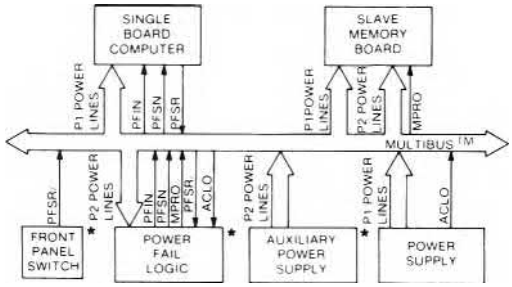


Figure 12. Typical Power Fail System Block Diagram

an auxiliary power supply have been included in the system.

The single board computer is powered from the P1 power lines and accesses the P2 signal lines PFIN/, PFSN/ and PFSR/ (only the P2 signal lines used by a particular functional block are shown on the block diagram). The PFSR/ line is driven from two sources: a front panel switch and the single board computer. The front panel switch is used during normal power-up to reset the power fail sense latch. The single board computer uses the PFSR/ line to reset the latch during a power-up sequence after a power failure. Current single board computers must access the PFSN/ and PFSR/ signals either directly with dedicated circuitry and a P2 pin connection or through the parallel I/O lines with a cable connection from the parallel I/O connector to the P2 connector.

The slave memory board uses both the P1 and P2 power lines, the P2 power lines are used (at all times) to power the memory circuit elements and other support circuits, the P1 power lines power all other circuitry. In addition, the MPRO/ line is input and used to sense when memory contents should be protected.

The power fail logic contains the power fail sense latch, and uses the PFSR/ and ACLO lines for inputs and the PFIN/ PFSN/, and MPRO/ lines for outputs. The power fail logic must be powered by the P2 power lines.

DC Requirements — The drive and load characteristics of the bus signals are listed in Appendix C. The physical locations of the drivers and loads, as well as the terminating resistor value for each bus line, are also specified. Appendix D contains the MULTIBUS power specifications.

MULTIBUS™ Slave Interface Circuit Elements

There are three basic elements of a slave bus interface: address decoders, bus drivers, and control signal logic. This section discusses each of these elements in general terms. A description of a detailed implementation of a slave interface is presented in a later section of this application note.

Address Decoding — This logic decodes the appropriate MULTIBUS address bits into RAM requests, ROM requests, or I/O selects. Care must be taken in the design of the address decode logic to ensure flexibility in the selection of base address assignments. Without this flexibility, restrictions may be placed upon various system configurations. Ideally, switches and jumper connections should be associated with the decode logic to permit field modification of base address assignments.

The initial step in designing the address decode portion of a MULTIBUS interface is to determine the required number of unique address locations. This decision is influenced by the fact that address decoding is usually done in two stages. The first stage decodes the base address, producing an enable for the second stage which generates the actual device selects for the user logic. A convenient implementation of this two stage decoding scheme utilizes a pair of decoders driven by the high order bits of the address for the first stage and a second decoder for the low order bits of the address bus. This technique forces the number of unique address locations to be a power of two, based at the address decoded by the first stage. Consider the scheme illustrated in Figure 13.

As shown in Figure 13, the address bits A_4 - A_B are used to produce switch selected outputs of the first stage of decoding. The 1 out of 8 binary decoders

have been used. The top decoder decodes address lines A_4 - A_7, and the bottom decoder decodes address lines A_8 - A_B. If only address lines A_0 - A_7 are being used for device selection, as in the case of I/O port selection in 8-bit systems, the bottom decoder may be disabled by setting switch S2 to the ground position. Address lines A_7 and A_B drive enable inputs E2 or E3 of the decoders. The address lines A_0 - A_3 enter the second stage address decoder to produce 8 user device selects. The second stage decoder must first be enabled by an address that corresponds to the switch-selected base address.

Address decoding must be completed before the arrival of a command. Since the command may become active within 50 ns after stable address, the decode logic should be kept simple with a minimal number of layers of logic. Furthermore, the timing is extremely critical in systems which make use of the inhibit lines.

A linear or unary select scheme in which no binary encoding of device address (e.g., address bit A_0 selects device 0, address bit A_1 selects device 1, etc.) is performed is not recommended because the scheme offers no protection in case multiple devices are simultaneously selected, and because the addressing within such a system is restricted by the extent of the address space occupied by such a scheme.

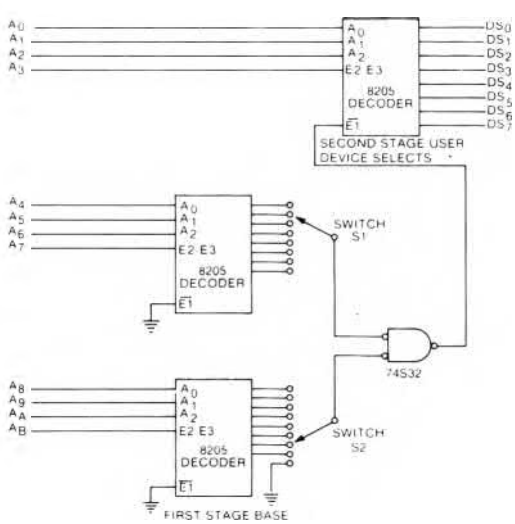


Figure 13. Two Stage Decoding Scheme

Data Bus Drivers — For user designed logic which simply receives data from the MULTIBUS data lines, this portion of the bus interface logic may only consist of buffers. Buffers are required to ensure that maximum allowable bus loading is not exceeded by the user logic.

In systems where the user designed logic must place data onto the MULTIBUS data lines, three-state drivers are required. These drivers should be enabled only when a memory read command (MRDC/) or an I/O read command (IORC/) is present and the module has been addressed.

When both the read and write functions are required, parallel bidirectional bus drivers (e.g., Intel 8226, 8287, etc.) are used. A note of caution must be included for the designer who uses this type of device. A problem may arise if data hold time requirements must be satisfied for user logic following write operations. When bus commands are used to directly produce both the chip select for the bidirectional bus driver and a strobe to a latch in the user logic, removal of that signal may not provide the user's latch with adequate data hold time. Depending on the specifics of the user logic, this problem may be solved by permanently enabling the data buffer's receiver circuits and controlling only the direction of the buffers.

Control Signal Logic — The control signal logic consists of the circuits that forward the I/O and memory read/write commands to their respective destinations, provide the bus with a transfer acknowledge response, and drive the system interrupt lines.

Bus Command Lines

The MULTIBUS information transfer protocol lines (MRDC/, MWTC/, IORD/. and IOWC/) should be buffered by devices with very high speed switching. Because the bus DC requirements specify that each board may load these lines with 2.0 mA, Schottky devices are recommended. LS devices are not recommended due to their poor noise immunity. The commands should be gated

with a signal indicating the base address has been decoded to generate read and write strobes for the user logic.

Transfer Acknowledge Generation

The user interface transfer acknowledge generation logic provides a transfer acknowledge response, XACK/, to notify the bus master that write data provided by the bus master has been accepted or that read data it has requested is available on the MULTIBUS data lines. XACK/ allows the bus master to conclude its current instruction.

Since XACK/ timing requirements depend on both the CPU of the bus master and characteristics of the user logic, a circuit is needed which will provide a range of easily modified acknowledge responses.

The transfer acknowledge signals must be driven by three-state drivers which are enabled when the bus interface is addressed and a command is present.

Interrupt Signal Lines

The asynchronous interrupt lines must be driven by open collector devices with a minimum drive of 16 mA.

In a typical Non Bus Vectored Interrupt system, logic must be provided to assert and latch-up an interrupt signal. In addition to driving the MULTIBUS interrupt lines, the latched interrupt signal would be read by an I/O operation such as reading the module's status. The interrupt signal would be cleared by writing to the status register.

III. MULTIBUS™ SLAVE DESIGN EXAMPLE

A MULTIBUS slave design example has been included in this application note to reinforce the theory previously discussed. The design example is of general purpose I/O slave interface. This design example could easily be modified to be used as a slave memory interface by buffering the address signals and using the appropriate MULTIBUS memory commands. In addition, to help the reader better understand an application for an I/O slave interface, two Intel 8255A Parallel Peripheral Interface (PPI) devices are shown connected to the slave interface.

The design example is shown in both 8/16-bit version and an 8-bit version. The 8/16-bit version is an I/O interface which will permit a 16-bit master to perform 8 or 16 bit data transfers. 8-bit masters may also use the 8/16-bit version of the design example to perform 8-bit data transfers.

The 8-bit version of the design example may be used by both 8 or 16-bit masters, but will only perform 8-bit data transfers. It does not contain the circuitry required to perform 16-bit data transfers.

Both the 8/16-bit version and the 8-bit version of the design example were implemented on an iSBC 905 prototype board. The schematics for each of the examples are given in Appendices F and G.

Functional/Programming Characteristics

This section describes the organization of the slave interface from two points of view, the functional point of view and the programming characteristics. First, the principal functions performed by the hardware are identified and the general data flow is illustrated. This point of view is intended as an introduction to the detailed description provided in the next section; Theory of Operation. In the second point of view, the information needed by a programmer to access the slave is summarized.

Functional Description — The function of this I/O slave is to provide the bus interface logic for general purpose I/O functions and for two Intel 8255A Parallel Peripheral Interface (PPI) devices. Eight device selects (port addresses) are available for general purpose I/O functions. One of these device select lines is used to read and reset the state of an interrupt status flip-flop, the other seven device selects are unused in this design. An additional eight I/O device port addresses are used by the two 8255A devices; four I/O port addresses per 8255A (three I/O port address for the three parallel ports A, B, and C and the fourth I/O port address for the device control register).

Figure 14 contains a functional block diagram of the slave design example. This block diagram shows the fundamental circuit elements of a bus slave: bidirectional data bus drivers/receivers, address decoding logic and bus control logic. Also shown is the address decoding logic for the low order four bits, the interrupt logic which is selected by this decoding logic, and the two 8255A devices.

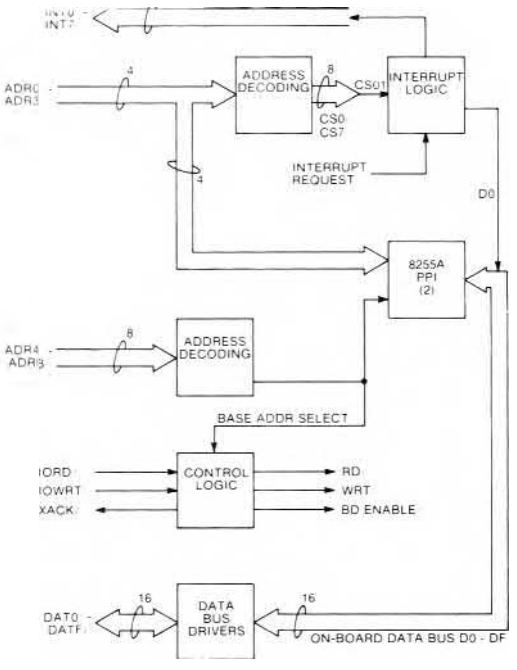


Figure 14. MULTIBUS™ Slave Design Example Functional Block Diagram

Programming Characteristics — The slave design example provides 16 I/O port addresses which may be accessed by user software. The base address of the 16 contiguous port addresses is selected by wire wrap connections on the prototype board. The wire wrap connections specify address bits ADR4/ - ADRB/. They allow the selection of a base address on any 16 byte boundary. Twelve address bits (ADR0/ - ADRB/) are used since 16-bit (8086 based) masters use 12 bits to specity I/O port addresses. If an 8 bit (8080 or 8085 based) master is used with this slave board, the high order address bits (ADR8/ - ADRB/) must not be used by the decoding circuits; a wire wrap jumper position (ground position) is provided for this.

The 16 I/O port addresses are divided into two groups of 8 port addresses by decoding address line ADR3/. Port addresses XX0 - XX7 are used for general I/O functions (XX indicates any hexidecimal digit combination). Port address XX0 is used for accessing the interrupt status flip-flop and are selected, then ADR0/ is used to specify which of two PPIs are selected. If the address is even (XX8, XXA, XXC, or XXE) then one PPI is selected. If the address is odd (XX9, XXB, XXD, or XXF), then the other PPI is selected. ADR1/ and ADR2/ are connected directly to the PPIs. Table 1 summarizes the I/O port addresses of the slave design example. Note that if a 16-bit master is used, it is possible to access the slave in a byte or word mode. If word access is used with port address XX8, XXA, XXC, or XXE, then 16 bit transfers will occur between the PPIs and the master. These 16 bit transfers occur because an even address has been specified and the MULTIBUS BHEN/ signal indicates that a 16-bit transfer is requested.

Theory of Operation

In the preceding section, each of the slave design example functional blocks was identified and briefly explained. This section explains how these functions are implemented. For detailed circuit information, refer to the schematics in Appendices F and G. The schematic in Appendix F is on a foldout page so that the following text may easily be related to the schematic.

The discussion of the theory of operation is divided into five segments, each of which discusses a different function performed by the MULTIBUS slave design example. The five segments are:

1. Bus address decoding
2. Data buffers
3. Control signals
4. Interrupt logic
5. PPI operation

Each of these topics are discussed with regard to the 8/16-bit version of the design example; followed by a discussion of the circuit elements which are required by the 8-bit version of the interface.

Bus Address Decoding — Bus address decoding is performed by two 8205 1 out of 8 binary decoders. One decoder (A3) decodes address bits ADR8/ - ADRB/ and the second decoder (A2) decodes address bits ADR4/ - ADR7/. The base address

Table 1
SLAVE DESIGN EXAMPLE PORT ADDRESSES

I/O PORT ADDRESS	READ	WRITE
BYTE ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX1 - XX7	Unused	Unused
XX8	Parallel Port A, Even PPI	Parallel Port A, Even PPI
XX9	Parallel Port A, Odd PPI	Parallel Port A, Odd PPI
XXA	Parallel Port B, Even PPI	Parallel Port B, Even PPI
XXB	Parallel Port B, Odd PPI	Parallel Port B, Odd PPI
XXC	Parallel Port C, Even PPI	Parallel Port C, Even PPI
XXD	Parallel Port C, Odd PPI	Parallel Port C, Odd PPI
XXE	Illegal Condition	Control, Even PPI
XXF	Illegal Condition	Control, Odd PPI
WORD ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX2 - XX6	Unused	Unused
XX8	Parallel Port A, Even and Odd PPIs	Parallel Port A, Even and Odd PPIs
XXA	Parallel Port B, Even and Odd PPIs	Parallel Port B, Even and Odd PPIs
XXC	Parallel Port C, Even and Odd PPIs	Parallel Port C, Even and Odd PPIs
XXE	Illegal Condition	Control, Even and Odd PPIs

XX = Any hex digits, assigned by jumpers; XX defines the base address.

selected is determined by the position of wire wrap jumpers. The outputs of the two decoders are ANDed together to form the BASE ADR SELECT signal. This signal specifies the base address for a group of 16 I/O ports. Using the wire wrap jumper positions shown in the schematic, a base address of E3 has been selected. Therefore, this MULTIBUS slave board will respond to I/O port addresses in the E30 - E3F range.

If this slave board is to be used with 8-bit MULTIBUS masters, the high order address bits must not be decoded. Therefore, the wire wrap jumper which selects the output of decoder A3 must be placed in the top (ground) position (pin 10 of gate A9 to ground).

The low order 4 address lines (ADR0 - ADR3) are buffered and inverted using 74LS04 inverters. These address lines are input to an 8205 for decoding a chip select for the interrupt logic; the address lines are also used directly by the PPIs. LS-Series logic is required for buffering to meet the MULTIBUS specification for I_{IL} (low level input current). S-Series or standard series logic will not meet this specification.

Address decoder A4 is used to decode addresses E30 - E37. The CS0 output of this decoder is used to select the interrupt logic, thus I/O port address E30 is used to read and reset the interrupt latch. The remaining outputs from decoder A4 (CS1 - CS7) are not used in this example. They would normally be used to select other functions in a slave board with more capability. Note that in the schematic shown in Appendix G for the 8-bit version of this slave design example, the high order (ADR8/ - ADRB/) address decoder is not included and the BHEN signal is not used.

Data Buffers — Intel 8287 8-bit parallel bidirectional bus drivers are used for the MULTIBUS data lines DAT0 - DATF. In the 8/16-bit version of the slave board, three 8287 drivers are used.

When an 8-bit data transfer is requested, either driver A5, which is connected to on-board data

lines D0 - D7, or driver A6, which is connected to on-board data lines D8 - DF, is used. If a byte transfer is requested from an even address, driver A5 will be selected. If a byte transfer from an odd address is requested, driver A6 will be selected. All byte transfers take place on MULTIBUS data lines DAT0/ - DAT7/. When a word (16-bit) transfer is requested from an even address, drivers A5 and A7 will be used. Note that if a user program requests a word transfer from an odd address, 16-bit masters in the iSBC product line will actually perform two byte transfer requests.

The logic which determines the chip selection (8287 input signal OE, output enable) signals for the bus drivers uses the low order address bit (ADR0/) and the buffered Byte High Enable signal (BHENBL/). Note that the MULTIBUS signal BHEN/ has been buffered with an 74LS04 inverter. This is done to meet the bus address line loading specification. The SWAP BYTE/ signal which is generated is qualified by the BD ENBL/ signal and used to select the bus drivers.

The steering pin for the 8287 drivers is labelled T (transmit) and is driven by the signal RD. When an input (read) request is active or when neither a read or write command is being serviced, the direction of data transfer of the 8287 will be set for B to A.

The 8287 drivers are set to point IN (direction B to A) when no MULTIBUS I/O transfer command is being serviced for two reasons. First, if the driver were pointed OUT (direction A to B) and a write command occured, it would be necessary to turn the buffers IN and set the OE (output enable) signal active before the data could be transferred to the on-board bus. A possibility of a "buffer-fight" could occur in some designs if the OE signal permitted an 8287 to drive the MULTIBUS data lines momentarily before the steering signal could switch the direction of the 8287. In this case, both the MULTIBUS master and the slave would be driving the data lines; this is not recommended. (In this particular design, the steering signal will always stabilize before the OE signal becomes active.)

The second reason the driver is pointing IN when no command is present is due to the "data valid after WRITE" requirements of the 8255As. The 8255A requires that data remain on its data lines for 30 ns after the WRITE command ($\overline{WR}$ at the 8255A) is removed. This requirement will be met if the direction of the 8287 drivers is not switched when the MULTIBUS IOWC/ signal is removed (WRT/ could have been used to steer the 8287 instead of RD); and if the capacitance of the on-board data bus lines is sufficient to hold the data values on the bus after the 8287 OE signal and the 8255A PPI WRT/ signal go inactive. The on-board data bus may easily be designed such that the capacitance of the lines is sufficient to meet the 30 ns data hold time requirement. In addition, the current leakage of all devices connected to the on-board bus must be kept small to meet the 30 ns data hold time requirement.

The 8-bit version of this design example uses only one 8287 instead of the three required by the 8/16-bit version. The logic required to control the swap byte buffer is also not necessary. The chip select signal used for the 8287 is the BD ENBL/ signal.

Control Signals — The MULTIBUS control signals used by this slave design example are IORC/, IOWC/, and XACK/. IORC/ and IOWC/ are qualified by the BASE ADR SELECT/ signal to form the signals RD and WRT. RD and WRT are used to drive the interrupt logic, the PPI logic and the XACK/ (transfer acknowledge) logic.

For the XACK/ logic RD and WRT are ORed to form the BD ENBL/ signal which is inverted and used to drive the CLEAR pin of a shift register. When the slave board is not being accessed, the CLEAR pin of the shift register will be low (BD ENBL/ is high). This causes the shift register to remain cleared and all outputs of the shift register will be low. When the slave board is accessed, the CLEAR pin will be high, and the A and B inputs (which are high) will be clocked to the output pins by CCLK/. To select a delay for the XACK/ signal, a jumper must be installed from one of the shift register output pins to the 8089 tri-state driver. Each of the shift register output pins select an integer multiple of CCLK/ periods for the signal delay. Since the CCLK/ signal is asynchronous, the actual delay selected may only be specified with a tolerance of one CCLK/ period. In this example a delay of 3 - 4 CCLK/ periods was selected; with a CCLK/ period of 100 ns, the XACK/ delay would occur somewhere within the range of 300 - 400 ns from the time when the CLEAR signal goes high.

The control signal logic used in the 8-bit version of the slave design example is identical to the logic used in the 8/16-bit version.

Interrupt Logic — The interrupt logic uses a 74S74 flip-flop to latch an asynchronous interrupt request from some external logic. The Q output of the INTERRUPT REQUEST LATCH is output through an open collector gate to one of the MULTIBUS interrupt lines. The state of the INTERRUPT REQUEST LATCH is transferred to the INTERRUPT STATUS LATCH when a read command is performed on I/O port BASE ADDRESS+0 (E30 for the jumper configuration shown). The Q output of INTERRUPT STATUS LATCH is used to drive data line D0 of the on-board data bus by using an 8089 tri-state driver. If a user program performs an INPUT from I/O port E30, data bit 0 will be set to 1 if the INTERRUPT REQUEST LATCH is set.

The purpose of INTERRUPT STATUS LATCH is to minimize the possibility of the asynchronous interrupt occuring while the interrupt status is being read by a bus master. If the latch was not included in the design and an asynchronous interrupt did occur while a bus master is reading MULTIBUS data line DAT0/, a data buffer on the master could go into a meta-stable state. By adding the extra latch, which is clocked by the IORD/ command for I/O port E30, the possibility of data line DAT0/ changing during a bus master read operation is eliminated.

The INTERRUPT REQUEST LATCH is cleared when a user program performs an OUTPUT to I/O port E30.

This interrupt structure assumes that several interrupt sources may exist on the same MULTIBUS interrupt line (for example, INT3/). When the MULTIBUS master gets interrupted, it must poll the possible sources of the interrupt received and after determining the source of the interrupt, it must clear the INTERRUPT REQUEST LATCH for that particular interrupt source.

The interrupt logic for the 8-bit version of the design example is identical to the interrupt logic of the 8/16-bit version of the design example.

PPI Operation — Two 8255A Parallel Peripheral Interface (PPI) devices are shown interfaced to the slave design example logic. One PPI is connected to the on-board data bus lines D0 - D7 and is addressed with the even I/O port addresses E38, E3A, E3C, and E3E. The second PPI is connected to data bus lines D8 - DF and is addressed with the odd I/O port addresses E39, E3B, E3D, and E3F. The even or odd I/O port selection is controlled by using the ADR0 address line in the chip select term of the PPIs. In addition, the odd PPI (A11) is selected when the BHENBL term is high. This occurs when the MULTIBUS signal BHEN/ is low indicating that a word (16-bit) I/O instruction is being executed. When a word I/O instruction is executed, both PPIs will perform the I/O operation specified.

The specifications of the 8255A device state that the address lines A0 and A1 and the chip select lines must be stable before the $\overline{RD}$ or $\overline{WR}$ lines are activated. The MULTIBUS specification address set-up time of 50 ns and the short gate propagation delays in this design assure that the address lines are stable before $\overline{RD}$ or $\overline{WR}$ are active.

The data hold requirements of the 8255A were discussed in a previous section. The 8255A specification states that data will be stable on the data bus lines a maximum of 250 ns after a READ command. This specification was used to select the delay for the XACK/ signal.

The PPI operation for the 8-bit version of the design example is slightly different than that used for the 8/16-bit version. The chip select signal for the bottom PPI does not use the BHENBL term since 16-bit data transfers are not possible with an 8-bit I/O slave board. Also, the chip select and address signals have been swapped so the top PPI occupies I/O address range X8 - XB, and the bottom PPI occupies I/O address range XC - XF (X is the base address of the 8-bit version). This swapping of the address lines was not necessary; however, it was thought to be more convenient to access the PPIs in two groups of 4 contiguous I/O port addresses.

IV. SUMMARY

This application note has shown the structure of the Intel MULTIBUS system bus. The structure supports a wide range of system modules from the Intel OEM Microcomputer Systems product line that can be extended with the addition of user designed modules. Because the user designed modules are no doubt unique to particular applications, a goal of this application note has been to describe in detail the singular common element - the bus interface. Material has also been presented to assist the systems designer to understanding the bus functions so that successful systems integration can be achieved.

APPENDIX A

PIN ASSIGNMENT OF BUS SIGNALS ON MULTIBUS BOARD P1 CONNECTOR

	(COMPONENT SIDE)			(CIRCUIT SIDE)		
	PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
POWER SUPPLIES	1	GND	Signal GND	2	GND	Signal GND
	3	+5V	+5Vdc	4	+5V	+5Vdc
	5	+5V	+5Vdc	6	+5V	+5Vdc
	7	+12V	+12Vdc	8	+12V	+12Vdc
	9	−5V	−5Vdc	10	−5V	−5Vdc
	11	GND	Signal GND	12	GND	Signal GND
BUS CONTROLS	13	BCLK/	Bus Clock	14	INIT/	Initialize
	15	BPRN/	Bus Pri. In	16	BPRO/	Bus Pri. Out
	17	BUSY/	Bus Busy	18	BREQ/	Bus Request
	19	MRDC/	Mem Read Cmd	20	MWTC/	Mem Write Cmd
	21	IORC/	I/O Read Cmd	22	IOWC/	I/O Write Cmd
	23	XACK/	XFER Acknowledge	24	INH1/	Inhibit 1 disable RAM
BUS CONTROLS AND ADDRESS	25		Reserved	26	INH2/	Inhibit 2 disable PROM or ROM
	27	BHEN/	Byte High Enable	28	AD10/	
	29	CBRQ/	Common Bus Request	30	AD11/	Address
	31	CCLK/	Constant Clk	32	AD12/	Bus
	33	INTA/	Intr Acknowledge	34	AD13/	
INTERRUPTS	35	INT6/	Parallel	36	INT7/	Parallel
	37	INT4/	Interrupt	38	INT5/	Interrupt
	39	INT2/	Requests	40	INT3/	Requests
	41	INT0/		42	INT1/	
ADDRESS	43	ADRE/		44	ADRF/	
	45	ADRC/		46	ADRD/	
	47	ADRA/	Address	48	ADRB/	Address
	49	ADR8/	Bus	50	ADR9/	Bus
	51	ADR6/		52	ADR7/	
	53	ADR4/		54	ADR5/	
	55	ADR2/		56	ADR3/	
	57	ADR0/		58	ADR1/	
DATA	59	DATE/		60	DATF/	
	61	DATC/		62	DATD/	
	63	DATA/	Data	64	DATB/	Data
	65	DAT8/	Bus	66	DAT9/	Bus
	67	DAT6/		68	DAT7/	
	69	DAT4/		70	DAT5/	
	71	DAT2/		72	DAT3/	
	73	DAT0/		74	DAT1/	
POWER SUPPLIES	75	GND	Signal GND	76	GND	Signal GND
	77		Reserved	78		Reserved
	79	−12V	−12Vdc	80	−12V	−12Vdc
	81	+5V	+5Vdc	82	+5V	+5Vdc
	83	+5V	+5Vdc	84	+5V	+5Vdc
	85	GND	Signal GND	86	GND	Signal GND

All Mnemonics © Intel Corporation 1978

APPENDIX A (Continued)

P2 CONNECTOR PIN ASSIGNMENT OF OPTIONAL BUS SIGNALS

	(COMPONENT SIDE)			(CIRCUIT SIDE)	
PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
1	GND	Signal GND	2	GND	Signal GND
3	5 VB	+5V Battery	4	5 VB	+5V Battery
5		Reserved	6	VCCPP	+5V Pulsed Power
7	-5 VB	-5V Battery	8	-5 VB	-5V Battery
9		Reserved	10	Reserved	
11	12 VB	+12V Battery	12	12 VB	+12V Battery
13	PFSR/	Power Fail Sense Reset	14	Reserved	
15	-12 VB	-12V Battery	16	-12 VB	-12V Battery
17	PFSN/	Power Fail Sense	18	ACLO	AC Low
19	PFIN/	Power Fail Interrupt	20	MPRO/	Memory Protect
21	GND	Signal GND	22	GND	Signal GND
23	+15V	+15V	24	+15V	+15V
25	-15V	-15V	26	-15V	-15V
27	PAR1/	Parity 1	28	HALT/	Bus Master HALT
29	PAR2/	Parity 2	30	WAIT/	Bus Master WAIT STATE
31			32	ALE	Bus Master ALE
33			34	Reserved	
35			36	Reserved	
37			38	AUX RESET/	Reset switch
39			40		
40			42		
43	Reserved		44		
45			46		
47			48		
49			50	Reserved	
51			52		
53			54		
55			56		
57			58		
59			60		

Notes:

1. PFIN, on slave modules, if possible, should have the option of connecting to INT0/ on P1.
2. All undefined pins are reserved for future use.

All Mnemonics © Intel Corporation 1978

APPENDIX B
BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BCY}	Bus Clock Period	100	D.C.	ns
t_{BW}	Bus Clock Width	0.35 t_{BCY}	0.65 t_{BCY}	
t_{SKEW}	BCLK/skew		3	ns
t_{PD}	Standard Bus Propagation Delay		3	
t_{AS}	Address Set-Up Time (at Slave Board)	50		ns
t_{DS}	Write Data Set Up Time	50		ns
t_{AH}	Address Hold Time	50		ns
t_{DHW}	Write Data Hold Time	50		ns
t_{DXL}	Read Data Set Up Time To XACK	0		ns
t_{DHR}	Read Data Hold Time	0	65	ns
t_{XAH}	Acknowledge Hold Time	0	65	ns
t_{XACK}	Acknowledge Time	0	t_{TOUT}	ns
t_{CMD}	Command Pulse Width	100	t_{TOUT}	ns
t_{ID}	Inhibit Delay	0	100 (Recommend < 100 ns)	ns
t_{XACKA}	Acknowledge Time of of an Inhibited Slave	t_{IAD} + 50 ns	t_{TOUT}	
t_{XACKB}	Acknowledge Time of an Inhibiting Slave	1.5	t_{TOUT}	μs
t_{IAD}	Acknowledge Disable from Inhibit (An internal parameter on an inhibited slave; used to determine t_{XACKA} Min.)	0	100 (arbitrary)	ns
t_{AIZ}	Address to Inhibits High delay		100	ns
t_{INTA}	INTA/ Width	250		ns
t_{CSEP}	Command Separation	100		ns

APPENDIX B (Continued)

BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BREQL}	↓BCLK/ to BREQ/ Low Delay	0	35	ns
t_{BREQH}	↓BCLK/ to BREQ/ High Delay	0	35	ns
t_{BPRNS}	BPRN/ to ↓BCLK/ Setup Time	22		ns
t_{BUSY}	BUSY/ delay from ↓BCLK/	0	70	ns
t_{BUSYS}	BUSY/ to ↓BCLK/ Setup Time	25		ns
t_{BPRO}	↓BCLK/ to BPRO/ (CLK to Priority Out)	0	40	ns
t_{BPRNO}	BPRN/ to BPRO/ (Priority In to Out)	0	30	ns
t_{CBRO}	↓BCLK/ to CBRQ/ (CLKto Common Bus Request)	0	60	ns
t_{CBRQS}	CBRQ/ to ↓BCLK/ Setup Time	35		ns
t_{CPM}	Central Priority Module Resolution Delay (Parallel Priority)	0	$t_{BCY}-t_{BREQ}-2t_{PD}-t_{BPRNS}-t_{SKEW}$	
t_{CCY}	C-clock Period	100	110	ns
t_{CW}	C-clock Width	0.35 t_{CCY}	0.65 t_{CCY}	ns
t_{INIT}	INIT/Width	5		ms
t_{INITS}	INIT/ to MPRO/ Setup Time	100		ns
t_{PBD}	Power Backup Logic Delay	0	200	ns
t_{PFINW}	PFIN/ Width	2.5		ms
t_{MPRO}	MPRO/ Delay	2.0	2.5	ms
t_{ACLOW}	ACLO/ Width	3.0		ms
t_{PFSRW}	PFSR/ Width	100		ns
t_{TOUT}	Timeout Delay	5	∞	ms
t_{DCH}	D C. Power Supply Hold from ALCO/	3.0		ms
t_{DCS}	D.C. Power Supply Setup to ACLO/	5		ms

Driver 1,3						Receiver 2,3				Termination			
Bus Signals	Location	Type	I_{OL} Min$_{ma}$	I_{OH} Min$_{\mu a}$	C_O Max$_{pf}$	Location	I_{IL} Max$_{ma}$	I_{IH} Max$_{\mu a}$	C_I Max$_{pf}$	Location	Type	R	Units
DAT0/→DATF/ (16 lines)	Masters and Slaves	TRI	16	-2000	300	Masters and Slaves	−0.8	125	18	1 place	Pullup	2.2	KΩ
ADR0/-ADRB/. BHEN/ (21 lines)	Masters	TRI	16	-2000	300	Slaves	−0.8	125	18	1 place	Pullup	2.2	KΩ
MRDC/,MWTC/	Masters	TRI	32	-2000	300	Slaves (Memory; memory-mapped I/O)	−2	125	18	1 place	Pullup	1	KΩ
IORC/,IOWC/	Masters	TRI	32	-2000	300	Slaves (I/O)	−2	125	18	1 place	Pullup	1	KΩ
XACK/	Slaves	TRI	32	-2000	300	Masters	−2	125	18	1 place	Pullup	510	Ω
INH1/,INH2/	Inhibiting Slaves	OC	16	—	300	Inhibited Slaves (RAM, PROM, ROM, Memory-Mapped I/O)	−2	50	18	1 place	Pullup	1	KΩ
BCLK/	1 place (Master us)	TTL	48	−3000	300	Master	−2	125	18	Mother-board	To +5V To GND	220 330	Ω Ω
BREQ/	Each Master	TTL	5	−400	60	Central Priority Module	2	50	18	Central Priority Module (not req)	Pullup	1	KΩ
BPRO/	Each Master	TTL	5	−400	60	Next Master in Serial Priority Chain at its BPRN/	−1.6	50	18	(not req)			
BPRN/	Parallel: Central Priority Module Serial Prev Masters BPRO/	TTL	5	−400	300	Master	−2	50		(not req)			
BUSY/, CBRQ	All Masters	O.C.	32	—	300	All Masters	−2	50	18	1 place	Pullup	1	KΩ
INIT/	Master	O.C.	32	—	300	All	−2	50	18	1 place	Pullup	2.2	KΩ
CCLK/	1 place	TTL	48	−3000	300	Any	−2	125	18	Mother-board	To +5V To GND	220 330	Ω Ω
INTA/	Masters	TRI	32	-2000	300	Slaves (Interrupting I/O)	−2	125	18	1 place	Pullup	1	KΩ
INT0/→INT7/ (8 lines)	Slaves	O.C.	16	—	300	Masters	−1.6	40	18	1 place	Pullup	1	KΩ
PFSR/	User's Fron Panel?	TTL	16	−400	300	Slaves, Masters	−1.6	40	18	1 place	Pullup	1	KΩ
PFSN/	Power Back Up Unit	TTL	16	−400	300	Masters	−1.6	40	16	1 place	Pullup	1	KΩ
ACLO	Power Supply	O.C.	16	−400	300	Slaves, Masters	−1.6	40	18	1 place	Pullup	1	KΩ
PFIN/	Power Back-Up Unit	O.C	16	−400	300	Masters	−1.6	40	18	1 place	Pullup	1	KΩ
MPRO/	Power Back-Up Unit	TTL	16	−400	300	Slaves Masters	−1.6	40	18	1 place	Pullup	1	KΩ

APPENDIX C (Continued)

BUS DRIVERS, RECEIVERS, AND TERMINATIONS

Driver 1,3						Receiver 2,3				Termination			
Bus Signals	Location	Type	I_{OL} Min_{ma}	I_{OH} $Min_{\mu a}$	C_O Max_{pf}	Location	I_{IL} Max_{ma}	I_{IH} $Max_{\mu a}$	C_I Max_{pf}	Location	Type	R	Units
Aux Reset/	User's Front Panel?	Switch to GND	—	—	—	Masters	−2	50	18	None			

Notes:

1. Driver Requirements

   I_{OH} = High Output Current Drive
   I_{OL} = Low Output Current Drive
   C_O = Capacitance Drive Capability
   TRI = 3-State Drive
   O.C. = Open Collector Driver
   TTL = Totem-pole Driver

2. Receiver Requirements

   I_{IH} = High Input Current Load
   I_{IL} = Low Input Current Load
   C_I = Capacitive Load

3. TTL low state must be ≥ −0.5v but ≤ 0.8v at the receivers
   TTL high state must be ≥ 2.0v but ≤ 5.5v at the receivers

4. For the iSBC 80/10 and the iSBC 80/10A use only a 1K pull-up resistor to +5v for BCLK/ and CCLK/ termination.

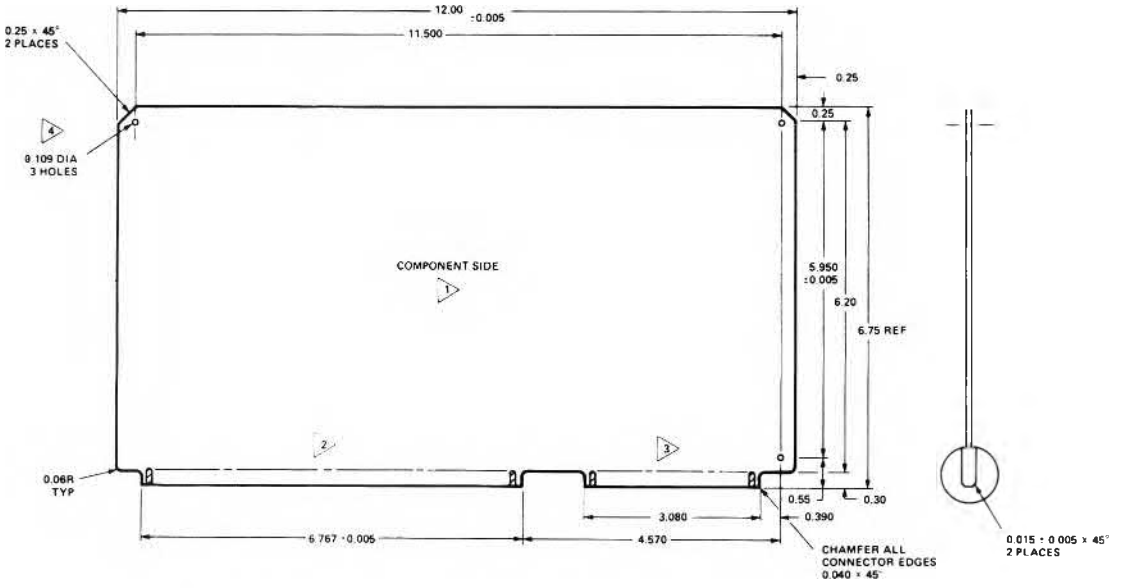
APPENDIX D
BUS POWER SPECIFICATIONS

	Standard (P1)				Optional (P2)					
					Analog Power		Battery Power Backup			
	Ground	**+5**	**+12**	**−12**	**+15**	**−15**	**+5**	**+12**	**−12**	**−5**
Mnemonic	GND	+5V	+12V	−12V	+15V	−15V	+5B	+12B	−12B	−5B
Bus Pins	P1+1,2, 11,12, 75,76 85,86	P1+3,4, 5,6,81, 82,83, 84	P1+7,8	P1+79, 80	P2+23, 24	P2+25, 26	P2+3,4, 5,6	P2+11, 12	P2+15, 16	P2−7,8
Nominal Output	Ref.	+5.0V	+12.0V	−12.0V	+15.0V	−15.0V	+5.0V	+12.0V	−12.0V	−5.0V
Tolerance from Nominal[1]	Ref.	±5%	±5%	±5%	±3%	±3%	±5%	±5%	±5%	±5%
Ripple (Pk-Pk)[2]	Ref.	50 mV	50 mV	50 mV	10 mV	10 mV	50 mV	50 mV	50 mV	50 mV
Transient Response Time[3]		500 μs	500μs	500 μs	100 μs	100 μs	500 μs	500 μs	500 μs	500 μs
Transient Deviation[4]		±10%	±10%	±10%	±10%	±10%	±10%	±10%	±10%	±10%

NOTES:

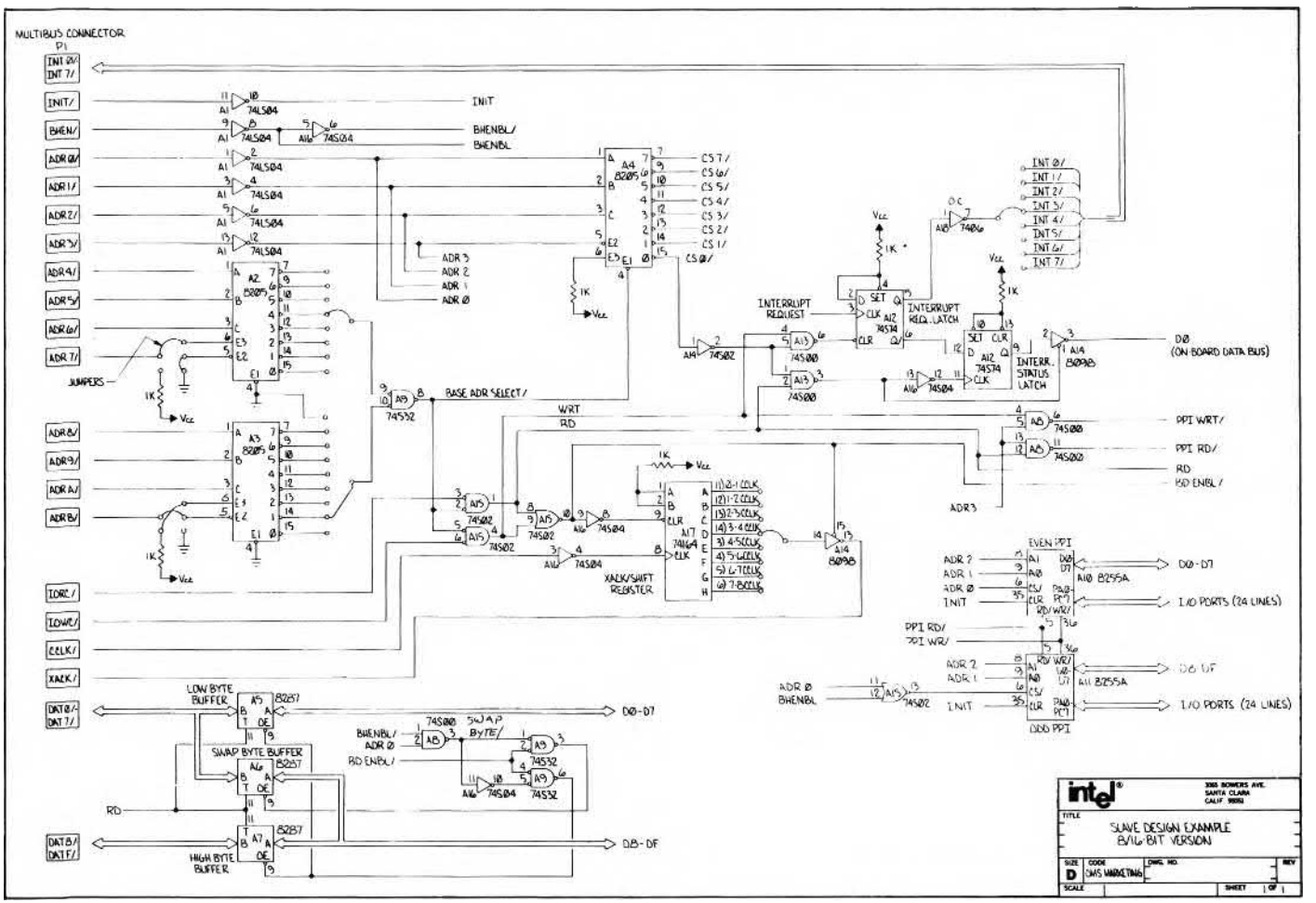
1. Tolerance is worst case, including initial voltage setting line and load effects of power source, temperature drift, and any additional steady state influences.
2. As measured over any bandwidth not to exceed 0 to 500 kHz.
3. As measured from the start of a load change to the time an output recovers within ±0.1% of final voltage.
4. Measured as the peak deviation from the initial voltage.

APPENDIX E
MECHANICAL SPECIFICATIONS

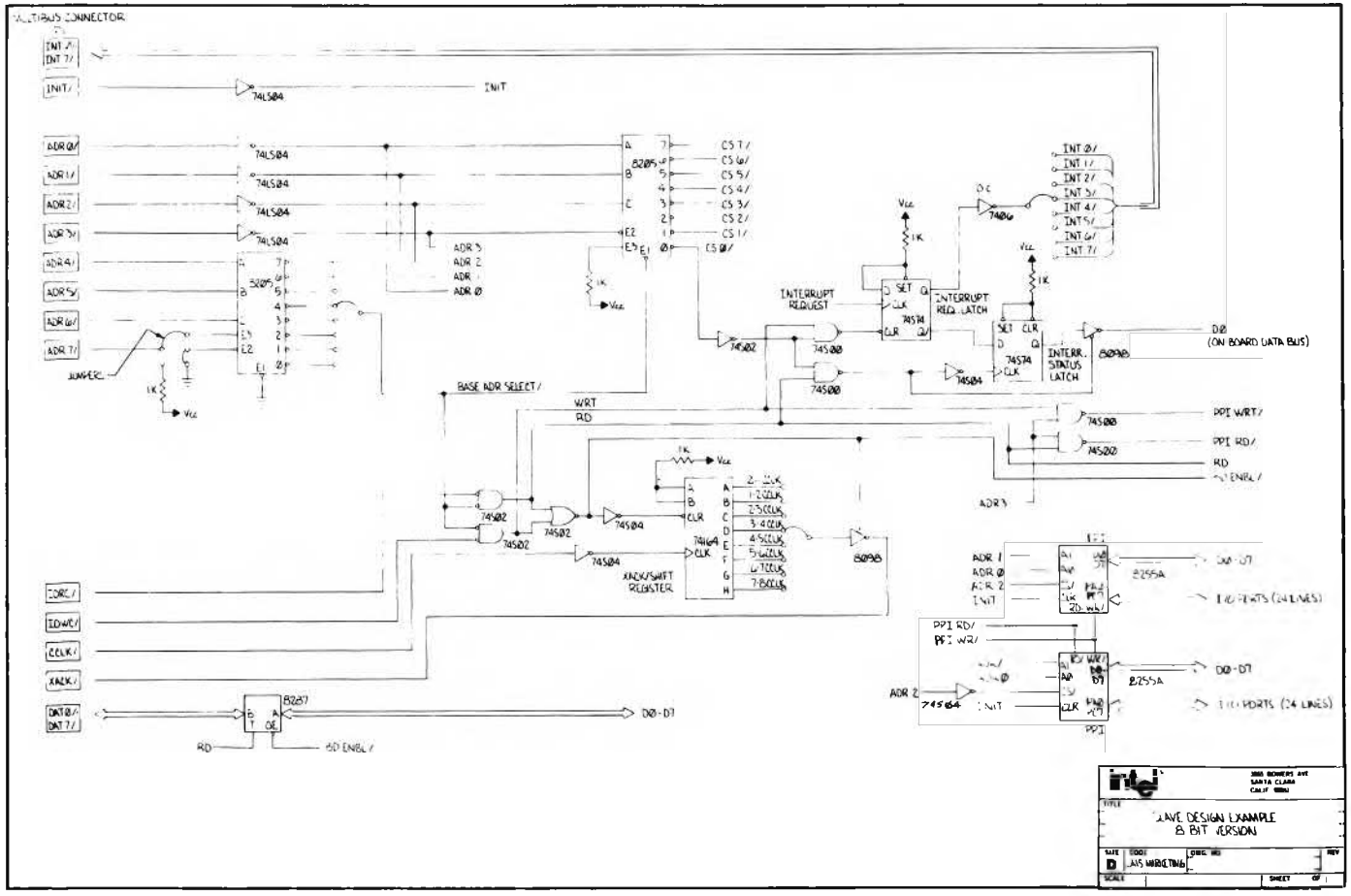


NOTES:

1. BOARD THICKNESS: 0.062
2. MULTIBUS CONNECTOR 86-PIN, 0.156 SPACING
   CDC VFB01E43D00A1
   VIKING 2VH43/1ANE5
3. AUXILIARY CONNECTOR 60-PIN, 0.100 SPACING
   CDC VPB01B30D00A1
   TI H311130
   AMP PE5 14559
4. EJECTOR TYPE: SCANBE #S203
5. BUS DRIVERS AND RECEIVERS SHOULD BE LOCATED AS CLOSE AS POSSIBLE TO THEIR RESPECTIVE MULTIBUS PIN CONNECTIONS
6. BOARD SPACING: 0.6
7. COMPONENT HEIGHT: 0.4
8. CLEARANCE ON CONDUCTOR NEAR EDGES: 0.050



MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8/16-BIT VERSION



MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8-BIT VERSION

intel®

APPLICATION NOTE

AP-43

November 1978

Using The iSBC™ 957 Execution Vehicle For Executing 8086 Program Code

Joe Barthmaier
OEM Microcomputer Systems Applications

©Intel Corporation, 1978

9800816

I. INTRODUCTION

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package contains the hardware and software required to interface an iSBC 86/12 Single Board Computer with an Intellec Microcomputer Development System. The iSBC 957 package gives the 8086 user the capability to develop software on an Intellec System and then debug this software on an iSBC 86/12 board using a program download capability and an interactive system monitor. The 8086 user has all the capabilities of the Intellec system at his disposal and has the powerful iSBC 86/12 system monitor commands to use for debugging 8086 programs.

The iSBC 86/12 board is an Intel 8086 based processor board which, in addition to the processor, contains 32K bytes of dual port RAM, sockets for up to 16K bytes of ROM/EPROM, a serial I/O port, 24 parallel I/O lines, 2 programmable counters, 9 levels of vectored priority interrupts, and an interface to the MULTIBUS™ system bus. The iSBC 957 package consists of monitor EPROMs for the iSBC 86/12 board, Loader software for the Intellec system, four (4) cable assemblies, assorted line drivers and terminators, and signal adapters.

The iSBC 957 package provides the capability of downloading and uploading program and data blocks between an iSBC 86/12 board and an Intellec system. In addition, monitor commands and displays may be input and viewed from the Intellec system console. The iSBC 957 package, when used with the iSBC 86/12 board and an Intellec Microcomputer Development System, provides the user with the capability to edit, compile or assemble, link, locate, download, and interactively debug programs for the 8086 processor. The iSBC 957 package and the iSBC 86/12 board form an excellent "execution vehicle" for users developing software for the 8086 processor regardless of whether the users are 8086 component users or iSBC 86/12 board users. Using the iSBC 957 package 8086 programs may be debugged at the full 5 MHz speed of the processor. The recommended hardware for the execution vehicle is an iSBC 660 system chassis with an 8 card slot backplane and power supply, an iSBC 032 32K byte RAM memory board, the iSBC 957 package, and the iSBC 86/12 board.

This application note will describe how the iSBC 957 package may be used to develop and debug 8086 programs. First a description of the iSBC 86/12 board will be presented. Readers familiar with the iSBC 86/12 board may want to skip this section. Next follows a detailed description of the iSBC 957 package and the iSBC 86/12 system monitor commands. A program example of a matrix multiplication routine will then be presented. This example will contain both assembly language and PL/M-86 procedures. The steps required to compile, assemble, link, locate and debug the program code will be explained in detail. A typical debugging session using the iSBC 86/12 system monitor will be presented.

II. THE iSBC™ 86/12 SINGLE BOARD COMPUTER

The iSBC 86/12 Single Board Computer, which is a member of Intel's complete line of iSBC 80/86 computer products, is a complete computer system on a single printed-circuit assembly. The iSBC 86/12 board includes a 16-bit central processing unit (CPU), 32K bytes of dynamic RAM, a serial communications interface, three programmable parallel I/O ports, programmable timers, priority interrupt control, MULTIBUS control logic, and bus expansion drivers for interface with other MULTIBUS-compatible expansion boards. Also included is dual port control logic to allow the iSBC 86/12 board to act as a slave RAM device to other MULTIBUS masters in the system. Provision is made for user installation of up to 16K bytes of read only memory. Figure 1 contains a block diagram of the iSBC 86/12 board and in Appendix A is a simplified logic diagram of the iSBC 86/12 board.

Central Processing Unit

The central processor for the iSBC 86/12 board is Intel's 8086, a powerful 16-bit H-MOS device. The 225 sq. mil chip contains 29,000 transistors and has a clock rate of 5MHz. The architecture includes four (4) 16-bit byte addressable data registers, two (2) 16-bit memory base pointer registers and two (2) 16-bit index registers, all accessed by a total of 24 operand addressing modes for complex data handling and very flexible memory addressing.

Instruction Set — The 8086 instruction repertoire includes variable length instruction format (including double operand instructions), 8-bit and 16-bit signed and unsigned arithmetic operators for binary, BCD and unpacked ASCII data, and iterative word and byte string manipulation functions. The instruction set of the 8086 is a functional superset of the 8080A/8085A family and with

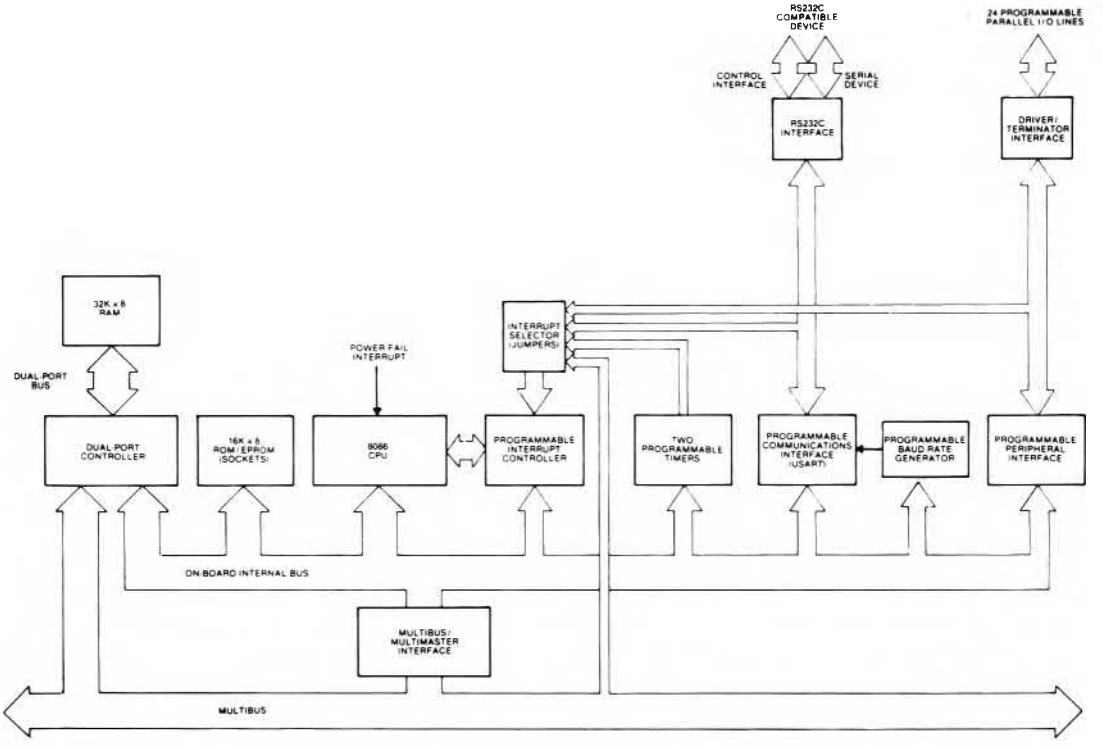


Figure 1. iSBC™ 86/12 Single Board Computer Block Diagram

available software tools, programs written for the 8080A/8085A can be easily converted and run on the 8086 processor.

Architectural Features — A 6-byte instruction queue provides pre-fetching of sequential instructions and can reduce the 1.2 μ sec minimum instruction cycle to 400 nsec by having the instruction already in the queue.

The stack oriented architecture facilitates nested sub-routines and co-routines, reentrant code and powerful interrupt handling. The memory expansion capabilities offer a 1 megabyte addressing range. The dynamic relocation scheme allows ease in segmentation of pure procedure and data for efficient memory utilization. Four segment registers (code, stack, data, extra) contain program loaded offset values which are used to map 16-bit addresses to 20-bit addresses. Each register maps 64K-bytes at a time and activation of a specific register is controlled explicitly by program control and is also selected implicitly by specific functions and instructions.

Bus Structure

The iSBC 86/12 board has an internal bus for communicating with on-board memory and I/O options, a system bus (the MULTIBUS) for referencing additional memory and I/O options, and the dual-port bus which allows access to RAM from the on-board CPU and the MULTIBUS System Bus. Local (on-board) accesses do not require MULTIBUS communication, making the system bus available for use by other MULTIBUS masters (i.e. DMA devices and other single board computers transferring to additional system memory). This feature allows true parallel processing in a multiprocessor environment. In addition, the MULTIBUS interface can be used for system expansion through the use of other 8- and 16-bit iSBC computers, memory and I/O expansion boards.

RAM Capabilities

The iSBC 86/12 board contains 32K-bytes of dynamic read/write memory. Power for the on-board RAM and refresh circuitry may be optionally provided on an auxiliary power bus, and

the on-board RAM from the iSBC 86/12's CPU and from any other MULTIBUS master via the system bus. The dual port controller allows 8- and 16-bit accesses from the MULTIBUS System Bus and the on-board CPU transfers data to RAM over a 16-bit data path. Priorities have been established such that memory refresh is guaranteed by the on-board refresh logic and that the on-board CPU has priority over MULTIBUS requests for access to RAM. The dual-port controller includes independent addressing logic for RAM access from the on-board CPU and from the MULTIBUS system bus. The on-board CPU will always access RAM starting at location 00000H. Address jumpers allow on-board RAM to be located starting on any 8K-byte boundary within a 1 megabyte address range for accesses from the MULTIBUS system bus. In conjunction with this feature, the iSBC 86/12 board has the ability to protect on-board memory from MULTIBUS access to any contiguous 8K-byte segments. These features allow multi-processor systems to establish local memory for each processor and shared system (MULTIBUS) memory configurations where the total system memory size (including local on-board memory) can exceed 1 megabyte without addressing conflicts.

EPROM/ROM Capabilities

Four sockets are provided for up to 16K-bytes of non-volatile read only memory on the iSBC 86/12 board. Configuration jumpers allow read only memory to be installed in 2K, 4K, or 8K increments.

On-board ROM is accessed via 16 bit data paths. System memory size is easily expanded by the addition of MULTIBUS compatible memory boards available in the iSBC 80/86 family.

Parallel I/O Interface

The iSBC 86/12 board contains 24 programmable parallel I/O lines implemented using the Intel 8255A Programmable Peripheral Interface. The system software is used to configure the I/O lines in any combination of unidirectional input/output and bidirectional ports.

Therefore, the I/O interface may be customized to meet specific peripheral requirements. In order to take full advantage of the large number of possible I/O configurations, sockets are provided for interchangeable I/O line drivers and terminators. Hence, the flexibility of the I/O interface is further polarity, and drive/termination characteristics for each application. The 24 programmable I/O lines and signal ground lines are brought out to a 50-pin edge connector that mates with flat, woven, or round cable.

Serial I/O

A programmable communications interface using the Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the iSBC 86/12 board. A software selectable baud rate generator provides the USART with all common communication frequencies. The USART can be programmed by the system software to select the desired asynchronous or synchronous serial data transmission technique (including IBM Bi-Sync). The mode of operation (i.e., synchronous or asynchronous), data format, control character format, parity, and baud rate are all under program control. The 8251A provides full duplex, double buffered transmit and receive capability. Parity, overrun, and framing error detection are all incorporated in the USART. The RS232C compatible interface on each board, in conjunction with the USART, provides a direct interface to RS232C compatible terminals, cassettes, and asynchronous and synchronous modems. The RS232C command lines, serial data lines, and signal ground line are brought out to a 26 pin edge connector that mates with RS232C compatible flat or round cable. The iSBC 530 teletypewriter adapter provides an optically isolated interface for those systems requiring a 20 mA current loop. The iSBC 530 adapter may be used to interface the iSBC 86/12 board to teletypewriters or other 20 mA current loop equipment.

Programmable Timers

The iSBC 86/12 board provides three independent, fully programmable 16-bit interval timers/event counters utilizing the Intel 8253 Programmable Interval Timer. Each counter is capable of operating in either BCD or binary modes. Two of these timers/counters are available to the systems designer to generate accurate time intervals under software control. Routing for the outputs and gate/trigger inputs of two of these counters is jumper selectable. The outputs may be independently routed to the 8259A Programmable Interrupt Controller and to the I/O line drivers associated with

the 8255A Programmable Peripheral Interface, or may be routed as inputs to the 8255A chip. The gate/trigger inputs may be routed to I/O terminators associated with the 8255A or as output connections from the 8255A. The third interval timer in the 8253 provides the programmable baud rate generator for the iSBC 86/12 RS232C USART serial port. In utilizing the iSBC 86/12, the systems designer simply configures, via software, each timer independently to meet system requirements. Whenever a given time delay or count is needed, software commands to the programmable timers/event counters select the desired function.

The contents of each counter may be read at any time during system operation with simple read operations for event counting applications, and special commands are included so that the contents can be ready "on the fly".

MULTIBUS™ and Multimaster Capabilities

The MULTIBUS system bus features asynchronous data transfers for the accommodation of devices with various transfer rates while maintaining maximum throughput. Twenty address lines and sixteen separate data lines eliminate the need for address/data multiplexing/demultiplexing logic used in other systems, and allow for data transfer rates up to 5 megawords/sec. A failsafe timer is included in the iSBC 86/12 board which can be used to generate an interrupt if an addressed device does not respond within 6 msec.

Multimaster Capabilities — The iSBC 86/12 board is a full computer on a single board with resources capable of supporting a great variety of OEM system requirements. For those applications requiring additional processing capacity and the benefits of multiprocessing (i.e., several CPUs and/or controllers logically sharing system tasks through communication over the system bus), the iSBC 86/12 board provides full MULTIBUS arbitration control logic. This control logic allows up to three iSBC 86/12 boards or other bus masters, including iSBC 80 family MULTIBUS compatible 8-bit single board computers, to share the system bus in serial (daisy chain) priority fashion, and up to 16 masters to share the MULTIBUS with the addition of an external priority network. The MULTIBUS arbitration logic operates synchronously with a MULTIBUS clock (provided by the iSBC 86/12 board or optionally provided directly from the MULTIBUS System Bus) while data is transferred via a handshake between the master and slave modules. This allows different speed controllers to share resources on the same bus, and transfers via the bus proceed asynchronously. Thus, transfer speed is dependent on transmitting and receiving devices only. This design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations, high speed direct memory access (DMA) operations, and high speed peripheral control, but are by no means limited to these three.

Interrupt Capability

The iSBC 86/12 board provides 9 vectored interrupt levels. The highest level is the NMI (Non-Maskable Interrupt) line which is directly tied to the 8086 CPU. This interrupt cannot be inhibited by software and is typically used for signalling catastrophic events (e.g., power failure).

The Intel 8259A Programmable Interrupt Controller (PIC) provides vectoring for the next eight interrupt levels.

The PIC accepts interrupt requests from the programmable parallel and serial I/O interfaces, the programmable timers, the system bus, or directly from peripheral equipment. The PIC then determines which of the incoming requests is of the highest priority, determines whether this request is of higher priority than the level currently being serviced, and, if appropriate, issues an interrupt to the CPU. Any combination of interrupt levels may be masked, via software, by storing a single byte in the interrupt mask register of the PIC. The PIC generates a unique memory address for each interrupt level. These addresses contain unique instruction pointers and code segment offset values (for expanded memory operation) for each interrupt level. In systems requiring additional interrupt levels, slave 8259A PIC's may be interfaced via the MULTIBUS system bus, to generate additional vector addresses, yielding a total of 65 unique interrupt levels.

Interrupt Request Generation — Interrupt requests may originate from 16 sources. Two jumper selectable interrupt requests can be automatically generated by the programmable peripheral interface.

Two jumper selectable interrupt requests can be automatically generated by the USART when a character is ready to be transferred to the CPU or a character is ready to be transmitted.

A jumper selectable request can also be generated by each of the programmable timers. Eight additional interrupt request lines are available to the user for direct interface to user designated peripheral devices via the system bus, and two interrupt request lines may be jumper routed directly from peripherals via the parallel I/O driver/terminator section.

Power-Fail Control

Control logic is also included to accept a power-fail interrupt in conjunction with the AC-low signal from the iSBC 635 Power Supply or equivalent.

Expansion Capabilities

Memory and I/O capacity may be expanded and additional functions added using Intel MULTIBUS compatible expansion boards. High speed integer and floating point arithmetic capabilities may be added by using the iSBC 310 high speed mathematics unit. Memory may be expanded to 1 megabyte by adding user specified combinations of RAM boards, EPROM boards, or combination boards. Input/output capacity may be increased by adding digital I/O and analog I/O expansion boards. Mass storage capability may be achieved by adding single or double density diskette controllers. Modular expandable backplanes and cardcages are available to support multiboard systems.

III. THE iSBC™ 957 PACKAGE

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package extends the software development capabilities of the Intellec Microcomputer Development systems to the Intel 8086 CPU. Programs for the 8086 may be written in PL/M-86 and/or assembly language and compiled or assembled on the Intellec system. These programs may then be downloaded from an Intellec ISIS-II disk file to the iSBC 86/12 board for execution and debug. The programs will execute at the full 5 MHz clock rate of the 8086 CPU with no speed degradation caused by the iSBC 957 hardware or software. Special communication software allows transparent access to the powerful interactive debug commands in the iSBC 86/12 monitor from the Intellec console terminal. These debug commands include single-step instruction execution, execution with breakpoints, memory and register displays, memory searches, comparison of two memory blocks and several other commands. After a debugging session, the debugged program code may be uploaded from the iSBC 86/12 board to an Intellec ISIS-II disk file.

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package consists of the following:

a. Four Intel 2716 EPROMs which contain the system monitor program for the iSBC 86/12 board.

b. An ISIS-II diskette containing loader software for execution in the Intellec which provides for communications between the user or an Intellec ISIS-II file and the iSBC 86/12 board. Also included on the diskette are a library of routines for system console I/O.

c. Four cable assemblies used for transmitting commands, code and data between the iSBC 86/12 board and the Intellec system.

d. An iSBC 530 adapter assembly which converts serial communications signals from current loop to RS232C.

e. Line drivers and terminators used for the iSBC 86/12 parallel ports.

f. A small printed circuit board which is plugged into an iSBC 86/12 receiver/terminator socket and is used when program code is downloaded or uploaded using the parallel cable.

iSBC™—Intellec™ Configurations

There are two distinct functional configurations for the iSBC 957 package; one configuration for the Intellec Series II, Models 220 or 230 development systems and another for the Intellec 800 series development systems.

Intellec Series II System Configurations

When used with Intellec Series II Model 220 or 230 systems, a set of cables are used to connect the serial I/O port edge connector on the iSBC 86/12 board and the SERIAL 1 output port on the Intellec system. This configuration is shown in Figure 2. How this system functions is explained in the following paragraphs.

The SERIAL 1 port on the Intellec Series II Model 220 or 230 system is an RS232 port which is designed for use with a data terminal. This port may be used on the Intellec system for interfacing to RS232 devices such as CRT terminals or printers. The serial ports on the iSBC 86/12 board and the Intellec systems are connected as shown in the Figure 2. The flat ribbon cable connected to the iSBC 86/12 board has an edge connector for connecting to the board on one end and a standard RS232 connector on the other end. The second cable, the RS232 Up/Down Load cable, has an RS232 connector on each end. This cable, however,

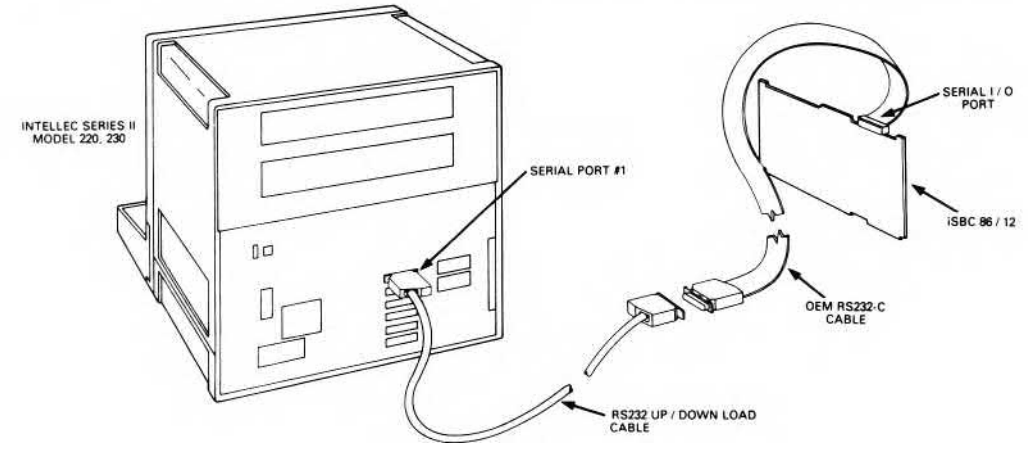


Figure 2. Intellec™ Series II Model 220, 230—iSBC™ 86/12 Configuration

is not a standard cable with the RS232 signals bussed between identically numbered pins on each of the connectors. The schematic for the cable is shown in Figure 3. Note that the TXD (transmit data) and the RXD (receive data) and the RTS (ready to send) and the CTS (clear to send) signals have been crossed. This is done because both the Intellec system and the iSBC 86/12 board are configured to act as data sets which are communicating with data terminals. Swapping these signals permits the units to communicate directly with no modifications to the Intellec or iSBC 86/12 systems themselves.

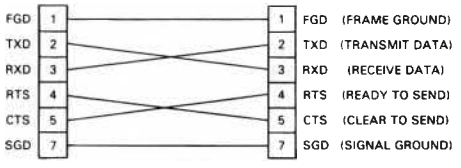


Figure 3. Intellec™—iSBC™ 86/12 RS232 UP/DOWN LOAD Cable

The software in the Intellec system accepts characters output from the iSBC 86/12 board through the Intellec SERIAL 1 port. The software then outputs these characters on the CRT monitor built into the Intellec Series II Model 220 or 230. In a similar fashion, characters input from the Intellec keyboard are passed down the serial link to the iSBC 86/12 monitor program. The integrated CRT monitor and keyboard on the Intellec system then becomes the "virtual terminal" of the iSBC 86/12 monitor program. If this were the only function of the iSBC 957 package, there would be no real benefit to the user. However, when the "virtual terminal" capability is combined with the capability to download and upload program code and data files between the Intellec ISIS-II file system and the iSBC 86/12 board, a very powerful software development tool is realized. The software in the Intellec system must examine the commands which are input from the keyboard and in the case of the LOAD and TRANSFER commands (see later sections for details on monitor commands), the software must open and read or write ISIS-II disk files.

Transfer rates using Intellec Series II Model 220 or 230 system are 9600 baud when transferring hexadecimal object files to or from a disk file and 600 baud when transferring commands between the iSBC 86/12 board and the CRT monitor and keyboard. With a 9600 baud transfer rate, it is possible to load 64K bytes of memory in about four minutes.

Intellec 800 System Configurations

The iSBC 957 package may be used with the Intellec 800 system in four different configurations. These four configurations are determined by two

variable is whether or not a parallel cable is used for uploading and downloading hexadecimal object files. Figures 4A and 4B illustrate the four configurations.

In Figure 4A, the configuration shows the TTY port of the Intellec 800 system connected to the iSBC 86/12 serial port using two cables and an iSBC 530 teletypewriter adapter. The TTY port of the Intellec 800 system is designed for using a teletypewriter as the Intellec console device. To use this port for communication with the iSBC 86/12 board, the current loop TTY signal must be converted to an RS232 compatible voltage signal. This function is performed by the iSBC 530 adapter. [illegible] Load cable described above. A schematic for this cable and all other components of the iSBC 957 package are included with the delivered product.

The transfer rate for both commands and data when the TTY port is connected to the iSBC 86/12 board is 110 baud. This means to download even moderately sized programs would require large amounts of time, several minutes or even hours. However, much faster times may be achieved by using the parallel ports of the iSBC 86/12 board and the Intellec system for downloading program files. This parallel port used on the Intellec 800 system is the output port labeled PROM which is normally used with the Universal Prom Pro-

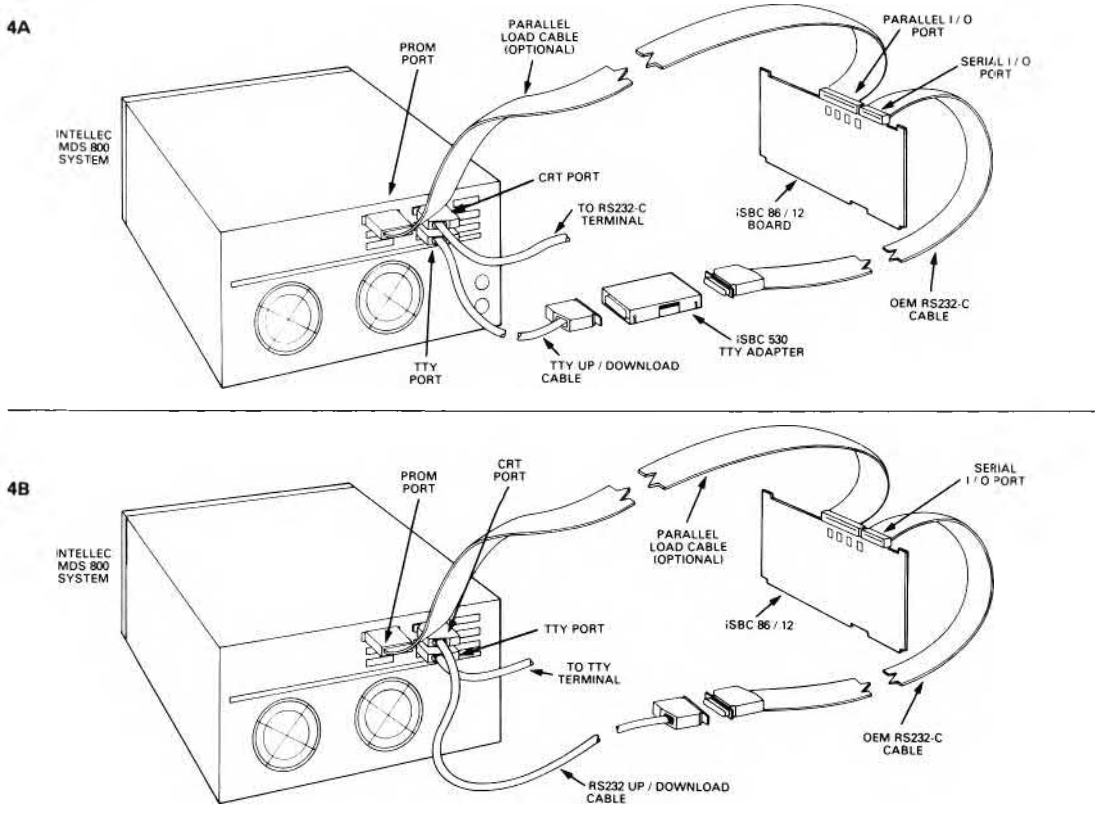


Figure 4A, 4B. Intellec™ 800—iSBC™ 86/12 Configurations

grammer. A cable is connected between the Intellec PROM port and the parallel I/O port, J1 of the iSBC 86/12 board. Parallel port B of the iSBC 86/12 board is used for the 8-bit byte transfers from the Intellec system to the iSBC 86/12 board, port A is used for the byte transfers from the iSBC 86/12 board to the Intellec system and port C is used for controlling the byte transfers. A special status adapter piggyback board must be inserted into a receiver/terminator socket of the iSBC 86/12 board. This status adapter circuit is required to provide the necessary handshaking signals from the iSBC 86/12 parallel ports to the Intellec PROM port.

The transfer rate achieved when downloading and uploading hexadecimal object files with the parallel cable is approximately 1,000 bytes per second. The time required to load 64K bytes of memory is approximately 2½ minutes.

Figure 4B shows a configuration with the Intellec 800 CRT port connected to the serial port of the iSBC 86/12 board. The TTY port of the Intellec 800 system is connected to a teletypewriter or some other current loop device to act as a system console. The optional parallel load cable is also shown. The cables used for this configuration are the same as those used with the Intellec Series II Configurations. Command transfer rates require 110 baud because the TTY port of the Intellec 800 system is used for communicating with the console device. However, hexadecimal object files can be loaded at 9600 baud since this operation uses only the Intellec to iSBC 86/12 RS232 link.

It is also possible to download files with the parallel cable, this mode being somewhat faster than the serial download mode (2½ minutes versus four minutes for 64K bytes of memory). Table I contains a summary of the command and memory transfer rates for each of the Intellec-iSBC 86/12 configurations.

Comparing the Intellec 800 configurations shown in Table 1 and in Figures 4A and 4B it should be noted:

1. Using the TTY port (Figure 4A) of the Intellec 800 system for communications with the iSBC 86/12 board (essentially) requires installation of the parallel cable and jumper modifications for downloading and uploading files, and thus, prevents the use of the parallel ports for other I/O functions.
2. Using the CRT port (Figure 4B) of the Intellec 800 system for communication with the iSBC 86/12 board provides for a fast serial download capability, thus freeing the parallel ports for other uses. However, this configuration requires a teletypewriter or a CRT capable of accepting a current loop input signal as the Intellec system console.

Table 1

COMMAND AND MEMORY TRANSFER RATES FOR INTELLEC—iSBC™ 86/12 CONFIGURATIONS

	INTELLEC SERIES II 220/230 SERIAL PORT TO iSBC 86/12	INTELLEC 800 TTY PORT TO iSBC 86/12	INTELLEC 800 CRT PORT TO iSBC 86/12
Effective Command Rate	600 Baud	110 Baud	110 Baud*
Load/Transfer Rate			
Serial	9600 Baud	110 Baud	9600 Baud
Parallel	N/A	1K bytes/sec**	1K bytes/sec**
Approximate Time to load 64K bytes of memory			
Serial	4 minutes	5 hours	4 minutes
Parallel	N/A	2.5 minutes	2.5 minutes

*The actual baud rate of the Intellec—iSBC 86/12 link is 9600 baud, but the effective command rate is determined by the slower Intellec— console serial link.

**Transmission rate over the parallel link is determined by the speed of the two processors and is approximately 1K bytes per second.

IV. THE iSBC 957—iSBC 86/12 MONITOR PROGRAM

The iSBC 86/12 monitor program is an EPROM resident program which facilitates debugging of user written programs. The monitor program used in the iSBC 86/12 board with the iSBC 957 package is the same monitor program written to interface the iSBC 86/12 directly to an RS232C data terminal. When interfaced directly to a terminal, the iSBC 86/12 board functions in a stand-alone environment communicating directly with the user via the data terminal. A user may use the monitor for entering small programs in hexadecimal format, executing a program, examining registers and memory contents, etc.

To use the monitor program with an Intellec system, the proper cables must be installed and the iSBC 957 Loader program must be loaded into Intellec memory and executed. The Loader program is resident on a file named SBC861, and when executed, the Loader outputs a sign-on message. Next, the iSBC 86/12 monitor program must be started and the baud rate of the iSBC 86/12 to Intellec serial communications link must be determined. This is done by pressing the RESET switch on the chassis

Table 2
MONITOR COMMAND LIST

COMMAND	FUNCTION AND SYNTAX
L Load Hex Object File	Loads hexadecimal object file from Intellec into iSBC 86/12 memory using serial (S) or parallel (P) mode. L {S|P},<filename>[,<bias addr>]<cr>
T Transfer Hex Object File	Transfers blocks of iSBC 86/12 memory to Intellec as a hex object file using serial (S) or parallel (P) mode. T[X] {S|P} ,<start addr>,<end addr>,<filename> [,<exec addr>]<cr>
E Exit	Exits the loader program and returns to ISIS. E<cr>
N Single Step	Executes one user program instruction. N[<addr>],[[<addr>],[*<cr>
G Go	Transfers control of the 8086 CPU to the user program with up to 2 optional breakpoints. G[<start addr>][,<break 1 addr> [,<break 2 addr>]]<cr>
S Substitute Memory	Displays/modifies memory locations in byte or word format. S[W]<addr>,[[new contents],]*<cr>
X Examine/Modify Register	Displays/modifies 8086 CPU registers. X[<reg>][[<new contents>],]*<cr>
D Display Memory	Displays contents of a memory block in byte or word format. D[W]<start addr>[,<end addr>]<cr>
M Move	Moves contents of a memory block. M<start addr>,<end addr>,<destination addr><cr>
C Compare	Compares two memory blocks. C<start addr>,<end addr>,<destination addr><cr>
F Find	Searches a memory block for a byte or word constant. F[W]<start addr>,<end addr>,<data><cr>
H Hex Arithmetic	Performs hexadecimal addition and subtraction. H<data 1>,<data 2><cr>
I Port Input	Inputs and displays byte or word data from input port. I[W]<port addr>,[,]*<cr>
O Port Output	Outputs byte or word data to output port. O[W]<port addr>,<data>[,<data>]*<cr>

Syntax conventions used in the command structure are as follows:

- [A] indicates that "A" is optional
- [A]* indicates one or more optional iterations of "A"
- <B> indicates that "B" is a variable
- {A|B} indicates "A" or "B"
- <cr> indicates a carriage return is entered

Numeric arguments can be expressed as a number, the contents of a register, or the sum or difference of numbers and register contents. Thus, addresses and data can be expressed as follows:

```
addr     ::= [<expr>:]<expr>
expr     ::= <number>|<register>|<expr> {+|-} <number>|
             <expr> {+|-} <register>
register ::= AX|BX|CX|DX|SP|BP|SI|DI|CS|DS|SS|ES|IP|FL
number   ::= <digit>|<digit><number>
digit    ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
```

Numeric fields within arguments are entered as hexadecimal numbers. The valid range of numerical values is from 0000-FFFF. Larger numbers may be entered, but only the last four digits (or two in the case of byte values) are significant. Leading zeros may be omitted.

An address argument consists of a segment value and an offset value separated by a colon (:). If a segment value is not specified, the default segment value is the CS register value.

containing the iSBC 86/12 board and typing two "U"s on the Intellec console. The ASCII uppercase character U has a binary pattern of alternating ones and zeros, the iSBC 86/12 monitor uses this pattern to determine the baud rate of the serial link. After the baud rate has been determined, the monitor program outputs a sign-on message to the console. An example of loader program execution and monitor program initialization is shown below (user entered characters are underlined).

:F1:SBC861
ISIS-II iSBC 86/12 LOADER, Vx.x
(user resets iSBC 86/12 board and types two "U"s)
iSBC 86/12 MONITOR, Vy.y

The monitor prompts with a period "." when it is ready for a command. The user can then enter a command file, which consists of a one- or two-character command followed by zero, one, or more arguments. The command may be separated from the first argument by an optional single space; a single comma is required as a delimiter between arguments. The command line is terminated by a carriage return or a comma depending on the command, and no action takes place until the command terminator is sensed. The user can cancel a command before entering the command terminator by pressing any illegal key (e.g., rubout or Control-X).

Table 2 contains a summary of the loader and monitor commands. These commands will not be explained in detail; instead, the next section of the application note will show examples of using these loader and monitor commands. The iSBC 957 User's Guide referenced at the front of this document does, however, contain a complete description of each of the monitor and loader commands.

Table 3 contains a list of the 8086 hardware registers and abbreviations used by the monitor program.

Table 3
8086 CPU REGISTERS

REGISTER NAME	ABBREVIATION
Accumulator	AX
Base	BX
Count	CX
Data	DX
Stack Pointer	SP
Base Pointer	BP
Source Index	SI
Destination Index	DI
Code Segment	CS
Data Segment	DS
Stack Segment	SS
Extra Segment	ES
Instruction Pointer	IP
Flag	FL

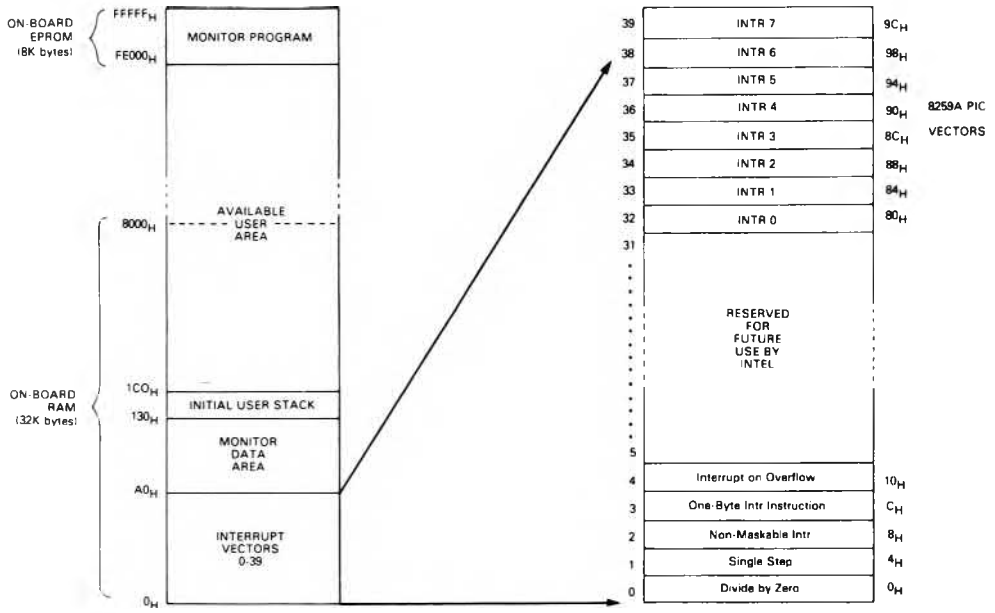


Figure 5. Memory Map of iSBC™ 86/12 Memory With Monitor Program

Figure 5 contains a memory map of the iSBC 86/12 memory with the monitor program. Note that the monitor uses the top 8K bytes of memory for its program code and the first 384 bytes of memory (locations Ø hex to 17F hex) for monitor and user stack, data and interrupt vectors. When the monitor program is reset, the segment registers, the IP and the flags are set to Ø; and the SP is set to Ø1CØH allowing 64 bytes for the user's stack. If 64 bytes is not sufficient for the user's application program, the SP should be set to some other value. The monitor program sets the single-step, one-byte instruction trap and non-maskable interrupt vectors to monitor entry points. The monitor also sets the 8259A Priority Interrupt Controller to fully nested mode with level Ø at the highest priority and all interrupts unmasked. The eight interrupt vector addresses for the 8259A are also set to addresses in the monitor. User programs may change the 8259A interrupt vectors to interrupt service routine addresses within the user programs; it is not necessary for users to program the 8259A chip directly. When an interrupt occurs, control passes to either the monitor or directly to user code depending on the address stored in the vector location. When the monitor responds to an interrupt, it acknowledges the interrupt and displays the interrupt level, CS and IP register values and next instruction byte on the system console (e.g., I = 3 @ 100:230F F5).

When a user requests a breakpoint with a "G" command, the monitor inserts the single byte instruction trap instructions (INT 3) in the location where the breakpoint is requested. It is also possible for the user to code an INT 3 instruction in his program. When a user coded INT 3 instruction is executed, the monitor will be re-entered and a line with the format @<CS Value>:<IP Value> <Instruction byte> will be displayed (e.g., @1200:3FO2 F5).

Included on the diskette with the Loader program are two libraries containing I/O routines for the console. The library files are named SBCIOS.LIB and SBCIOL.LIB; they contain similar routines. The routines in SBCIOS.LIB are written to be called with intrasegment subroutine calls, a PL/M-86 module compiled with the "small" control generates this type of call. The routines in SBCIOL.LIB are written to be called with intersegment subroutine calls, a PL/M-86 module compiled with either the "medium" or "large" control generates this type of call.

The console input output routines, CI and CO, contained in the library should be used when performing character input and output on the console. Example PL/M-86 calls to the two routines are:

```
CO: PROCEDURE (X) EXTERNAL;
    DECLARE X BYTE;
    END CO;
      .
      .
DECLARE INPUT$CHAR,
        OUTPUT$CHAR BYTE;
      .
      .
      .
INPUT$CHAR = CI;
      .
      .
      .
CALL CO(OUTPUT$CHAR);
      .
      .
      .
```

General Comments on Use of the iSBC 957 Package

1. If the iSBC 86/12 board is reset any time after the initial baud rate search, it is not necessary to reload the iSBC 957 Loader program or to download the program code a second time to the iSBC 86/12 board. It is only necessary to re-establish the communications link by typing two "U"s for the baud rate search.
2. The iSBC 86/12 board should not be plugged into an available card slot in an Intellec chassis; a separate chassis should be used. There are at least three reasons for this:
   a. There is only one RESET signal available on the Intellec system bus. Thus, each processor may not be reset independently. This means that the iSBC 86/12 board cannot be reset without re-booting the ISIS-II operating system and restarting the iSBC 957 Loader.
   b. The Intellec system uses five of the eight available interrupts on the system bus. This severely restricts the range of interrupts available to the iSBC 86/12 board. Also, the iSBC 86/12 board cannot turn-off the interrupt lamps on the Intellec front panel.
   c. The iSBC 86/12 board may address up to 1 Megabyte of memory using a 20 bit address. Many Intellec systems contain boards which generate and decode only the low order 16 address bits. For example, the iSBC 016 memory expansion board and the Intellec 800 systems is difficult since the boards which decode only 16 bits will force "holes" in the address space above 64K.
3. The iSBC 86/12 board is delivered with two inputs to the 8259A Priority Interrupt Controller connected. Interrupt request 2 (IR2) is connected to the counter Ø output of the 8253 Programmable Interval Timer. IR5 is connected to the INT5/signal of the MULTIBUS System Bus. If these interrupts are not desired, the wire wrap jumpers making the connections should be removed from the iSBC 86/12 board. A particular problem may exist with the counter Ø connection to IR2. If the 8253 counter Ø is not specifically initialized with software, a low frequency square wave output will exist at counter Ø's output. This may cause unwanted interrupts when interrupts are enabled by user programs.
4. If the iSBC 86/12 board is used in a system with expansion boards, it is important that the MULTIBUS bus exchange pins be properly jumpered. For example, if the iSBC 86/12 board is used with an iSBC 032 expansion memory board in a system, the BPRN/ MULTIBUS pin for the iSBC 86/12 board should be grounded.

   In addition, if any interrupts are used with the iSBC 86/12 board the BPRN/ pin must be grounded. This is true in both single and multiple board systems.
5. Certain user systems require more than one single board computer in the system for performing the functions required by the application. The MULTIBUS System Bus has been specifically designed to permit multiple CPU boards to communicate and to share system resources. However, debugging systems with multiple CPUs has always posed somewhat of a problem. The iSBC 957 package provides a solution to this problem. The serial cable which connects the iSBC 86/12 board to the Intellec system may be removed after the program has been downloaded to the iSBC 86/12 board. A console CRT may then be connected directly to the iSBC 86/12 board and the monitor program may be used to debug the program running on the board. Other iSBC 86/12 boards may also be downloaded from the Intellec system and then switched to their own local terminals. An 8-bit processor board, such as the iSBC 80/30 board, may also be included

in the system and ICE-85™ may be used for debugging the iSBC 80/30 program concurrently with the iSBC 86/12 programs. Using this scheme, it is possible to debug a system which has several CPU boards by setting breakpoints and using other debugging features on each of the individual CPUs.

V. MATRIX MULTIPLICATION EXAMPLE

To illustrate how the iSBC 957 package can be used to assist in the writing and debugging of 8086 programs on the iSBC 86/12 board, an example program of a matrix multiplication will be presented. The example chosen has been intentionally kept simple and straightforward. The emphasis of this section will be to document the steps required to assemble, compile, link, locate and debug software using an Intellec system, the iSBC 957 package and the iSBC 86/12 board. Part of the example will be written in 8086 assembly language and part in PL/M-86.

The main program is written in PL/M-86. The main program first performs some initialization and the matrix multiplication, then the program calls an assembly language procedure (subroutine), a PL/M-86 procedure and the console output procedure CO supplied in the I/O library on the iSBC 957 diskette. A flow diagram for the example program is shown in Figure 6.

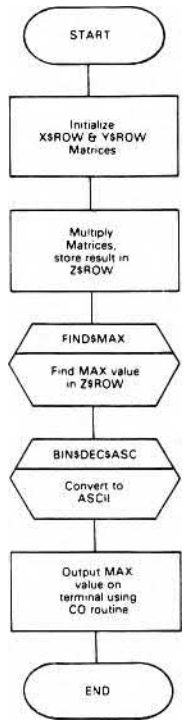


Figure 6.
Flow Diagram of Matrix Multiplication Example

Explanation of the Program Code

The program code is contained in three software modules EXECUTION$VEHICLE, FIND, and SBCCO. EXECUTION$VEHICLE contains the main program coded in PL/M-86 and the binary to ASCII conversion procedure BINDECASC also coded in PL/M-86. The module FIND contains the assembly language procedure FIND$MX which searches a matrix for its maximum value. The module SBCCO resides in the library of console I/O routines supplied with the iSBC 957 package. The procedure CO will be used from this library.

The program code for the EXECUTION$VEHICLE and FIND modules will be explained in the following paragraphs. Appendix B contains compilation and assembly listings for the two modules; also contained in Appendix B is a memory and debug map for the linked modules. The listings contain circled reference letters (e.g., Ⓐ) which are referred to by the code description below. The listings in the appendix have been printed on fold-out pages so that they may easily be seen when reading the text.

Much of the description given below assumes that the reader is familiar with the PL/M-86 language and compiler, the 8086 assembler, and the link and locate program QRL86. It is recommended that the reader have at least a cursory knowledge of these subjects. The Intel literature for these subjects is listed near the front of this application note.

The EXECUTION$VEHICLE Module

Ⓐ The first section of the module includes introductory comments and then statements to declare the matrices, other variables, and procedures used in the program. Note that the matrix dimensions are declared using the literals M, N, and P which are initially set to 6, 5, and 3. Later in this note, other values for M, N, and P will be used.

Ⓑ The next section of code contains the statements which initialize the two matrices that will be multiplied X$ROW and Y$ROW.

As a result of this initialization, the two matrices will contain values as shown in Figure 7.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{bmatrix} \qquad \begin{bmatrix} 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \end{bmatrix}$$

X$ROW (6X5) **Y$ROW (5X3)**

Figure 7.
X$ROW and Y$ROW Matrices After Initialization

(C) The next program section performs the matrix multiplication. The algorithm required to multiply two matrices X and Y, storing the result in a third matrix Z is:

$$Z_{mp} = \sum_{i=1}^{n} X_{mi} * Y_{ip}$$

Assuming X to be 6X5 matrix and Y a 5X3 matrix then

$$Z_{11} = X_{11}Y_{11} + X_{12}Y_{21} + X_{13}Y_{31} + X_{14}X_{41} + X_{15}Y_{51}$$

Thus, the upper left term is equal to the sum of the products of the top row of the X matrix times the left column of the Y matrix. The result that is obtained by multiplying the two matrices X$ROW and Y$ROW after they are initialized as explained above, is shown in Figure 8.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & -5 & -10 \\ 0 & -10 & -20 \\ 0 & -15 & -30 \\ 0 & -20 & -40 \\ 0 & -25 & -50 \end{bmatrix}$$

Z$ROW (6X3)

Figure 8. Result of Multiplying the Initialized Matrices X$ROW and Y$ROW

(D) The external assembly language procedure FIND$MX is called to determine the maximum value in the matrix. The procedure is a typed procedure and returns the maximum value to the calling program which stores it in the integer variable MAX.

(E) The maximum value is then converted to a six (6) digit ASCII character string by the procedure BINDECASC. The character string is stored in the array MAXASCARRAY, which contains the sign of the number and five (5) digits for the magnitude.

(F) Finally, the characters "MAX VALUE =" are output on the system console followed by the 6 ASCII characters containing the maximum value. The PL/M-86 built-in procedure SIZE returns the number of bytes of the array TEXT as a word value. The PL/M-86 built-in procedure SIGNED changes the type of the value from WORD to INTEGER. This is required so that the type of the arguments in the DO statement agree. The console output procedure CO is used to output the characters on the system console.

(G) Also contained in the module MATRIX.PLM is the binary to ASCII conversion procedure BINDECASC. The first portion of the code contains the comments explaining the parameters and the calling sequence followed by the declarations. Note that the address of the array where the characters are to be stored is passed to the procedure and that the characters will be stored in the array using based variables. The next section of the code stores either a + or − sign in the first character position of the ASCII array and stores the absolute value of VALUE in the variable TEMP. Finally, the binary value is converted to ASCII using the algorithm explained in the comments. The MOD operator returns the remainder of the division by 10. The UNSIGN built-in procedure is required to change the type of the expression from INTEGER to WORD.

The FIND Module

(H) The FIND module contains the assembly language procedure FINDMX. The calling sequence and the parameters are explained in the comments at the beginning of the listing. Note that the label FINDMX has been declared PUBLIC so the link program can fill in its address in the CALL statement in the main program of module EXECUTION$VEHICLE.

(I) The FIND module will contain three segments: a data segment, a stack segment and a code segment. It will be both convenient and pragmatic to append these three segments to the code, data and stack segments created by the

compiler for the EXECUTION$VEHICLE module. To accomplish this, the three segments must be given the same SEGMENT and CLASS names as those given these segments by the compiler. The SEGMENT and CLASS names used by the compiler are CODE, DATA, and STACK. The GROUP statements are used to place the segments DATA and STACK in the group DGROUP and the segment CODE in the group CGROUP. These group definitions conform with the group definitions generated by the PL/M-86 compiler when the SMALL size control option is used. A group is a collection of segments which requires less than 64K bytes of memory.

The ASSUME directive informs the assembler that the DS and SS registers will contain the base address of DGROUP and the CS register will contain the base address of CGROUP. This information will be used by the assembler when constructing machine instructions.

(J) The first segment appearing in the module is the data segment. The order of the segments is arbitrary, although it is recommended that the data segment precede the code segment to minimize forward references to variables which may cause the assembler to generate longer instruction codes. The data segment is declared PUBLIC, aligned on a WORD boundary and given both a segment and class name of DATA. Then follows the contents of the segment. In this particular example, only one word of storage is required. The ENDS directive indicates the end of the segment.

(K) Next comes the stack segment which is given the segment name of STACK, the combine-type attribute of STACK and the class name of STACK. The combine-type attribute of STACK assures that the stack storage required in this module will be appended to the storage required in the PL/M-86 compiled modules. Two bytes of stack are required by the code in this module, however, the monitor uses 13 words of stack when breakpoints and interrupts are used. Therefore, 14 words are reserved for the stack.

(L) Finally comes the code segment. The code segment has been given a segment name and class name of CODE and a group name of CGROUP, and has been declared PUBLIC. The alignment attribute of BYTE is specified since it is desired that the code from this module be appended directly to the code from other modules without gaps between the code modules.

The assembly language code follows next. The code for the procedure must be enclosed between a pair of PROC, ENDP statements. The PROC statement is given the label FINDMX and specified as a NEAR procedure indicating it will be called with a near (intra-segment) CALL instruction and not a far (inter-segment) CALL instruction.

The comments at the beginning of the module and adjacent to the program statements explain the function being performed by the assembly language code.

The SBCCO Module

(M) The console output procedure CO is contained in the object module SBCCO of the library file SBCIOS.LIB. SBCIOS.LIB is part of the iSBC 957 package I/O libraries. The calling sequence and parameters for CO may be seen in the external procedure declaration in the EXECUTION$VEHICLE module.

Compiling the EXECUTION$VEHICLE Module

The EXECUTION$VEHICLE module is stored on a file named MATRIX.PLM on disk device :F1:. To compile the module, the following command line is used:

– PLM86 :F1:MATRIX.PLM DEBUG

This command line will cause the module stored in the file :F1:MATRIX.PLM to be compiled. The object code generated will be stored in a file with the default name :F1:MATRIX.OBJ and the listing generated will be stored in a file with the default name :F1:MATRIX.LST. To override the default object and listing files, the NOOBJECT and NOLIST compiler control switches can be used. File names for the listing and object files may also be specified in the command line. The DEBUG compiler control switch causes the compiler to generate extra symbol and line number information which will be used during debugging of the program. A listing of the compiled EXECUTION$VEHICLE module is contained in Appendix B.

To aid in the debugging of the program, the module was compiled a second time with the following command line:

CODE DEBUG PRINT (:F1:MATRIX.XLS)

This command line specified that no object file is to be created and a listing file should be stored in the file :F1:MATRIX.XLS. The CODE compiler control switch causes the compiler to list the assembly language statements which the compiler has generated for each line of PL/M code. The listing stored in the file MATRIX.XLS is contained in Appendix C.

Assembly of the FIND Module

The assembly language module FIND is stored on a file named FIND.ASM, to assemble this module the following command line is used:

ASM86 :F1:FIND.ASM DEBUG

This command line will cause the FIND module to be assembled with the object code stored in the default file :F1:FIND.OBJ and the listing stored in the default file :F1:FIND.LST. The listing of the assembled FIND module is contained in Appendix B.

Linking and Locating the Object Module

To link and locate the object modules, the QRL86 program will be used. The QRL86 program performs both the linking and the locating of the object modules in a single step. QRL86 is primarily designed for the debugging stages of program development. Some applications may require the extended capabilities of the separate LINK and LOCATE programs when the final link and locate is performed. The command line used to invoke the QRL86 program is:

QRL86 :F1:MATRIX.OBJ, :F1:FIND.OBJ,
SBCIOS.LIB ORIGIN (1000H)

This command line will cause QRL86 to link the code from the three modules and to locate the resultant absolute object module starting at location 1000 hexadecimal. The iSBC 86/12 monitor uses the first 180H bytes of memory for the monitor stack, data and interrupt vectors, 1000H was chosen as a convenient starting address for the program. The absolute object code will be stored in a default file :F1:MATRIX (note no file name extension is used). By default, the memory and debug maps which are generated are stored in the file :F1:MATRIX.MPQ and are contained in Appendix B.

(N) The memory map contains the starting addresses and sizes of the CODE, CONST, DATA, STACK and MEMORY segments of the object module. Note that the start address indicating a CS value of 0000H and an IP value of 0002H or an absolute value of 01002H. The first two bytes of the code segment contain address values which the code generated by the compiler will use for setting up the DS and SS registers. The memory map shows the code segments from the three modules collected into the group CGROUP. The code segment from the EXECUTION$VEHICLE module is given the segment and class names of CODE and is put into CGROUP by the PL/M compiler. To assure that the code segment from the FIND module is concatenated with the code segment from the EXECUTION$VEHICLE module the identical class, segment and group names were specified in the SEGMENT and GROUP statements in the FIND module. Next, the group DGROUP is shown in the memory map. DGROUP contains 4 segments labelled CONST, DATA, STACK and MEMORY. Putting all of these segments in the same group tells the linker that they will all be in the same 64K block of memory. The SMALL size control option of the compiler, which was invoked by default, creates CGROUP, DGROUP, and the segments contained in them.

(P) The debug map contains the memory address of variables, instruction labels and the addresses of each code line of the PL/M-86 module. Notice that the variable storage labels have their addresses specified in the format (DS register value, displacement). For example, the variable TEMP has an address of DS=012AH, displacement = 000CH or an absolute address of 0136H. Instruction labels and line numbers use the format (CS register value, IP register value). Thus, line number six (6) in the module EXECUTION$VEHICLE has the address CS=0100H, IP=0B5H or 011B5H.

Object to Hex Conversion

Before downloading the program to the iSBC 86/12, the format of the object module must be converted from the absolute object module format which QRL86 creates to a hexadecimal/ASCII representation of the object module. This is done using the program OH86 with the following command line:

OH86 :F1:MATRIX TO :F1:MATRIX.HEX

Downloading and Debugging the Program

The hardware configuration used for debugging the matrix multiplication example program code was

an Intellec Series II Model 230 development system, the iSBC 957 package, an iSBC 86/12 board, and an iSBC 660 system chassis. What follows is the system-user dialog for a typical debugging session.

The first step required is to bootstrap load the ISIS-II operating system by hitting the RESET switch of the Intellec. The Intellec resident loader software is then loaded and executed. Throughout the dialog which follows operator entered characters will be underlined:

```
ISIS-II, V3.4
-SBC861

ISIS-II ISBC 86/12 LOADER, V1.2
```

To initialize the iSBC 86/12 monitor, the user must hit the RESET switch on the iSBC 660 chassis and type two "U"s on the system console. The monitor program will output a line on the console when it is properly initialized.

```
ISBC 86/12 MONITOR, V1.2
```

The monitor command "X" is typed to check that the monitor is properly operating and to examine the contents of the 8086 registers.

```
.X
AX=0000 BX=0000 CX=0000 DX=0000 SP=01C0 BP=0000 SI=0000
DI=0000 CS=0000 DS=0000 SS=0000 ES=0000 IP=0000 FL=0000
```

To download the hex object file to the iSBC 86/12, the "L" command is used. Because an Intellec Series II Model 230 is being used, a serial download is specified. The hex file name is MATRIX.HEX which is resident on disk device :F1:.

```
.LS,:F1:MATRIX.HEX
```

The "X" command is used again to examine the CPU registers. Note that the monitor has changed the contents of the CS and IP registers to the value of the starting address of the program.

```
.X
AX=0000 BX=0000 CX=0000 DX=0000 SP=01C0 BP=0000 SI=0000
DI=0000 CS=0100 DS=0000 SS=0000 ES=0000 IP=0002 FL=0000
```

The "D" command is next used to display the first 101 bytes of the program code. Unless another segment register is specified, the display command assumes all addresses specified are relative to the CS register. Thus, the code displayed will be from absolute addresses 1000 through 1100. The program code displayed may be compared with program code generated by the PL/M-86 compiler shown in Appendix C, code line 36.

```
.D0,100
0000 2A 01 FA 2E 8E 16 00 00 BC D0 00 8B EC 16 1F FB
0010 C7 06 8E 00 00 00 81 3E 8E 00 05 00 7E 03 E9 3C
0020 00 C7 06 90 00 00 00 81 3E 90 00 04 00 7E 03 E9
0030 22 00 8B 06 9E 00 B9 0A 00 F7 E9 8B 36 90 00 D1
0040 E6 89 C3 8B 0E 8E 00 89 88 10 00 81 06 90 00 01
0050 00 E9 D3 FF 81 06 8E 00 01 00 E9 B9 FF C7 06 8E
0060 00 00 00 81 3E 8E 00 04 00 7E 03 E9 40 00 C7 06
0070 90 00 00 00 81 3E 90 00 02 00 7E 03 E9 26 00 8B
0080 06 90 00 F7 D8 50 8B 06 8E 00 B9 06 00 F7 E9 8B
0090 36 90 00 D1 E6 89 C3 59 89 88 4C 00 81 06 90 00
00A0 01 00 E9 CF FF 81 06 8E 00 01 00 E9 B5 FF C7 06
00B0 92 00 00 00 81 3E 92 00 02 00 7E 03 E9 8C 00 C7
00C0 06 8E 00 00 00 81 3E 8E 00 05 00 7E 03 E9 72 00
00D0 8B 06 8E 00 B9 06 00 F7 E9 8B 36 92 00 D1 E6 89
00E0 C3 C7 80 6A 00 00 00 C7 06 90 00 00 00 81 3E 90
00F0 00 04 00 7E 03 E9 41 00 8B 06 8E 00 B9 0A 00 F7
0100 E9
```

The PL/M-86 compiler ends the main program in the EXECUTION$VEHICLE module with a halt instruction. After execution of the program it is more desirable to return to the monitor. To accomplish this, an INT 3 instruction (code=CC) will be substituted for the halt instruction (code=F4) at the address of 1B4H relative to a CS value of 100H. First the "D" command is used to verify the address of the halt instruction, then the "S" command is used to change the instruction to an INT 3 instruction.

```
.D1B4
01B4 F4
.S1B4. F4- CC
```

To execute the PL/M-86 main program, the "G" command is used. After the "G" is typed, the current contents of the IP are output, followed by the contents of the byte pointed to by the IP. A new value for the IP or breakpoint addresses may be specified before a carriage return <CR> is typed. In this example, only a <CR> is typed.

```
.G 0002- FA
MAX VALUE = -00050
@0100:01B5 55
```

The program executes and outputs the maximum value of the matrix calculated. The INT 3 instruction is executed which causes a return to the monitor. The monitor types out an at-sign (@) followed by the CS and IP register values and the first byte of the instruction following the INT 3 instruction.

The "X" command is typed to examine the CPU registers. Note that the program has set both the SS and DS registers to Ø12A. (Ø12AØH is the address of the DGROUP as shown in the memory map.)

```
.X
AX=0030 BX=0005 CX=000A DX=0000 SP=00D0 BP=00D0 SI=0001
DI=0006 CS=0100 DS=012A SS=012A ES=0000 IP=01B5 FL=F202
```

The three matrices are displayed. Note that a word

display has been specified by using the "DW" Command and that the addresses have been specified relative to the DS register. The addresses of XROW, YROW, and Z$ROW may be found in the debug map given by QRL86. Note that the values stored in the matrices are the same as those shown in Figures 8 and 9.

```
.DW DS:10,4A
0010 0000 0000 0000 0000 0000 0001 0001 0001
0020 0001 0001 0002 0002 0002 0002 0002 0003
0030 0003 0003 0003 0003 0004 0004 0004 0004
0040 0004 0005 0005 0005 0005 0005
.DW DS:4C,68
004C 0000 FFFF
0050 FFFE 0000 FFFF FFFE 0000 FFFF FFFE 0000
0060 FFFF FFFE 0000 FFFF FFFE
.DW DS:6A,8C
006A 0000 0000 0000
0070 0000 FFFB FFF6 0000 FFF6 FFEC 0000 FFF1
0080 FFE2 0000 FFEC FFD8 0000 FFE7 FFCE
```

The "G" Command is used to reset the IP register to the start address of the program (ØØ2) and to specify a breakpoint at address ØAEH, which is the address of statement 57 of the main program. Statement 57 is the point in the program after the X$ROW and Y$ROW matrices have been initialized, but before the matrix multiplication is performed. After the <CR> is typed, the program executes until the breakpoint is encountered. At this point, the monitor outputs a line specifying the number of the breakpoint, the CS and IP values and the first byte of the next instruction to be executed.

```
.G 01B5- 55 002,AE

BR1 @0100:00AE C7
```

Next, the single-step capability is used with the "N" command to execute single instructions. At any time, CPU registers may be examined or changed. In this example, the "X" command is used. Execution of succeeding instructions is caused by typing a comma (,).

```
.N 00AE- C7 ↵
00B4- 81 ,
00BA- 7E ,
00BF- C7
.X
AX=0018 BX=0018 CX=FFFE DX=0000 SP=00D0 BP=00D0 SI=0004
DI=0006 CS=0100 DS=012A SS=012A ES=0000 IP=00BF FL=F293
.N 00BF- C7 ↵
00C5- 81 ↵
00CB- 7E
```

The contents of the X$ROW and Y$ROW matrices are examined and changed with the "SW" (substitute word) command. If a comma (,) is typed after the contents of memory are displayed, then the contents are left unchanged and the next word of memory is displayed. If a value followed by a comma or <CR> is entered, then the contents are changed. If a <CR> is entered, the substitute sequence is terminated.

```
.SW DS:1A. 0001- ↵
001C 0001- ↵
001E 0001- 10
.SW DS:5A. FFFF- ↵
005C FFFE- ↵
005E 0000- 1
0060 FFFF- 64
```

After the matrices are modified, execution is resumed with the "G" command. The max value is output and the INT 3 instruction executed. Finally, the contents of the 3 matrices are displayed.

```
.G 00CB- 7E
MAX VALUE = +00400
@0100:01B5 55
.DW DS:10,BC
0010 0000 0000 0000 0000 0000 0001 0001 0010
0020 0001 0001 0002 0002 0002 0002 0002 0003
0030 0003 0003 0003 0003 0004 0004 0004 0004
0040 0004 0005 0005 0005 0005 0005 0000 FFFF
0050 FFFE 0000 FFFF FFFE 0000 FFFF FFFE 0000
0060 0064 FFFE 0000 FFFF FFFE 0000 0000 0000
0070 0000 0051 FFD8 0000 00C0 FFEC 0000 0120
0080 FFE2 0000 0180 FFD8 0000 01E0 FFCE
```

Expanding the Example Program's Memory Requirements

To illustrate how the iSBC 86/12 board may be used for executing 8086 programs which require large amounts of RAM, the example program will be modified. The matrix dimensions of the example will be changed from values of 6, 5 and 3 for the literal symbols of M, N, and P to values of 100, 50, 70. The three matrices will then be of size 100X50, 50X70, and 100X70. The memory required for these matrices is 15.5K words or 31K bytes. The data, constant, stack and memory segments which are contained in the group DGROUP will now comprise almost 32K bytes of memory.

The extra memory requirements will be supplied by using an iSBC 032 board with the iSBC 86/12 board in the iSBC 660 chassis. The iSBC 032 board is a 32K byte RAM board which is compatible with both 8- and 16-bit CPU boards. The base address of the board may be selected anywhere in a 0 to 1 megabyte range on any 16K byte boundary. 8- or 16-bit data transfers may be selected. The iSBC 032 board will be jumpered to respond to addresses in the 512K or 544K address space (20 bit hex address range to 8ØØØØH to 87FFFH). This will illustrate the capabilities of the 8086 to access a 20-bit, 1 megabyte address range.

One other modification is required to the program. The magnitude of the numbers which would result from multiplying matrices of this size would greatly exceed the capacity of the 16-bit integer storage, even with the two matrices initialized to the small

values they presently contain. To keep the example simple, the initialization values will be changed so all elements of the X$ROW matrix are set equal to 2 and all elements of the Y$ROW matrix are set equal to 3. The result of the multiplication should make all the elements of Z$ROW equal to 300.

The modified lines of program code are shown below.

```
                    /* MATRIX DIMENSIONS */
27   1        DECLARE M LITERALLY '100';
28   1        DECLARE N LITERALLY '50';
29   1        DECLARE P LITERALLY '70';


36   1        DO I = 0 TO (M-1);
37   2          DO J = 0 TO (N-1);
38   3            X$ROW(I).COL(J) = 2;
39   3          END;
40   2        END;


41   1        DO I = 0 TO (N-1);
42   2          DO J = 0 TO (P-1);
43   3            Y$ROW(I).COL(J) = 3;
44   3          END;
45   2        END;
```

The EXECUTION$VEHICLE module must be recompiled and then the three program modules must be linked and located using the QRL86 program. Specifying the SEGMENTS option of QRL86, the origin of the CODE segment which is in the group CGROUP is set at 1000H, as in the first example. However, the origin of the CONST, DATA STACK and MEMORY segments which make up the group DGROUP is set at 80000H.

```
QRL86 :F1:MATRIX.OBJ,:F1:FIND.OBJ,
SBCIOS.LIB SEGMENTS (CODE(1000H),
CONST (80000H), DATA STACK, MEMORY)
```

The memory map generated by QRL86 shows the CGROUP having a start address of 01000H and the DGROUP having a start address of 80000H.

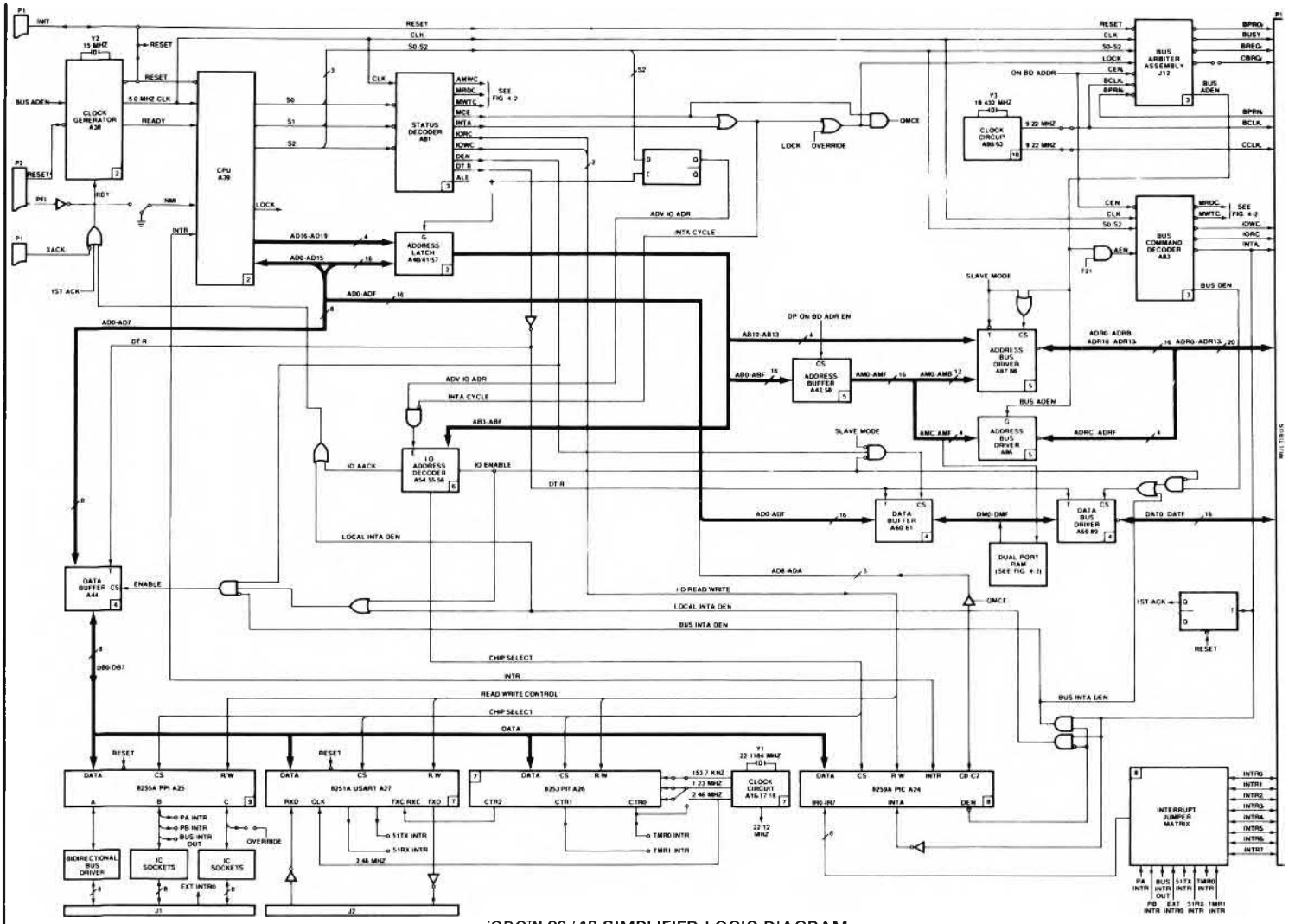
```
INVOKED BY:
QRL86 :F1:MATRIY.OBJ,:F1:FIND.OBJ,SBCIOS.LIB &
SEGMENTS(CODE(1000H),CONST(80000H),DATA,STACK,MEMORY)

INPUT MODULES INCLUDED:
:F1:MATRIY.OBJ(EXECUTIONVEHICLE)
:F1:FIND.OBJ(FIND)
SBCIOS.LIB(SBCCO)

RESULT WRITTEN TO :F1:MATRIY(EXECUTIONVEHICLE)
START ADDRESS IS (0100H,0002H)

 START   LTH  ALIGN NAME                      CLASS

01000H  298H   G   /GS/ CGROUP
01000H  21DH   W   CODE(EXECUTIONVEHICLE)     CODE
0121DH   41H   B   CODE(FIND)                 CODE
0125EH   3AH   W   CODE(SBCCO)                CODE
                   /GE/ CGROUP
80000H 7970H   G   /GS/ DGROUP
80000H    CH   W   CONST(EXECUTIONVEHICLE)    CONST
8000CH    0H   W   CONST(SBCCO)               CONST
8000CH 792AH   W   DATA(EXECUTIONVEHICLE)     DATA
87936H    2H   W   DATA(FIND)                 DATA
87938H    0H   W   DATA(SBCCO)                DATA
87940H   30H  SW   STACK                      STACK
87970H    0H   W   MEMORY                     MEMORY
                   /GE/ DGROUP
87970H    0H   G   ??SEG(FIND)                (NULL)
```

The object code is then converted to hex format and downloaded to the iSBC 86/12 board. When the program is executed, the maximum value is calculated and output on the console.

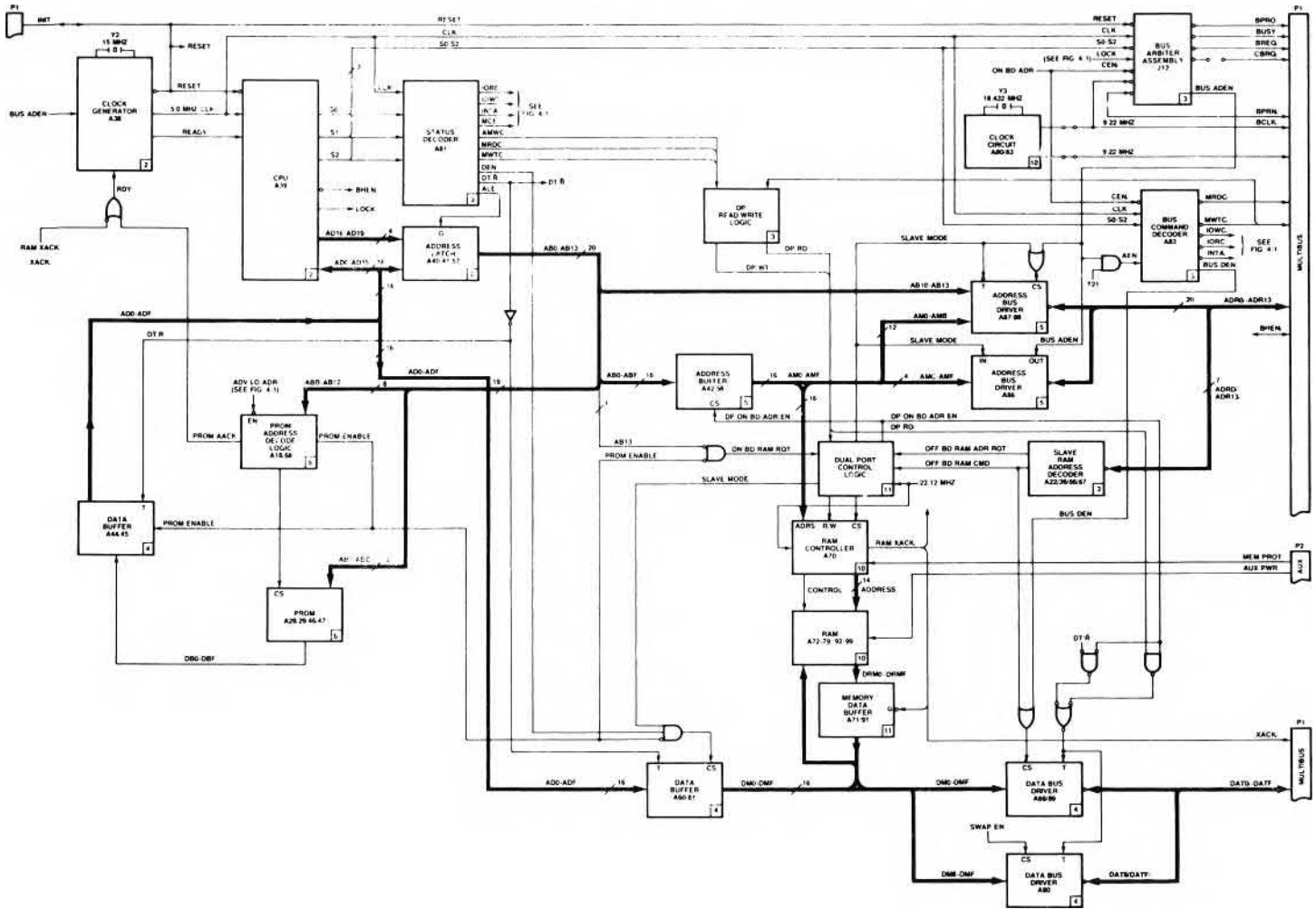
```
-SBC861

ISIS-II ISBC 86/12 LOADER, V1.2

ISBC 86/12 MONITOR,,V1.2
.LS,:F1:MATRIY.HEX

.S1AC, F4- CC
.G 0002- FA
MAX VALUE = +00300
00100:01AD 55
```

VI. CONCLUSION

This application note has described the iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package, and how this package may be used to develop and debug programs for the 8086 processor. First, the iSBC 86/12 single board computer was described, followed by a detailed description of the iSBC 957 package and the iSBC 86/12 system monitor commands. The power and versatility of the iSBC 957 package and monitor commands for developing and debugging programs for the 8086 were illustrated by a program example. In the example a program which consisted of PL/M-86 and assembly language routines was presented. The program code was explained, and the steps required to compile, assemble, link, locate, and debug the program were illustrated. Finally, a typical debugging session using the iSBC 86/12 system monitor which illustrates the powerful capabilities of the monitor was presented.



iSBC™ 86/12 SIMPLIFIED LOGIC DIAGRAM
INPUT/OUTPUT AND INTERRUPT



iSBC™ 86/12 SIMPLIFIED LOGIC DIAGRAM
ROM/EPROM AND DUAL PORT RAM

APPENDIX B

PROGRAM LISTINGS FOR EXECUTION$VEHICLE AND FIND MODULES

```
PL/M-86 COMPILER    EXECUTIONVEHICLE

ISIS-II PL/M-86 V1.Ø COMPILATION OF MODULE EXECUTIONVEHICLE
OBJECT MODULE PLACED IN :F1:MATRIX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:MATRIX.PLM DEBUG

                /*      MATRIX MULTIPLICATION EXAMPLE PROGRAM

                    PL/M-86 MAIN PROGRAM WHICH:
                        A) INITIALIZES TWO INTEGER MATRICES
                        B) MULTIPLIES THE TWO MATRICES AND STORES THE RESULT IN A
                           THIRD MATRIX
                        C) CALLS AN ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES THE
                           THIRD MATRIX FOR THE MAXIMUM VALUE
                        D) CALLS A PL/M PROCEDURE WHICH CONVERTS THE MAXIMUM VALUE
                           FROM INTEGER TO ASCII
                        E) CALLS A PROCEDURE WHICH OUTPUTS THE ASCII CHARACTERS ON
                           THE SYSTEM CONSOLE
                */

  1             EXECUTION$VEHICLE:
                DO;


                /*  FINDSMX - EXTERNAL ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES A
                        MATRIX FOR THE LARGEST ABSOLUTE MAGNITUDE.
                        PARAMETERS:
                            MATRIXSADR -  ADDRESS OF THE MATRIX TO BE SEARCHED
                            ROWS - NUMBER OF ROWS IN THE MATRIX
                            COLS - NUMBER OF COLUMNS IN THE MATRIX
                */
  2   1         FINDSMX: PROCEDURE (MATRIXSPTR, ROWS, COLS) INTEGER EXTERNAL;
  3   2         DECLARE (ROWS, COLS) INTEGER;
  4   2         DECLARE MATRIXSPTR POINTER;
  5   2         END FINDSMX;


                /*  BINSDECSASC - BINARY TO DECIMAL ASCII CONVERSION PROCEDURE
                        PARAMETERS:
                            VALUE - INTEGER VALUE TO BE CONVERTED TO ASCII
                            CHARSARRAYSADR - ADDRESS OF 6 BYTE ARRAY WHERE ASCII
                                STRING CONTAINING THE VALUE WILL BE STORED
                */
  6   1         BINSDECSASC: PROCEDURE (VALUE, CHARSARRAYSADR);

  7   2         DECLARE (VALUE, TEMP, I) INTEGER;
  8   2         DECLARE CHARSARRAYSADR POINTER;
  9   2         DECLARE (CHARSARRAY BASED CHARSARRAYSADR) (6) BYTE;

 10   2         IF VALUE < Ø THEN
 11   2         DO;
 12   3           CHARSARRAY(Ø) = '-';    /* SIGN CHARACTER */
 13   3           TEMP = -VALUE;
 14   3         END;
                ELSE
 15   2         DO;        Ø
 16   3           CHARSARRAY(Ø) = '+';
 17   3           TEMP = VALUE;
 18   3         END;
 19   2         DO I = 5 TO 1 BY -1;
 20   3           CHARSARRAY(I) = UNSIGN(TEMP MOD 1Ø) + 3ØH;
 21   3           TEMP = TEMP/1Ø;
                  /* ASCII CHARACTERS 3Ø THRU 39 HEX REPRESENT THE DIGITS Ø THRU 9. THUS
                    TO CONVERT AN INTEGER TO ASCII REPEATED DIVISIONS BY 1Ø AND ADDING
                    THE REMAINDER TO 3Ø HEX WILL ACCOMPLISH THE CONVERSION */
 22   3         END;

 23   2         END BINSDECSASC;


                /*  CO - EXTERNAL PROCEDURE TO OUTPUT A CHARACTER TO THE SYSTEM CONSOLE.
                      THIS PROCEDURE IS PART OF THE ISBC 957 LIBRARY FOR CONSOLE I/O
                      PARAMETER:
                        CHAR - ASCII CHARACTER TO BE OUTPUT ON THE CONSOLE
                */
 24   1         CO: PROCEDURE (CHAR) EXTERNAL;
 25   2         DECLARE CHAR BYTE;
 26   2         END CO;


                    /* MATRIX DIMENSIONS */
 27   1         DECLARE M LITERALLY '6';
 28   1         DECLARE N LITERALLY '5';
 29   1         DECLARE P LITERALLY '3';

                /* THE THREE MATRICES ARE DECLARED AS ARRAYS OF STRUCTURES.  XSROW IS COMPOSED
                   OF M STRUCTURES EACH OF WHICH IS COMPOSED OF N INTEGER ELEMENTS.  THUS
                   XSROW MAY BE THOUGHT OF AS A M X N MATRIX.  THE MATRIX WILL BE STORED AS
                   A ROW-ORDER MATRIX WITH THE ELEMENTS OF EACH ROW STORED IN ADJACENT MEMORY
                   LOCATIONS.  YSROW IS DECLARED AS A N X P MATRIX AND ZSROW AS A N X P MATRIX */
 30   1         DECLARE XSROW(M) STRUCTURE (COL(N) INTEGER);
 31   1         DECLARE YSROW(N) STRUCTURE (COL(P) INTEGER);
 32   1         DECLARE ZSROW(M) STRUCTURE (COL(P) INTEGER);
 33   1         DECLARE (I,J,K,MAX) INTEGER;
 34   1         DECLARE MAXSASCSARRAY(6) BYTE;
 35   1         DECLARE TEXT(*) BYTE DATA ('MAX VALUE = ');
```

```
                /* INITIALIZE XSROW SUCH THAT THE FIRST ROW IS SET EQUAL TO 0, THE SECOND
                   ROW EQUAL TO 1, THE THIRD ROW EQUAL TO 2, ETC.  */
 36   1        DO I = 0 TO (M-1);
 37   2          DO J = 0 TO (N-1);
 38   3            XSROW(I).COL(J) = I;
 39   3          END;
 40   2        END;
B
                /* INITIALIZE YSROW SUCH THAT THE FIRST COLUMN IS SET EQUAL TO 0, THE
                   SECOND COLUMN EQUAL TO -1, AND THE THIRD COLUMN EQUAL TO -2.  */
 41   1        DO I = 0 TO (N-1);
 42   2          DO J = 0 TO (P-1);
 43   3            YSROW(I).COL(J) = -J;
 44   3          END;
 45   2        END;
                /* PERFORM MATRIX MULTIPLICATION  */
 46   1        DO K = 0 TO (P-1);
 47   2          DO I = 0 TO (M-1);
 48   3            ZSROW(I).COL(K) = 0;  /* SET ZSROW ELEMENT TO 0 */
 49   3            DO J = 0 TO (N-1); /* SUM THE PRODUCT OF XSROW ROW TERMS AND YSROW COLUMN TERMS */
C
 50   4              ZSROW(I).COL(K) = ZSROW(I).COL(K) + ( XSROW(I).COL(J) * YSROW(J).COL(K) );
 51   4            END;
 52   3          END;
 53   2        END;
D
 54   1        MAX = FINDSMX (@ZSROW, M, P); /* FIND MAX VALUE OF ZSROW */
E
 55   1        CALL BINSDECSASC (MAX, @MAXSASCSARRAY); /* CONVERT TO DECIMAL ASCII */
 56   1        DO I = 0 TO (SIGNED(SIZE(TEXT)) - 1); /* OUTPUT HEADER TEXT */
 57   2          CALL CO(TEXT(I));
 58   2        END;
F
 59   1        DO I = 0 TO 5; /* OUTPUT ASCII MAX VALUE */
 60   2          CALL CO(MAXSASCSARRAY(I));
 61   2        END;

 62   1        END EXECUTIONSVEHICLE;
```

```
MODULE INFORMATION:

     CODE AREA SIZE     = 0225H    549D
     CONSTANT AREA SIZE = 000CH     12D
     VARIABLE AREA SIZE = 0090H    144D
     MAXIMUM STACK SIZE = 0008H      8D
     137 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
ISIS-II MCS-86 ASSEMBLER      ASSEMBLY OF MODULE FIND
OBJECT MODULE PLACED IN :F1:FIND.OBJ
ASSEMBLER INVOKED BY: ASM86 :F1:FIND.ASM DEBUG

LOC  OBJ                LINE   SOURCE

                          1            NAME    FIND
                          2            PUBLIC  FINDMX
                          3    ;
                          4    ;
                          5    ;
                          6    ;                                  FINDMX
                          7    ;       ASSEMBLY LANGUAGE PROCEDURE TO FIND THE ELEMENT OF AN INTEGER
                          8    ;       MATRIX WITH THE LARGEST ABSOLUTE MAGNITUDE.  THE VALUE OF THE
                          9    ;       ELEMENT IS RETURNED IN THE AX REGISTER.
                         10    ;
                         11    ;       PL/M CALLING SEQUENCE:
                         12    ;               MAXSVALUE = FINDSMX(ADRSOFSMATRIX, #SOFSROWS, #SOFSCOLS);
                         13    ;
                         14    ;       PARAMETERS:
                         15    ;               ADRSOFSMATRIX - ADDRESS OF THE MATRIX WHICH WILL BE SEARCHED
H                        16    ;               #SOFSROWS - NUMBER OF ROWS IN THE MATRIX
                         17    ;               #SOFSCOLS - NUMBER OF COLUMNS IN THE MATRIX
                         18    ;
                         19    ;       PL/M WILL PASS THE THREE PARAMETERS IN THE CALL TO THIS PROCEDURE ON
                         20    ;       THE STACK.  ON ENTRY TO THE PROCEDURE SP+6 WILL POINT TO THE FIRST
                         21    ;       PARAMETER(ADRSOFSMATRIX) AND SP+4 AND SP+2 WILL POINT TO THE SECOND
                         22    ;       AND THIRD PARAMETERS.
                         23    ;
                         24    ;       THE PROCEDURE IS A TYPED PROCEDURE WHICH ASSIGNS THE MAXIMUM VALUE
                         25    ;       IN THE MATRIX TO A VARIABLE (IN THIS CASE MAXSVALUE) IN A PL/M
                         26    ;       ASSIGNMENT STATEMENT.  TO ACCOMPLISH THIS ASSIGNMENT THE VALUE IS
                         27    ;       RETURNED IN THE AX REGISTER.
                         28    ;
                         29    ;
                         30    ;       THE ALGORITHM USED IS SIMILAR TO THE FOLLOWING PL/M CODE:
                         31    ;               FOR I = 0 TO (#SOFSROWS - 1);
                         32    ;                 FOR J = 0 TO (#SOFSCOLS - 1);
                         33    ;                   IF IABS(MATRIX(I).Y(J)) > IABS(MAX) THEN MAX = MATRIX(I).Y(J);
                         34    ;                 END;
                         35    ;               END;
                         36    ;
                         37    ;               WHERE IABS(XYZ) REPRESENTS THE ABSOLUTE VALUE OF THE INTEGER XYZ
                         38    ;
                         39    ;
```

```
LOC  OBJ                    LINE   SOURCE

                             40    ;
                             41    ;        DEFINE GROUPS TO CONFORM WITH PL/M-86 CONVENTIONS. DATA, STACK, AND
                             42    ;        CODE SEGMENTS WILL BE APPENDED TO THEIR RESPECTIVE SEGMENTS IN THE
                             43    ;        PL/M-86 MODULES.
                             44    DGROUP   GROUP    DATA,STACK
                             45    CGROUP   GROUP    CODE
                             46    ;
                             47    ;        INSTRUCT THE ASSEMBLER THAT THE DS, SS, AND CS REGISTERS WILL CONTAIN
                             48    ;        THE BASE ADDRESS VALUES FOR THE DGROUP, DGROUP AND CGROUP GROUPS.
                             49             ASSUME   DS:DGROUP,SS:DGROUP,CS:CGROUP
                             50    ;
                             51    ;
                             52    ;
                             53    ;***************DATA SEGMENT
                             54    ;
----                         55    DATA     SEGMENT WORD PUBLIC 'DATA'
0000 0000                    56    MAX      DW       0
----                         57    DATA     ENDS
                             58    ;
                             59    ;***************STACK SEGMENT
                             60    ;
----                         61    STACK    SEGMENT STACK 'STACK'
0000 (14                     62             DW       14 DUP (0)       ;RESERVE 13 WORDS OF STACK FOR MONITOR
     0000
     )
                             63                                       ;AND 1 WORD FOR FINDMX PROCEDURE
----                         64    STACK    ENDS
                             65    ;
                             66    ;***************CODE SEGMENT
                             67    ;
----                         68    CODE     SEGMENT BYTE PUBLIC 'CODE'
                             69    ;
                             70    ;PARAMETERS ON STACK, DISPLACEMENT FROM TOS INCREASED BY TWO DUE TO INITIAL PUSH
  0006[]                     71    NO_OF_ROWS       EQU      WORD PTR [BP+6]
  0004[]                     72    NO_OF_COLS       EQU      WORD PTR [BP+4]
  0008[]                     73    ADR_OF_MATRIX    EQU      WORD PTR [BP+8]
                             74    ;
0000                         75    FINDMX   PROC     NEAR             ;PROCEDURE DECLARATION
0000 55                      76             PUSH     BP               ;SAVE BP REGISTER
0001 8BEC                    77             MOV      BP,SP            ;BP POINTS TO PARAMETERS ON STACK
0003 33D2                    78             XOR      DX,DX            ;SET DX = ABS OF CURRENT MAX = 0
0005 8BFA                    79             MOV      DI,DX            ;DI = I(ROW INDEX) = 0
0007 8BF2                    80             MOV      SI,DX            ;SI = J(COLUMN INDEX) = 0
0009 89160000          R     81             MOV      MAX,DX           ;MAX = CURRENT MAX = 0
000D 8B4E04                  82             MOV      CX,NO_OF_COLS
0010 D1E1                    83             SHL      CX,1             ;CX = (#$OF$COLS) * 2
                             84                                       ;TERMINATION FOR J(SI) INDEX
0012 8B5E08                  85             MOV      BX,ADR_OF_MATRIX         ;ADRSOFSMATRIX PARAMETER
                             86                                       ;BX POINTS TO FIRST ELEMENT OF A GIVEN ROW
0015 8B00                    87    ABC:     MOV      AX,[BX][SI]      ;GET ELEMENT OF MATRIX
0017 0BC0                    88             OR       AX,AX            ;SET FLAGS
0019 7902                    89             JNS      DEF              ;JUMP IF SIGN = 0
001B F7D8                    90             NEG      AX               ;NEGATE TO FORM POSITIVE NUMBER
001D 3BC2                    91    DEF:     CMP      AX,DX            ;COMPARE TO CURRENT MAX
001F 7C07                    92             JL       XYZ              ;JUMP IF LESS THAN CURRENT MAX
0021 8BD0                    93             MOV      DX,AX            ;MOVE TO ABS OF CURRENT MAX
0023 8B00                    94             MOV      AX,[BX][SI]      ;MOVE MATRIX VALUE TO CURRENT MAX
0025 A30000            R     95             MOV      MAX,AX
0028 83C602                  96    XYZ:     ADD      SI,2             ;INCREMENT J INDEX BY TWO
002B 3BF1                    97             CMP      SI,CX            ;END OF THIS ROW ??
002D 72E6                    98             JB       ABC              ;IF NO, LOOP BACK FOR NEXT ELEMENT OF THIS ROW
002F 8D18                    99             LEA      BX,[BX+SI]       ;BX = BX + (2 * #$OF$COLS), BX POINTS TO NEXT ROW
0031 BE0000                 100             MOV      SI,0             ;J = 0
0034 47                     101             INC      DI               ;I = I + 1
0035 3B7E06                 102             CMP      DI,NO_OF_ROWS    ;LAST ROW ??
0038 72DB                   103             JB       ABC              ;IF NO, DO THE NEXT ROW
003A A10000            R    104             MOV      AX,MAX           ;RETURN MAX VALUE IN AX REGISTER
003D 5D                     105             POP      BP               ;RESTORE BP REGISTER
003E C20600                 106             RET      6                ;INCREMENT SP BY 6 AND RETURN TO CALLER
                            107    ;
                            108    FINDMX   ENDP
                            109    ;
----                        110    CODE     ENDS
                            111    ;
                            112             END
```

SYMBOL TABLE LISTING
------ ----- -------

NAME	TYPE	VALUE	ATTRIBUTES
??SEG	SEGMENT		SIZE=0000H PARA PUBLIC
ABC	L NEAR	0015H	CODE
ADR_OF_MATRIX	V WORD	0008H	[BP]
CGROUP. . . .	GROUP		CODE
CODE.	SEGMENT		SIZE=0041H BYTE PUBLIC 'CODE'
DATA.	SEGMENT		SIZE=0002H WORD PUBLIC 'DATA'
DEF	L NEAR	001DH	CODE
DGROUP. . . .	GROUP		DATA STACK
FINDMX. . . .	L NEAR	0000H	CODE PUBLIC
MAX	V WORD	0000H	DATA
NO_OF_COLS. .	V WORD	0004H	[BP]
NO_OF_ROWS. .	V WORD	0006H	[BP]
STACK	SEGMENT		SIZE=001CH PARA STACK 'STACK'
XYZ	L NEAR	0028H	CODE

ASSEMBLY COMPLETE, NO ERRORS FOUND

```
ISIS-II QRL-86, V1.1

INVOKED BY:
QRL86 :F1:MATRIX.OBJ,:F1:FIND.OBJ,SBCIOS.LIB ORIGIN(100H)

INPUT MODULES INCLUDED:
:F1:MATRIX.OBJ(EXECUTIONVEHICLE)
:F1:FIND.OBJ(FIND)
SBCIOS.LIB(SBCCO)

RESULT WRITTEN TO :F1:MATRIX(EXECUTIONVEHICLE)
START ADDRESS IS (0100H,0002H)

 START  LTH  ALIGN NAME                      CLASS

0100H  2A0H   G   /GS/ CGROUP
0100H  225H   W   CODE(EXECUTIONVEHICLE)     CODE
0125H   41H   B   CODE(FIND)                 CODE
0166H   3AH   W   CODE(SBCCO)                CODE
                  /GE/ CGROUP
02A0H   D0H   G   /GS/ DGROUP
02A0H    CH   W   CONST(EXECUTIONVEHICLE)    CONST
02ACH    8H   W   CONST(SBCCO)               CONST
02ACH   90H   W   DATA(EXECUTIONVEHICLE)     DATA
0133CH   2H   W   DATA(FIND)                 DATA
0133EH   0H   W   DATA(SBCCO)                DATA
0134ØH  30H  SW   STACK                      STACK
0137ØH   0H   W   MEMORY                     MEMORY
                  /GE/ DGROUP
0137ØH   0H   G   ??SEG(FIND)                (NULL)
```

```
DEBUG MAP OF :F1:MATRIX(EXECUTIONVEHICLE)

            MODULE: EXECUTIONVEHICLE   0100H,01E1H  LINE #: 19          0100H,0139H  LINE #: 52
02AH,00D0H  SYMBOL: MEMORY             0100H,01F8H  LINE #: 20          0100H,0142H  LINE #: 53
0100H,01B5H SYMBOL: BINDECASC          0100H,0213H  LINE #: 21          0100H,014BH  LINE #: 54
02AH,00CH   SYMBOL: TEMP               0100H,021EH  LINE #: 22          0100H,015EH  LINE #: 55
02AH,00EH   SYMBOL: I                  0100H,0221H  LINE #: 23          0100H,0169H  LINE #: 56
02AH,0010H  SYMBOL: XROW               0100H,0002H  LINE #: 36          0100H,017AH  LINE #: 57
02AH,004CH  SYMBOL: YROW               0100H,0021H  LINE #: 37          0100H,0185H  LINE #: 58
02AH,006AH  SYMBOL: ZROW               0100H,0032H  LINE #: 38          0100H,018EH  LINE #: 59
02AH,008EH  SYMBOL: I                  0100H,004BH  LINE #: 39          0100H,019FH  LINE #: 60
02AH,0090H  SYMBOL: J                  0100H,0054H  LINE #: 40          0100H,01AAH  LINE #: 61
02AH,0092H  SYMBOL: K                  0100H,005DH  LINE #: 41          0100H,01B3H  LINE #: 62
02AH,0094H  SYMBOL: MAX                0100H,006EH  LINE #: 42                       MODULE: FIN
02AH,0096H  SYMBOL: MAXASCARRAY        0100H,007FH  LINE #: 43          0100H,023AH  SYMBOL: ABC
02AH,0000H  SYMBOL: TEXT               0100H,009CH  LINE #: 44          0100H,0242H  SYMBOL: DEF
0100H,01B5H LINE #: 6                  0100H,00A5H  LINE #: 45          0100H,0225H  SYMBOL: FINDMX
0100H,01B8H LINE #: 10                 0100H,00AEH  LINE #: 46          02AH,009CH   SYMBOL: MAX
0100H,01C2H LINE #: 12                 0100H,00BFH  LINE #: 47          0100H,024DH  SYMBOL: XYZ
0100H,01C8H LINE #: 13                 0100H,00D0H  LINE #: 48          0100H,0225H  PUBLIC: FINDMX
0100H,01D1H LINE #: 14                 0100H,00E7H  LINE #: 49                       MODULE: SBCCO
0100H,01D4H LINE #: 16                 0100H,00F8H  LINE #: 50          0100H,0266H  PUBLIC: CO
0100H,01DAH LINE #: 17                 0100H,0130H  LINE #: 51
```

APPENDIX C

PROGRAM LISTING FOR EXECUTION$VEHICLE MODULE WITH CODE EXPANSION

```
PL/M-86 COMPILER    EXECUTIONVEHICLE


ISIS-II PL/M-86 V1.0 COMPILATION OF MODULE EXECUTIONVEHICLE
NO OBJECT MODULE REQUESTED
COMPILER INVOKED BY:  PLM86 :F1:MATRIX.PLM DEBUG CODE NOOBJECT PRINT(:F1:MATRIX.XLS)


                /*      MATRIX MULTIPLICATION EXAMPLE PROGRAM

                    PL/M-86 MAIN PROGRAM WHICH:
                        A) INITIALIZES TWO INTEGER MATRICES
                        B) MULTIPLIES THE TWO MATRICES AND STORES THE RESULT IN A
                           THIRD MATRIX
                        C) CALLS AN ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES THE
                           THIRD MATRIX FOR THE MAXIMUM VALUE
                        D) CALLS A PL/M PROCEDURE WHICH CONVERTS THE MAXIMUM VALUE
                           FROM INTEGER TO ASCII
                        E) CALLS A PROCEDURE WHICH OUTPUTS THE ASCII CHARACTERS ON
                           THE SYSTEM CONSOLE
                */

  1             EXECUTION$VEHICLE:
                DO;


                /*  FINDSMX - EXTERNAL ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES A
                        MATRIX FOR THE LARGEST ABSOLUTE MAGNITUDE.
                        PARAMETERS:
                            MATRIX$ADR -  ADDRESS OF THE MATRIX TO BE SEARCHED
                            ROWS - NUMBER OF ROWS IN THE MATRIX
                            COLS - NUMBER OF COLUMNS IN THE MATRIX
                */
  2   1         FINDSMX: PROCEDURE (MATRIX$PTR, ROWS, COLS) INTEGER EXTERNAL;
  3   2         DECLARE (ROWS, COLS) INTEGER;
  4   2         DECLARE MATRIX$PTR POINTER;
  5   2         END FINDSMX;


                /*  BINSDEC$ASC - BINARY TO DECIMAL ASCII CONVERSION PROCEDURE
                        PARAMETERS:
                            VALUE - INTEGER VALUE TO BE CONVERTED TO ASCII
                            CHAR$ARRAY$ADR - ADDRESS OF 6 BYTE ARRAY WHERE ASCII
                                STRING CONTAINING THE VALUE WILL BE STORED
                */
  6   1         BINSDEC$ASC: PROCEDURE (VALUE, CHAR$ARRAY$ADR);
                                                        ; STATEMENT # 6
                            BINDECASC       PROC NEAR
            01B5  55                PUSH    BP
            01B6  8BEC              MOV     BP,SP

  7   2         DECLARE (VALUE, TEMP, I) INTEGER;
  8   2         DECLARE CHAR$ARRAY$ADR POINTER;
  9   2         DECLARE (CHAR$ARRAY BASED CHAR$ARRAY$ADR) (6) BYTE;

 10   2         IF VALUE < 0 THEN
                                                        ; STATEMENT # 10
            01B8  817E060000        CMP     [BP].VALUE,0H
            01BD  7C03              JL      $+5H
            01BF  E9120             JMP     @1
 11   2         DO;
 12   3           CHAR$ARRAY(0) = '-';    /* SIGN CHARACTER */
                                                        ; STATEMENT # 12
            01C2  8B5E04            MOV     BX,[BP].CHARARRAYADR
            01C5  C6072D            MOV     CHARARRAY[BX],2DH
 13   3           TEMP = -VALUE;
                                                        ; STATEMENT # 13
            01C8  8B4606            MOV     AX,[BP].VALUE
            01CB  F7D8              NEG     AX
            01CD  8906000C          MOV     TEMP,AX
 14   3         END;
                                                        ; STATEMENT # 14
            01D1  E90D00            JMP     @2
                        @1:
                ELSE
 15   2         DO;
 16   3           CHAR$ARRAY(0) = '+';
                                                        ; STATEMENT # 16
            01D4  8B5E04            MOV     BX,[BP].CHARARRAYADR
            01D7  C6072B            MOV     CHARARRAY[BX],2BH
 17   3           TEMP = VALUE;
                                                        ; STATEMENT # 17
            01DA  8B4606            MOV     AX,[BP].VALUE
            01DD  8906000C          MOV     TEMP,AX
 18   3         END;
                        @2:
 19   2         DO I = 5 TO 1 BY -1;
                                                        ; STATEMENT # 19
            01E1  C7060200500       MOV     I,5H
            01E7  E9060             JMP     @5
                        @3:
            01EA  81060200FFFF      ADD     I,0FFFFH
```

```
                    @5:
        01F0  813E0700010C    CMP     I,1H
        01F6  7D03            JGE     $+5H
        01F8  E926FF          JMP     @4
20  3       CHARSARRAY(I) = UNSIGN(TEMP MOD 10) + 30H;
                                                  ; STATEMENT # 20
        01FB  8B060000        MOV     AX,TEMP
        01FF  B90A00          MOV     CX,0AH
        0202  33D2            XOR     DX,DX
        0204  F7F9            IDIV    CX
        0206  81C23000        ADD     DX,30H
        020A  8B5E04          MOV     BX,[BP].CHARARRAYADR
        020D  8B360200        MOV     SI,I
        0211  8810            MOV     [BX].CHARARRAY[SI],DL
21  3       TEMP = TEMP/10;
                                                  ; STATEMENT # 21
            /* ASCII CHARACTERS 30 THRU 39 HEX REPRESENT THE DIGITS 0 THRU 9. THUS
              TO CONVERT AN INTEGER TO ASCII REPEATED DIVISIONS BY 10 AND ADDING
              THE REMAINDER TO 30 HEX WILL ACCOMPLISH THE CONVERSION */
        0213  8B060000        MOV     AX,TEMP
        0217  99              CWD
        0218  F7F9            IDIV    CX
        021A  89060000        MOV     TEMP,AX
22  3     END;
                                                  ; STATEMENT # 22
        021E  E9C9FF          JMP     @3
                    @4:
23  2     END BINSDECSASC;
                                                  ; STATEMENT # 23
        0221  5D              POP     BP
        0222  C20400          RET     4H
                  BINDECASC       ENDP


          /*  CO - EXTERNAL PROCEDURE TO OUTPUT A CHARACTER TO THE SYSTEM CONSOLE.
                THIS PROCEDURE IS PART OF THE ISBC 957 LIBRARY FOR CONSOLE I/O
                PARAMETER:
                  CHAR - ASCII CHARACTER TO BE OUTPUT ON THE CONSOLE
          */
24  1     CO: PROCEDURE (CHAR) EXTERNAL;
25  2     DECLARE CHAR BYTE;
26  2     END CO;


            /* MATRIX DIMENSIONS */
27  1     DECLARE M LITERALLY '6';
28  1     DECLARE N LITERALLY '5';
29  1     DECLARE P LITERALLY '3';

          /* THE THREE MATRICES ARE DECLARED AS ARRAYS OF STRUCTURES.  XSROW IS COMPOSED
             OF M STRUCTURES EACH OF WHICH IS COMPOSED OF N INTEGER ELEMENTS.  THUS
             XSROW MAY BE THOUGHT OF AS A M X N MATRIX.  THE MATRIX WILL BE STORED AS
             A ROW-ORDER MATRIX WITH THE ELEMENTS OF EACH ROW STORED IN ADJACENT MEMORY
             LOCATIONS.  YSROW IS DECLARED AS A N X P MATRIX AND ZSROW AS A N X P MATRIX */
30  1     DECLARE XSROW(M) STRUCTURE (COL(N) INTEGER);
31  1     DECLARE YSROW(N) STRUCTURE (COL(P) INTEGER);
32  1     DECLARE ZSROW(M) STRUCTURE (COL(P) INTEGER);

33  1     DECLARE (I,J,K,MAX) INTEGER;
34  1     DECLARE MAXSASCSARRAY(6) BYTE;
35  1     DECLARE TEXT(*) BYTE DATA ('MAX VALUE = ');


          /* INITIALIZE XSROW SUCH THAT THE FIRST ROW IS SET EQUAL TO 0, THE SECOND
             ROW EQUAL TO 1, THE THIRD ROW EQUAL TO 2, ETC.  */
36  1     DO I = 0 TO (M-1);
                                                  STATEMENT # 36
        0002  FA              CLI
        0003  2E8E160000      MOV     SS,CS:@@STACK$FRAME
        0008  BC0800          MOV     SP,@@STACK$OFFSET
        000B  8BEC            MOV     BP,SP
        000D  16              PUSH    SS
        000E  1F              POP     DS
        000F  FB              STI
        0010  C706820000000   MOV     I,0H
                    @6:
        0016  813E820005000   CMP     I,5H
        001C  7E03            JLE     $+5H
        001E  E93C00          JMP     @7
37  2       DO J = 0 TO (N-1);
                                                  ; STATEMENT # 37
        0021  C706840000000   MOV     J,0H
                    @8:
        0027  813E840004000   CMP     J,4H
        002D  7E03            JLE     $+5H
        002F  E92200          JMP     @9
38  3         XSROW(I).COL(J) = I;
                                                  ; STATEMENT # 38
        0032  8B068200        MOV     AX,I
        0036  B90A00          MOV     CX,0AH
        0039  F7E9            IMUL    CX
        003B  8B368400        MOV     SI,J
        003F  D1E6            SHL     SI,1
        0041  89C3            MOV     BX,AX
        0043  8B0E8200        MOV     CX,I
        0047  898800          MOV     [BX].XROW[SI],CX
39  3       END;
```

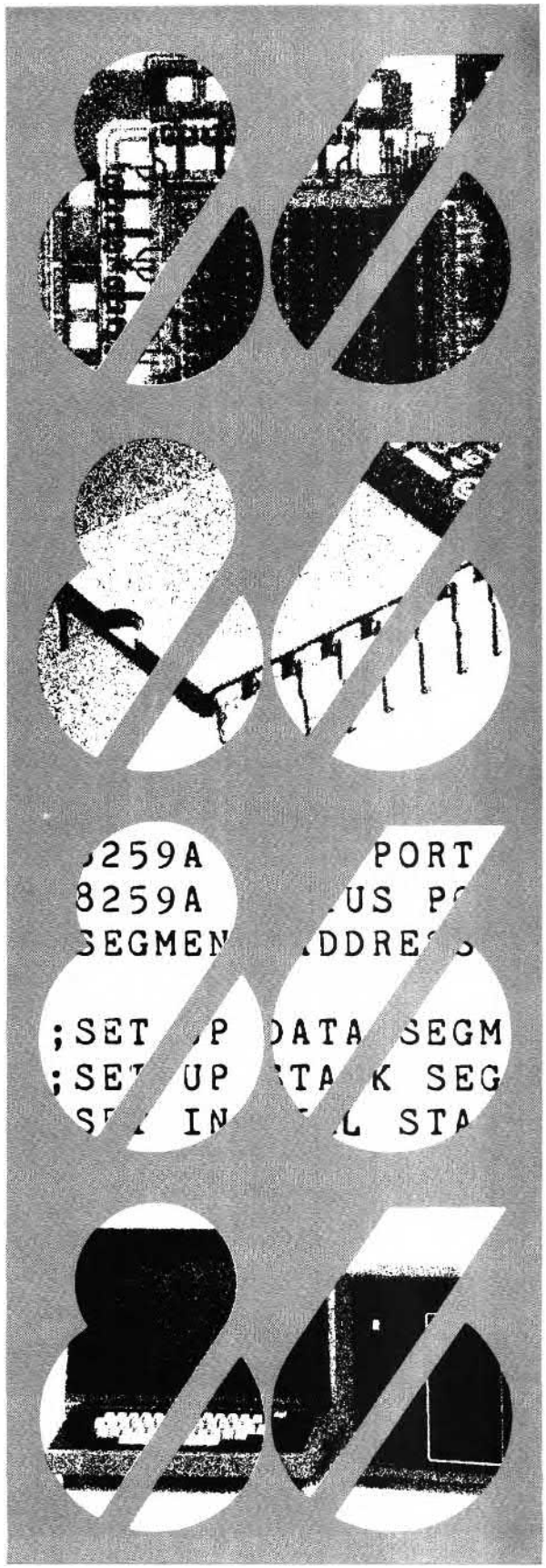
```
                                                        ; STATEMENT # 39
        0049  810684000100    ADD     J,1H
        0051  E9D3FF          JMP     @8
                    @9:
40   2     END;
                                                        ; STATEMENT # 40
        0054  810682000100    ADD     I,1H
        005A  E9B9FF          JMP     @6
                    @7:

           /* INITIALIZE YSROW SUCH THAT THE FIRST COLUMN IS SET EQUAL TO 0, THE
              SECOND COLUMN EQUAL TO -1, AND THE THIRD COLUMN EQUAL TO -2.  */
41   1     DO I = 0 TO (N-1);
                                                        ; STATEMENT # 41
        005D  C706820000000   MOV     I,0H
                    @10:
        0063  813E8200040 0   CMP     I,4H
        0069  7E03            JLE     $+5H
        006B  E94000          JMP     @11
42   2       DO J = 0 TO (P-1);
                                                        ; STATEMENT # 42
        006E  C706840000000   MOV     J,0H
                    @12:
        0074  813E840002 00   CMP     J,2H
        007A  7E03            JLE     $+5H
        007C  E92600          JMP     @13
43   3         YSROW(I).COL(J) = -J;
                                                        ; STATEMENT # 43
        007F  8B068400        MOV     AX,J
        0083  F7D8            NEG     AX
        0085  50              PUSH    AX      ; 1
        0086  8B068200        MOV     AX,I
        008A  B90600          MOV     CX,6H
        008D  F7E9            IMUL    CX
        008F  8B368400        MOV     SI,J
        0093  D1E6            SHL     SI,1
        0095  89C3            MOV     BX,AX
        0097  59              POP     CX      ; 1
        0098  89884000        MOV     [BX].YROW[SI],CX
44   3       END;
                                                        ; STATEMENT # 44
        009C  810684000100    ADD     J,1H
        00A2  E9CFFF          JMP     @12
                    @13:
45   2     END;
                                                        ; STATEMENT # 45
        00A5  810682000100    ADD     I,1H
        00AB  E9B5FF          JMP     @10
                    @11:
           /* PERFORM MATRIX MULTIPLICATION  */
46   1     DO K = 0 TO (P-1);
                                                        ; STATEMENT # 46
        00AE  C706860000000   MOV     K,0H
                    @14:
        00B4  813E860002 00   CMP     K,2H
        00BA  7E03            JLE     $+5H
        00BC  E98C00          JMP     @15
47   2       DO I = 0 TO (M-1);
                                                        ; STATEMENT # 47
        00BF  C706820000000   MOV     I,0H
                    @16:
        00C5  813E820005 80   CMP     I,5H
        00CB  7E03            JLE     $+5H
        00CD  E972 00         JMP     @17
48   3         ZSROW(I).COL(K) = 0;  /* SET ZSROW ELEMENT TO 0 */
                                                        ; STATEMENT # 48
        00D0  8B068200        MOV     AX,I
        00D4  B906 00         MOV     CX,6H
        00D7  F7E9            IMUL    CX
        00D9  8B3686 00       MOV     SI,K
        00DD  D1E6            SHL     SI,1
        00DF  89C3            MOV     BX,AX
        00E1  C78 05E000000   MOV     [BX].ZROW[SI],0H
49   3         DO J = 0 TO (N-1); /* SUM THE PRODUCT OF XSROW ROW TERMS AND YSROW COLUMN TERMS */
                                                        ; STATEMENT # 49
        00E7  C706840000000   MOV     J,0H
                    @18:
        00ED  813E840004 00   CMP     J,4H
        00F3  7E03            JLE     $+5H
        00F5  E94300          JMP     @19
50   4           ZSROW(I).COL(K) = ZSROW(I).COL(K) + ( XSROW(I).COL(J) * YSROW(J).COL(K) );
                                                        ; STATEMENT # 50
        00F8  8B068200        MOV     AX,I
        00FC  B90A00          MOV     CX,0AH
        00FF  F7E9            IMUL    CX
        0101  8B368400        MOV     SI,J
        0105  D1E6            SHL     SI,1
        0107  50              PUSH    AX      ; 1
        0108  8B068400        MOV     AX,J
        010C  B90600          MOV     CX,6H
        010F  F7E9            IMUL    CX
        0111  8B3E8600        MOV     DI,K
        0115  D1E7            SHL     DI,1
        0117  89C3            MOV     BX,AX
        0119  8B814000        MOV     AX,[BX].YROW[DI]
        011D  5B              POP     BX      ; 1
        011E  F7A80400        IMUL    [BX].XROW[SI]
        0122  50              PUSH    AX      ; 1
        0123  8B068200        MOV     AX,I
        0127  F7E9            IMUL    CX
        0129  89C3            MOV     BX,AX
```

```
          012B  58                POP     AX          ; 1
          012C  01815E00          ADD     [BX].ZROW[DI],AX
 51   4          END;
                                                      ; STATEMENT # 51
          0130  810684000100      ADD     J,1H
          0136  E9B4FF            JMP     @18
                        @19:
 52   3        END;
                                                      ; STATEMENT # 52
          0139  810682000100      ADD     I,1H
          013F  E983FF            JMP     @16
                        @17:
 53   2      END;
                                                      ; STATEMENT # 53
          0142  810686000100      ADD     K,1H
          0148  E969FF            JMP     @14
                        @15:

 54   1      MAX = FINDSMX (@ZSROW, M, P); /* FIND MAX VALUE OF ZSROW */
                                                      ; STATEMENT # 54

          014B  B85E00            MOV     AX,OFFSET(ZROW)
          014E  50                PUSH    AX          ; 1
          014F  B80600            MOV     AX,6H
          0152  50                PUSH    AX          ; 2
          0153  B80300            MOV     AX,3H
          0156  50                PUSH    AX          ; 3
          0157  E82000            CALL    FINDMX
          015A  89068800          MOV     MAX,AX
 55   1      CALL BINSDECSASC (MAX, @MAXSASCSARRAY); /* CONVERT TO DECIMAL ASCII */
                                                      ; STATEMENT # 55
          015E  FF368800          PUSH    MAX         ; 1
          0162  B88A00            MOV     AX,OFFSET(MAXASCARRAY)
          0165  50                PUSH    AX          ; 2
          0166  E84C00            CALL    BINDECASC

 56   1      DO I = 0 TO (SIGNED(SIZE(TEXT)) - 1); /* OUTPUT HEADER TEXT */
                                                      ; STATEMENT # 56
          0169  C706820000000     MOV     I,0H
                        @20:
          016F  813E82000B00      CMP     I,0BH
          0175  7E03              JLE     $+5H
          0177  E91400            JMP     @21
 57   2        CALL CO(TEXT(I));
                                                      ; STATEMENT # 57
          017A  8B1E8200          MOV     BX,I
          017E  FFB70000          PUSH    TEXT[BX]; 1
          0182  E80000            CALL    CO
 58   2      END;
                                                      ; STATEMENT # 58
          0185  810682000100      ADD     I,1H
          018B  E9E1FF            JMP     @20
                        @21:

 59   1      DO I = 0 TO 5; /* OUTPUT ASCII MAX VALUE */
                                                      ; STATEMENT # 59
          018E  C706820000000     MOV     I,0H
                        @22:
          0194  813E82000500      CMP     I,5H
          019A  7E03              JLE     $+5H
          019C  E91400            JMP     @23
 60   2        CALL CO(MAXSASCSARRAY(I));
                                                      ; STATEMENT # 60
          019F  8B1E8200          MOV     BX,I
          01A3  FFB78A00          PUSH    MAXASCARRAY[BX]; 1
          01A7  E80000            CALL    CO
 61   2      END;
                                                      ; STATEMENT # 61
          01AA  810682000100      ADD     I,1H
          01B0  E9E1FF            JMP     @22
                        @23:


 62   1      END EXECUTIONSVEHICLE;
                                                      ; STATEMENT # 62
          01B3  FB                STI
          01B4  F4                HLT


MODULE INFORMATION:

     CODE AREA SIZE     = 0225H     549D
     CONSTANT AREA SIZE = 000CH      12D
     VARIABLE AREA SIZE = 0090H     144D
     MAXIMUM STACK SIZE = 0008H       8D
     137 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Appendix B
Device Specifications

- **8086 Family**
- 8085 Peripherals*
- Standard Peripherals**
- RAM Memories***
- EPROM Memories***
- Development Tools

*For complete specifications refer to the Intel MCS-85 User's Manual.
**For complete specifications refer to the Intel Peripheral Design Handbook.
***For complete specifications refer to the 1979 Intel Component Data Catalog.



intel®

PRELIMINARY

8086/8086-2/8086-4
16-BIT HMOS MICROPROCESSOR

- **Direct Addressing Capability to 1 MByte of Memory**
- **Assembly Language Compatible with 8080/8085**
- **14 Word, By 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8-and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **5 MHz Clock Rate (8 MHz for 8086-2) (4 MHz for 8086-4)**
- **MULTIBUS™ System Compatible Interface**

The Intel® 8086 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance.

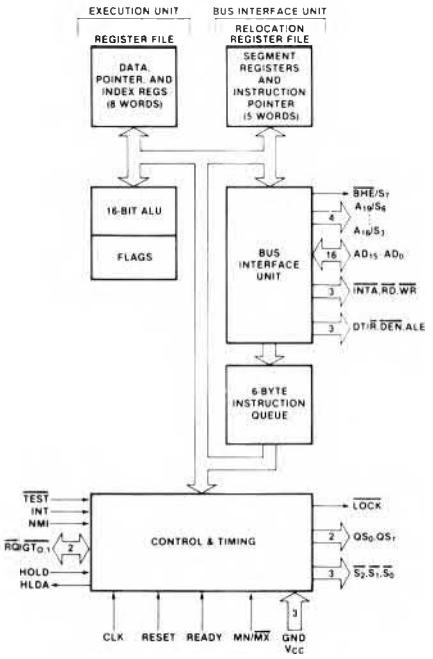
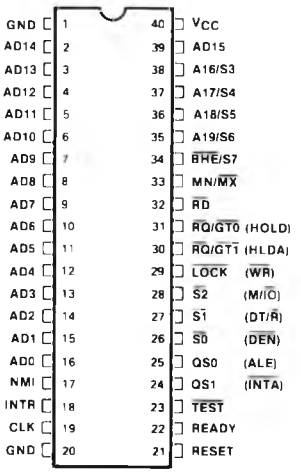


Figure 1. 8086 CPU Functional Block Diagram



40 LEAD

Figure 2. 8086 Pin Diagram

The internal functions of the 8086 processor are partitioned logically into two processing units. The first is the Bus Interface Unit (BIU) and the second is the Execution Unit (EU) as shown in the block diagram of Figure 1.

These units can interact directly but for the most part perform as separate asynchronous operational processors. The bus interface unit provides the functions related to instruction fetching and queuing, operand fetch and store, and address relocation. This unit also provides the basic bus control. The overlap of instruction pre-fetching provided by this unit serves to increase processor performance through improved bus bandwidth utilization. Up to 6 bytes of the instruction stream can be queued while waiting for decoding and execution.

The instruction stream queuing mechanism allows the BIU to keep the memory utilized very efficiently. Whenever there is space for at least 2 bytes in the queue, the BIU will attempt a word fetch memory cycle. This greatly reduces "dead time" on the memory bus. The queue acts as a First-In-First-Out (FIFO) buffer, from which the EU extracts instruction bytes as required. If the queue is empty (following a branch instruction, for example), the first byte into the queue immediately becomes available to the EU.

The execution unit receives pre-fetched instructions from the BIU queue and provides un-relocated operand addresses to the BIU. Memory operands are passed through the BIU for processing by the EU, which passes results to the BIU for storage. See the Instruction Set description for further register set and architectural descriptions.

MEMORY ORGANIZATION

The processor provides a 20-bit address to memory which locates the byte being referenced. The memory is logically organized as a linear array of 1 million bytes, addressed as 00000(H) to FFFFF(H). The memory can be further logically divided into code, data, alternate data, and stack segments of up to 64K bytes each, with each segment falling on 16-byte boundaries. (See Figure 3a.)

Word (16-bit) operands can be located on even or odd address boundaries and are thus not constrained to even boundaries as is the case in many 16-bit computers. For address and data operands, the least significant byte of the word is stored in the lower valued address location and the most significant byte in the next higher address location. The BIU automatically performs the proper number of memory accesses, one if the word operand is on an even byte boundary and two if it is on an odd byte boundary. Except for the performance penalty, this double access is transparent to the software. This performance penalty does not occur for instruction fetches, only word operands.

Physically, the memory is organized as a high bank (D_{15}-D_8) and a low bank (D_7-D_0) of 512K 8-bit bytes addressed in parallel by the processor's address lines (A_0 HIGH) is transferred on the D_{15}-D_8 bus lines. The processor provides two enable signals, $\overline{BHE}$ and A_0, to selectively allow reading from or writing into either an odd byte location, even byte location, or both. The instruction stream is fetched from memory as words and is addressed internally by the processor to the byte level as necessary.

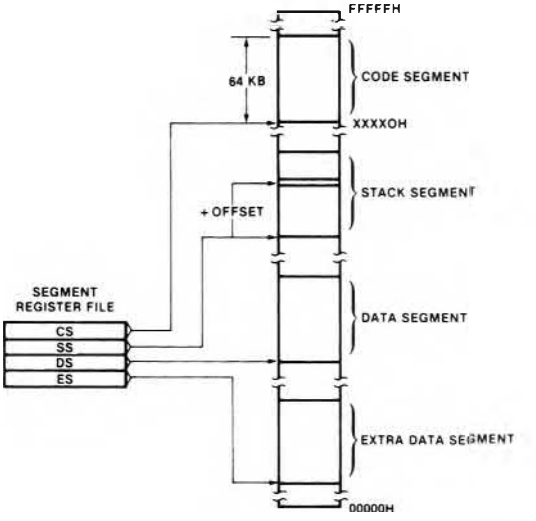


Figure 3a. Memory Organization

In referencing word data the BIU requires one or two memory cycles depending on whether or not the starting byte of the word is on an even or odd address, respectively. Consequently, in referencing word operands performance can be optimized by locating data on even address boundaries. This is an especially useful technique for using the stack, since odd address references to the stack may adversely affect the context switching time for interrupt processing or task multiplexing.

Certain locations in memory are reserved for specific CPU operations (see Figure 3b.) Locations from address FFFF0H through FFFFFH are reserved for operations including a jump to the initial program loading routine. Following RESET, the CPU will always begin execution at location FFFF0H where the jump must be. Locations 00000H through 003FFH are reserved for interrupt operations. Each of the 256 possible interrupt types has its service routine pointed to by a 4-byte pointer element consisting of a 16-bit segment address and a 16-bit offset address. The pointer elements are assumed to have been stored at the respective places in reserved memory prior to occurrence of interrupts.

PRELIMINARY

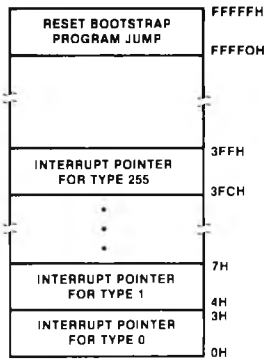


Figure 3b. Reserved Memory Locations

MINIMUM AND MAXIMUM MODES

The requirements for supporting minimum and maximum 8086 systems are sufficiently different that they cannot be done efficiently with 40 uniquely defined pins. Consequently, the 8086 is equipped with a strap pin (MN/$\overline{MX}$) which defines the system configuration. The definition of a certain subset of the pins changes dependent on the condition of the strap pin. When MN/$\overline{MX}$ pin is strapped to GND, the 8086 treats pins 24 through 31 in maximum mode. An 8288 bus controller interprets status information coded into $\overline{S}_0,\overline{S}_1,\overline{S}_2$ to generate bus timing and control signals compatible with the MULTIBUS™ architecture. When the MN/$\overline{MX}$ pin is strapped to V_{CC}, the 8086 generates bus control signals itself on pins 24 through 31, as shown in parentheses in Figure 2. Examples of minimum mode and maximum mode systems are shown in Figure 4.

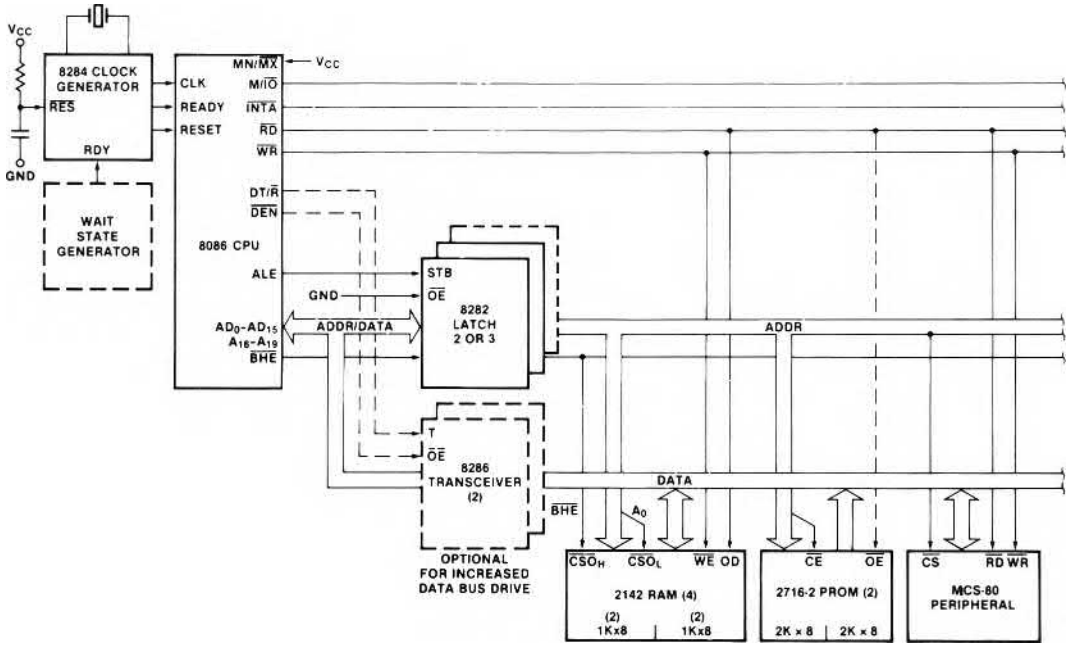


Figure 4a. Minimum Mode 8086 Typical System Configuration

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

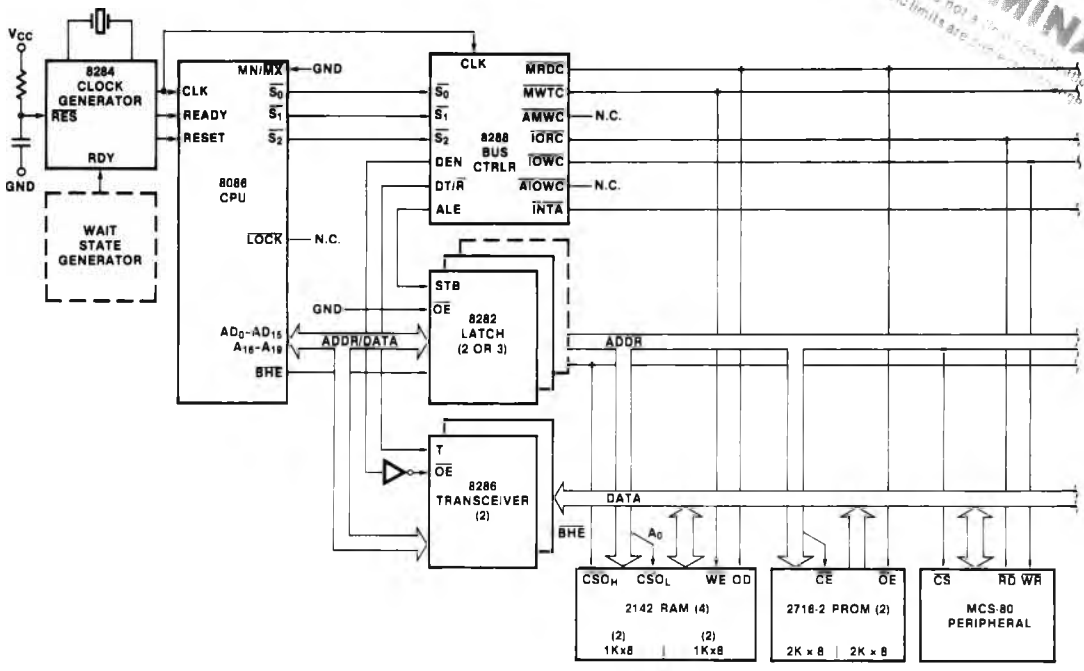


Figure 4b. Maximum Mode 8086 Typical System Configuration

BUS OPERATION

The 8086 has a combined address and data bus commonly referred to as a time multiplexed bus. This technique provides the most efficient use of pins on the processor while permitting the use of a standard 40-lead package. This "local bus" can be buffered directly and used throughout the system with address latching provided on memory and I/O modules. In addition, the bus can also be demultiplexed at the processor with a single set of address latches if a standard non-multiplexed bus is desired for the system.

Each processor bus cycle consists of at least four CLK cycles. These are referred to as T_1, T_2, T_3 and T_4 (see Figure 5). The address is emitted from the processor during T_1 and data transfer occurs on the bus during T_3 and T_4. T_2 is used primarily for changing the direction of the bus during read operations. In the event that a "NOT READY" indication is given by the addressed device, "Wait" states (T_W) are inserted between T_3 and T_4. Each inserted "Wait" state is of the same duration as a CLK cycle. Periods can occur between 8086 bus cycles. These are referred to as "Idle" states (T_I) or inactive CLK cycles. The processor uses these cycles for internal housekeeping.

During T_1 of any bus cycle the ALE (Address Latch Enable) signal is emitted (by either the processor or the 8288 bus controller, depending on the MN/$\overline{MX}$ strap). At the trailing edge of this pulse, a valid address and certain status information for the cycle may be latched.

Status bits $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ are used, in maximum mode, by the bus controller to identify the type of bus transaction according to the following table:

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	Halt
1 (HIGH)	0	0	Instruction Fetch
1	0	1	Read Data from Memory
1	1	0	Write Data to Memory
1	1	1	Passive (no bus cycle)

Status bits S_3 through S_7 are multiplexed with high-order address bits and the $\overline{BHE}$ signal, and are therefore valid during T_2 through T_4. S_3 and S_4 indicate which segment register (see Instruction Set description) was used for this bus cycle in forming the address, according to the following table:

S_4	S_3	
0 (LOW)	0	Alternate Data (extra segment)
0	1	Stack
1 (HIGH)	0	Code or None
1	1	Data

S_5 is a reflection of the PSW interrupt enable bit. $S_6 = 0$ and S_7 is a spare status bit.

PRELIMINARY
Notice: This is not a final specification. Parametric limits are subject to change.

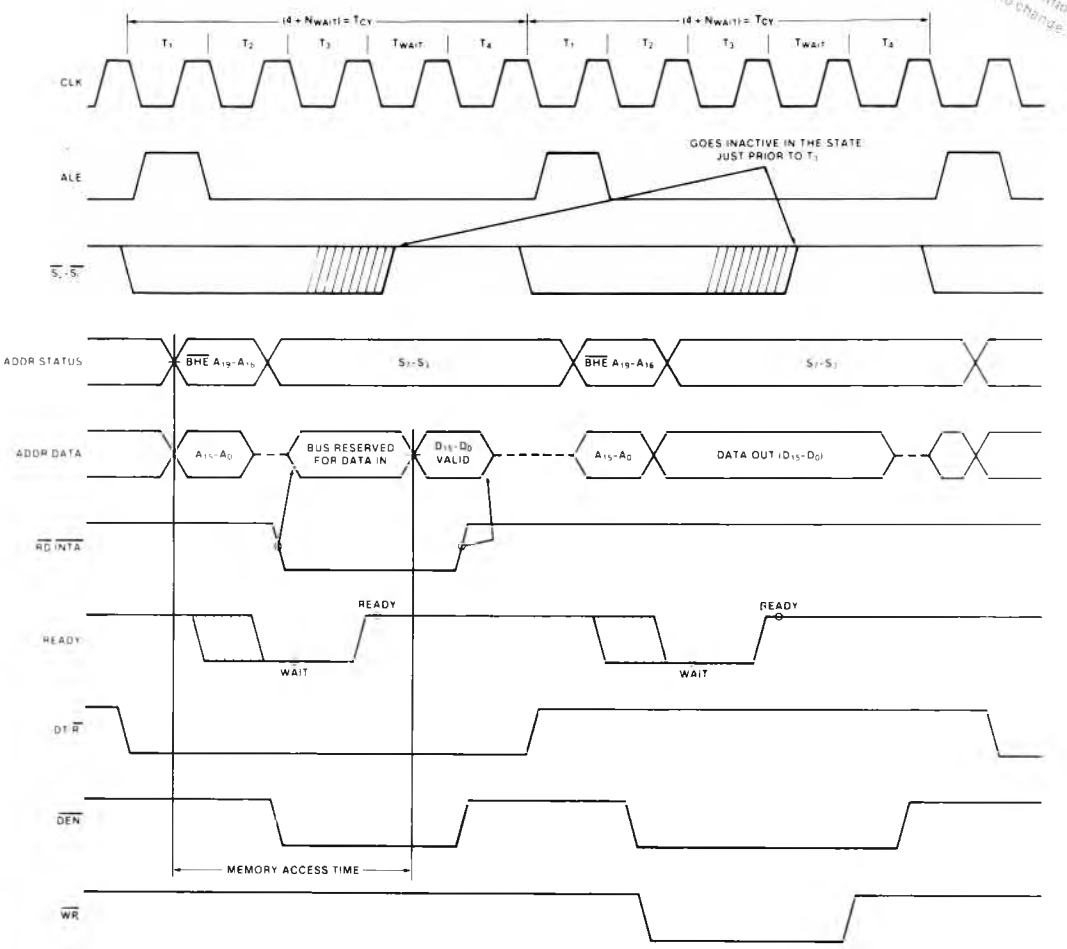


Figure 5. Basic System Timing

I/O ADDRESSING

In the 8086, I/O operations can address up to a maximum of 64K I/O byte registers or 32K I/O word registers. The I/O address appears in the same format as the memory address on bus lines A_{15}–A_0. The address lines A_{19}–A_{16} are zero in I/O operations. The variable I/O instructions which use register DX as a pointer have full address capability while the direct I/O instructions directly address one or two of the 256 I/O byte locations in page 0 of the I/O address space.

I/O ports are addressed in the same manner as memory locations. Even addressed bytes are transferred on the D_7–D_0 bus lines and odd addressed bytes on D_{15}–D_8. Care must be taken to assure that each register within an 8-bit peripheral located on the lower portion of the bus be addressed as even.

EXTERNAL INTERFACE

PROCESSOR RESET AND INITIALIZATION

Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin. The 8086 RESET is required to be HIGH for greater than 4 CLK cycles. The

PRELIMINARY

8086 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH. The low-going transition of RESET triggers an internal reset sequence for approximately 10 CLK cycles. After this interval the 8086 operates normally beginning with the instruction in absolute location FFFF0H (see Figure 3b). The details of this operation are specified in the Instruction Set description of the MCS-86 Users' Manual. The RESET input is internally synchronized to the processor clock. At initialization the HIGH-to-LOW transition of RESET must occur no sooner than 50 μs after power-up, to allow complete initialization of the 8086.

If INTR is asserted sooner than 9 CLK cycles after the end of RESET, the processor may execute one instruction before responding to the interrupt. NMI may not be asserted prior to the 2nd CLK cycle following the end of RESET.

INTERRUPT OPERATIONS

Interrupt operations fall into two classes; software or hardware initiated. The software initiated interrupts and software aspects of hardware interrupts are specified in the Instruction Set description. Hardware interrupts can be classified as non-maskable or maskable.

Interrupts result in a transfer of control to a new program location. A 256-element table containing address pointers to the interrupt service program locations resides in absolute locations 0 through 3FFH (see Figure 3b), which are reserved for this purpose. Each element in the table is 4 bytes in size and corresponds to an interrupt "type". An interrupting device supplies an 8-bit type number, during the interrupt acknowledge sequence, which is used to "vector" through the appropriate element to the new interrupt service program location.

NON-MASKABLE INTERRUPT (NMI)

The processor provides a single non-maskable interrupt pin (NMI) which has higher priority than the maskable interrupt request pin (INTR). A typical use would be to activate a power failure routine. The NMI is edge-triggered on a LOW-to-HIGH transition. The activation of this pin causes a type 2 interrupt. (See Instruction Set description.)

NMI is required to have a duration in the HIGH state of greater than two CLK cycles, but is not required to be synchronized to the clock. Any high-going transition of NMI is latched on-chip and will be serviced at the end of the current instruction or between whole moves of a block-type instruction. Worst case response to NMI would be for multiply, divide, and variable shift instructions. There is no specification on the occurrence of the low-going edge; it may occur before, during, or after the servicing of NMI. Another high-going edge triggers another response if it occurs after the start of the NMI procedure. The signal must be free of logical spikes in general and be free of bounces on the low-going edge to avoid triggering extraneous responses.

MASKABLE INTERRUPT (INTR)

The 8086 provides a single interrupt request input (INTR) which can be masked internally by software with the resetting of the interrupt enable FLAG status bit. The interrupt request signal is level triggered. It is internally synchronized during each clock cycle on the high-going edge of CLK. To be responded to, INTR must be present (HIGH) during the clock period preceding the end of the current instruction or the end of a whole move for a block-type instruction. During the interrupt response sequence further interrupts are disabled. The enable bit is reset as part of the response to any interrupt (INTR, NMI, software interrupt or single-step), although the

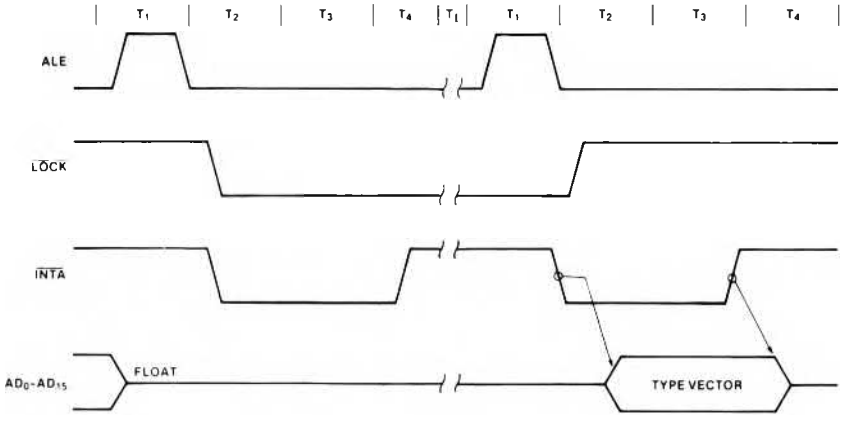


Figure 6. Interrupt Acknowledge Sequence

PRELIMINARY

FLAGS register which is automatically pushed onto the stack reflects the state of the processor prior to the interrupt. Until the old FLAGS register is restored the enable bit will be zero unless specifically set by an instruction.

During the response sequence (figure 6) the processor executes two successive (back-to-back) interrupt acknowledge cycles. The 8086 emits the LOCK signal from T_2 of the first bus cycle until T_2 of the second. A local bus "hold" request will not be honored until the end of the second bus cycle. In the second bus cycle a byte is fetched from the external interrupt system (e.g., 8259A PIC) which identifies the source (type) of the interrupt. This byte is multiplied by four and used as a pointer into the interrupt vector lookup table. An INTR signal left HIGH will be continually responded to within the limitations of the enable bit and sample period. The INTERRUPT RETURN instruction includes a FLAGS pop which returns the status of the original interrupt enable bit when it restores the FLAGS.

HALT

When a software "HALT" instruction is executed the processor indicates that it is entering the "HALT" state in one of two ways depending upon which mode is strapped. In minimum mode, the processor issues one ALE with no qualifying bus control signals. In Maximum Mode, the processor issues appropriate HALT status on $\overline{S}_2\overline{S}_1\overline{S}_0$ and the 8288 bus controller issues one ALE. The 8086 will not leave the "HALT" state when a local bus "hold" is entered while in "HALT". In this case, the processor reissues the HALT indicator. An interrupt request or RESET will force the 8086 out of the "HALT" state.

READ/MODIFY/WRITE (SEMAPHORE) OPERATIONS VIA LOCK

The $\overline{LOCK}$ status information is provided by the processor when directly consecutive bus cycles are required during the execution of an instruction. This provides the processor with the capability of performing read/modify/write operations on memory (via the Exchange Register With Memory instruction, for example) without the possibility of another system bus master receiving intervening memory cycles. This is useful in multiprocessor system configurations to accomplish "test and set lock" operations. The $\overline{LOCK}$ signal is activated (forced LOW) in the clock cycle following the one in which the software "LOCK" prefix instruction is decoded by the EU. It is deactivated at the end of the last bus cycle of the instruction following the "LOCK" prefix instruction. While $\overline{LOCK}$ is active all interrupts are masked and a request on a $\overline{RQ}/\overline{GT}$ pin will be recorded and then honored at the end of the $\overline{LOCK}$.

EXTERNAL SYNCHRONIZATION VIA TEST

As an alternative to the interrupts and general I/O capabilities, the 8086 provides a single software-testable input known as the $\overline{TEST}$ signal. At any time the program may execute a WAIT instruction. If at that time the $\overline{TEST}$ signal is inactive (HIGH), program execution becomes suspended while the processor waits for $\overline{TEST}$ to become active. It must remain active for at least 5 CLK cycles. The WAIT instruction is re-executed repeatedly until that time. This activity does not consume bus cycles. The processor remains in an idle state while waiting. All 8086 drivers go to 3-state OFF if bus "Hold"is entered. If interrupts are enabled, they may occur while the processor is waiting. When this occurs the processor fetches the WAIT instruction one extra time, processes the interrupt, and then re-fetches and re-executes the WAIT instruction upon returning from the interrupt.

8086 COMPARED WITH 8080/8085

While the 8086 has new instruction coding patterns to allow for the greatly expanded capabilities, all functions of the 8080/8085 may be performed by the 8086 with identical program semantics to their 8080/8085 versions. For every 8080/8085 instruction there is a corresponding 8086 instruction (or, in rare cases, a short sequence of instructions). Virtually all 8086 data manipulation instructions may be specified to operate on either the full set of 16-bit registers or on an 8-bit subset of them which corresponds to the 8080 register set. This relationship is shown in Figure 7 where the shaded registers (names in parentheses) represent the 8080 register set.

BASIC SYSTEM TIMING

Typical system configurations for the processor operating in minimum mode and in maximum mode are shown in Figures 4a and 4b, respectively. In minimum mode, the MN/$\overline{MX}$ pin is strapped to V_{CC} and the processor emits bus control signals in a manner similar to the 8085. In maximum mode, the MN/$\overline{MX}$ pin is strapped to V_{SS} and the processor emits coded status information which the 8288 bus controller uses to generate MULTIBUS compatible bus control signals. Figure 5 illustrates the signal timing relationships.

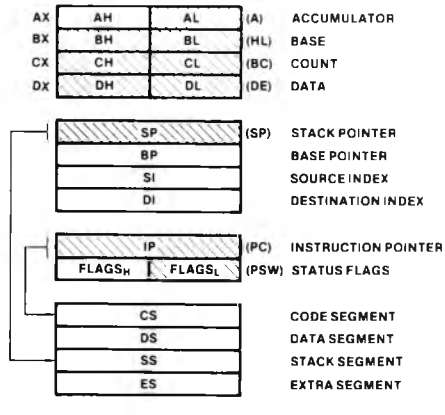


Figure 7. 8086 Register Model; (8080 Registers Shaded)

SYSTEM TIMING — MINIMUM SYSTEM

The read cycle begins in T_1 with the assertion of the Address Latch Enable (ALE) signal. The trailing (low-going) edge of this signal is used to latch the address information, which is valid on the local bus at this time, into the 8282/8283 latch. The $\overline{BHE}$ and A_0 signals address the low, high, or both bytes. From T_1 to T_4 the M/$\overline{IO}$ signal indicates a memory or I/O operation. At T_2 the address is removed from the local bus and the bus goes to a high impedance state. The read control signal is also asserted at T_2. The read ($\overline{RD}$) signal causes the addressed device to enable its data bus drivers to the local bus. Some time later valid data will be available on the bus and the addressed device will drive the READY line HIGH. When the processor returns the read signal to a HIGH level, the addressed device will again 3-state its bus drivers. If a transceiver (8286/8287) is required to buffer the 8086 local bus, signals DT/$\overline{R}$ and $\overline{DEN}$ are provided by the 8086.

A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O write operation. In the T_2 immediately following the address emission the processor emits the data to be written into the addressed location. This data remains valid until the middle of T_4. During T_2, T_3, and T_W the processor asserts the write control signal. The write ($\overline{WR}$) signal becomes active at the beginning of T_2 as opposed to the read which is delayed somewhat into T_2 to provide time for the bus to float.

The $\overline{BHE}$ and A_0 signals are used to select the proper byte(s) of the memory/IO word to be read or written according to the following table:

$\overline{BHE}$	A0	
0	0	Whole word
0	1	Upper byte from/ to odd address
1	0	Lower byte from/ to even address
1	1	None

I/O ports are addressed in the same manner as memory location. Even addressed bytes are transferred on the D_7–D_0 bus lines and odd addressed bytes on D_{15}–D_8.

The basic difference between the interrupt acknowledge cycle and a read cycle is that the interrupt acknowledge signal ($\overline{INTA}$) is asserted in place of the read ($\overline{RD}$) signal and the address bus is floated. (See Figure 6.) In the second of two successive INTA cycles, a byte of information is read from bus lines D_7–D_0 as supplied by the interrupt system logic (i.e., 8259A Priority Interrupt Controller). This byte identifies the source (type) of the interrupt. It is multiplied by four and used as a pointer into an interrupt vector lookup table, as described earlier.

BUS TIMING — MEDIUM COMPLEXITY SYSTEMS

For medium complexity systems the MN/$\overline{MX}$ pin is connected to V_{SS} and the 8288 Bus Controller is added to the system as well as an 8282/8283 latch for latching the system address, and a 8286/8287 transceiver to allow for bus loading greater than the 8086 is capable of handling. Signals ALE, DEN, and DT/$\overline{R}$ are generated by the 8288 instead of the processor in this configuration although their timing remains relatively the same. The 8086 status outputs ($\overline{S}_2$, $\overline{S}_1$, and $\overline{S}_0$) provide type-of-cycle information and become 8288 inputs. This bus cycle information specifies read (code, data, or I/O), write (data or I/O), interrupt acknowledge, or software halt. The 8288 thus issues control signals specifying memory read or write, I/O read or write, or interrupt acknowledge. The 8288 provides two types of write strobes, normal and advanced, to be applied as required. The normal write strobes have data valid at the leading edge of write. The advanced write strobes have the same timing as read strobes, and hence data isn't valid at the leading edge of write. The 8286/8287 transceiver receives the usual T and $\overline{OE}$ Inputs from the 8288's DT/$\overline{R}$ and DEN.

The pointer into the interrupt vector table, which is passed during the second INTA cycle, can derive from an 8259A located on either the local bus or the system bus. If the master 8259A Priority Interrupt Controller is positioned on the local bus, a TTL gate is required to disable the 8286/8287 transceiver when reading from the master 8259A during the interrupt acknowledge sequence and software "poll".

PRELIMINARY

8086 FUNCTIONAL PIN DEFINITION

The following pin function descriptions are for 8086 systems in either minimum or maximum mode. The "Local Bus" in these descriptions is the direct multiplexed bus interface connection to the 8086 (without regard to additional bus buffers).

AD_{15}-AD_0 (INPUT/OUTPUT 3-STATE)

These lines constitute the time multiplexed memory/IO address (T_1) and data (T_2, T_3, T_W, T_4) bus. A_0 is analogous to $\overline{BHE}$ for the lower byte of the data bus, pins D_7-D_0. It is LOW during T_1 when a byte is to be transferred on the lower portion of the bus in memory or I/O operations. Eight-bit oriented devices tied to the lower half would normally use A_0 to condition chip select functions. (See table on page 8.) These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus "hold acknowledge".

A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3 (OUTPUT 3-STATE)

During T_1 these are the four most significant address lines for memory operations. During I/O operations these lines are LOW. During memory and I/O operations, status information is available on these lines during T_2, T_3, T_W, and T_4. The status of the interrupt enable FLAG bit (S_5) is updated at the beginning of each CLK cycle. A_{17}/S_4 and A_{16}/S_3 are encoded as follows:

A_{17}/S_4	A_{16}/S_3	
0 (LOW)	0	Alternate Data
0	1	Stack
1 (HIGH)	0	Code or None
1	1	Data
S_6 is 0 (LOW		

This information indicates which relocation register is presently being used for data accessing.

These lines float to 3-state OFF during local bus "hold acknowledge".

$\overline{BHE}/S_7$ (OUTPUT 3-STATE)

During T_1 the bus high enable signal ($\overline{BHE}$) should be used to enable data onto the most significant half of the data bus, pins D_{15}-D_8. Eight-bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during T_1 for read, write, and interrupt acknowledge cycles when a byte is to be transferred on the high portion of the bus. (See table on page 8.) The S_7 status information is available during T_2, T_3, and T_4. The signal is active LOW, and floats to 3-state OFF in "hold". It is LOW during T_1 for the first interrupt acknowledge cycle.

$\overline{RD}$ (OUTPUT 3-STATE)

Read strobe indicates that the processor is performing a memory or I/O read cycle, depending on the state of the S_2 pin. This signal is used to read devices which reside on the 8086 local bus. $\overline{RD}$ is active LOW during T_2, T_3 and T_W of any read cycle, and is guaranteed to remain HIGH in T_2 until the 8086 local bus has floated.

This signal floats to 3-state OFF in "hold acknowledge".

READY (INPUT)

READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The RDY signal from memory/IO is synchronized by the 8284 Clock Generator to form READY. This signal is active HIGH.

INTR (INPUT)

Interrupt request is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.

$\overline{TEST}$ (INPUT)

The $\overline{TEST}$ input is examined by the "Wait" instruction. If the $\overline{TEST}$ input is LOW execution continues, otherwise the processor waits in an "Idle" state. This input is synchronized internally during each clock cycle on the leading edge of CLK.

NMI (INPUT)

Non-maskable interrupt is an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from a LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.

RESET (INPUT)

RESET causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the Instruction Set description, when RESET returns LOW. RESET is internally synchronized.

CLK (INPUT)

The clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.

V_{CC}

V_{CC} is the +5V ±10% (±5% on 8086-2, 8086-4) power supply pin.

GND

GND are the ground pins

PRELIMINARY

The following pin function descriptions are for the 8086/8288 system in maximum mode (i.e., MN/$\overline{MX}$ = V_{SS}). Only the pin functions which are unique to maximum mode are described; all other pin functions are as described above.

$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ (OUTPUT 3-STATE)

These status lines are encoded as follows:

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1 (HIGH)	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

Status is active during T_4, T_1, and T_2 and is returned to the passive state (1,1,1) during T_3 or during T_W when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory and I/O access control signals. Any change by $\overline{S_2}$, $\overline{S_1}$, or $\overline{S_0}$ during T_4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T_3 or T_W is used to indicate the end of a bus cycle.

These signals float to 3-state OFF in "hold acknowledge".

$\overline{RQ}/\overline{GT_0}$, $\overline{RQ}/\overline{GT_1}$ (INPUT/OUTPUT)

The request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}/\overline{GT_0}$ having higher priority than $\overline{RQ}/\overline{GT_1}$. $\overline{RQ}/\overline{GT}$ has an internal pull-up resistor so may be left unconnected. The request/grant sequence is as follows (see Figure 14):

1. A pulse of 1 CLK wide from another local bus master indicates a local bus request ("hold") to the 8086 (pulse 1).
2. During the CPU's next T_4 or T_1, a pulse 1 CLK wide from the 8086 to the requesting master (pulse 2), indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at the next CLK . The CPU's bus interface unit is disconnected logically from the local bus during "hold acknowledge".
3. A pulse 1 CLK wide from the requesting master indicates to the 8086 (pulse 3) that the "hold" request is about to end and that the 8086 can reclaim the local bus at the next CLK.

Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW.

$\overline{LOCK}$ (OUTPUT 3-STATE)

The $\overline{LOCK}$ output indicates that other system bus masters are not to gain control of the system bus while $\overline{LOCK}$ is active LOW. The $\overline{LOCK}$ signal is activated by the "LOCK" prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state OFF in "hold acknowledge".

QS_1, QS_0 (OUTPUT)

QS_1 and QS_0 provide status to allow external tracking of the internal 8086 instruction queue.

QS_1	QS_0	
0 (LOW)	0	No Operation
0	1	First Byte of Op Code from Queue
1 (HIGH)	0	Empty the Queue
1	1	Subsequent Byte from Queue

The queue status is valid during the CLK cycle after which the queue operation is performed.

PRELIMINARY

The following pin function descriptions are for the 8086 minimum mode (i.e., MN/$\overline{MX}$ = V_{CC}). Only the pin functions which are unique to minimum mode are described; all other pin functions are as described above.

M/$\overline{IO}$ (OUTPUT 3-STATE)

This status line is logically equivalent to S_2 in the maximum mode. It is used to distinguish a memory access from an I/O access. M/$\overline{IO}$ becomes valid in the T_4 preceding a bus cycle and remains valid until the final T_4 of the cycle (M = HIGH, IO = LOW). M/$\overline{IO}$ floats to 3-state OFF in local bus "hold acknowledge".

$\overline{WR}$ (OUTPUT 3-STATE)

Write strobe indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the M/$\overline{IO}$ signal. $\overline{WR}$ is active for T_2, T_3 and T_W of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge".

$\overline{INTA}$ (OUTPUT)

$\overline{INTA}$ is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T_2, T_3 and T_W of each interrupt acknowledge cycle. $\overline{INTA}$ floats to 3-state OFF in "hold acknowledge".

ALE (OUTPUT)

Address latch enable is provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during T_1 of any bus cycle. Note that ALE is never floated.

DT/$\overline{R}$ (OUTPUT 3-STATE)

Data transmit/receive is needed in minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically DT/$\overline{R}$ is equivalent to $\overline{S_1}$ in the maximum mode, and its timing is the same as for M/$\overline{IO}$.(T = HIGH, R = LOW.) This signal floats to 3-state OFF in local bus "hold acknowledge".

$\overline{DEN}$ (OUTPUT 3-STATE)

Data enable is provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{DEN}$ is active LOW during each memory and I/O access and for INTA cycles. For a read or INTA cycle it is active from the middle of T_2 until the middle of T_4, while for a write cycle it is active from the beginning of T_2 until the middle of T_4. $\overline{DEN}$ floats to 3-state OFF in local bus "hold acknowledge".

HOLD (INPUT), HLDA (OUTPUT)

HOLD indicates that another master is requesting a local bus "hold". To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement in the middle of T_4 or T_I. Simultaneous with the issuance of HLDA the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will LOWer HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines. (See Figure 15.)

HOLD is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the setup time.

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . – 65°C to + 150°C
Voltage on Any Pin with
Respect to Ground . . . . . – 1.0 to + 7V
Power Dissipation . . . . . 2.5 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS

8086: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$
8086-2/8086-4: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 5\%$

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	– 0.5	+ 0.8	V	
V_{IH}	Input High Voltage	2.0	$V_{CC} + 0.5$	V	
V_{OL}	Output Low Voltage		0.45	V	$I_{OL} = 2.0$ mA
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\ \mu A$
I_{CC}	Power Supply Current 8086/8086-4 8086-2		 340 350	 mA mA	$T_A = 25°C$
I_{LI}	Input Leakage Current		± 10	μA	$0V < V_{IN} < V_{CC}$
I_{LO}	Output Leakage Current		± 10	μA	$0.45V \leqslant V_{OUT} \leqslant V_{CC}$
V_{CL}	Clock Input Low Voltage	– 0.5	+ 0.6	V	
V_{CH}	Clock Input High Voltage	3.9	$V_{CC} + 1.0$	V	
C_{IN}	Capacitance of Input Buffer (All input except $AD_0 - AD_{15}$, $\overline{RQ}/\overline{GT}$)		10	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer ($AD_0 - AD_{15}$, $\overline{RQ}/\overline{GT}$)		20	pF	fc = 1 MHz

PRELIMINARY
Notice: This is not a final specification. Parametric limits are subject to change.

A.C. CHARACTERISTICS

8086: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$
8086-2/8086-4: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 5\%$

8086 MINIMUM COMPLEXITY SYSTEM (Figures 8, 9, 12, 15)
TIMING REQUIREMENTS

		8086/8086-4		8086-2			
Symbol	Parameter	Min.	Max.	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period — 8086 — 8086-4	200 250	500 500	125	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL) – 15		(⅔ TCLCL) – 15		ns	
TCHCL	CLK High Time	(⅓ TCLCL) + 2		(⅓ TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		20		ns	
TCLDX	Data In Hold Time	10		10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		0		ns	
TRYHCH	READY Setup Time into 8086	(⅔ TCLCL) – 15		(⅔ TCLCL) – 15		ns	
TCHRYX	READY Hold Time into 8086	30		20		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	– 8		– 8		ns	
THVCH	HOLD Setup Time	35		20		ns	
TINVCH	INTR, NMI, $\overline{TEST}$ Setup Time (See Note 2)	30		15		ns	

TIMING RESPONSES

		8086/8086-4		8086-2			
Symbol	Parameter	Min.	Max.	Min.	Max.	Units	Test Conditions
TCLAV	Address Valid Delay	10	110	10	60	ns	
TCLAX	Address Hold Time	10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	TCLAX	50	ns	
TLHLL	ALE Width	TCLCH-20		TCLCH-10		ns	
TCLLH	ALE Active Delay		80		50	ns	
TCHLL	ALE Inactive Delay		85		55	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL-10		TCHCL-10		ns	
TCLDV	Data Valid Delay	10	110	10	60	ns	C_L = 20-100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCHDX	Data Hold Time	10		10		ns	
TWHDX	Data Hold Time After WR	TCLCH-30		TCLCH-30		ns	
TCVCTV	Control Active Delay 1	10	110	10	70	ns	
TCHCTV	Control Active Delay 2	10	110	10	60	ns	
TCVCTX	Control Inactive Delay	10	110	10	70	ns	
TAZRL	Address Float to READ Active	0		0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	10	100	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	10	80	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL-45		TCLCL -40		ns	
TCLHAV	HLDA Valid Delay	10	160	10	100	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL-75		2TCLCL-50		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL-60		2TCLCL-40		ns	
TAVAL	Address Valid to ALE Low	TCLCH-60		TCLCH-40		ns	

NOTES: 1. Signal at 8284 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state. (8 ns into T3)

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

8086 MAX MODE SYSTEM (USING 8288 BUS CONTROLLER) (Figures 10-14)
TIMING REQUIREMENTS

Symbol	Parameter	8086/8086-4 Min.	8086/8086-4 Max.	8086-2 Min.	8086-2 Max.	Units	Test Conditions
TCLCL	CLK Cycle Period — 8086 — 8086-4	200 250	500 500	125	500	ns	
TCLCH	CLK Low Time	(2/3 TCLCL) – 15		(2/3 TCLCL) – 15		ns	
TCHCL	CLK High Time	(1/3 TCLCL) + 2		(1/3 TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		20		ns	
TCLDX	Data In Hold Time	10		10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		0		ns	
TRYHCH	READY Setup Time into 8086	(2/3 TCLCL) – 15		(2/3 TCLCL) – 15		ns	
TCHRYX	READY Hold Time into 8086	30		20		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	– 8		– 8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, $\overline{TEST}$) (See Note 2)	30		15		ns	
TGVCH	$\overline{RQ}/\overline{GT}$ Setup Time	30		15		ns	
TCHGX	$\overline{RQ}$ Hold Time into 8086	40		30		ns	

TIMING RESPONSES

Symbol	Parameter	8086/8086-4 Min.	8086/8086-4 Max.	8086-2 Min.	8086-2 Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	10	35	ns	
TCLMH	Command Inactive Delay (See Note 1)	10	35	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110		65	ns	
TCHSV	Status Active Delay	10	110	10	60	ns	
TCLSH	Status Inactive Delay	10	130	10	70	ns	
TCLAV	Address Valid Delay	10	110	10	60	ns	
TCLAX	Address Hold Time	10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	TCLAX	50	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15		15	ns	C_L = 20-100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCLMCL	MCE Inactive Delay (See Note 1)		15		15	ns	
TCLDV	Data Valid Delay	10	110	10	60	ns	
TCHDX	Data Hold Time	10		10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	10	45	ns	
TAZRL	Address Float to Read Active	0		0		ns	
TCLRL	RD Active Delay	10	165	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	80	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL-45		TCLCL-40		ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30		30	ns	
TCLGL	$\overline{GT}$ Active Delay	0	85	0	50	ns	
TCLGH	$\overline{GT}$ Inactive Delay	0	85	0	50	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL-75		2TCLCL-50		ns	

NOTES: 1. Signal at 8284 or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T3 and wait states.
4. Applies only to T2 state (8 ns into T3).

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

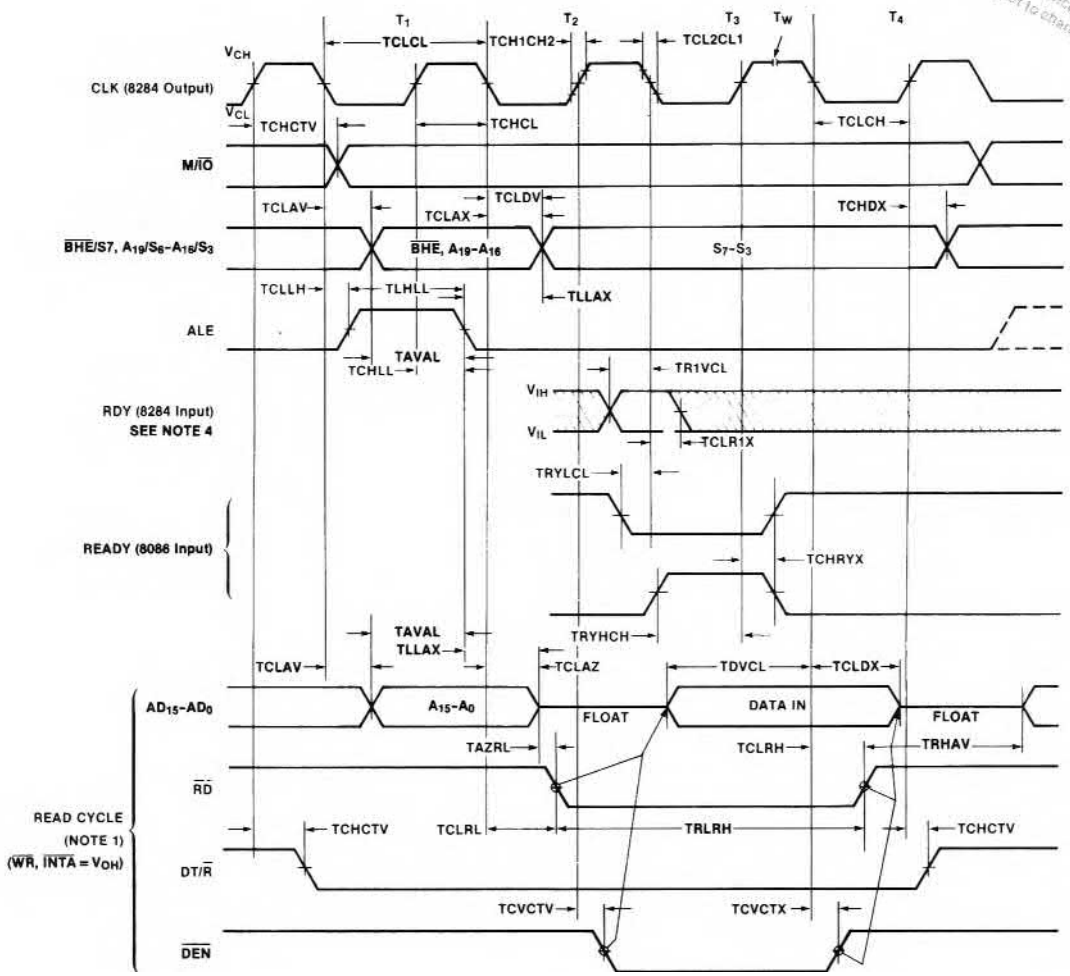
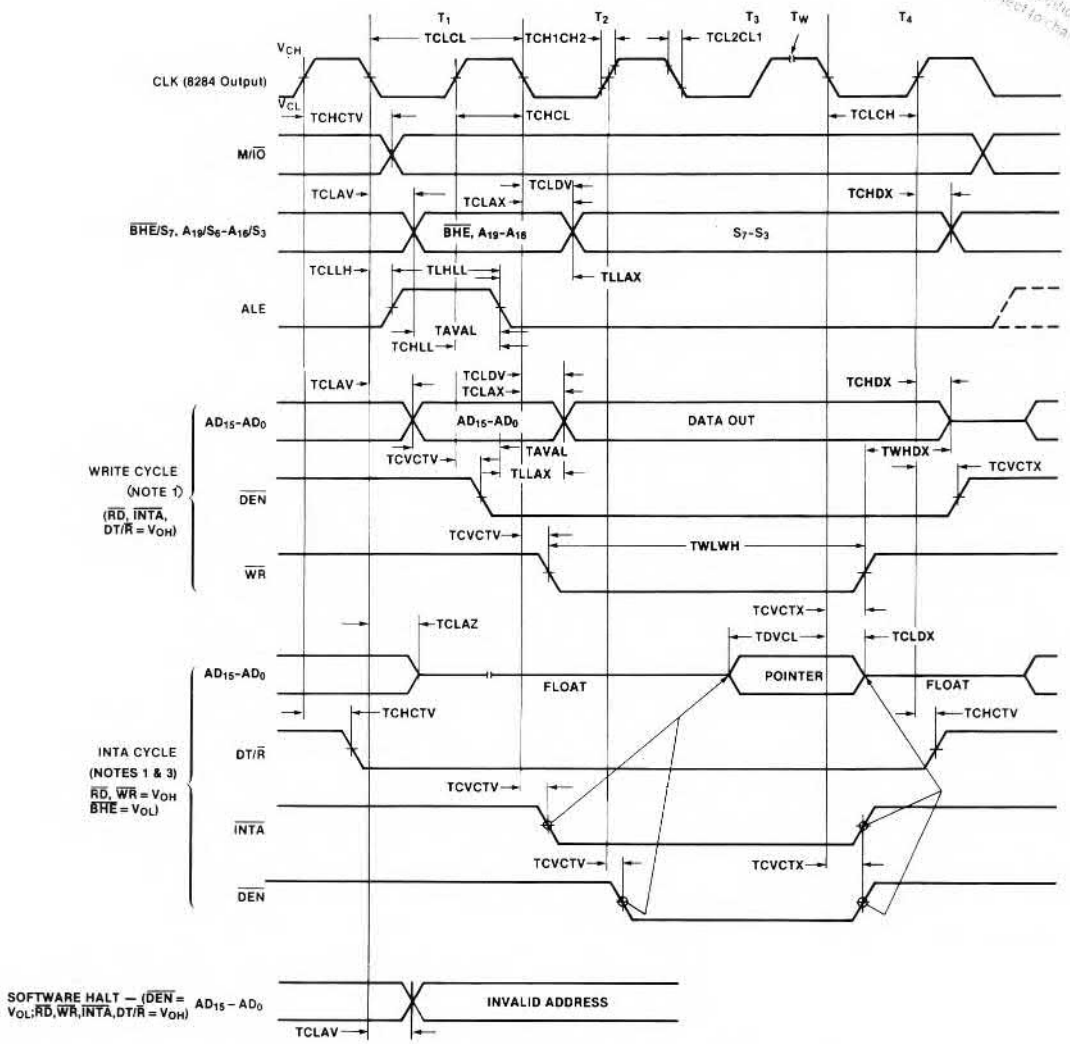


Figure 8. 8086 Bus Timing — Minimum Mode System

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8086 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL SIGNALS SHOWN FOR SECOND INTA CYCLE.
4. SIGNALS AT 8284 ARE SHOWN FOR REFERENCE ONLY.
5. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

Figure 9. 8086 Bus Timing — Minimum Mode System (cont'd)

PRELIMINARY

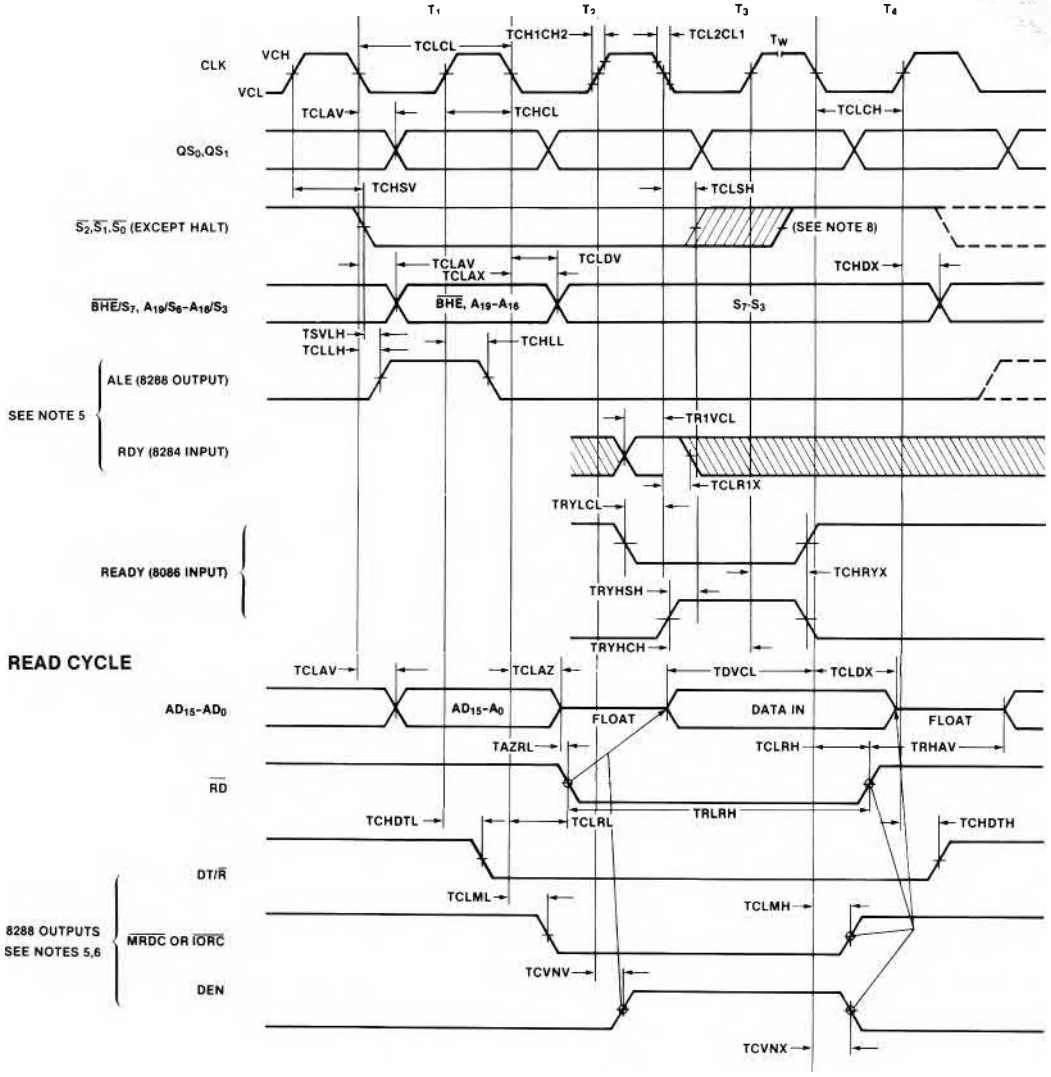
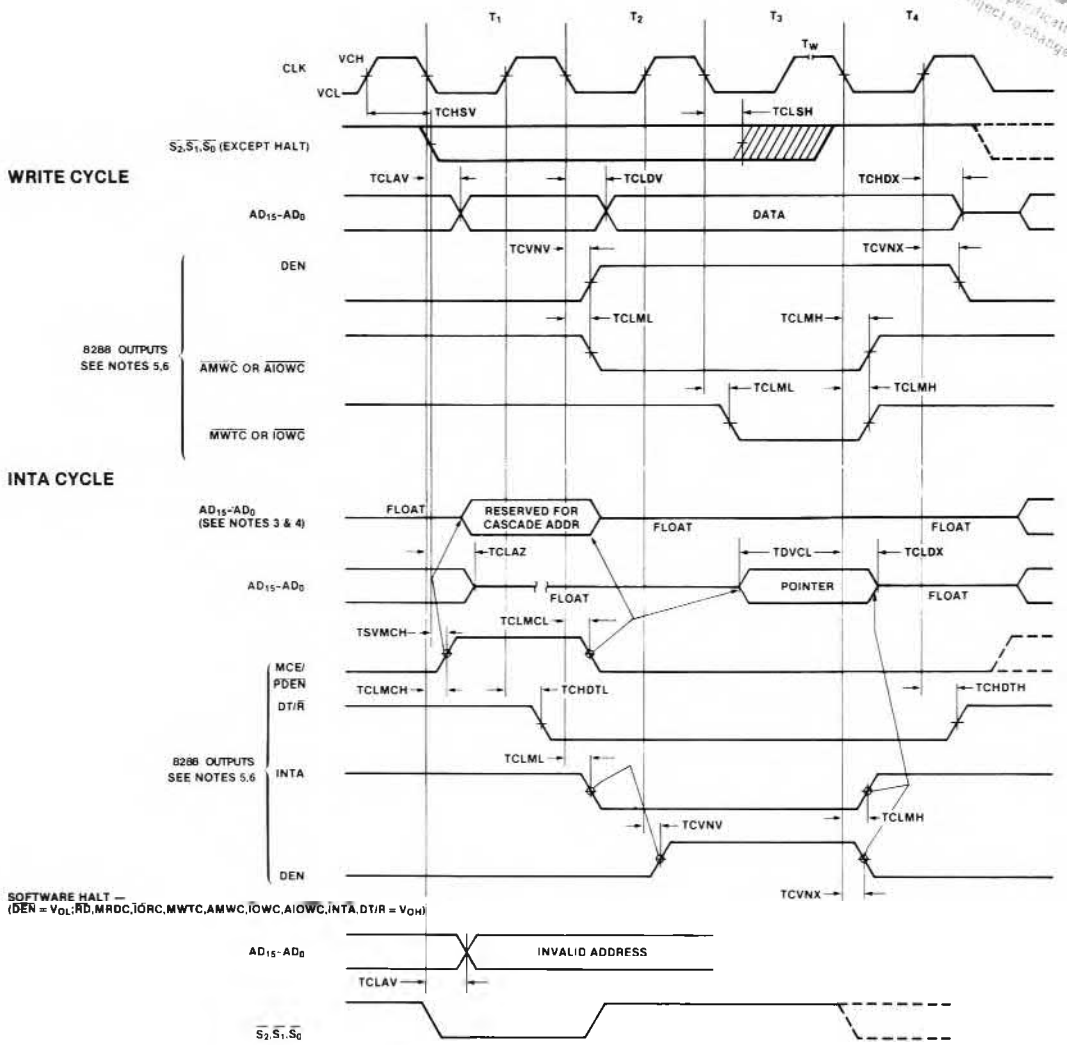


Figure 10. 8086 Bus Timing — Maximum Mode System (Using 8288)

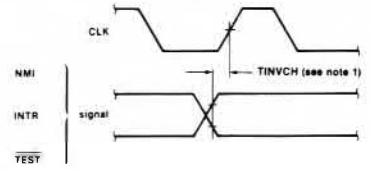
PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. CASCADE ADDRESS IS VALID BETWEEN FIRST AND SECOND INTA CYCLE.
4. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8086 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL FOR POINTER ADDRESS IS SHOWN FOR SECOND INTA CYCLE.
5. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
6. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS (MRDC, MWTC, AMWC, IORC, IOWC, AIOWC, INTA AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
7. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
8. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.

Figure 11. 8086 Bus Timing — Maximum Mode System (Using 8288) (cont.)

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.



NOTE:

1. SETUP REQUIREMENTS FOR ASYNCHRONOUS SIGNALS ONLY TO GUARANTEE RECOGNITION AT NEXT CLK

Figure 12. Asynchronous Signal Recognition

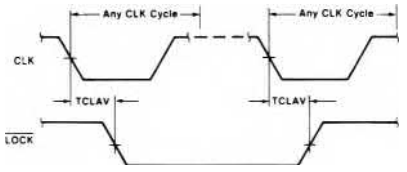
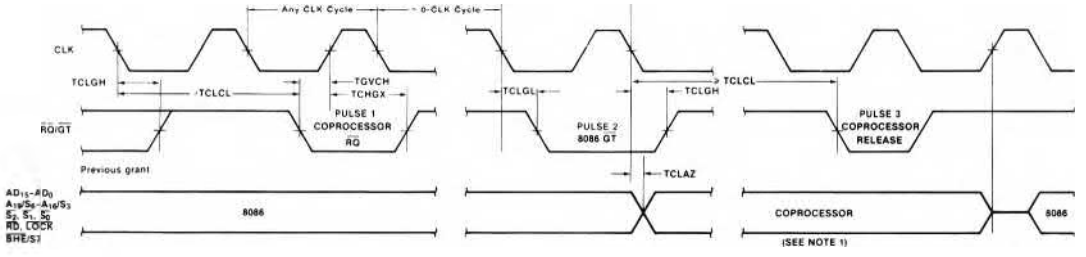


Figure 13. Bus Lock Signal Timing (Maximum Mode Only)



NOTES: 1. THE COPROCESSOR MAY NOT DRIVE THE BUSES OUTSIDE THE REGION SHOWN WITHOUT RISKING CONTENTION.

Figure 14. Request/Grant Sequence Timing (Maximum Mode Only)

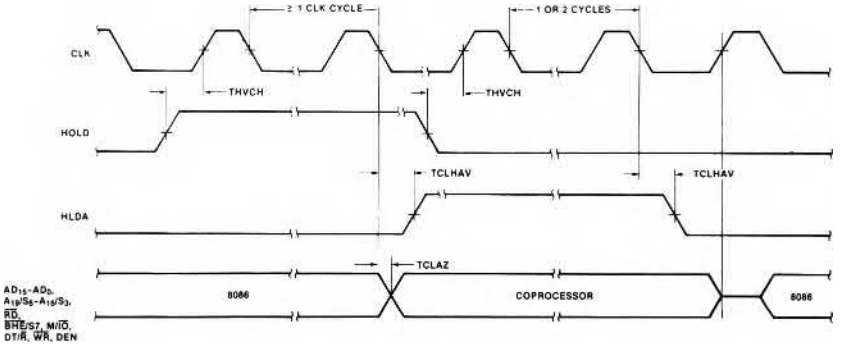


Figure 15. Hold/Hold Acknowledge Timing (Minimum Mode Only)

PRELIMINARY
Notice: This is not a final specification. Some parameters are subject to change.

8086
INSTRUCTION SET SUMMARY

DATA TRANSFER

MOV = Move:	76543210	76543210	76543210	76543210
Register/memory to/from register	100010dw	mod reg r/m		
Immediate to register/memory	1100011w	mod000r/m	data	data if w=1
Immediate to register	1011w reg	data	data if w=1	
Memory to accumulator	1010000w	addr-low	addr-high	
Accumulator to memory	1010001w	addr-low	addr-high	
Register/memory to segment register	10001110	mod0reg r/m		
Segment register to register/memory	10001100	mod0reg r/m		

PUSH = Push:				
Register/memory	11111111	mod110r/m		
Register	01010 reg			
Segment register	000reg110			

POP = Pop:				
Register/memory	10001111	mod000r/m		
Register	01011 reg			
Segment register	000reg111			

XCHG = Exchange:				
Register/memory with register	1000011w	mod reg r/m		
Register with accumulator	10010 reg			

IN=Input from:				
Fixed port	1110010w	port		
Variable port	1110110w			

OUT = Output to:				
Fixed port	1110011w	port		
Variable port	1110111w			
XLAT=Translate byte to AL	11010111			
LEA=Load EA to register	10001101	mod reg r/m		
LDS=Load pointer to DS	11000101	mod reg r/m		
LES=Load pointer to ES	11000100	mod reg r/m		
LAHF=Load AH with flags	10011111			
SAHF=Store AH into flags	10011110			
PUSHF=Push flags	10011100			
POPF=Pop flags	10011101			

ARITHMETIC

ADD = Add:				
Reg./memory with register to either	000000dw	mod reg r/m		
Immediate to register/memory	100000sw	mod000r/m	data	data if s w=01
Immediate to accumulator	0000010w	data	data if w=1	

ADC = Add with carry:				
Reg./memory with register to either	000100dw	mod reg r/m		
Immediate to register/memory	100000sw	mod010r/m	data	data if s w=01
Immediate to accumulator	0001010w	data	data if w=1	

INC = Increment:				
Register/memory	1111111w	mod000r/m		
Register	01000 reg			
AAA=ASCII adjust for add	00110111			
DAA=Decimal adjust for add	00100111			

SUB = Subtract:				
Reg./memory and register to either	001010dw	mod reg r/m		
Immediate from register/memory	100000sw	mod101r/m	data	data if s w=01
Immediate from accumulator	0010110w	data	data if w=1	

SBB = Subtract with borrow				
Reg./memory and register to either	000110dw	mod reg r/m		
Immediate from register/memory	100000sw	mod011r/m	data	data if s w=01
Immediate from accumulator	0001110w	data	data if w=1	

DEC = Decrement	76543210	76543210	76543210	76543210
Register/memory	1111111w	mod001r/m		
Register	01001 reg			
NEG=Change sign	1111011w	mod011r/m		

CMP = Compare:				
Register/memory and register	001110dw	mod reg r/m		
Immediate with register/memory	100000sw	mod111r/m	data	data if s w=01
Immediate with accumulator	0011110w	data	data if w=1	
AAS=ASCII adjust for subtract	00111111			
DAS=Decimal adjust for subtract	00101111			
MUL=Multiply (unsigned)	1111011w	mod100r/m		
IMUL=Integer multiply (signed)	1111011w	mod101r/m		
AAM=ASCII adjust for multiply	11010100	00001010		
DIV=Divide (unsigned)	1111011w	mod110r/m		
IDIV=Integer divide (signed)	1111011w	mod111r/m		
AAD=ASCII adjust for divide	11010101	00001010		
CBW=Convert byte to word	10011000			
CWD=Convert word to double word	10011001			

LOGIC

NOT=Invert	1111011w	mod010r/m		
SHL/SAL=Shift logical/arithmetic left	110100vw	mod100r/m		
SHR=Shift logical right	110100vw	mod101r/m		
SAR=Shift arithmetic right	110100vw	mod111r/m		
ROL=Rotate left	110100vw	mod000r/m		
ROR=Rotate right	110100vw	mod001r/m		
RCL=Rotate through carry flag left	110100vw	mod010r/m		
RCR=Rotate through carry right	110100vw	mod011r/m		

AND = And:				
Reg./memory and register to either	001000dw	mod reg r/m		
Immediate to register/memory	1000000w	mod100r/m	data	data if w=1
Immediate to accumulator	0010010w	data	data if w=1	

TEST = And function to flags, no result:				
Register/memory and register	1000010w	mod reg r/m		
Immediate data and register/memory	1111011w	mod000r/m	data	data if w=1
Immediate data and accumulator	1010100w	data	data if w=1	

OR = Or				
Reg./memory and register to either	000010dw	mod reg r/m		
Immediate to register/memory	1000000w	mod001r/m	data	data if w=1
Immediate to accumulator	0000110w	data	data if w=1	

XOR = Exclusive or:				
Reg./memory and register to either	001100dw	mod reg r/m		
Immediate to register/memory	1000000w	mod110r/m	data	data if w=1
Immediate to accumulator	0011010w	data	data if w=1	

STRING MANIPULATION

REP=Repeat	1111001z
MOVS=Move byte/word	1010010w
CMPS=Compare byte/word	1010011w
SCAS=Scan byte/word	1010111w
LODS=Load byte/wd to AL/AX	1010110w
STDS=Stor byte/wd from AL/A	1010101w

Mnemonics ©Intel, 1978

PRELIMINARY
Notice: This is not a final specification
Parametric limits are subject to change

CONTROL TRANSFER

CALL = Call:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	

JMP = Unconditional Jump:			
Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within segment-short	1 1 1 0 1 0 1 1	disp	
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

RET = Return from CALL:			
Within segment	1 1 0 0 0 0 1 1		
Within seg. adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment, adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high

JE/JZ=Jump on equal/zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE=Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG=Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE=Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA=Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp	
JP/JPE=Jump on parity/parity even	0 1 1 1 1 0 1 0	disp	
JO=Jump on overflow	0 1 1 1 0 0 0 0	disp	
JS=Jump on sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ=Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE=Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG=Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp	

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNB/JAE: Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp
JNBE/JA: Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp
JNP/JPO: Jump on not par/par odd	0 1 1 1 1 0 1 1	disp
JNO: Jump on not overflow	0 1 1 1 0 0 0 1	disp
JNS: Jump on not sign	0 1 1 1 1 0 0 1	disp
LOOP: Loop CX times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE: Loop while zero/equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE: Loop while not zero/equal	1 1 1 0 0 0 0 0	disp
JCXZ: Jump on CX zero	1 1 1 0 0 0 1 1	disp

INT = Interrupt		
Type specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO: Interrupt on overflow	1 1 0 0 1 1 1 0	
IRET: Interrupt return	1 1 0 0 1 1 1 1	

PROCESSOR CONTROL

CLC: Clear carry	1 1 1 1 1 0 0 0	
CMC: Complement carry	1 1 1 1 0 1 0 1	
STC: Set carry	1 1 1 1 1 0 0 1	
CLD: Clear direction	1 1 1 1 1 1 0 0	
STD: Set direction	1 1 1 1 1 1 0 1	
CLI: Clear interrupt	1 1 1 1 1 0 1 0	
STI: Set interrupt	1 1 1 1 1 0 1 1	
HLT: Halt	1 1 1 1 0 1 0 0	
WAIT: Wait	1 0 0 1 1 0 1 1	
ESC: Escape (to external device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK: Bus lock prefix	1 1 1 1 0 0 0 0	

Footnotes:

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value.
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

Mnemonics © Intel, 1978

if s w = 01 then 16 bits of immediate data form the operand
if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1, if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

intel®

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change

M8086
16-BIT HMOS MICROPROCESSOR

- **Direct Addressing Capability to 1 MByte of Memory**
- **Assembly Language Compatible with 8080/8085**
- **14 Word, By 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8-and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **5 MHz Clock Rate**
- **MULTIBUS™ System Compatible Interface**
- **Full Military Temperature Range −55°C to +125°C**

The Intel® M8086 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance.

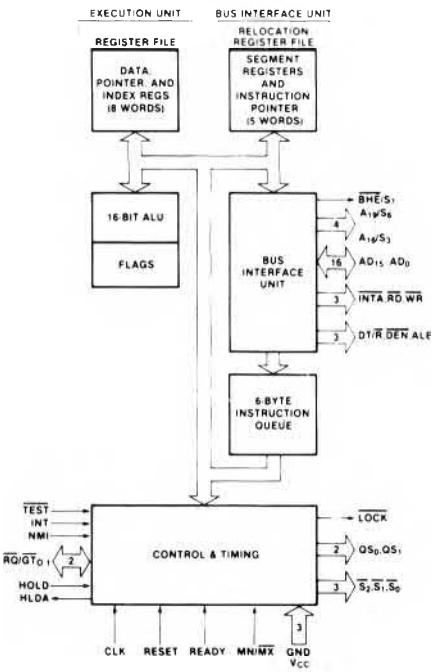
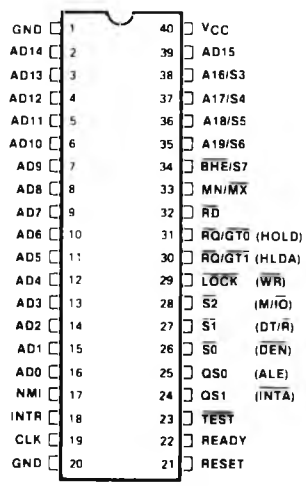


Figure 1. M8086 CPU Functional Block Diagram



40 LEAD

Figure 2. M8086 Pin Diagram

I8086
16-BIT HMOS MICROPROCESSOR

- **Direct Addressing Capability to 1 MByte of Memory**
- **Assembly Language Compatible with 8080/8085**
- **14 Word, By 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8-and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **5 MHz Clock Rate**
- **MULTIBUS™ System Compatible Interface**
- **Industrial Temperature Range −40°C to +85°C**

The Intel® I8086 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance.

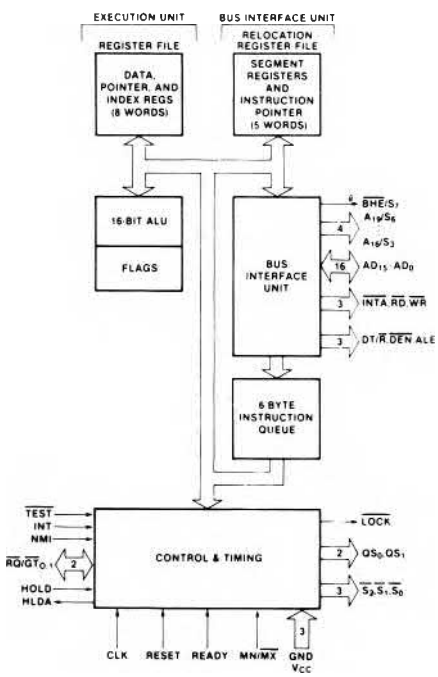


Figure 1. I8086 CPU Functional Block Diagram

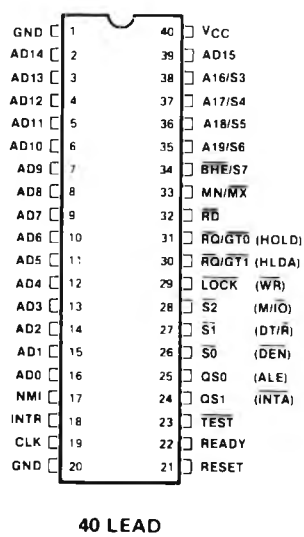


Figure 2. I8086 Pin Diagram

intel®

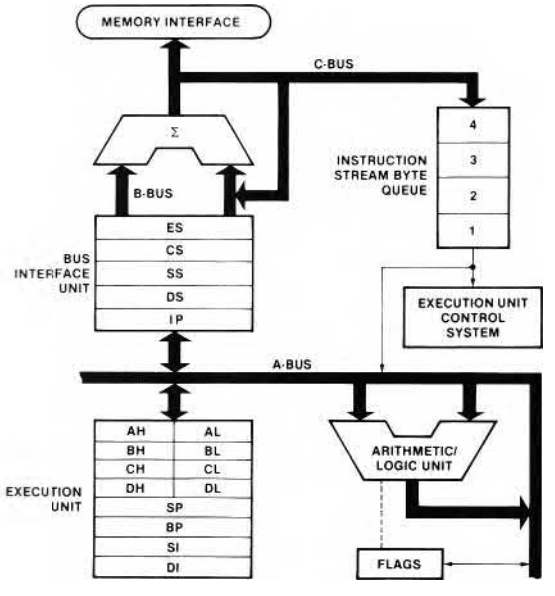
PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

8088
8-BIT HMOS MICROPROCESSOR

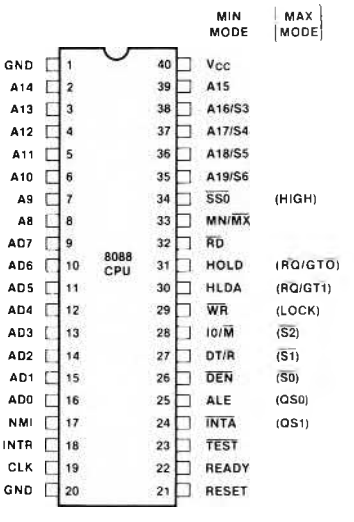
- 8-Bit Data Bus Interface
- 16-Bit Internal Architecture
- Direct Addressing Capability to 1 Mbyte of Memory
- Direct Software Compatibility with 8086
- 14-Word by 16-Bit Register Set with Symmetrical Operations
- 24 Operand Addressing Modes
- Byte, Word, and Block Operations
- 8-Bit and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal, including Multiply and Divide
- Compatible with 8155-2, 8755A-2 and 8185-2 Multiplexed Peripherals

The Intel® 8088 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8 and 16-bit microprocessors. It is directly compatible with 8086 software and 8080/8085 hardware and peripherals.

8088 CPU FUNCTIONAL BLOCK DIAGRAM



8088 PIN DIAGRAM



PRELIMINARY

FUNCTIONAL DESCRIPTION

Memory Organization

The processor provides a 20-bit address to memory which locates the byte being referenced. The memory is logically organized as a linear array of 1 million bytes, addressed as 00000(H) to FFFFF(H). The memory can be further logically divided into code, data, alternate data, and stack segments of up to 64K bytes each, with each segment falling on 16-byte boundaries. (See Figure 1.)

Word (16-bit) operands can be located on even or odd address boundaries. For address and data operands, the least significant byte of the word is stored in the lower valued address location and the most significant byte in the next higher address location. The BIU will automatically execute two fetch or write cycles for 16-bit operands.

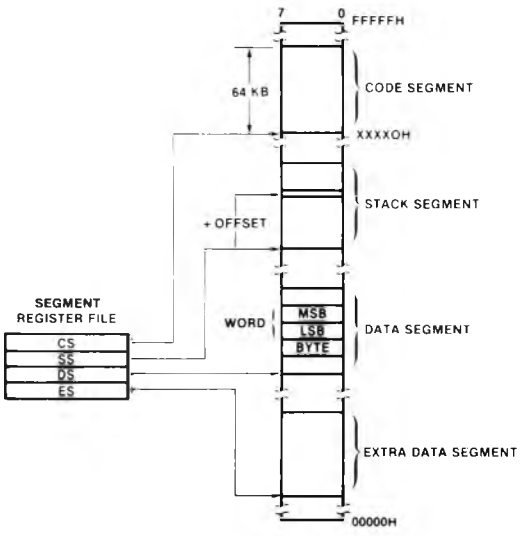


Figure 1. Memory Organization

Certain locations in memory are reserved for specific CPU operations. (See Figure 2.) Locations from addresses FFFF0H through FFFFFH are reserved for operations including a jump to the initial system initialization routine. Following RESET, the CPU will always begin execution at location FFFF0H where the jump must be located. Locations 00000H through 003FFH are reserved for interrupt operations. Four-byte pointers consisting of a 16-bit segment address and a 16-bit offset address direct program flow to one of the 256 possible interrupt service routines. The pointer elements are assumed to have been stored at their respective places in reserved memory prior to the occurrence of interrupts.

Minimum and Maximum Modes

The requirements for supporting minimum and maximum 8088 systems are sufficiently different that they cannot be done efficiently with 40 uniquely defined pins. Consequently, the 8088 is equipped with a strap pin (MN/$\overline{MX}$) which defines the system configuration. The definition of a certain subset of the pins changes, dependent on the condition of the strap pin. When the MN/$\overline{MX}$ pin is strapped to GND, the 8088 defines pins 24 through 31 and 34 in maximum mode. When the MN/$\overline{MX}$ pin is strapped to V_{CC}, the 8088 generates bus control signals itself on pins 24 through 31 and 34.

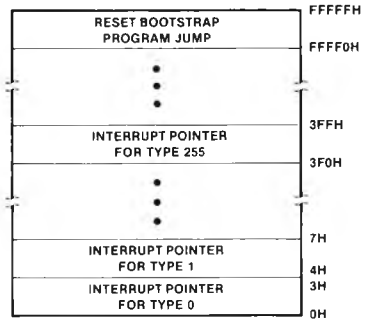


Figure 2. Reserved Memory Locations

The minimum mode 8088 can be used with either a multiplexed or demultiplexed bus. The multiplexed bus configuration is compatible with the MCS-85™ multiplexed bus peripherals (8155, 8156, 8355, 8755A, and 8185). This configuration (See Figure 3) provides the user with a minimum chip count system. This architecture provides the 8088 processing power in a highly integrated form.

The demultiplexed mode requires one latch (for 64K addressability) or two latches (for a full megabyte of addressing). A third latch can be used for buffering if the address bus loading requires it. An 8286 or 8287 transceiver can also be used if data bus buffering is required. (See Figure 4.) The 8088 provides $\overline{DEN}$ and DT/$\overline{R}$ to control the transceiver, and ALE to latch the addresses. This configuration of the minimum mode provides the standard demultiplexed bus structure with heavy bus buffering and relaxed bus timing requirements.

The maximum mode employs the 8288 bus controller. (See Figure 5.) The 8288 decodes status lines $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$, and provides the system with all bus control signals. Moving the bus control to the 8288 provides better source and sink current capability to the control lines, and frees the 8088 pins for extended large system features. Hardware lock, queue status, and two request/grant interfaces are provided by the 8088 in maximum mode. These features allow co-processors in local bus and remote bus configurations.

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

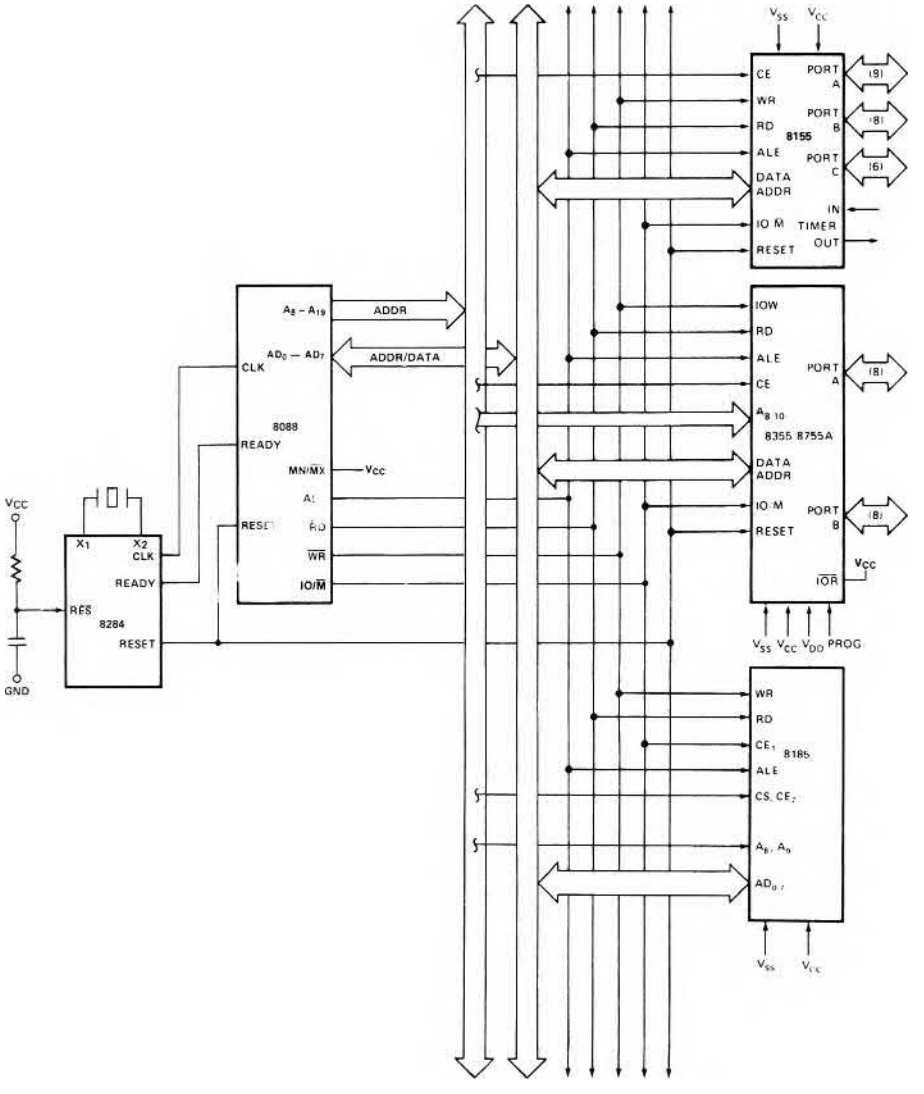


Figure 3. Multiplexed Bus Configuration

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

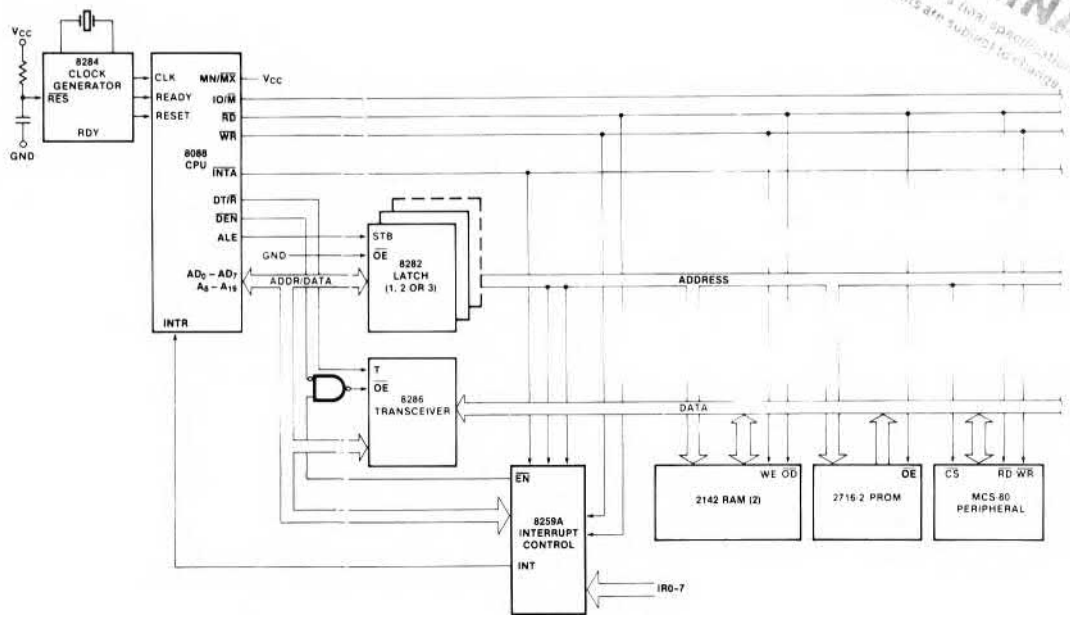


Figure 4. Demultiplexed Bus Configuration

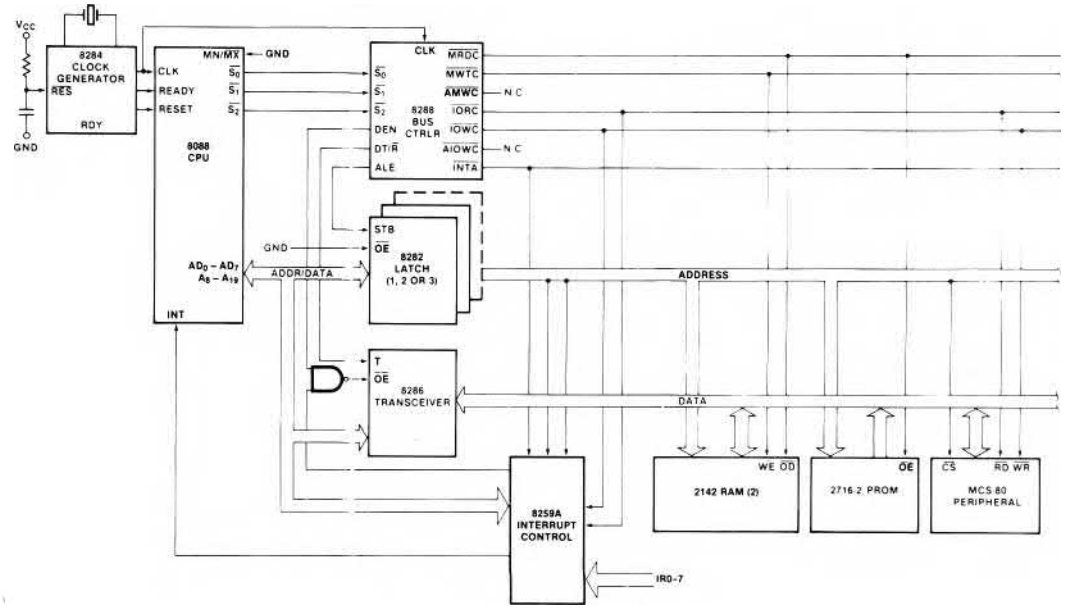


Figure 5. Fully Buffered System Using Bus Controller

PRELIMINARY

Bus Operation

The 8088 address/data bus is broken into three parts — the lower eight address/data bits (AD0-AD7), the middle eight address bits (A8-A15), and the upper four address bits (A16-A19). The address/data bits and the highest four address bits are time multiplexed. This technique provides the most efficient use of pins on the processor, permitting the use of a standard 40 lead package. The middle eight address bits are not multiplexed, i.e. they remain valid throughout each bus cycle. In addition, the bus can be demultiplexed at the processor with a single address latch if a standard non-multiplexed bus is desired for the system.

Each processor bus cycle consists of at least four CLK cycles. These are referred to as T1, T2, T3, and T4. (See Figure 6). The address is emitted from the processor during T1 and data transfer occurs on the bus during T3 and T4. T2 is used primarily for changing the direction of the bus during read operations. In the event that a "NOT READY" indication is given by the addressed device,

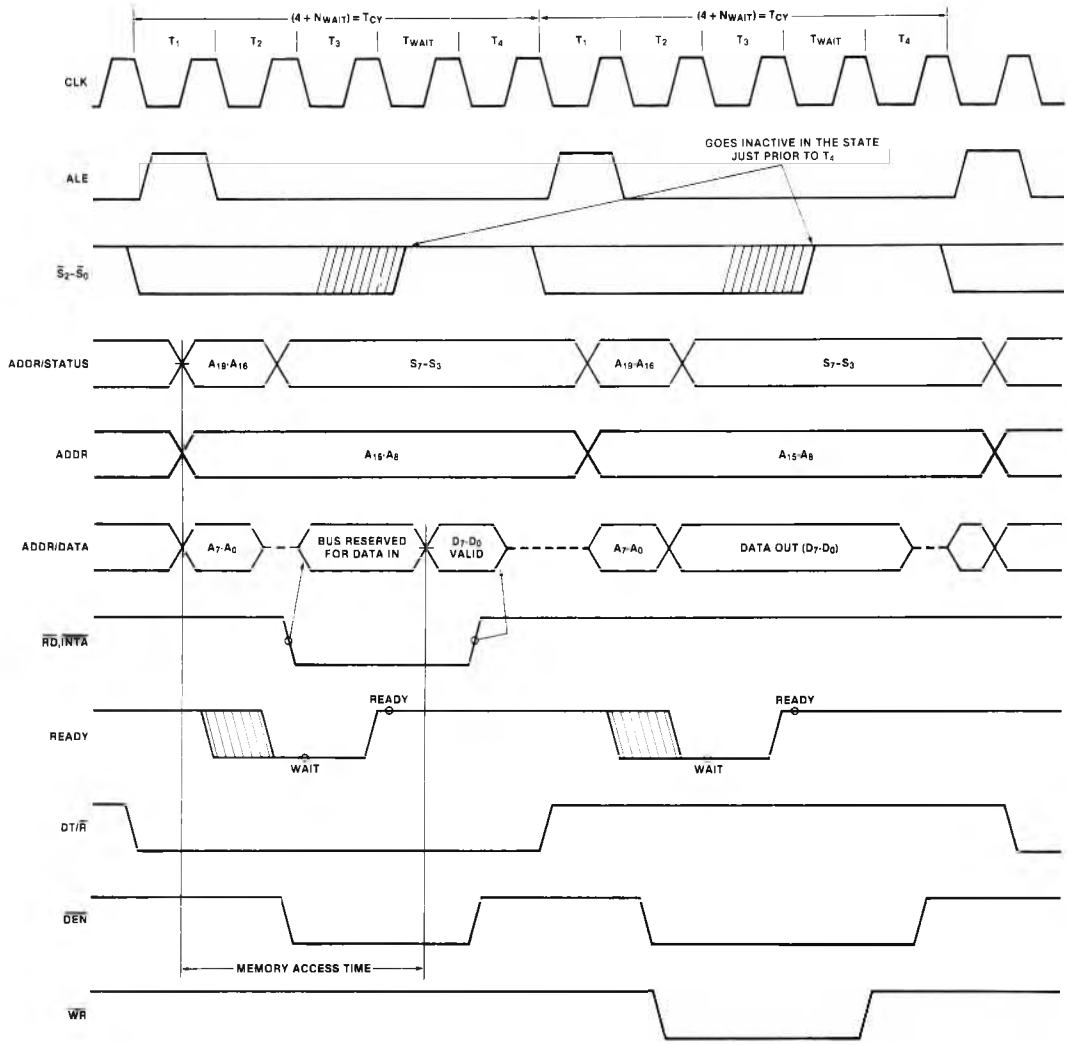


Figure 6. Basic System Timing

PRELIMINARY

"wait" states (Tw) are inserted between T3 and T4. Each inserted "wait" state is of the same duration as a CLK cycle. Periods can occur between 8088 driven bus cycles. These are referred to as "idle" states (Ti), or inactive CLK cycles. The processor uses these cycles for internal housekeeping.

During T1 of any bus cycle, the ALE (address latch enable) signal is emitted (by either the processor or the 8288 bus controller, depending on the MN/$\overline{MX}$ strap). At the trailing edge of this pulse, a valid address and certain status information for the cycle may be latched.

Status bits $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ are used by the bus controller, in maximum mode, to identify the type of bus transaction according to the following table:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0 (Low)	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	Halt
1 (High)	0	0	Instruction fetch
1	0	1	Read data from memory
1	1	0	Write data to memory
1	1	1	Passive (no bus cycle)

Status bits S3 through S6 are multiplexed with high order address bits and are therefore valid during T2 through T4. S3 and S4 indicate which segment register was used for this bus cycle in forming the address according to the following table:

S4	S3	
0 (Low)	0	Alternate data (Extra Segment)
0	1	Stack
1 (High)	0	Code or none
1	1	Data

S5 is a reflection of the PSW interrupt enable bit. S6 is always equal to 0.

I/O Addressing

In the 8088, I/O operations can address up to a maximum of 64K I/O registers. The I/O address appears in the same format as the memory address on bus lines A15–A0. The address lines A19–A16 are zero in I/O operations. The variable I/O instructions, which use register DX as a pointer, have full address capability, while the direct I/O instructions directly address one or two of the 256 I/O byte locations in page 0 of the I/O address space. I/O ports are addressed in the same manner as memory locations.

Designers familiar with the 8085 or upgrading an 8085 design should note that the 8085 addresses I/O with an 8-bit address on both halves of the 16-bit address bus. The 8088 uses a full 16-bit address on its lower 16 address lines.

EXTERNAL INTERFACE

Processor Reset and Initialization

Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin. The 8088 RESET is required to be HIGH for greater than four clock cycles. The 8088 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH. The low-going transition of RESET triggers an internal reset sequence for approximately 7 clock cycles. After this interval the 8088 operates normally, beginning with the instruction in absolute location FFFF0H. (See Figure 2.) The RESET input is internally synchronized to the processor clock. At initialization, the HIGH to LOW transition of RESET must occur no sooner than 50 μs after power up, to allow complete initialization of the 8088.

If INTR is asserted sooner than nine clock cycles after the end of RESET, the processor may execute one instruction before responding to the interrupt.

All 3-state outputs float to 3-state OFF during RESET. Status is active in the idle state for the first clock after RESET becomes active and then floats to 3-state OFF.

Interrupt Operations

Interrupt operations fall into two classes; software or hardware initiated. The software initiated interrupts and software aspects of hardware interrupts are specified in the instruction set description found in Chapter 2 of the 8086 Family User's Manual. Hardware interrupts can be classified as non-maskable or maskable.

Interrupts result in a transfer of control to a new program location. A 256 element table containing address pointers to the interrupt service program locations resides in absolute locations 0 through 3FFH (see Figure 2), which are reserved for this purpose. Each element in the table is 4 bytes in size and corresponds to an interrupt "type". An interrupting device supplies an 8-bit type number, during the interrupt acknowledge sequence, which is used to vector through the appropriate element to the new interrupt service program location.

Non-Maskable Interrupt (NMI)

The processor provides a single non-maskable interrupt (NMI) pin which has higher priority than the maskable interrupt request (INTR) pin. A typical use would be to activate a power failure routine. The NMI is edge-triggered on a LOW to HIGH transition. The activation of this pin causes a type 2 interrupt.

NMI is required to have a duration in the HIGH state of greater than two clock cycles, but is not required to be synchronized to the clock. Any higher going transition of NMI is latched on-chip and will be serviced at the end of the current instruction or between whole moves (2 bytes in the case of word moves) of a block type instruction. Worst case response to NMI would be for multiply, divide, and variable shift instructions. There is no specification on the occurrence of the low-going edge; it may occur before, during, or after the servicing of NMI. Another high-going edge triggers another response if it

occurs after the start of the NMI procedure. The signal must be free of logical spikes in general and be free of bounces on the low-going edge to avoid triggering extraneous responses.

Maskable Interrupt (INTR)

The 8088 provides a single interrupt request input (INTR) which can be masked internally by software with the resetting of the interrupt enable (IF) flag bit. The interrupt request signal is level triggered. It is internally synchronized during each clock cycle on the high-going edge of CLK. To be responded to, INTR must be present (HIGH) during the clock period preceding the end of the current instruction or the end of a whole move for a block type instruction. During interrupt response sequence, further interrupts are disabled. The enable bit is reset as part of the response to any interrupt (INTR, NMI, software interrupt, or single step), although the FLAGS register which is automatically pushed onto the stack reflects the state of the processor prior to the interrupt. Until the old FLAGS register is restored, the enable bit will be zero unless specifically set by an instruction.

During the response sequence (See Figure 7), the processor executes two successive (back to back) interrupt acknowledge cycles. The 8088 emits the LOCK signal (maximum mode only) from T2 of the first bus cycle until T2 of the second. A local bus "hold" request will not be honored until the end of the second bus cycle. In the second bus cycle, a byte is fetched from the external interrupt system (e.g., 8259A PIC) which identifies the source (type) of the interrupt. This byte is multiplied by four and used as a pointer into the interrupt vector lookup table. An INTR signal left HIGH will be continually responded to within the limitations of the enable bit and sample period. The interrupt return instruction includes a flags pop which returns the status of the original interrupt enable bit when it restores the flags.

HALT

When a software HALT instruction is executed, the processor indicates that it is entering the HALT state in one of two ways, depending upon which mode is strapped. In minimum mode, the processor issues ALE, delayed by one clock cycle, to allow the system to latch the halt status. Halt status is available on IO/$\overline{M}$, DT/$\overline{R}$, and $\overline{SSO}$. In maximum mode, the processor issues appropriate HALT status on $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$, and the 8288 bus controller issues one ALE. The 8088 will not leave the HALT state when a local bus hold is entered while in HALT. In this case, the processor reissues the HALT indicator at the end of the local bus hold. An interrupt request or RESET will force the 8088 out of the HALT state.

Read/Modify/Write (Semaphore) Operations via LOCK

The LOCK status information is provided by the processor when consecutive bus cycles are required during the execution of an instruction. This allows the processor to perform read/modify/write operations on memory (via the "exchange register with memory" instruction), without another system bus master receiving intervening memory cycles. This is useful in multiprocessor system configurations to accomplish "test and set lock" operations. The $\overline{LOCK}$ signal is activated (LOW) in the clock cycle following decoding of the LOCK prefix instruction. It is deactivated at the end of the last bus cycle of the instruction following the LOCK prefix. While $\overline{LOCK}$ is active, all interrupts are masked and a request on a $\overline{RQ}$/$\overline{GT}$ pin will be recorded, and then honored at the end of the LOCK.

External Synchronization via $\overline{TEST}$

As an alternative to interrupts, the 8088 provides a single software-testable input pin ($\overline{TEST}$). This input is utilized by executing a WAIT instruction. The single

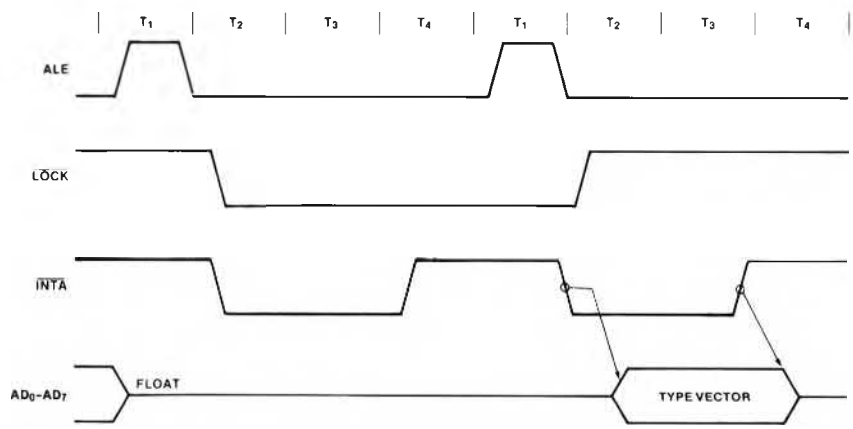


Figure 7. Interrupt Acknowledge Sequence

WAIT instruction is repeatedly executed until the $\overline{TEST}$ input goes active (LOW). The execution of WAIT does not consume bus cycles once the queue is full.

If a local bus request occurs during WAIT execution, the 8088 3-states all output drivers. If interrupts are enabled, the 8088 will recognize interrupts and process them. The WAIT instruction is then refetched, and reexecuted.

Basic System Timing

In minimum mode, the MN/$\overline{MX}$ pin is strapped to V_{CC} and the processor emits bus control signals compatible with the 8085 bus structure. In maximum mode, the MN/$\overline{MX}$ pin is strapped to GND and the processor emits coded status information which the 8288 bus controller uses to generate MULTIBUS compatible bus control signals.

System Timing — Minimum System

(See Figure 6.)

The read cycle begins in T1 with the assertion of the address latch enable (ALE) signal. The trailing (low going) edge of this signal is used to latch the address information, which is valid on the address/data bus (AD0-AD7) at this time, into the 8282/8283 latch. Address lines A8 through A15 do not need to be latched because they remain valid throughout the bus cycle. From T1 to T4 the IO/$\overline{M}$ signal indicates a memory or I/O operation. At T2 the address is removed from the address/data bus and the bus goes to a high impedance state. The read control signal is also asserted at T2. The read ($\overline{RD}$) signal causes the addressed device to enable its data bus drivers to the local bus. Some time later, valid data will be available on the bus and the addressed device will drive the READY line HIGH. When the processor returns the read signal to a HIGH level, the addressed device will again 3-state its bus drivers. If a transceiver (8286/8287) is required to buffer the 8088 local bus, signals DT/$\overline{R}$ and $\overline{DEN}$ are provided by the 8088.

A write cycle also begins with the assertion of ALE and the emission of the address. The IO/$\overline{M}$ signal is again asserted to indicate a memory or I/O write operation. In T2, immediately following the address emission, the processor emits the data to be written into the addressed location. This data remains valid until at least the middle of T4. During T2, T3, and T_W, the processor asserts the write control signal. The write ($\overline{WR}$) signal becomes active at the beginning of T2, as opposed to the read, which is delayed somewhat into T2 to provide time for the bus to float.

The basic difference between the interrupt acknowledge cycle and a read cycle is that the interrupt acknowledge ($\overline{INTA}$) signal is asserted in place of the read ($\overline{RD}$) signal and the address bus is floated. (See Figure 7.) In the second of two successive $\overline{INTA}$ cycles, a byte of information is read from the data bus, as supplied by the interrupt system logic (i.e. 8259A priority interrupt controller). This byte identifies the source (type) of the interrupt. It is multiplied by four and used as a pointer into the interrupt vector lookup table, as described earlier.

Bus Timing — Medium Complexity Systems

(See Figure 8.)

For medium complexity systems, the MN/$\overline{MX}$ pin is connected to GND and the 8288 bus controller is added to the system, as well as an 8282/8283 latch for latching the system address, and an 8286/8287 transceiver to allow for bus loading greater than the 8088 is capable of handling. Signals ALE, $\overline{DEN}$, and DT/$\overline{R}$ are generated by the 8288 instead of the processor in this configuration, although their timing remains relatively the same. The 8088 status outputs ($\overline{S2}$, $\overline{S1}$, and $\overline{S0}$) provide type of cycle information and become 8288 inputs. This bus cycle information specifies read (code, data, or I/O), write (data or I/O), interrupt acknowledge, or software halt. The 8288 thus issues control signals specifying memory read or write, I/O read or write, or interrupt acknowledge. The 8288 provides two types of write strobes, normal and advanced, to be applied as required. The normal write strobes have data valid at the leading edge of write. The advanced write strobes have the same timing as read strobes, and hence, data is not valid at the leading edge of write. The 8286/8287 transceiver receives the usual T and $\overline{OE}$ inputs from the 8288's DT/$\overline{R}$ and $\overline{DEN}$ outputs.

The pointer into the interrupt vector table, which is passed during the second $\overline{INTA}$ cycle, can derive from an 8259A located on either the local bus or the system bus. If the master 8289A priority interrupt controller is positioned on the local bus, a TTL gate is required to disable the 8286/8287 transceiver when reading from the master 8259A during the interrupt acknowledge sequence and software "poll".

The 8088 Compared to the 8086

The 8088 CPU is an 8-bit processor designed around the 8086 internal structure. Most internal functions of the 8088 are identical to the equivalent 8086 functions. The 8088 handles the external bus the same way the 8086 does with the distinction of handling only 8 bits at a time. Sixteen-bit operands are fetched or written in two consecutive bus cycles. Both processors will appear identical to the software engineer, with the exception of execution time. The internal register structure is identical and all instructions have the same end result. The differences between the 8088 and 8086 are outlined below. The engineer who is unfamiliar with the 8086 is referred to the 8086 Family User's Manual, Chapters 2 and 4, for function description and instruction set information.

Internally, there are three differences between the 8088 and the 8086. All changes are related to the 8-bit bus interface.

- The queue length is 4 bytes in the 8088, whereas the 8086 queue contains 6 bytes, or three words. The queue was shortened to prevent overuse of the bus by the BIU when prefetching instructions. This was required because of the additional time necessary to fetch instructions 8 bits at a time.

PRELIMINARY

- To further optimize the queue, the prefetching algorithm was changed. The 8088 BIU will fetch a new instruction to load into the queue each time there is a 1 byte hole (space available) in the queue. The 8086 waits until a 2-byte space is available.
- The internal execution time of the instruction set is affected by the 8-bit interface. All 16-bit fetches and writes from/to memory take an additional four clock cycles. The CPU is also limited by the speed of instruction fetches. This latter problem only occurs when a series of simple operations occur. When the more sophisticated instructions of the 8088 are being used, the queue has time to fill and the execution proceeds as fast as the execution unit will allow.

The 8088 and 8086 are completely software compatible by virture of their identical execution units. Software that is system dependent may not be completely transferable, but software that is not system dependent will operate equally as well on an 8088 or an 8086.

The hardware interface of the 8088 contains the major differences between the two CPUs. The pin assignments are nearly identical, however, with the following functional changes:

- A8–A15 — These pins are only address outputs on the 8088. These address lines are latched internally and remain valid throughout a bus cycle in a manner similar to the 8085 upper address lines.

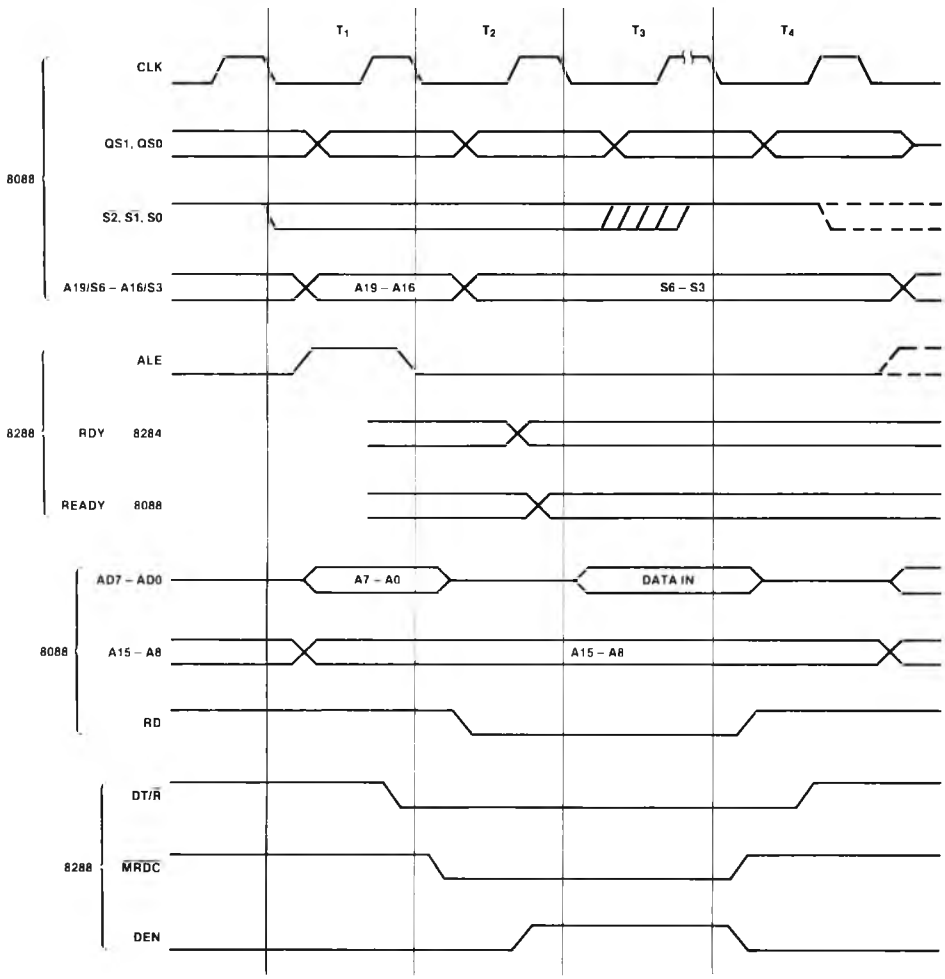


Figure 8. Medium Complexity System Timing

PRELIMINARY

- $\overline{BHE}$ has no meaning on the 8088 and has been eliminated.
- $\overline{SSO}$ provides the $\overline{SO}$ status information in the minimum mode. This output occurs on pin 34 in minimum mode only. DT/$\overline{R}$, IO/$\overline{M}$, and $\overline{SSO}$ provide the complete bus status in minimum mode.
- IO/$\overline{M}$ has been inverted to be compatible with the MCS-85 bus structure.
- ALE is delayed by one clock cycle in the minimum mode when entering HALT, to allow the status to be latched with ALE.

8088 FUNCTIONAL PIN DEFINITIONS

The following pin function descriptions are for 8088 systems in either minimum or maximum mode. The "local bus" in these descriptions is the direct multiplexed bus interface connection to the 8088 (without regard to additional bus buffers).

AD7-AD0 (Input/Output, 3-State)

These lines constitute the time multiplexed memory/IO address (T1) and data (T2, T3, Tw, and T4) bus. These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus "hold acknowledge"

A15-A8 (Output, 3-State)

These lines provide address bits 8 through 15 for the entire bus cycle (T1-T4). These lines do not have to be latched by ALE to remain valid. A15-A8 are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus "hold acknowledge".

A19/S6, A18/S5, A17/S4, A16/S3 (Output, 3-State)

During T1, these are the four most significant address lines for memory operations. During I/O operations, these lines are LOW. During memory and I/O operations, status information is available on these lines during T2, T3, Tw, and T4. S6 is always low. The status of the interrupt enable flag bit (S5) is updated at the beginning of each clock cycle. S4 and S3 are encoded as follows:

S4	S3	
0 (LOW)	0	Alternate Data
0	1	Stack
1 (HIGH)	0	Code or None
1	1	Data
S6 is 0 (LOW)		

This information indicates which segment register is presently being used for data accessing.

These lines float to 3-state OFF during local bus "hold acknowledge"

$\overline{RD}$ (Output, 3-State)

Read strobe indicates that the processor is performing a memory or I/O read cycle, depending on the state of the IO/$\overline{M}$ pin or S2. This signal is used to read devices which reside on the 8088 local bus. $\overline{RD}$ is active LOW during T2, T3 and T_W of any read cycle, and is guaranteed to remain HIGH in T2 until the 8088 local bus has floated.

This signal floats to 3-state OFF in "hold acknowledge".

READY (Input)

READY is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The RDY signal from memory or I/O is synchronized by the 8284 clock generator to form READY. This signal is active HIGH.

INTR (Input)

Interrupt request is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.

$\overline{TEST}$ (Input)

The $\overline{TEST}$ input is examined by the "wait for test" instruction. If the $\overline{TEST}$ input is LOW, execution continues, otherwise the processor waits in an "idle" state. This input is synchronized internally during each clock cycle on the leading edge of CLK.

NMI (Input)

Non-maskable interrupt is an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from a LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.

RESET (Input)

RESET causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the instruction set description, when RESET returns LOW. RESET is internally synchronized.

CLK (Input)

The clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.

V_{CC}

V_{CC} is the +5V ± 10% power supply pin.

GND

GND are the ground pins.

PRELIMINARY

MINIMUM MODE PIN DESCRIPTIONS

The following pin function descriptions are for the 8088 minimum mode (i.e., MN/$\overline{MX}$ = V_{CC}). Only the pin functions which are unique to minimum mode are described; all other pin functions are as described above.

IO/$\overline{M}$ (Output, 3-State)

This status line is an inverted maximum mode $\overline{S2}$. It is used to distinguish a memory access from an I/O access. IO/$\overline{M}$ becomes valid in the T4 preceding a bus cycle and remains valid until the final T4 of the cycle (I/O = HIGH, M = LOW). IO/$\overline{M}$ floats to 3-state OFF in local bus "hold acknowledge".

$\overline{WR}$ (Output, 3-State)

Write strobe indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the IO/$\overline{M}$ signal. WR is active for T2, T3, and Tw of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge".

$\overline{INTA}$ (Output, 3-State)

$\overline{INTA}$ is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T2, T3, and Tw of each interrupt acknowledge cycle. $\overline{INTA}$ floats to 3-state OFF in "hold acknowledge".

ALE (Output)

Address latch enable (ALE) is provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during clock low of T1 of any bus cycle. Note that ALE is never floated.

DT/$\overline{R}$ (Output, 3-State)

Data transmit/receive is needed in a minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically, DT/$\overline{R}$ is equivalent to $\overline{S1}$ in the maximum mode, and its timing is the same as for IO/$\overline{M}$ (T = HIGH, R = LOW). This signal floats to 3-state OFF in local "hold acknowledge".

$\overline{DEN}$ (Output, 3-State)

Data enable is provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{DEN}$ is active LOW during each memory and I/O access, and for $\overline{INTA}$ cycles. For a read or $\overline{INTA}$ cycle, it is active from the middle of T2 until the middle of T4, while for a write cycle, it is active from the beginning of T2 until the middle of T4. $\overline{DEN}$ floats to 3-state OFF during local bus "hold acknowledge".

HOLD (Input), HLDA (Output)

HOLD indicates that another master is requesting a local bus "hold". To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement, in the middle of T4 or TI. Simultaneous with the issuance of HLDA, the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor lowers HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines.

$\overline{SSO}$

This status line is logically equivalent to $\overline{S0}$ in the maximum mode. The combination of $\overline{SSO}$, IO/$\overline{M}$ and DT/$\overline{R}$ allows the system to completely decode the current bus cycle status.

IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SSO}$	
1 (HIGH)	0	0	Interrupt Acknowledge
1	0	1	Read I/O port
1	1	0	Write I/O port
1	1	1	Halt
0 (LOW)	0	0	Code access
0	0	1	Read memory
0	1	0	Write memory
0	1	1	Passive

MAXIMUM MODE PIN DESCRIPTIONS

The following pin function descriptions are for the 8088, 8228 system in maximum mode (i.e., MN/$\overline{MX}$ = GND. Only the pin functions which are unique to maximum mode are described; all other pin functions are as described above.

$\overline{S2}$, $\overline{S1}$, $\overline{S0}$ (Output, 3-State)

These status lines are encoded as follows:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1 (HIGH)	0	0	Code access
1	0	0	Read memory
1	1	0	Write memory
1	1	1	Passive

Status is active during clock high of T4, T1, and T2, and is returned to the passive state (1,1,1) during T3 or during T_W when READY is HIGH. This status is used by the 8288 bus controller to generate all memory and I/O access control signals. Any change by $\overline{S2}$, $\overline{S1}$, or $\overline{S0}$ during T4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T3 or T_W is used to indicate the end of a bus cycle.

These signals float to 3-state OFF during "hold acknowledge". During the first clock cycle after RESET becomes active, these signals are active HIGH. After this first clock, they float to 3-state OFF.

$\overline{RQ}$/$\overline{GT0}$, $\overline{RQ}$/$\overline{GT1}$ (Input/Output)

The request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin

is bidirectional with $\overline{RQ}/\overline{GT0}$ having higher priority than $\overline{RQ}/\overline{GT1}$. $\overline{RQ}/\overline{GT}$ has an internal pull-up resistor, so may be left unconnected. The request/grant sequence is as follows (See Figure 6):

1. A pulse of one CLK wide from another local bus master indicates a local bus request ("hold") to the 8088 (pulse 1).

2. During the CPU's next T4 or TI, a pulse one clock wide from the 8088 to the requesting master (pulse 2), indicates that the 8088 has allowed the local bus to float and that it will enter the "hold acknowledge" state at the next CLK. The CPU's bus interface unit is disconnected logically from the local bus during "hold acknowledge".

3. A pulse one CLK wide from the requesting master indicates to the 8088 (pulse 3) that the "hold" request is about to end and that the 8088 can reclaim the local bus at the next CLK. The CPU then enters T4.

Each master-master exchange of the local bus is a sequence of three pulses. There must be one idle CLK cycle after each bus exchange. Pulses are active LOW.

$\overline{LOCK}$ (Output, 3-State)

The $\overline{LOCK}$ output indicates that other system bus masters are not to gain control of the system bus while $\overline{LOCK}$ is active (LOW). The $\overline{LOCK}$ signal is activated by the "LOCK" prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state off in "hold acknowledge".

QS1, QS0 (Output)

QS1 and QS0 provide status to allow external tracking of the internal 8088 instruction queue.

QS1	QS0	
0 (LOW)	0	No operation
0	1	First byte of opcode from queue
1 (HIGH)	0	Empty the queue
1	1	Subsequent byte from queue

The queue status is valid during the CLK cycle after which the queue operation is performed.

PIN 34 (Output)

Pin 34 is always high in the maximum mode.

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . – 65°C to + 150°C
Voltage on Any Pin with
Respect to Ground . . . . . – 0.3 to + 7V
Power Dissipation . . . . . 2.5 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS

8088: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	– 0.5	+ 0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} + 0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = 400 μA
I_{CC}	Power Supply Current		340	mA	
I_{LI}	Input Leakage Current		± 10	μA	$V_{IN} = V_{CC}$
I_{LO}	Output Leakage Current		± 10	μA	$0.45V \leq V_{OUT} \leq V_{CC}$
V_{CL}	Clock Input Low Voltage	– 0.5	+ 0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC} + 1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except AD_0-AD_7 RQ/GT)		10	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer (AD_0-AD_7 RQ/GT)		20	pF	fc = 1 MHz

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

A.C. CHARACTERISTICS

8088: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

8088 MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(2/3TCLCL)-15		ns	
TCHCL	CLK High Time	(1/3TCLCL)+2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		ns	
TRYHCH	READY Setup Time into 8088	(2/3TCLCL)-15		ns	
TCHRYX	READY Hold Time into 8088	30		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	-8		ns	
THVCH	HOLD Setup Time	35		ns	
TINVCH	INTR, NMI, $\overline{TEST}$ Setup Time (See Note 2)	30		ns	

TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLAV	Address Valid Delay	15	110	ns	C_L = 20-100 pF for all 8088 Outputs in addition to internal loads
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TLHLL	ALE Width	TCLCH-20		ns	
TCLLH	ALE Active Delay		80	ns	
TCHLL	ALE Inactive Delay		85	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL-10		ns	
TCLDV	Data Valid Delay	10	110	ns	
TCHDX	Data Hold Time	10		ns	
TWHDX	Data Hold Time After $\overline{WR}$	TCLCH-30		ns	
TCVCTV	Control Active Delay 1	10	110	ns	
TCHCTV	Control Active Delay 2	10	110	ns	
TCVCTX	Control Inactive Delay	10	110	ns	
TAZRL	Address Float to READ Active	0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL-45		ns	
TCLHAV	HLDA Valid Delay	10	160	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL-75		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL-60		ns	
TAVAL	Address Valid to ALE Low	TCLCH-60		ns	

NOTES: 1. Signal at 8284 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state (8 ns into T3 state).

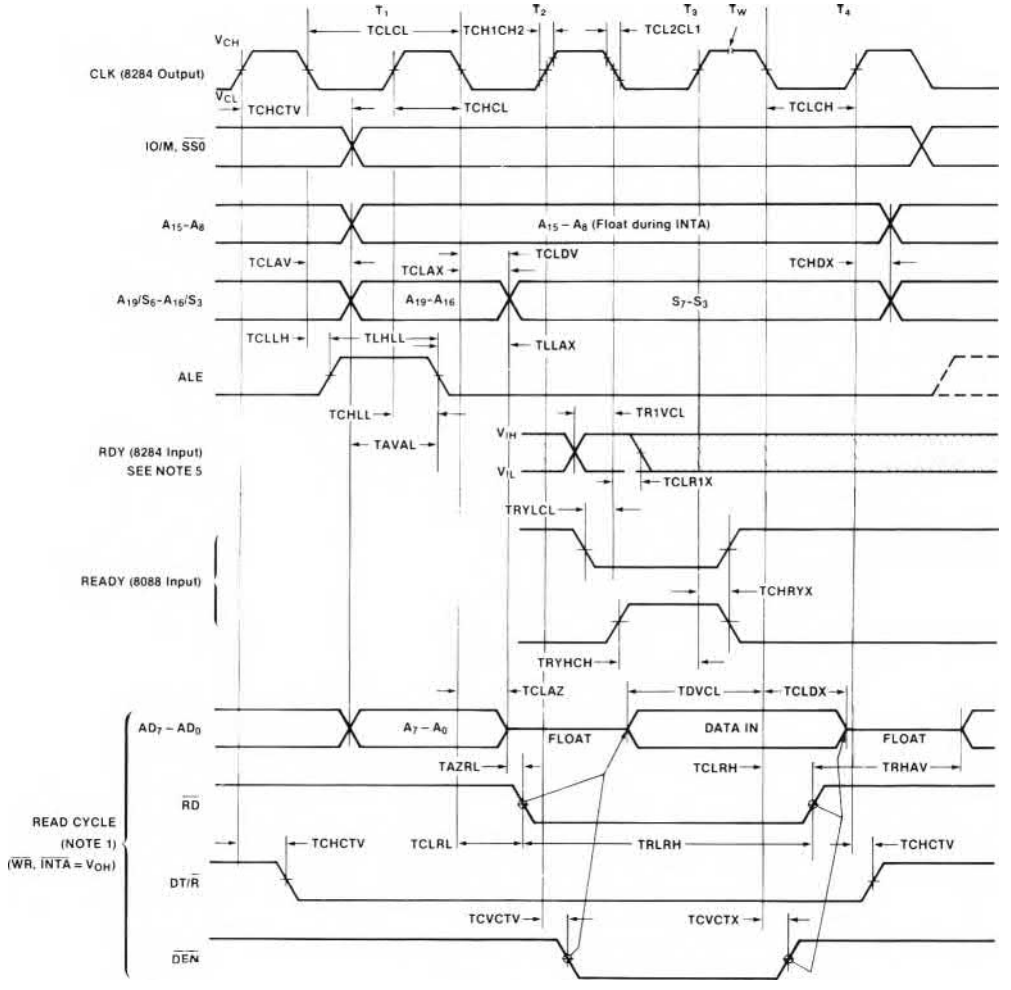
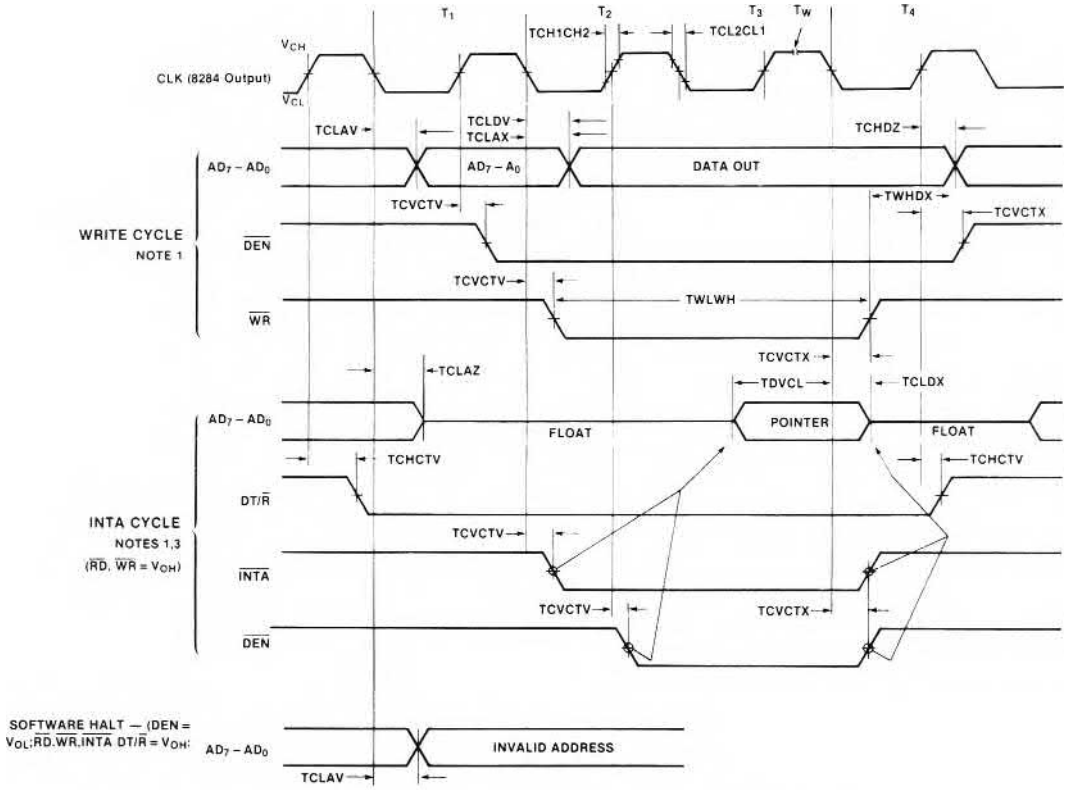


Figure 9. 8088 Bus Timing — Minimum Mode System

PRELIMINARY



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL SIGNALS ARE SHOWN FOR THE SECOND INTA CYCLE.
4. SIGNALS AT 8284 ARE SHOWN FOR REFERENCE ONLY.
5. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

Figure 10. 8088 Bus Timing — Minimum Mode System (cont.)

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

8088 MAX MODE SYSTEM (USING 8288 BUS CONTROLLER) TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(2/3TCLCL)-15		ns	
TCHCL	CLK High Time	(1/3TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		ns	
TRYHCH	READY Setup Time into 8088	(2/3TCLCL)-15		ns	
TCHRYX	READY Hold Time into 8088	30		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	-8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, $\overline{TEST}$) (See Note 2)	30		ns	
TGVCH	$\overline{RQ}/\overline{GT}$ Setup Time	30		ns	
TCHGX	$\overline{RQ}$ Hold Time into 8086	40		ns	

TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	ns	
TCLMH	Command Inactive Delay (See Note 1)	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110	ns	
TCHSV	Status Active Delay	10	110	ns	
TCLSH	Status Inactive Delay	10	130	ns	
TCLAV	Address Valid Delay	15	110	ns	
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15	ns	C_L = 20-100 pF for all 8088 Outputs in addition to internal loads
TCLMCL	MCE Inactive Delay (See Note 1)		15	ns	
TCLDV	Data Valid Delay	15	110	ns	
TCHDX	Data Hold Time	10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	ns	
TAZRL	Address Float to Read Active	0		ns	
TCLRL	RD Active Delay	10	165	ns	
TCLRH	RD Inactive Delay	10	150	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL-45		ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30	ns	
TCLGL	$\overline{GT}$ Active Delay		110	ns	
TCLGH	$\overline{GT}$ Inactive Delay		85	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL-75		ns	

NOTES: 1. Signal at 8284 or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T3 and wait states.
4. Applies only to T2 state (8 ns into T3 state).

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

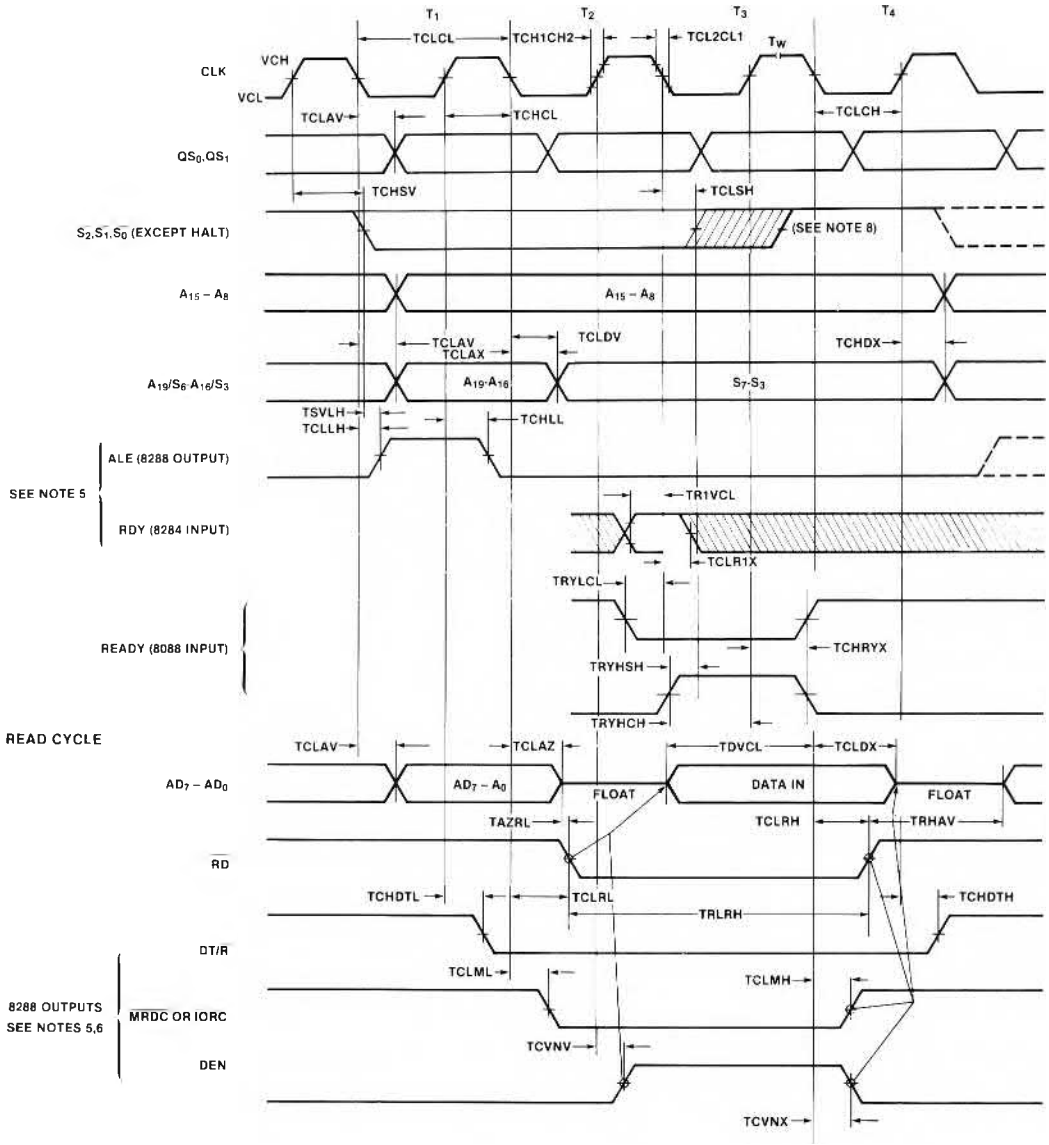
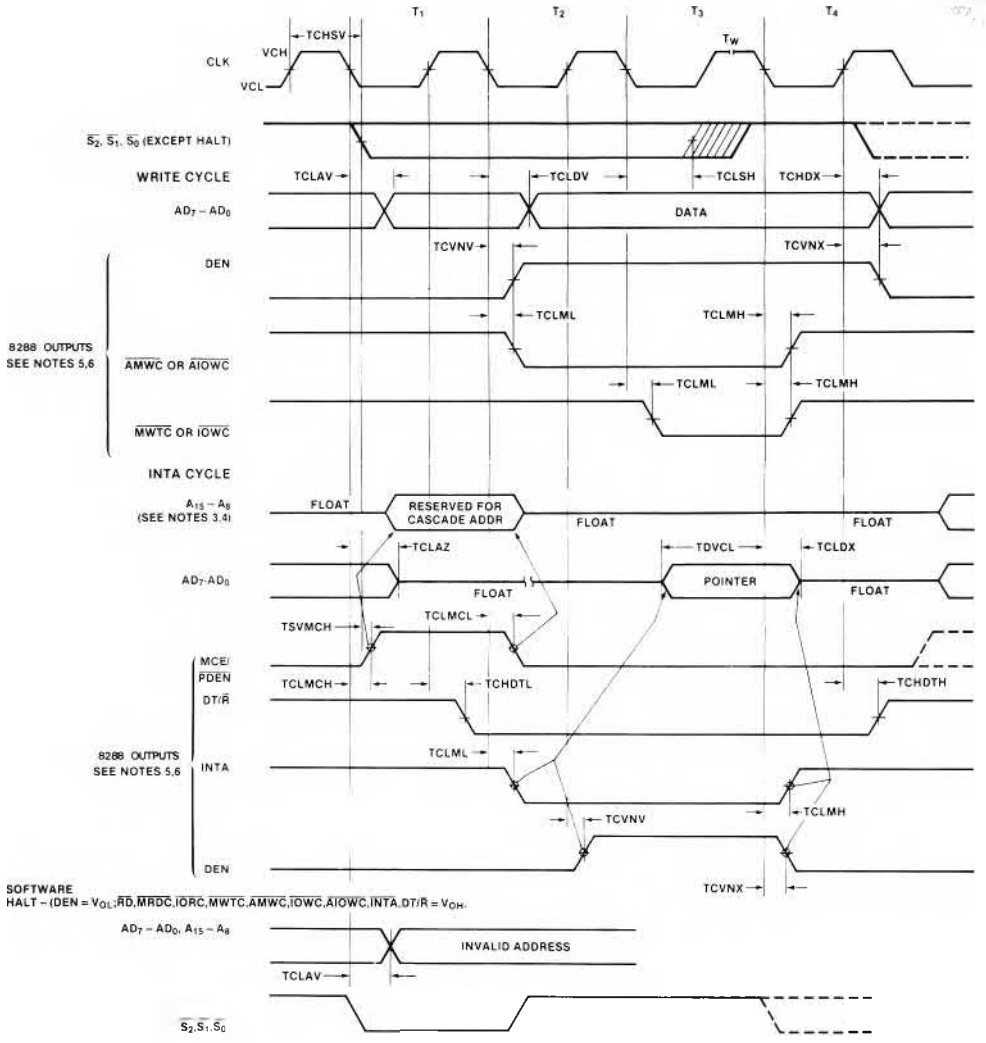


Figure 11. 8088 Bus Timing — Maximum Mode System (Using 8288)

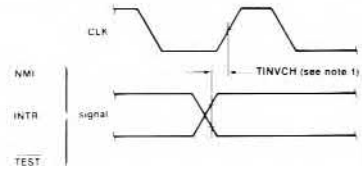
PRELIMINARY



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. CASCADE ADDRESS IS VALID BETWEEN FIRST AND SECOND INTA CYCLES.
4. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL FOR POINTER ADDRESS IS SHOWN FOR SECOND INTA CYCLE.
5. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
6. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS (MRDC, MWTC, AMWC, IORC, IOWC, AIOWC, INTA AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
7. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
8. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.

Figure 12. 8088 Bus Timing — Maximum Mode System (Using 8288)

PRELIMINARY



NOTE:
1. SETUP REQUIREMENTS FOR ASYNCHRONOUS SIGNALS ONLY TO GUARANTEE RECOGNITION AT NEXT CLK

Figure 13. Asynchronous Signal Recognition

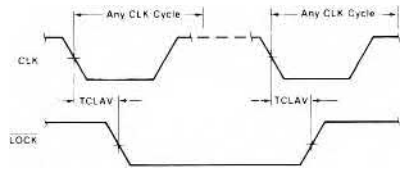
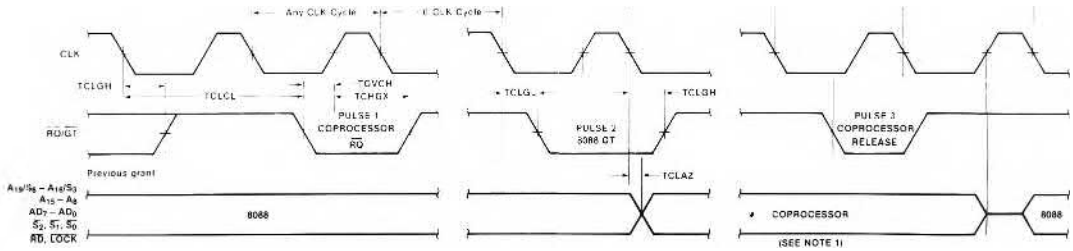


Figure 14. Bus Lock Signal Timing (Maximum Mode Only)



NOTE: 1. THE COPROCESSOR MAY NOT DRIVE THE BUSSES OUTSIDE THE REGION SHOWN WITHOUT RISKING CONTENTION.

Figure 15. Request/Grant Sequence Timing (Maximum Mode Only)

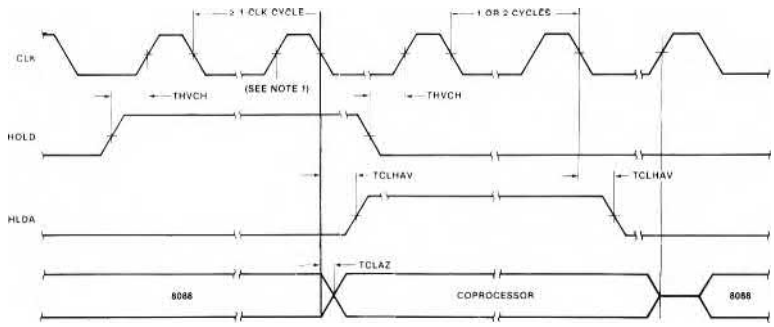


Figure 16. Hold/Hold Acknowledge Timing (Minimum Mode Only)

PRELIMINARY

8086/8088
INSTRUCTION SET SUMMARY

DATA TRANSFER

MOV Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to register	1 0 1 1 w reg	data	data if w = 1	
Memory to accumulator	1 0 1 0 0 0 0 w	addr low	addr high	
Accumulator to memory	1 0 1 0 0 0 1 w	addr low	addr high	
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		

PUSH Push

Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			

POP Pop

Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			

XCHG Exchange

Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with accumulator	1 0 0 1 0 reg			

IN = Input from

Fixed port	1 1 1 0 0 1 0 w	port		
Variable port	1 1 1 0 1 1 0 w			

OUT = Output to

Fixed port	1 1 1 0 0 1 1 w	port		
Variable port	1 1 1 0 1 1 1 w			
XLAT Translate byte to AL	1 1 0 1 0 1 1 1			
LEA Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF Load AH with flags	1 0 0 1 1 1 1 1			
SAHF Store AH into flags	1 0 0 1 1 1 1 0			
PUSHF Push flags	1 0 0 1 1 1 0 0			
POPF Pop flags	1 0 0 1 1 1 0 1			

ARITHMETIC

ADD Add

Reg./memory with register to either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s w = 01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w = 1	

ADC Add with carry

Reg./memory with register to either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s w = 01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w = 1	

INC Increment

Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA ASCII adjust for add	0 0 1 1 0 1 1 1			
DAA Decimal adjust for add	0 0 1 0 0 1 1 1			

SUB Subtract

Reg./memory and register to either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s w = 01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w = 1	

SBB Subtract with borrow

Reg./memory and register to either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w = 01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w = 1	

DEC Decrement

Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		

CMP Compare

Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS ASCII adjust for subtract	0 0 1 1 1 1 1 1			
DAS Decimal adjust for subtract	0 0 1 0 1 1 1 1			
MUL Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW Convert byte to word	1 0 0 1 1 0 0 0			
CWD Convert word to double word	1 0 0 1 1 0 0 1			

LOGIC

NOT Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL Shift logical/arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		

AND And

Reg./memory and register to either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w = 1	

TEST And function to flags, no result

Register/memory and register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data	data if w = 1	

OR Or

Reg./memory and register to either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w = 1	

XOR Exclusive or

Reg./memory and register to either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w = 1	

STRING MANIPULATION

REP = Repeat	1 1 1 1 0 0 1 z
MOVS = Move byte/word	1 0 1 0 0 1 0 w
CMPS = Compare byte/word	1 0 1 0 0 1 1 w
SCAS = Scan byte/word	1 0 1 0 1 1 1 w
LODS = Load byte/wd to AL/AX	1 0 1 0 1 1 0 w
STOS = Stor byte/wd from AL/A	1 0 1 0 1 0 1 w

Mnemonics ©Intel, 1978

PRELIMINARY

CONTROL TRANSFER

CALL Call	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	

JMP Unconditional Jump			
Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within segment short	1 1 1 0 1 0 1 1	disp	
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

RET Return from CALL			
Within segment	1 1 0 0 0 0 1 1		
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ Jump on equal/zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp	
JP/JPE Jump on parity/parity even	0 1 1 1 1 0 1 0	disp	
JO Jump on overflow	0 1 1 1 0 0 0 0	disp	
JS Jump on sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp	

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNB/JAE Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp
JNBE/JA Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp
JNP/JPO Jump on not par/par odd	0 1 1 1 1 0 1 1	disp
JNO Jump on not overflow	0 1 1 1 0 0 0 1	disp
JNS Jump on not sign	0 1 1 1 1 0 0 1	disp
LOOP Loop CX times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE Loop while zero/equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE Loop while not zero/equal	1 1 1 0 0 0 0 0	disp
JCXZ Jump on CX zero	1 1 1 0 0 0 1 1	disp

INT Interrupt		
Type specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO Interrupt on overflow	1 1 0 0 1 1 1 0	
IRET Interrupt return	1 1 0 0 1 1 1 1	

PROCESSOR CONTROL

CLC Clear carry	1 1 1 1 1 0 0 0
CMC Complement carry	1 1 1 1 0 1 0 1
STC Set carry	1 1 1 1 1 0 0 1
CLD Clear direction	1 1 1 1 1 0 0 0
STD Set direction	1 1 1 1 1 0 0 1
CLI Clear interrupt	1 1 1 1 1 0 1 0
STI Set interrupt	1 1 1 1 1 0 1 1
HLT Halt	1 1 1 1 0 4 0 0
WAIT Wait	1 0 0 1 1 0 1 1
ESC Escape (to external device)	1 1 0 1 1 x x x mod x x x r/m
LOCK Bus lock prefix	1 1 1 1 0 0 0 0

Footnotes:

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if s w = 01 then 16 bits of immediate data form the operand
if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high : disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high : disp-low

REG is assigned according to the following table

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file

FLAGS = X X X X (OF) (DF) (IF) (TF) (SF) (ZF) X (AF) X (PF) X (CF)

Mnemonics © Intel, 1978

intel®

8089
8/16-BIT HMOS I/O PROCESSOR

- **High Speed DMA capabilities including I/O to memory, memory to I/O, memory to memory and I/O to I/O**
- **MCS-80™, MCS-85™, MCS-86™ and 8088 compatible, removes I/O overhead**
- **Allows mixed interface of 8/16-bit peripherals, to 8/16-bit processor busses**
- **1 Mbyte addressability**
- **Memory based communication with CPU**
- **Supports LOCAL or REMOTE I/O processing**
- **Flexible, intelligent DMA functions including Translation, Search, Word Assembly/Disassembly**
- **MULTIBUS™ compatible system interface**

The Intel® 8089 is a revolutionary concept in microprocessor input/output processing. Packaged in a 40-pin DIP package, the 8089 is a high performance processor implemented in N-channel, depletion load silicon gate technology (HMOS). The 8089's instruction set and capabilities are optimized for high speed, flexible and efficient I/O handling. It allows easy interface of Intel's 16-bit 8086 and 8-bit 8088 microprocessors with 8/16-bit peripherals. In the REMOTE mode, the 8089 bus is user definable allowing it to be compatible with any 8/16-bit Intel microprocessor, interfacing easily to the Intel multiprocessor system bus standard MULTIBUS™.

The 8089 performs the function of an intelligent DMA controller for the Intel MCS-86 family and with its processing power, can remove I/O overhead from the 8086 or 8088. It may operate completely in parallel with a CPU, giving dramatically improved performance in I/O intensive applications. The 8089 provides two I/O channels, each supporting a transfer rate up to 1.25 mbyte/sec at the standard clock frequency of 5 MHz. Memory based communication between the IOP and CPU enhances system flexibility and encourages software modularity, yielding more reliable, easier to develop systems.

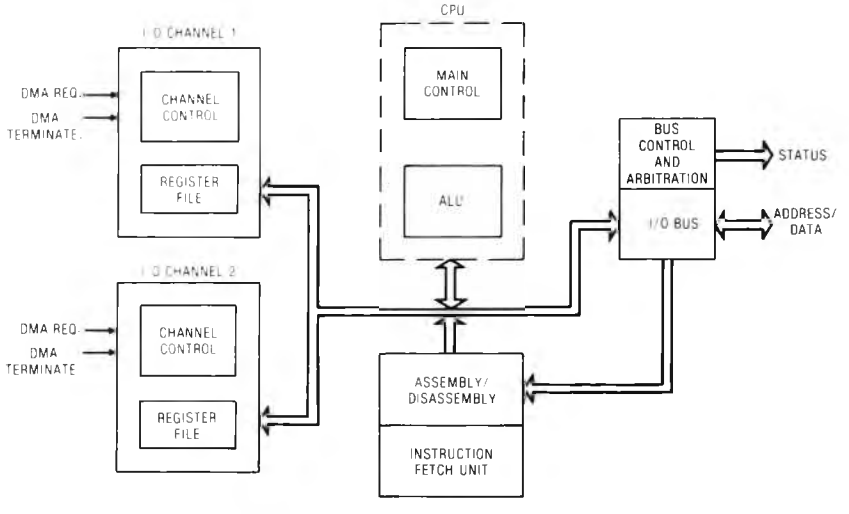


Figure 1. 8089 I/O Processor Block Diagram

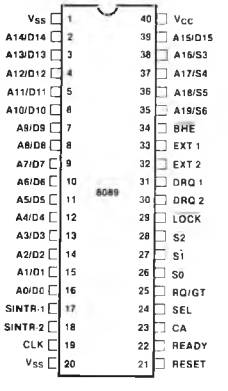


Figure 2. 8089 Pin Diagram

PRELIMINARY

FUNCTIONAL DESCRIPTION

The 8089 IOP has been designed to remove I/O processing, control and high speed transfers from the central processing unit. Its major capabilities include that of initializing and maintaining peripheral components and supporting versatile DMA. This DMA function boasts flexible termination conditions (such as external terminate, mask compare, single transfer and byte count expired). The DMA function of the 8089 IOP uses a two cycle approach where the information actually flows through the 8089 IOP. This approach to DMA vastly simplifies the bus timings and enhances compatibility with memory and peripherals, in addition to allowing operations to be performed on the data as it is transferred. Operations can include such constructs as translate, where the 8089 automatically vectors through a lookup table and mask compare, both on the "fly".

The 8089 is functionally compatible with Intel's 8086, 8088 family. It supports any combination of 8/16-bit busses. In the REMOTE mode it can be used to complement other Intel processor families. Hardware and communication architecture are designed to provide simple mechanisms for system upgrade.

The only direct communication between the IOP and CPU is handled by the Channel Attention and Interrupt lines. Status information, parameters and task programs are passed via blocks of shared memory, simplifying hardware interface and encouraging structured programming.

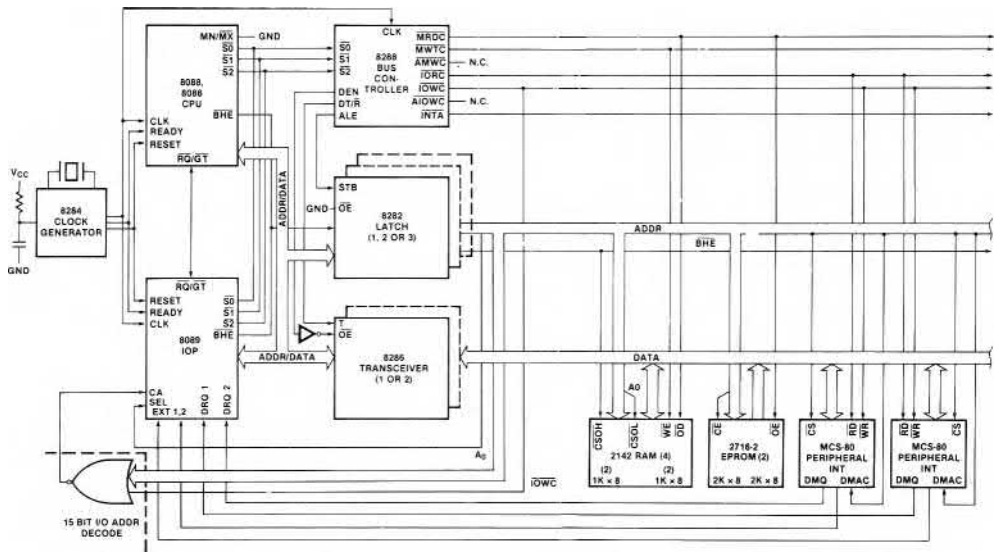
The 8089 can be used in applications such as file and buffer management in hard disk or floppy disk control. It can also provide for soft error recovery routines and scan control. CRT control, such as cursor control and auto scrolling, is simplified with the 8089. Keyboard control, communication control and general I/O are just a few of the typical applications for the 8089.

Remote and Local Modes

Shown in Figure 3 is the 8089 configured in a LOCAL mode. The 8086 (or 8088) is used in its maximum mode configuration. The 8089 and 8086 reside on the same local bus, sharing the same set of system buffers. Peripherals located on the system bus can be addressed by either the 8086 or the 8089. The 8089 requests the use of the LOCAL bus by means of the $\overline{RQ}/\overline{GT}$ line. This performs a similar function to that of HOLD and HLDA on the Intel 8085A, 8080A and 8086 minimum mode, but is implemented on one physical line. When the 8086 relinquishes the system bus, the 8089 uses the same bus control, latches and transceiver components to generate the system address, control and data lines. This mode allows a more economical system configuration at the expense of reduced CPU thruput due to IOP bus utilization.

A typical REMOTE configuration is shown in Figure 4. In this mode, the IOP's bus is physically separated from the system bus by means of system buffers/latches. The IOP maintains its own local bus and can operate out of local or system memory. The system bus interface contains the following components:

- Up to three 8282 buffer/latches to latch the address to the system bus
- Up to two 8286 devices bidirectionally buffer the system data bus



NOTE: ONLY ONE LATCH IS NEEDED IF CONFIGURED WITH 8088 AND ONLY 64K ADDRESSING IS USED. ONLY ONE TRANSCEIVER IS NEEDED IF USING A PHYSICAL 8-BIT DATA BUS (8088)

Figure 3. Typical 8088, 8086/8089 Configuration with 8089 in LOCAL Mode, 8088, 8086 in MAX Mode

- An 8288 bus controller supplies the control signals necessary for buffer operation as well as MRDC (Memory Read) and MWTC (Memory Write) signals.
- An 8289 bus arbiter performs all the functions necessary to arbitrate the use of the system bus. This is used in place of the $\overline{RQ}/\overline{GT}$ logic in the LOCAL mode. This arbiter decodes type of cycle information from the 8089 status lines to determine if the IOP desires to perform a transfer over the "common" or system bus.

The peripheral devices P1 and P2 are supported on their own data and address bus. The 8089 communicates with the peripherals without affecting system bus operation. Optional buffers may be used on the local bus when capacitive loading conditions so dictate. I/O programs and RAM buffers may also reside on the local bus to further reduce system bus utilization.

COMMUNICATION MECHANISM

Fundamentally, communication between the CPU and IOP is performed through messages prepared in shared memory. The CPU can cause the 8089 to execute a program by placing it in the 8089's memory space and/or directing the 8089's attention to it by asserting a hardware Channel Attention (CA) signal to the IOP, activating the proper I/O channel. The SEL Pin indicates to the IOP which channel is being addressed. Communication from the IOP to the processor can be performed in a similar manner via a system interrupt (SINTR 1,2), if the CPU has enabled interrupts for this purpose. Additionally, the 8089 can store messages in memory regarding its status and the status of any peripherals. This communication mechanism is supported by a hierarchial data structure to provide a maximum amount of flexibility of memory use with the added capability of handling multiple IOP's.

Illustrated in Figure 5 is an overview of the communication data structure hierarchy that exists for the 8089 I/O processor. Upon the first CA from RESET, 5 bytes of information are read into the 8089 starting at location FFFF6 (FFFF6, FFFF8-FFFFB) where the type of system bus (16-bit or 8-bit) and pointers to the system configuration are obtained. This is the only fixed location the 8089 accesses. The remaining addresses are obtained via the data structure hierarchy. The 8089 determines addresses in the same manner as does the 8086; i.e., a 16-bit relocation pointer is offset left 4 bits and added to the 16-bit address offset, obtaining a 20-bit address. Once these 20-bit addresses are formed, they are stored as such, as all the 8089 address registers are 20-bits long. After the system configuration pointer address is formed, the 8089 IOP accesses the system configuration block.

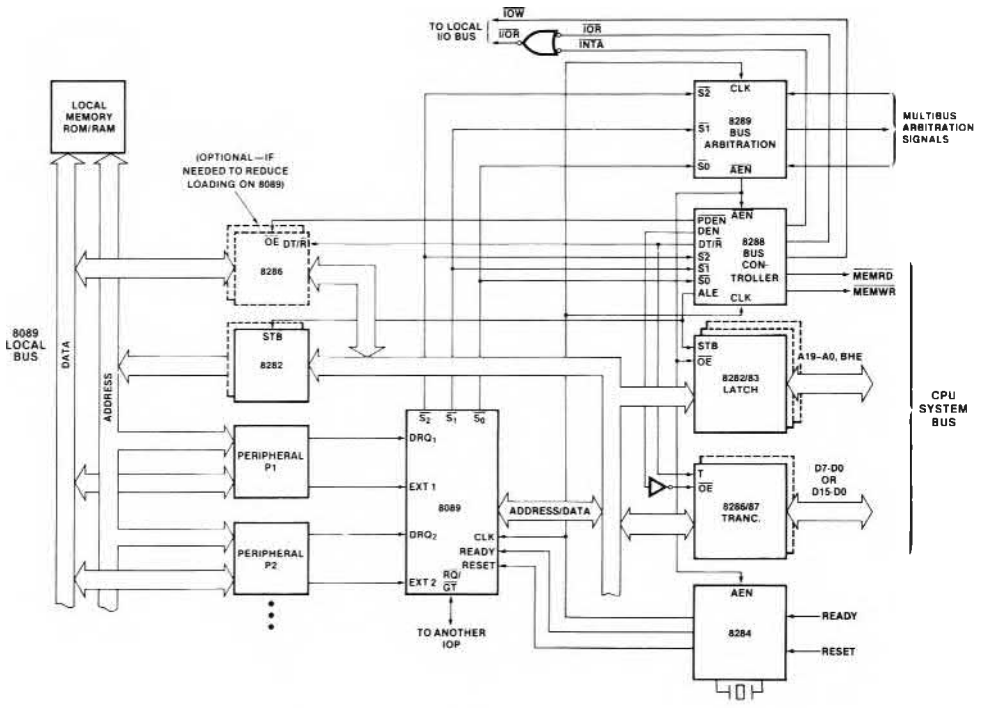


Figure 4. Typical REMOTE Configuration

PRELIMINARY

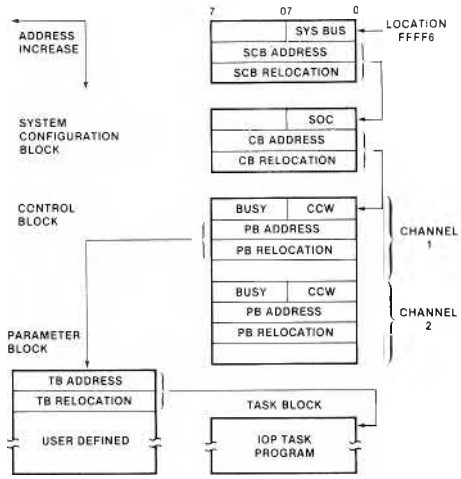


Figure 5. Communication Data Structure Hierarchy

The System Configuration Block (SCB), used only during startup, points to the Control Block (CB) and provides IOP system configuration data via the SOC byte. The SOC byte initializes IOP I/O bus width to 8/16, and defines one of two IOP $\overline{RQ}/\overline{GT}$ operating modes. For $\overline{RQ}/\overline{GT}$ mode 0, the IOP is typically initialized as SLAVE and has its $\overline{RQ}/\overline{GT}$ line tied to a MASTER CPU (typical LOCAL configuration). In this mode, the CPU normally has control of the bus, grants control to the IOP as needed, and has the bus restored to it upon IOP task completion (IOP request—CPU grant—IOP done). For $\overline{RQ}/\overline{GT}$ mode 1, useful only in remote mode between two IOPs, MASTER/SLAVE designation is used only to initialize bus control: from then on, each IOP requests and grants as the bus is needed (IOP1 request—IOP2 grant—IOP2 request—IOP1 grant). Thus, each IOP retains bus control until the other requests it. The completion of initialization is signalled by the IOP clearing the BUSY flag in the CB. This type of startup allows the user to have the startup pointers in ROM with the SCB in RAM. Allowing the SCB to be in RAM gives the user the flexibility of being able to initialize multiple IOPs.

The Control Block furnishes bus control Initialization for the IOP operation (CCW or Channel Control Word) and provides pointers to the Parameter Block or "data" memory for both channels 1 and 2. The CCW is retrieved and analyzed upon all CA's other than the first after a reset. The CCW byte is decoded to determine channel operation.

The Parameter Block contains the address of the Task Block and acts as a messge center between the IOP and CPU. Parameters or variable information is passed from the CPU to its IOP in this block to customize the software interface to the peripheral device. It is also used for transferring data and status information between the IOP and CPU.

The Task Block contains the instructions for the respective channel. This block can reside on the local bus of the IOP, allowing the IOP to operate concurrently with the CPU, or reside in system memory.

The advantage of this type of communication between the processor, IOP and peripheral, is that it allows for a very clean method for the operating system to handle I/O routines. Canned programs or "Task Blocks" allow for execution of general purpose I/O routines with the status and peripheral command information being passed via the Parameter Block ("data" memory). Task Blocks (or "program" memory) can be terminated or restarted by the CPU, if need be. Clearly, the flexibility of this communication lends itself to modularity and applicability to a large number of peripheral devices and upward compatibility to future end user systems and microprocessor families.

Register Set

The 8089 maintains separate registers for its two I/O channels as well as some common registers (see Figure 6). There are sufficient registers for each channel to sustain its own DMA transfers, and process its own instruction stream. The basic DMA pointer registers (GA, GB — 20 bits each), can point to either the system bus or local bus, DMA source or destination, and can be autoincremented. A third register set (GC) can be used to allow translation during the DMA process through a lookup table it points to. Additionally, registers are provided for a masked compare during the data transfer and can be set up to act as one of the termination conditions. Other registers are also provided. Many of these registers can be used as general purpose registers during program execution, when the IOP is not performing DMA cycles.

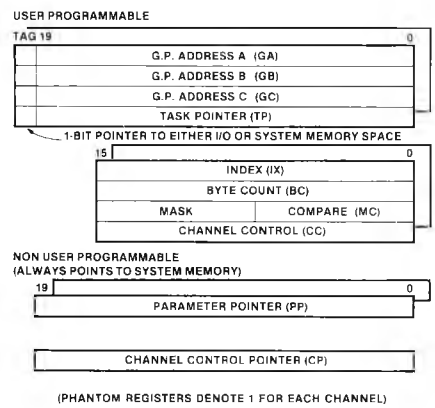


Figure 6. Register Model

Bus Operation

The 8089 utilizes the same bus structure as the 8086/8088 in their maximum mode configurations (see Figure 7). The address is time multiplexed with the data on the first 16/8 lines. A16 through A19 are time multiplexed with four status lines S3-S6. For 8089 cycles, S4 and S3 determine what type of cycle (DMA versus non-DMA) is being performed on channels 1 or 2. S5 and S6

are a unique code assigned to the 8089 IOP, enabling the user to detect which processor is performing a bus cycle in a multiprocessing environment.

The first three status lines, S0-S2, are used with an 8288 bus controller to determine if an instruction fetch or data transfer is being performed in I/O or system memory space.

DMA transfers require at least two bus cycles with each bus cycle requiring a minimum of four clock cycles. Additional clock cycles are added if wait states are required. This two cycle approach simplifies considerably the bus timings in burst DMA. The 8089 optimizes the transfer between two different bus widths by using three bus cycles versus four to transfer 1 word. More than one read (write) is performed when mapping an 8-bit bus onto a 16-bit bus (vice versa). For example, a data transfer from an 8-bit peripheral to a 16-bit physical location in memory is performed by first doing two reads, with word assembly within the IOP assembly register file and then one write.

As can be expected, the data bandwidth of the IOP is a function of the physical bus width of the system and I/O busses. Table 1 gives the bandwidth, latency and bus utilization of the 8089. The system bus is assumed to be 16-bits wide with either an 8-bit peripheral (under byte column) or 16-bit peripheral (word column) being shown.

The latency refers to the worst case response time by the IOP to a DMA request, without the bus arbitration times. Notice that the word transfer allows 50% more bandwidth. This occurs since three bus cycles are required to map 8-bit data into a 16-bit location, versus two for a 16-bit to 16-bit transfer. Note that it is possible to fully saturate the system bus in the LOCAL mode whereas in the REMOTE mode this is reduced to a maximum of 50%.

	Local		Remote	
	Byte	Word	Byte	Word
Bandwidth	830 KB/S	1250 KB/S	830 KB/S	1250 KB/S
Latency	1.0/2.4 μsec*	1.0/2.4 μsec*	1.0/2.4 μsec*	1.0/2.4 μsec*
System Bus Utilization	2.4 μsec PER TRANSFER	1.6 μsec PER TRANSFER	0.8 μsec PER TRANSFER	0.8 μsec PER TRANSFER

Table 1. 5 MHz 8089 Operation — With 16-Bit BUS

*2.4 μsec if interleaving with other channel and no wait states. 1μsec if channel is waiting for request.

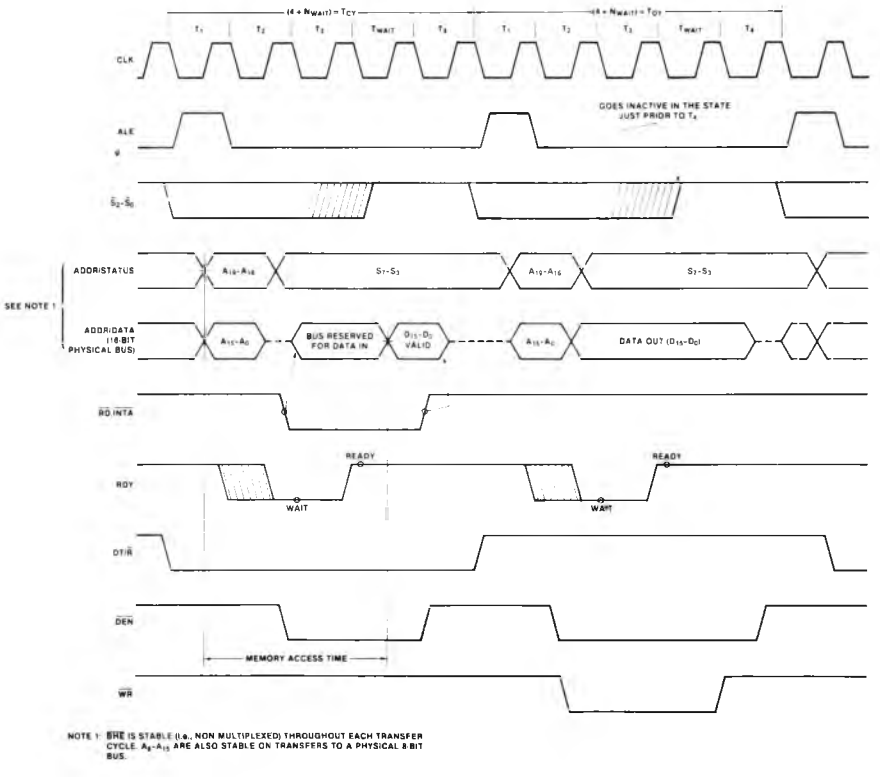


Figure 7. 8089 Bus Operation

PIN DESCRIPTION

Pin Name(s)	I/O	Description
A0–A15/ D0–D15	I/O	Multiplexed address and data bus. The function of these lines are defined by the state of $\overline{S0}$, $\overline{S1}$ and $\overline{S2}$ lines. The pins are floated after reset and when the bus is not acquired. A8–A15 are stable on transfers to a physical 8-bit data bus (same bus as 8088), and are multiplexed with data on transfers to a 16-bit physical bus.
A16–A19/ S3–S6	O	Multiplexed most significant address lines and status information. The address lines are active only when addressing memory. Otherwise, the status lines are active and are encoded as shown below. The pins are floated after reset and when the bus is not acquired. **S6 S5 S4 S3** 1 1 0 0 DMA cycle on CH1 1 1 0 1 DMA cycle on CH2 1 1 1 0 Non-DMA cycle on CH1 1 1 1 1 Non-DMA cycle on CH2
$\overline{BHE}$	O	The Bus High Enable signal is used to enable data operations on the most significant half of the data bus (D8–D15). The signal is active low when a byte is to be transferred on the upper half of the data bus. The pin is floated after reset and when the bus is not acquired. $\overline{BHE}$ does not have to be latched.
$\overline{S0}$, $\overline{S1}$, $\overline{S2}$	O	These are the status pins that define the IOP activity during any given cycle. They are encoded as shown below: **$\overline{S2}$ $\overline{S1}$ $\overline{S0}$** 0 0 0 Instruction fetch; I/O space 0 0 1 Data fetch; I/O space 0 1 0 Data store; I/O space 0 1 1 Not used 1 0 0 Instruction fetch; System Memory 1 0 1 Data fetch; System Memory 1 1 0 Data store; System Memory 1 1 1 Passive The status lines are utilized by the bus controller and bus arbiter to generate all memory and I/O control signals. The signals change during T4 if a new cycle is to be entered while the return to passive state in T3 or T_W indicates the end of a cycle. The pins are floated after system reset and when the bus is not acquired.
READY	I	The ready signal received from the addressed device indicates that the device is ready for data transfer. The signal is active high and is synchronized by the 8284 clock generator.
$\overline{LOCK}$	O	The lock output signal indicates to the bus controller that the bus is needed for more than one contiguous cycle. It is set via the channel control register, and during the TSL instruction. The pin floats after reset and when the bus is not acquired. This output is active low.
RESET	I	The receipt of a reset signal causes the IOP to suspend all its activities and enter an idle state until a channel attention is received.
CLK	I	System clock which provides all timing needed for internal IOP operation.
CA	I	Channel Attention. Used to get the attention of the IOP. Upon the falling edge of this signal, the SEL input pin is examined to determine Master/Slave or CH1/CH2 information. This input is active high.
SEL	I	The first CA received after system reset informs the IOP via the SEL line, whether it is a Master or Slave (0/1 for Master/Slave respectively) and starts the initialization sequence. During any other CA the SEL line signifies the selection of CH1/CH2. (0/1 respectively)
DRQ1–2	I	DMA request inputs which signal the IOP that a peripheral is ready to transfer/receive data using channels 1 or 2 respectively. The signals are active high.
$\overline{RQ}/\overline{GT}$	I/O	The ReQuest GranT pin implements the communication dialogue required to arbitrate the use of the system bus (between IOP and CPU, LOCAL mode) or I/O bus when two IOPs share the same bus (REMOTE mode). The $\overline{RQ}/\overline{GT}$ signal is active low. An internal pull-up permits $\overline{RQ}/\overline{GT}$ to be left floating if not used.
SINTR1–2	O	Interrupt outputs from channels 1 and 2 respectively. The interrupts may be sent directly to the CPU or through the 8259A interrupt controller. They are used to indicate to the system the occurrence of user defined events.
EXT1–2	I	External terminate inputs for channels 1 and 2 respectively. The EXT signals will cause the termination of the current DMA transfer operation if the channel is so programmed by the channel control register. The signals are active high.
V_{CC}		+5 volt power input.
V_{SS}		Ground pins.

PRELIMINARY

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . – 65°C to + 150°C
Voltage on Any Pin with
Respect to Ground . . . . . – 0.3 to + 7V
Power Dissipation . . . . . 2.5 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

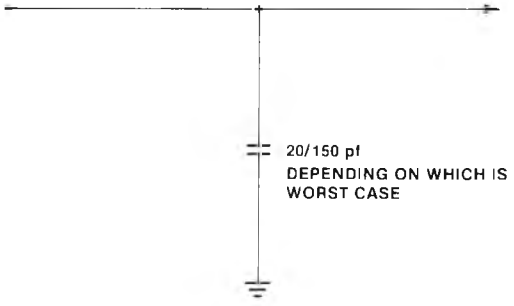
D.C. CHARACTERISTICS

8089: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

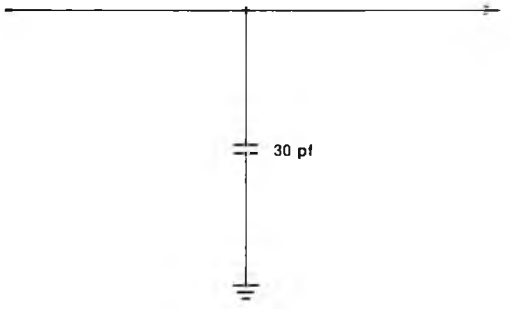
Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	– 0.5	+ 0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} + 1.0	V	
V_{OL}	Output Low Voltage		0.45	V	$I_{OL} = 2.0$ mA(2)
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\ \mu A$
I_{CC}	Power Supply Current		350	mA	$T_A = 25°C$
I_{LI}	Input Leakage Current(1)		± 10	μA	$V_{IN} = V_{CC}$
I_{LO}	Output Leakage Current		± 10	μA	$0.45V \leq V_{OUT} \leq V_{CC}$
V_{CL}	Clock Input Low Voltage	– 0.5	+ 0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC} + 1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except $AD_0 - AD_{15}$, $\overline{RQ}/\overline{GT}$)		10	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer ($AD_0 - AD_{15}$, $\overline{RQ}/\overline{GT}$)		20	pF	fc = 1 MHz

NOTES: 1. Except $\overline{RQ}/\overline{GT}$
2. Test Circuits:

ALL OUTPUTS EXCEPT: $\overline{RQ}/\overline{GT}$



$\overline{RQ}/\overline{GT}$



PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

A.C. CHARACTERISTICS

8089: $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

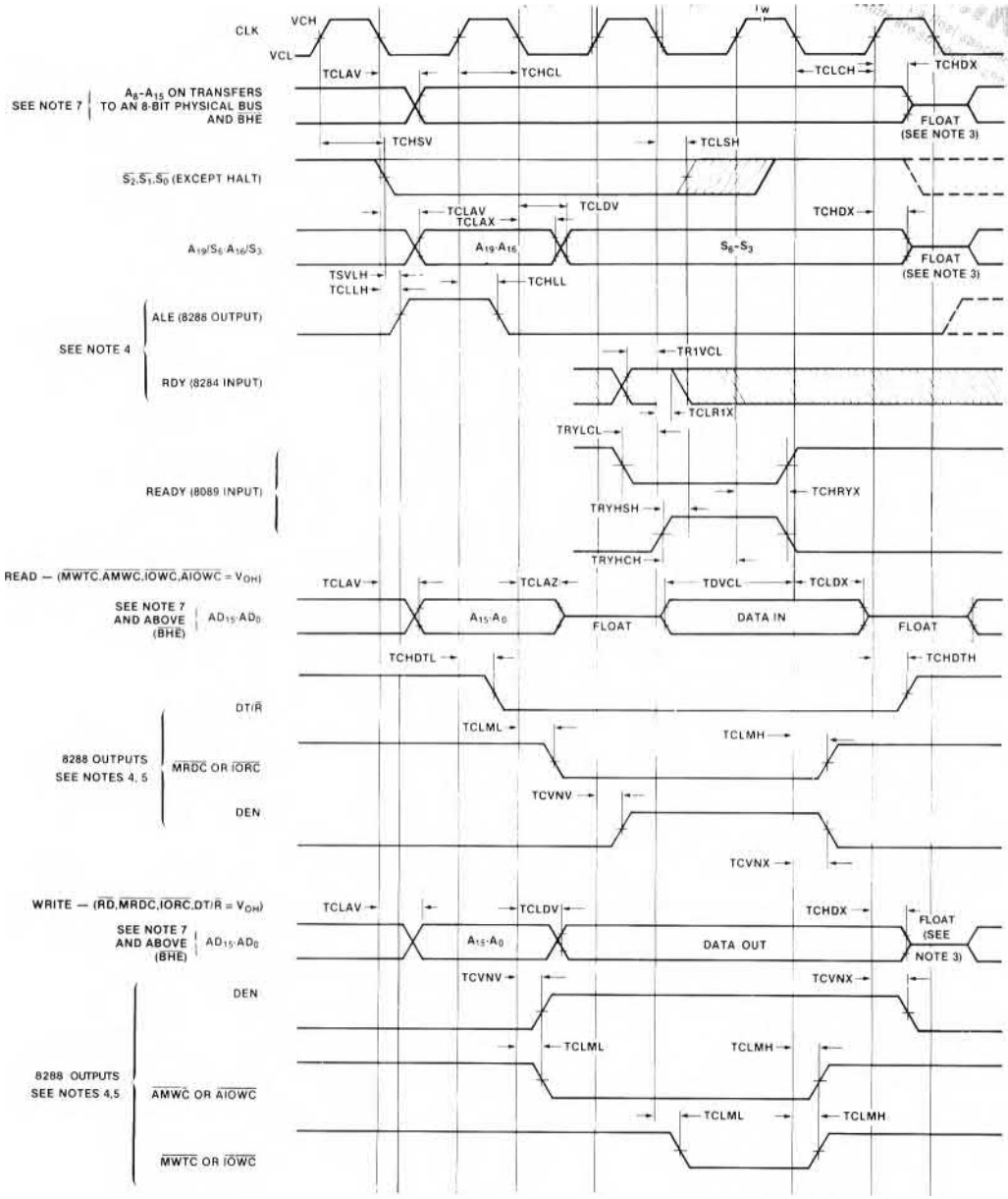
8089/8086 MAX MODE SYSTEM (USING 8288 BUS CONTROLLER) TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(⅔TCLCL) – 15		ns	
TCHCL	CLK High Time	(⅓TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		ns	
TRYHCH	READY Setup Time into 8089	(⅔TCLCL) – 15		ns	
TCHRYX	READY Hold Time into 8089	30		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	– 8		ns	
TINVCH	Setup Time Recognition (DRQ 1,2 RESET, Ext 1,2) (See Note 2)	30		ns	
TGVCH	$\overline{RQ}/\overline{GT}$ Setup Time	30		ns	
TCAHCAL	CA Width	95		ns	
TSLVCAL	SEL Setup Time	75		ns	
TCALSLX	SEL Hold Time	0		ns	
TCHGX	$\overline{RQ}$ Hold Time into 8089	40		ns	

TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	ns	$C_L = 80$ pF
TCLMH	Command Inactive Delay (See Note 1)	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110	ns	
TCHSV	Status Active Delay	10	110	ns	
TCLSH	Status Inactive Delay	10	130	ns	
TCLAV	Address Valid Delay	10	110	ns	
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15	ns	$C_L = 150$ pf
TCHLL	ALE Inactive Delay (See Note 1)		15	ns	
TCLDV	Data Valid Delay	10	110	ns	
TCHDX	Data Hold Time	10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30	ns	
TCLGL	$\overline{GT}$ Active Delay	0	85	ns	$C_L = 30$ pF
TCLGH	$\overline{GT}$ Inactive Delay		85	ns	$C_L = 30$ pF
TCLSRV	SINTR Valid Delay		150	ns	$C_L = 100$ pF

NOTES: 1. Signal at 8284 or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T3 and TW states.
4. Applies only to T2 state.

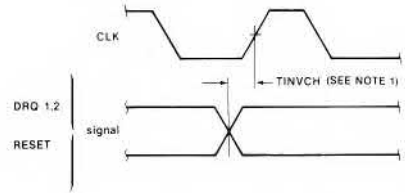


NOTES:

1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINE STATES ARE TO BE INSERTED.
3. FOLLOWING A WRITE CYCLE DATA REMAINS VALID ON THE 8089 LOCAL BUS UNTIL A LOCAL BUS MASTER DECIDES TO RUN ANOTHER BUS CYCLE. THE LOCAL BUS IS FLOATED BY THE 8089 WHEN THE 8089 ENTERS A REQUEST BUS ACKNOWLEDGE STATE.
4. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
5. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS (MRDC, MWTC, AMWC, IORC, IOWC, AIOWC, INTA, AND DEN) CAUSE THE ACTIVE HIGH 8288 CEN.
6. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
7. A_8-A_{15} ARE STABLE ON TRANSFERS TO AN 8-BIT PHYSICAL DATA BUS (i.e. no A_{16}-A_{19} DON'T FLOAT ON A READ FROM AN 8-BIT PHYSICAL BUS OR MULTIPLEX WITH DATA ON A WRITE TO AN 8-BIT PHYSICAL BUS. BHE IS STABLE (NON-MULTIPLEXED) FOR ALL TRANSFERS.

Figure 8. 8089 Bus Timing — (Using 8288)

PRELIMINARY
Device: This is not a final specification. Some parameter limits are subject to change.

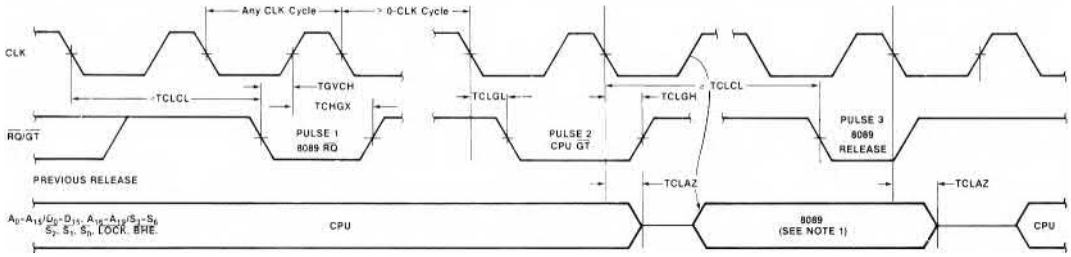


NOTES:

1. SETUP REQUIREMENTS FOR ASYNCHRONOUS SIGNALS ONLY TO GUARANTEE RECOGNITION AT NEXT CLK.
2. ALL INPUTS EXCEPT CA ARE LATCHED ON A CLK EDGE. THE CA INPUT IS NEGATIVE EDGE TRIGGERED.
3. DRQ BECOMING ACTIVE GREATER THAN 30 ns AFTER THE RISING EDGE OF CLK WILL GUARANTEE NON-RECOGNITION UNTIL THE NEXT RISING CLOCK EDGE.

Figure 9. Asynchronous Signal Recognition

Figure 10. Bus Lock Signal Timing and SINTR Timing



NOTES:

1. THE CPU MAY NOT DRIVE THE BUSES INSIDE THE REGION SHOWN WITHOUT RISKING CONTENTION.
2. IN THE REMOTE CONFIGURATION, THE 8089 IOP CAN EITHER ISSUE OR RESPOND TO RQ/GT, THUS ALLOWING THE USER TO TIE 2 8089's TOGETHER. THE PROTOCOL OF RQ/GT IN THIS CONFIGURATION CONSISTS OF ONLY ONE PULSE TO TRANSFER THE BUS.

Figure 11. Request/Grant Sequence Timing

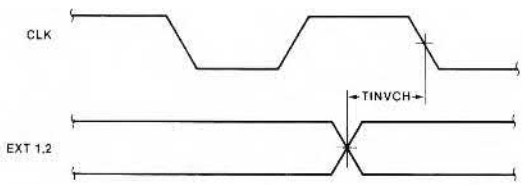


Figure 12. External Terminate Setup Timing

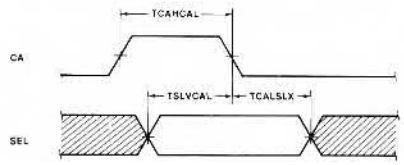


Figure 13. SEL Setup and Hold Timing

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

8089 INSTRUCTION SET SUMMARY

Data Transfers

			OPCODE 7 … 0	OPCODE 7 … 0
		POINTER INSTRUCTIONS		
LPD	P,M	Load Pointer PPP from Addressed Location	P P P 0 0 A A 1	1 0 0 0 1 0 M M
LPDI	P,I	Load Pointer PPP Immediate 4 Bytes	P P P 1 0 0 0 1	0 0 0 0 1 0 0 0
MOVP	M,P	Store Contents of Pointer PPP in Addressed Location	P P P 0 0 A A 1	1 0 0 1 1 0 M M
MOVP	P,M	Restore Pointer	P P P 0 0 A A 1	1 0 0 0 1 1 M M
		MOVE DATA		
MOV	M,M	Move from Source to Destination — Source—	0 0 0 0 0 A A W	1 0 0 1 0 0 M M
		Destination—	0 0 0 0 0 A A W	1 1 0 0 1 1 M M
MOV	R,M	Load Register RRR from Addressed Location	R R R 0 0 A A W	1 0 0 0 0 0 M M
MOV	M,R	Store Contents of Register RRR in Addressed Location	R R R 0 0 A A W	1 0 0 0 0 1 M M
MOVI	R	Load Register RRR Immediate (Byte) Sign Extend	R R R wb 0 0 W	0 0 1 1 0 0 0 0
MOVI	M	Move Immediate to Addressed Location	0 0 0 wb A A W	0 1 0 0 1 1 M M

Control Transfer

		CALLS		
*CALL		Call Unconditional	1 0 0 wb A A W	1 0 0 1 1 1 M M
		JUMP		
JMP		Unconditional	1 0 0 wb 0 0 W	0 0 1 0 0 0 0 0
JZ	M	Jump on Zero Memory	0 0 0 wb A A W	1 1 1 0 0 1 M M
JZ	R	Jump on Zero Register	R R R wb 0 0 0	0 1 0 0 0 1 0 0
JNZ	M	Jump on Non-Zero Memory	0 0 0 wb A A W	1 1 1 0 0 0 M M
JNZ	R	Jump on Non-Zero Register	R R R wb 0 0 0	0 1 0 0 0 0 0 0
JBT		Test Bit and Jump if True	B B B wb A A 0	1 0 1 1 1 1 M M
JNBT		Test Bit and Jump if Not True	B B B wb A A 0	1 0 1 1 1 0 M M
JMCE		Mask/Compare and Jump on Equal	0 0 0 wb A A 0	1 0 1 1 0 0 M M
JMCNE		Mask/Compare and Jump on Non-Equal	0 0 0 wb A A 0	1 0 1 1 0 1 M M

Arithmetic and Logic Instructions

		INCREMENT, DECREMENT		
INC	M	Increment Addressed Location	0 0 0 0 0 A A W	1 1 1 0 1 0 M M
INC	R	Increment Register	R R R 0 0 0 0 0	0 0 1 1 1 0 0 0
DEC	M	Decrement Addressed Location	0 0 0 0 0 A A W	1 1 1 0 1 1 M M
DEC	R	Decrement Register	R R R 0 0 0 0 0	0 0 1 1 1 1 0 0
		ADD		
ADDI	M,I	ADD Immediate to Memory	0 0 0 wb A A W	1 1 0 0 0 0 M M
ADDI	R,I	ADD Immediate to Register	R R R wb 0 0 W	0 0 1 0 0 0 0 0
ADD	M,R	ADD Register to Memory	R R R 0 0 A A W	1 1 0 1 0 0 M M
ADD	R,M	ADD Memory to Register	R R R 0 0 A A W	1 0 1 0 0 0 M M
		AND		
ANDI	M,I	AND Memory with Immediate	0 0 0 wb A A W	1 1 0 0 1 0 M M
ANDI	R,I	AND Register with Immediate	R R R wb 0 0 W	0 0 1 0 1 0 0 0
AND	M,R	AND Memory with Register	R R R 0 0 A A W	1 1 0 1 1 0 M M
AND	R,M	AND Register with Memory	R R R 0 0 A A W	1 0 1 0 1 0 M M
		OR		
ORI	M,I	OR Memory with Immediate	0 0 0 wb A A 0	1 1 0 0 0 1 M M
ORI	R,I	OR Register with Immediate	R R R wb A A 0	0 0 1 0 0 1 0 0
OR	M,R	OR Memory with Register	R R R 0 0 A A W	1 1 0 1 0 1 M M
OR	R,M	OR Register with Memory	R R R 0 0 A A W	1 0 1 0 0 1 M M

PRELIMINARY
Notice: This is not a final specification. Parametric limits are subject to change.

Arithmetic and Logic Instructions (cont.)

			OPCODE 7	0	7	0
		NOT				
NOT	R	Complement Register	R R R 0	0 0 0 0	0 0 1 0	1 1 0 0
NOT	M	Complement Memory	0 0 0 0	0 A A W	1 1 0 1	1 1 M M
NOT	R,M	Complement Memory, Place in Register	R R R 0	0 A A W	1 0 1 0	1 1 M M

Bit Manipulation and Test Instructions

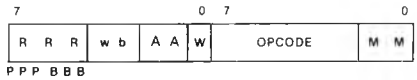
		BIT MANIPULATION				
SET		Set the Selected Bit	B B B 0	0 A A 0	1 1 1 1	0 1 M M
CLR		Clear the Selected Bit	B B B 0	0 A A 0	1 1 1 1	1 0 M M
		TEST				
TSL		Test and Set Lock	0 0 0 1	1 A A 0	1 0 0 1	0 1 M M

Control

HLT		Halt Channel Execution	0 0 1 0	0 0 0 0	0 1 0 0	1 0 0 0
SINTR		Set Interrupt Service Flip Flop	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0
NOP		No Operation	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
XFER		Enter DMA Transfer	0 1 1 0	0 0 0 0	0 0 0 0	0 0 0 0
WID		Set Source, Destination Bus Width; S,D 0 = 8, 1 = 16	1 S D 0	0 0 0 0	0 0 0 0	0 0 0 0

NOTES:

*II field in call instruction can be 00, 01, 10 only.
**OPCODE is second byte fetched.
All instructions consist of at least 2 bytes, while some instructions may use up to 3 additional bytes to specify literals and displacement data. The definition of the various fields within each instruction is given below:



MM Base Pointer Select

00 GA
01 GB
10 GC
11 PP

RRR Register Field

The RRR field specifies a 16-bit register to be used in the instruction. If GA, GB, GC or TP, are referenced by the RRR field, the upper 4 bits of the registers are loaded with the sign bit (Bit 15). PPP registers are used as 20-bit address pointers.

RRR

000 r0 GA
001 r1 GB
010 r2 GC
011 r3 BC ; byte count
100 r4 TP ; task block
101 r5 IX ; index register
110 r6 CC ; channel control (mode)
111 r7 MC ; mask/compare

ppp

000 p0 GA ;
001 p1 GB ;
010 p2 GC ;
100 p4 TP ; task block pointer

BBB Bit Select Field

The bit select field replaces the RRR field in bit manipulation instructions and is used to select a bit to be operated on by those instructions. Bit 0 is the least significant bit.

wb

01 1 byte literal or 1 byte displacement
10 2 byte (word) literal or 2 byte (word) displacement

AA Field

00 The selected pointer contains the operand address.
01 The operand address is formed by adding an 8-bit, unsigned, offset contained in the instruction to the selected pointer. The contents of the pointer are unchanged.
10 The operand address is formed by adding the contents of the Index register to the selected pointer. Both registers remain unchanged.
11 Same as 10 except the Index register is post auto-incremented (by 1 for 8-bit transfer, by 2 for 16-bit transfer).

W Width Field

0 The selected operand is 1 byte long.
1 The selected operand is 2 bytes long.

MNEMONICS © 1979 INTEL CORP.

PRELIMINARY
Notice: This is not a final specification.
Parametric limits are subject to change.

Additional Bytes

OFFSET : 8-bit unsigned offset.
SDISP : 8/16-bit signed displacement.
LITERAL : 8/16-bit literal.

The order in which the above optional bytes appear in IOP instructions is given below:

OFFSET	LITERAL	SDISP

Offsets are treated as unsigned numbers. Literals and displacements are sign extended (2's complement).

intel®

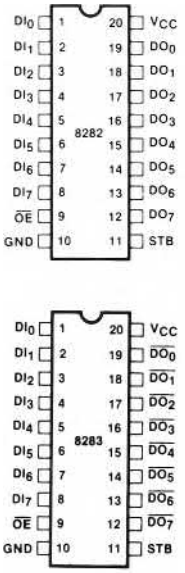
PRELIMINARY

8282/8283
OCTAL LATCH

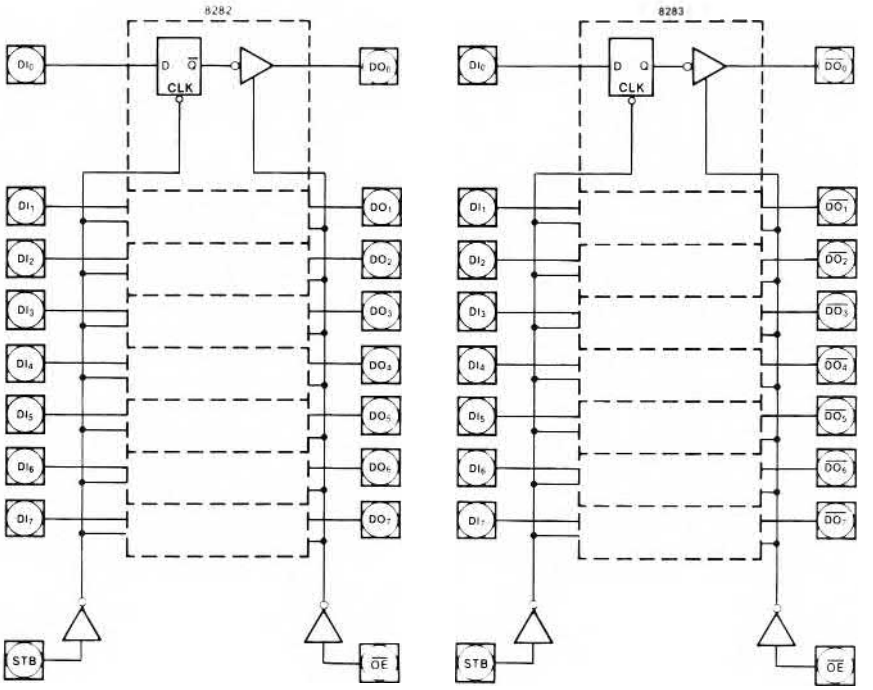
- **Fully Parallel 8-Bit Data Register and Buffer**
- **Transparent during Active Strobe**
- **Supports 8080, 8085, 8048, and 8086 Systems**
- **High Output Drive Capability for Driving System Data Bus**
- **3-State Outputs**
- **20-Pin Package with 0.3" Center**
- **No Output Low Noise when Entering or Leaving High Impedance State**

The 8282 and 8283 are 8-bit bipolar latches with 3-state output buffers. They can be used to implement latches, buffers, or multiplexers. The 8283 inverts the input data at its outputs while the 8282 does not. Thus, all of the principal peripheral and input/output functions of a microcomputer system can be implemented with these devices.

PIN CONFIGURATIONS



LOGIC DIAGRAMS



PIN NAMES

DI_0-DI_7	DATA IN
DO_0-DO_7	DATA OUT
$\overline{OE}$	OUTPUT ENABLE
STB	STROBE

PRELIMINARY

PIN DEFINITIONS

Pin	Description
STB	STROBE (Input). STB is an input control pulse used to strobe data at the data input pins (A_0-A_7) into the data latches. This signal is active HIGH to admit input data. The data is latched at the HIGH to LOW transition of STB.
$\overline{OE}$	OUTPUT ENABLE (Input). $\overline{OE}$ is an input control signal which when active LOW enables the contents of the data latches onto the data output pin (B_0-B_7). OE being inactive HIGH forces the output buffers to their high impedance state.
DI_0-DI_7	DATA INPUT PINS (Input). Data presented at these pins satisfying setup time requirements when STB is strobed and latched into the data input latches.
DO_0-DO_7 (8282) $\overline{DO_0}$-$\overline{DO_7}$ (8283)	DATA OUTPUT PINS (Output). When $\overline{OE}$ is true, the data in the data latches is presented as inverted (8283) or non-inverted (8282) data onto the data output pins.

OPERATIONAL DESCRIPTION

The 8282 and 8283 octal latches are 8-bit latches with 3-state output buffers. Data having satisfied the setup time requirements is latched into the data latches by strobing the STB line HIGH to LOW. Holding the STB line in its active HIGH state makes the latches appear transparent. Data is presented to the data output pins by activating the $\overline{OE}$ input line. When $\overline{OE}$ is inactive HIGH the output buffers are in their high impedance state. Enabling or disabling the output buffers will not cause negative-going transients to appear on the data output bus.

D.C. AND OPERATING CHARACTERISTICS

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . – 65°C to + 150°C
All Output and Supply Voltages . . . . . – 0.5V to + 7V
All Input Voltages . . . . . – 1.0V to + 5.5V
Power Dissipation . . . . . 1 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS FOR 8282/8283

Conditions: $V_{CC} = 5V \pm 5\%$, $T_A = 0°C$ to $70°C$

Symbol	Parameter	Min	Max	Units	Test Conditions
V_C	Input Clamp Voltage		– 1	V	I_C = – 5 mA
I_{CC}	Power Supply Current		160	mA	
I_F	Forward Input Current		– 0.2	mA	V_F = 0.45V
I_R	Reverse Input Current		50	μA	V_R = 5.25V
V_{OL}	Output Low Voltage		0.50	V	I_{OL} = 32 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = – 5 mA
I_{OFF}	Output Off Current		± 50	μA	V_{OFF} = 0.45 to 5.25V
V_{IL}	Input Low Voltage		0.8	V	V_{CC} = 5.0V See Note 1
V_{IH}	Input High Voltage	2.0		V	V_{CC} = 5.0V See Note 1
C_{IN}	Input Capacitance		12	pF	F = 1 MHz V_{BIAS} = 2.5V, V_{CC} = 5V T_A = 25°C

Notes: 1. Output Loading I_{OL} = 32 mA, I_{OH} = – 5 mA, C_L = 300 pF

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

A.C. CHARACTERISTICS FOR 8282/8283

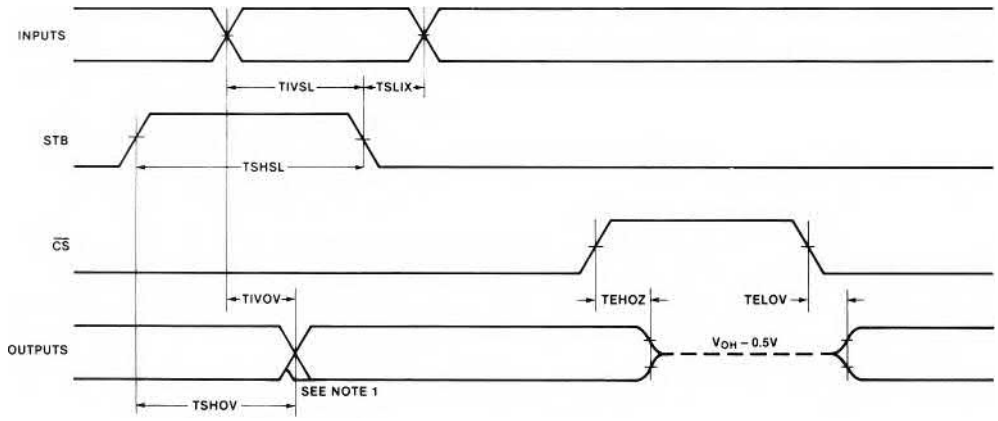
Conditions: $V_{CC} = 5V \pm 5\%$, $T_A = 0°C$ to $70°C$

Loading: Outputs — I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF

Symbol	Parameter	Min	Max	Units	Test Conditions
TIVOV	Input to Output Delay				(See Note 1)
	—Inverting		25	ns	
	—Non-Inverting		35	ns	
TSHOV	STB to Output Delay				
	—Inverting		45	ns	
	—Non-Inverting		55	ns	
TEHOZ	Output Disable Time		25	ns	
TELOV	Output Enable Time	10	50	ns	
TIVSL	Input to STB Setup Time	0		ns	
TSLIX	Input to STB Hold Time	25		ns	
TSHSL	STB High Time	15		ns	

NOTE: 1. See waveforms and test load circuit on following page.

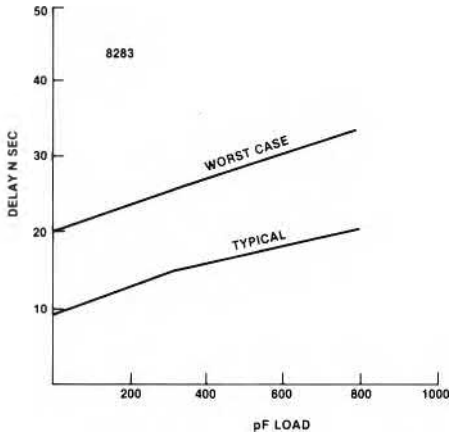
8282/8283 TIMING

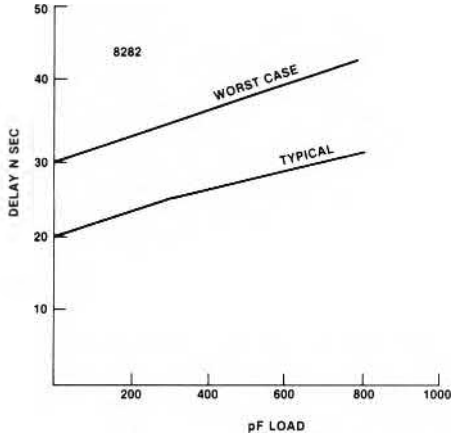


NOTE: 1. 8283 ONLY — OUTPUT MAY BE MOMENTARILY INVALID FOLLOWING THE HIGH GOING STB TRANSITION.
2. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED

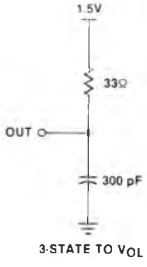
PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

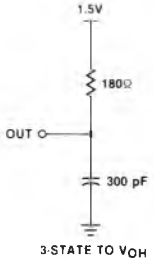
OUTPUT DELAY VS. CAPACITANCE

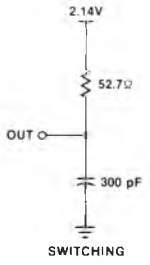




OUTPUT TEST LOAD CIRCUITS







intel®

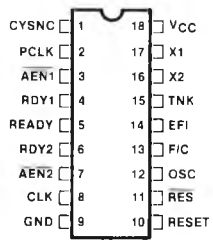
PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

8284
CLOCK GENERATOR AND DRIVER FOR 8086, 8088, 8089 PROCESSORS

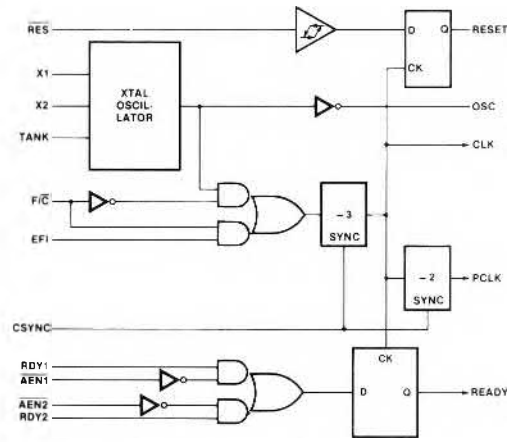
- Generates the System Clock for the 8086, 8088 and 8089
- Uses a Crystal or a TTL Signal for Frequency Source
- Single +5V Power Supply
- 18-Pin Package
- Generates System Reset Output from Schmitt Trigger Input
- Provides Local Ready and MULTIBUS™ Ready Synchronization
- Capable of Clock Synchronization with other 8284's

The 8284 is a bipolar clock generator/driver designed to provide clock signals for the 8086, 8088 & 8089 and peripherals. It also contains READY logic for operation with two MULTIBUS™ systems and provides the processors required READY synchronization and timing. Reset logic with hysteresis and synchronization is also provided.

8284 PIN CONFIGURATION



8284 BLOCK DIAGRAM



8284 PIN NAMES

X1, X2	CONNECTIONS FOR CRYSTAL
TANK	USED WITH OVERTONE CRYSTAL
F/C	CLOCK SOURCE SELECT
EFI	EXTERNAL CLOCK INPUT
CSYNC	CLOCK SYNCHRONIZATION INPUT
RDY1, RDY2	READY SIGNAL FROM TWO MULTIBUS™ SYSTEMS
AEN1, AEN2	ADDRESS ENABLED QUALIFIERS FOR RDY1,2
RES	RESET INPUT
RESET	SYNCHRONIZED RESET OUTPUT
OSC	OSCILLATOR OUTPUT
CLK	MOS CLOCK FOR THE PROCESSOR
PCLK	TTL CLOCK FOR PERIPHERALS
READY	SYNCHRONIZED READY OUTPUT
V_{CC}	+5 VOLTS
GND	0 VOLTS

PRELIMINARY

PIN DEFINITIONS

Pin	I/O	Definition
$\overline{AEN1}$, $\overline{AEN2}$	I	ADDRESS ENABLE. $\overline{AEN}$ is an active LOW signal. $\overline{AEN}$ serves to qualify its respective Bus Ready Signal (RDY1 or RDY2). $\overline{AEN1}$ validates RDY1 while $\overline{AEN2}$ validates RDY2. Two AEN signal inputs are useful in system configurations which permit the processor to access two Multi-Master System Busses. In non Multi-Master configurations the $\overline{AEN}$ signal inputs are tied true (LOW).
RDY1, RDY2	I	BUS READY (Transfer Complete). RDY is an active HIGH signal which is an indication from a device located on the system data bus that data has been received, or is available. RDY1 is qualified by $\overline{AEN1}$ while RDY2 is qualified by $\overline{AEN2}$.
READY	O	READY. READY is an active HIGH signal which is the synchronized RDY signal input. Since RDY occurs asynchronously with respect to the clock (CLK) it may be necessary for them to be synchronized before being presented to the 8284. READY is cleared after the guaranteed hold time to the processor has been met.
X1, X2, TNK	I	CRYSTAL IN. X1 and X2 are the pins to which a crystal is attached with TNK (TANK) serving as the overtone input. The crystal frequency is 3 times the desired processor clock frequency.
F/$\overline{C}$	I	FREQUENCY/CRYSTAL SELECT. F/$\overline{C}$ is a strapping option. When strapped LOW, F/$\overline{C}$ permits the processor's clock to be generated by the crystal. When F/$\overline{C}$ is strapped HIGH, CLK is generated from the EFI input.
EFI	I	EXTERNAL FREQUENCY IN. When F/$\overline{C}$ is strapped HIGH, CLK is generated from the input frequency appearing on this pin. The input signal is a square wave 3 times the frequency of the desired CLK output.
CLK	O	PROCESSOR CLOCK. CLK is the clock output used by the processor and all devices which directly connect to the processor's local bus (i.e., the bipolar support chips and other MOS devices). CLK has an output frequency which is 1/3 of the crystal or EFI input frequency and a 1/3 duty cycle. An output HIGH of 4.5 volts (V_{CC} = 5V) is provided on this pin to drive MOS devices.
PCLK	O	PERIPHERAL CLOCK. PCLK is a TTL level peripheral clock signal whose output frequency is 1/2 that of CLK and has a 50% duty cycle.
OSC	O	OSCILLATOR OUTPUT. OSC is the TTL level output of the internal oscillator circuitry. Its frequency is equal to that of the crystal.
$\overline{RES}$	I	RESET IN. $\overline{RES}$ is an active LOW signal which is used to generate RESET. The 8284 provides a Schmitt trigger input so that an RC connection can be used to establish the power-up reset of proper duration.
RESET	O	RESET. Reset is an active HIGH signal which is used to reset the 8086 family processors. Its timing characteristics are determined by $\overline{RES}$.
CSYNC	I	CLOCK SYNCHRONIZATION. CSYNC is an active HIGH signal which allows multiple 8284's to be synchronized to provide clocks that are in phase. When CSYNC is HIGH the internal counters are reset. When CSYNC goes LOW the internal counters are allowed to resume counting. CSYNC needs to be externally synchronized to EFI. When using the internal oscillator CSYNC should be hardwired to ground.
GND		Ground
V_{CC}		+ 5V supply

FUNCTIONAL DESCRIPTION

GENERAL

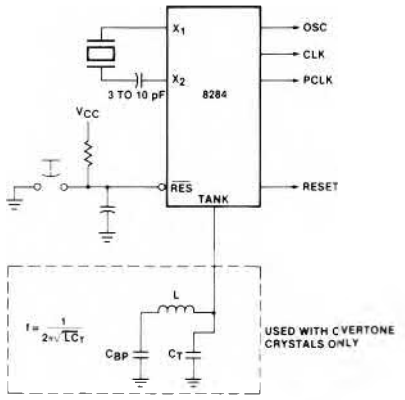
The 8284 is a single chip clock generator/driver for the 8086, 8088 & 8089 processors. The chip contains a crystal controlled oscillator, a "divide by three" counter, complete MULTIBUS™ "Ready" synchronization and reset logic.

OSCILLATOR

The oscillator circuit of the 8284 is designed primarily for use with an external series resonant, fundamental mode, crystal from which the basic operating frequency is derived. However, overtone mode crystals can be used with a tank circuit as shown in Figure 1.

The crystal frequency should be selected at three times the required CPU clock. X_1 and X_2 are the two crystal input crystal connections.

The output of the oscillator is buffered and brought out on OSC so that other system timing signals can be derived from this stable, crystal-controlled source.



The tank input to the oscillator allows the use of overtone mode crystals. The tank circuit shunts the crystal's fundamental and high overtone frequencies and allows the third harmonic to oscillate. The external LC network is connected to the TANK input and is AC coupled to ground.

Figure 1

CLOCK GENERATOR

The clock generator consists of a synchronous divide-by-three counter with a special clear input that inhibits the counting. This clear input (CSYNC) allows the output clock to be synchronized with an external event (such as another 8284 clock). It is necessary to synchronize the CSYNC input to the EFI clock external to the 8284. This is accomplished with two Schottky flip-flops. (See Figure 2.) The counter output is a 33% duty cycle clock at one-third the input frequency.

The F/$\overline{C}$ input is a strapping pin that selects either the crystal oscillator or the EFI input as the clock for the ÷ 3 counter. If the EFI input is selected as the clock source, the oscillator section can be used independently for another clock source. Output is taken from OSC.

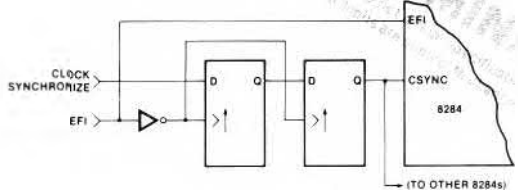


Figure 2. CSYNC Synchronization

CLOCK OUTPUTS

The CLK output is a 33% duty cycle MOS clock driver designed to drive the 8086 processor directly. PCLK is a TTL level peripheral clock signal whose output frequency is 1/2 that of CLK. PCLK has a 50% duty cycle.

RESET LOGIC

The reset logic provides a Schmitt trigger input ($\overline{RES}$) and a synchronizing flip-flop to generate the reset timing. The reset signal is synchronized to the falling edge of CLK. A simple RC network can be used to provide power on reset by utilizing this function of the 8284.

READY SYNCHRONIZATION

Two READY inputs (RDY1, RDY2) are provided to accomomodate two Multi-Master system busses. Each input has a qualifier ($\overline{AEN1}$ and $\overline{AEN2}$, respectively). The $\overline{AEN}$ signals validate their respective RDY signals. If a Multi-Master system is not being used the $\overline{AEN}$ pin should be tied LOW.

Synchronization is required for all asynchronous active going edges of either RDY input to guarantee that the RDY setup and hold times are met. Inactive going edges of RDY in normally ready systems do not require synchronization but must satisfy RDY setup and hold as a matter of proper system design. Synchronization may be accomplished by inserting a D flip flop between an asynchronous RDY source and the 8284 and clocking the flip flop on the rising edge of CLK. The 8284 READY logic guarantees the required 8086 READY hold time before clearing the READY signal.

D.C. AND OPERATING CHARACTERISTICS

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . − 65°C to + 150°C
All Output and Supply Voltages . . . . . − 0.5V to + 7V
All Input Voltages . . . . . − 1.0V to + 5.5V
Power Dissipation . . . . . 1 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

PRELIMINARY

D.C. CHARACTERISTICS FOR 8284

Conditions: $T_A = 0°C$ to $70°C$; $V_{CC} = 5V \pm 10\%$

Symbol	Parameter	Min	Max	Units	Test Conditions
I_F	Forward Input Current		−0.5	mA	$V_F = 0.45V$
I_R	Reverse Input Current		50	μA	$V_R = 5.25V$
V_C	Input Forward Clamp Voltage		−1.0	V	$I_C = -5$ mA
I_{CC}	Power Supply Current		140	mA	
V_{IL}	Input LOW Voltage		0.8	V	$V_{CC} = 5.0V$
V_{IH}	Input HIGH Voltage	2.0		V	$V_{CC} = 5.0V$
V_{IH_R}	Reset Input HIGH Voltage	2.6		V	$V_{CC} = 5.0V$
V_{OL}	Output LOW Voltage		0.45	V	5 mA
V_{OH}	Output HIGH Voltage CLK Other Outputs	4 2.4		V V	−1 mA −1 mA
V_{IH_R}-V_{IL_R}	$\overline{RES}$ Input Hysteresis	0.25		V	$V_{CC} = 5.0V$

A.C. CHARACTERISTICS FOR 8284

Conditions: $T_A = 0°C$ to $70°C$; $V_{CC} = 5V \pm 10\%$

TIMING REQUIREMENTS

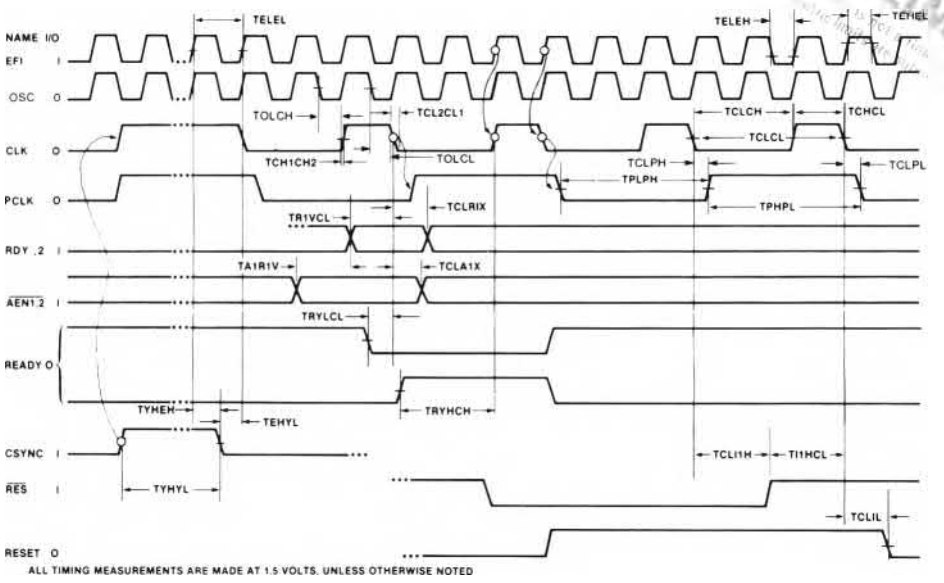
Symbol	Parameter	Min	Max	Units	Test Conditions
TEHEL	External Frequency High Time	13		ns	90% - 90% V_{IN}
TELEH	External Frequency Low Time	13		ns	10% - 10% V_{IN}
TELEL	EFI Period	TEHEL + TELEH + δ		ns	(Note 1)
	XTAL Frequency	12	25	MHz	
TR1VCL	RDY1, RDY2 Set-Up to CLK	35		ns	
TCLR1X	RDY1, RDY2 Hold to CLK	0		ns	
TA1VR1V	$\overline{AEN1}$, $\overline{AEN2}$ Set-Up to RDY1, RDY2	15		ns	
TCLA1X	$\overline{AEN1}$, $\overline{AEN2}$ Hold to CLK	0		ns	
TYHEH	CSYNC Set-Up to EFI	20		ns	
TEHYL	CSYNC Hold to EFI	20		ns	
TYHYL	CSYNC Width	2 TELEL		ns	
TI1HCL	$\overline{RES}$ Set-Up to CLK	65		ns	(Note 2)
TCLI1H	$\overline{RES}$ Hold to CLK	20		ns	(Note 2)

TIMING RESPONSES

Symbol	Parameter	Min	Max	Units	Test Conditions
TCLCL	CLK Cycle Period	125		ns	
TCHCL	CLK High Time	(⅓TCLCL) + 2.0		ns	Fig. 3 & Fig. 4
TCLCH	CLK Low Time	(⅔TCLCL) − 15.0		ns	Fig. 3 & Fig. 4
TCH1CH2 TCL2CL1	CLK Rise or Fall Time		10	ns	1.0V to 3.5V
TPHPL	PCLK High Time	TCLCL − 20		ns	
TPLPH	PCLK Low Time	TCLCL − 20		ns	
TRYLCL	Ready Inactive to CLK (See Note 4)	−8		ns	Fig. 5 & Fig. 6
TRYHCH	Ready Active to CLK (See Note 3)	(⅔TCLCL)−15.0		ns	Fig. 5 & Fig. 6
TCLIL	CLK to Reset Delay	40		ns	
TCLPH	CLK to PCLK High Delay		22	ns	
TCLPL	CLK to PCLK Low Delay		22	ns	
TOLCH	OSC to CLK High Delay	−5	12	ns	
TOLCL	OSC to CLK Low Delay	2	20	ns	

Notes: 1. δ = EFI rise (5 ns max) + EFI fall (5 ns max).
2. Set up and hold only necessary to guarantee recognition at next clock.
3. Applies only to T3 and TW states.
4. Applies only to T2 states.

PRELIMINARY



A.C. TEST CIRCUITS

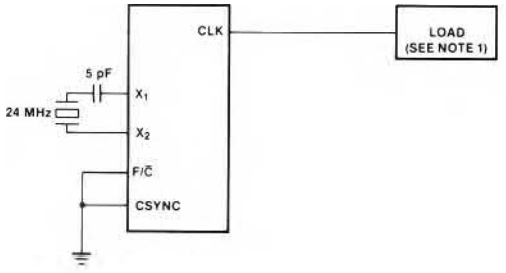


Figure 3. Clock High and Low Time

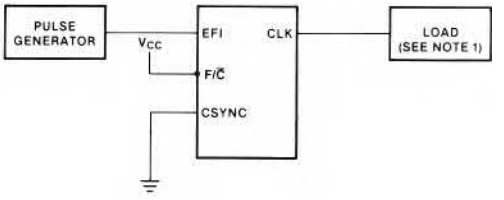


Figure 4. Clock High and Low Time

PRELIMINARY
Notice. This is not a final specification. Some parametric limits are subject to change.

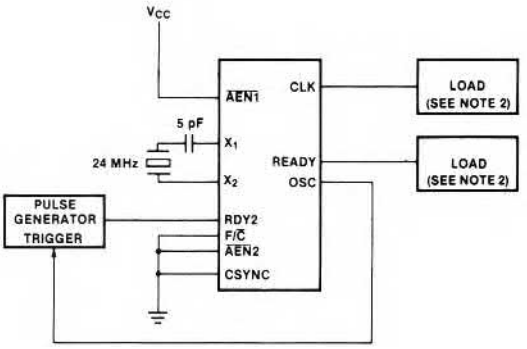


Figure 5. Ready to Clock

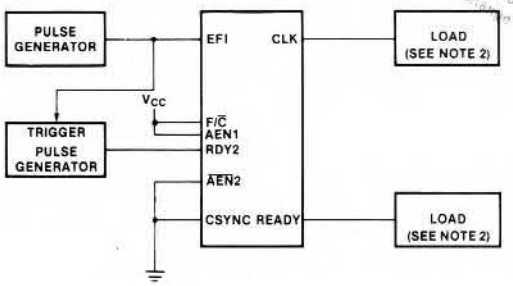
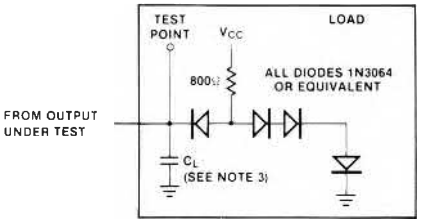


Figure 6. Ready to Clock



NOTES: 1. $C_L = 100$ pF
2. $C_L = 30$ pF
3. C_L INCLUDES PROBE AND JIG CAPACITANGE

intel®

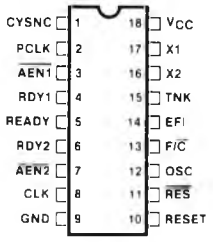
PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

M8284
CLOCK GENERATOR AND DRIVER FOR 8086, 8088, 8089 PROCESSORS

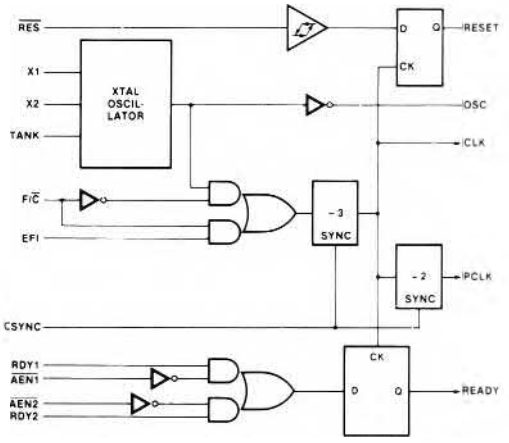
- **Generates the System Clock for the 8086, 8088 and 8089**
- **Uses a Crystal or a TTL Signal for Frequency Source**
- **Single +5V Power Supply**
- **18-Pin Package**
- **Generates System Reset Output from Schmitt Trigger Input**
- **Provides Local Ready and MULTIBUS™ Ready Synchronization**
- **Capable of Clock Synchronization with other 8284's**
- **Full Military Temperature Range −55° to +125°C**

The M8284 is a bipolar clock generator/driver designed to provide clock signals for the 8086, 8088 & 8089 and peripherals. It also contains READY logic for operation with two MULTIBUS™ systems and provides the processors required READY synchronization and timing. Reset logic with hysteresis and synchronization is also provided.

M8284 PIN CONFIGURATION



M8284 BLOCK DIAGRAM



M8284 PIN NAMES

X1, X2	CONNECTIONS FOR CRYSTAL
TANK	USED WITH OVERTONE CRYSTAL
F/C	CLOCK SOURCE SELECT
EFI	EXTERNAL CLOCK INPUT
CSYNC	CLOCK SYNCHRONIZATION INPUT
RDY1, RDY2	READY SIGNAL FROM TWO MULTIBUS™ SYSTEMS
$\overline{AEN1}$, $\overline{AEN2}$	ADDRESS ENABLED QUALIFIERS FOR RDY1,2
$\overline{RES}$	RESET INPUT
RESET	SYNCHRONIZED RESET OUTPUT
OSC	OSCILLATOR OUTPUT
CLK	MOS CLOCK FOR THE PROCESSOR
PCLK	TTL CLOCK FOR PERIPHERALS
READY	SYNCHRONIZED READY OUTPUT
V_{CC}	+5 VOLTS
GND	0 VOLTS

intel®

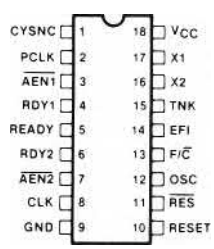
PRELIMINARY

I8284
CLOCK GENERATOR AND DRIVER FOR 8086, 8088, 8089 PROCESSORS

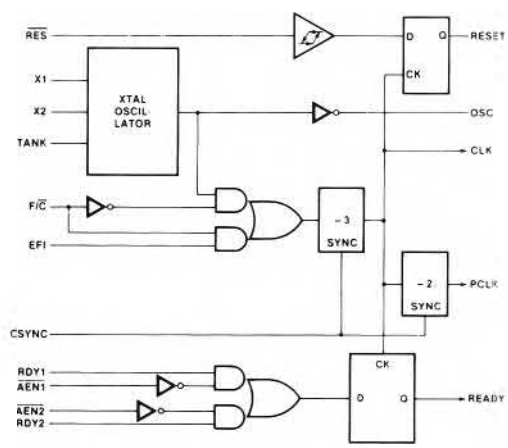
- **Generates the System Clock for the 8086, 8088 and 8089**
- **Uses a Crystal or a TTL Signal for Frequency Source**
- **Single +5V Power Supply**
- **18-Pin Package**
- **Generates System Reset Output from Schmitt Trigger Input**
- **Provides Local Ready and MULTIBUS™ Ready Synchronization**
- **Capable of Clock Synchronization with other 8284's**
- **Industrial Temperature Range −40° to +85°C**

The I8284 is a bipolar clock generator/driver designed to provide clock signals for the 8086, 8088 & 8089 and peripherals. It also contains READY logic for operation with two MULTIBUS™ systems and provides the processors required READY synchronization and timing. Reset logic with hysteresis and synchronization is also provided.

I8284 PIN CONFIGURATION



I8284 BLOCK DIAGRAM



I8284 PIN NAMES

X1, X2	CONNECTIONS FOR CRYSTAL
TANK	USED WITH OVERTONE CRYSTAL
F/C	CLOCK SOURCE SELECT
EFI	EXTERNAL CLOCK INPUT
CSYNC	CLOCK SYNCHRONIZATION INPUT
RDY1, RDY2	READY SIGNAL FROM TWO MULTIBUS™ SYSTEMS
AEN1, AEN2	ADDRESS ENABLED QUALIFIERS FOR RDY1,2
RES	RESET INPUT
RESET	SYNCHRONIZED RESET OUTPUT
OSC	OSCILLATOR OUTPUT
CLK	MOS CLOCK FOR THE PROCESSOR
PCLK	TTL CLOCK FOR PERIPHERALS
READY	SYNCHRONIZED READY OUTPUT
VCC	+5 VOLTS
GND	0 VOLTS

intel®

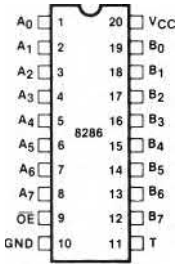
PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

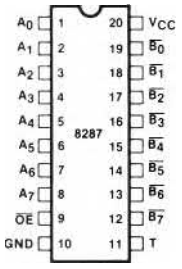
8286/8287
OCTAL BUS TRANSCEIVER

- **Data Bus Buffer Driver for MCS-86™, MCS-80™, MCS-85™, and MCS-48™ Families**
- **High Output Drive Capability for Driving System Data Bus**
- **Fully Parallel 8-Bit Transceivers**
- **3-State Outputs**
- **20-Pin Package with 0.3" Center**
- **No Output Low Noise when Entering or Leaving High Impedance State**

The 8286 and 8287 are 8-bit bipolar transceivers with 3-state outputs. The 8287 inverts the input data at its outputs while the 8286 does not. Thus, a wide variety of applications for buffering in microcomputer systems can be met.

PIN CONFIGURATIONS

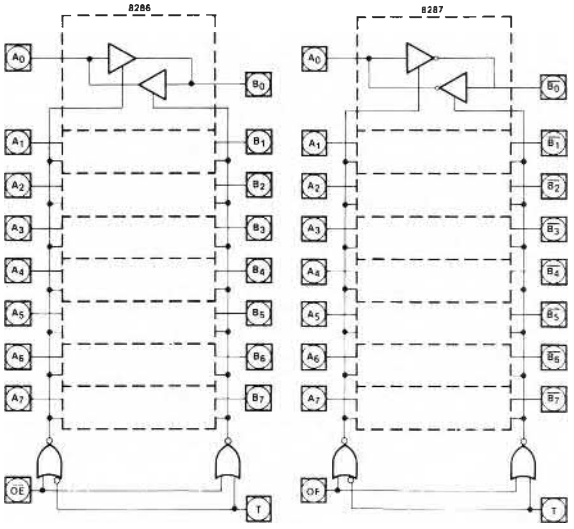




PIN NAMES

A_0-A_7	LOCAL BUS DATA
B_0-B_7	SYSTEM BUS DATA
$\overline{OE}$	OUTPUT ENABLE
T	TRANSMIT

LOGIC DIAGRAMS



PRELIMINARY

PIN DEFINITIONS

Pin	Description
T	TRANSMIT (Input). T is an input control signal used to control the direction of the transceivers. When HIGH, it configures the transceiver's B_0-B_7 as outputs with A_0-A_7 as inputs. T LOW configures A_0-A_7 as the outputs with B_0-B_7 serving as the inputs.
$\overline{OE}$	OUTPUT ENABLE (Input). $\overline{OE}$ is an input control signal used to enable the appropriate output driver (as selected by T) onto its respective bus. This signal is active LOW.
A_0-A_7	LOCAL BUS DATA PINS (Input/Output). These pins serve to either present data to or accept data from the processor's local bus depending upon the state of the T pin.
B_0-B_7 (8286) $\overline{B_0}$-$\overline{B_7}$ (8287)	SYSTEM BUS DATA PINS (Input/Output). These pins serve to either present data to or accept data from the system bus depending upon the state of the T pin.

OPERATIONAL DESCRIPTION

The 8286 and 8287 transceivers are 8-bit transceivers with high impedance outputs. With T active HIGH and $\overline{OE}$ active LOW, data at the A_0-A_7 pins is driven onto the B_0-B_7 pins. With T inactive LOW and $\overline{OE}$ active LOW, data at the B_0-B_7 pins is driven onto the A_0-A_7 pins. No output low glitching will occur whenever the transceivers are entering or leaving the high impedance state.

D.C. AND OPERATING CHARACTERISTICS

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . −65°C to +150°C
All Output and Supply Voltages . . . . . −0.5V to +7V
All Input Voltages . . . . . −1.0V to +5.5V
Power Dissipation . . . . . 1 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS FOR 8286/8287

Conditions: V_{CC} = 5V ±5%, T_A = 0°C to 70°C

Symbol	Parameter	Min	Max	Units	Test Conditions
V_C	Input Clamp Voltage		-1	V	I_C = -5 mA
I_{CC}	Power Supply Current—8287 —8286		130 160	mA mA	
I_F	Forward Input Current		-0.2	mA	V_F = 0.45V
I_R	Reverse Input Current		50	μA	V_R = 5.25V
V_{OL}	Output Low Voltage —B Outputs —A Outputs		0.5 0.5	V V	I_{OL} = 32 mA I_{OL} = 10 mA
V_{OH}	Output High Voltage —B Outputs —A Outputs	2.4 2.4		V V	I_{OH} = -5 mA I_{OH} = -1 mA
I_{OFF} I_{OFF}	Output Off Current Output Off Current		I_F I_R		V_{OFF} = 0.45V V_{OFF} = 5.25V
V_{IL}	Input Low Voltage —A Side —B Side		0.8 0.9	V V	V_{CC} = 5.0V, See Note 1 V_{CC} = 5.0V, See Note 1
V_{IH}	Input High Voltage	2.0		V	V_{CC} = 5.0V, See Note 1
C_{IN}	Input Capacitance		12	pF	F = 1 MHz V_{BIAS} = 2.5V, V_{CC} = 5V T_A = 25°C

Note: 1. B Outputs — I_{OL} = 32 mA, I_{OH} = -5 mA, C_L = 300 pF A Outputs — I_{OL} = 10 mA, I_{OH} = -1 mA, C_L = 100 pF

PRELIMINARY

A.C. CHARACTERISTICS FOR 8286/8287

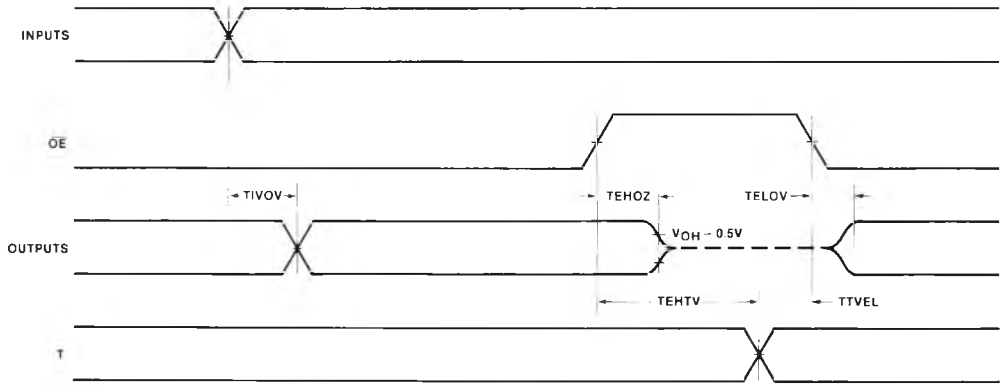
Conditions: V_{CC} = 5V ±5%, T_A = 0°C to 70°C

Loading: B Outputs — I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF
A Outputs — I_{OL} = 10 mA, I_{OH} = −1 mA, C_L = 100 pF

Symbol	Parameter	Min	Max	Units	Test Conditions
TIVOV	Input to Output Delay Inverting Non-Inverting		 25 35	 ns ns	(See Note 1)
TEHTV	Transmit/Receive Hold Time	TEHOZ		ns	
TTVEL	Transmit/Receive Setup	30		ns	
TEHOZ	Output Disable Time		25	ns	
TELOV	Output Enable Time	10	50	ns	

Note: 1. See waveforms and test load circuit on following page

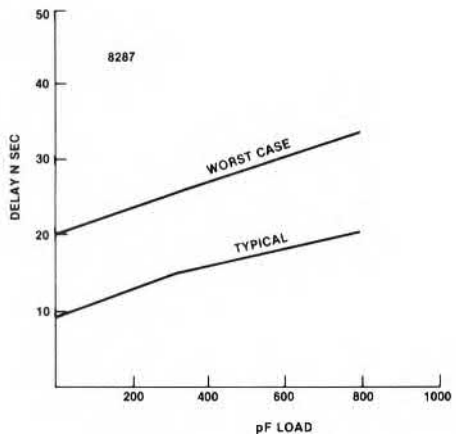
8286/8287 TIMING

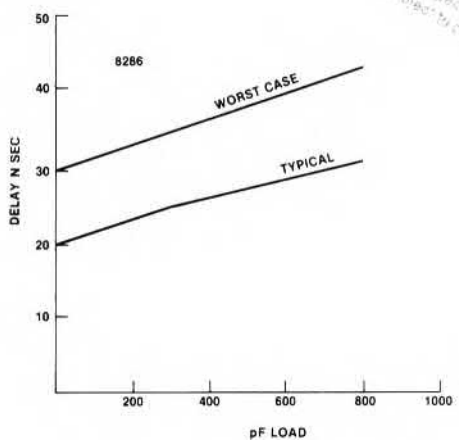


NOTE: 1. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

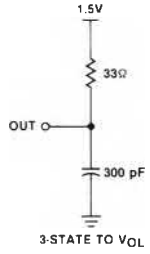
PRELIMINARY
Notice: This is not a final specification. Some parameter limits are subject to change.

OUTPUT DELAY VS. CAPACITANCE

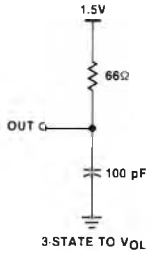




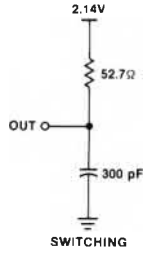
TEST LOAD CIRCUITS



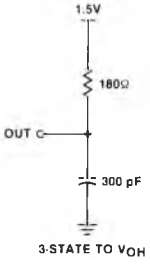
B OUTPUT



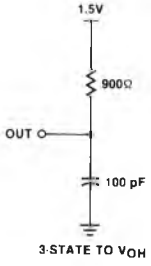
A OUTPUT



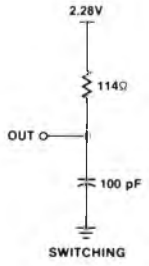
B OUTPUT



B OUTPUT



A OUTPUT



A OUTPUT

intel®

PRELIMINARY

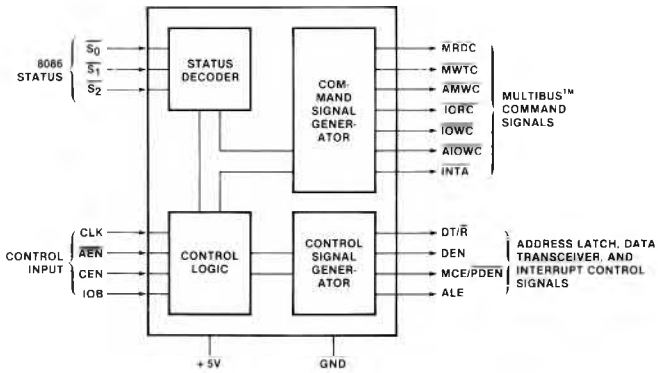
8288
BUS CONTROLLER
FOR 8086, 8088, 8089 PROCESSORS

- **Bipolar Drive Capability**
- **Provides Advanced Commands**
- **Provides Wide Flexibility in System Configurations**
- **3-State Command Output Drivers**
- **Configurable for Use with an I/O Bus**
- **Facilitates Interface to One or Two Multi-Master Busses**

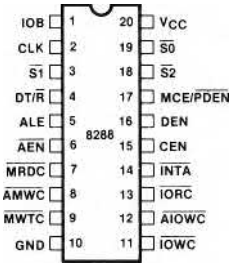
The Intel® 8288 Bus Controller is a 20-pin bipolar component for use with medium-to-large 8086 processing systems. The bus controller provides command and control timing generation as well as bipolar bus drive capability while optimizing system performance.

A strapping option on the bus controller configures it for use with a multi-master system bus and separate I/O bus.

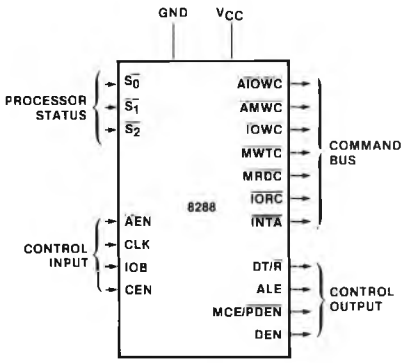
BLOCK DIAGRAM



PIN CONFIGURATION



FUNCTIONAL PIN-OUT



PRELIMINARY

PIN DEFINITIONS

Name	I/O	Function
V_{CC}		+5V supply.
GND		Ground.
$\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$	I	Status Input Pins: These pins are the status input pins from the 8086, 8088 or 8089 processors. The 8288 decodes these inputs to generate command and control signals at the appropriate time. When these pins are not in use (passive) they are all HIGH. (See chart under Command and Control Logic.)
CLK	I	Clock: This is a clock signal from the 8284 clock generator and serves to establish when command and control signals are generated.
ALE	O	Address Latch Enable: This signal serves to strobe an address into the address latches. This signal is active HIGH and latching occurs on the falling (HIGH to LOW) transition. ALE is intended for use with transparent D type latches.
DEN	O	Data Enable: This signal serves to enable data transceivers onto either the local or system data bus. This signal is active HIGH.
DT/$\overline{R}$	O	Data Transmit/Receive: This signal establishes the direction of data flow through the transceivers. A HIGH on this line indicates Transmit (write to I/O or memory) and a LOW indicates Receive (Read).
$\overline{AEN}$	I	Address Enable: $\overline{AEN}$ enables command outputs of the 8288 Bus Controller at least 105 ns after it becomes active (LOW). $\overline{AEN}$ going inactive immediately 3-states the command output drivers. $\overline{AEN}$ does not affect the I/O command lines if the 8288 is in the I/O Bus mode (IOB tied HIGH).
CEN	I	Command Enable: When this signal is LOW all 8288 command outputs and the DEN and $\overline{PDEN}$ control outputs are forced to their inactive state. When this signal is HIGH, these same outputs are enabled.
IOB	I	Input/Output Bus Mode: When the IOB is strapped HIGH the 8288 functions in the I/O Bus mode. When it is strapped LOW, the 8288 functions in the System Bus mode. (See sections on I/O Bus and System Bus modes).

Name	I/O	Function
$\overline{AIOWC}$	O	Advanced I/O Write Command: The $\overline{AIOWC}$ issues an I/O Write Command earlier in the machine cycle to give I/O devices an early indication of a write instruction. Its timing is the same as a read command signal. $\overline{AIOWC}$ is active LOW.
$\overline{IOWC}$	O	I/O Write Command: This command line instructs an I/O device to read the data on the data bus. This signal is active LOW.
$\overline{IORC}$	O	I/O Read Command: This command line instructs an I/O device to drive its data onto the data bus. This signal is active LOW.
$\overline{AMWC}$	O	Advanced Memory Write Command: The $\overline{AMWC}$ issues a memory write command earlier in the machine cycle to give memory devices an early indication of a write instruction. Its timing is the same as a read command signal. $\overline{AMWC}$ is active LOW.
$\overline{MWTC}$	O	Memory Write Command: This command line instructs the memory to record the data present on the data bus. This signal is active LOW.
$\overline{MRDC}$	O	Memory Read Command: This command line instructs the memory to drive its data onto the data bus. This signal is active LOW.
$\overline{INTA}$	O	Interrupt Acknowledge: This command line tells an interrupting device that its interrupt has been acknowledged and that it should drive vectoring information onto the data bus. This signal is active LOW.
MCE/$\overline{PDEN}$	O	This is a dual function pin. MCE (IOB is tied LOW): Master Cascade Enable occurs during an interrupt sequence and serves to read a Cascade Address from a master PIC (Priority Interrupt Controller) onto the data bus. The MCE signal is active HIGH. $\overline{PDEN}$ (IOB is tied HIGH): Peripheral Data Enable enables the data bus transceiver for the I/O bus during I/O instructions. It performs the same function for the I/O bus that DEN performs for the system bus. $\overline{PDEN}$ is active LOW.

PRELIMINARY

COMMAND AND CONTROL LOGIC

The command logic decodes the three 8086, 8088 or 8089 CPU status lines ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) to determine what command is to be issued.

This chart shows the meaning of each status "word".

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Processor State	8288 Command
0	0	0	Interrupt Acknowledge	$\overline{INTA}$
0	0	1	Read I/O Port	$\overline{IORC}$
0	1	0	Write I/O Port	$\overline{IOWC}$,$\overline{AIOWC}$
0	1	1	Halt	None
1	0	0	Code Access	$\overline{MRDC}$
1	0	1	Read Memory	$\overline{MRDC}$
1	1	0	Write Memory	$\overline{MWTC}$,$\overline{AMWC}$
1	1	1	Passive	None

The command is issued in one of two ways dependent on the mode of the 8288 Bus Controller.

I/O Bus Mode — The 8288 is in the I/O Bus mode if the IOB pin is strapped HIGH. In the I/O Bus mode all I/O command lines ($\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$) are always enabled (i.e., not dependent on $\overline{AEN}$). When an I/O command is initiated by the processor, the 8288 immediately activates the command lines using $\overline{PDEN}$ and DT/$\overline{R}$ to control the I/O bus transceiver. The I/O command lines should not be used to control the system bus in this configuration because no arbitration is present. This mode allows one 8288 Bus Controller to handle two external busses. No waiting is involved when the CPU wants to gain access to the I/O bus. Normal memory access requires a "Bus Ready" signal ($\overline{AEN}$ LOW) before it will proceed. It is advantageous to use the IOB mode if I/O or peripherals dedicated to one processor exist in a multi-processor system.

System Bus Mode — The 8288 is in the System Bus mode if the IOB pin is strapped LOW. In this mode no command is issued until 105 ns after the $\overline{AEN}$ Line is activated (LOW). This mode assumes bus arbitration logic will inform the bus controller (on the $\overline{AEN}$ line) when the bus is free for use. Both memory and I/O commands wait for bus arbitration. This mode is used when only one bus exists. Here, both I/O and memory are shared by more than one processor.

Command Outputs

The advanced write commands are made available to initiate write procedures early in the machine cycle. This signal can be used to prevent the processor from entering an unnecessary wait state.

The command outputs are:

$\overline{MRDC}$ — Memory Read Command
$\overline{MWTC}$ — Memory Write Command
$\overline{IORC}$ — I/O Read Command
$\overline{IOWC}$ — I/O Write Command
$\overline{AMWC}$ — Advanced Memory Write Command
$\overline{AIOWC}$ — Advanced I/O Write Command
$\overline{INTA}$ — Interrupt Acknowledge

$\overline{INTA}$ (Interrupt Acknowledge) acts as an I/O read during an interrupt cycle. Its purpose is to inform an interrupting device that its interrupt is being acknowledged and that it should place vectoring information onto the data bus.

Control Outputs

The control outputs of the 8288 are Data Enable (DEN), Data Transmit/Receive (DT/$\overline{R}$) and Master Cascade Enable/Peripheral Data Enable (MCE/$\overline{PDEN}$). The DEN signal determines when the external bus should be enabled onto the local bus and the DT/$\overline{R}$ determines the direction of data transfer. These two signals usually go to the chip select and direction pins of a transceiver.

The MCE/$\overline{PDEN}$ pin changes function with the two modes of the 8288. When the 8288 is in the IOB mode (IOB HIGH) the $\overline{PDEN}$ signal serves as a dedicated data enable signal for the I/O or Peripheral System bus.

Interrupt Acknowledge and MCE

The MCE signal is used during an interrupt acknowledge cycle if the 8288 is in the System Bus mode (IOB LOW). During any interrupt sequence there are two interrupt acknowledge cycles that occur back to back. During the first interrupt cycle no data or address transfers take place. Logic should be provided to mask off MCE during this cycle. Just before the second cycle begins the MCE signal gates a master Priority Interrupt Controller's (PIC) cascade address onto the processor's local bus where ALE (Address Latch Enable) strobes it into the address latches. On the leading edge of the second interrupt cycle the addressed slave PIC gates an interrupt vector onto the system data bus where it is read by the processor.

If the system contains only one PIC, the MCE signal is not used. In this case the second Interrupt Acknowledge signal gates the interrupt vector onto the processor bus.

Address Latch Enable and Halt

Address Latch Enable (ALE) occurs during each machine cycle and serves to strobe the current address into the address latches. ALE also serves to strobe the status ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) into a latch for halt state decoding.

Command Enable

The Command Enable (CEN) input acts as a command qualifier for the 8288. If the CEN pin is high the 8288 functions normally. If the CEN pin is pulled LOW, all command lines are held in their inactive state (not 3-state). This feature can be used to implement memory partitioning and to eliminate address conflicts between system bus devices and resident bus devices.

PRELIMINARY
Advance Information
This is not a final specification
Parametric limits are subject to change

D.C. AND OPERATING CHARACTERISTICS

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias ..... 0°C to 70°C
Storage Temperature ..... −65°C to +150°C
All Output and Supply Voltages ..... −0.5V to +7V
All Input Voltages ..... −1.0V to +5.5V
Power Dissipation ..... 1.5 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS FOR THE 8288

Conditions: $V_{CC} = 5V \pm 10\%$, $T_A = 0°C$ to $70°C$

Symbol	Parameter	Min	Max	Unit	Test Conditions
V_C	Input Clamp Voltage		−1	V	$I_C = -5$ mA
I_{CC}	Power Supply Current		230	mA	
I_F	Forward Input Current		−0.7	mA	$V_F = 0.45V$
I_R	Reverse Input Current		50	μA	$V_R = V_{CC}$
V_{OL}	Output Low Voltage—Command Outputs Control Outputs		0.5 0.5	V V	$I_{OL} = 32$ mA $I_{OL} = 16$ mA
V_{OH}	Output High Voltage—Command Outputs Control Outputs	2.4 2.4		V V	$I_{OH} = -5$ mA $I_{OH} = -1$ mA
V_{IL}	Input Low Voltage		0.8	V	
V_{IH}	Input High Voltage	2.0		V	
I_{OFF}	Output Off Current		100	μA	$V_{OFF} = 0.4$ to $5.25V$

A.C. CHARACTERISTICS FOR THE 8288

Conditions: $V_{CC} = 5V \pm 10\%$, $T_A = 0°C$ to $70°C$

TIMING REQUIREMENTS

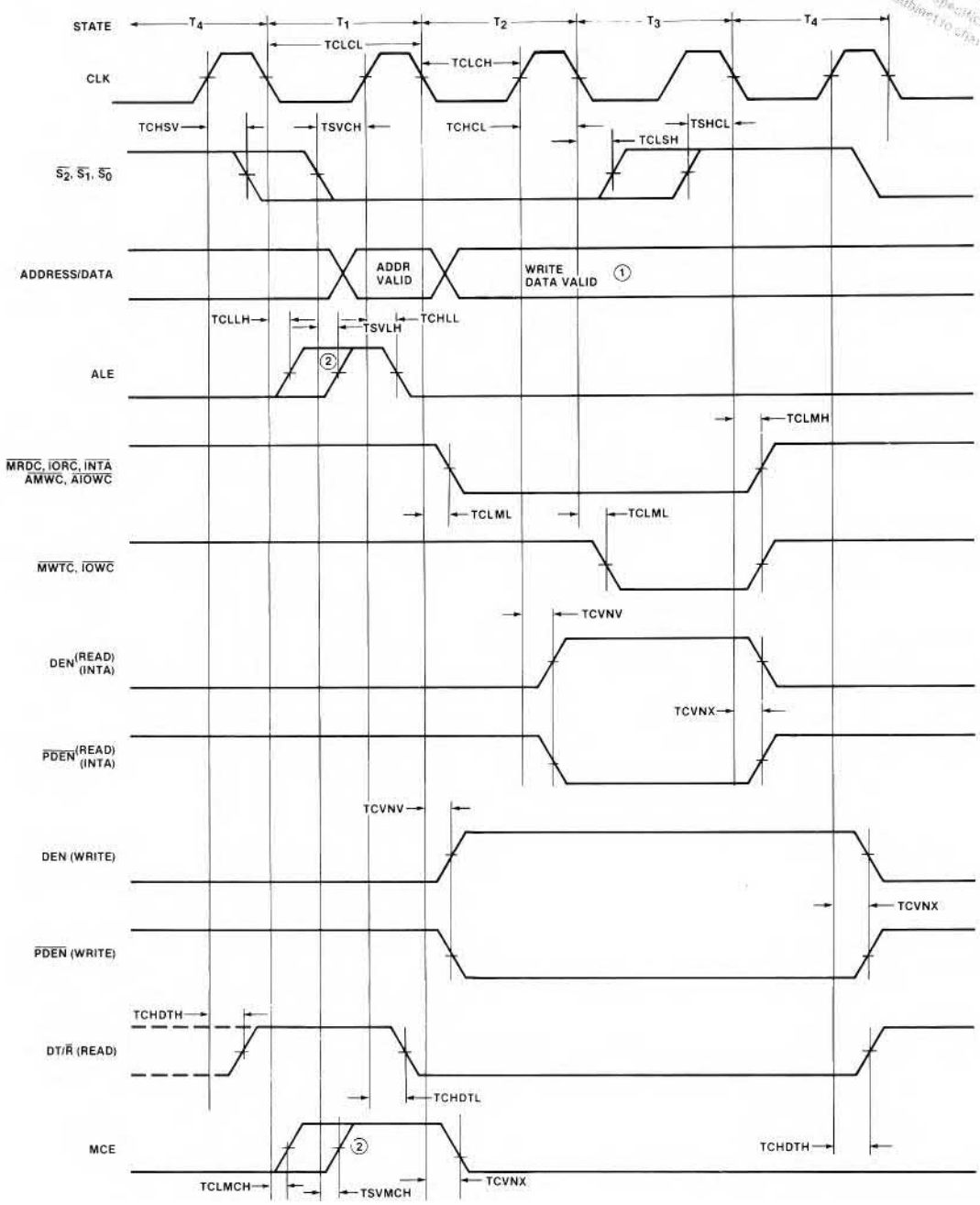
Symbol	Parameter	Min	Max	Unit	Loading
TCLCL	CLK Cycle Period	125		ns	
TCLCH	CLK Low Time	66		ns	
TCHCL	CLK High Time	40		ns	
TSVCH	Status Active Setup Time	65		ns	
TCHSV	Status Active Hold Time	10		ns	
TSHCL	Status Inactive Setup Time	55		ns	
TCLSH	Status Inactive Hold Time	10		ns	

TIMING RESPONSES

Symbol	Parameter	Min	Max	Unit	Loading
TCVNV	Control Active Delay	5	45	ns	
TCVNX	Control Inactive Delay	10	45	ns	
TCLLH, TCLMCH	ALE MCE Active Delay (from CLK)		15	ns	
TSVLH, TSVMCH	ALE MCE Active Delay (from Status)		15	ns	
TCHLL	ALE Inactive Delay		15	ns	$\overline{MRDC}$, $\overline{IORC}$, $\overline{MWTC}$, $\overline{IOWC}$, $\overline{INTA}$, $\overline{AMWC}$, $\overline{AIOWC}$: $I_{OL} = 32$ mA, $I_{OH} = -5$ mA, $C_L = 300$ pF
TCLML	Command Active Delay	10	35	ns	
TCLMH	Command Inactive Delay	10	35	ns	
TCHDTL	Direction Control Active Delay		50	ns	
TCHDTH	Direction Control Inactive Delay		30	ns	
TAELCH	Command Enable Time		40	ns	
TAEHCZ	Command Disable Time		40	ns	
TAELCV	Enable Delay Time	105	275	ns	Other: $I_{OL} = 16$ mA, $I_{OH} = -1$ mA, $C_L = 80$ pF
TAEVNV	$\overline{AEN}$ to DEN		20	ns	
TCEVNV	CEN to DEN, PDEN		20	ns	
TCELRH	CEN to Command		TCLML	ns	

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

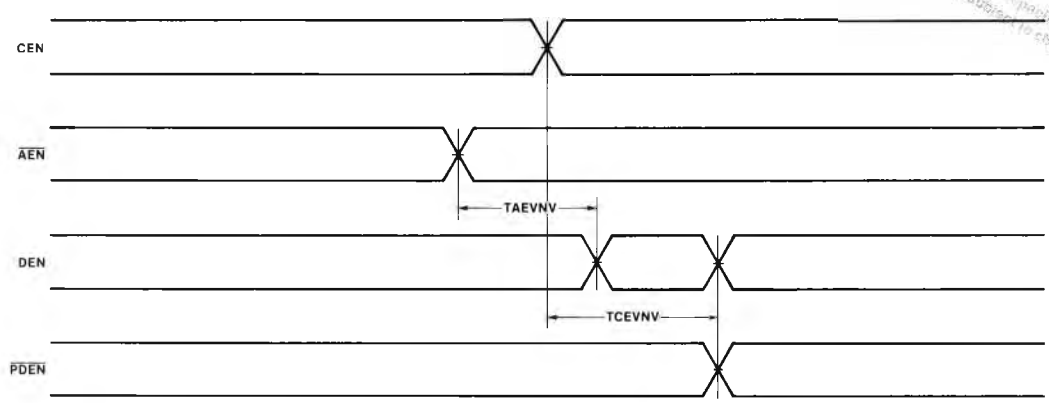
8288 TIMING DIAGRAM



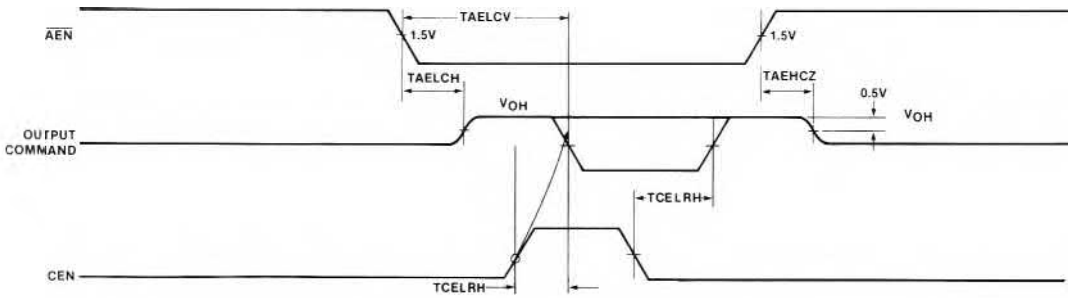
NOTES:
1. ADDRESS/DATA BUS IS SHOWN ONLY FOR REFERENCE PURPOSES
2. LEADING EDGE OF ALE AND MCE IS DETERMINED BY THE FALLING EDGE OF CLK OR STATUS GOING ACTIVE, WHICHEVER OCCURS LAST
3. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS SPECIFIED OTHERWISE

PRELIMINARY
Notice: This is not a final specification. Parametric limits are subject to change

DEN, PDEN QUALIFICATION TIMING

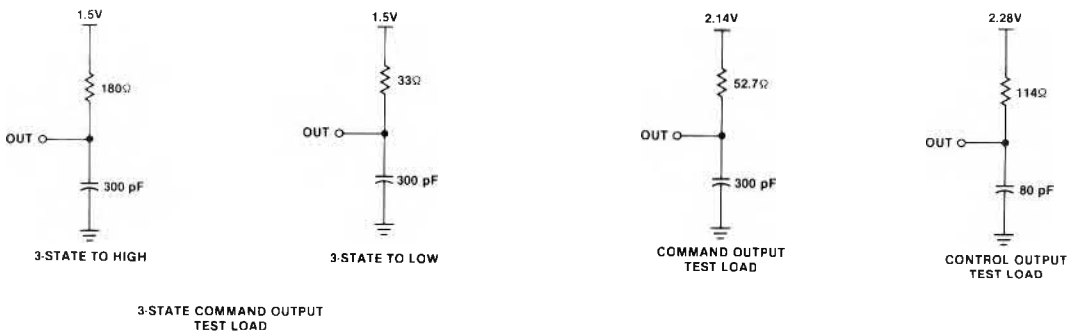


8288 ADDRESS ENABLE ($\overline{AEN}$) TIMING (3-STATE ENABLE/DISABLE)



NOTE: CEN MUST BE LOW OR VALID PRIOR TO T2 TO PREVENT THE COMMAND FROM BEING GENERATED

TEST LOAD CIRCUITS



intel®

PRELIMINARY

8289
BUS ARBITER

- Provides Multi-Master System Bus Protocol
- Synchronizes 8086/8088 Processors With Multi-Master Bus
- Provides Simple Interface With 8288 Bus Controller
- Four Operating Modes For Flexible System Configuration
- Compatible with Intel Bus Standard MULTIBUS™
- Provides System Bus Arbitration For 8089 IOP In Remote Mode

The Intel 8289 Bus Arbiter is a 20-pin, 5-volt-only bipolar component for use with medium to large 8086/8088 multi-master/multiprocessing systems. The 8289 provides system bus arbitration for systems with multiple bus masters, such as an 8086 CPU with 8089 IOP in its REMOTE mode, while providing bipolar buffering and drive capability.

BLOCK DIAGRAM

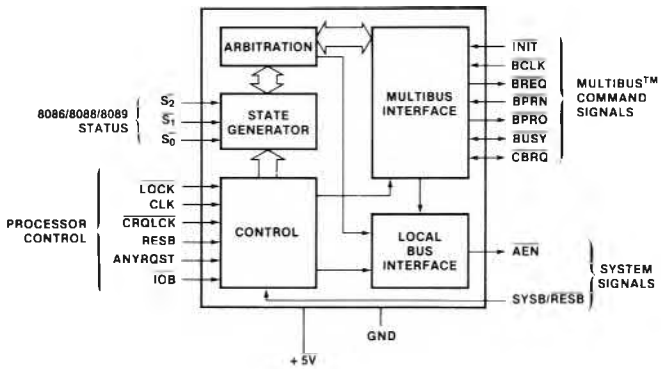


Figure 1. Block Diagram.

PIN DIAGRAM

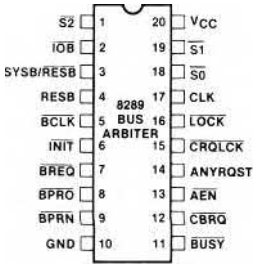


Figure 2. Pin Diagram.

FUNCTIONAL PINOUT

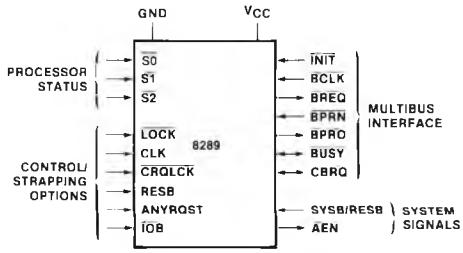


Figure 3. Functional Pinout.

PRELIMINARY

FUNCTIONAL DESCRIPTION

The 8289 Bus Arbiter operates in conjunction with the 8288 Bus Controller to interface 8086/8088/8089 processors to a multi-master system bus (both the 8086 and 8088 are configured in their max mode). The processor is unaware of the arbiter's existence and issues commands as though it has exclusive use of the system bus. If the processor does not have the use of the multi-master system bus, the arbiter prevents the Bus Controller (8288), the data transceivers and the address latches from accessing the system bus (e.g. all bus driver outputs are forced into the high impedance state). Since the command sequence was not issued by the 8288, the system bus will appear as "Not Ready" and the processor will enter wait states. The processor will remain in Wait until the Bus Arbiter acquires the use of the multi-master system bus whereupon the arbiter will allow the bus controller, the data transceivers, and the address latches to access the system. Typically, once the command has been issued and a data transfer has taken place, a transfer acknowledge (XACK) is returned to the processor to indicate "READY" from the accessed slave device. The processor then completes its transfer cycle. Thus the arbiter serves to multiplex a processor (or bus master) onto a multi-master system bus and avoid contention problems between bus masters.

ARBITRATION BETWEEN BUS MASTERS

In general, higher priority masters obtain the bus when a lower priority master completes its present transfer cycle. Lower priority bus masters obtain the bus when a higher priority master is not accessing the system bus. A strapping option (ANYRQST) is provided to allow the arbiter to surrender the bus to a lower priority master as though it were a master of higher priority. If there are no other bus masters requesting the bus, the arbiter maintains the bus so long as its processor has not entered the HALT State. The arbiter will not voluntarily surrender the system bus and has to be forced off by another master's bus request, the HALT State being the only exception. Additional strapping options permit other modes of operation wherein the multi-master system bus is surrendered or requested under different sets of conditions.

PRIORITY RESOLVING TECHNIQUES

Since there can be many bus masters on a multi-master system bus, some means of resolving priority between bus masters simultaneously requesting the bus must be provided. The 8289 Bus Arbiter provides several resolving techniques. All the techniques are based on a priority concept that at a given time one bus master will have priority above all the rest. There are provisions for using parallel priority resolving techniques, serial priority resolving techniques, and rotating priority techniques.

Parallel Priority Resolving

The parallel priority resolving technique uses a separate bus request line ($\overline{BREQ}$) for each arbiter on the multi-master system bus, see Figure 4. Each $\overline{BREQ}$ line enters into a priority encoder which generates the binary address of the highest priority $\overline{BREQ}$ line which is active. The binary address is decoded by a decoder to select the corresponding $\overline{BPRN}$ (Bus Priority In) line to be returned to the highest priority requesting arbiter. The arbiter receiving priority ($\overline{BPRN}$ true) then allows its associated bus master onto the multi-master system bus as soon as it becomes available (i.e., the bus is no longer busy). When one bus arbiter gains priority over another arbiter it cannot immediately seize the bus, it must wait until the present bus transaction is complete. Upon completing its transaction the present bus occupant recognizes that it no longer has priority and surrenders the bus by releasing $\overline{BUSY}$. $\overline{BUSY}$ is an active low "OR" tied signal line which goes to every bus arbiter on the system bus. When $\overline{BUSY}$ goes inactive (high), the arbiter which presently has bus priority ($\overline{BPRN}$ true) then seizes the bus and pulls $\overline{BUSY}$ low to keep other arbiters off of the bus. See waveform timing diagram, Figure 5.

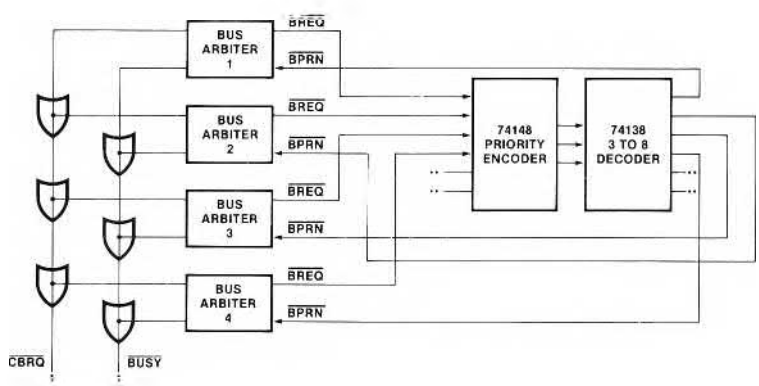
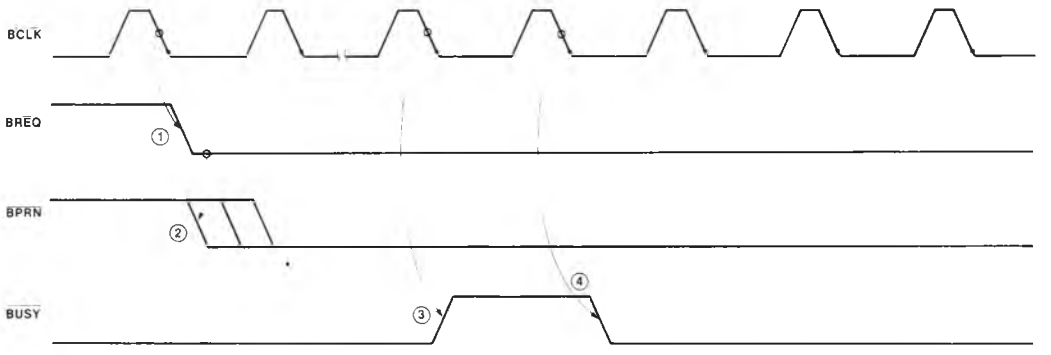


Figure 4. Parallel Priority Resolving Technique.



① HIGHER PRIORITY BUS ARBITER REQUESTS THE MULTI-MASTER SYSTEM BUS.
② ATTAINS PRIORITY.
③ LOWER PRIORITY BUS ARBITER RELEASES BUSY.
④ HIGHER PRIORITY BUS ARBITER THEN ACQUIRES THE BUS AND PULLS BUSY DOWN.

Figure 5. Higher Priority Arbiter obtaining the Bus from a Lower Priority Arbiter.

Note that all multi-master system bus transactions are synchronized to the bus clock (BCLK). This allows the parallel priority resolving circuitry or any other priority resolving scheme employed to settle.

Serial Priority Resolving

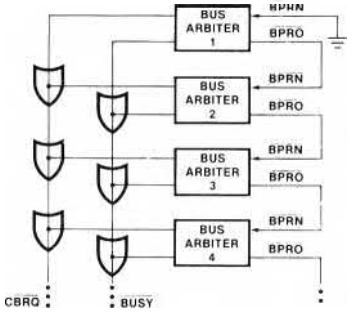
The serial priority resolving technique eliminates the need for the priority encoder-decoder arrangement by daisy-chaining the bus arbiters together, connecting the higher priority bus arbiter's $\overline{\text{BPRO}}$ (Bus Priority Out) output to the $\overline{\text{BPRN}}$ of the next lower priority. See Figure 6.

Rotating Priority Resolving

The rotating priority resolving technique is similar to that of the parallel priority resolving technique except that priority is dynamically re-assigned. The priority encoder is replaced by a more complex circuit which rotates priority between requesting arbiters thus allowing each arbiter an equal chance to use the multi-master system bus, over time.

WHICH PRIORITY RESOLVING TECHNIQUE TO USE

There are advantages and disadvantages for each of the techniques described above. The rotating priority resolving technique requires substantial external logic to implement while the serial technique uses no external logic but can accommodate only a limited number of bus arbiters before the daisy-chain propagation delay exceeds the multi-master's system bus clock ($\overline{\text{BCLK}}$). The parallel priority resolving technique is in general a good compromise between the other two techniques. It allows for many arbiters to be present on the bus while not requiring too much logic to implement.



THE NUMBER OF ARBITERS THAT MAY BE DAISY CHAINED TOGETHER IN THE SERIAL PRIORITY RESOLVING SCHEME IS A FUNCTION OF BCLK AND THE PROPAGATION DELAY FROM ARBITER TO ARBITER. NORMALLY, AT 10 MHz ONLY 3 ARBITER MAY BE DAISY CHAINED.

Figure 6. Serial Priority Resolving.

PRELIMINARY

8289 MODES OF OPERATION

There are two types of processors in the 8086 family. An Input/Output processor (the 8089 IOP) and the 8086/8088 CPUs. Consequently, there are two basic operating modes in the 8289 bus arbiter. One, the $\overline{IOB}$ (I/O Peripheral Bus) mode, permits the processor access to both an I/O Peripheral Bus and a multi-master system bus. The second, the RESB (Resident Bus mode), permits the processor to communicate over both a Resident Bus and a multi-master system bus. An I/O Peripheral Bus is a bus where all devices on that bus, including memory, are treated as I/O devices and are addressed by I/O commands. All memory commands are directed to another bus, the multi-master system bus. A Resident Bus can issue both memory and I/O commands, but it is a distinct and separate bus from the multi-master system bus. The distinction is that the Resident Bus has only one master, providing full availability and being dedicated to that one master.

The $\overline{IOB}$ strapping option configures the 8289 Bus Arbiter into the $\overline{IOB}$ mode and the strapping option RESB configures it into the RESB mode. It might be noted at this point that if both strapping options are strapped false, the arbiter interfaces the processor to a multi-master system bus only (see Figure 7). With both options strapped true, the arbiter interfaces the processor to a multi-master system bus, a Resident Bus, and an I/O Bus.

In the $\overline{IOB}$ mode, the processor communicates and controls a host of peripherals over the Peripheral Bus. When the I/O Processor needs to communicate with system memory, it does so over the system memory bus. Figure 8 shows a possible I/O Processor system configuration.

The 8086 and 8088 processor can communicate with a Resident Bus and a multi-master system bus. Two bus controllers and only one Bus Arbiter would be needed in such a configuration as shown in Figure 9. In such a system configuration the processor would have access to memory and peripherals of both busses. Memory mapping techniques are applied to select which bus is to be accessed. The SYSB/$\overline{RESB}$ input on the arbiter serves to instruct the arbiter as to whether or not the system bus is to be accessed. The signal connected to SYSB/$\overline{RESB}$ also enables or disables commands from one of the bus controllers.

A summary of the modes that the 8289 has, along with its response to its status lines inputs, is summarized in Table 1.

*In some system configurations it is possible for a non-I/O Processor to have access to more than one Multi-Master System Bus. see 8289 Application Note.

	Status Lines From 8086 or 8088 or 8089			IOB Mode Only	RESB (Mode) Only $\overline{IOB}$ = High RESB = High		IOB Mode RESB Mode $\overline{IOB}$ = Low RESB = High		Single Bus Mode $\overline{IOB}$ = High RESB = Low
	$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	$\overline{IOB}$ = Low	SYSB/$\overline{RESB}$ = High	SYSB/$\overline{RESB}$ = Low	SYSB/$\overline{RESB}$ = High	SYSB/$\overline{RESB}$ = Low	
I/O COMMANDS	0	0	0	x	✓	x	x	x	✓
	0	0	1	x	✓	x	x	x	✓
	0	1	0	x	✓	x	x	x	✓
HALT	0	1	1	x	x	x	x	x	x
MEM COMMANDS	1	0	0	✓	✓	x	✓	x	✓
	1	0	1	✓	✓	x	✓	x	✓
	1	1	0	✓	✓	x	✓	x	✓
IDLE	1	1	1	x	x	x	x	x	x

NOTES:

1. X = Multi-Master System Bus is allowed to be Surrendered
2. ✓ = Multi-Master System Bus is Requested.

Mode	Pin Strapping	Multi-Master System Bus	
		Requested**	Surrendered*
Single Bus Multi-Master Mode	$\overline{IOB}$ = High RESB = Low	Whenever the processor's status lines go active	HLT + TI • CBRQ + HPBRQ†
RESB Mode Only	$\overline{IOB}$ = High RESB = High	SYSB/$\overline{RESB}$ = High • ACTIVE STATUS	(SYSB/$\overline{RESB}$ = Low + TI) • CBRQ + HLT + HPBRQ
IOB Mode Only	$\overline{IOB}$ = Low RESB = Low	Memory Commands	(I/O Status + TI) • CBRQ + HLT + HPBRQ
IOB Mode RESB Mode	$\overline{IOB}$ = Low RESB = High	(Memory Command) • (SYSB/$\overline{RESB}$ = High)	((I/O Status Commands) + SYSB/$\overline{RESB}$ = LOW)) • CBRQ + HPBRQ† + HLT

NOTES:

*$\overline{LOCK}$ prevents surrender of Bus to any other arbiter, $\overline{CRQLCK}$ prevents surrender of Bus to any lower priority arbiter.
**Except for HALT and Passive or IDLE Status.
†HPBRQ. Higher priority Bus request or $\overline{BPRN}$ = 1.

1. $\overline{IOB}$ Active Low
2. RESB Active High.
3. + is read as "OR" and • as "AND."
4. TI = Processor Idle Status $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ = 111
5. HLT = Processor Halt Status $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ = 011

Table 1. Summary of 8289 Modes, Requesting and Relinquishing the Multi-master system bus.

PRELIMINARY
Notice: This is not a final specification. Parametric limits are subject to change.

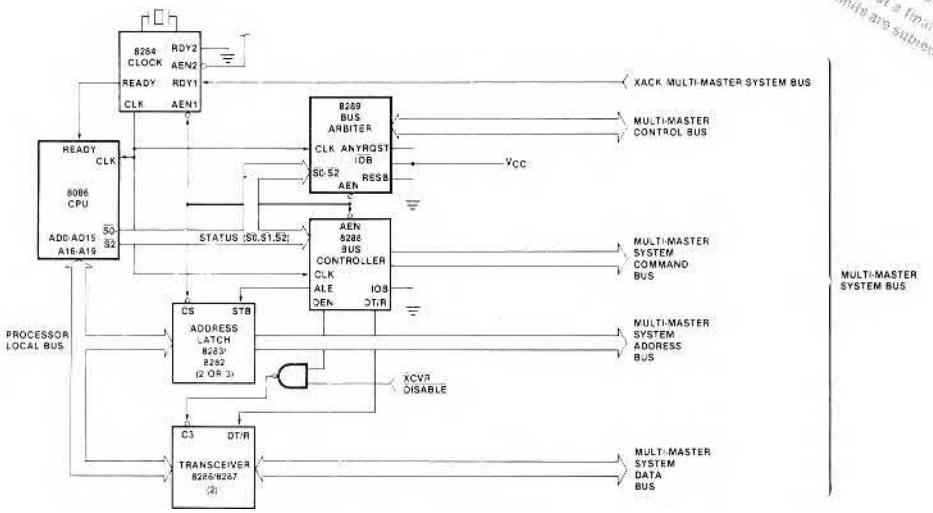


Figure 7. Typical Medium Complexity CPU System.

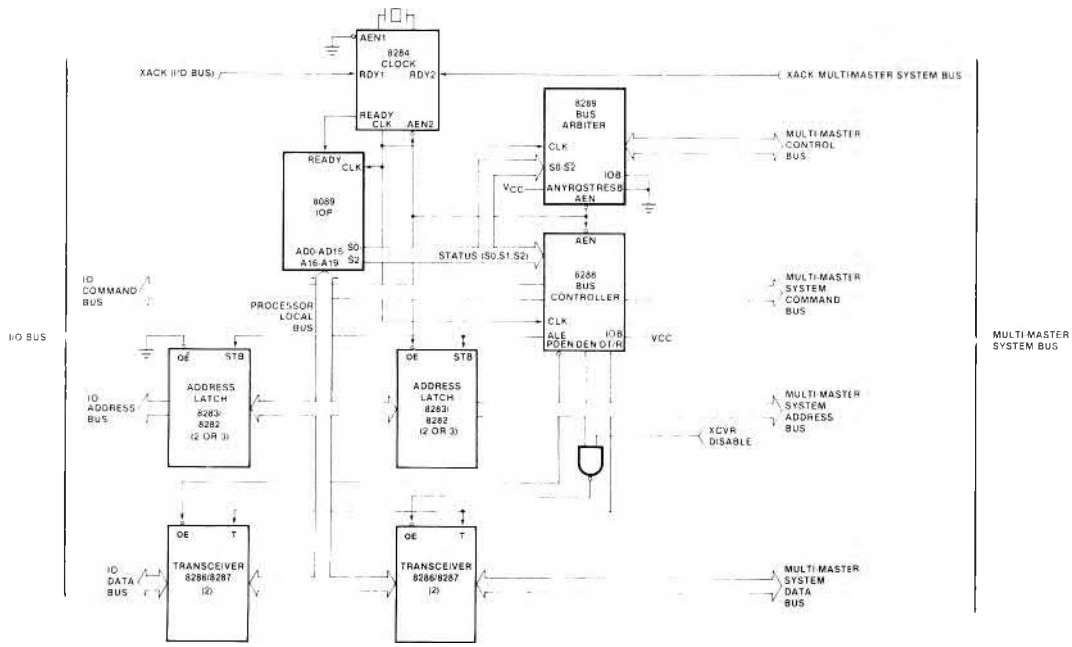
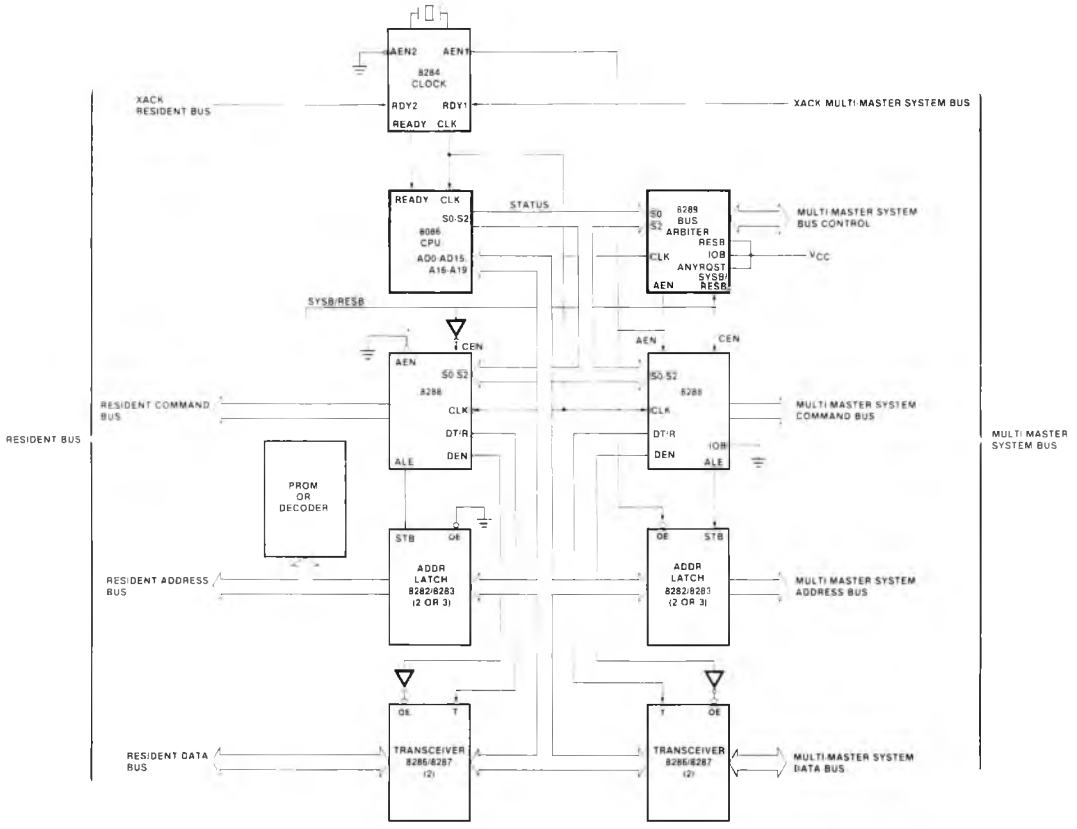


Figure 8. Typical Medium Complexity IOB System.

PRELIMINARY



*BY ADDING ANOTHER 8289 ARBITER AND CONNECTING ITS AEN TO THE 8288 WHOSE AEN IS PRESENTLY GROUNDED, THE PROCESSOR COULD HAVE ACCESS TO TWO MULTI-MASTER BUSES

Figure 9. 8289 Bus Arbiter Shown in System-Resident Bus Configuration.

PRELIMINARY

PIN DEFINITIONS

Name	I/O	Function
V_{CC}		+ 5V supply ± 10%
GND		Ground
$\overline{S0}$,$\overline{S1}$,$\overline{S2}$	I	STATUS INPUT PINS: These pins are the status input pins from an 8086, 8088 or 8089 processor. The 8289 decodes these pins to initiate bus request and surrender actions. (See Table 1)
CLK	I	CLOCK: This is the clock from the 8284 clock chip and serves to establish when bus arbiter actions are initiated.
$\overline{LOCK}$	I	LOCK: $\overline{LOCK}$ is a processor generated signal which when activated (low) serves to prevent the arbiter from surrendering the multi-master system bus to any other bus arbiter, regardless of its priority.
$\overline{CRQLCK}$	I	COMMON REQUEST LOCK: $\overline{CRQLCK}$ is an active low signal which serves to prevent the arbiter from surrendering the multi-master system bus to any other bus arbiter requesting the bus through the $\overline{CBRQ}$ input pin.
RESB	I	RESB: RESIDENT BUS is a strapping option to configure the arbiter to operate in systems having both a multi-master system bus and a Resident Bus. When it is strapped high the multi-master system bus is requested or surrendered as a function of the SYSB/$\overline{RESB}$ input pin. When it is strapped low the SYSB/$\overline{RESB}$ input is ignored.
ANYRQST	I	ANY REQUEST: ANYRQST is a strapping option which permits the multi-master system bus to be surrendered to a lower priority arbiter as though it were an arbiter of higher priority (i.e., when a lower priority arbiter requests the use of the multi-master system bus, the bus is surrendered as soon as it is possible). Strapping $\overline{CBRQ}$ low and ANYRQST high forces the 8289 arbiter to surrender the multi-master system bus after each transfer cycle. Note that when surrender occurs BREQ is driven false (high).
$\overline{IOB}$	I	IO BUS: $\overline{IOB}$ is a strapping option which configures the 8289 Arbiter to operate in systems having both an IO Bus (Peripheral Bus) and a multi-master system bus. The arbiter requests and surrenders the use of the multi-master system bus as a function of the status line, $\overline{S2}$. The multi-master system bus is permitted to be surrendered while the processor is performing IO commands and is requested whenever the processor performs a memory command. Interrupt cycles are assumed as coming from the peripheral bus and are treated as would be an IO command.
$\overline{AEN}$	O	ADDRESS ENABLE. $\overline{AEN}$ is the output of the 8289 Arbiter to the processor's address latches, to the 8288 Bus Controller and 8284 Clock Generator. $\overline{AEN}$ serves to instruct the Bus Controller and address latches when to tri-state their output drivers.
SYSB/$\overline{RESB}$	I	SYSTEM BUS/$\overline{RESIDENT}$ $\overline{BUS}$: SYSB/$\overline{RESB}$ is an input signal when the arbiter is configured in the S.R. Mode (RESB is strapped high) which serves to determine when the multi-master system bus is requested and when the multi-master system bus surrendering is permitted. The signal is intended to originate from some form of address mapping circuitry such as a decoder or PROM attached to the resident address bus. Signal transitions and glitches are permitted on this pin from Ø1 of T4 to Ø 1 to T2 of the processor cycle. During the period from Ø1 of T2 to Ø1 of T4 only clean transitions are permitted on this pin (no glitches). If a glitch does occur the arbiter may capture or miss it, and the multi-master system bus may be requested or surrendered, depending upon the state of the glitch. The arbiter requests the multi-master system bus in the S.R. Mode when the state of the SYSB/$\overline{RESB}$ pin is high and permits the bus to be surrendered when this pin is low.
$\overline{CBRQ}$	I/O	COMMON BUS REQUEST: $\overline{CBRQ}$ is an input signal which serves to instruct the arbiter if there are any other arbiters of lower priority requesting the use of the multi-master system bus. The $\overline{CBRQ}$ pins (open-collector output) of all the 8289 Bus Arbiters which are to surrender the multi-master system bus upon request are connected together. The Bus Arbiter running the current transfer cycle will not itself pull the $\overline{CBRQ}$ line low. Any other arbiter connected to the $\overline{CBRQ}$ line can request the multi-master system bus. The arbiter presently running the current transfer cycle drops its $\overline{BREQ}$ signal and surrenders the bus whenever the

PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

PIN DEFINITIONS (Cont'd)

Name	I/O	Function
		proper surrender conditions exist. Strapping $\overline{CBREQ}$ low and ANYRQST high allows the multi-master system bus to be surrendered after each transfer cycle. See the pin definition of ANYRQST.
$\overline{INIT}$	I	INITIALIZE: $\overline{INIT}$ is an active low multi-master system bus input signal which is used to reset all the bus arbiters on the multi-master system bus. After initialization, no arbiters have the use of the multi-master system bus.
$\overline{BCLK}$	I	BUS CLOCK: $\overline{BCLK}$ is the multi-master system bus clock to which all multi-master system bus interface signals are synchronized.
$\overline{BREQ}$	O	BUS REQUEST: $\overline{BREQ}$ is an active low output signal in the parallel Priority Resolving Scheme which the arbiter activates to request the use of the multi-master system bus.
$\overline{BPRN}$	I	BUS PRIORITY IN: $\overline{BPRN}$ is the active low signal returned to the arbiter to instruct it that it may acquire the multi-master system bus on the next falling edge of BCLK. $\overline{BPRN}$ indicates to the arbiter that it is the highest priority requesting arbiter presently on the bus. The loss of $\overline{BPRN}$ instructs the arbiter that it has loss priority to a higher priority arbiter.
$\overline{BPRO}$	O	BUS PRIORITY OUT: $\overline{BPRO}$ is an active low output signal which is used in the serial priority resolving scheme where $\overline{BPRO}$ is daisy chained to $\overline{BPRN}$ of the next lower priority arbiter.
$\overline{BUSY}$	I/O	BUSY: $\overline{BUSY}$ is an active low open collector multi-master system bus interface signal which is used to instruct all the arbiters on the bus when the multi-master system bus is available. When the multi-master system bus is available the highest requesting arbiter (determined by $\overline{BPRN}$) seizes the bus and pulls $\overline{BUSY}$ low to keep other arbiters off of the bus. When the arbiter is done with the bus it releases the $\overline{BUSY}$ signal permitting it to go high and thereby allowing another arbiter to acquire the multi-master system bus.

PRELIMINARY
Notice: This is not a final specification. Some parameters are subject to change.

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . – 65°C to + 150°C
All Output and Supply Voltages . . . . . – 0.5V to + 7V
All Input Voltages . . . . . – 1.0V to + 5.5V
Power Dissipation . . . . . 1.5 Watt

COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

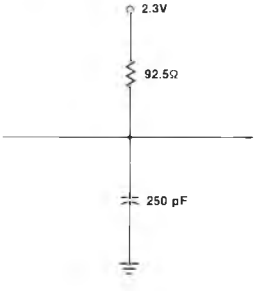
D.C. CHARACTERISTICS FOR THE 8289

CONDITIONS: $T_A = 0°$ to 70°C, $V_{CC} = 5V \pm 10\%$

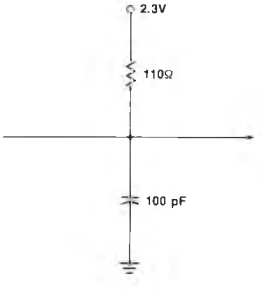
Symbol	Parameter	Min.	Max.	Units	Test Condition
V_C	Input Clamp Voltage		– 1.0	V	$V_{CC} = 4.50V$, $I_C = -5$ mA
I_F	Input Forward Current		– 0.5	mA	$V_{CC} = 5.50V$, $V_F = 0.45V$
I_R	Reverse Input Leakage Current		60	μA	$V_{CC} = 5.50$, $V_R = 5.50$
V_{OL}	Output Low Voltage				
	$\overline{BUSY}$, $\overline{CBRQ}$		0.45	V	$I_{OL} = 20$ mA, $C_L = 250$ pF 1)
	$\overline{AEN}$		0.45	V	$I_{OL} = 16$ mA, $C_L = 100$ pF 2)
	$\overline{BPRO}$, $\overline{BREQ}$		0.45	V	$I_{OL} = 10$ mA, $C_L = 60$ pF 3)
V_{OH}	Output High Voltage				
	$\overline{BUSY}$, $\overline{CBRQ}$	Open Collector			
	All Other Outputs	2.4		V	$I_{OH} = 400$ μA
I_{CC}	Power Supply Current		165	mA	
V_{IL}	Input Low Voltage		.8	V	
V_{IH}	Input High Voltage	2.0		V	
Cin Status	Input Capacitance		25	pF	
Cin (Others)	Input Capacitance		12	pF	

TEST CIRCUITS:

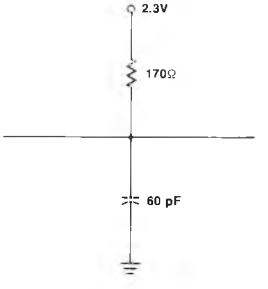
1) $\overline{BUSY}$, $\overline{CBRQ}$



2) $\overline{AEN}$



3) $\overline{BPRO}$, $\overline{BREQ}$



PRELIMINARY
Notice: This is not a final specification. Some parametric limits are subject to change.

A.C. CHARACTERISTICS FOR THE 8289

CONDITIONS: $V_{CC} = 5V \pm 10\%$, $T_A = 0°C$ to $70°C$

Timing Requirements

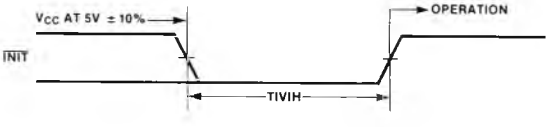
Symbol	Parameter	Min.	Max.	Unit
TCLCL	CLK Cycle Period	125		ns
TCLCH	CLK Low Time	65		ns
TCHCL	CLK High Time	35		ns
TSVCH	Status Active Setup	65	TCLCL-10	ns
TSHCL	Status Inactive Setup	50	TCLCL-10	ns
THVCH	Status Active Hold	10		ns
THVCL	Status Inactive Hold	10		ns
TBYSBL	$\overline{BUSY}$↑↓Setup to $\overline{BCLK}$↓	20		ns
TCBSBL	$\overline{CBRQ}$↑↓Setup to $\overline{BCLK}$↓	20		ns
TBLBL	$\overline{BCLK}$ Cycle Time	100		ns
TBHCL	$\overline{BCLK}$ High Time	30	.65[TBLBL]	ns
TCLLL1	$\overline{LOCK}$ Inactive Hold	20		ns
TCLLL2	$\overline{LOCK}$ Active Setup	40		ns
TPNBL	$\overline{BPRN}$↓↑to $\overline{BCLK}$ Setup Time	15		ns
TCLSR1	SYSB/$\overline{RESB}$ Setup	0		ns
TCLSR2	SYSB/$\overline{RESB}$ Hold	20		ns
TIVIH	Initialization Pulse Width	3 TBLBL + 3 TCLCL		ns

Timing Responses

Symbol	Parameter	Min.	Max.	Unit	Loading
TBLBRL	$\overline{BCLK}$ to $\overline{BREQ}$ Delay↓↑		35	ns	
TBLPOH	$\overline{BCLK}$ to $\overline{BPRO}$↓↑ (See Note 1)		40	ns	
TPNPO	$\overline{BPRN}$↓↑to $\overline{BPRO}$↓↑Delay (See Note 1)		25	ns	
TBLBYL	$\overline{BCLK}$ to $\overline{BUSY}$ Low		60	ns	
TBLBYH	$\overline{BCLK}$ to $\overline{BUSY}$ Float (See Note 2)		35	ns	
TCLAEH	CLK to $\overline{AEN}$ High		65	ns	
TBLAEL	$\overline{BCLK}$ to $\overline{AEN}$ Low		40	ns	
TBLCBL	$\overline{BCLK}$ to $\overline{CBRQ}$ Low		60	ns	
TBLCBH	$\overline{BCLK}$ to $\overline{CBRQ}$ Float (See Note 2)		35	ns	

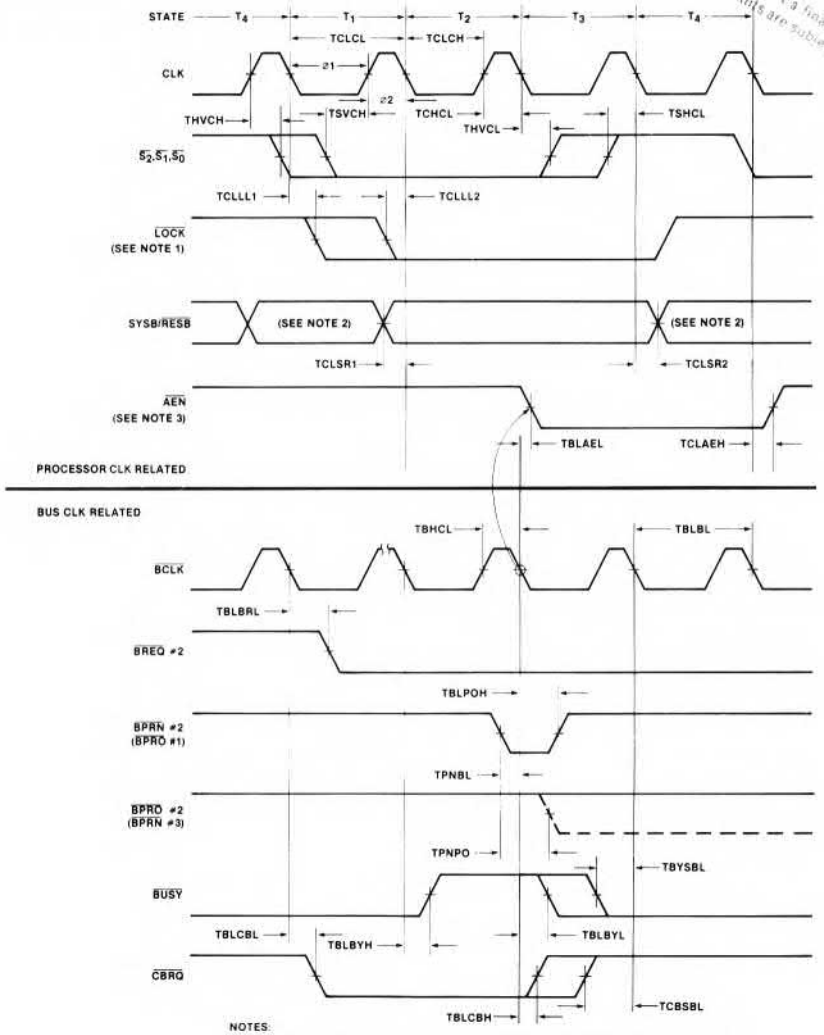
↓↑ Denotes that spec applies to both transitions of the signal.
NOTE 1: BCLK generates the first BPRO wherein subsequent BPRO changes lower in the chain are generated through BPRN.
NOTE 2: Measured at .5V above GND

INITIALIZATION: (INIT can be either pulsed or held low through power up)



PRELIMINARY
Notice: This is not a final specification. Parametric limits are subject to change.

8289 TIMING DIAGRAM



NOTES:
1. LOCK ACTIVE CAN OCCUR DURING ANY T STATE, AS LONG AS THE RELATIONSHIPS SHOWN ABOVE WITH RESPECT TO THE CLK ARE MAINTAINED. LOCK INACTIVE HAS NO CRITICAL TIME AND CAN BE ASYNCHRONOUS.
   - CRQLCK HAS NO CRITICAL TIMING AND IS CONSIDERED AN ASYNCHRONOUS INPUT SIGNAL
2. GLITCHING OF SYSB/RESB PIN IS PERMITTED DURING THIS TIME. AFTER Ø 2 OF T1, AND BEFORE Ø1 OF T4, ONLY CLEAN TRANSITIONS ARE ACCEPTED.
3. AEN LEADING EDGE IS RELATED TO BCLK, TRAILING EDGE TO CLK. THE TRAILING EDGE OF AEN OCCURS AFTER BUS PRIORITY IS LOST.

ADDITIONAL NOTES:

The signals related to CLK are typical processor signals, and do not relate to the depicted sequence of events of the signals referenced to BCLK. The signals shown related to the BCLK represent a hypothetical sequence of events for illustration. Assume 3 bus arbiters of priorities 1, 2 and 3 configured in serial priority resolving scheme as shown in Figure 6. Assume arbiter 1 has the bus and is holding busy low. Arbiter #2 detects its processor wants the bus and pulls low BREQ#2. If BPRN#2 is high (as shown), arbiter #2 will pull low CBRQ line. CBRQ signals to the higher priority arbiter #1 that a lower priority arbiter wants the bus. [A higher priority arbiter would be granted BPRN when it makes the bus request rather than having to wait for another arbiter to release the bus through CBRQ].** Arbiter #1 will relinquish the multi-master system bus when it enters a state not requiring it (see Table 1), by lowering its BPRO#1 (tied to BPRN#2) and releasing BUSY. Arbiter #2 now sees that it has priority from BPRN#2 being low and releases CBRQ. As soon as BUSY signifies the bus is available (high), arbiter #2 pulls BUSY low on next falling edge of BCLK. Note that if arbiter #2 didn't want the bus at the time it received priority, it would pass priority to the next lower priority arbiter by lowering its BPRO #2 [TPNPO].

**Note that even a higher priority arbiter which is acquiring the bus through BPRN will momentarily drop CBRQ until it has acquired the bus.

8237/8237-2
HIGH PERFORMANCE PROGRAMMABLE DMA CONTROLLER

- **Enable/Disable Control of Individual DMA Requests**
- **Four Independent DMA Channels**
- **Independent Autoinitialization of all Channels**
- **Memory-to-Memory Transfers**
- **Memory Block Initialization**
- **Address Increment or Decrement**
- **High Performance: Transfers up to 1.6M Bytes/Second with 5 MHz 8237-2**
- **Directly Expandable to any Number of Channels**
- **End of Process Input for Terminating Transfers**
- **Software DMA Requests**
- **Independent Polarity Control for DREQ and DACK Signals**

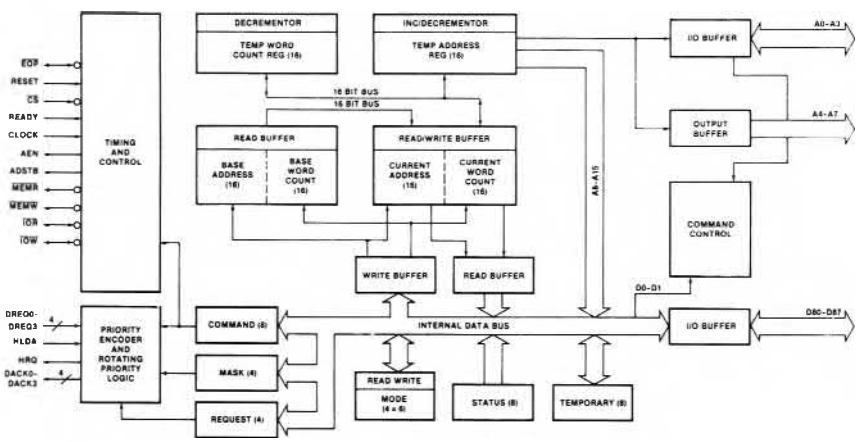
The 8237 Multimode Direct Memory Access (DMA) Controller is a peripheral interface circuit for microprocessor systems. It is designed to improve system performance by allowing external devices to directly transfer information to or from the system memory. Memory-to-memory transfer capability is also provided. The 8237 offers a wide variety of programmable control features to enhance data throughput and system optimization and to allow dynamic reconfiguration under program control.

The 8237 is designed to be used in conjunction with an external 8-bit address register such as the 8282. It contains four independent channels and may be expanded to any number of channels by cascading additional controller chips.

The three basic transfer modes allow programmability of the types of DMA service by the user. Each channel can be individually programmed to Autoinitialize to its original condition following an End of Process (EOP).

Each channel has a full 64K address and word count capability.

The 8237-2 is a 5 MHz selected version of the standard 3 MHz 8237.



BLOCK DIAGRAM

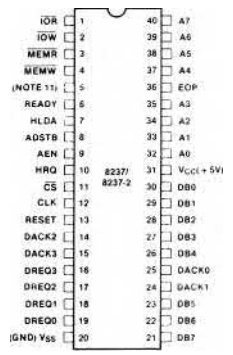


Figure 1. Pin Configuration

PIN DEFINITIONS

V_{CC}: +5 volt supply

V_{SS}: Ground

CLK (Clock, Input)

This input controls the internal operations of the 8237 and its rate of data transfers. The input may be driven at up to 3 MHz for the standard 8237 and up to 5 MHz for the 8237-2.

$\overline{CS}$ (Chip Select, Input)

Chip Select is an active low input used to select the 8237 as an I/O device during the Idle cycle. This allows CPU communication on the data bus.

RESET (Reset, Input)

Reset is an asynchronous active high input which clears the Command, Status, Request and Temporary registers. It also clears the first/last flip/flop and sets the Mask register. Following a Reset the device is in the Idle cycle.

READY (Ready, Input)

Ready is an input used to extend the memory read and write pulses from the 8237 to accommodate slow memories or I/O peripheral devices.

HLDA (Hold Acknowledge, Input)

The active high Hold Acknowledge from the CPU Indicates that control of the system buses have been relinquished.

DREQ0-DREQ3 (DMA Request, Input)

The DMA Request lines are individual asynchronous channel request inputs used by peripheral circuits to obtain DMA service. In Fixed Priority, DREQ0 has the highest priority and DREQ3 has the lowest priority. A request is generated by activating the DREQ line of a channel. DACK will acknowledge the recognition of DREQ signal. Polarity of DREQ is programmable. Reset initializes these lines to active high. DREQ must be maintained until the corresponding DACK goes active.

DB0-DB7 (Data Bus, Input/Output)

The Data Bus lines are bidirectional three-state signals connected to the system data bus. The outputs are enabled in the Program Condition during the I/O Read to output the contents of an Address register, a Status register, the Temporary register or a Word Count register to the CPU. The outputs are disabled and the inputs are read during an I/O Write cycle when the CPU is programming the 8237 control registers. During DMA cycles the most significant 8 bits of the address are output onto the data bus to be strobed into an external latch by ADSTB. In memory-to-memory operations, data from the memory comes into the 8237 on the data bus during the read-from-memory transfer. In the write-to-memory transfer, the data bus outputs place the data into the new memory location.

$\overline{IOR}$ (I/O Read, Input/Output)

I/O Read is a bidirectional active low three-state line. In the Idle cycle, it is an input control signal used by the CPU to read the control registers. In the Active cycle, it is an output control signal used by the 8237 to access data from a peripheral during a DMA Write transfer.

$\overline{IOW}$ (I/O Write, Input/Output)

I/O Write is a bidirectional active low three-state line. In the Idle cycle, it is an input control signal used by the CPU to load information into the 8237. In the Active cycle, it is an output control signal used by the 8237 to load data to the peripheral during a DMA Read transfer.

$\overline{EOP}$ (End of Process, Input/Output)

$\overline{EOP}$ is an active low bidirectional signal. Information concerning the completion of DMA services is available at the bidirectional $\overline{EOP}$ pin. The 8237 allows an external signal to terminate an active DMA service. This is accomplished by pulling the $\overline{EOP}$ input low with an external $\overline{EOP}$ signal. The 8237 also generates a pulse when the terminal count (TC) for any channel is reached. This generates an $\overline{EOP}$ signal which is output through the $\overline{EOP}$ Line. The reception of $\overline{EOP}$, either internal or external, will cause the 8237 to terminate the service, reset the request, and, if Autoinitialize is enabled, to write the base registers to the current registers of that channel. The mask bit and TC bit in the status word will be set for the currently active channel by $\overline{EOP}$ unless the channel is programmed for Autoinitialize. In that case, the mask bit remains clear. During memory-to-memory transfers, $\overline{EOP}$ will be output when the TC for channel 1 occurs. $\overline{EOP}$ should be tied high with a pull-up resistor if it is not used to prevent erroneous end of process inputs.

A0-A3 (Address, Input/Output)

The four least significant address lines are bidirectional three-state signals. In the Idle cycle they are inputs and are used by the 8237 to address the control register to be loaded or read. In the Active cycle they are outputs and provide the lower 4 bits of the output address.

A4-A7 (Address, Output)

The four most significant address lines are three-state outputs and provide 4 bits of address. These lines are enabled only during the DMA service.

HRQ (Hold Request, Output)

This is the Hold Request to the CPU and is used to request control of the system bus. If the corresponding mask bit is clear, the presence of any valid DREQ causes the 8237 to issue the HRQ. After HRQ goes active at least one clock cycle (TCY) must occur before HLDA goes active.

DACK0-DACK3 (DMA Acknowledge, Output)

DMA Acknowledge is used to notify the individual peripherals when one has been granted a DMA cycle. The sense of these lines is programmable. Reset initializes them to active low.

AEN (Address Enable, Output)

This output enables the 8-bit latch containing the upper 8 address bits onto the system address bus. AEN can also be used to disable other system bus drivers during DMA transfers. AEN is active HIGH.

ADSTB (Address Strobe, Output)

The active high Address Strobe is used to strobe the upper address byte into an external latch.

$\overline{\text{MEMR}}$ (Memory Read, Output)

The Memory Read signal is an active low three-state output used to access data from the selected memory location during a DMA Read or a memory-to-memory transfer.

$\overline{\text{MEMW}}$ (Memory Write, Output)

The Memory Write signal is an active low three-state output used to write data to the selected memory location during a DMA Write or a memory-to-memory transfer.

FUNCTIONAL DESCRIPTION

The 8237 block diagram includes the major logic blocks and all of the internal registers. The data interconnection paths are also shown. Not shown are the various control signals between the blocks. The 8237 contains 344 bits of internal memory in the form of registers. Figure 2 lists these registers by name and shows the size of each. A detailed description of the registers and their functions can be found under Register Description.

Name	Size	Number
Base Address Registers	16 bits	4
Base Word Count Registers	16 bits	4
Current Address Registers	16 bits	4
Current Word Count Registers	16 bits	4
Temporary Address Register	16 bits	1
Temporary Word Count Register	16 bits	1
Status Register	8 bits	1
Command Register	8 bits	1
Temporary Register	8 bits	1
Mode Registers	6 bits	4
Mask Register	4 bits	1
Request Register	4 bits	1

Figure 2. 8237 Internal Registers

The 8237 contains three basic blocks of control logic. The Timing Control block generates internal timing and external control signals for the 8237. The Program Command Control block decodes the various commands given to the 8237 by the microprocessor prior to servicing a DMA Request. It also decodes the Mode Control word used to select the type of DMA during the servicing. The Priority Encoder block resolves priority contention between DMA channels requesting service simultaneously.

The Timing Control block derives internal timing from the clock input. In 8237 systems this input will usually be the $\phi 2$ TTL clock from an 8224 or CLK from an 8085A. However, any appropriate system clock will suffice.

DMA OPERATION

The 8237 is designed to operate in two major cycles. These are called Idle and Active cycles. Each device cycle is made up of a number of states. The 8237 can assume seven separate states, each composed of one full clock period. State I (SI) is the inactive state. It is entered when the 8237 has no valid DMA requests pending. While in SI, the DMA controller is inactive but may be in the Program Condition, being programmed by the processor. State O (SO) is the first state of a DMA service. The 8237 has requested a hold but the processor has not yet returned an acknowledge. An acknowledge from the CPU will signal that transfers may begin. S1, S2, S3 and S4 are the working states of the DMA service. If more time is needed to complete a transfer than is available with normal timing, wait states (SW) can be inserted between S2 or S3 and S4 by the use of the Ready line on the 8237.

Memory-to-memory transfers require a read-from and a write-to-memory to complete each transfer. The states, which resemble the normal working states, use two digit numbers for identification. Eight states are required for a single transfer. The first four states (S11, S12, S13, S14) are used for the read-from-memory half and the last four states (S21, S22, S23, S24) for the write-to-memory half of the transfer.

IDLE CYCLE

When no channel is requesting service, the 8237 will enter the Idle cycle and perform "SI" states. In this cycle the 8237 will sample the DREQ lines every clock cycle to determine if any channel is requesting a DMA service. The device will also sample $\overline{\text{CS}}$, looking for an attempt by the microprocessor to write or read the internal registers of the 8237. When $\overline{\text{CS}}$ is low and HRQ is low, the 8237 enters the Program Condition. The CPU can now establish, change or inspect the internal definition of the part by reading from or writing to the internal registers. Address lines A0-A3 are inputs to the device and select which registers will be read or written. The $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ lines are used to select and time reads or writes. Due to the number and size of the internal registers, an internal flip-flop is used to generate an additional bit of address. This bit is used to determine the upper or lower byte of the 16-bit Address and Word Count registers. The flip-flop is reset by Master Clear or Reset. A separate software command can also reset this flip-flop.

Special software commands can be executed by the 8237 in the Program Condition. These commands are decoded as sets of addresses with the $\overline{\text{CS}}$ and $\overline{\text{IOW}}$. The commands do not make use of the data bus. Instructions include Clear First/Last Flip-flop and Master Clear.

ACTIVE CYCLE

When the 8237 is in the Idle cycle and a channel requests a DMA service, the device will output an HRQ to the microprocessor and enter the Active cycle. It is in this cycle that the DMA service will take place, in one of four modes:

Single Transfer Mode — In Single Transfer mode the device is programmed to make one transfer only. The

word count will be decremented and the address decremented or incremented following each transfer. When the word count goes to zero, a Terminal Count (TC) will cause an Autoinitialize if the channel has been programmed to do so.

DREQ must be held active until DACK becomes active in order to be recognized. If DREQ is held active throughout the single transfer, HRQ will go inactive and release the bus to the system. It will again go active and, upon receipt of a new HLDA, another single transfer will be performed. In 8080A/8085A systems this will ensure one full machine cycle execution between DMA transfers. Details of timing between the 8237 and other bus control protocols will depend upon the characteristics of the microprocessor involved.

Block Transfer Mode — In Block Transfer mode the device is activated by DREQ to continue making transfers during the service until a TC, caused by word count going to zero, or an external End of Process ($\overline{EOP}$) is encountered. DREQ need only be held active until DACK becomes active. Again, an Autoinitialization will occur at the end of the service if the channel has been programmed for it.

Demand Transfer Mode — In Demand Transfer mode the device is programmed to continue making transfers until a TC or external $\overline{EOP}$ is encountered or until DREQ goes inactive. Thus transfers may continue until the I/O device has exhausted its data capacity. After the I/O device has had a chance to catch up, the DMA service is re-established by means of a DREQ. During the time between services when the microprocessor is allowed to operate, the intermediate values of address and word count are stored in the 8237 Current Address and Current Word Count registers. Only an $\overline{EOP}$ can cause an Autoinitialize at the end of the service. $\overline{EOP}$ is generated either by TC or by an external signal.

Cascade Mode — This mode is used to cascade more than one 8237 together for simple system expansion. The HRQ and HLDA signals from the additional 8237 are connected to the DREQ and DACK signals of a channel of the initial 8237. This allows the DMA requests of the additional device to propagate through the priority network circuitry of the preceding device. The priority chain is preserved and the new device must wait for its turn to acknowledge requests. Since the cascade channel in the initial device is used only for prioritizing the additional device, it does not output any address or control signals of its own. These would conflict with the outputs of the active channel in the added device. The 8237 will respond to DREQ and DACK but all other outputs except HRQ will be disabled.

Figure 3 shows two additional devices cascaded into an initial device using two of the previous channels. This forms a two level DMA system. More 8237s could be added at the second level by using the remaining channels of the first level. Additional devices can also be added by cascading into the channels of the second level devices, forming a third level.

TRANSFER TYPES

Each of the three active transfer modes can perform three different types of transfers. These are Read, Write and Verify. Write transfers move data from an I/O device to the memory by activating $\overline{MEMW}$ and $\overline{IOR}$. Read transfers move data from memory to an I/O device by activating MEMR and IOW. Verify transfers are pseudo transfers. The 8237 operates as in Read or Write transfers generating addresses, and responding to EOP, etc. However, the memory and I/O control lines all remain inactive.

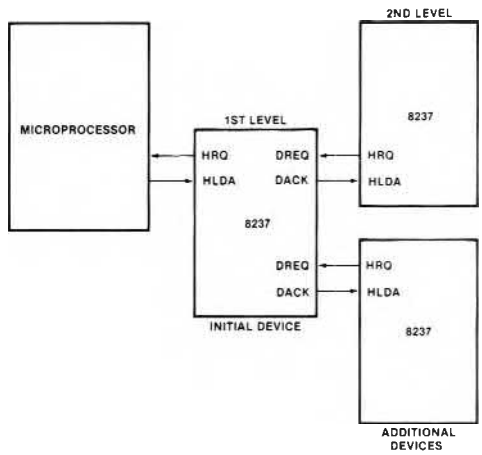


Figure 3. Cascaded 8237s

Memory-to-Memory — To perform block moves of data from one memroy address space to another with a minimum of program effort and time, the 8237 includes a memory-to-memory transfer feature. Programming a bit in the Command register selects channels 0 and 1 to operate as memory-to-memory transfer channels. The transfer is initiated by setting the software DREQ for channel 0. The 8237 requests a DMA service in the normal manner. After HLDA is true, the device, using eight-state transfers in Block Transfer mode, reads data from the memory. The channel 0 Current Address register is the source for the address used and is decremented or incremented in the normal manner. The data byte read from the memory is stored in the 8237 internal Temporary register. Channel 1 then writes the data from the Temporary register to memory using the address in its Current Address register and incrementing or decrementing it in the normal manner. The channel 1 Current Word Count is decremented. When the word count of channel 1 goes to zero, a TC is generated causing an $\overline{EOP}$ output, terminating the service.

Channel 0 may be programmed to retain the same address for all transfers. This allows a single word to be written to a block of memory.

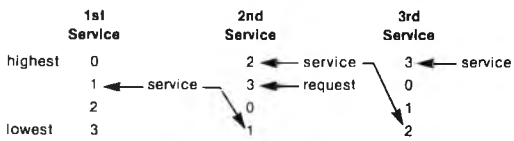
The 8237 will respond to external $\overline{EOP}$ signals during memory-to-memory transfers. Data comparators in block search schemes may use this input to terminate the service when a match is found. The timing of memory-to-memory transfers is found in Diagram 4. Memory-to-memory operations can be detected as an active AEN with no DACK outputs.

Autoinitialize — By programming a bit in the Mode register, a channel may be set up as an Autoinitialize

channel. During Autoinitialize initialization, the original values of the Current Address and Current Word Count registers are automatically restored from the Base Address and Base Word Count registers of that channel following $\overline{EOP}$. The base registers are loaded simultaneously with the current registers by the microprocessor and remain unchanged throughout the DMA service. The mask bit is not set when the channel is in Autoinitialize. Following Autoinitialize the channel is ready to perform another service without CPU intervention.

Priority — The 8237 has two types of priority encoding available as software selectable options. The first is Fixed Priority which fixes the channels in priority order based upon the descending value of their number. The channel with the lowest priority is 3 followed by 2, 1 and the highest priority channel, 0. After the recognition of any one channel for service, the other channels are prevented from interferring with that service until it is completed.

The second scheme is Rotating Priority. The last channel to get service becomes the lowest priority channel with the others rotating accordingly.



With Rotating Priority in a single chip DMA system, any device requesting service is guaranteed to be recognized after no more than three higher priority services have occurred. This prevents any one channel from monopolizing the system.

Compressed Timing — In order to achieve even greater throughput where system characteristics permit, the 8237 can compress the transfer time to two clock cycles. From Timing Diagram 3 it can be seen that state S3 is used to extend the access time of the read pulse. By removing state S3, the read pulse width is made equal to the write pulse width and a transfer consists only of state S2 to change the address and state S4 to perform the read/write. S1 states will still occur when A8–A15 need updating (see Address Generation). Timing for compressed transfers is found in Diagram 6.

Address Generation — In order to reduce pin count, the 8237 multiplexes the eight higher order address bits on the data lines. State S1 is used to output the higher order address bits to an external latch from which they may be placed on the address bus. The falling edge of Address Strobe (ADSTB) is used to load these bits from the data lines to the latch. Address Enable (AEN) is used to enable the bits onto the address bus through a three-state enable. The lower order address bits are output by the 8237 directly. Lines A0–A7 should be connected to the address bus. Timing Diagram 3 shows the time relationships between CLK, AEN, ADSTB, DB0–DB7 and A0–A7.

During Block and Demand Transfer mode services, which include multiple transfers, the addresses generated will be sequential. For many transfers the data held in the external address latch will remain the same. This data need only change when a carry or borrow from A7 to A8 takes place in the normal sequence of addresses. To save time and speed transfers, the 8237 executes S1 states only when updating of A8–A15 in the latch is necessary. This means for long services, S1 states may occur only once every 256 transfers, a savings of 255 clock cycles for each 256 transfers.

REGISTER DESCRIPTION

Current Address Register — Each channel has a 16-bit Current Address register. This register holds the value of the address used during DMA transfers. The address is automatically incremented or decremented after each transfer and the intermediate values of the address are stored in the Current Address register during the transfer. This register is written or read by the microprocessor in successive 8-bit bytes. It may also be reinitialized by an Autoinitialize back to its original value. Autoinitialize takes place only after an $\overline{EOP}$.

Current Word Register — Each channel has a 16-bit Current Word Count register. This register holds the number of transfers to be performed. The word count is decremented after each transfer. The intermediate value of the word count is stored in the register during the transfer. When the value in the register goes to zero, a TC will be generated. This register is loaded or read in successive 8-bit bytes by the microprocessor in the Program Condition. Following the end of a DMA service it may also be reinitialized by an Autoinitialization back to its original value. Autoinitialize can occur only when an $\overline{EOP}$ occurs.

Base Address and Base Word Count Registers — Each channel has a pair of Base Address and Base Word Count registers. These 16-bit registers store the original value of their associated current registers. During Autoinitialize these values are used to restore the current registers to their original values. The base registers are written simultaneously with their corresponding current register in 8-bit bytes in the Program Condition by the microprocessor. These registers cannot be read by the microprocessor.

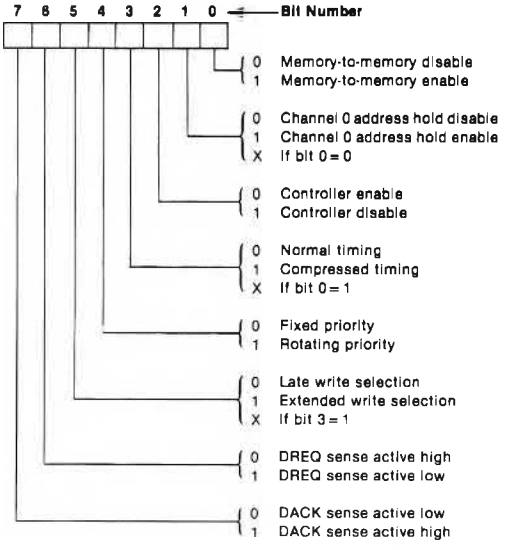
Command Register — This 8-bit register controls the operation of the 8237. It is programmed by the microprocessor in the Program Condition and is cleared by Reset. The following table lists the function of the command bits. See Figure 6 for address coding.

Mode Register — Each channel has a 6-bit Mode register associated with it. When the register is being written to by the microprocessor in the Program Condition, bits 0 and 1 determine which channel Mode register is to be written.

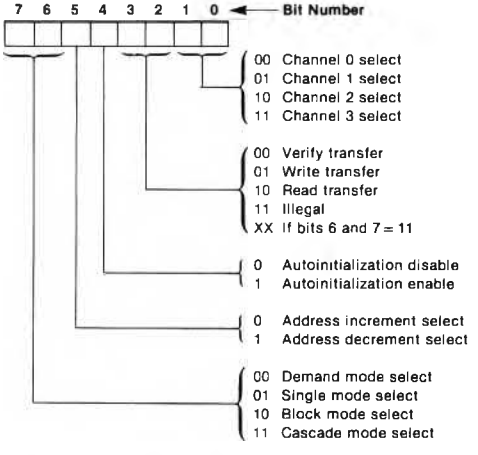
Request Register — The 8237 can respond to requests for DMA service which are initiated by software as well as by a DREQ. Each channel has a request bit associated with it in the 4-bit Request register. These are non-

maskable and subject to prioritization by the Priority Encoder network. Each register bit is set or reset separately under software control or is cleared upon generation of a TC or external $\overline{EOP}$. The entire register is cleared by a Reset. To set or reset a bit, the software loads the proper form of the data word. See Figure 4 for address coding.

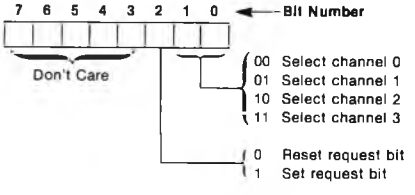
Command Register



Mode Register

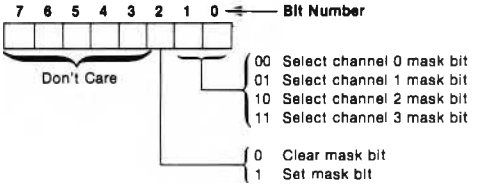


Request Register

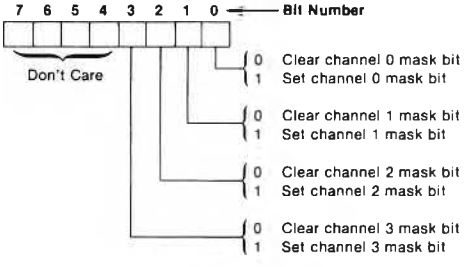


Software requests will be serviced only if the channel is in Block mode. When initiating a memory-to-memory transfer, the software request for channel 0 should be set.

Mask Register — Each channel has associated with it a mask bit which can be set to disable the incoming DREQ. Each mask bit is set when its associated channel produces an $\overline{EOP}$ if the channel is not programmed for Autoinitialize. Each bit of the 4-bit Mask register may also be set or cleared separately under software control. The entire register is also set by a Reset. This disables all DMA requests until a clear Mask register instruction allows them to occur. The instruction to separately set or clear the mask bits is similar in form to that used with the Request register. See Figure 4 for instruction addressing.



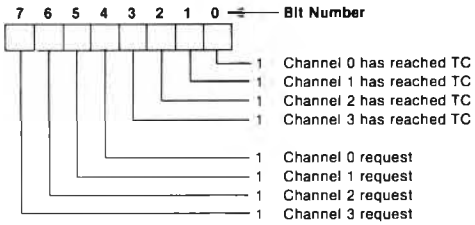
All four bits of the Mask register may also be written with a single command.



Register	Operation	Signals						
		$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A3	A2	A1	A0
Command	Write	0	1	0	1	0	0	0
Mode	Write	0	1	0	1	0	1	1
Request	Write	0	1	0	1	0	0	1
Mask	Set/Reset	0	1	0	1	0	1	0
Mask	Write	0	1	0	1	1	1	1
Temporary	Read	0	0	1	1	1	0	1
Status	Read	0	0	1	1	0	0	0

Figure 4. Definition of Register Codes

Status Register — The Status register is available to be read out of the 8237 by the microprocessor. It contains information about the status of the devices at this point. This information includes which channels have reached a terminal count and which channels have pending DMA requests. Bits 0-3 are set every time a TC is reached by that channel or an external $\overline{EOP}$ is applied. These bits are cleared upon Reset and on each Status Read. Bits 4-7 are set whenever their corresponding channel is requesting service.



Temporary Register — The Temporary register is used to hold data during memory-to-memory transfers. Following the completion of the transfers, the last word moved can be read by the microprocessor in the Program Condition. The Temporary register always contains the last byte transferred in the previous memory-to-memory operation, unless cleared by a Reset.

Software Commands — These are additional special software commands which can be executed in the Program Condition. They do not depend on any specific bit pattern on the data bus. The two software commands are:

Clear First/Last Flip-Flop: This command is executed prior to writing or reading new address or word count information to the 8237. This initializes the flip-flop to a known state so that subsequent accesses to register contents by the microprocessor will address upper and lower bytes in the correct sequence.

Master Clear: This software instruction has the same effect as the hardware Reset. The Command, Status, Request, Temporary, and Internal First/Last Flip-Flop registers are cleared and the Mask register is set. The 8237 will enter the Idle cycle.

Figure 5 lists the address codes for the software commands:

Signals						Operation
A3	A2	A1	A0	$\overline{IOR}$	$\overline{IOW}$	
1	0	0	0	0	1	Read Status Register
1	0	0	0	1	0	Write Command Register
1	0	0	1	0	1	Illegal
1	0	0	1	1	0	Write Request Register
1	0	1	0	0	1	Illegal
1	0	1	0	1	0	Write Single Mask Register Bit
1	0	1	1	0	1	Illegal
1	0	1	1	1	0	Write Mode Register
1	1	0	0	0	1	Illegal
1	1	0	0	1	0	Clear Byte Pointer Flip/Flop
1	1	0	1	0	1	Read Temporary Register
1	1	0	1	1	0	Master Clear
1	1	1	0	0	1	Illegal
1	1	1	0	1	0	Illegal
1	1	1	1	0	1	Illegal
1	1	1	1	1	0	Write All Mask Register Bits

Figure 5. Software Command Codes

Channel	Register	Operation	Signals							Internal Flip-Flop	Data Bus DB0-DB7
			$\overline{CS}$	$\overline{IOR}$	$\overline{IOW}$	A3	A2	A1	A0		
0	Base and Current Address	Write	0	1	0	0	0	0	0	0	A0-A7
			0	1	0	0	0	0	0	1	A8-A15
	Current Address	Read	0	0	1	0	0	0	0	0	A0-A7
			0	0	1	0	0	0	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	0	0	1	0	W0-W7
			0	1	0	0	0	0	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	0	0	1	0	W)0-W7
			0	0	1	0	0	0	1	1	W8-W15
1	Base and Current Address	Write	0	1	0	0	0	1	0	0	A0-A7
			0	1	0	0	0	1	0	1	A8-A15
	Current Address	Read	0	0	1	0	0	1	0	0	A0-A7
			0	0	1	0	0	1	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	W0-W7
			0	1	0	0	0	1	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	0	1	1	0	W)0-W7
			0	0	1	0	0	1	1	1	W8-W15
2	Base and Current Address	Write	0	1	0	0	1	0	0	0	A0-A7
			0	1	0	0	1	0	0	1	A8-A15
	Current Address	Read	0	0	1	0	1	0	0	0	A0-A7
			0	0	1	0	1	0	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	1	0	1	0	W0-W7
			0	1	0	0	1	0	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	1	0	1	0	W)0-W7
			0	0	1	0	1	0	1	1	W8-W15
3	Base and Current Address	Write	0	1	0	0	1	1	0	0	A0-A7
			0	1	0	0	1	1	0	1	A8-A15
	Current Address	Read	0	0	1	0	1	1	0	0	A0-A7
			0	0	1	0	1	1	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	1	1	1	0	W0-W7
			0	1	0	0	1	1	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	1	1	1	0	W)0-W7
			0	0	1	0	1	1	1	1	W8-W15

Figure 6. Word Count and Address Register Command Codes

APPLICATION INFORMATION

Figure 7 shows a convenient method for configuring a DMA system with the 8237 controller and an 8080A/8085A microprocessor system. The multimode DMA controller issues a HRQ to the processor whenever there is at least one valid DMA request from a peripheral device. When the processor replies with a HLDA signal, the 8237 takes control of the address bus, the data bus and the control bus. The address for the first transfer operation comes out in two bytes — the least significant 8 bits on the eight address outputs and the most significant 8 bits on the data bus. The contents of the data bus are then latched into the 8282 8-bit latch to complete the full 16 bits of the address bus. The 8282 is a high speed, 8-bit, three-state latch in a 20-pin package. After the initial transfer takes place, the latch is updated only after a carry or borrow is generated in the least significant address byte. Four DMA channels are provided when one 8237 is used.

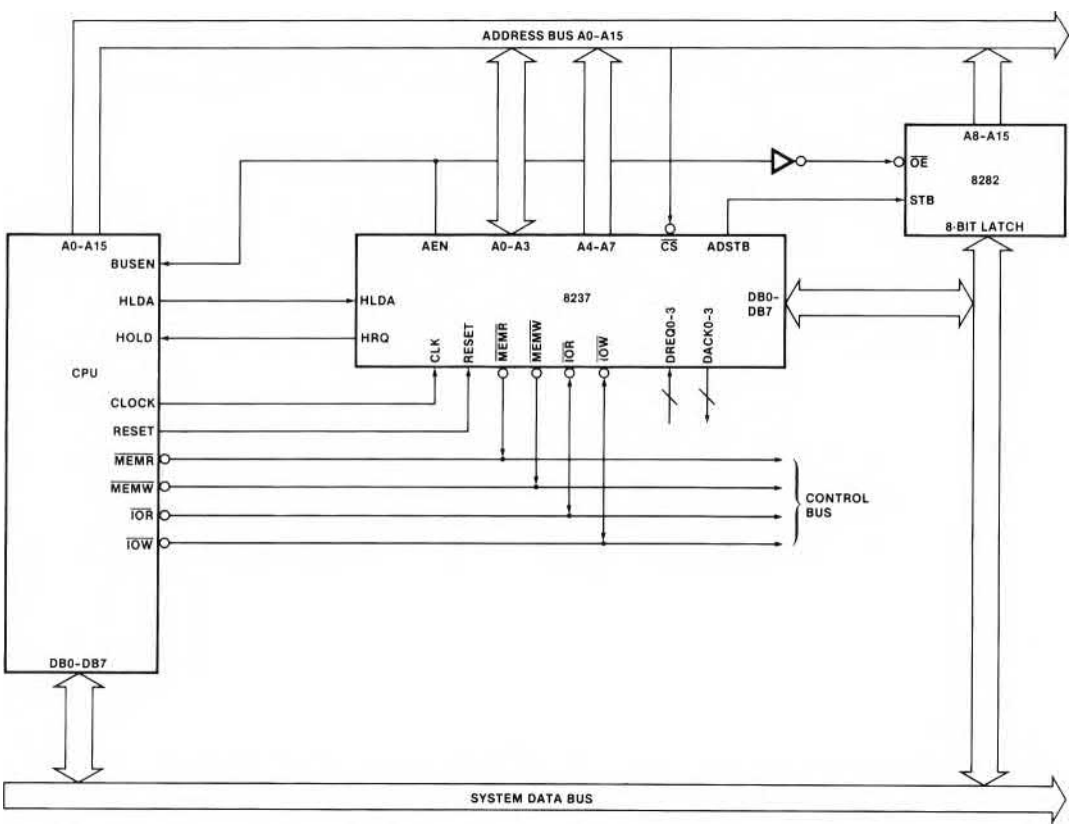


Figure 7

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . – 65°C to + 150°C
Voltage on any Pin with
Respect to Ground . . . . . – 0.5 to 7V
Power Dissipation . . . . . 1.5 Watt

*COMMENT: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

D.C. CHARACTERISTICS

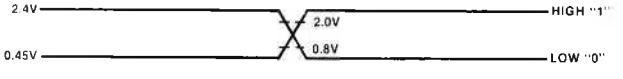
$T_A = 0°C$ to $70°C$, $V_{CC} = 5.0V \pm 5\%$, GND = 0V

Symbol	Parameter	Min.	Typ.(1)	Max.	Unit	Test Conditions
V_{OH}	Output HIGH Voltage	2.4			V	$I_{OH} = -200\ \mu A$
		3.3			V	$I_{OH} = -100\ \mu A$ (HRQ Only)
V_{OL}	Output LOW Voltage			0.4	V	$I_{OL} = 3.2$ mA
V_{IH}	Input HIGH Voltage	2.0		$V_{CC} + 0.5$	V	
V_{IL}	Input LOW Voltage	– 0.5		0.8	V	
I_{LI}	Input Load Current			± 10	μA	$V_{SS} \leq V_I \leq V_{CC}$
I_{LO}	Output Leakage Current			± 10	μA	$V_{CC} \leq V_O \leq V_{SS} + 0.40$
I_{CC}	V_{CC} Supply Current		65	130	mA	$T_A = +25°C$
			75	150	mA	$T_A = 0°C$
C_O	Output Capacitance		4	8	pF	fc = 1.0 MHz, Inputs = 0V
C_I	Input Capacitance		8	15	pF	
C_{IO}	I/O Capacitance		10	18	pF	

Notes:

1. Typical values are for $T_A = 25°C$, nominal supply voltage and nominal processing parameters.
2. Input timing parameters assume transition times of 20 ns or less. Waveform measurement points for both input and output signals are 2.0V for HIGH and 0.8V for LOW, unless otherwise noted.
3. Output loading is 1 TTL gate plus 50 pF capacitance, unless otherwise noted.
4. The net $\overline{IOW}$ or $\overline{MEMW}$ Pulse width for normal write will be TCY-100 ns and for extended write will be 2TCY-100 ns. The net $\overline{IOR}$ or $\overline{MEMR}$ pulse width for normal read will be 2TCY-50 ns and for compressed read will be TCY-50 ns.
5. TDQ is specified for two different output HIGH levels. TDQ1 is measured at 2.0V. TDQ2 is measured at 3.3V. The value for TDQ2 assumes an external 3.3 kΩ pull-up resistor connected from HRQ to V_{CC}.
6. DREQ should be held active until DACK is returned.
7. DREQ and DACK signals may be active high or active low. Timing diagrams assume the active high mode.
8. Output loading on the data bus is 1 TTL gate plus 100 pF capacitance.
9. Successive read and/or write operations by the external processor to program or examine the controller must be timed to allow at least 600 ns for the 8237 and at least 400 ns for the 8237-2 as recovery time between active read or write pulses.
10. Parameters are listed in alphabetical order.
11. Pin 5 is an input that should always be at a logic high level. An internal pull-up resistor will establish a logic high when the pin is left floating. Alternatively, pin 5 may be tied to V_{CC}.

A.C. TEST WAVEFORM



A.C. CHARACTERISTICS: DMA (MASTER) MODE

$T_A = 0°C$ to $70°C$, $V_{CC} = 5.0V \pm 5\%$, GND = 0V

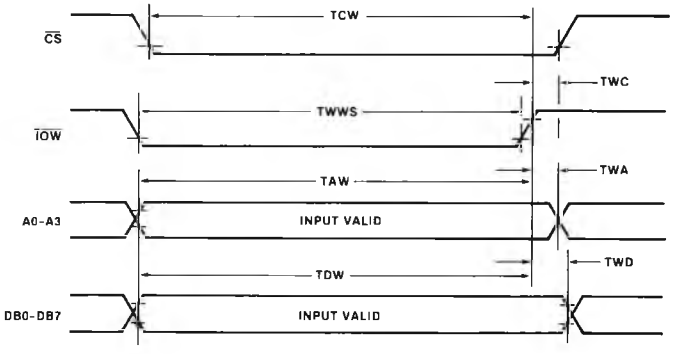
Symbol	Parameter	8237		8237-2		Unit
		Min.	Max.	Min.	Max.	
TAEL	AEN HIGH from CLK LOW (S1) Delay Time		300		200	ns
TAET	AEN LOW from CLK HIGH (S1) Delay Time		200		130	ns
TAFAB	ADR Active to Float Delay from CLK HIGH		150		90	ns
TAFC	$\overline{READ}$ or $\overline{WRITE}$ Float from CLK HIGH		150		120	ns
TAFDB	DB Active to Float Delay from CLK HIGH		250		170	ns
TAHR	ADR from $\overline{READ}$ HIGH Hold Time	TCY-100		TCY-100		ns
TAHS	DB from ADSTB LOW Hold Time	50		30		ns
TAHW	ADR from $\overline{WRITE}$ HIGH Hold Time	TCY-50		TCY-50		ns
TAK	DACK Valid from CLK LOW Delay Time		250		170	ns
	$\overline{EOP}$ HIGH from CLK HIGH Delay Time		250		170	ns
	$\overline{EOP}$ LOW to CLK HIGH Delay Time		250		100	ns
TASM	ADR Stable from CLK HIGH		250		170	ns
TASS	DB to ADSTB LOW Setup Time	100		100		ns
TCH	Clock High Time (Transitions ≤10 ns)	120		70		ns
TCL	Clock LOW Time (Transitions ≤10 ns)	150		50		ns
TCY	CLK Cycle Time	320		200		ns
TDCL	CLK HIGH to $\overline{READ}$ or $\overline{WRITE}$ LOW Delay (Note 4)		270		190	ns
TDCTR	$\overline{READ}$ HIGH from CLK HIGH (S4) Delay Time (Note 4)		270		190	ns
TDCTW	$\overline{WRITE}$ HIGH from CLK HIGH (S4) Delay Time (Note 4)		200		130	ns
TDQ1	HRQ Valid from CLK HIGH Delay Time (Note 5)		160		120	ns
TDQ2			250		120	ns
TEPS	$\overline{EOP}$ LOW from CLK LOW Setup Time	60		40		ns
TEPW	$\overline{EOP}$ Pulse Width	300		220		ns
TFAAB	ADR Float to Active Delay from CLK HIGH		250		170	ns
TFAC	$\overline{READ}$ or $\overline{WRITE}$ Active from CLK HIGH		200		150	ns
TFADB	DB Float to Active Delay from CLK HIGH		300		200	ns
THS	HCDA Valid to CLK HIGH Setup Time	100		75		ns
TIDH	Input Data from $\overline{MEMR}$ HIGH Hold Time	0		0		ns
TIDS	Input Data to $\overline{MEMR}$ HIGH Setup Time	250		170		ns
TODH	Output Data from $\overline{MEMW}$ HIGH Hold Time	20		10		ns
TODV	Output Data Valid to $\overline{MEMW}$ HIGH	200		130		ns
TQS	DREQ to CLK LOW (SI, S4) Setup Time	0		0		ns
TRH	CLK to READY LOW Hold Time	20		20		ns
TRS	READY to CLK LOW Setup Time	100		75		ns
TSTL	ADSTB HIGH from CLK HIGH Delay Time		200		130	ns
TSTT	ADSTB LOW from CLK HIGH Delay Time		140		90	ns

A.C. CHARACTERISTICS: PERIPHERAL (SLAVE) MODE

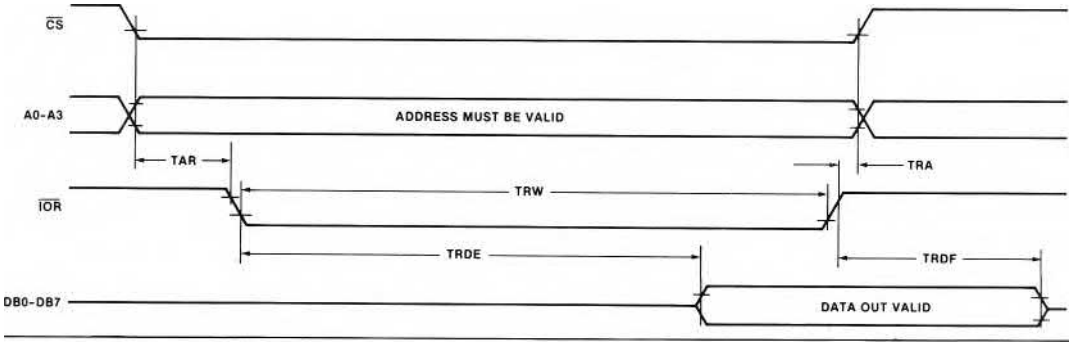
$T_A = 0°C$ to $70°C$, $V_{CC} = 5.0V \pm 5\%$, GND = 0V

Symbol	Parameter	8237		8237-2		Unit
		Min.	Max.	Min.	Max.	
TAR	ADR Valid or $\overline{CS}$ LOW to $\overline{READ}$ LOW	50		50		ns
TAW	ADR Valid to $\overline{WRITE}$ HIGH Setup Time	200		160		ns
TCW	CS LOW to $\overline{WRITE}$ HIGH Setup Time	200		160		ns
TDW	Data Valid to $\overline{WRITE}$ HIGH Setup Time	200		160		ns
TRA	ADR or CS Hold from $\overline{READ}$ HIGH	0		0		ns
TRDE	Data Access from $\overline{READ}$ LOW (Note 8)		200		140	ns
TRDF	DB Float Delay from $\overline{READ}$ HIGH	20	100	0	70	ns
TRSTD	Power Supply HIGH to RESET LOW Setup Time	500		500		μs
TRSTS	RESET to First $\overline{IOWR}$	2TCY		2TCY		ns
TRSTW	RESET Pulse Width	300		300		ns
TRW	$\overline{READ}$ Width	300		200		ns
TWA	ADR from $\overline{WRITE}$ HIGH Hold Time	20		0		ns
TWC	$\overline{CS}$ HIGH from $\overline{WRITE}$ HIGH Hold Time	20		0		ns
TWD	Data from $\overline{WRITE}$ HIGH Hold Time	30		10		ns
TWWS	Write Width	200		160		ns

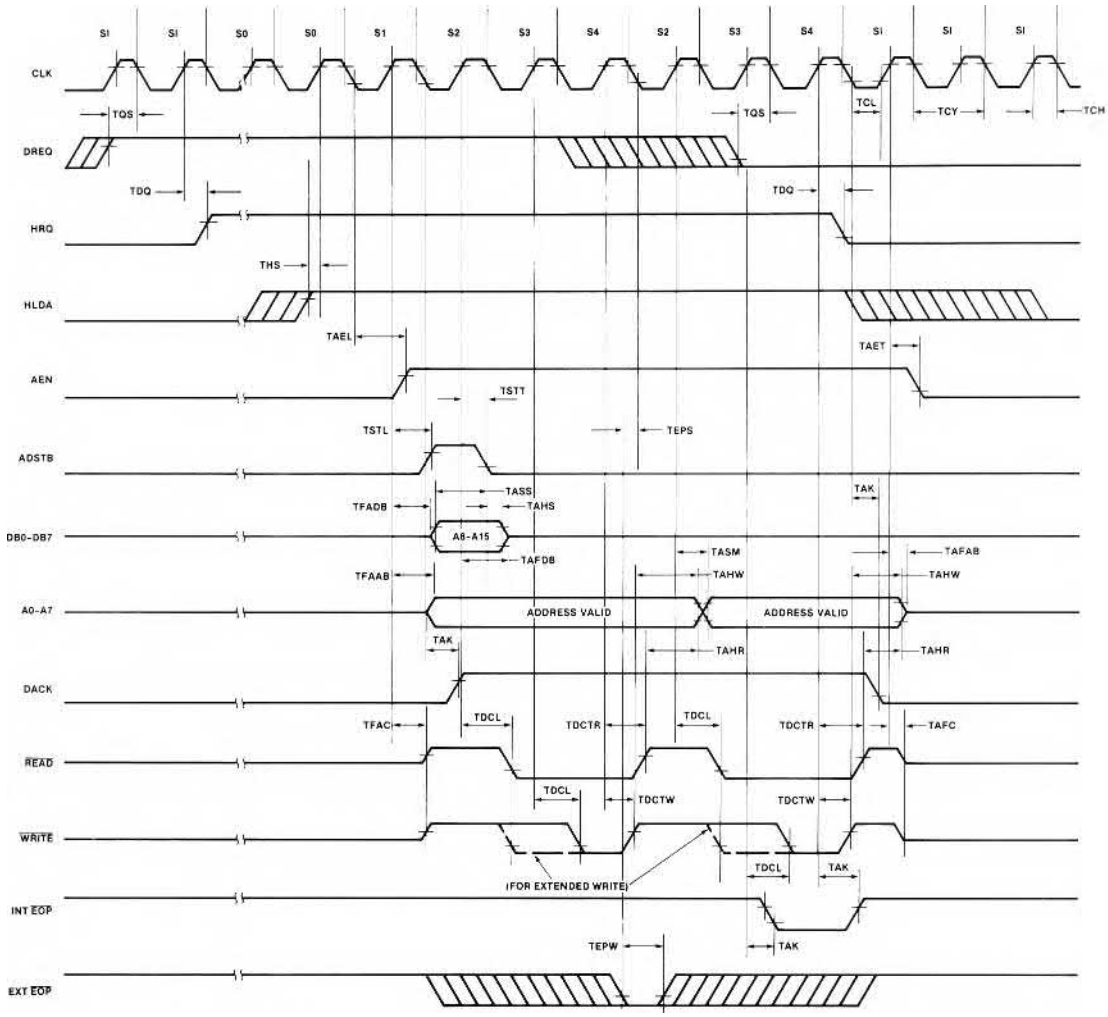
TIMING DIAGRAM #1 — SLAVE MODE WRITE TIMING



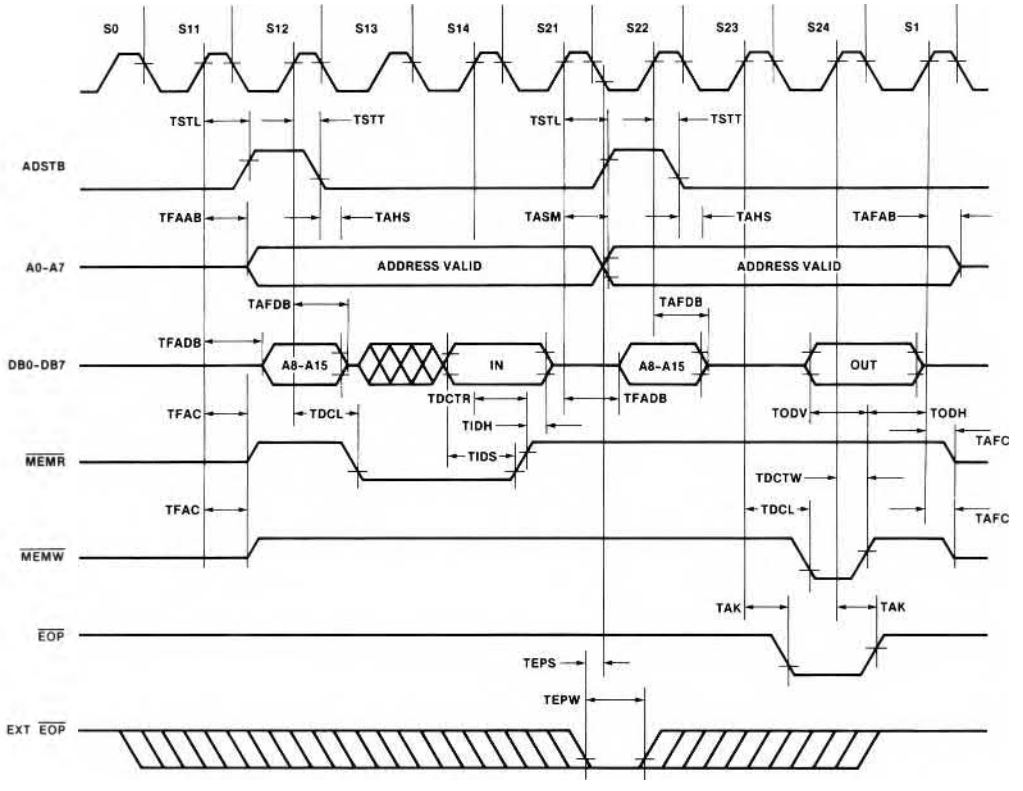
TIMING DIAGRAM #2 — SLAVE MODE READ TIMING



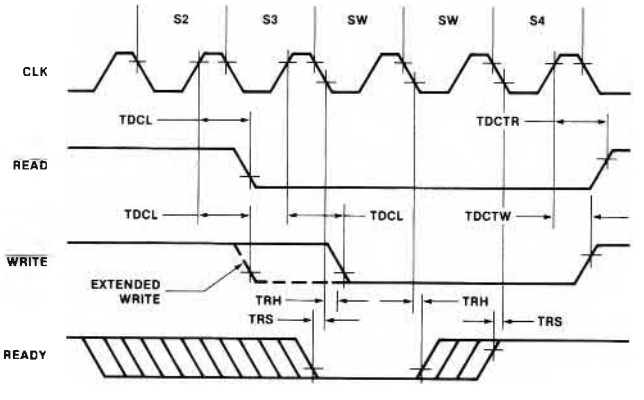
TIMING DIAGRAM #3 — DMA TRANSFER TIMING



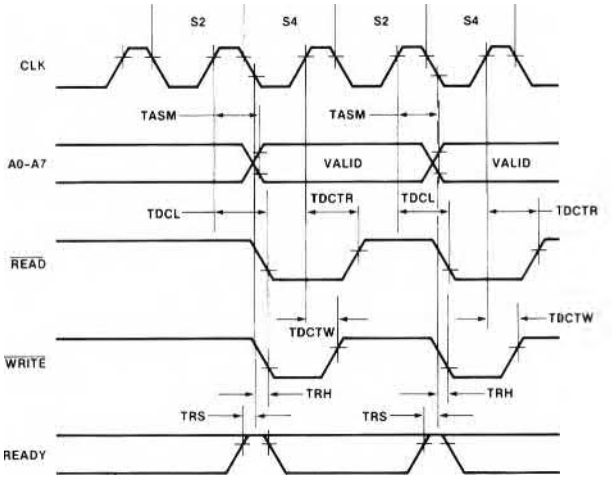
TIMING DIAGRAM #4 — MEMORY TO MEMORY TRANSFER TIMING



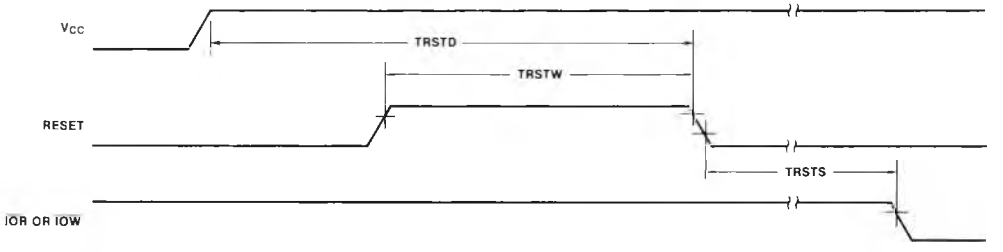
TIMING DIAGRAM #5 — READY TIMING



TIMING DIAGRAM #6 — COMPRESSED TRANSFER TIMING



TIMING DIAGRAM #7 — RESET TIMING



8259A/8259A-2/8259A-8
PROGRAMMABLE INTERRUPT CONTROLLER

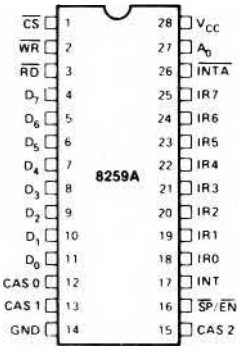
- **8086/8088 Compatible**
- **MCS-80/85™ Compatible**
- **Eight-Level Priority Controller**
- **Expandable to 64 Levels**
- **Programmable Interrupt Modes**
- **Individual Request Mask Capability**
- **Single +5V Supply (No Clocks)**
- **28-Pin Dual-In-Line Package**

The Intel® 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. It is packaged in a 28-pin DIP, uses NMOS technology and requires a single +5V supply. Circuitry is static, requiring no clock input.

The 8259A is designed to minimize the software and real time overhead in handling multi-level priority interrupts. It has several modes, permitting optimization for a variety of system requirements.

The 8259A is fully upward compatible with the Intel® 8259. Software originally written for the 8259 will operate the 8259A in all 8259 equivalent modes (MCS-80/85, Non-Buffered, Edge Triggered).

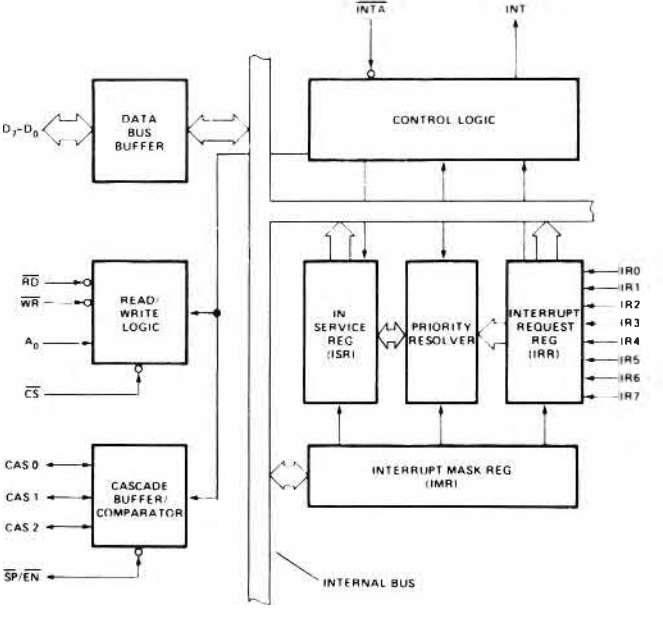
PIN CONFIGURATION



PIN NAMES

D_7 D_0	DATA BUS (BI DIRECTIONAL)
$\overline{RD}$	READ INPUT
$\overline{WR}$	WRITE INPUT
A_0	COMMAND SELECT ADDRESS
$\overline{CS}$	CHIP SELECT
CAS2 CAS0	CASCADE LINES
$\overline{SP}/\overline{EN}$	SLAVE PROGRAM ENABLE BUFFER
INT	INTERRUPT OUTPUT
$\overline{INTA}$	INTERRUPT ACKNOWLEDGE INPUT
IR0-IR7	INTERRUPT REQUEST INPUTS

BLOCK DIAGRAM



INTERRUPTS IN MICROCOMPUTER SYSTEMS

Microcomputer system design requires that I/O devices such as keyboards, displays, sensors and other components receive servicing in an efficient manner so that large amounts of the total system tasks can be assumed by the microcomputer with little or no effect on throughput.

The most common method of servicing such devices is the *Polled* approach. This is where the processor must test each device in sequence and in effect "ask" each one if it needs servicing. It is easy to see that a large portion of the main program is looping through this continuous polling cycle and that such a method would have a serious, detrimental effect on system throughput, thus limiting the tasks that could be assumed by the microcomputer and reducing the cost effectiveness of using such devices.

A more desirable method would be one that would allow the microprocessor to be executing its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is complete, however, the processor would resume exactly where it left off.

This method is called *Interrupt*. It is easy to see that system throughput would drastically increase, and thus more tasks could be assumed by the microcomputer to further enhance its cost effectiveness.

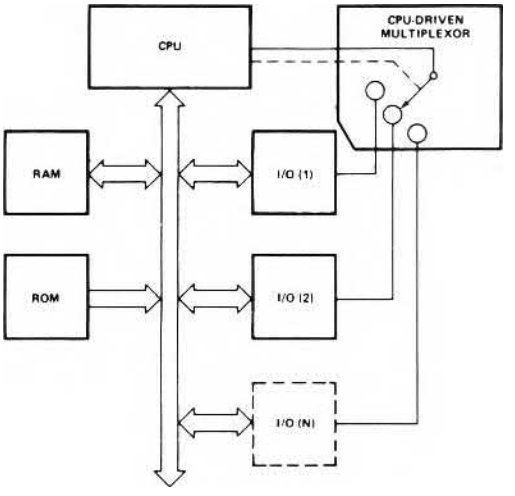
The Programmable Interrupt Controller (PIC) functions as an overall manager in an Interrupt-Driven system environment. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.

Each peripheral device or structure usually has a special program or "routine" that is associated with its specific functional or operational requirements; this is referred to as a "service routine". The PIC, after issuing an Interrupt to the CPU, must somehow input information into the CPU that can "point" the Program Counter to the service routine associated with the requesting device. This "pointer" is an address in a vectoring table and will often be referred to, in this document, as vectoring data.

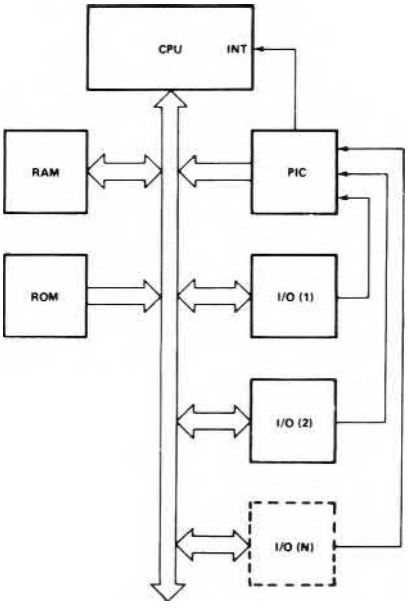
8259A BASIC FUNCTIONAL DESCRIPTION

GENERAL

The 8259A is a device specifically designed for use in real time, interrupt driven microcomputer systems. It manages eight levels or requests and has built-in features for expandability to other 8259A's (up to 64 levels). It is programmed by the system's software as an I/O peripheral. A selection of priority modes is available to the programmer so that the manner in which the requests are processed by the 8259A can be configured to match his system requirements. The priority modes can be changed or reconfigured dynamically at any time during the main program. This means that the complete interrupt structure can be defined as required, based on the total system environment.



Polled Method



Interrupt Method

INTERRUPT REQUEST REGISTER (IRR) AND IN-SERVICE REGISTER (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to store all the interrupt levels which are requesting service; and the ISR is used to store all the interrupt levels which are being serviced.

PRIORITY RESOLVER

This logic block determines the priorities of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during $\overline{INTA}$ pulse.

INTERRUPT MASK REGISTER (IMR)

The IMR stores the bits which mask the interrupt lines to be masked. The IMR operates on the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower priority.

INT (INTERRUPT)

This output goes directly to the CPU interrupt input. The V_{OH} level on this line is designed to be fully compatible with the 8080A, 8085A, 8086 and 8088.

$\overline{INTA}$ (INTERRUPT ACKNOWLEDGE)

$\overline{INTA}$ pulses will cause the 8259A to release vectoring information onto the data bus. The format of this data depends on the system mode (μPM) of the 8259A.

DATA BUS BUFFER

This 3-state, bidirectional 8-bit buffer is used to interface the 8259A to the system Data Bus. Control words and status information are transferred through the Data Bus Buffer.

READ/WRITE CONTROL LOGIC

The function of this block is to accept OUTput commands from the CPU. It contains the Initialization Command Word (ICW) registers and Operation Command Word (OCW) registers which store the various control formats for device operation. This function block also allows the status of the 8259A to be transferred onto the Data Bus.

$\overline{CS}$ (CHIP SELECT)

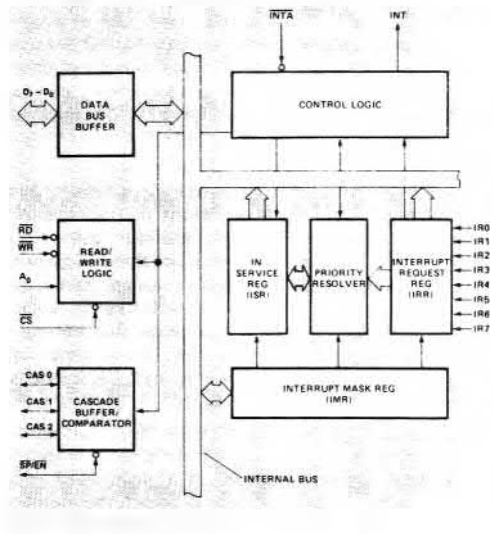
A LOW on this input enables the 8259A. No reading or writing of the chip will occur unless the device is selected.

$\overline{WR}$ (WRITE)

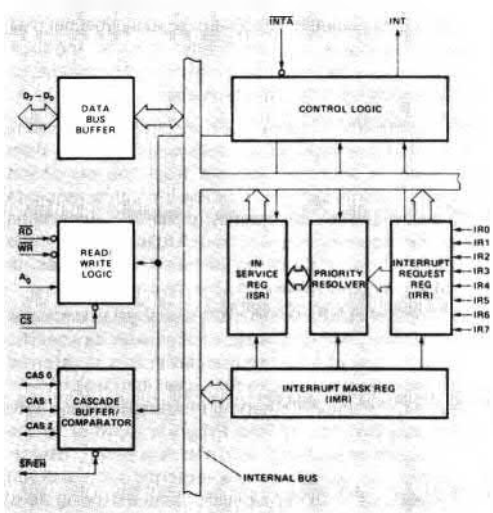
A LOW on this input enables the CPU to write control words (ICWs and OCWs) to the 8259A.

$\overline{RD}$ (READ)

A LOW on this input enables the 8259A to send the status of the Interrupt Request Register (IRR), In Service Register (ISR), the Interrupt Mask Register (IMR), or the Interrupt level onto the Data Bus.



8259A Block Diagram



8259A Block Diagram

A_0

This input signal is used in conjunction with $\overline{WR}$ and $\overline{RD}$ signals to write commands into the various command registers, as well as reading the various status registers of the chip. This line can be tied directly to one of the address lines.

THE CASCADE BUFFER/COMPARATOR

This function block stores and compares the IDs of all 8259A's used in the system. The associated three I/O pins (CAS0-2) are outputs when the 8259A is used as a master and are inputs when the 8259A is used as a slave. As a master, the 8259A sends the ID of the interrupting slave device onto the CAS0-2 lines. The slave thus selected will send its preprogrammed subroutine address onto the Data Bus during the next one or two consecutive $\overline{INTA}$ pulses. (See section "Cascading the 8259A".)

INTERRUPT SEQUENCE

The powerful features of the 8259A in a microcomputer system are its programmability and the interrupt routine addressing capability. The latter allows direct or indirect jumping to the specific interrupt routine requested without any polling of the interrupting devices. The normal sequence of events during an interrupt depends on the type of CPU being used.

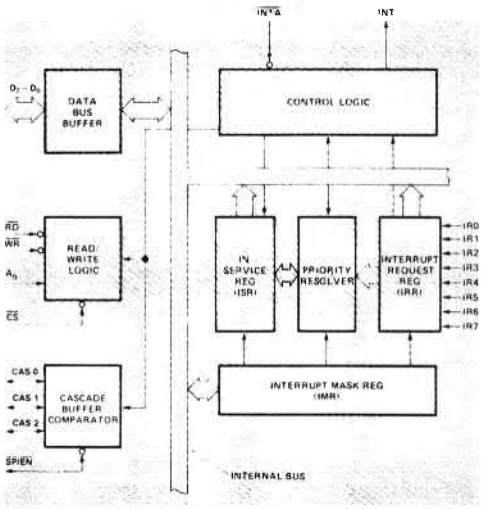
The events occur as follows in an MCS-80/85 system:

1. One or more of the INTERRUPT REQUEST lines (IR7-0) are raised high, setting the corresponding IRR bit(s).
2. The 8259A evaluates these requests, and sends an INT to the CPU, if appropriate.
3. The CPU acknowledges the INT and responds with an $\overline{INTA}$ pulse.
4. Upon receiving an $\overline{INTA}$ from the CPU group, the highest priority ISR bit is set, and the corresponding IRR bit is reset. The 8259A will also release a CALL instruction code (11001101) onto the 8-bit Data Bus through its D7-0 pins.
5. This CALL instruction will initiate two more $\overline{INTA}$ pulses to be sent to the 8259A from the CPU group.
6. These two $\overline{INTA}$ pulses allow the 8259A to release its preprogrammed subroutine address onto the Data Bus. The lower 8-bit address is released at the first $\overline{INTA}$ pulse and and the higher 8-bit address is released at the second $\overline{INTA}$ pulse.
7. This completes the 3-byte CALL instruction released by the 8259A. In the AEOI mode the ISR bit is reset at the end of the third $\overline{INTA}$ pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt sequence.

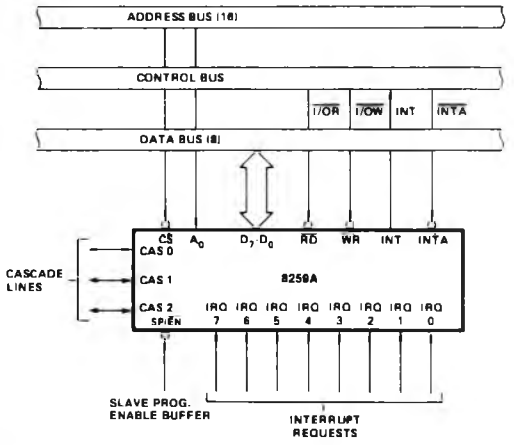
The events occuring in an 8086/8088 system are the same until step 4.

4. Upon receiving an $\overline{INTA}$ from the CPU group, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not drive the Data Bus during this cycle.
5. The 8086/8088 CPU will initiate a second $\overline{INTA}$ pulse. During this pulse, the 8259A releases an 8-bit pointer onto the Data Bus where it is read by the CPU.
6. This completes the interrupt cycle. In the AEOI mode the ISR bit is reset at the end of the second $\overline{INTA}$ pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt subroutine.

If no interrupt request is present at step 4 of either sequence (i.e., the request was too short in duration) the 8259A will issue an interrupt level 7. Both the vectoring bytes and the CAS lines will look like an interrupt level 7 was requested.



8259A Block Diagram



8259A Interface to Standard System Bus

INTERRUPT SEQUENCE OUTPUTS

MCS-80/85 MODE

This sequence is timed by three $\overline{INTA}$ pulses. During the first $\overline{INTA}$ pulse the CALL opcode is enabled onto the data bus.

Content of First Interrupt Vector Byte

	D7	D6	D5	D4	D3	D2	D1	D0
CALL CODE	1	1	0	0	1	1	0	1

During the second $\overline{INTA}$ pulse the lower address of the appropriate service routine is enabled onto the data bus. When Interval = 4 bits A_5-A_7 are programmed, while A_0-A_4 are automatically inserted by the 8259A. When Interval = 8 only A_6 and A_7 are programmed, while A_0-A_5 are automatically inserted.

Content of Second Interrupt Vector Byte

IR	Interval = 4							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0
4	A7	A6	A5	1	0	0	0	0
3	A7	A6	A5	0	1	1	0	0
2	A7	A6	A5	0	1	0	0	0
1	A7	A6	A5	0	0	1	0	0
0	A7	A6	A5	0	0	0	0	0

IR	Interval = 8							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	1	1	1	0	0	0
6	A7	A6	1	1	0	0	0	0
5	A7	A6	1	0	1	0	0	0
4	A7	A6	1	0	0	0	0	0
3	A7	A6	0	1	1	0	0	0
2	A7	A6	0	1	0	0	0	0
1	A7	A6	0	0	1	0	0	0
0	A7	A6	0	0	0	0	0	0

During the third INTA pulse the higher address of the appropriate service routine, which was programmed as byte 2 of the initialization sequence ($A_8 - A_{15}$), is enabled onto the bus.

Content of Third Interrupt Vector Byte

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

8086/8088 Mode

8086/8088 mode is similar to MCS80/85 mode except that only two Interrupt Acknowledge cycles are issued by the processor and no CALL opcode is sent to the processor. The first interrupt acknowledge cycle is similar to that of MCS-80/85 systems in that the 8259A uses it to internally freeze the state of the interrupts for priority resolution and as a master it issues the interrupt code on the cascade lines at the end of the $\overline{INTA}$ pulse. On this first cycle it does not issue any data to the processor and leaves its data bus buffers disabled. On the second interrupt acknowledge cycle in 8086/8088 mode the master (or slave if so programmed) will send a byte of data to the processor with the acknowledged interrupt code composed as follows (note the state of the ADI mode control is ignored and A_5-A_{11} are unused in 8086/8088 mode):

	D7	D6	D5	D4	D3	D2	D1	D0
IR7	T7	T6	T5	T4	T3	1	1	1
IR6	T7	T6	T5	T4	T3	1	1	0
IR5	T7	T6	T5	T4	T3	1	0	1
IR4	T7	T6	T5	T4	T3	1	0	0
IR3	T7	T6	T5	T4	T3	0	1	1
IR2	T7	T6	T5	T4	T3	0	1	0
IR1	T7	T6	T5	T4	T3	0	0	1
IR0	T7	T6	T5	T4	T3	0	0	0

PROGRAMMING THE 8259A

The 8259A accepts two types of command words generated by the CPU:

1. *Initialization Command Words (ICWs):* Before normal operation can begin, each 8259A in the system must be brought to a starting point — by a sequence of 2 to 4 bytes timed by $\overline{WR}$ pulses. This sequence is described in Figure 1.

2. *Operation Command Words (OCWs):* These are the command words that are sent to the 8259A for various forms of operation, such as:

- Interrupt Masking
- End of Interrupt
- Priority Rotation
- Interrupt Status

The OCWs can be written into the 8259A anytime after initialization.

INITIALIZATION

GENERAL

Whenever a command is issued with A0=0 and D4=1, this is interpreted as Initialization Command Word 1 (ICW1). ICW1 starts the initialization sequence during which the following automatically occur.

a. The edge sense circuit is reset, which means that following initialization, an interrupt request (IR) input must make a low-to-high transition to generate an interrupt.
b. The Interrupt Mask Register is cleared.
c. R7 input is assigned priority 7.
d. The slave mode address is set to 7.
e. Special Mask Mode is cleared and Status Read is set to IRR.
f. If IC4=0, then all functions selected in ICW4 are set to zero. (Non-Buffered mode*, no Auto-EOI, MCS-80/85 system).

*Note: Master/Slave in ICW4 is only used in the buffered mode

A_0	D_4	D_3	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	INPUT OPERATION (READ)
0			0	1	0	IRR, ISR or Interrupting Level → DATA BUS (Note 1)
1			0	1	0	IMR → DATA BUS
						OUTPUT OPERATION (WRITE)
0	0	0	1	0	0	DATA BUS → OCW2
0	0	1	1	0	0	DATA BUS → OCW3
0	1	X	1	0	0	DATA BUS → ICW1
1	X	X	1	0	0	DATA BUS → OCW1, ICW2, ICW3, ICW4 (Note 2)
						DISABLE FUNCTION
X	X	X	1	1	0	DATA BUS — 3-STATE NO OPERATION
X	X	X	X	X	1	DATA BUS — 3-STATE (NO OPERATION)

Notes: 1 Selection of IRR, ISR or Interrupting Level is based on the content of OCW3 written before the READ operation
2 On-chip sequencer logic queues these commands into proper sequence

8259A Basic Operation

INITIALIZATION COMMAND WORDS 1 AND 2 (ICW1, ICW2)

A_5-A_{15}: *Page starting address of service routines.* In an MCS 80/85 system, the 8 request levels will generate CALLs to 8 locations equally spaced in memory. These can be programmed to be spaced at intervals of 4 or 8 memory locations, thus the 8 routines will occupy a page of 32 or 64 bytes, respectively.

The address format is 2 bytes long (A_0-A_{15}). When the routine interval is 4, A_0-A_4 are automatically inserted by the 8259A, while A_5-A_{15} are programmed externally. When the routine interval is 8, A_0-A_5 are automatically inserted by the 8259A, while A_6-A_{15} are programmed externally.

The 8-byte interval will maintain compatibility with current software, while the 4-byte interval is best for a compact jump table.

In an MCS-86 system T7-T3 are inserted in the five most significant bits of the vectoring byte and the 8259A sets the three least significant bits according to the interrupt level. A_{10}-A_5 are ignored and ADI (Address Interval) has no effect.

LTIM: If LTIM = 1, then the 8259A will operate in the level interrupt mode. Edge detect logic on the interrupt inputs will be disabled.

ADI: CALL address interval. ADI = 1 then interval = 4; ADI = 0 then interval = 8.

SNGL: Single. Means that this is the only 8259A in the system. If SNGL = 1 no ICW3 will be issued.

IC4: If this bit is set — ICW4 has to be read. If ICW4 is not needed, set IC4 = 0.

INITIALIZATION COMMAND WORD 3 (ICW3)

This word is read only when there is more than one 8259A in the system and cascading is used, in which case SNGL = 0. It will load the 8-bit slave register. The functions of this register are:

a. In the master mode (either when $\overline{SP}$ = 1, or in buffered mode when M/S = 1 in ICW4) a "1" is set for each slave in the system. The master then will release byte 1 of the call sequence (for MCS-80/85 system) and will enable the corresponding slave to release bytes 2 and 3 (for 8086/8088 only byte 2) through the cascade lines.

b. In the slave mode (either when $\overline{SP}$ = 0, or if BUF = 1 and M/S = 0 in ICW4) bits 2-0 identify the slave. The slave compares its cascade input with these bits and if they are equal, bytes 2 and 3 of the call sequence (or just byte 2 for 8086/8088) are released by it on the Data Bus.

INITIALIZATION COMMAND WORD 4 (ICW4)

SFNM: If SFNM = 1 the special fully nested mode is programmed.

BUF: If BUF = 1 the buffered mode is programmed. In buffered mode $\overline{SP}/\overline{EN}$ becomes an enable output and the master/slave determination is by M/S.

M/S: If buffered mode is selected: M/S = 1 means the 8259A is programmed to be a master, M/S = 0 means the 8259A is programmed to be a slave. If BUF = 0, M/S has no function.

AEOI: If AEOI = 1 the automatic end of interrupt mode is programmed.

μPM: Microprocessor mode: μPM = 0 sets the 8259A for MCS-80/85 system operation, μPM = 1 sets the 8259A for MCS-86 system operation.

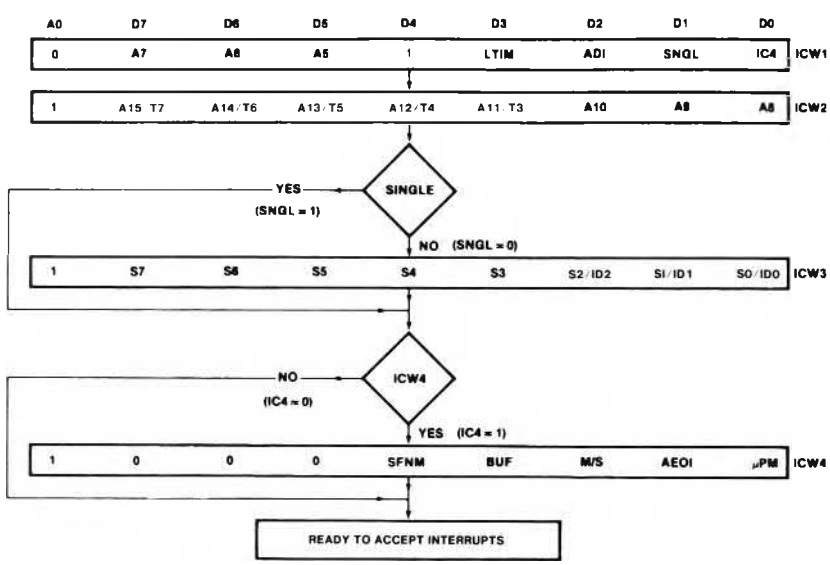
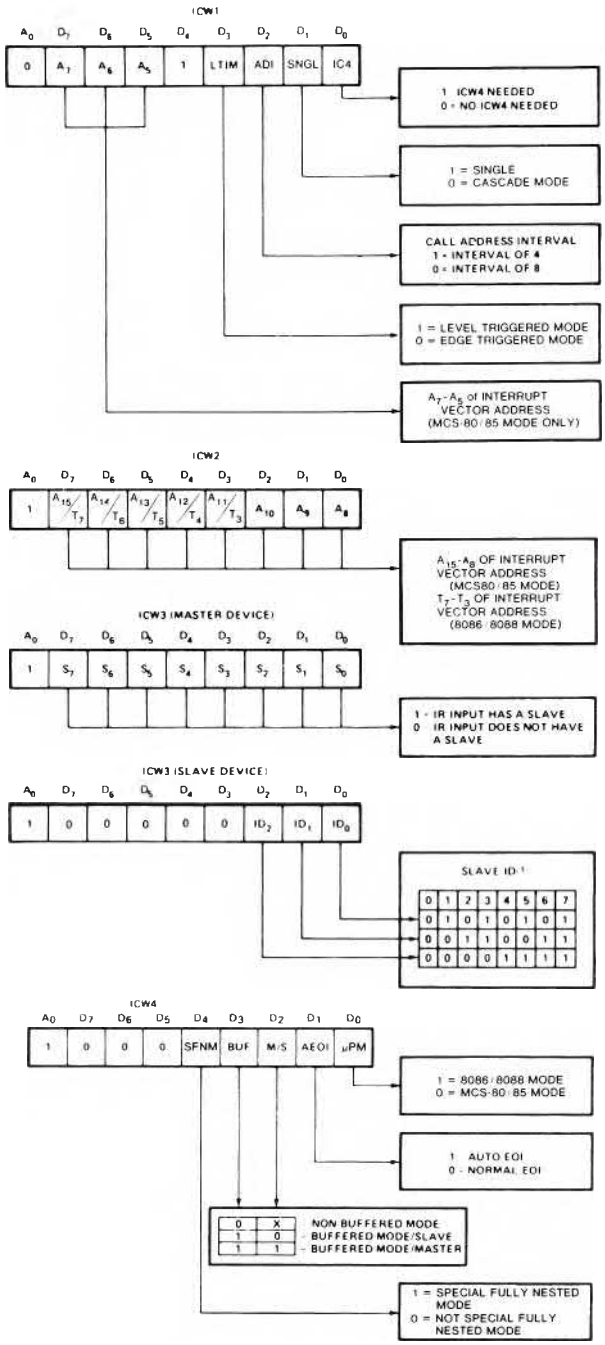


Figure 1. Initialization Sequence



NOTE 1 SLAVE ID IS EQUAL TO THE CORRESPONDING MASTER IR INPUT

Initialization Command Word Format

OPERATION COMMAND WORDS (OCWs)

After the Initialization Command Words (ICWs) are programmed into the 8259A, the chip is ready to accept Interrupt requests at its input lines. However, during the 8259A operation, a selection of algorithms can command the 8259A to operate in various modes through the Operation Command Words (OCWs).

OPERATION CONTROL WORDS (OCWs)

	OCW1							
A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

	OCW2							
0	R	SL	EOI	0	0	L2	L1	L0

	OCW3							
0	0	ESMM	SMM	0	1	P	RR	RIS

OPERATION CONTROL WORD 1 (OCW1)

OCW1 sets and clears the mask bits in the interrupt Mask Register (IMR). $M_7 - M_0$ represent the eight mask bits. M = 1 indicates the channel is masked (inhibited), M = 0 indicates the channel is enabled.

OPERATION CONTROL WORD 2 (OCW2)

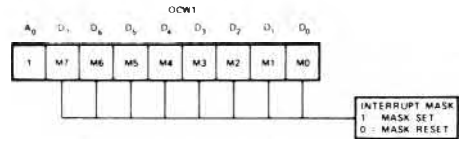
R, SL, EOI — These three bits control the Rotate and End if Interrupt modes and combinations of the two. A chart of these combinations can be found on the Operation Command Word Format.

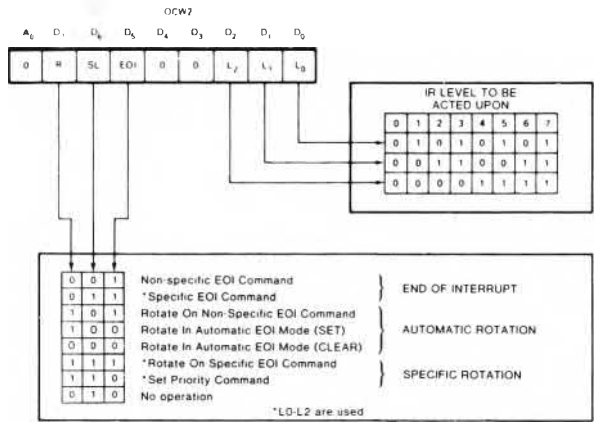
L_2, L_1, L_0 — These bits determine the interrupt level acted upon when the SEOI bit is active.

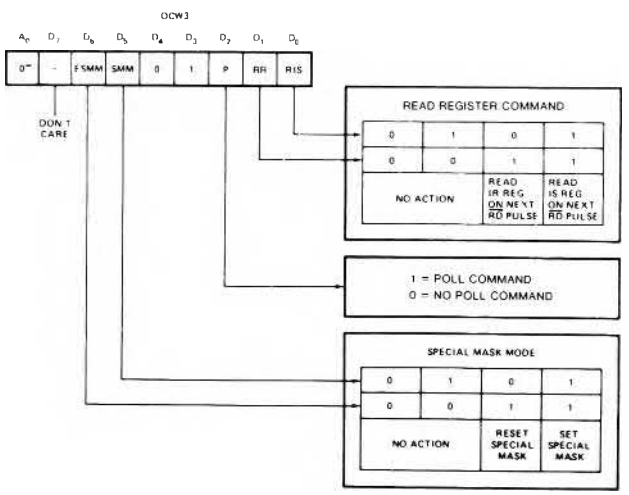
OPERATION CONTROL WORD 3 (OCW3)

ESMM — Enable Special Mask Mode. When this bit is set to 1 it enables the SMM bit to set or reset the Special Mask Mode. When ESMM = 0 the SMM bit becomes a "don't care".

SMM — Special Mask Mode. If ESMM = 1 and SMM = 1 the 8259A will enter Special Mask Mode. If ESMM = 1 and SMM = 0 the 8259A will revert to normal mask mode. When ESMM = 0, SMM has no effect.







Operation Command Word Format

INTERRUPT MASKS

Each Interrupt Request input can be masked individually by the Interrupt Mask Register (IMR) programmed through OCW1. Each bit in the IMR masks one interrupt channel if it is set (1). Bit 0 masks IR0, Bit 1 masks IR1 and so forth. Masking an IR channel does not affect the other channels operation.

SPECIAL MASK MODE

Some applications may require an interrupt service routine to dynamically alter the system priority structure during its execution under software control. For example, the routine may wish to inhibit lower priority requests for a portion of its execution but enable some of them for another portion.

The difficulty here is that if an Interrupt Request is acknowledged and an End of Interrupt command did not reset its IS bit (i.e., while executing a service routine), the 8259A would have inhibited all lower priority requests with no easy way for the routine to enable them

That is where the Special Mask Mode comes in. In the special Mask Mode, when a mask bit is set in OCW1, it inhibits further interrupts at that level *and enables* interrupts from *all other* levels (lower as well as higher) that are not masked.

Thus, any interrupts may be selectively enabled by loading the mask register.

The special Mask Mode is set by OCW3 where: SMM = 1, SMM = 1, and cleared where SMM = 1, SMM = 0.

BUFFERED MODE

When the 8259A is used in a large system where bus driving buffers are required on the data bus and the cascading mode is used, there exists the problem of enabling buffers.

The buffered mode will structure the 8259A to send an enable signal on $\overline{SP}/\overline{EN}$ to enable the buffers. In this mode, whenever the 8259A's data bus outputs are enabled, the $\overline{SP}/\overline{EN}$ output becomes active.

This modification forces the use of software programming to determine whether the 8259A is a master or a slave. Bit 3 in ICW4 programs the buffered mode, and bit 2 in ICW4 determines whether it is a master or a slave.

FULLY NESTED MODE

This mode is entered after initialization unless another mode is programmed. The interrupt requests are ordered in priority form 0 through 7 (0 highest). When an interrupt is acknowledged the highest priority request is determined and its vector placed on the bus. Additionally, a bit of the Interrupt Service register (ISO-7) is set. This bit remains set until the microprocessor issues an End of Interrupt (EOI) command immediately before returning from the service routine, or if AEOI (Automatic End of Interrupt) bit is set, until the trailing edge of the last INTA. While the IS bit is set, all further interrupts of the same or lower priority are inhibited, while higher levels will generate an interrupt (which will be acknowledged only if the microprocessor internal Interrupt enable flip-flop has been re-enabled through software).

After the initialization sequence, IR0 has the highest priority and IR7 the lowest. Priorities can be changed, as will be explained, by priority rotation.

THE SPECIAL FULLY NESTED MODE

This mode will be used in the case of a big system where cascading is used, and the priority has to be conserved within each slave. In this case the special fully nested mode will be programmed to the master (using) ICW4). This mode is similar to the normal fully nested mode with the following exceptions:

a. When an interrupt request from a certain slave is in service this slave is not locked out from the master's priority logic and further interrupt requests from higher priority IR's within the slave will be recognized by the master and will initiate interrupts to the processor. (In the normal nested mode a slave is masked out when its request is in service and no higher requests from the same slave can be serviced.)

b. When exiting the Interrupt Service routine the software has to check whether the interrupt serviced was the only one from that slave. This is done by sending a non-specific End of Interrupt (EOI) command to the slave and then reading its In-Service register and checking for zero. If it is empty, a non-specific EOI can be sent to the master too. If not, no EOI should be sent.

POLL

In this mode the microprocessor internal Interrupt Enable flip-flop is reset, disabling its interrupt input. Service to devices is achieved by programmer initiative using a Poll command.

The Poll command is issued by setting P = "1" in OCW3. The 8259A treats the next $\overline{RD}$ pulse to the 8259A (i.e., $\overline{RD}$ = 0, $\overline{CS}$ = 0) as an interrupt acknowledge, sets the appropriate IS bit if there is a request, and reads the priority level. Interrupt is frozen from $\overline{WR}$ to $\overline{RD}$.

The word enabled onto the data bus during $\overline{RD}$ is:

D7	D6	D5	D4	D3	D2	D1	D0
I	–	–	–	–	W2	W1	W0

W0-W2 Binary code of the highest priority level requesting service

I Equal to a "1" if there is an interrupt.

This mode is useful if there is a routine command common to several levels so that the $\overline{INTA}$ sequence is not needed (saves ROM space). Another application is to use the poll command to expand the number of priority levels to more than 64.

END OF INTERRUPT (EOI)

The In Service (IS) bit can be reset either automatically following the trailing edge of the last in sequence $\overline{INTA}$ pulse (when AEOI bit in ICW1 is set) or by a command word that must be issued to the 8259A before returning from a service routine (EOI command). An EOI command must be issued twice, once for themaster and once for the corresponding slave if slaves are in use.

There are two forms of EOI command: Specific and Non-Specific. When the 8259A is operated in modes which preserve the fully nested structure, it can determine which IS bit to reset on EOI. When a Non-Specific EOI command is issued the 8259A will automatically reset the highest IS bit of those that are set, since in the nested mode the highest IS level was necessarily the last level acknowledged and serviced.

However, when a mode is used which may disturb the fully nested structure the 8259A may no longer be able to determine the last level acknowledged. In this case a Specific End of Interrupt (SEOI) must be issued which includes as part of the command the IS level to be reset. EOI is issued whenever EOI = 1, in OCW2, where L0-L2 is the binary level of the IS bit to be reset. Note that although the Rotate command can be issued together with an EOI where EOI = 1, it is not necessarily tied to it.

It should be noted that an IS bit that is masked by an IMR bit will not be cleared by a non specific EOI if the 8259A is in the Special Mask Mode.

AUTOMATIC END OF INTERRUPT (AEOI) MODE

If AEOI = 1 in ICW4, then the 8259A will operate in AEOI mode continuously until reprogrammed by ICW4. In this mode the 8259A will automatically perform a non-specific EOI operation at the trailing edge of the last interrupt acknowledge pulse (third pulse in MCS-80/85, second in MCS-86). Note that from a system standpoint, this mode should be used only when a nested multilevel interrupt structure is not required within a single 8259A.

To achieve automatic rotation within AEOI, there is a special rotate flip-flop. It is set by OCW2 with R = 1, SL = 0, EOI = 0, and cleared with R = 0, SEOI = 0, EOI = 0.

AUTOMATIC ROTATION (Equal Priority Devices)

In some applications there are a number of interrupting devices of equal priority. In this mode a device, after being serviced, receives the lowest priority, so a device requesting an interrupt will have to wait, in the worst case until each of 7 other devices are serviced at most *once*. For example, if the priority and "in service" status is:

Before Rotate (IR4 the highest priority requiring service)

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
"IS" Status	0	1	0	1	0	0	0	0
	Lowest Priority							Highest Priority
Priority Status	7	6	5	4	3	2	1	0

After Rotate (IR4 was serviced, all other priorities rotated correspondingly)

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
"IS" Status	0	1	0	0	0	0	0	0
	Highest Priority							Lowest Priority
Priority Status	2	1	0	7	6	5	4	3

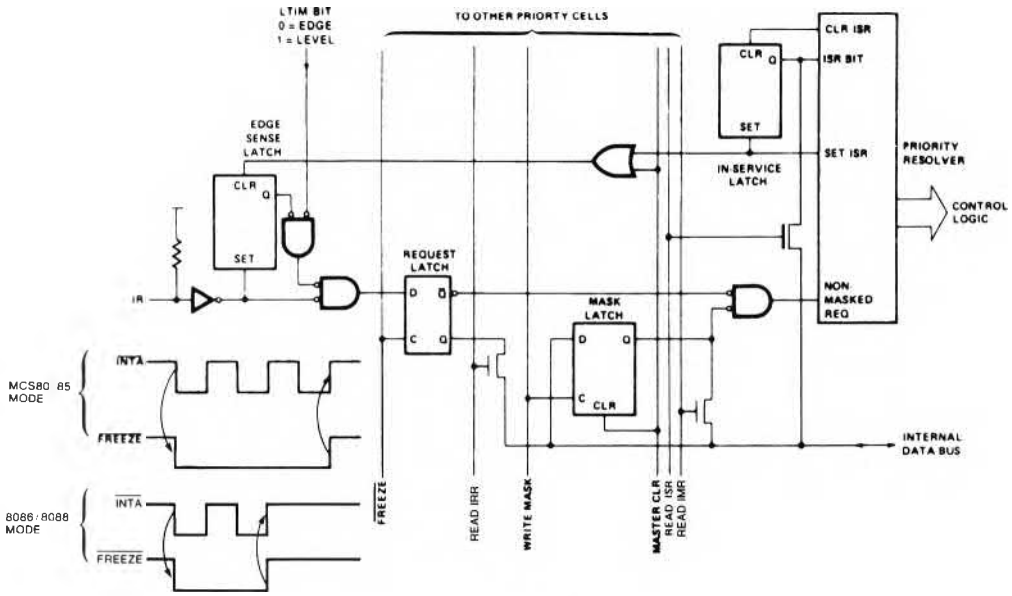
There are two ways to accomplish Automatic Rotation using OCW2, the Rotate on Non-Specific EOI Command (R = 1, SL = 0, EOI = 1) and the Rotate in Automatic EOI Mode which is set by (R = 1, SL = 0, EOI = 0) and cleared by (R = 0 SL = 0, EOI = 0).

SPECIFIC ROTATION (Specific Priority)

The programmer can change priorities by programming the bottom priority and thus fixing all other priorities; i.e., if IR5 is programmed as the bottom priority device, then IR6 will have the highest one.

The Set Priority command is issued in OCW2 where: R = 1, SEOI = 1; L0-L2 is the binary priority level code of the bottom priority device.

Observe that in this mode internal status is updated by software control during OCW2. However, it is independent of the End of Interrupt (EOI) command (also executed by OCW2). Priority changes can be executed during an EOI command by using the Rotate on Specific EOI Command in OCW2 (R = 1, SL = 1, EOI = 1 and L0-L2 = IR level to receive bottom priority).



NOTES

1. MASTER CLEAR ACTIVE ONLY DURING ICW1
2. FREEZE/ IS ACTIVE DURING $\overline{\text{INTA}}$/ AND POLL SEQUENCES ONLY
3. TRUTH TABLE FOR D-LATCH

C	D	Q	OPERATION
1	D_n	D_n	FOLLOW
0	X	Q_{n-1}	HOLD

Priority Cell — Simplified Logic Diagram

LEVEL TRIGGERED MODE

This mode is programmed using bit 3 in ICW1.

If LTIM = '1,' an interrupt request will be recognized by a 'high' level on IR Input, and there is no need for an edge detection. The interrupt request must be removed before the EOI command is issued or the CPU interrupt is enabled to prevent a second interrupt from occurring.

The above figure shows a conceptual circuit to give the reader an understanding of the level sensitive and edge sensitive input circuitry of the 8259A. Be sure to note that the request latch is a transparent D type latch.

READING THE 8259A STATUS

The input status of several internal registers can be read to update the user information on the system. The following registers can be read via OCW3 (IRR and ISR or OCW1 (IMR).

Interrupt Request Register (IRR): 8-bit register which contains the levels requesting an interrupt to be acknowledged. The highest request level is reset from the IRR when an interrupt is acknowledged. (Not affected by IMR).

In-Service Register (ISR): 8-bit register which contains the priority levels that are being serviced. The ISR is updated when an End of Interrupt command is issued.

Interrupt Mask Register: 8-bit register which contains the interrupt request lines which are masked.

The IRR can be read when, prior to the $\overline{\text{RD}}$ pulse, a Read Register Command is issued with OCW3 (RR = 1, RIS = 0).

The ISR can be read when, prior to the $\overline{\text{RD}}$ pulse, a Read Register Command is issued with OCW3 (RR = 1, RIS = 1).

There is no need to write an OCW3 before every status read operation, as long as the status read corresponds with the previous one; i.e., the 8259A "remembers" whether the IRR or ISR has been previously selected by the OCW3. This is not true when poll is used

After initialization the 8259A is set to IRR.

For reading the IMR, no OCW3 is needed. The output data bus will contain the IMR whenever $\overline{\text{RD}}$ is active and A0 = 1 (OCW1).

Polling overrides status read when P = 1, RR = 1 in OCW3.

SUMMARY OF 8259A INSTRUCTION SET

Inst. #	Mnemonic		A0	D7	D6	D5	D4	D3	D2	D1	D0	Operation Description	
1	ICW1	A	0	A7	A6	A5	1	0	1	1	0		Format = 4, single, edge triggered
2	ICW1	B	0	A7	A6	A5	1	1	1	1	0		Format = 4, single, level triggered
3	ICW1	C	0	A7	A6	A5	1	0	1	0	0	Byte 1 Initialization	Format = 4, not single, edge triggered
4	ICW1	D	0	A7	A6	A5	1	1	1	0	0		Format = 4, not single, level triggered
5	ICW1	E	0	A7	A6	0	1	0	0	1	0	No ICW4 Required	Format = 8, single, edge triggered
6	ICW1	F	0	A7	A6	0	1	1	0	1	0		Format = 8, single, level triggered
7	ICW1	G	0	A7	A6	0	1	0	0	0	0		Format = 8, not single, edge triggered
8	ICW1	H	0	A7	A6	0	1	1	0	0	0		Format = 8, not single, level triggered
9	ICW1	I	0	A7	A6	A5	1	0	1	1	1		Format = 4, single, edge triggered
10	ICW1	J	0	A7	A6	A5	1	1	1	1	1		Format = 4, single, level triggered
11	ICW1	K	0	A7	A6	A5	1	0	1	0	1	Byte 1 Initialization	Format = 4, not single, edge triggered
12	ICW1	L	0	A7	A6	A5	1	1	1	0	1		Format = 4, not single, level triggered
13	ICW1	M	0	A7	A6	0	1	0	0	1	1	ICW4 Required	Format = 8, single, edge triggered
14	ICW1	N	0	A7	A6	0	1	1	0	1	1		Format = 8, single, level triggered
15	ICW1	O	0	A7	A6	0	1	0	0	0	1		Format = 8, not single, edge triggered
16	ICW1	P	0	A7	A6	0	1	1	0	0	1		Format = 8, not single, level triggered
17	ICW2		1	A15	A14	A13	A12	A11	A10	A9	A8	Byte 2 initialization	
18	ICW3	M	1	S7	S6	S5	S4	S3	S2	S1	S0	Byte 3 initialization — master	
19	ICW3	S	1	0	0	0	0	0	S2	S1	S0	Byte 3 initialization — slave	
20	ICW4	A	1	0	0	0	0	0	0	0	0	No action, redundant	
21	ICW4	B	1	0	0	0	0	0	0	0	1	Non-buffered mode, no AEOI, 8086/8088	
22	ICW4	C	1	0	0	0	0	0	0	1	0	Non-buffered mode, AEOI, MCS-80/85	
23	ICW4	D	1	0	0	0	0	0	0	1	1	Non-buffered mode, AEOI, 8086/8088	
24	ICW4	E	1	0	0	0	0	0	1	0	0	No action, redundant	
25	ICW4	F	1	0	0	0	0	0	1	0	1	Non-buffered mode, no AEOI, 8086/8088	
26	ICW4	G	1	0	0	0	0	0	1	1	0	Non-buffered mode, AEOI, MCS-80/85	
27	ICW4	H	1	0	0	0	0	0	1	1	1	Non-buffered mode, AEOI, 8086/8088	
28	ICW4	I	1	0	0	0	0	1	0	0	0	Buffered mode, slave, no AEOI, MCS-80/85	
29	ICW4	J	1	0	0	0	0	1	0	0	1	Buffered mode, slave, no AEOI, 8086/8088	
30	ICW4	K	1	0	0	0	0	1	0	1	0	Buffered mode, slave, AEOI, MCS-80/85	
31	ICW4	L	1	0	0	0	0	1	0	1	1	Buffered mode, slave, AEOI, 8086/8088	
32	ICW4	M	1	0	0	0	0	1	1	0	0	Buffered mode, master, no AEOI, MCS-80/85	
33	ICW4	N	1	0	0	0	0	1	1	0	1	Buffered mode, master, no AEOI, 8086/8088	
34	ICW4	O	1	0	0	0	0	1	1	1	0	Buffered mode, master, AEOI, MCS-80/85	
35	ICW4	P	1	0	0	0	0	1	1	1	1	Buffered mode, master AEOI, 8086, 8088	
36	ICW4	NA	1	0	0	0	1	0	0	0	0	Fully nested mode, MCS-80, non buffered, no AEOI	
37	ICW4	NB	1	0	0	0	1	0	0	0	1	ICW4 NB through ICW4 ND are identical to	
38	ICW4	NC	1	0	0	0	1	0	0	1	0	ICW4 B through ICW4 D with the addition of	
39	ICW4	ND	1	0	0	0	1	0	0	1	1	Fully Nested Mode	
40	ICW4	NE	1	0	0	0	1	0	1	0	0	Fully Nested Mode, MCS-80/85, non-buffered, no AEOI	
41	ICW4	NF	1	0	0	0	1	0	1	0	1		
42	ICW4	NG	1	0	0	0	1	0	1	1	0		
43	ICW4	NH	1	0	0	0	1	0	1	1	1		
44	ICW4	NI	1	0	0	0	1	1	0	0	0		
45	ICW4	NJ	1	0	0	0	1	1	0	0	1		
46	ICW4	NK	1	0	0	0	1	1	0	1	0	ICW4 NF through ICW4 NP are identical to	
47	ICW4	NL	1	0	0	0	1	1	0	1	1	ICW4 F through ICW4 P with the addition of	
48	ICW4	NM	1	0	0	0	1	1	1	0	0	Fully Nested Mode	
49	ICW4	NN	1	0	0	0	1	1	1	0	1		
50	ICW4	NO	1	0	0	0	1	1	1	1	0		
51	ICW4	NP	1	0	0	0	1	1	1	1	1		
52	OCW1		1	M7	M6	M5	M4	M3	M2	M1	M0	Load mask register, read mask register	
53	OCW2	E	0	0	0	1	0	0	0	0	0	Non-specific EOI	
54	OCW2	SE	0	0	1	1	0	0	L2	L1	L0	Specific EOI, L0-L2 code of IS FF to be reset	
55	OCW2	RE	0	1	0	1	0	0	0	0	0	Rotate on Non-Specific EOI	
56	OCW2	RSE	0	1	1	1	0	0	L2	L1	L0	Rotate on Specific EOI L0-L2 code of line	
57	OCW2	R	0	1	0	0	0	0	0	0	0	Rotate in Auto EOI (set)	
58	OCW2	CR	0	0	0	0	0	0	0	0	0	Rotate in Auto EOI (clear)	
59	OCW2	RS	0	1	1	0	0	0	L2	L1	L0	Set Priority Command	
60	OCW3	P	0	0	0	0	0	1	1	0	0	Poll mode	
61	OCW3	RIS	0	0	0	0	0	1	0	1	1	Read IS register	

SUMMARY OF 8259A INSTRUCTION SET (Cont.)

Inst. #	Mnemonic	A0	D7	D6	D5	D4	D3	D2	D1	D0	Operation Description
46	OCW3 RR	0	0	0	0	0	1	0	1	0	Read request register
47	OCW3 SM	0	0	1	1	0	1	0	0	0	Set special mask mode
48	OCW3 RSM	0	0	1	0	0	1	0	0	0	Reset special mask mode

Note: 1 In the master mode $\overline{SP}$ pin = 1, in slave mode $\overline{SP}$ = 0

Cascading

The 8259A can be easily interconnected in a system of one master with up to eight slaves to handle up to 64 priority levels.

A typical MCS-80/85 system is shown in Figure 2. The master controls, through the 3 line cascade bus, which one of the slaves will release the corresponding address.

As shown in Figure 2, the slave interrupt outputs are connected to the master interrupt request inputs. When a slave request line is activated and afterwards acknowledged, the master will enable the corresponding slave to release the device routine address during bytes 2 and 3 of $\overline{INTA}$. (Byte 2 only for 8086/8088).

The cascade bus lines are normally low and will contain the slave address code from the trailing edge of the first $\overline{INTA}$ pulse to the trailing edge of the third pulse. It is obvious that each 8259A in the system must follow a separate initialization sequence and can be programmed to work in a different mode. An EOI command must be issued twice: once for the master and once for the corresponding slave. An address decoder is required to activate the Chip Select ($\overline{CS}$) input of each 8259A.

The cascade lines of the Master 8259A are activated for any interrupt input, even if no slave is connected to that input.

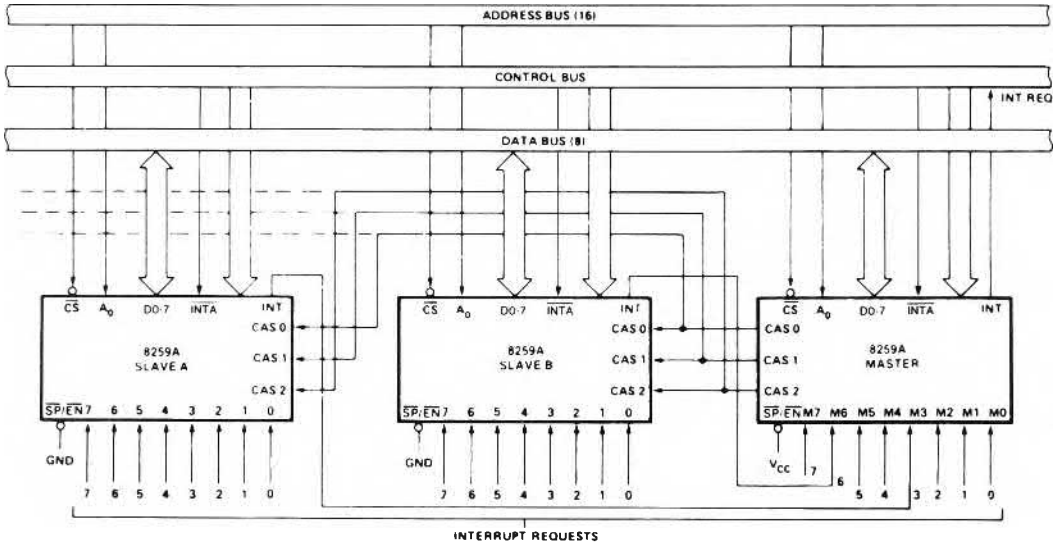


Figure 2. Cascading the 8259A

PIN FUNCTIONS

NAME	I/O	PIN#	FUNCTION
V_{CC}	I	28	+5v supply
GND	I	14	Ground
$\overline{CS}$	I	1	Chip Select A low on this pin enables RD and WR communication between the CPU and the 8259A. INTA functions are independent of CS.
$\overline{WR}$	I	2	Write: A low on this pin when CS is low, enables the 8259A to accept command words from the CPU.
$\overline{RD}$	I	3	Read: A low on this pin when CS is low enables the 8259A to release status onto the data bus for the CPU.
D_7-D_0	I/O	4-11	Bidirectional Data Bus: Control, status and interrupt-vector information is transferred via this bus.
CAS_0-CAS_2	I/O	12,13,15	Cascade Lines: The CAS lines form a private 8259A bus to control a multiple 8259A structure. These pins are outputs for a master 8259A and inputs for a slave 8259A.
$\overline{SP}/\overline{EN}$	I/O	16	Slave Program/Enable Buffer: This is a dual function pin. When in the Buffered Mode it can be used as an output to control buffer transceivers (EN). When not in the buffered mode it is used as an input to designate a master (SP = 1) or slave (SP = 0).
INT	O	17	Interrupt: This pin goes high whenever a valid interrupt request is asserted. It is used to interrupt the CPU, thus it is connected to the CPU's interrupt pin.
IR_0-IR_7	I	18-25	Interrupt Requests: Asynchronous inputs. An interrupt request can be generated by raising an IR input (low to high) and holding it high until it is acknowledged (Edge Triggered Mode), or just by a high level on an IR input (Level Triggered Mode).
$\overline{INTA}$	I	26	Interrupt Acknowledge: This pin is used to enable 8259A interrupt-vector data onto the data bus. This is done by a sequence of interrupt acknowledge pulses issued by the CPU.
A_0	I	27	A0 Address Line: This pin acts in conjunction with the $\overline{CS}$, $\overline{WR}$, and $\overline{RD}$ pins. It is used by the 8259A to decipher between various Command Words the CPU writes and status the CPU wishes to read. It is typically connected to the CPU A0 address line (A1 for 8086/8088).

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ..... −40°C to 85°C
Storage Temperature ..... −65°C to +150°C
Voltage On Any Pin
With Respect to Ground ..... −0.5V to +7V
Power Dissipation ..... 1 Watt

*COMMENT
Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied

D.C. CHARACTERISTICS

$T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$ (8259-A), $V_{CC} = 5V \pm 10\%$ (8259A)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	-.5	V		
V_{IH}	Input High Voltage	2.0	V_{CC} + .5V	V	
V_{OL}	Output Low Voltage		.45	V	$I_{OL} = 2.2$ mA
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\ \mu A$
$V_{OH(INT)}$	Interrupt Output High Voltage	3.5 2.4		C V	$I_{OH} = -100\ \mu A$ $I_{OH} = -400\ \mu A$
I_{LI}	Input Load Current		10	μA	$V_{IN} = V_{CC}$ to 0V
I_{LOL}	Output Leakage Current		-10	μA	$V_{OUT} = 0.45V$
I_{CC}	V_{CC} Supply Current		85	mA	
I_{LIR}	IR Input Load Current		-300 10	μA μA	$V_{IN} = 0$ $V_{IN} = V_{CC}$

8259A A.C. CHARACTERISTICS

$T_A = 0°C$ to $70°C$ $V_{CC} = 5V \pm 5\%$ (8259A-8) $V_{CC} = 5V \pm 10\%$ (8259A)

TIMING REQUIREMENTS

Symbol	Parameter	8259A-8		8259A		8259A-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TAHRL	A0/$\overline{CS}$ Setup to $\overline{RD}$/$\overline{INTA}$↓	50		0		0		ns	
TRHAX	A0/$\overline{CS}$ Hold after $\overline{RD}$/$\overline{INTA}$↑	5		0		0		ns	
TRLRH	$\overline{RD}$ Pulse Width	420		235		160		ns	
TAHWL	A0/$\overline{CS}$ Setup to $\overline{WR}$↓	50		0		0		ns	
TWHAX	A0/$\overline{CS}$ Hold after $\overline{WR}$↑	20		0		0		ns	
TWLWH	$\overline{WR}$ Pulse Width	400		290		190		ns	
TDVWH	Data Setup to $\overline{WR}$↑	300		240		160		ns	
TWHDX	Data Hold after $\overline{WR}$↑	40		0		0		ns	
TJLJH	Interrupt Request Width (Low)	100		100		100		ns	See Note 1
TCVIAL	Cascade Setup to Second or Third $\overline{INTA}$↓ (Slave Only)	55		55		40		ns	
TRHRL	End of $\overline{RD}$ to Next Command	160		160		160		ns	
TWHRL	End of $\overline{WR}$ to Next Command	190		190		190		ns	

Note: This is the low time required to clear the input latch in the edge triggered mode.

TIMING RESPONSES

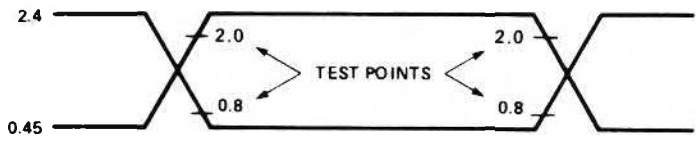
Symbol	Parameter	8259A-8		8259A		8259A-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TRLDV	Data Valid from $\overline{RD}$/$\overline{INTA}$↓		300		200		120	ns	C of Data Bus = 100 pF
TRHDZ	Data Float after $\overline{RD}$/$\overline{INTA}$↑	10	200		100		85	ns	C of Data Bus Max text C = 100 pF Min. test C = 15 pF
TJHIH	Interrupt Output Delay		400		350		300	ns	
TIAHCV	Cascade Valid from First $\overline{INTA}$↓ (Master Only)		565		565		360	ns	C_{INT} = 100 pF
TRLEL	Enable Active from $\overline{RD}$↓ or $\overline{INTA}$↓		160		125		100	ns	$C_{CASCADE}$ = 100 pF
TRHEH	Enable Inactive from $\overline{RD}$↑ or $\overline{INTA}$↑		325		150		d150	ns	
TAHDV	Data Valid from Stable Address		350		200		200	ns	
TCVDV	Cascade Valid to Valid Data		300		300		200	ns	

CAPACITANCE

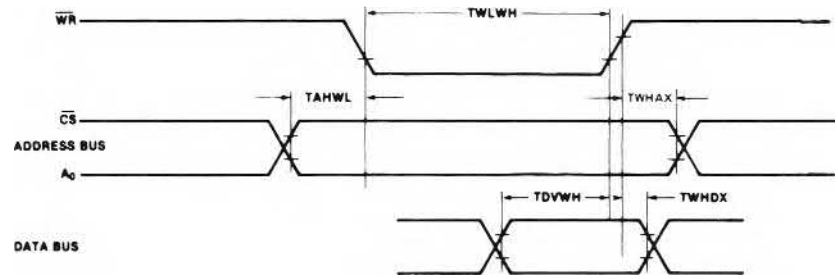
$T_A = 25°C$; $V_{CC} = GND = 0V$

Symbol	Parameter	Min.	Typ.	Max.	Unit	Test Conditions
C_{IN}	Input Capacitance			10	pF	fc = 1 MHz
$C_{I/O}$	I/O Capacitance			20	pF	Unmeasured pins returned to V_{SS}

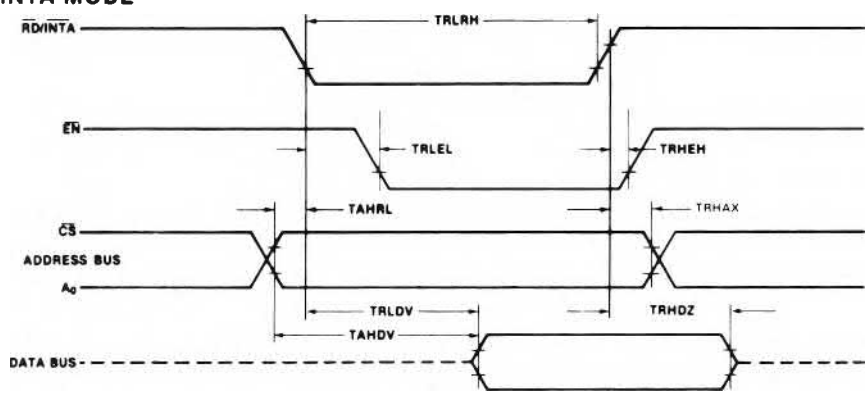
Input and Output Waveforms for A.C. Tests



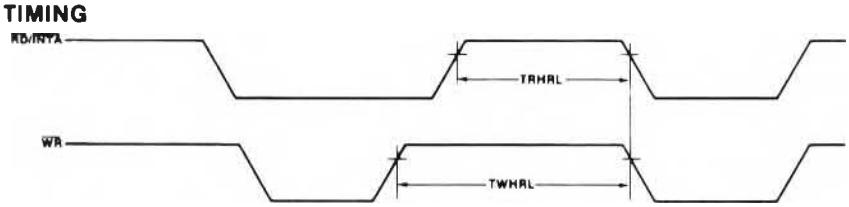
WRITE MODE



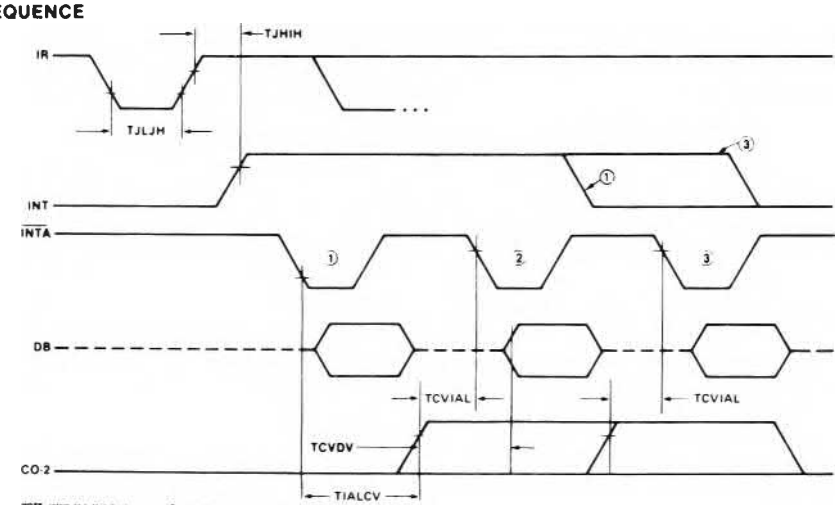
READ/INTA MODE



OTHER TIMING

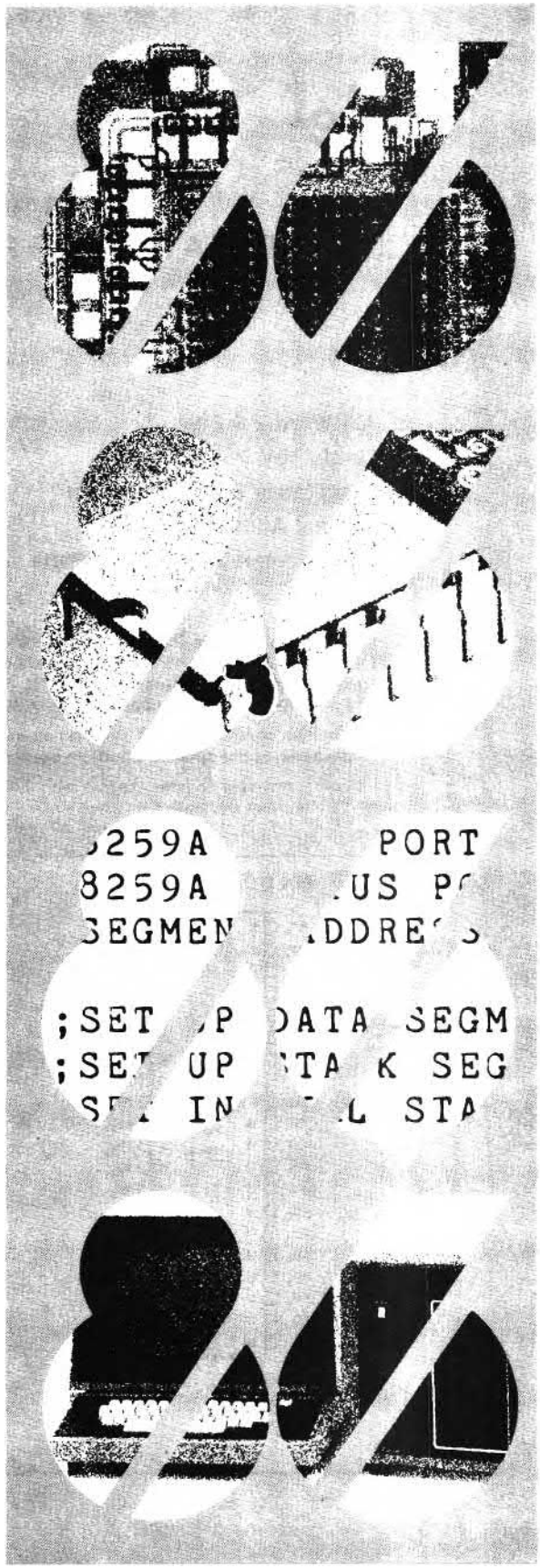


INTA SEQUENCE



Appendix B
Device Specifications

- 8086 Family
- **8085 Peripherals***
- Standard Peripherals**
- RAM Memories***
- EPROM Memories***
- Development Tools



8155/8156/8155-2/8156-2
2048 BIT STATIC MOS RAM WITH I/O PORTS AND TIMER

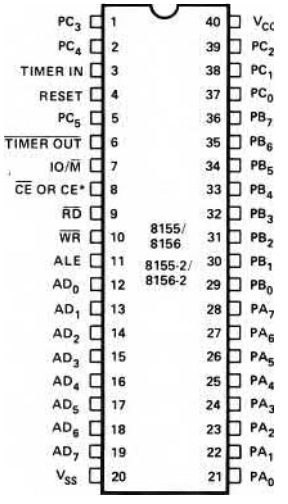
- **256 Word x 8 Bits**
- **Single +5V Power Supply**
- **Completely Static Operation**
- **Internal Address Latch**
- **2 Programmable 8 Bit I/O Ports**
- **1 Programmable 6-Bit I/O Port**
- **Programmable 14-Bit Binary Counter/ Timer**
- **Compatible with 8085A and 8088 CPU**
- **Multiplexed Address and Data Bus**
- **40 Pin DIP**

The 8155 and 89156 are RAM and I/O chips to be used in the 8085A and 8088 microprocessor systems. The RAM portion is designed with 2048 static cells organized as 256 x 8. They have a maximum access time of 400 ns to permit use with no wait states in 8085A CPU. The 8155-2 and 8156-2 have maximum access times of 330 ns for use with the 8085A-2 and the full speed 5 MHz 8088 CPU.

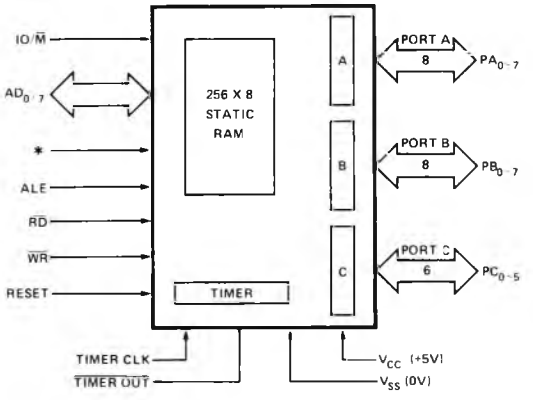
The I/O portion consists of three general purpose I/O ports. One of the three ports can be programmed to be status pins, thus allowing the other two ports to operate in handshake mode.

A 14-bit programmable counter/timer is jalso included on chip to provide either a square wave or terminal count pulse for the CPU system depending on timer mode.

PIN CONFIGURATION



BLOCK DIAGRAM



*: 8155/8155-2 = $\overline{CE}$, 8156/8156-2 = CE

intel®

8185/8185-2
1024 x 8-BIT STATIC RAM FOR MCS-85™

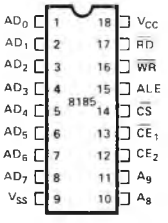
- **Multiplexed Address and Data Bus**
- **Directly Compatible with 8085A and 8088 Microprocessors**
- **Low Operating Power Dissipation**
- **Low Standby Power Dissipation**
- **Single +5V Supply**
- **High Density 18-Pin Package**

The Intel® 8185 is an 8192-bit static random access memory (RAM) organized as 1024 words by 8-bits using N-channel Silicon-Gate MOS technology. The multiplexed address and data bus allows the 8185 to interface directly to the 8085A and 8088 microprocessors to provide a maximum level of system integration.

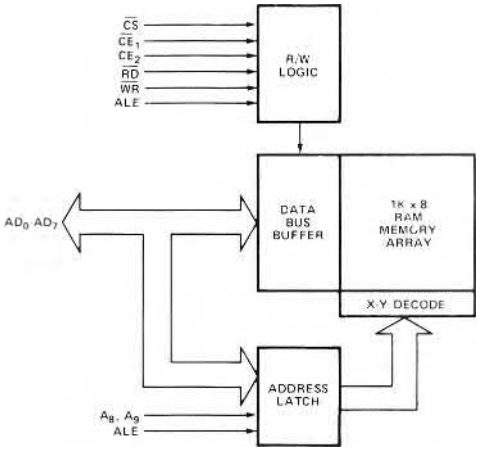
The low standby power dissipation minimizes system power requirements when the 8185 is disabled.

The 8185-2 is a high-speed selected version of the 8185 that is compatible with the 5 MHz 8085A-2 and the full speed 5 MHz 8088.

PIN CONFIGURATION



BLOCK DIAGRAM



PIN NAMES

AD_0 AD_7	ADDRESS/DATA LINES
A_8, A_9	ADDRESS LINES
$\overline{CS}$	CHIP SELECT
$\overline{CE_1}$	CHIP ENABLE (IO/$\overline{M}$)
CE_2	CHIP ENABLE
ALE	ADDRESS LATCH ENABLE
$\overline{RD}$	READ ENABLE
$\overline{WR}$	WRITE ENABLE

8355/8355-2
16,384-BIT ROM WITH I/O

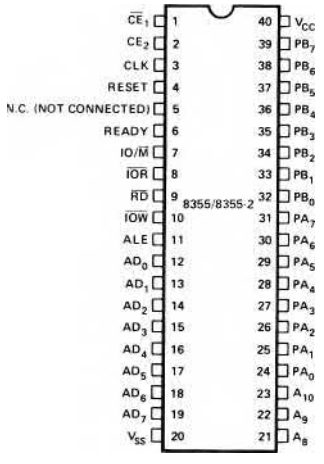
- 2048 Words × 8 Bits
- Single +5V Power Supply
- Directly compatible with 8085A and 8088 Microprocessors
- 2 General Purpose 8-Bit I/O Ports
- Each I/O Port Line Individually Programmable as Input or Output
- Multiplexed Address and Data Bus
- Internal Address Latch
- 40-Pin DIP

The Intel® 8355 is a ROM and I/O chip to be used in the 8085A and 8088 microprocessor systems. The ROM portion is organized as 2048 words by 8 bits. It has a maximum acess time of 400 ns to permit use with no wait states in the 8085A CPU.

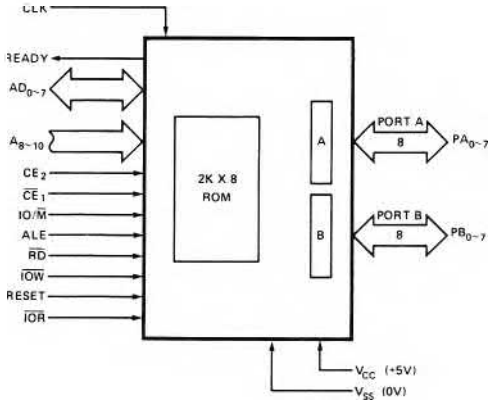
The I/O portion consists of 2 general purpose I/O ports. Each I/O port has 8 port lines and each I/O port line is individually programmable as input or output.

The 8355-2 has a 300ns access time for compatibility with the 8085A-2 and full speed 5 MHz 8088 microprocessors.

PIN CONFIGURATION



BLOCK DIAGRAM



8755A/8755A-2
16,384-BIT EPROM WITH I/O

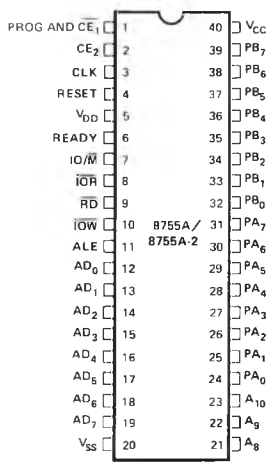
- 2048 Words × 8 Bits
- Single +5V Power Supply (V_{CC})
- Directly Compatible with 8085A and 8088 Microprocessors
- U.V. Erasable and Electrically Reprogrammable
- Internal Address Latch
- 2 General Purpose 8-Bit I/O Ports
- Each I/O Port Line Individually Programmable as Input or Output
- Multiplexed Address and Data Bus
- 40-Pin DIP

The Intel® 8755A is an erasable and electrically reprogrammable ROM (EPROM) and I/O chip to be used in the 8085A and 8088 microprocessor systems. The EPROM portion is organized as 2048 words by 8 bits. It has a maximum access time of 450 ns to permit use with no wait states in an 8085A CPU.

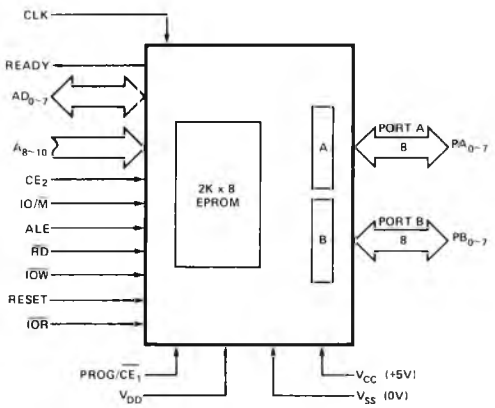
The I/O portion consists of 2 general purpose I/O ports. Each I/O port has 8 port lines, and each I/O port line is individually programmable as input or output.

The 8755A-2 is a high speed selected version of the 8755A compatible with the 5 MHz 8085A-2 and the full speed 5 MHz 8088.

PIN CONFIGURATION



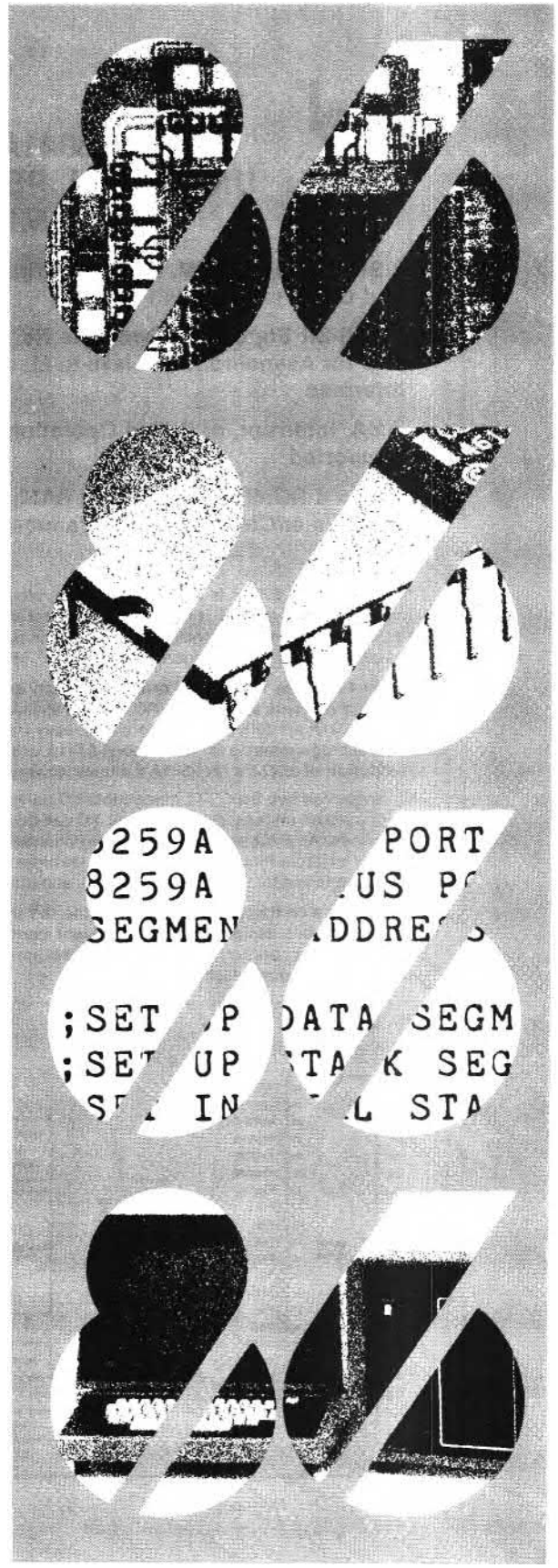
BLOCK DIAGRAM



Appendix B
Device Specifications

- 8086 Family
- 8085 Peripherals*
- **Standard Peripherals****
- RAM Memories***
- EPROM Memories***
- Development Tools

*For complete specifications refer to the Intel MCS-85 User's Manual.
**For complete specifications refer to the Intel Peripheral Design Handbook.
***For complete specifications refer to the 1979 Intel Component Data Catalog.



8041A/8641A/8741A
UNIVERSAL PERIPHERAL INTERFACE
8-BIT MICROCOMPUTER

- **8-Bit CPU plus ROM, RAM, I/O, Timer and Clock in a Single Package**
- **One 8-Bit Status and Two Data Registers for Asynchronous Slave-to-Master Interface**
- **DMA, Interrupt, or Polled Operation Supported**
- **1024 × 8 ROM/EPROM, 64 × 8 RAM, 8-Bit Timer/Counter, 18 Programmable I/O Pins**
- **Fully Compatible with MCS-48™, MCS-80™, MCS-85™, and MCS-86™ Microprocessor Families**
- **Interchangeable ROM and EPROM Versions**
- **3.6 MHz 8741A-8 Available**
- **Expandable I/O**
- **RAM Power-Down Capability**
- **Over 90 Instructions: 70% Single Byte**
- **Single 5V Supply**

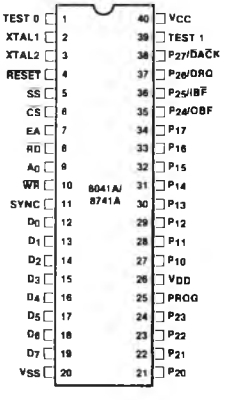
The Intel® 8041A/8741A is a general purpose, programmable interface device designed for use with a variety of 8-bit microprocessor systems. It contains a low cost microcomputer with program memory, data memory, 8-bit CPU, I/O ports, timer/counter, and clock in a single 40-pin package. Interface registers are included to enable the UPI device to function as a peripheral controller in MCS-48™, MCS-80™, MCS-85™, MCS-86™, and other 8-bit systems.

The UPI-41A™ has 1K words of program memory and 64 words of data memory on-chip. To allow full user flexibility the program memory is available as ROM in the 8041A version or as UV-erasable EPROM in the 8741A version. The 8741A and the 8041A are fully pin compatible for easy transition from prototype to production level designs. The 8641A is a one-time programmable (at the factory) 8741A which can be ordered as the first 25 pieces of a new 8041A order. The substitution of 8641A's for 8041A's allows for very fast turnaround for initial code verification and evaluation results.

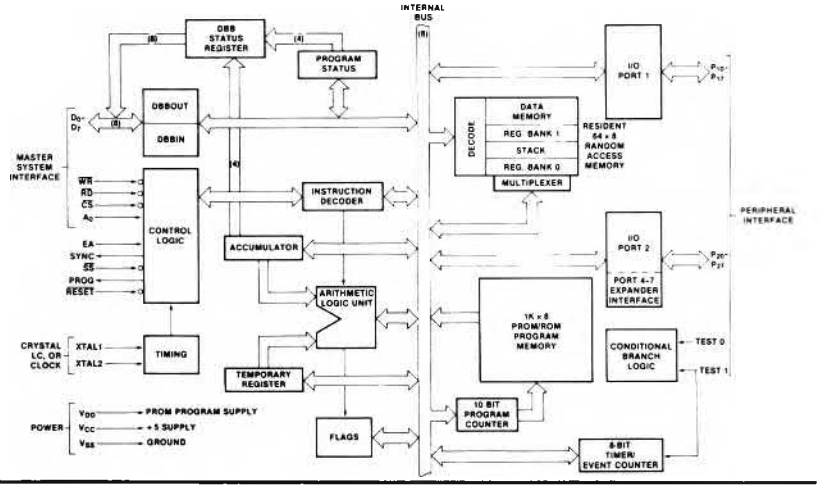
The device has two 8-bit, TTL compatible I/O ports and two test inputs. Individual port lines can function as either inputs or outputs under software control. I/O can be expanded with the 8243 device which is directly compatible and has 16 I/O lines. An 8-bit programmable timer/counter is included in the UPI device for generating timing sequences or counting external inputs. Additional UPI features include: single 5V supply, low power standby mode (in the 8041A), single-step mode for debug (in the 8741A), and dual working register banks.

Because it's a complete microcomputer, the UPI provides more flexibility for the designer than conventional LSI interface devices. It is designed to be an efficient controller as well as an arithmetic processor. Applications include keyboard scanning, printer control, display multiplexing and similar functions which involve interfacing peripheral devices to microprocessor systems.

PIN CONFIGURATION



BLOCK DIAGRAM



intel®

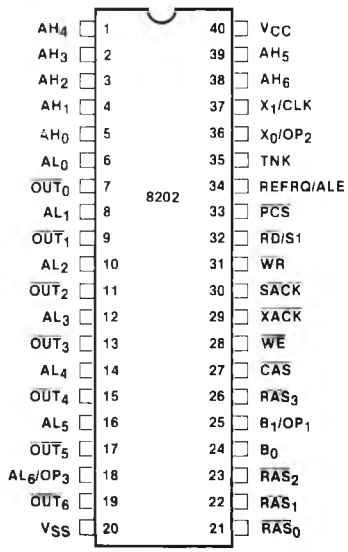
PRELIMINARY
Caution: This is not a final specification. Some parameters are subject to change.

8202
DYNAMIC RAM CONTROLLER

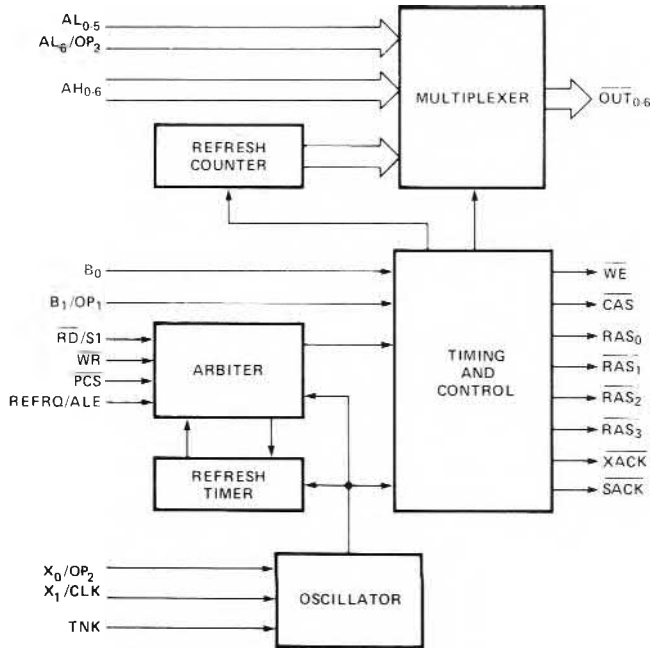
- **Provides All Signals Necessary to Control 2104A, 2117, or 2118 Dynamic Memories**
- **Directly Addresses and Drives Up to 128K Bytes Without External Drivers**
- **Provides Address Multiplexing and Strobes**
- **Provides a Refresh Timer and a Refresh Counter**
- **Refresh Cycles May be Internally or Externally Requested**
- **Provides Transparent Refresh Capability**
- **Fully Compatible with Intel® 8080A, 8085A and 8086 Microprocessors**
- **Decodes 8085A Status for Advanced Read Capability**
- **Provides System Acknowledge and Transfer Acknowledge Signals**
- **Internal or External Clock Capability**

The 8202 is a Dynamic RAM System Controller designed to provide all signals necessary to use 2104A, 2117, or 2118 Dynamic RAMs in microcomputer systems. The 8202 provides multiplexed addresses and address strobes, as well as refresh/access arbitration. Refresh cycles can be started internally or externally.

PIN CONFIGURATION



8202 BLOCK DIAGRAM



8205

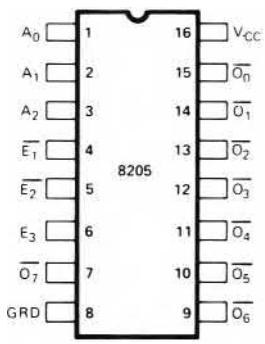
HIGH SPEED 1 OUT OF 8 BINARY DECODER

- **I/O Port or Memory Selector**
- **Simple Expansion — Enable Inputs**
- **High Speed Schottky Bipolar Technology — 18ns Max. Delay**
- **Directly Compatible with TTL Logic Circuits**
- **Low Input Load Current — .25 mA max., 1/6 Standard TTL Input Load**
- **Minimum Line Reflection — Low Voltage Diode Input Clamp**
- **Outputs Sink 10 mA min.**
- **16-Pin Dual-In-Line Ceramic or Plastic Package**

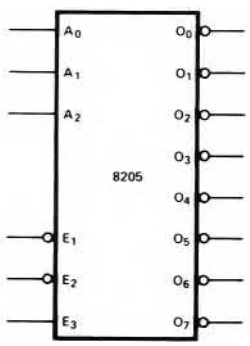
The 8205 decoder can be used for expansion of systems which utilize input ports, output ports, and memory components with active low chip select input. When the 8205 is enabled, one of its eight outputs goes "low", thus a single row of a memory system is selected. The 3 chip enable inputs on the 8205 allow easy system expansion. For very large systems, 8205 decoders can be cascaded such that each decoder can drive eight other decoders for arbitrary memory expansions.

The Intel®8205 is packaged in a standard 16 pin dual-in-line package; and its performance is specified over the temperature range of 0°C to +75°C, ambient. The use of Schottky barrier diode clamped transistors to obtain fast switching speeds results in higher performance than equivalent devices made with a gold diffusion process.

PIN CONFIGURATION



LOGIC SYMBOL



PIN NAMES

A_0 A_2	ADDRESS INPUTS
$\overline{E_1}$ E_3	ENABLE INPUTS
$\overline{O_0}$ $\overline{O_7}$	DECODED OUTPUTS

ADDRESS			ENABLE			OUTPUTS							
A_0	A_1	A_2	$\overline{E_1}$	$\overline{E_2}$	E_3	$\overline{0}$	$\overline{1}$	$\overline{2}$	$\overline{3}$	$\overline{4}$	$\overline{5}$	$\overline{6}$	$\overline{7}$
L	L	L	L	L	H	L	H	H	H	H	H	H	H
H	L	L	L	L	H	H	L	H	H	H	H	H	H
L	H	L	L	L	H	H	H	L	H	H	H	H	H
H	H	L	L	L	H	H	H	H	L	H	H	H	H
L	L	H	L	L	H	H	H	H	H	L	H	H	H
H	L	H	L	L	H	H	H	H	H	H	L	H	H
L	H	H	L	L	H	H	H	H	H	H	H	L	H
H	H	H	L	L	H	H	H	H	H	H	H	H	L
X	X	X	L	L	L	H	H	H	H	H	H	H	H
X	X	X	H	L	L	H	H	H	H	H	H	H	H
X	X	X	L	H	L	H	H	H	H	H	H	H	H
X	X	X	H	H	L	H	H	H	H	H	H	H	H
X	X	X	H	L	H	H	H	H	H	H	H	H	H
X	X	X	L	H	H	H	H	H	H	H	H	H	H
X	X	X	H	H	H	H	H	H	H	H	H	H	H

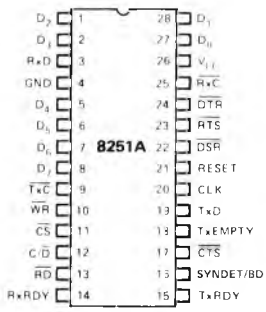
intel®

8251A
PROGRAMMABLE COMMUNICATION INTERFACE

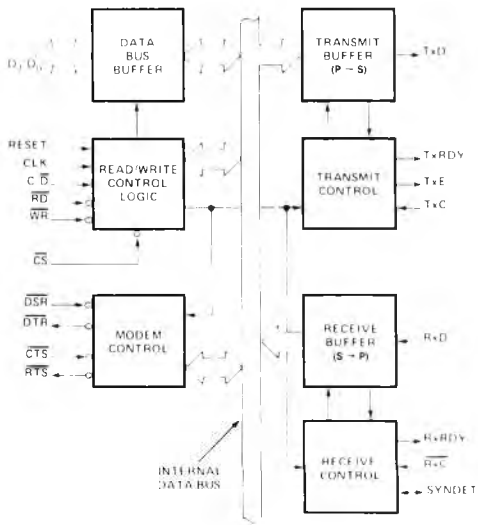
- **Synchronous and Asynchronous Operation**
- **Synchronous 5-8 Bit Characters; Internal or External Character Synchronization; Automatic Sync Insertion**
- **Asynchronous 5-8 Bit Characters; Clock Rate—1, 16 or 64 Times Baud Rate; Break Character Generation; 1, 1½, or 2 Stop Bits; False Start Bit Detection; Automatic Break Detect and Handling.**
- **Synchronous Baud Rate — DC to 64K Baud**
- **Asynchronous Baud Rate — DC to 19.2K Baud**
- **Full Duplex, Double Buffered, Transmitter and Receiver**
- **Error Detection — Parity, Overrun and Framing**
- **Fully Compatible with 8080/8085 CPU**
- **28-Pin DIP Package**
- **All Inputs and Outputs are TTL Compatible**
- **Single +5V Supply**
- **Single TTL Clock**

The Intel® 8251A is the enhanced version of the industry standard, Intel® 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART), designed for data communications with Intel's new high performance family of microprocessors such as the 8085. The 8251A is used as a peripheral device and is programmed by the CPU to operate using virtually any serial data transmission technique presently in use (including IBM "bi-sync"). The USART accepts data characters from the CPU in parallel format and then converts them into a continuous serial data stream for transmission. Simultaneously, it can receive serial data streams and convert them into parallel data characters for the CPU. The USART will signal the CPU whenever it can accept a new character for transmission or whenever it has received a character for the CPU. The CPU can read the complete status of the USART at any time. These include data transmission errors and control signals such as SYNDET, TxEMPTY. The chip is constructed using N-channel silicon gate technology.

PIN CONFIGURATION



BLOCK DIAGRAM



PIN NAMES

D7 D0	Data Bus (8 bits)
C/D	Control or Data is to be Written or Read
RD	Read Data Command
WR	Write Data or Control Command
CS	Chip Enable
CLK	Clock Pulse (TTL)
RESET	Reset
TxC	Transmitter Clock
TxD	Transmitter Data
RxC	Receiver Clock
RxD	Receiver Data
RxRDY	Receiver Ready (has character for 8080)
TxRDY	Transmitter Ready (ready for char. from 8080)

DSR	Data Set Ready
DTR	Data Terminal Ready
SYNDET/BD	Sync Detect/ Break Detect
RTS	Request to Send Data
CTS	Clear to Send Data
TxE	Transmitter Empty
VCC	+5 Volt Supply
GND	Ground

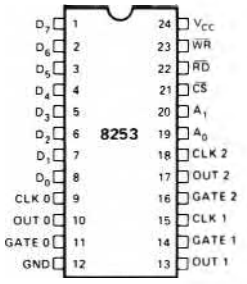
8253/8253-5
PROGRAMMABLE INTERVAL TIMER

- **MCS—85™ Compatible 8253-5**
- **3 Independent 16-Bit Counters**
- **DC to 2 MHz**
- **Programmable Counter Modes**
- **Count Binary or BCD**
- **Single +5V Supply**
- **24-Pin Dual In-Line Package**

The Intel® 8253 is a programmable counter/timer chip designed for use as an Intel microcomputer peripheral. It uses nMOS technology with a single +5V supply and is packaged in a 24-pin plastic DIP.

It is organized as 3 independent 16-bit counters, each with a count rate of up to 2 MHz. All modes of operation are software programmable.

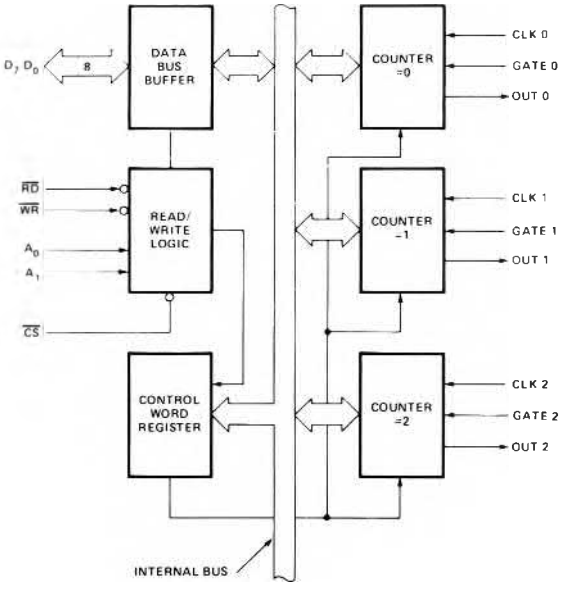
PIN CONFIGURATION



PIN NAMES

D_7 D_0	DATA BUS (8 BIT)
CLK N	COUNTER CLOCK INPUTS
GATE N	COUNTER GATE INPUTS
OUT N	COUNTER OUTPUTS
$\overline{RD}$	READ COUNTER
$\overline{WR}$	WRITE COMMAND OR DATA
$\overline{CS}$	CHIP SELECT
A_0 A_1	COUNTER SELECT
V_{CC}	+5 VOLTS
GND	GROUND

BLOCK DIAGRAM

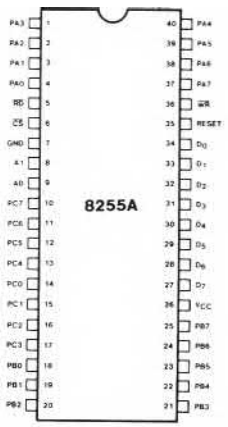


8255A/8255A-5
PROGRAMMABLE PERIPHERAL INTERFACE

- **MCS-85™ Compatible 8255A-5**
- **24 Programmable I/O Pins**
- **Completely TTL Compatible**
- **Fully Compatible with Intel® Microprocessor Families**
- **Improved Timing Characteristics**
- **Direct Bit Set/Reset Capability Easing Control Application Interface**
- **40-Pin Dual In-Line Package**
- **Reduces System Package Count**
- **Improved DC Driving Capability**

The Intel® 8255A is a general purpose programmable I/O device designed for use with Intel® microprocessors. It has 24 I/O pins which may be individually programmed in 2 groups of 12 and used in 3 major modes of operation. In the first mode (MODE 0), each group of 12 I/O pins may be programmed in sets of 4 to be input or output. In MODE 1, the second mode, each group may be programmed to have 8 lines of input or output. Of the remaining 4 pins, 3 are used for handshaking and interrupt control signals. The third mode of operation (MODE 2) is a bidirectional bus mode which uses 8 lines for a bidirectional bus, and 5 lines, borrowing one from the other group, for handshaking.

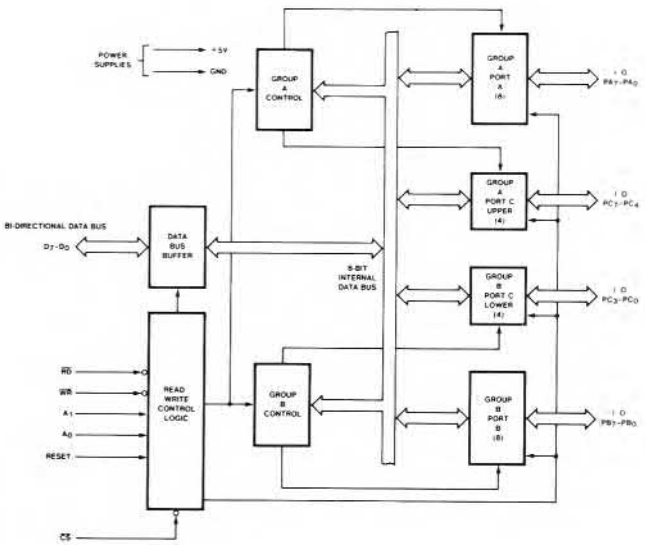
PIN CONFIGURATION



PIN NAMES

D_7–D_0	DATA BUS (BI DIRECTIONAL)
RESET	RESET INPUT
$\overline{CS}$	CHIP SELECT
$\overline{RD}$	READ INPUT
$\overline{WR}$	WRITE INPUT
A0, A1	PORT ADDRESS
PA7-PA0	PORT A (BIT)
PB7-PB0	PORT B (BIT)
PC7-PC0	PORT C (BIT)
V_{CC}	+5 VOLTS
GND	0 VOLTS

8255A BLOCK DIAGRAM

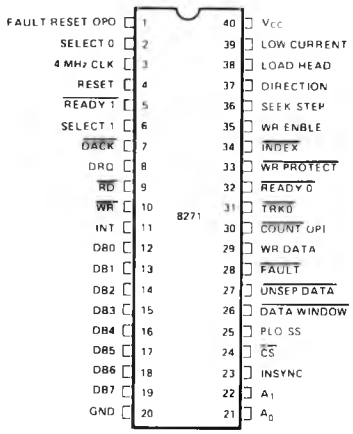


8271/8271-6/8271-8
PROGRAMMABLE FLOPPY DISK CONTROLLER

- **IBM 3740 Soft Sectored Format Compatible**
- **Programmable Record Lengths**
- **Multi-Sector Capability**
- **Maintain Dual Drives with Minimum Software Overhead Expandable to 4 Drives**
- **Automatic Read/Write Head Positioning and Verification**
- **Internal CRC Generation and Checking**
- **Programmable Step Rate, Settle-Time, Head Load Time, Head Unload Index Count**
- **Fully MCS-80™ and MCS-85™ Compatible**
- **Single +5V Supply**
- **40-Pin Package**

The Intel® 8271 Programmable Floppy Disk Controller (FDC) is an LSI component designed to interface one to 4 floppy disk drives to an 8-bit microcomputer system. Its powerful control functions minimize both hardware and software overhead normally associated with floppy disk controllers.

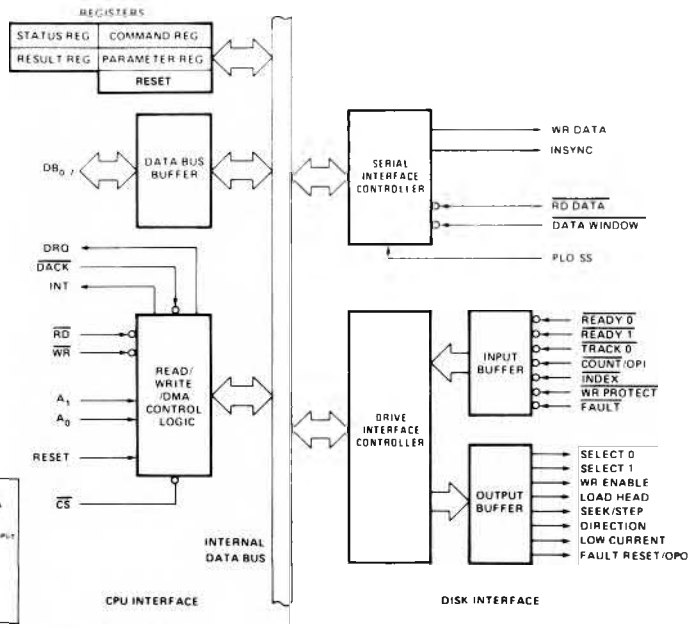
PIN CONFIGURATION



PIN NAMES

DB_7 - DB_0	DATA BUS (BI-DIRECTIONAL)	PLO SS	PLO SINGLE SHOT
CLK	CLOCK INPUT (TTL)	DATA WINDOW	DATA WINDOW
SELECT 1 0	SELECT 1 0	UNSEP DATA	UNSEPARATED DATA
FAULT RESET/OPO	FAULT RESET/OPTIONAL OUTPUT	FAULT	FAULT
RESET	CHIP RESET	WR DATA	WRITE DATA
READY 1 0	READY 1 0	COUNT OPI	COUNT-OPTIONAL INPUT
DACK	DMA ACKNOWLEDGE	TRK 0	TRACK 0
DRQ	DMA REQUEST	WR PROTECT	WRITE PROTECT
RD	CPU READ INPUT	INDEX	INDEX
WR	CPU WRITE INPUT	WR ENABLE	WRITE ENABLE
INT	INTERRUPT	SEEK/STEP	SEEK/STEP
A1 0	REGISTER SELECT	DIRECTION	DIRECTION
INSYNC	READ DATA INSYNC	LOAD HEAD	LOAD HEAD
CS	CHIP SELECT	LOW CURRENT	LOW CURRENT

BLOCK DIAGRAM



intel®

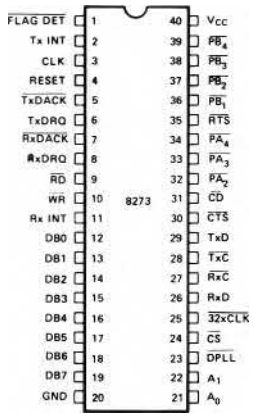
PRELIMINARY

8273
PROGRAMMABLE HDLC/SDLC PROTOCOL CONTROLLER

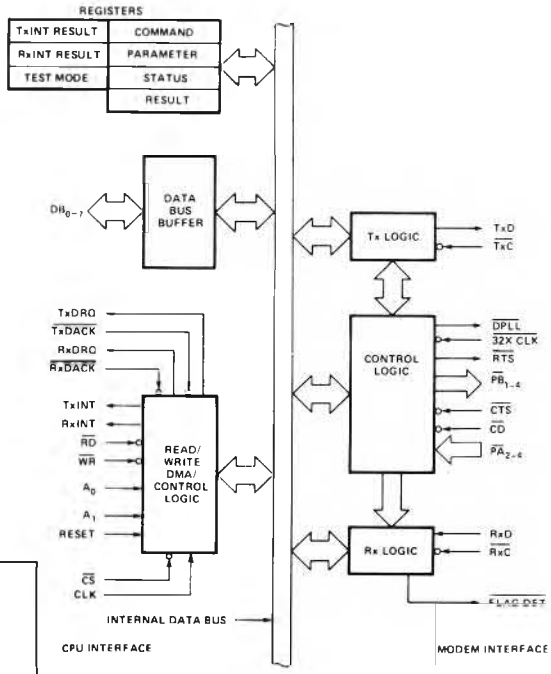
- **HDLC/SDLC Compatible**
- **Frame Level Commands**
- **Full Duplex, Half Duplex, or Loop SDLC Operation**
- **Up to 64K Baud Transfers**
- **Two User Programmable Modem Control Ports**
- **Automatic FCS (CRC) Generation and Checking**
- **Programmable NRZI Encode/Decode**
- **N-Bit Reception Capability**
- **Digital Phase Locked Loop Clock Recovery**
- **Minimum CPU Overhead**
- **Fully Compatible with 8080/8085 CPUs**
- **Single +5V Supply**
- **40-Pin Package**

The Intel® 8273 Programmable HDLC/SDLC Protocol Controller is a dedicated device designed to support the ISO/C-CITT's HDLC and IBM's SDLC communication line protocols. It is fully compatible with Intel's new high performance microcomputer systems such as the MCS-85™. A frame level command set is achieved by a unique microprogrammed dual processor chip architecture. The processing capability supported by the 8273 relieves the system CPU of the low level real-time tasks normally associated with controllers.

PIN CONFIGURATION



BLOCK DIAGRAM



PIN NAMES

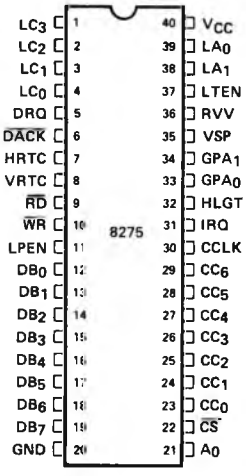
DB0–DB7	DATA BUS (8 BITS)	$\overline{CS}$	CHIP SELECT
$\overline{FLAG\ DET}$	FLAG DETECT	32xCLK	32 TIMES CLOCK
TxINT	TRANSMITTER INTERRUPT	Rx D	RECEIVER DATA
CLK	CLOCK INPUT	$\overline{Rx\ C}$	RECEIVER CLOCK
RESET	RESET	Tx C	TRANSMITTER CLOCK
$\overline{Tx\ DACK}$	TRANSMITTER DMA ACKNOWLEDGE	Tx D	TRANSMITTER DATA
TxDRQ	TRANSMITTER DMA REQUEST	$\overline{CTS}$	CLEAR TO SEND
$\overline{RD}$	READ INPUT	$\overline{CD}$	CARRIER DETECT
$\overline{WR}$	WRITE INPUT	$\overline{PA_2}$–$\overline{PA_4}$	GP INPUT PORTS
$\overline{Rx\ DACK}$	RECEIVER DMA ACKNOWLEDGE	$\overline{PB_1}$–$\overline{PB_4}$	GP OUTPUT PORTS
Rx DRQ	RECEIVER DMA REQUEST	$\overline{RTS}$	REQUEST TO SEND
Rx INT	RECEIVER INTERRUPT	V_{CC}	+5 VOLT SUPPLY
A0–A1	COMMAND REGISTER SELECT ADDRESS	GND	GROUND
$\overline{DPLL}$	DIGITAL PHASE LOCKED LOOP		

8275
PROGRAMMABLE CRT CONTROLLER

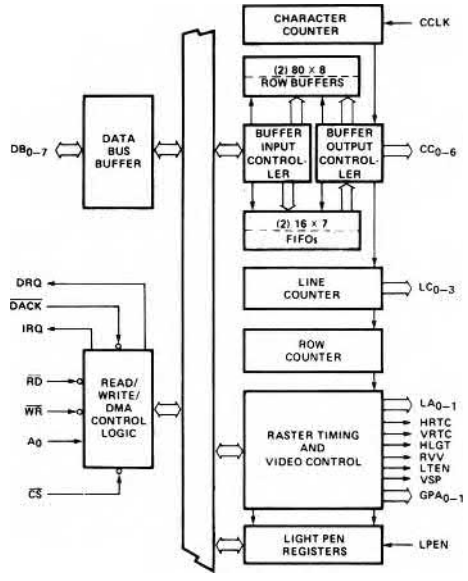
- **Programmable Screen and Character Format**
- **6 Independent Visual Field Attributes**
- **11 Visual Character Attributes (Graphic Capability)**
- **Cursor Control (4 Types)**
- **Light Pen Detection and Registers**
- **Fully MCS-80™ and MCS-85™ Compatible**
- **Dual Row Buffers**
- **Programmable DMA Burst Mode**
- **Single +5V Supply**
- **40-Pin Package**

The Intel® 8275 Programmable CRT Controller is a single chip device to interface CRT raster scan displays with Intel® microcomputer systems. Its primary function is to refresh the display by buffering the information from main memory and keeping track of the display position of the screen. The flexibility designed into the 8275 will allow simple interface to almost any raster scan CRT display with a minimum of external hardware and software overhead.

PIN CONFIGURATION



BLOCK DIAGRAM



PIN NAMES

DB_{0-1}	BI-DIRECTIONAL DATA BUS	LC_{0-3}	LINE COUNTER OUTPUTS
DRQ	DMA REQUEST OUTPUT	LA_{0-1}	LINE ATTRIBUTE OUTPUTS
$\overline{DACK}$	DMA ACKNOWLEDGE INPUT	HRTC	HORIZONTAL RETRACE OUTPUT
IRQ	INTERRUPT REQUEST OUTPUT	VRTC	VERTICAL RETRACE OUTPUT
$\overline{RD}$	READ STROBE INPUT	HLGT	HIGHLIGHT OUTPUT
$\overline{WR}$	WRITE STROBE INPUT	RVV	REVERSE VIDEO OUTPUT
A_0	REGISTER ADDRESS INPUT	LTEN	LIGHT ENABLE OUTPUT
CS	CHIP SELECT INPUT	VSP	VIDEO SUPPRESS OUTPUT
CCLK	CHARACTER CLOCK INPUT	GPA_{0-1}	GENERAL PURPOSE ATTRIBUTE OUTPUTS
CC_{0-6}	CHARACTER CODE OUTPUTS	LPEN	LIGHT PEN INPUT

intel®

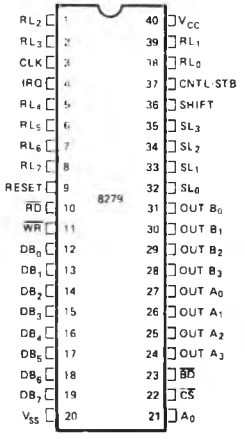
8279/8279-5
PROGRAMMABLE KEYBOARD/DISPLAY INTERFACE

- **MCS-85™ Compatible 8279-5**
- **Simultaneous Keyboard Display Operations**
- **Scanned Keyboard Mode**
- **Scanned Sensor Mode**
- **Strobed Input Entry Mode**
- **8-Character Keyboard FIFO**
- **2-Key Lockout or N-Key Rollover with Contact Debounce**
- **Dual 8- or 16-Numerical Display**
- **Single 16-Character Display**
- **Right or Left Entry 16-Byte Display RAM**
- **Mode Programmable from CPU**
- **Programmable Scan Timing**
- **Interrupt Output on Key Entry**

The Intel® 8279 is a general purpose programmable keyboard and display I/O interface device designed for use with Intel® microprocessors. The keyboard portion can provide a scanned interface to a 64-contact key matrix. The keyboard portion will also interface to an array of sensors or a strobed interface keyboard, such as the hall effect and ferrite variety. Key depressions can be 2-key lockout or N-key rollover. Keyboard entries are debounced and strobed in an 8-character FIFO. If more than 8 characters are entered, overrun status is set. Key entries set the interrupt output line to the CPU.

The display portion provides a scanned display interface for LED, incandescent, and other popular display technologies. Both numeric and alphanumeric segment displays may be used as well as simple indicators. The 8279 has 16X8 display RAM which can be organized into dual 16X4. The RAM can be loaded or interrogated by the CPU. Both right entry, calculator and left entry typewriter display formats are possible. Both read and write of the display RAM can be done with auto-increment of the display RAM address.

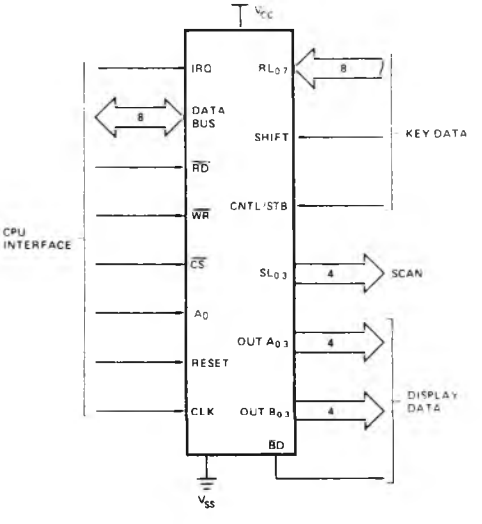
PIN CONFIGURATION



PIN NAMES

DB_{0-7}	I/O	DATA BUS (BI-DIRECTIONAL)
CLK	I	CLOCK INPUT
RESET	I	RESET INPUT
$\overline{CS}$	I	CHIP SELECT
$\overline{RD}$	I	READ INPUT
$\overline{WR}$	I	WRITE INPUT
A_0	I	BUFFER ADDRESS
IRQ	O	INTERRUPT REQUEST OUTPUT
SL_{0-3}	O	SCAN LINES
RL_{0-7}	I	RETURN LINES
SHIFT	I	SHIFT INPUT
CNTL/STB	I	CONTROL STROBE INPUT
OUT A_{0-3}	O	DISPLAY (A) OUTPUTS
OUT B_{0-3}	O	DISPLAY (B) OUTPUTS
$\overline{BD}$	O	BLANK DISPLAY OUTPUT

LOGIC SYMBOL



intel

PRELIMINARY

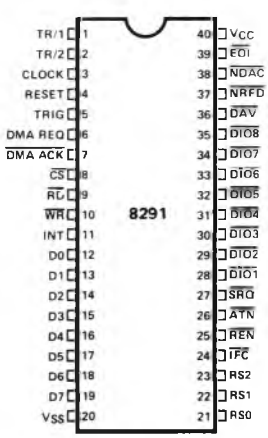
Notice: This is not a final specification. Some parametric limits are subject to change.

8291
GPIB TALKER/LISTENER

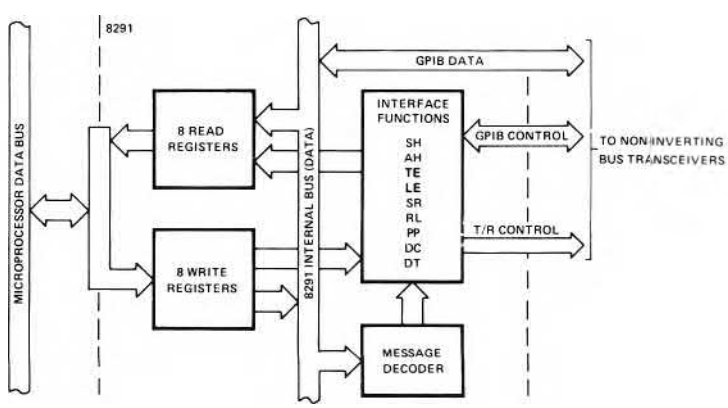
- Designed to Interface Microprocessors (e.g., 8080, 8085, 8086, 8048) to an IEEE Standard 488 Digital Interface Bus
- Programmable Data Transfer Rate
- Complete Source and Acceptor Handshake
- Complete Talker and Listener Functions with Extended Addressing
- Service Request, Parallel Poll, Device Clear, Device Trigger, Remote/Local Functions
- Selectable Interrupts
- On-Chip Primary and Secondary Address Recognition
- Automatic Handling of Addressing and Handshake Protocol
- Provision for Software Implementation of Additional Features
- 1 – 8 MHz Clock Range
- 16 Registers (8 Read, 8 Write), 2 for Data Transfer, the Rest for Interface Function Control, Status, etc.
- Directly Interfaces to External Non-Inverting Transceivers for Connection to the GPIB
- Provides Three Addressing Modes, Allowing the Chip to be Addressed Either as a Major or a Minor Talker/Listener with Primary or Secondary Addressing
- DMA Handshake Provision Allows for Bus Transfers without CPU Intervention
- Trigger Output Pin
- On-Chip EOS (End of Sequence) Message Recognition Facilitates Handling of Multi-Byte Transfers

The 8291 GPIB Talker/Listener is a microprocessor-controlled chip designed to interface microprocessors (e.g., 8048, 8080, 8085, 8086) to an IEEE Standard 488 Instrumentation Interface Bus. It implements all of the Standard's interface functions except for the controller.

PIN CONFIGURATION



BLOCK DIAGRAM



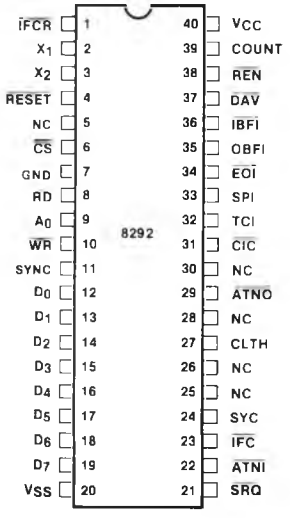
8292
GPIB CONTROLLER

FEATURES:

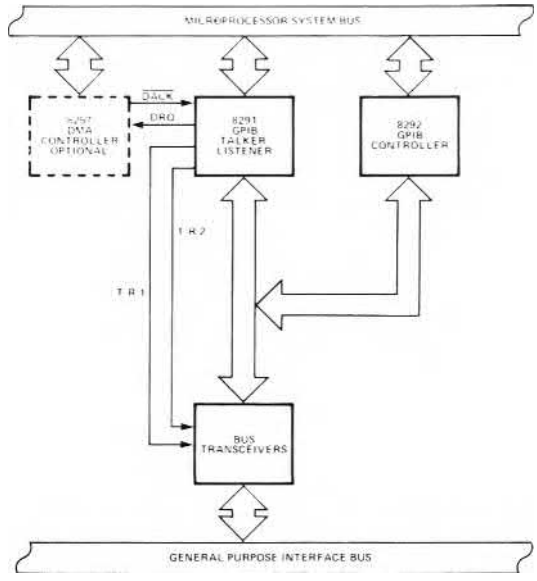
- **Complete IEEE Standard 488 Controller Function.**
- **Interface Clear (IFC) Sending Capability Allows for Seizure of Control and/or Initialization of the Bus.**
- **Responds to Service Requests (SRQ).**
- **Sends (REN), Allowing Instruments to Switch to Remote Control.**
- **Complete Implementation of Transfer Control Protocol.**
- **Synchronous Control Seizure Prevents the Destruction of any Data Transmission in Progress.**
- **Connects with the 8291 to Form a Complete IEEE Standard 488 Interface Talker/Listener/Controller.**

The 8292 GPIB CONTROLLER is a microprocessor-controlled chip designed to connect with the 8291 GPIB TALKER/LISTENER to implement the full IEEE Standard 488 controller function, including transfer control protocol. The 8292 is a pre-programmed UPI-41A.™

PIN CONFIGURATION



8291, 8292 SYSTEM DIAGRAM



intel®

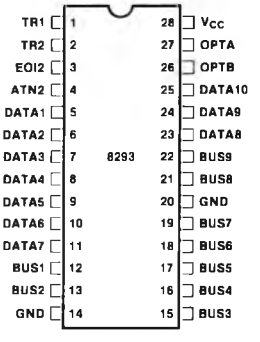
ADVANCE INFORMATION

8293
GPIB TRANSCEIVER

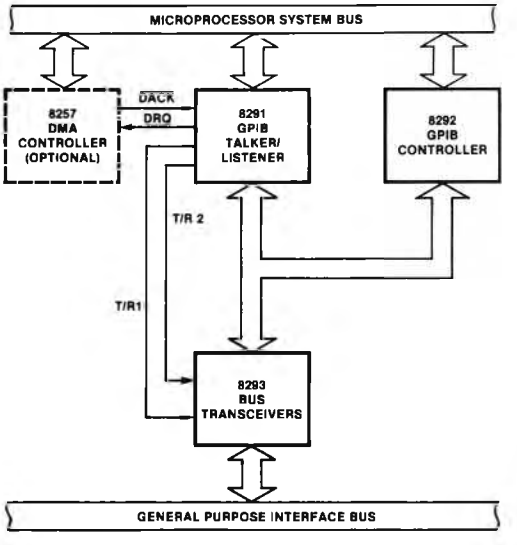
- Nine Open-collector or Three-state Line Drivers
- 48 mA Sink Current Capability on Each Line Driver
- Nine Schmitt-type Line Receivers
- High Capacitance Load Drive Capability
- Single 5V Power Supply
- 28-Pin Package
- Low Power HMOS Design
- On-chip Decoder for Mode Configuration
- Power Up/Power Down Protection to Prevent Disrupting the IEEE Bus
- Connects with the 8291 and 8292 to Form an IEEE Standard 488 Interface Talker/Listener/Controller with no Additional Components
- Only Two 8293's Required per GPIB Interface

The Intel® 8293 GPIB Transceiver is a high current, non-inverting buffer chip designed to interface the 8291 GPIB Talker/Listener or the 8292 GPIB Controller with the 8291 to the IEEE Standard 488-1978 Instrumentation Interface Bus. Each GPIB interface would contain two 8293 Bus Transceivers. In addition, the 8293 can also be used as a general purpose bus driver.

PIN CONFIGURATION



8291, 8292, 8293 SYSTEM DIAGRAM



8294
DATA ENCRYPTION UNIT

- **Certified by National Bureau of Standards**
- **80 Byte/Sec Data Conversion Rate**
- **64-Bit Data Encryption Using 56-Bit Key**
- **DMA Interface**
- **3 Interrupt Outputs to Aid in Loading and Unloading Data**
- **7-Bit User Output Port**
- **Single 5V ± 10% Power Supply**
- **Peripheral to MCS-86™, MCS-85™, MCS-80™ and MCS-48™ Processors**
- **Implements Federal Information Processing Data Encryption Standard**
- **Encrypt and Decrypt Modes Available**

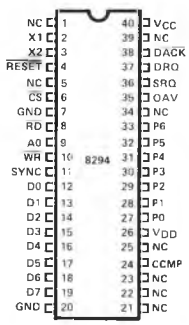
DESCRIPTION

The Intel® 8294 Data Encryption Unit (DEU) is a microprocessor peripheral device designed to encrypt and decrypt 64-bit blocks of data using the algorithm specified in the Federal Information Processing Data Encryption Standard. The DEU operates on 64-bit text words using a 56-bit user-specified key to produce 64-bit cipher words. The operation is reversible: if the cipher word is operated upon, the original text word is produced. The algorithm itself is permanently contained in the 8294; however, the 56-bit key is user-defined and may be changed at any time.

The 56-bit key and 64-bit message data are transferred to and from the 8294 in 8-bit bytes by way of the system data bus. A DMA interface and three interrupt outputs are available to minimize software overhead associated with data transfer. Also, by using the DMA interface two or more DEUs may be operated in parallel to achieve effective system conversion rates which are virtually any multiple of 80 bytes/second. The 8294 also has a 7-bit TTL compatible output port for user-specified functions.

Because the 8294 implements the NBS encryption algorithm it can be used in a variety of Electronic Funds Transfer applications as well as other electronic banking and data handling applications where data must be encrypted.

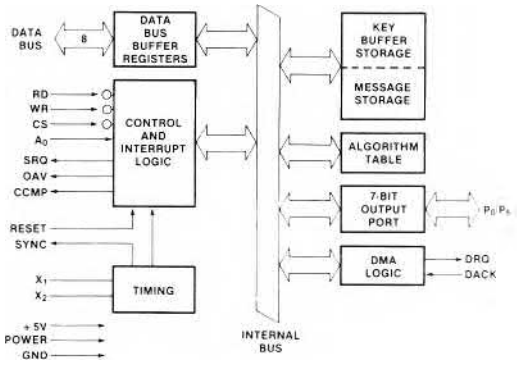
PIN CONFIGURATION



PIN NAMES

PIN NAME	FUNCTION
D_7-D_0	DATA BUS
RD,WR	READ,WRITE STROBES
CS	CHIP SELECT
A_0	CONTROL/DATA SELECT
RESET	RESET INPUT
X_1,X_2	FREQUENCY REFERENCE INPUT
SYNC	HIGH FREQUENCY OUTPUT
DRQ,DACK	DMA REQUEST,DMA ACKNOWLEDGE
SRQ,OAV,CCMP	INTERRUPT REQUEST OUTPUTS
P_6-P_0	OUTPUT PORT LINES
V_{CC},V_{DD},GND	+5V POWER,GND
NC	NO CONNECTION

BLOCK DIAGRAM



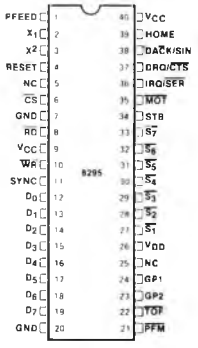
8295
DOT MATRIX PRINTER CONTROLLER

- **Interfaces Dot Matrix Printers to MCS-48™, MCS-80/85™, MCS-86™ Systems**
- **40 Character Buffer On Chip**
- **Serial or Parallel Communication with Host**
- **DMA Transfer Capability**
- **Programmable Character Density (10 or 12 Chararcters/Inch)**
- **Programmable Print Intensity**
- **Single or Double Width Printing**
- **Programmable Multiple Line Feeds**
- **3 Tabulations**
- **2 General Purpose Outputs**

The Intel® 8295 Dot Matrix Printer Controller provides an interface for microprocessors to the LRC 7040 Series dot matrix impact printers. It may also be used as an interface to other similar printers.

The chip may be used in a serial or parallel communication mode with the host processor. In parallel mode, data transfers are based on polling, interrupts, or DMA. Furthermore, it provides internal buffering of up to 40 characters and contains a 7 x 7 matrix character generator accommodating 64 ASCII characters.

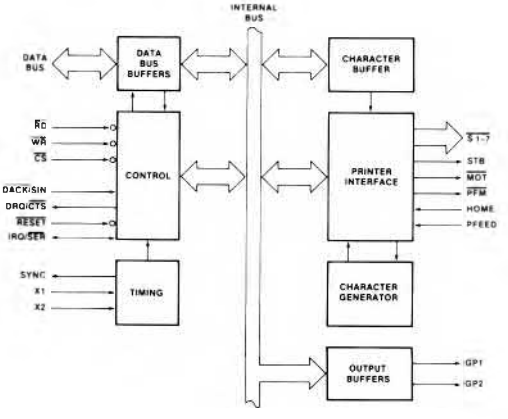
PIN CONFIGURATION



PIN NAMES

PIN NAME	FUNCTION
D_0–D_7	DATA BUS
RD, WR	READ, WRITE STROBES
CS	CHIP SELECT
RESET	RESET INPUT
X_1, X_2	FREQUENCY REFERENCE INPUTS
SYNC	HIGH FREQUENCY OUTPUT
MOT, PFM	MAIN, PAPER FEED MOTOR DRIVES
DRQ, DACK	DMA REQUEST, ACKNOWLEDGE
SIN, CTS	SERIAL INPUT, CLEAR-TO-SEND
IRQ/SER	INTERRUPT REQUEST, SERIAL GROUND
S_1–S_7	SOLENOID DRIVE OUTPUTS
PFEED	PAPER FEED INPUT
HOME, TOF	HOME, TOP-OF-FORM INPUTS
STB	SOLENOID STROBE OUTPUT
GP1, GP2	GENERAL PURPOSE OUTPUTS
VCC, VDD, GND	+5V POWER, GND

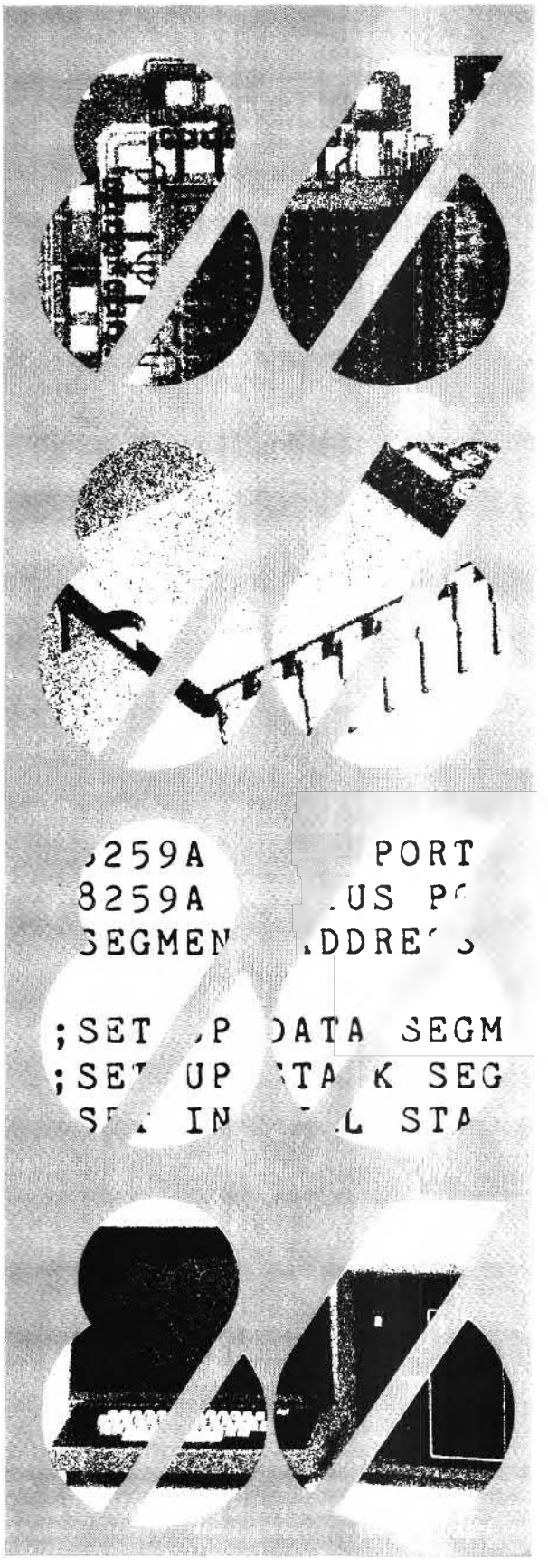
BLOCK DIAGRAM



Appendix B
Device Specifications

- 8086 Family
- 8085 Peripherals*
- Standard Peripherals**
- **RAM Memories*****
- EPROM Memories***
- Development Tools

*For complete specifications refer to the Intel MCS-85 User's Manual.
**For complete specifications refer to the Intel Peripheral Design Handbook.
***For complete specifications refer to the 1979 Intel Component Data Catalog.



2114A
1024 X 4 BIT STATIC RAM

	2114AL-2	2114AL-3	2114AL-4	2114A-4	2114A-5
Max. Access Time (ns)	120	150	200	200	250
Max. Current (mA)	40	40	40	70	70

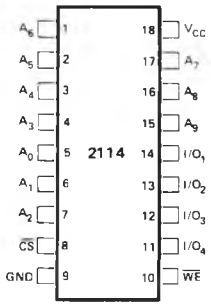
- **HMOS Technology**
- **Low Power, High Speed**
- **Identical Cycle and Access Times**
- **Single +5V Supply ±10%**
- **High Density 18 Pin Package**
- **Completely Static Memory - No Clock or Timing Strobe Required**
- **Directly TTL Compatible: All Inputs and Outputs**
- **Common Data Input and Output Using Three-State Outputs**
- **2114 Replacement**

The Intel 2114A is a 4096-bit static Random Access Memory organized as 1024 words by 4-bits using HMOS, a high performance MOS technology. It uses fully DC stable (static) circuitry throughout, in both the array and the decoding, therefore it requires no clocks or refreshing to operate. Data access is particularly simple since address setup times are not required. The data is read out nondestructively and has the same polarity as the input data. Common input/output pins are provided.

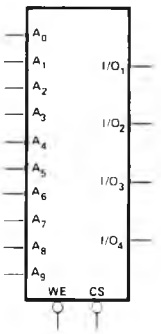
The 2114A is designed for memory applications where the high performance and high reliability of HMOS, low cost, large bit storage, and simple interfacing are important design objectives. The 2114A is placed in an 18-pin package for the highest possible density.

It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply. A separate Chip Select ($\overline{CS}$) lead allows easy selection of an individual package when outputs are or-tied.

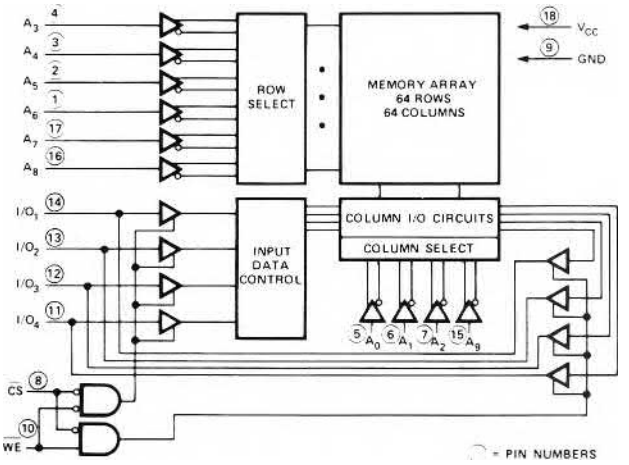
PIN CONFIGURATION



LOGIC SYMBOL



BLOCK DIAGRAM



PIN NAMES

$A_0 - A_9$	ADDRESS INPUTS	V_{CC} POWER (+5V)
$\overline{WE}$	WRITE ENABLE	GND GROUND
$\overline{CS}$	CHIP SELECT	
$I/O_1 - I/O_4$	DATA INPUT/OUTPUT	

2142
1024 X 4 BIT STATIC RAM

	2142-2	2142-3	2142	2142L2	2142L3	2142L
Max. Access Time (ns)	200	300	450	200	300	450
Max. Power Dissipation (mw)	525	525	525	370	370	370

- **High Density 20 Pin Package**
- **Access Time Selections From 200-450ns**
- **Identical Cycle and Access Times**
- **Low Operating Power Dissipation .1mW/Bit Typical**
- **Single +5V Supply**
- **No Clock or Timing Strobe Required**
- **Completely Static Memory**
- **Directly TTL Compatible: All Inputs and Outputs**
- **Common Data Input and Output Using Three-State Outputs**

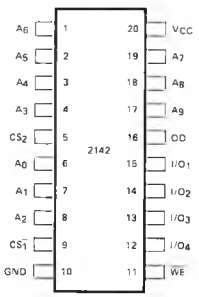
The Intel® 2142 is a 4096-bit static Random Access Memory organized as 1024 words by 4-bits using N-channel Silicon-Gate MOS technology. It uses fully DC stable (static) circuitry throughout — in both the array and the decoding — and therefore requires no clocks or refreshing to operate. Data access is particularly simple since address setup times are not required. The data is read out nondestructively and has the same polarity as the input data. Common input/output pins are provided.

The 2142 is designed for memory applications where high performance, low cost, large bit storage, and simple interfacing are important design objectives. It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply.

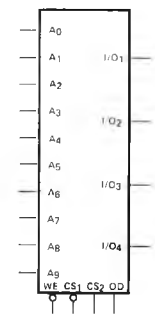
The 2142 is placed in a 20-pin package. Two Chip Selects ($\overline{CS}_1$ and CS_2) are provided for easy and flexible selection of individual packages when outputs are OR-tied. An Output Disable is included for direct control of the output buffers.

The 2142 is fabricated with Intel's N-channel Silicon-Gate technology — a technology providing excellent protection against contamination permitting the use of low cost plastic packaging.

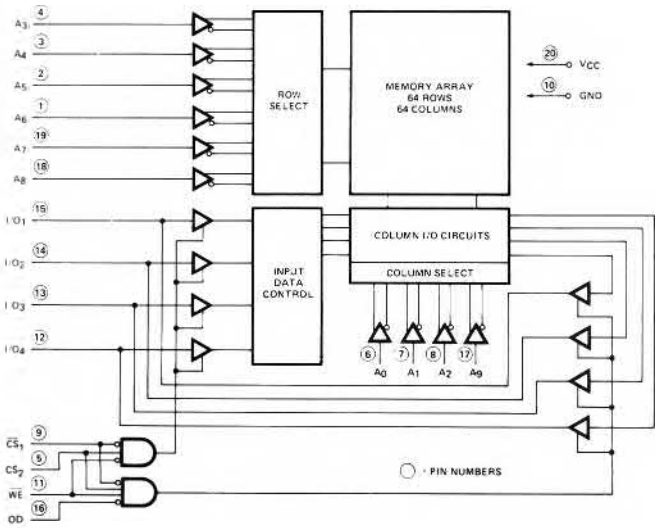
PIN CONFIGURATION



LOGIC SYMBOL



BLOCK DIAGRAM



PIN NAMES

A_0–A_9	ADDRESS INPUTS	OD	OUTPUT DISABLE
$\overline{WE}$	WRITE ENABLE	V_{CC}	POWER (+5V)
$\overline{CS}_1$, CS_2	CHIP SELECT	GND	GROUND
I/O_1–I/O_4	DATA INPUT/OUTPUT		

2148
1024 × 4 BIT STATIC RAM

	2148-3	2148	2148-6
Max. Access Time (ns)	55	70	85
Max. Active Current (mA)	125	125	125
Max. Standby Current (mA)	30	30	30

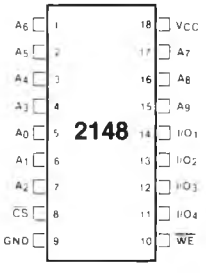
- **HMOS Technology**
- **Completely Static Memory — No Clock or Timing Strobe Required**
- **Equal Access and Cycle Times**
- **Single +5V Supply**
- **Automatic Power-Down**
- **High Density 18-Pin Package**
- **Directly TTL Compatible — All Inputs and Outputs**
- **Common Data Input and Output**
- **Three-State Output**

The Intel® 2148 is a 4096-bit static Random Access Memory organized as 1024 words by 4 bits using HMOS, a high-performance MOS technology. It uses a uniquely innovative design approach which provides the ease-of-use features associated with non-clocked static memories and the reduced standby power dissipation associated with clocked static memories. To the user this means low standby power dissipation without the need for clocks, address setup and hold times, nor reduced data rates due to cycle times that are longer than access times.

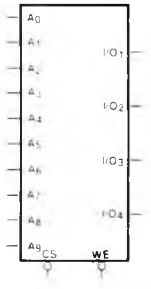
$\overline{CS}$ controls the power-down feature. In less than a cycle time after $\overline{CS}$ goes high — disabling the 2148 — the part automatically reduces its power requirements and remains in this low power standby mode as long as $\overline{CS}$ remains high. This device feature results in system power savings as great as 85% in larger systems, where the majority of devices are disabled.

The 2148 is assembled in an 18-pin package configured with the industry standard 1K × 4 pinout. It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply. The data is read out nondestructively and has the same polarity as the input data.

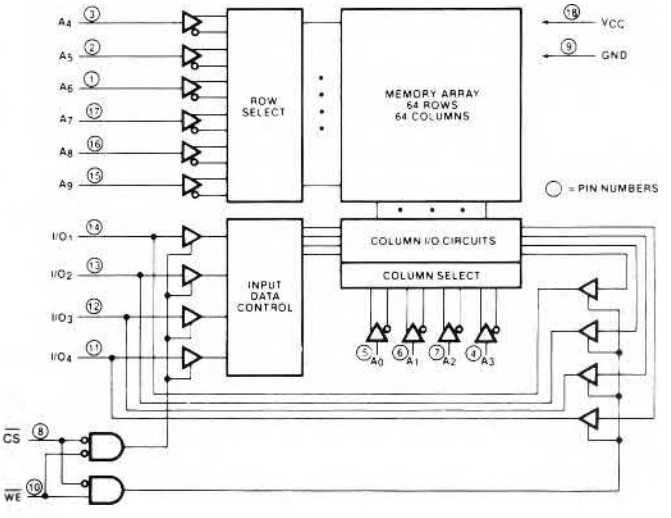
PIN CONFIGURATION



LOGIC SYMBOL



BLOCK DIAGRAM



PIN NAMES

A_0 - A_9	ADDRESS INPUTS
$\overline{WE}$	WRITE ENABLE
$\overline{CS}$	CHIP SELECT
I/O_1 - I/O_4	DATA INPUT/OUTPUT
V_{CC}	POWER (+5V)
GND	GROUND

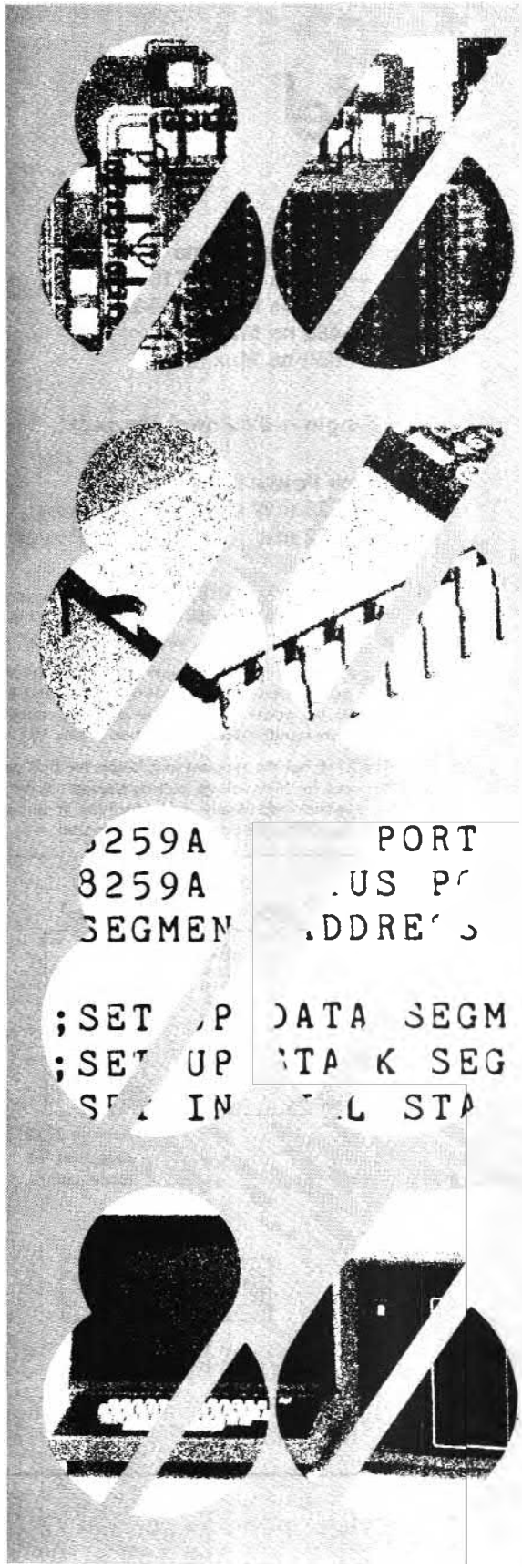
TRUTH TABLE

$\overline{CS}$	$\overline{WE}$	MODE	I/O	POWER
H	X	NOT SELECTED	HIGH-Z	STANDBY
L	L	WRITE	D_{IN}	ACTIVE
L	H	READ	D_{OUT}	ACTIVE

Appendix B
Device Specifications

- 8086 Family
- 8085 Peripherals*
- Standard Peripherals**
- RAM Memories***
- **EPROM Memories*****
- Development Tools

*For complete specifications refer to the Intel MCS-85 User's Manual.
**For complete specifications refer to the Intel Peripheral Design Handbook.
***For complete specifications refer to the 1979 Intel Component Data Catalog.



2716
16K (2K × 8) UV ERASABLE PROM

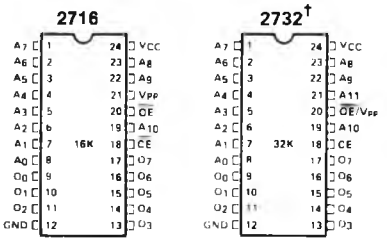
- **Fast Access Time**
  - **— 350 ns Max. 2716-1**
  - **— 390 ns Max. 2716-2**
  - **— 450 ns Max. 2716**
  - **— 650 ns Max. 2716-6**
- **Single +5V Power Supply**
- **Low Power Dissipation**
  - **— 525 mW Max. Active Power**
  - **— 132 mW Max. Standby Power**
- **Pin Compatible to Intel® 2732 EPROM**
- **Simple Programming Requirements**
  - **— Single Location Programming**
  - **— Programs with One 50 ms Pulse**
- **Inputs and Outputs TTL Compatible during Read and Program**
- **Completely Static**

The Intel® 2716 is a 16,384-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The 2716 operates from a single 5-volt power supply, has a static standby mode, and features fast single address location programming. It makes designing with EPROMs faster, easier and more economical.

The 2716, with its single 5-volt supply and with an access time up to 350 ns, is ideal for use with the newer high performance +5V microprocessors such as Intel's 8085 and 8086. The 2716 is also the first EPROM with a static standby mode which reduces the power dissipation without increasing access time. The maximum active power dissipation is 525 mW while the maximum standby power dissipation is only 132 mW, a 75% savings.

The 2716 has the simplest and fastest method yet devised for programming EPROMs — single pulse TTL level programming. No need for high voltage pulsing because all programming controls are handled by TTL signals. Program any location at any time—either individually, sequentially or at random, with the 2716's single address location programming. Total programming time for all 16,384 bits is only 100 seconds.

PIN CONFIGURATION



†Refer to 2732 data sheet for specifications

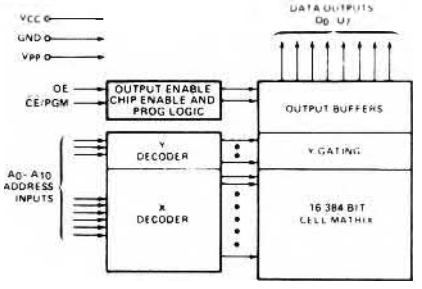
PIN NAMES

$A_0 - A_{10}$	ADDRESSES
CE/PGM	CHIP ENABLE/PROGRAM
$\overline{OE}$	OUTPUT ENABLE
$O_0 - O_7$	OUTPUTS

MODE SELECTION

MODE \ PINS	CE/PGM (18)	OE (20)	VPP (21)	VCC (24)	OUTPUTS (9-11, 13-17)
Read	V_{IL}	V_{IL}	+5	+5	D_{OUT}
Standby	V_{IH}	Don't Care	+5	+5	High Z
Program	Pulsed V_{IL} to V_{IH}	V_{IH}	+25	+5	D_{IN}
Program Verify	V_{IL}	V_{IL}	+25	+5	D_{OUT}
Program Inhibit	V_{IL}	V_{IH}	+25	+5	High Z

BLOCK DIAGRAM



intel®

PRELIMINARY

2732
32K (4K x 8) UV ERASABLE PROM

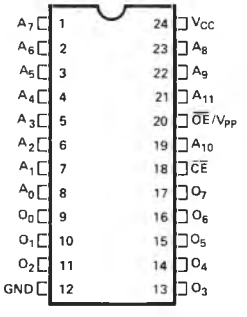
- **Fast Access Time:**
  - **— 450 ns Max. 2732**
  - **— 550 ns Max. 2732-6**
- **Single +5V ± 5% Power Supply**
- **Output Enable for MCS-85™ and MCS-86™ Compatibility**
- **Low Power Dissipation:**
  150mA Max. Active Current
  30mA Max. Standby Current
- **Pin Compatible to Intel® 2716 EPROM**
- **Completely Static**
- **Simple Programming Requirements**
  - **— Single Location Programming**
  - **— Programs with One 50ms Pulse**
- **Three-State Output for Direct Bus Interface**

The Intel® 2732 is a 32,768-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The 2732 operates from a single 5-volt power supply, has a standby mode, and features an output enable control. The total programming time for all bits is three and a half minutes. All these features make designing with the 2732 in microcomputer systems faster, easier, and more economical.

An important 2732 feature is the separate output control, Output Enable ($\overline{OE}$), from the Chip Enable control ($\overline{CE}$). The $\overline{OE}$ control eliminates bus contention in multiple bus microprocessor systems. Intel's Application Note AP-30 describes the microprocessor system implementation of the $\overline{OE}$ and $\overline{CE}$ controls on Intel's 2716 and 2732 EPROMs. AP-30 is available from Intel's Literature Department.

The 2732 has a standby mode which reduces the power dissipation without increasing access time. The maximum active current is 150mA, while the maximum standby current is only 30mA, an 80% savings. The standby mode is achieved by applying a TTL-high signal to the $\overline{CE}$ input.

PIN CONFIGURATION



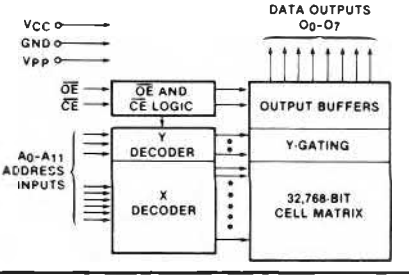
PIN NAMES

A_0-A_{11}	ADDRESSES
$\overline{CE}$	CHIP ENABLE
$\overline{OE}$	OUTPUT ENABLE
O_0-O_7	OUTPUTS

MODE SELECTION

PINS / MODE	$\overline{CE}$ (18)	$\overline{OE}/V_{PP}$ (20)	V_{CC} (24)	OUTPUTS (9-11,13-17)
Read	V_{IL}	V_{IL}	+5	D_{OUT}
Standby	V_{IH}	Don't Care	+5	High Z
Program	V_{IL}	V_{PP}	+5	D_{IN}
Program Verify	V_{IL}	V_{IL}	+5	D_{OUT}
Program Inhibit	V_{IH}	V_{PP}	+5	High Z

BLOCK DIAGRAM



2758
8K (1K × 8) UV ERASABLE LOW POWER PROM

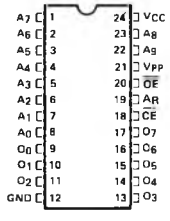
- **Single +5V Power Supply**
- **Simple Programming Requirements**
  - **Single Location Programming**
  - **Programs with One 50 ms Pulse**
- **Low Power Dissipation**
  525 mW Max. Active Power
  132 mW Max. Standby Power
- **Fast Access Time: 450 ns Max. in Active and Standby Power Modes**
- **Inputs and Outputs TTL Compatible during Read and Program**
- **Completely Static**
- **Three-State Outputs for OR-Ties**

The Intel® 2758 is a 8192-bit ultraviolet erasable and electrically programmable read-only memory (EPROM). The 2758 operates from a single 5-volt power supply, has a static standby mode, and features fast single address location programming. It makes designing with EPROMs faster, easier and more economical. The total programming time for all 8192 bits is 50 seconds.

The 2758 has a static standby mode which reduces the power dissipation without increasing access time. The maximum active power dissipation is 525 mW, while the maximum standby power dissipation is only 132 mW, a 75% savings. Power-down is achieved by applying a TTL-high signal to the $\overline{CE}$ input.

A 2758 system may be designed for total upwards compatibility with Intel's 16K 2716 EPROM (see Applications Note 30). The 2758 maintains the simplest and fastest method yet devised for programming EPROMs — single pulse TTL-level programming. There is no need for high voltage pulsing because all programming controls are handled by TTL signals. Program any location at any time — either individually, sequentially, or at random, with the single address location programming.

PIN CONFIGURATION



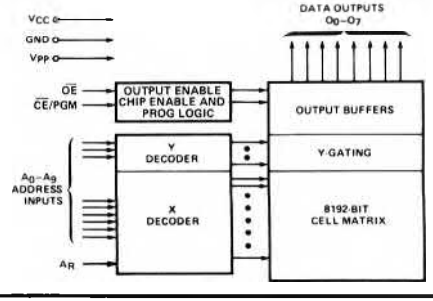
MODE SELECTION

PINS / MODE	$\overline{CE}$/PGM (18)	A_R (19)	$\overline{OE}$ (20)	V_{PP} (21)	V_{CC} (24)	OUTPUTS (9-11, 13-17)
Read	V_{IL}	V_{IL}	V_{IL}	+5	+5	D_{OUT}
Standby	V_{IH}	V_{IL}	Don't Care	+5	+5	High Z
Program	Pulsed V_{IL} to V_{IH}	V_{IL}	V_{IH}	+25	+5	D_{IN}
Program Verify	V_{IL}	V_{IL}	V_{IL}	+25	+5	D_{OUT}
Program Inhibit	V_{IL}	V_{IL}	V_{IH}	+25	+5	High Z

PIN NAMES

A_0–A_9	ADDRESSES
$\overline{CE}$/PGM	CHIP ENABLE/PROGRAM
$\overline{OE}$	OUTPUT ENABLE
O_0–O_7	OUTPUTS
A_R	SELECT REFERENCE INPUT LEVEL

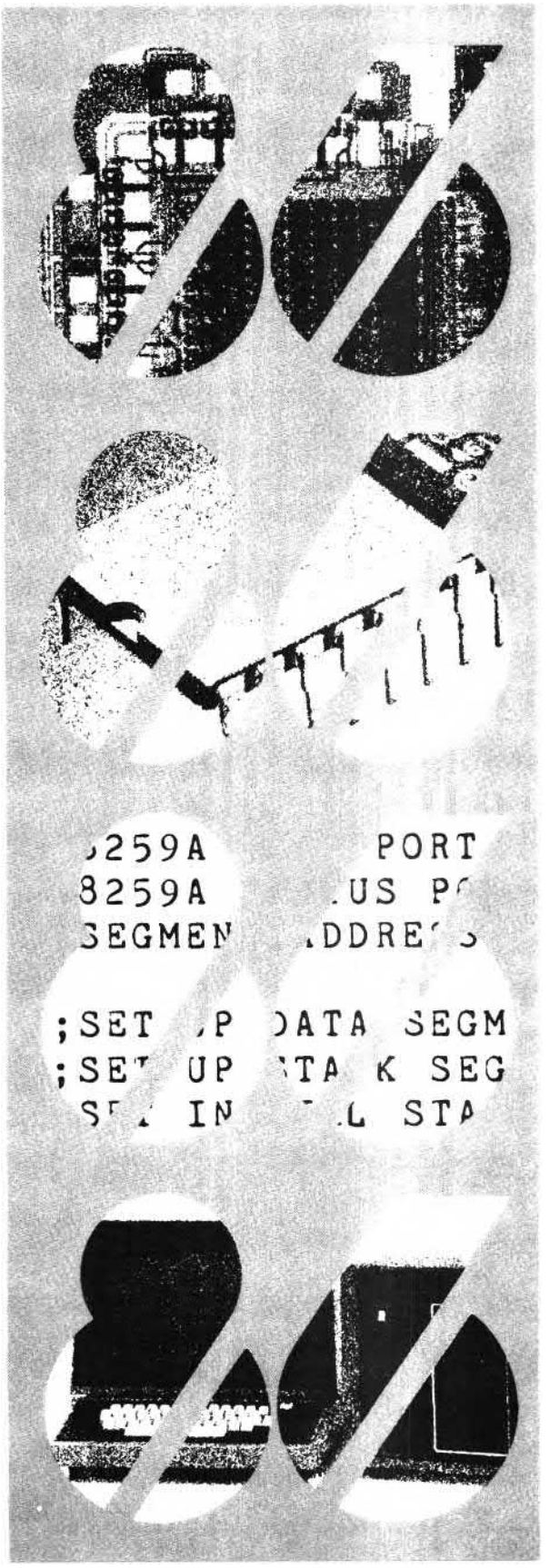
BLOCK DIAGRAM



Appendix B
Device Specifications

- 8085 Family
- 8085 Peripherals*
- Standard Peripherals**
- RAM Memories***
- EPROM Memories***
- **Development Tools**

*For complete specifications refer to the Intel MCS-85 User's Manual.
**For complete specifications refer to the Intel Peripheral Design Handbook.
***For complete specifications refer to the 1979 Intel Component Data Catalog.



MODEL 230
INTELLEC SERIES II
MICROCOMPUTER DEVELOPMENT SYSTEM

Complete microcomputer development center for Intel MCS-86, MCS-80, MCS-85 and MCS-48 microprocessor families

LSI electronics board with CPU, RAM, ROM, I/O, and interrupt circuitry

64K bytes RAM memory

Self-test diagnostic capability

Eight-level nested, maskable priority interrupt system

Built-in interfaces for high speed paper tape reader/punch, printer, and universal PROM programmer

Integral CRT with detachable upper/lower case typewriter-style full ASCII keyboard

Powerful ISIS-II Diskette Operating System software with relocating macroassembler, linker, and locater

1 million bytes (expandable to 2.5M bytes) of diskette storage

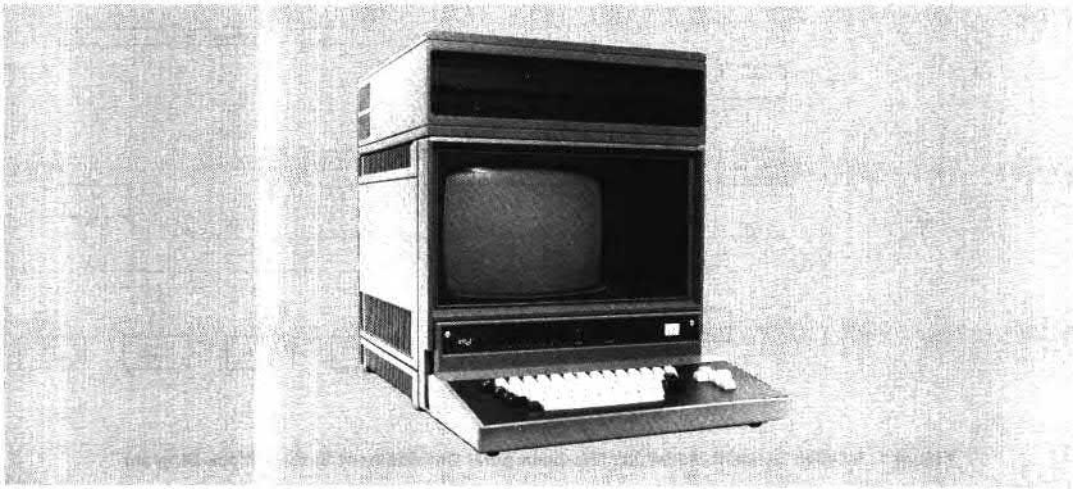
Supports PL/M and FORTRAN high level languages

Standard MULTIBUS with multiprocessor and DMA capability

Compatible with standard Intellec/iSBC expansion modules

Software compatible with previous Intellec systems

The Model 230 Intellec Series II Microcomputer Development System is a complete center for the development of microcomputer-based products. It includes a CPU, 64K bytes of RAM, 4K bytes of ROM memory, a 2000-character CRT, a detachable full ASCII keyboard, and dual double density diskette drives providing over 1 million bytes of on-line data storage. Powerful ISIS-II Diskette Operating System software allows the Model 230 to be used quickly and efficiently for assembling and/or compiling and debugging programs for Intel's MCS-86, MCS-80, MCS-85, or MCS-48 microprocessor families without the need for handling paper tape. ISIS-II performs all file handling operations, leaving the user free to concentrate on the details of his own application. When used in conjunction with an optional in-circuit emulator (ICE) module, the Model 230 provides all the hardware and software development tools necessary for the rapid development of a microcomputer-based product.



FUNCTIONAL DESCRIPTION

Hardware Components

The Intellec Series II Model 230 is a packaged, highly integrated microcomputer development system consisting of a CRT chassis with a 6-slot cardcage, power supply, fans, cables, and five printed circuit cards. A separate, full ASCII keyboard is connected with a cable. A second chassis contains two floppy disk drives capable of double-density operation along with a separate power supply, fans, and cables for connection to the main chassis. A block diagram of the Model 230 is shown in Figure 1.

CPU Cards — The master CPU card contains its own microprocessor, memory, I/O, interrupt and bus interface circuitry fashioned from Intel's high technology LSI components. Known as the integrated processor board (IPB), it occupies the first slot in the cardcage. A second slave CPU card is responsible for all remaining I/O control including the CRT and keyboard interface. This card, mounted on the rear panel, also contains its own microprocessor, RAM and ROM memory, and I/O interface logic, thus, in effect, creating a dual processor environment. Known as the I/O controller (IOC), the slave CPU card communicates with the IPB over an 8-bit bidirectional data bus.

Memory and Control Cards — In addition, 32K bytes of RAM (bringing the total to 64K bytes) is located on a separate card in the main cardcage. Fabricated from Intel's 16K RAMs, the board also contains all necessary address decoding and refresh logic. Two additional boards in the cardcage are used to control the two double-density floppy disk drives.

Expansion — Two remaining slots in the cardcage are available for system expansion. Additional expansion of 4 slots can be achieved through the addition of an Intellec Series II expansion chassis.

System Components

The heart of the IPB is an Intel NMOS 8-bit microprocessor, the 8080A-2, running at 2.6 MHz. 32K bytes of RAM memory are provided on the board using Intel 16K RAMs. 4K of ROM is provided, preprogrammed with system bootstrap "self-test" diagnostics and the Intellec Series II System Monitor. The eight-level vectored priority interrupt system allows interrupts to be individually masked. Using Intel's versatile 8259A interrupt controller, the interrupt system may be user programmed to respond to individual needs.

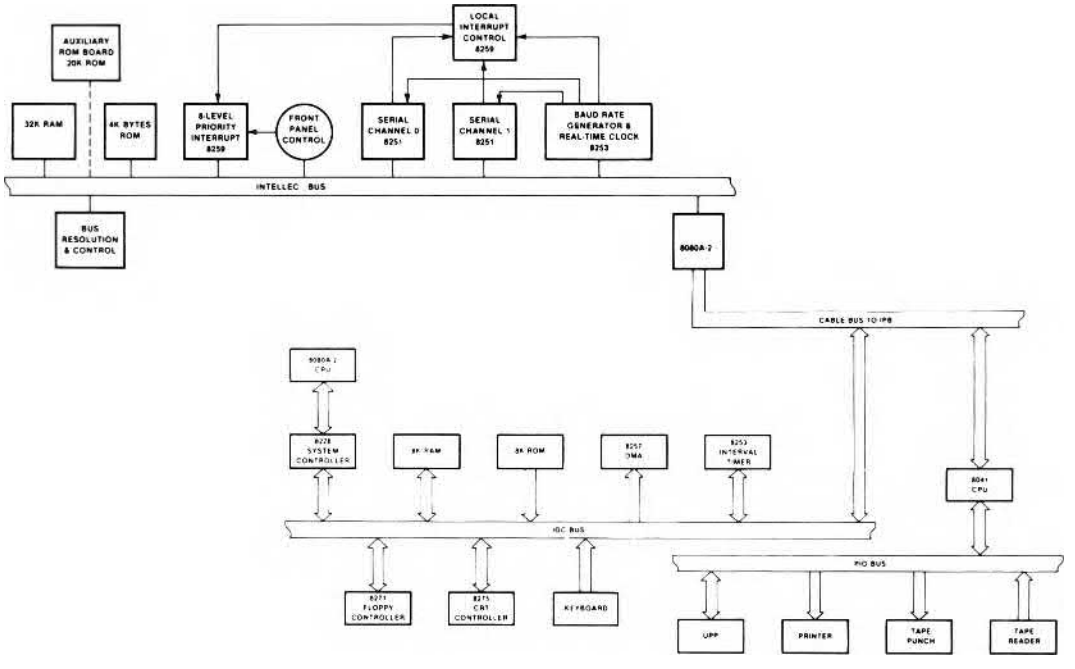


Figure 1. Intellec Series II Model 230 Microcomputer Development System Block Diagram

Input/Output

IPB Serial Channels — The I/O subsystem in the Model 230 consists of two parts: the IOC card and two serial channels on the IPB itself. Each serial channel is RS232 compatible and is capable of running asynchronously from 110 to 9600 baud or synchronously from 150 to 56K baud. Both may be connected to a user defined data set or terminal. One channel contains current loop adapters. Both channels are implemented using Intel's 8251A USART. They can be programmatically selected to perform a variety of I/O functions. Baud rate selection is accomplished programmatically through an Intel 8253 interval timer. The 8253 also serves as a real-time clock for the entire system. I/O activity through both serial channels is signaled to the system through a second 8259 interrupt controller, operating in a polled mode nested to the primary 8259.

IOC Interface — The remainder of system I/O activity takes place in the IOC. The IOC provides interface for the CRT, keyboard, and standard Intellec peripherals including printer, high speed paper tape reader/punch, and universal PROM programmer. The IOC contains its own independent microprocessor, also an 8080A-2. The CPU controls all I/O operations as well as supervising communications with the IPB. 8K bytes of ROM contain all I/O control firmware. 8K bytes of RAM are used for CRT screen refresh storage. These do not occupy space in Intellec Series II main memory since the IOC is a totally independent microcomputer subsystem.

Integral CRT

Display — The CRT is a 12-inch raster scan type monitor with a 50/60 Hz vertical scan rate and 15.5 kHz horizontal scan rate. Controls are provided for brightness and contrast adjustments. The interface to the CRT is provided through an Intel 8275 single chip programmable CRT controller. The master processor on the IPB transfers a character for display to the IOC, where it is stored in RAM. The CRT controller reads a line at a time into its line buffer through an Intel 8257 DMA controller and then feeds one character at a time to the character generator to produce the video signal. Timing for the CRT control is provided by an Intel 8253 interval timer. The screen display is formatted as 25 rows of 80 characters. The full set of ASCII characters are displayed, including lower case alphas.

Keyboard — The keyboard interfaces directly to the IOC processor via an 8-bit data bus. The keyboard contains an Intel UPI-41 Universal Peripheral Interface, which scans the keyboard, encodes the characters, and buffers the characters to provide N-key rollover. The keyboard itself is a high quality typewriter style keyboard containing the full ASCII character set. An upper/lower case switch allows the system to be used for document preparation. Cursor control keys are also provided.

Peripheral Interface

A UPI-41 Universal Peripheral Interface on the IOC board performs similar functions to the UPI-41 on the PIO board in the Model 210. It provides interface for other standard Intellec peripherals including a printer, high speed paper tape reader, high speed paper tape punch, and universal PROM programmer. Communication between the IPB and IOC is maintained over a separate 8-bit bidirectional data bus. Connectors for the four devices named above, as well as the two serial channels, are mounted directly on the IOC itself.

Control

User control is maintained through a front panel, consisting of a power switch and indicator, reset/boot switch, run/halt light, and eight interrupt switches and indicators. The front panel circuit board is attached directly to the IPB, allowing the eight interrupt switches to connect to the primary 8259A, as well as to the Intellec Series II bus.

Diskette System

The Intellec Series II double density diskette system provides direct access bulk storage, intelligent controller, and two diskette drives. Each drive provides ½ million bytes of storage with a data transfer rate of 500,000 bits/second. The controller is implemented with Intel's powerful Series 3000 Bipolar Microcomputer Set. The controller provides an interface to the Intellec Series II system bus, as well as supporting up to four diskette drives. The diskette system records all data in soft sector format. The diskette system is capable of performing seven different operations: recalibrate, seek, format track, write data, write deleted data, read data, and verify CRC.

Diskette Controller Boards — The diskette controller consists of two boards, the channel board and the interface board. These two PC boards reside in the Intellec Series II system chassis and constitute the diskette controller. The channel board receives, decodes and responds to channel commands from the 8080A-2 CPU in the Model 230. The interface board provides the diskette controller with a means of communication with the diskette drives and with the Intellec system bus. The interface board validates data during reads using a cyclic redundancy check (CRC) polynomial and generates CRC data during write operations. When the diskette controller requires access to Intellec system memory, the interface board requests and maintains DMA master control of the system bus, and generates the appropriate memory command. The interface board also acknowledges I/O commands as required by the Intellec bus. In addition to supporting a second set of double density drives, the diskette controller may co-reside with the Intel single density controller to allow up to 2.5 million bytes of on-line storage.

MULTIBUS Capability

All Intellec Series II models implement the industry standard MULTIBUS. MULTIBUS enables several bus masters, such as CPU and DMA devices, to share the bus and memory by operating at different priority levels. Resolution of bus exchanges is synchronized by a bus clock signal derived independently from processor clocks. Read/write transfers may take place at rates up to 5 MHz. The bus structure is suitable for use with any Intel microcomputer family.

SPECIFICATIONS

Host Processor (IPB)

RAM — 64K (system monitor occupies 62K through 64K)
ROM — 4K (2K in monitor, 2K in boot/diagnostic)

Diskette System Capacity (Basic Two Drives)

Unformatted
Per Disk: 6.2 megabits
Per Track: 82.0 kilobits

Formatted
Per Disk: 4.1 megabits
Per Track: 53.2 kilobits

Diskette Performance

Diskette System Transfer Rate — 500 kilobits/sec

Diskette System Access Time
Track-to-Track: 10 ms
Head Settling Time: 10 ms

Average Random Positioning Time — 260 ms

Rotational Speed — 360 rpm
Average Rotational Latency — 83 ms
Recording Mode — M^2FM

Physical Characteristics

Width — 17.37 in. (44.12 cm)
Height — 15.81 in. (40.16 cm)
Depth — 19.13 in. (48.59 cm)
Weight — 73 lb (33 kg)

Keyboard
Width — 17.37 in. (44.12 cm)
Height — 3.0 in. (7.62 cm)
Depth — 9.0 in. (22.86 cm)
Weight — 6 lb (3 kg)

Dual Drive Chassis
Width — 16.88 in. (42.88 cm)
Height — 12.08 in. (30.68 cm)
Depth — 19.0 in. (48.26 cm)
Weight — 64 lb (29 kg)

Electrical Characteristics

DC Power Supply

Volts Supplied	Amps Supplied	Typical System Requirements
+ 5 ± 5%	30	14.25
+ 12 ± 5%	2.5	0.2
− 12 ± 5%	0.3	0.05
− 10 ± 5%	1.5	15
* + 15 ± 5%	1.5	1.3
* + 24 ± 5%	1.7	

*Not available on bus.

AC Requirements — 50/60 Hz, 115/230V AC

Environmental Characteristics

Operating Temperature — 0° to 35°C (95°F)

Equipment Supplied

Model 230 chassis
Integrated processor board (IPB)
I/O controller board (IOC)
32K RAM board
CRT and keyboard
Double density floppy disk controller (2 boards)
Dual drive floppy disk chassis and cables
2 floppy disk drives (512K byte capacity each)
ROM-resident system monitor
ISIS-II system diskette with MCS-80/MCS-85 macroassembler

Reference Manuals

9800558 — A Guide to Microcomputer Development Systems (SUPPLIED)

9800550 — Intellec Series II Installation and Service Guide (SUPPLIED)

9800306 — ISIS-II System User's Guide (SUPPLIED)

9800556 — Intellec Series II Hardware Reference Manual (SUPPLIED)

9800301 — 8080/8085 Assembly Language Programming Manual (SUPPLIED)

9800292 — ISIS-II 8080/8085 Assembler Operator's Manual (SUPPLIED)

9800605 — Intellec Series II Systems Monitor Source Listing (SUPPLIED)

9800554 — Intellec Series II Schematic Drawings (SUPPLIED)

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

ORDERING INFORMATION

Part Number	Description
MDS-230	Intellec Series II Model 230 microcomputer development system (110V/60 Hz)
MDS-231	Intellec Series II Model 230 microcomputer development system (220V/50 Hz)

intel®

8086/8088 SOFTWARE DEVELOPMENT PACKAGE

PL/M-86 high level programming language

ASM86 macro assembler for 8086/8088 assembly language programming

LINK86 and LOC86 linkage and relocation utilities

CONV86 converter for conversion of 8080/8085 assembly language source code to 8086/8088 assembly language-source code

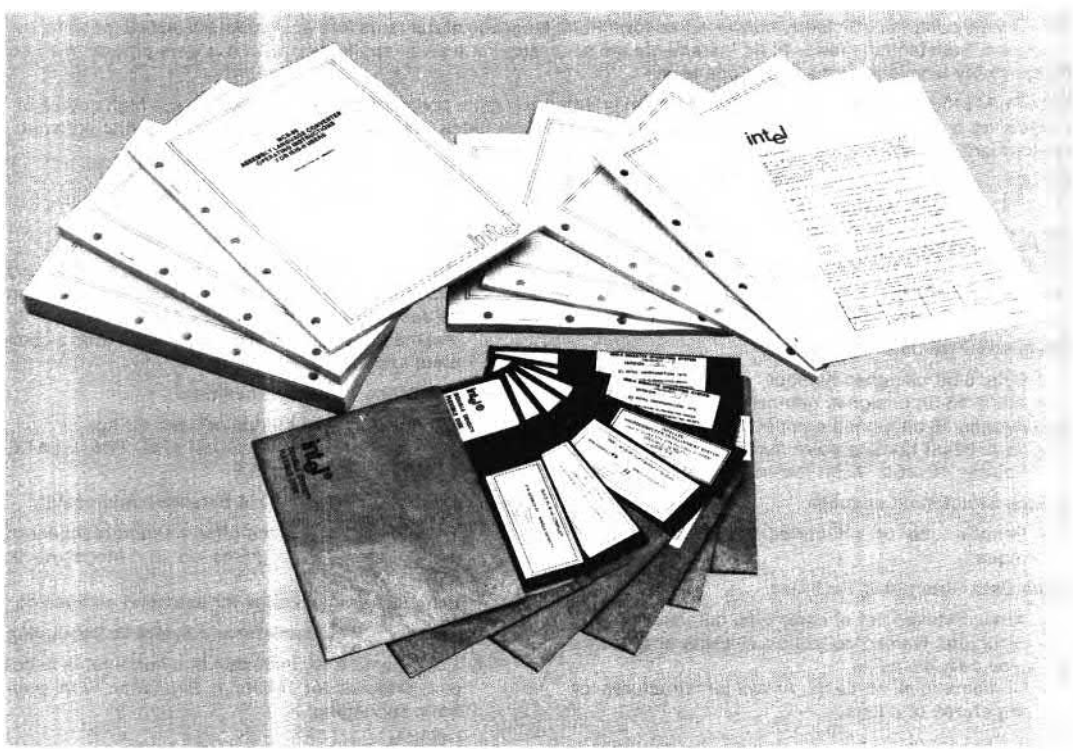
OH86 object-to-hexadecimal converter

LIB86 library manager

The 8086/8088 software development package provides a set of software development tools for the 8086 and the 8088 microprocessors and iSBC 86/12 single board computer. The package operates under the ISIS-II operating system on Intellec Microcomputer Development Systems—Model 800 or Series II—thus minimizing requirements for additional hardware or training for Intel Microcomputer Development System users.

The package permits 8080/8085 users to efficiently convert existing programs into 8086/8088 object code from either 8080/8085 assembly language source code or PL/M-80 source code.

For the new Intel Microcomputer Development System user, the package operating on an Intellec Model 230 Microcomputer Development System provides total 8086/8088 software development capability.



PL/M-86 HIGH LEVEL PROGRAMMING LANGUAGE

Sophisticated new compiler design allows user to achieve maximum benefits of 8086/8088 capabilities

Language is upward compatible from PL/M-80, assuring MCS-80/85 design portability

Supports 16-bit signed integer and 32-bit floating point arithmetic

Produces relocatable and linkable object code

Supports full extended addressing features of the 8086 and the 8088 microprocessors

Code optimization assures efficient code generation and minimum application memory utilization

Like its counterpart for MCS-80/85 program development, PL/M-86 is an advanced structured high level programming language. PL/M-86 is a new compiler created specifically for performing software development for the Intel 8086 and 8088 Microprocessors.

PL/M-86 has significant new capabilities over PL/M-80 that take advantage of the new facilities provided by the 8086 and the 8088 microprocessors, yet the PL/M-86 language remains upward compatible from PL/M-80.

With the exception of interrupts, hardware flags, and time-critical code sequences, PL/M-80 programs may be recompiled under PLM-86 with little or no conversion required. PL/M-86, like PL/M-80, is easy to learn, facilitates rapid program development, and reduces program maintenance costs.

PL/M is a powerful, structured high level algorithmic language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the system implementation without concern for burdensome details of assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M-86 compiler efficiently converts free-form PL/M language statements into equivalent 8086/8088 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

Since PL/M programs are implementation problem oriented and more compact, use of PL/M results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

FEATURES

Major features of the Intel PL/M-86 compiler and programming language include:

- **Supports Five Data Types**
  - — Byte: 8-bit unsigned number
  - — Word: 16-bit unsigned number
  - — Integer: 16-bit signed number
  - — Real: 32-bit floating point number
  - — Pointer: 16-bit or 32-bit memory address indicator
- **Block Structured Language**
  - — Permits use of structured programming techniques
- **Two Data Structuring Facilities**
  - — Array: Indexed list of same type data elements
  - — Structure: Named collection of same or different type data elements
  - — Combinations of Each: Arrays of structures or structures of arrays
- **Relocatable and Linkable Object Code**
  - — Permits PL/M-86 programs to be developed and debugged in small modules. These modules can be easily linked with other PL/M-86 or ASM86 object modules and/or library routines to form a complete application system.
- **Built-In String Handling Facilities**
  - — Operates on byte strings or word strings
  - — Six Functions: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET
- **Automatic Support for 8086 Extended Addressing**
  - — Three compiler options offer a separate model of computation for programs up to 1-Megabyte in size
  - — Language transparency for extended addressing
- **Support for ICE-86 Emulator and Symbolic Debugging**
  - — Debug option for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging

- **Numerous Compiler Options**
  - A host of 26 compiler options including:
    - Conditional compilation
    - Included file or copy facility
    - Two levels of optimization
    - Intra-module and inter-module cross reference
    - Arbitrary placement of compiler and user files on any available combination of disk drives
- **Reentrant and Interrupt Procedures**
  - May be specified as user options

BENEFITS

PL/M-86 is designed to be an efficient, cost-effective solution to the special requirements of 8086/8088 Microcomputer Software Development, as illustrated by the following benefits of PL/M-86 use:

- **Reduced Learning Effort** — PL/M-86 is easy to learn and to use, even for the novice programmer.
- **Earlier Project Completion** — Critical projects are completed much earlier than otherwise possible because PL/M-86, a structured high-level language, increases programmer productivity.
- **Lower Development Cost** — Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.
- **Increased Reliability** — PL/M-86 is designed to aid in the development of reliable software (PL/M-86 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.
- **Easier Enhancements and Maintenance** — Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.
- **Simpler Project Development** — The Intellec Development Systems offer a cost-effective hardware base for the development of 8086 and 8088 designs. PL/M-86 and other elements of ISIS-II and the 8086/8088 Software Development Package are all that is needed for development of software for the 8086 and the 8088 microcomputers and iSBC 86/12 single board computer. This further reduces development time and costs because expensive (and remote) time sharing of large computers is not required. Present users of Intel Intellec Development Systems can begin to develop 8086 and 8088 designs without expensive hardware reinvestment or costly retraining.

SAMPLE PROGRAM

```
STATISTICS: DO;

/*The procedure in this module computes the mean and
variance of an array of data, X, of length N + 1, according
to the method of Kahan and Parlett (University of Cali-
fornia, Berkeley, Memo no. UCB/ERL M77/21.*/

STAT:  PROCEDURE(X$PTR,N,MEAN$PTR,
       VARIANCE$PTR) PUBLIC;

DECLARE
       (X$PTR,MEAN$PTR,VARIANCE$PTR)
       POINTER,X BASED X$PTR (1) REAL,
       N INTEGER,
       MEAN BASED MEAN$PTR REAL,
       VARIANCE BASED VARIANCE$PTR REAL,
       (M,Q,DIFF) REAL,
       I INTEGER;

M = X(0);
M = 0.0;

DO  I = 1 TO N;
    DIFF = X(I) - M;
    M = M + DIFF/FLOAT(I + 1);
    Q = Q + DIFF*DIFF*FLOAT(I)/FLOAT(I + 1);
    END;

    MEAN = M;
    VARIANCE = Q/FLOAT(N);

    END STAT;

    END STATISTICS;
```

ASM86 MACRO ASSEMBLER

Powerful and flexible text macro facility with three macro listing options to aid debugging

Highly mnemonic and compact language, most mnemonics represent several distinct machine instructions

"Strongly typed" assembler helps detect errors at assembly time

High-level data structuring facilities such as "STRUCTUREs" and "RECORDs"

Over 120 detailed and fully documented error messages

Produces relocatable and linkable object code

ASM86 is the "high-level" macro assembler for the 8086/8088 assembly language. ASM86 translates symbolic 8086/8088 assembly language mnemonics into 8086/8088 machine code.

ASM86 should be used where maximum code efficiency and hardware control is needed. The 8086/8088 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 8086/8088 machine instructions. ASM86 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

ASM86 offers many features normally found only in high-level languages. The 8086/8088 assembly language is strongly typed. The assembler performs extensive checks on the usage of variables and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

FEATURES

Major features of the Intel 8086/8088 assembler and assembly language include:

- **Powerful and Flexible Text Macro Facility**
  - — Macro calls may appear anywhere
  - — Allows user to define the syntax of each macro
  - — Built-in functions
    - conditional assembly (IF-THEN-ELSE, WHILE)
    - repetition (REPEAT)
    - string processing functions (MATCH)
    - support of assembly time I/O to console (IN, OUT)
  - — Three Macro Listing Options include a GEN mode which provides a complete trace of all macro calls and expansions
- **High-Level Data Structuring Capability**
  - — STRUCTURES: Defined to be a template and then used to allocate storage. The familiar dot notation may be used to form instruction addresses with structure fields.
  - — ARRAYS: Indexed list of same type data elements.
  - — RECORDS: Allows bit-templates to be defined and used as instruction operands and/or to allocate storage.
- **Fully Supports 8086/8088 Addressing Modes**
  - — Provides for complex address expressions involving base and indexing registers and (structure) field offsets.
  - — Powerful EQU facility allows complicated expressions to be named and the name can be used as a synonym for the expression throughout the module.
- **Powerful STRING MANIPULATION INSTRUCTIONS**
  - — Permit direct transfers to or from memory or the accumulator.
  - — Can be prefixed with a repeat operator for repetitive execution with a count-down and a condition test.
- **Over 120 Detailed Error Messages**
  - — Appear both in regular list file and error print file.
  - — User documentation fully explains the occurrence of each error and suggests a method to correct it.

- **Generates Relocatable and Linkable Object Code—Fully Compatible with LINK86, LOC86 and LIB86**
  - Permits ASM86 programs to be developed and debugged in small modules. These modules can be easily linked with other ASM86 or PL/M-86 object modules and/or library routines to form a complete application system.

- **Support for ICE-86 Emulation and Symbolic Debugging**
  - Debug options for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging.

BENEFITS

The 8086/8088 macro assembler allows the extensive capabilities of the 8086/8088 to be fully exploited. In any application, time and space critical routines can be effectively written in ASM86. The 8086/8088 assembler outputs relocatable and linkable object modules. These object modules may be easily combined with object modules written in PL/M-86—Intel's structured, high-level programming language. ASM86 compliments PLM-86 as the programmer may choose to write each module in the language most appropriate to the task and then combine the modules into the complete applications program using the 8086/8088 relocation and linkage utilities.

CONV86

MCS-80/85 to MCS-86 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

Translates 8080/8085 Assembly Language Source Code to 8086/8088 Assembly Language Source Code

Provides a fast and accurate means to convert 8080/8085 programs to the 8086 and the 8088, facilitating program portability

Automatically generates proper ASM-86 directives to set up a "virtual 8080" environment that is compatible with PLM-86

In support of Intel's commitment to software portability, CONV86 is offered as a tool to move 8080/8085 programs to the 8086 and the 8088. A comprehensive manual, "MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users" (9800642), covers the entire conversion process. Detailed methodology of the conversion process is fully described therein.

CONV86 will accept as input an error-free 8080/8085 assembly-language source file and optional controls, and produce as output, optional PRINT and OUTPUT files.

The PRINT file is a formatted copy of the 8080/8085 source and the 8086/8088 source file with embedded caution messages.

The OUTPUT file is an 8086/8088 source file.

CONV86 issues a caution message when it detects a potential problem in the converted 8086/8088 code.

A transliteration of the 8080/8085 programs occurs, with each 8080/8085 construct mapped to its exact 8086/8088 counterpart:

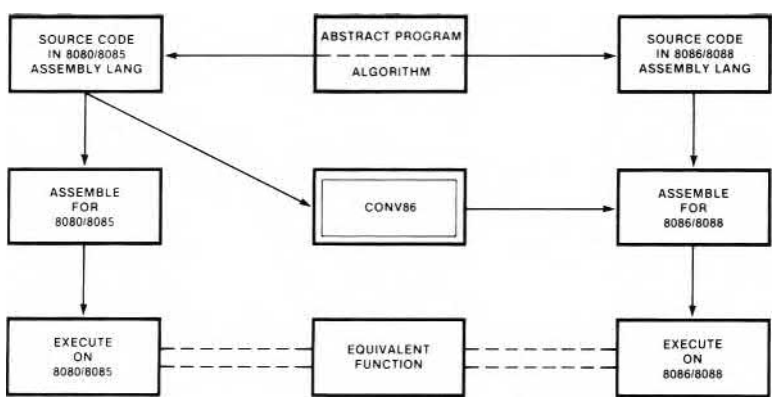
—Registers
—Condition flags
—Instructions
—Operands
—Assembler directives
—Assembler control lines
—Macros

Because CONV86 is a transliteration process, there is the possibility of as much as a 15%-20% code expansion over the 8080/8085 code. For compactness and efficiency it is recommended that critical portions of programs be re-coded in 8086/8088 assembly language.

Also, as a consequence of the transliteration, some manual editing may be required for converting instruction sequences dependent on:

-instruction length, timing, or encoding
-interrupt processing } mechanical editing procedures
-PL/M parameter passing conventions } for these are suggested in the converter manual.

The accompanying diagram illustrates the flow of the conversion process. Initially, the abstract program may be represented in 8080/8085 or 8086/8088 assembly language to execute on that respective target machine. The conversion process is porting a source destined for the 8080/8085 to the 8086 or the 8088 via CONV86.



PORTING 8080/8085 SOURCE CODE TO THE 8086/8088

LINK86

Automatic combination of separately compiled or assembled 8086/8088 programs into a relocatable module

Automatic selection of required modules from specified libraries to satisfy symbolic references

Extensive debug symbol manipulation, allowing line numbers, local symbols, and public symbols to be purged and listed selectively

Automatic generation of a summary map giving results of the LINK86 process

Abbreviated control syntax

Relocatable modules may be merged into a single module suitable for inclusion in a library

Supports "incremental" linking

Supports type checking of public and external symbols

LINK86 combines object modules specified in the LINK86 input list into a single output module. LINK86 combines segments from the input modules according to the order in which the modules are listed.

Support for incremental linking is provided since an output module produced by LINK86 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

LINK86 supports type checking of public and external symbols reporting an error if their types are not consistent.

LINK86 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the LINK86 process and to control the content of the output module.

LINK86 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using LOC86, and enter final testing with much of the work accomplished.

LOC86

Automatic and independent relocation of segments. Segments may be relocated to best match users memory configuration

Extensive debug symbol manipulation, allowing line numbers, local symbols, and public symbols to be purged and listed selectively

Automatic generation of a summary map giving starting address, segment addresses and lengths, and debug symbols and their addresses

Extensive capability to manipulate the order and placement of segments in 8086/8088 memory

Abbreviated control syntax

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

LOC86 converts relative addresses in an input module to absolute addresses. LOC86 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

LOC86 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the LOC86 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program on the Intellec development system and then simply relocate the object code to suit your application.

OH86

Converts an 8086/8088 absolute object module to symbolic hexadecimal format

Converts an absolute module to a more readable format that can be displayed on a CRT or printed for debugging

Facilitates preparing a file for later loading by a symbolic hexadecimal loader, such as the iSBC Monitor or Universal PROM Mapper

The OH86 command converts an 8086/8088 absolute object module to the hexadecimal format. This conversion may be necessary to format a module for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or Universal Prom Mapper. The conversion may also be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute format; the output from LOC86 is in absolute format.

LIB86

LIB86 is a library manager program which allows you to:

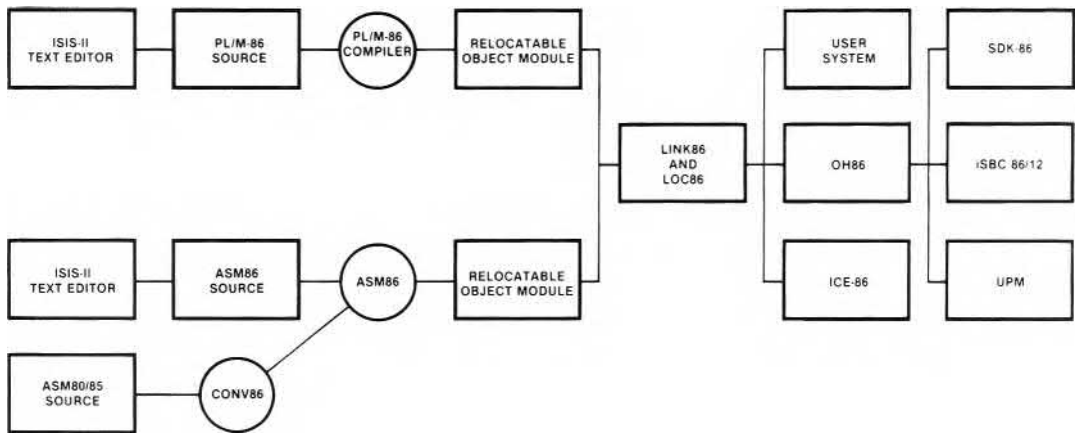
- **Create specially formatted files to contain libraries of object modules**
- **Maintain these libraries by adding or deleting modules**
- **Print a listing of the modules and public symbols in a library file**

Libraries can be used as input to LINK86 which will automatically link modules from the library that satisfy external references in the modules being linked

Abbreviated control syntax

Libraries aid in the job of building programs. The library manager program, LIB86, creates and maintains files containing object modules. The operation of LIB86 is controlled by commands to indicate which operation LIB86 is to perform. The commands are:

CREATE — creates an empty library file
ADD — adds object modules to a library file
DELETE — deletes modules from a library file
LIST — lists the module directory of library files
EXIT — terminates the LIB86 program and returns control to ISIS-II


ISIS-II
TEXT EDITOR
PL/M-86
SOURCE
PL/M-86
COMPILER
RELOCATABLE
OBJECT MODULE
ISIS-II
TEXT EDITOR
ASM86
SOURCE
ASM86
RELOCATABLE
OBJECT MODULE
ASM80/85
SOURCE
CONV86
LINK86
AND
LOC86
USER
SYSTEM
OH86
ICE-86
SDK-86
iSBC 86/12
UPM

SPECIFICATIONS

Operating Environment

Required Hardware

Intellec Microcomputer Development System

— MDS-800, MDS-888
— Series II MDS-220 or MDS-230
64K Bytes of RAM Memory

Dual Diskette Drives

— Single or Double* Density

System Console

— CRT or Hardcopy Interactive Device

Optional Hardware

Universal PROM Programmer
Line Printer*
ICE-86™*

Required Software

ISIS-II Diskette Operating System

— Single or Double* Density

Documentation Package

PL/M-86 Programming Manual (9800466)
ISIS-II PL/M-86 Compiler Operator's Manual (9800478)
MCS-86 User's Manual (9800722)
MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users (9800639)
MCS-86 Macro Assembly Language Reference Manual (9800640)
MCS-86 Macro Assembler Operating Instructions for ISIS-II Users (9800641)
MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users (9800642)
Universal PROM Programmer User's Manual (9800819A)

Flexible Diskettes

— Single and Double* Density

*Recommended

ORDERING INFORMATION

Part Number	Description
MDS-311	8086/8088 Software Development Package

Also available in the following development support packages:

Part Number	Description
SP86A-KIT	SP86A Support Package (for Intellec Model 800) Includes ICE-86 In-Circuit Emulator (MDS-86-ICE) and 8086/8088 Software Development Package (MDS-311)
SP86B-KIT	SP86B Support Package (for Series II) Includes ICE-86 In-Circuit Emulator (MDS-86-ICE), 8086/8088 Software Development Package (MDS-311), and Series II Expansion Chassis (MDS-201)

8089 ASSEMBLER SUPPORT PACKAGE

8089 I/O processor program generation on the Intellec Microcomputer Development System.

Relocatable object module compatible with the 8086 and 8088 Microprocessors.

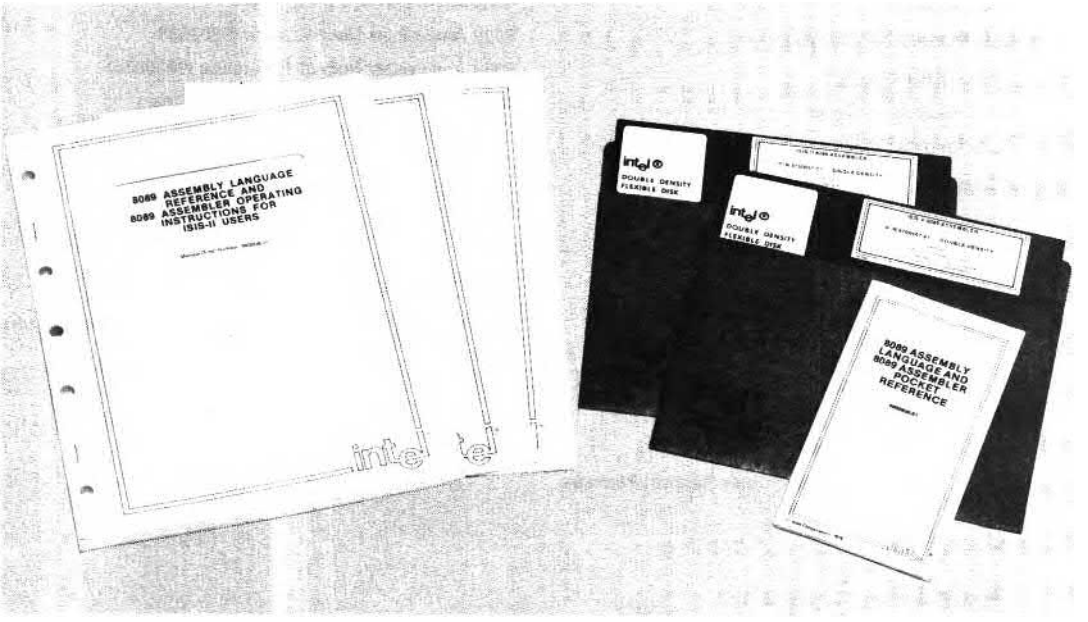
Supports 8089-based addressing modes with a structure facility that enables easy access to based data.

Fully detailed set of error messages.

Includes software development utilities to facilitate 8089 design.

- **—LINK86: Combines 8086 or 8088 object modules with 8089 object modules and resolves external references.**
- **—LOC86: Assigns absolute memory addresses to 8089 object modules.**
- **—OH86: Converts 8086/8088/8089 object code to symbolic hexadecimal format.**
- **—UPM86: A PROM programming aid which has been updated to support PROM programming for 8086, 8088 and 8089 applications.**

The 8089 Assembler Support Package extends Intellec microcomputer development system support to the 8089 I/O Processor. The assembler translates 8089 assembly language source instructions into appropriate machine operation codes. The 8089 Assembler Support Package allows the programmer to fully utilize the capabilities of the 8089 I/O Processor.



FUNCTIONAL DESCRIPTION

The 8089 Assembler Support Package contains the 8089 assembler (ASM89) as well as LINK86 and LOC86—relocation and linkage utilities, OH86—8086/8088/8089 object code to hexadecimal converter, and UPM86—PROM programming software updated to program object code in the 8086 formats. ASM89 translates symbolic 8089 assembly language instructions into the appropriate machine operation codes. The ability to refer to program addresses with symbolic names eliminates the errors of hand translation and makes it easier to modify programs when adding or deleting instructions.

ASM89 provides relocatable object module compatibility with the 8086 and 8088 microprocessors. This object module compatibility, along with the 8086/8088 relocation and linkage utilities, facilitates the designing of the 8089 into an 8086 or 8088 system.

ASM89 fully supports the based addressing modes of the 8089. A structure facility in the assembler provides easy access to based data. The structure facility allows the user to define a template that enables accessing of based data symbolically.

A sample assembly listing is shown in table 1.

```
-ASM89 ASSEMBLER                                                    PAGE    1

ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE CONSOL
OBJECT MODULE PLACED IN :F0:CONSOL.OBJ
ASSEMBLER INVOKED BY ASM89 CONSOL.SRC

0000                    1 CONSOLE  SEGMENT
                        2
                        3 ; INITIALIZE 8275 CRT AND 8279 KEYBOARD CONTROLLERS
                        4
                        5 CONTROL        STRUC
                        6   ; 8275 PORTS
0000                    7   PARA75:  DS   1            ; PARAMETER PORT
0001                    8   NULL82:  DS   1
0002                    9   STAT75:  DS   1            ; STATUS/COMMAND PORT
                       10
                       11   ; 8279 PORTS
0003                   12   NULL82:  DS   3
0006                   13   STAT79:  DS   1            ; STATUS/COMMAND PORT
0007                   14 CONTROL       ENDS
                       15
0000   1138 0040        16   MOVI   GA,4000H           ; SET PORT BASE ADDRESS
0004   8A4C 02 00       17   MOVB1 [GA] STAT75.0        ; INITIALIZE 8275
0008   8A4C 00 4F       18   MOVB1 [GA] PARA75.4FH
000C   8A4C 00 03       19   MOVB1 [GA] PARA75.03H
0010   8A4C 00 6A       20   MOVB1 [GA] PARA75.6AH
0014   8A4C 00 19       21   MOVB1 [GA] PARA75.19H
                        22
0018   8A4C 06 00       23   MOVB1 [GA] STAT79.0        ; INITIALIZE 8279
001C   8A4C 06 38       24   MOVB1 [GA] STAT79.38H
                        25
0020                    26 CONSOLE ENDS
                        27
                        28         END
8089 ASSEMBLER                                                      PAGE    2

SYMBOL TABLE
------------

 DEFN VALUE TYPE  NAME
----- ----- ----  ----

    1  0000  SYM  CONSOLE
    5  0000  STR  CONTROL
    8  0001  SYM  NULL81
   12  0003  SYM  NULL82
    7  0000  SYM  PARA75
    9  0002  SYM  STAT75
   13  0006  SYM  STAT79

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Table 1. Sample 8089 Assembly Listing

SPECIFICATIONS

Operating Environment

Required Hardware

Intellec Microcomputer Development System

—MDS-800, MDS-888

—Series II Models 220 or 230

64K Bytes of RAM Memory

Minimum One Diskette Drive

—Single or Double* Density

System Console

—CRT or Hardcopy Interactive Device

Optional Hardware

Universal PROM Programmer*
Line Printer*

Required Software

ISIS-II Diskette Operating System

—Single or Double* Density

Documentation Package

8089 Assembler User's Guide (9800938)

8089 Assembler Pocket Reference (9800936)

MCS-86 Software Development Utilities
Operating Instructions for ISIS-II User's (9800639)

MCS-86 Absolute Object File Formats (9800821)

Universal PROM Programmer User's Manual (9800819)

Flexible Diskettes

—Single and Double* Density

*Recommended

ORDERING INFORMATION:

Part Number	Description
MDS-312	8089 Assembler Support Package

ICE-86™
8086 IN-CIRCUIT EMULATOR

Hardware in-circuit emulation

Full symbolic debugging

Breakpoints to halt emulation on a wide variety of conditions

Comprehensive trace of program execution, both conditional and unconditional

Disassembly of trace or memory from object code into assembler mnemonics

2K bytes of high speed ICE-86 mapped memory

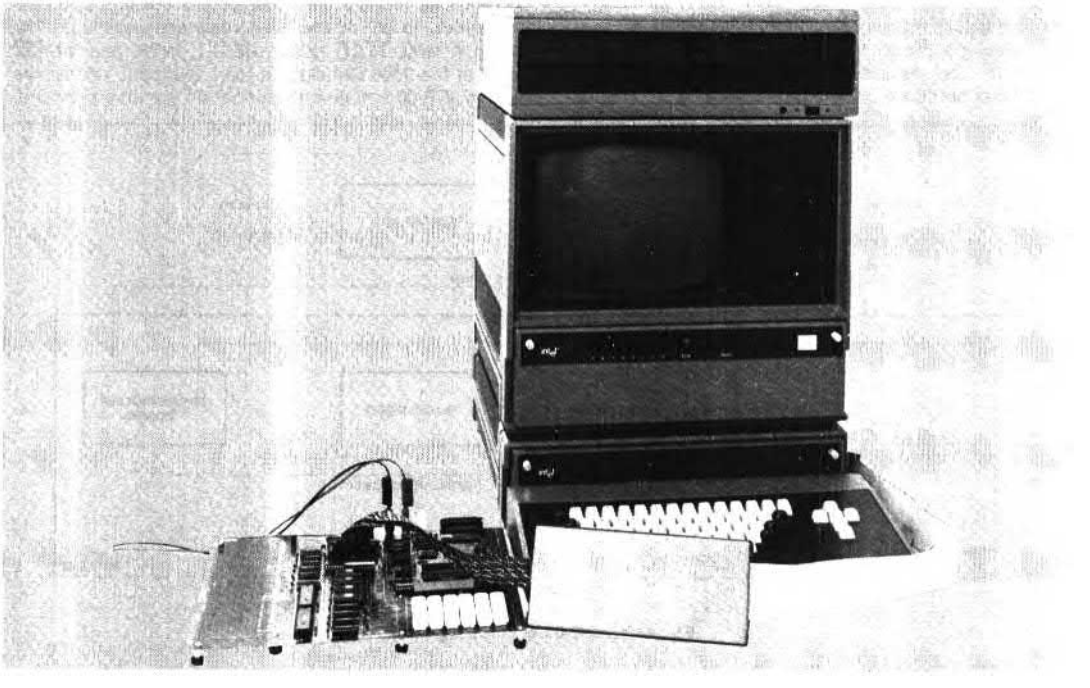
Software debugging with or without user system

Handles full 1 megabyte addressability of 8086

Compound commands

Command macros

The ICE-86 module provides In-Circuit Emulation for the 8086 microprocessor and the iSBC 86/12 Single Board Computer. It includes three circuit boards which reside in Intellec® Microcomputer Development Systems. A cable and buffer box connect the Intellec system to the user system by replacing the user's 8086. Powerful Intellec debug functions are thus extended into the user system. Using the ICE-86 module, the designer can execute prototype software in continuous or single-step mode and can substitute blocks of Intellec system memory for user equivalents. Breakpoints allow the user to stop emulation on user-specified conditions, and the trace capability gives a detailed history of the program execution prior to the break. All user access to the prototype system software may be done symbolically by referring to the source program variables and labels.



INTEGRATED HARDWARE/SOFTWARE DEVELOPMENT

The ICE-86 emulator allows hardware and software development to proceed interactively. This is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-86 module, prototype hardware can be added to the system as it is designed. Software and hardware testing occurs while the product is being developed.

Conceptually, the ICE-86 emulator assists three stages of development:

1. It can be operated without being connected to the user's system, so ICE-86 debugging capabilities can be used to facilitate program development before any of the user's hardware is available.
2. Integration of software and hardware can begin when any functional element of the user system hardware is connected to the 8086 socket. Through ICE-86 mapping capabilities, Intellec memory, ICE memory, or diskette memory can be substituted for missing prototype memory. Time-critical program modules are debugged before hardware implementation by using the 2K-bytes of high-speed ICE-resident memory. As each section of the user's hardware is completed, it is added to the prototype. Thus each section of the hardware and software is "system" tested as it becomes available.
3. When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-86 module is then used for real time emulation of the 8086 to debug the system as a completed unit.

Thus the ICE-86 module provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

SYMBOLIC DEBUGGING

Symbols and PL/M statement numbers may be substituted for numeric values in any of the ICE-86 commands. This allows the user to make symbolic references to I/O ports, memory addresses, and data in the user program. Thus the user need not remember the addresses of variables or program subroutines.

Symbols can be used to reference variables, procedures, program labels, and source statements. A variable can be displayed or changed by referring to it by name rather than by its absolute location in memory. Using symbols for statement labels, program labels, and procedure names simplifies both tracing and breakpoint setting. Disassembly of a section of code from either trace or program memory into its assembly mnemonics is readily accomplished.

Furthermore, each symbol may have associated with it one of the data types BYTE, WORD, INTEGER, SINTEGER (for short, 8-bit integer) or POINTER. Thus the user need not remember the type of a source program variable when examining or modifying it. For example, the command "!VAR" displays the value in memory of variable VAR in a format appropriate to its type, while the command "!VAR = !VAR + 1" increments the value of the variable.

The user symbol table generated along with the object file during a PL/M-86 compilation or an ASM-86 assembly is loaded into memory along with the user program which is to be emulated. The user may add to this symbol table any additional symbolic values for memory addresses, constants, or variables that are found useful during system debugging.

The ICE-86 module provides access through symbolic definition to all of the 8086 registers and flags. The READY, NMI, $\overline{\text{TEST}}$, HOLD, RESET, INTR, and MN/$\overline{\text{MX}}$ pins of the 8086 can also be read. Symbolic references to key ICE-86 emulation information are also provided.

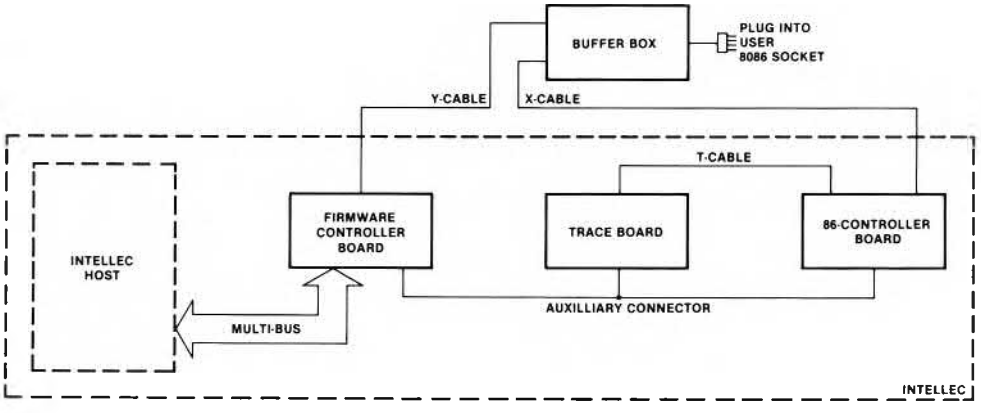
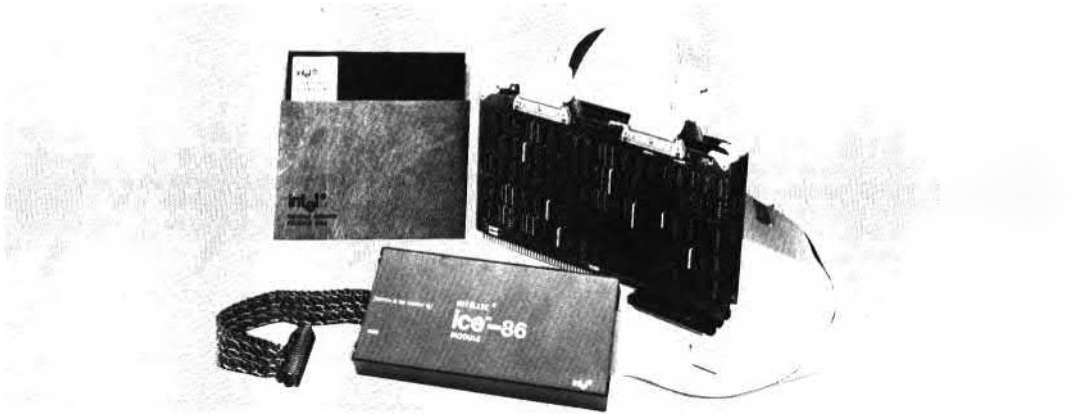


Figure 1. ICE-86 Block Diagram



A typical ICE-86 development configuration. It is based on a Model 230 Development System, which also includes a Double Density Diskette Operating System and a Model 201 Expansion Chassis (which holds the ICE-86 emulator). The ICE-86 module is shown connected to a user prototype system, in this case an SDK-86.

MACROS AND COMPOUND COMMANDS

The ICE-86 module provides a programmable diagnostic facility which allows the user to tailor its operation using macro commands and compound commands.

A macro is a set of ICE-86 commands which is given a single name. Thus, a sequence of commands which is executed frequently may be invoked simply by typing in a single command. The user first defines the macro by entering the entire sequence of commands which he wants to execute. He then names the macro and stores it for future use. He executes the macro by typing its name and passing up to ten parameters to the commands in the macro. Macros may be saved on a disk file for use in subsequent debugging sessions.

Compound commands provide conditional execution of commands(IF), and execution of commands until a condition is met or until they have been executed a specified number of times (COUNT, REPEAT).

Compound commands and macros may be nested any number of times.

MEMORY MAPPING

Memory for the user system can be resident in the user system or "borrowed" from the Intellec System through ICE-86's mapping capability.

The ICE-86 emulator allows the memory which is addressed by the 8086 to be mapped in 1K-byte blocks to:

1. Physical memory in the user's system,
2. Either of two 1K-byte blocks of ICE-86 high speed memory,
3. Intellec memory,
4. A random-access diskette file.

The user can also designate a block of memory as nonexistent. The ICE-86 module issues an error message when any such "guarded" memory is addressed by the user program.

Command	Description
GO	Initializes emulation and allows the user to specify the starting point and breakpoints. Example: GO FROM .START TILL .DELAY EXECUTED where START and DELAY are statement labels.
STEP	Allows the user to single-step through the program.

Table 1. Summary of ICE-86 Emulation Commands

OPERATION MODES

The ICE-86 software is a RAM-based program that provides the user with easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-86 commands are configured with a broad range of modifiers which provide the user with maximum flexibility in describing the operation to be performed.

Emulation

Emulation commands to the ICE-86 emulator control the process of setting up, running and halting an emulation of the user's 8086. Breakpoints and tracepoints enable ICE-86 to halt emulation and provide a detailed trace of execution in any part of the user's program. A summary of the emulation commands is shown in Table 1.

Breakpoints — The ICE-86 module has two breakpoint registers that allow the user to halt emulation when a specified condition is met. The breakpoint registers may be set up for execution or non-execution breaking. An execution breakpoint consists of a single address which causes a break whenever the 8086 executes from its queue an instruction byte which was obtained from

the address. A non-execution breakpoint causes an emulation break when a specified condition other than an instruction execution occurs. A non-execution breakpoint condition, using one or both breakpoint registers, may be specified by any one of or a combination of:

1. *A set of address values.* Break on a set of address values has three valuable features:
   a. Break on a single address.
   b. The ability to set any number of breakpoints within a limited range (1024 bytes maximum) of memory.
   c. The ability to break in an unlimited range. Execution is halted on any memory access to an address greater than (or less than) any 20-bit breakpoint address.
2. *A particular status of the 8086 bus* (one or more of: memory or I/O read or write, instruction fetch, halt, or interrupt acknowledge).
3. *A set of data values* (features comparable to break on a set of address values, explained in point one).
4. *A segment register* (break occurs when the register is used in an effective address calculation).

An external breakpoint match output for user access is provided on the buffer box. This allows synchronization of other test equipment when a break occurs.

Tracepoints — The ICE-86 module has two tracepoint registers which establish match conditions to conditionally start and stop trace collection. The trace information is gathered at least twice per bus cycle, first when the address signals are valid and second when the data signals are valid. If the 8086 execution queue is otherwise active, additional frames of trace are collected.

Each trace frame contains the 20 address/data lines and detailed information on the status of the 8086. The trace memory can store 1,023 frames, or an average of about 300 bus cycles, providing ample data for determining how the 8086 was reacting prior to emulation break. The trace memory contains the last 1,023 frames of trace data collected, even if this spans several separate emulations. The user has the option of displaying each frame of the trace data or displaying by instruction in actual ASM-86 Assembler mnemonics. Unless the user chooses to disable trace, the trace information is always available after an emulation.

Interrogation and Utility

Interrogation and utility commands give the user convenient access to detailed information about the user program and the state of the 8086 that is useful in debugging hardware and software. Changes can be made in both memory and the 8086 registers, flags, input pins, and I/O ports. Commands are also provided for various utility operations such as loading and saving program files, defining symbols and macros, displaying trace data, setting up the memory map, and returning control to ISIS-II. A summary of the basic interrogation and utility commands is shown in Table 2.

Memory/Register Commands

Display or change the contents of:
- Memory
- 8086 Registers
- 8086 Status flags
- 8086 Input pins
- 8086 I/O ports
- ICE-86 Pseudo-Registers (e.g. emulation timer)

Memory Mapping Commands

Display, declare, set, or reset the ICE-86 memory mapping.

Symbol Manipulation Commands

Display any or all symbols, program modules, and program line numbers and their associated values (locations in memory).

Set the domain (choose the particular program module) for the line numbers.

Define new symbols as they are needed in debugging.

Remove any or all symbols, modules, and program statements.

Change the value of any symbol.

TYPE

Assign or change the type of any symbol in the symbol table.

ASM

Disassemble user program memory into ASM-86 Assembler mnemonics.

PRINT

Display the specified portion of the trace memory.

LOAD

Fetch user symbol table and object code from the input file.

SAVE

Send user symbol table and object code to the output file.

LIST

Send a copy of all output (including prompts, input line echos, and error messages) to the chosen output device (e.g. disk, printer) as well as the console.

EVALUATE

Display the value of an expression in binary, octal, decimal, hexadecimal, and ASCII.

SUFFIX/BASE

Establish the default base for numeric values in input text/output display (binary, octal, decimal, or hexadecimal).

CLOCK

Select the internal (ICE-86 provided, for stand-alone mode only) or an external (user-provided) system clock.

RWTIMEOUT

Allows the user to time out READ/WRITE command signals based on the time taken by the 8086 to access Intellec memory or diskette memory.

ENABLE/DISABLE RDY

Enable or disable logical AND of ICE-86 Ready with the user Ready signal for accessing Intellec memory, ICE memory, or diskette memory.

Table 2. Summary of Basic ICE-86 Interrogation and Utility Commands

DIFFERENCES BETWEEN ICE-86 EMULATION AND THE 8086 MICROPROCESSOR

The ICE-86 module emulates the actual operation of the 8086 microprocessor with the following exceptions:

- The ICE-86 module will not respond to a user system NMI or RESET signal when it is out of emulation.
- Trap is ignored in single step mode and on the first instruction step of an emulation.
- The MIN/MAX line, which chooses the "minimum" or "maximum" configuration of the 8086, must not change dynamically in the user system.
- In the "minimum" mode, the user HOLD signal must remain active until HLDA is output by the ICE-86 emulator.
- The $\overline{RQ}/\overline{GT}$ lines in the "maximum" configuration are not supported.

The speed of run emulation by the ICE-86 module depends on where the user has mapped his memory. As the user prototype progresses to include memory, emulation becomes real time.

Memory Mapped To	Estimated Speed
User System	100% of real time*, up to 4 MHz clock
ICE	2 wait states per 8086-controlled bus cycle
Intellec	Approximately 0.02% of real time at 4 MHz clock
Diskette	**

*100% of real time is emulation at the user system clock rate with no wait states.

**The emulation speed from diskette is comparable to Intellec memory, but emulation must wait when a new page is accessed on the diskette.

DC CHARACTERISTICS OF ICE-86 USER CABLE

1. Output Low Voltages [V_{OL}(Max) = 0.4V]

	I_{OL} (Min)
AD0-AD15	8 mA (24 mA @ 0.5V)
A16/S3-A19/S7, $\overline{BHE}$/S7, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, M/$\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$	8 mA (16 mA @ 0.5V)
HLDA	7 mA
$\overline{MATCH0}$ OR $\overline{MATCH1}$ (on buffer box)	16 mA

2. Output High Voltages [V_{OH} (Min) = 2.4V]

	I_{OH}(Min)
AD0-AD15	– 2 mA
A16/S3-A19/S7, $\overline{BHE}$/S7, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, M/$\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$, HLDA	– 1 mA
$\overline{MATCH0}$ OR $\overline{MATCH1}$ (on buffer box)	– 0.8 mA

3. Input Low Voltages [V_{IL}(Max) = 0.8V]

	I_{IL} (Max)
AD0-AD15	– 0.2 mA
NMI, CLK	– 0.4 mA
READY	– 0.8 mA
INTR, HOLD, $\overline{TEST}$, RESET	– 1.4 mA
MN/$\overline{MX}$ (0.1 μf to GND)	– 3.3 mA

4. Input High Voltages [V_{IH}(Min) = 2.0V]

	I_{IH}(Max)
AD0-AD15	80 μA
NMI, CLK	20 μA
READY	40 μA
INTR, HOLD, $\overline{TEST}$, RESET	– 0.4 mA
MN/$\overline{MX}$ (0.1 μF to GND)	– 1.1 mA

5. $\overline{RQ}/\overline{GT0}$, $\overline{RQ}/\overline{GT1}$ are pulled up to + 5V through a 5.6K ohm resistor. No current is taken from user circuit at V_{CC} pin.

SPECIFICATIONS

ICE-86 Operating Environment

Required Hardware

Intellec microcomputer development system with:

1. Three adjacent slots for the ICE-86 module (Series II requires Model 201 Expansion Chassis.)
2. 64K bytes of Intellec memory. If user prototype program memory is desired, additional memory above the basic 64K is required.

System console
Intellec diskette operating system
ICE-86 module

Required Software

System monitor
ISIS-II, version 3.4 or subsequent
ICE-86 software

Equipment Supplied

Printed circuit boards (3)
Interface cable and emulation buffer module
Operator's manual
ICE-86 software, diskette-based

Emulation Clock

User system clock up to 4 MHz or 2 MHz ICE-86 internal clock in stand-alone mode

Physical Characteristics

Printed Circuit Boards

Width: 12.00 in (30.48 cm)
Height: 6.75 in (17.15 cm)
Depth: 0.50 in (1.27 cm)
Packaged Weight: 9.00 lb (4.10 kg)

Electrical Characteristics

DC Power

V_{CC} = +5V +5% −1%
I_{CC} = 15A maximum; 11A typical
V_{DD} = +12V ±5%
I_{DD} = 120 mA maximum; 80 mA typical
V_{BB} = −10V ±5% or −12V ±5% (optional)
I_{BB} = 15 mA maximum; 12 mA typical

Environmental Characteristics

Operating Temperature: 0° to 40°C

Operating Humidity: Up to 95% relative humidity without condensation.

ORDERING INFORMATION

Part Number	Description
MDS-86-ICE	8086 CPU in-circuit emulator

iSBC 86/12A
SINGLE BOARD COMPUTER

8086 16 bit HMOS microprocessor central processor unit

32K-bytes of dual-port read/write memory expandable on-board to 64K-bytes with on-board refresh

Sockets for up to 16K-bytes of read only memory expandable on-board to 32K-bytes

System memory expandable to 1 megabyte

24 programmable parallel I/O lines with sockets for interchangeable line drivers and terminators

Programmable synchronous/asynchronous RS232C compatible serial interface with software selectable baud rates

Two programmable 16-bit BCD or binary timers/event counters

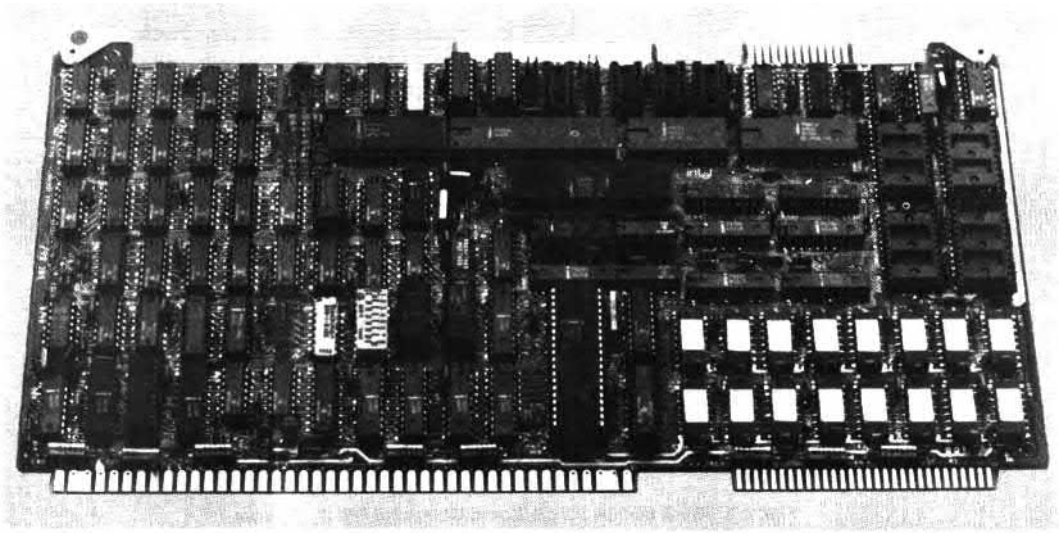
9 levels of vectored interrupt control, expandable to 65 levels

Auxiliary power bus and power fail interrupt control logic for read/write memory battery backup

MULTIBUS interface for multimaster configurations and system expansion

Compatible with iSBC 80 family single board computers, memory, digital and analog I/O, and peripheral controller boards

The iSBC 86/12A Single Board Computer is a member of Intel's complete line of OEM microcomputer systems which take full advantage of Intel's LSI technology to provide economical self-contained computer based solutions for OEM applications. The iSBC 86/12A board is a complete computer system on a single 6.75 x 12.00-inch printed circuit card. The CPU, system clock, read/write memory, nonvolatile read only memory, I/O ports and drivers, serial communications interface, priority interrupt logic and programmable timers, all reside on the board. Full MULTIBUS interface logic is included to offer compatibility with the Intel OEM Microcomputer Systems family of Single Board Computers, expansion memory options, digital and analog I/O expansion boards and peripheral controllers.



FUNCTIONAL DESCRIPTION

Central Processing Unit

The central processor for the iSBC 86/12A board is Intel's 8086, a powerful 16-bit HMOS device. The 225 sq. mil chip contains 29,000 transistors and has a clock rate of 5MHz. The architecture includes four (4) 16-bit byte addressable data registers, two (2) 16-bit memory base pointer registers and two (2) 16-bit index registers, all accessed by a total of 24 operand addressing modes for complex data handling and very flexible memory addressing.

Instruction Set — The 8086 instruction repertoire includes variable length instruction format (including double operand instructions), 8-bit and 16-bit signed and unsigned arithmetic operators for binary, BCD and unpacked ASCII data, and iterative word and byte string manipulation functions. The instruction set of the 8086 is a superset of the 8080A/8085A family and with available software tools, programs written for the 8080A/8085A can be easily converted and run on the 8086 processor.

Architectural Features — A 6-byte instruction queue provides pre-fetching of sequential instructions and can reduce the 1.2μsec minimum instruction cycle to 400 nsec for queued instructions. The stack oriented architecture facilitates nested subroutines and co-routines, reentrant code and powerful interrupt handling. The memory expansion capabilities offer a 1 megabyte addressing range. The dynamic relocation scheme allows ease in segmentation of pure procedure and data for efficient memory utilization. Four segment registers (code, stack, data, extra) contain program loaded offset values which are used to map 16-bit addresses to 20-bit addresses. Each register maps 64K-bytes at a time and activation of a specific register is controlled explicitly by program control and is also selected implicitly by specific functions and instructions.

Bus Structure

The iSBC 86/12A microcomputer has three buses: an internal bus for communicating with on-board memory and I/O options, the MULTIBUS system bus for referencing additional memory and I/O options, and the dual-port bus which allows access to RAM from the on-board CPU and the MULTIBUS system bus. Local (on-board) accesses do not require MULTIBUS communication, making the system bus available for use by other MULTIBUS masters (i.e. DMA devices and other single board computers transferring to additional system memory). This feature allows true parallel processing in a multiprocessor environment. In addition, the MULTIBUS interface can be used for system expansion through the use of other 8- and 16-bit iSBC computers, memory and I/O expansion boards.

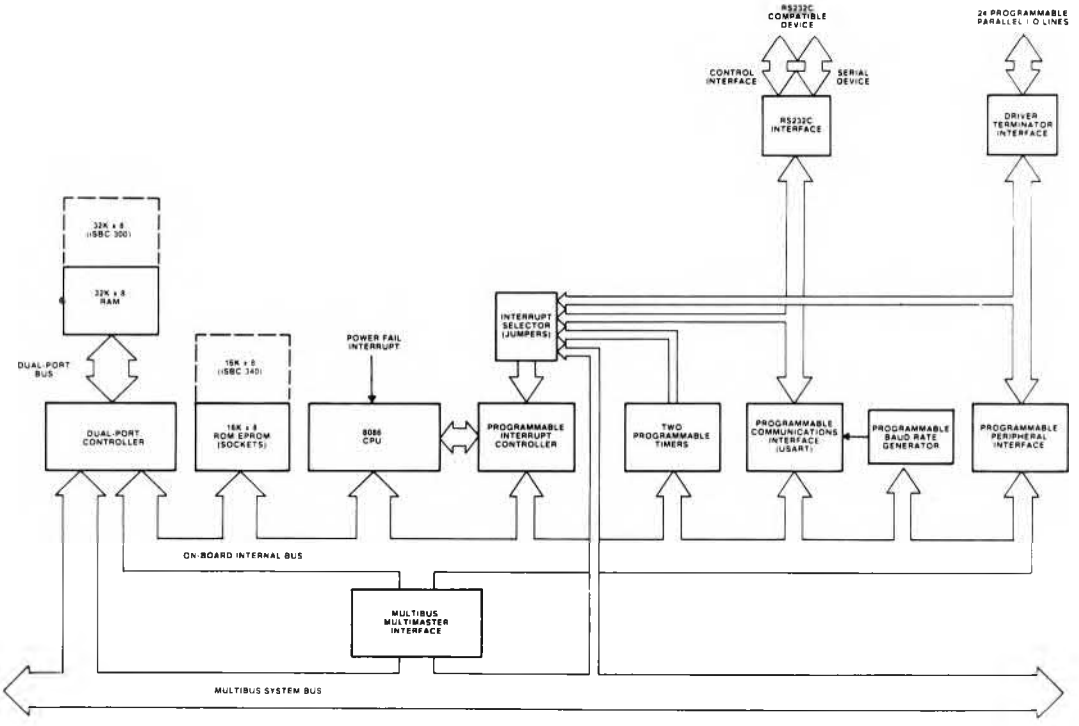


Figure 1. iSBC 86/12A Single Board Computer Block Diagram

RAM Capabilities

The iSBC 86/12A microcomputer contains 32K bytes of dynamic read/write memory using 16K-bit 2117 RAMs. In addition, the on-board RAM complement may be expanded to 64K bytes with the iSBC 300 32K-byte RAM expansion module. Power for the on-board RAM and refresh circuitry may be optionally provided on an auxiliary power bus, and memory protect logic is included for RAM battery backup requirements. The iSBC 86/12A board contains a dual port controller which allows access to the on-board RAM (32K bytes or 64K bytes when the iSBC 300 module is included with the iSBC 86/12A board) from the iSBC 86/12A CPU and from any other MULTIBUS master via the system bus. The dual port controller allows 8- and 16-bit accesses from the MULTIBUS system bus, and the on-board CPU transfers data to RAM over a 16-bit data path. Priorities have been established such that memory refresh is guaranteed by the on-board refresh logic and that the on-board CPU has priority over MULTIBUS system bus requests for access to RAM. The dual port controller includes independent addressing logic for RAM access from the on-board CPU and from the MULTIBUS system bus. The on-board CPU will always access RAM starting at location 00000_H. Address jumpers allow on-board RAM to be located starting on any 8K-byte boundary within a 1 megabyte address range for accesses from the MULTIBUS system bus. In conjunction with this feature, the iSBC 86/12A microcomputer has the ability to protect on-board memory from MULTIBUS access to any contiguous 8K-byte segments (or 16K-byte segments with iSBC 300 module). These features allow multiprocessor systems to establish local memory for each processor and shared system (MULTIBUS) memory configurations where the total system memory size (including local on-board memory) can exceed 1 megabyte without addressing conflicts.

EPROM/ROM Capabilities

Four sockets are provided for up to 16K-bytes of nonvolatile read only memory on the iSBC 86/12A board. EPROM/ROM may be added in 2K-byte increments up to a maximum of 4K-bytes by using Intel 2758 electrically programmable ROMs (EPROMs); in 4K-byte increments up to 8K bytes by using Intel 2716 EPROMs or Intel 2316E masked ROMs; or in 8K-byte increments up to 16K bytes by using Intel 2732 EPROMs or 2332A ROMs. On-board EPROM/ROM is accessed via 16-bit data paths. On-board EPROM/ROM capacity may be expanded to 32K bytes with the addition of the iSBC 340 16K-byte EPROM expansion module. It provides an additional four sockets for Intel 2732 EPROMs or Intel 2332A ROMs. With user modification of the iSBC 86/12A's on-board memory and MULTIBUS address decode, Intel 2758 and 2716 EPROMs or 2316E ROMs may be optionally supported. System memory size is easily expanded by the addition of MULTIBUS system bus compatible memory boards available in the iSBC product family.

Parallel I/O Interface

The iSBC 86/12A single board computer contains 24 programmable parallel I/O lines implemented using the Intel 8255A Programmable Peripheral Interface. The system software is used to configure the I/O lines in any combination of unidirectional input/output and bidirectional ports indicated in Table 1. Therefore, the I/O interface may be customized to meet specific peripheral requirements. In order to take full advantage of the large number of possible I/O configurations, sockets are provided for interchangeable I/O line drivers and terminators. Hence, the flexibility of the I/O interface is further enhanced by the capability of selecting the appropriate combination of optional line drivers and terminators to provide the required sink current, polarity, and drive/termination characteristics for each application. The 24 programmable I/O lines and signal ground lines are brought out to a 50-pin edge connector that mates with flat, woven, or round cable.

Serial I/O

A programmable communications interface using the Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the iSBC 86/12A board. A software selectable baud rate generator provides the USART with all common communication

Port	Lines (qty)	Mode of Operation					Control
		Unidirectional				Bidirectional	
		Input		Output			
		Latched	Latched & Strobed	Latched	Latched & Strobed		
1	8	X	X	X	X	X	
2	8	X	X	X	X		
3	4	X		X			X[1]
	4	X		X			X[1]

Note
1. Part of port 3 must be used as a control port when either port 1 or port 2 are used as a latched and strobed input or a latched and strobed output port or port 1 is used as a bidirectional port.

Table 1. Input/Output Port Modes of Operation

frequencies. The USART can be programmed by the system software to select the desired asynchronous or synchronous serial data transmission technique (including IBM Bi-Sync). The mode of operation (i.e., synchronous or asynchronous), data format, control character format, parity, and baud rate are all under program control. The 8251A provides full duplex, double buffered transmit and receive capability. Parity, overrun, and framing error detection are all incorporated in the USART. The RS232C compatible interface on each board, in conjunction with the USART, provides a direct interface to RS232C compatible terminals, cassettes, and asynchronous and synchronous modems. The RS232C command lines, serial data lines, and signal ground line are brought out to a 26 pin edge connector that mates with RS232C compatible flat or round cable. The iSBC 530 Teletypewriter Adapter provides an optically isolated interface for those systems requiring a 20 mA current loop. The iSBC 530 unit may be used to interface the iSBC 86/12A board to teletypewriters or other 20 mA current loop equipment.

Programmable Timers

The iSBC 86/12A board provides three independent, fully programmable 16-bit interval timers/event counters utilizing the Intel 8253 Programmable Interval Timer. Each counter is capable of operating in either BCD or binary modes. Two of these timers/counters are available to the systems designer to generate accurate time intervals under software control. Routing for the outputs and gate/trigger inputs of two of these counters is jumper selectable. The outputs may be independently routed to the 8259A Programmable Interrupt Controller and to the I/O line drivers associated with the 8255A Programmable Peripheral Interface, or may be routed as inputs to the 8255A chip. The gate/trigger inputs may be routed to I/O terminators associated with the 8255A or as output connections from the 8255A. The third interval timer in the 8253 provides the programmable baud rate generator for the iSBC 86/12A board RS232C USART serial port. In utilizing the iSBC 86/12A board the systems designer simply configures, via software, each timer independently to meet system requirements. Whenever a given time delay or count is needed, software commands to the programmable timers/event counters select the desired function. Seven functions are available, as shown in Table 2. The contents of each counter may be read at any time during system operation with simple read operations for event counting applications, and special commands are included so that the contents can be read "on the fly".

MULTIBUS System Bus and Multimaster Capabilities

The MULTIBUS system bus features asynchronous data transfers for the accommodation of devices with various transfer rates while maintaining maximum throughput. Twenty address lines and sixteen separate data lines eliminate the need for address/data multiplexing/demultiplexing logic used in other systems, and allow for data transfer rates up to 5 megawords/sec. A failsafe timer is included in the iSBC 86/12A board which can be used to generate an interrupt if an addressed device does not respond within 6 msec.

Function	Operation
Interrupt on terminal count	When terminal count is reached, an interrupt request is generated. This function is extremely useful for generation of real-time clocks.
Programmable one-shot	Output goes low upon receipt of an external trigger edge or software command and returns high when terminal count is reached. This function is retriggerable.
Rate generator	Divide by N counter. The output will go low for one input clock cycle, and the period from one low going pulse to the next is N times the input clock period.
Square-wave rate generator	Output will remain high until one-half the count has been completed, and go low for the other half of the count.
Software triggered strobe	Output remains high until software loads count (N). N counts after count is loaded, output goes low for one input clock period.
Hardware triggered strobe	Output goes low for one clock period N counts after rising edge counter trigger input. The counter is retriggerable.
Event counter	On a jumper selectable basis, the clock input becomes an input from the external system. CPU may read the number of events occurring after the counting "window" has been enabled or an interrupt may be generated after N events occur in the system.

Table 2. Programmable Timer Functions

Multimaster Capabilities — The iSBC 86/12A board is a full computer on a single board with resources capable of supporting a great variety of OEM system requirements. For those applications requiring additional processing capacity and the benefits of multiprocessing (i.e., several CPUs and/or controllers logically sharing system tasks through communication over the system bus), the iSBC 86/12A board provides full MULTIBUS arbitration control logic. This control logic allows up to three iSBC 86/12A boards or other bus masters, including iSBC 80 family MULTIBUS compatible 8-bit single board computers, to share the system bus in serial (daisy chain) priority fashion and up to 16 masters to share the MULTIBUS system bus with the addition of an external priority network. The MULTIBUS arbitration logic operates synchronously with a MULTIBUS clock (provided by the iSBC 86/12A board or optionally provided directly from the MULTIBUS) while data is transferred via a handshake between the master and slave modules. This allows different speed controllers to share resources on the same bus, and transfers via the bus proceed asynchronously. Thus, transfer speed is dependent on transmitting and

receiving devices only. This design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations, high speed peripheral control, but are by no means limited to these three.

Interrupt Capablity

The iSBC 86/12A board provides 9 vectored interrupt levels. The highest level is the NMI (Non-maskable Interrupt) line which is directly tied to the 8086 CPU. This interrupt cannot be inhibited by software and is typically used for signalling catastrophic events (i.e., power failure). On servicing this interrupt, program control will be implicitly transferred through location 00008_H. The Intel 8259A Programmable Interrupt Controller (PIC) provides vectoring for the next eight interrupt levels. As shown in Table 3, a selection of four priority processing modes is available to the systems designer for use in designing request processing configurations to match system requirements. Operating mode and priority assignments may be reconfigured dynamically via software at any time during system operation. The PIC accepts interrupt requests from the programmable parallel and serial I/O interfaces, the programmable timers, the system bus, or directly from peripheral equipment. The PIC then determines which of the incoming requests is of the highest priority, determines whether this request is of higher priority than the level currently being serviced, and, if appropriate, issues an interrupt to the CPU. Any combination of interrupt levels may be masked, via software, by storing a single byte in the interrupt mask register of the PIC. The PIC generates a unique memory address for each interrupt level. These addresses are equally spaced at 4 byte intervals. This 32-byte block may begin at any 32-byte boundary in the lowest 1K-bytes of memory,* and contains unique instruction pointers and code segment offset values (for expanded memory operation) for each interrupt level. After acknowledging an interrupt and obtaining a device identifier byte from the 8259A PIC, the CPU will store its status flags on the stack and execute an indirect CALL instruction through the vector location (derived from the device identifier) to the interrupt service routine. In systems requiring additional interrupt levels, slave 8259A PIC's may be interfaced via the MULTIBUS system bus, to generate additional vector addresses, yielding a total of 65 unique interrupt levels.

Interrupt Request Generation — Interrupt requests may originate from 17 sources. Two jumper selectable interrupt requests can be automatically generated by the programmable peripheral interface when a byte of information is ready to be transferred to the CPU (i.e., input buffer is full) or a byte of information has been transferred to a peripheral device (i.e., output buffer is empty). Two jumper selectable interrupt requests can be automatically generated by the USART when a character is ready to be transferred to the CPU (i.e., receive channel buffer is full, or a character is ready to be transmitted (i.e., transmit channel data buffer is empty). A jumper selectable request can be generated by each of the programmable timers. An additional interrupt request line may be jumpered directly from the parallel I/O driver terminator section. Eight prioritized interrupt request lines allow the iSBC 86/12A board to recognize and service interrupts originating from peripheral boards interfaced via the MULTIBUS system bus. The MULTIBUS fail safe timer also can be selected as an interrupt source.

*Note: The first 32 vector locations are reserved by Intel for dedicated vectors. Users who wish to maintain compatibility with present and future Intel products should not use these locations for user-defined vector addresses.

Mode	Operation
Fully nested	Interrupt request line priorities fixed at 0 as highest, 7 as lowest.
Auto-rotating	Equal priority. Each level, after receiving service, becomes the lowest priority level until next interrupt occurs.
Specific priority	System software assigns lowest priority level. Priority of all other levels based in sequence numerically on this assignment.
Polled	System software examines priority-encoded system interrupt status via interrupt status register.

Table 3. Programmable Interrupt Modes

Power-Fail Control

Control logic is also included to accept a power-fail interrupt in conjunction with the AC-low signal from the iSBC 635 and iSBC 640 Power Supply or equivalent.

Expansion Capabilities

Memory and I/O capacity may be expanded and additional functions added using Intel MULTIBUS compatible expansion boards. Memory may be expanded by adding user specified combinations of RAM boards, EPROM boards, or combination boards. Input/output capacity may be increased by adding digital I/O and analog I/O expansion boards. Mass storage capability may be achieved by adding single or double density diskette controllers, or hard disk controllers. Modular expandable backplanes and cardcages are available to support multiboard systems.

Note: Certain system restrictions may be incurred by the inclusion of some of the iSBC 80 family options in an iSBC 86/12A system. Consult the Intel OEM Microcomputer System Configuration Guide for specific data.

System Development Capabilities

The development cycle of iSBC 86/12A products can be significantly reduced by using the Intellec® series microcomputer development systems. The Assembler, Locating Linker, Library Manager, Text Editor and system monitor are all supported by the ISIS-II disk based operating system. A minimum of 64K-bytes of RAM is needed in the Intellec system to support program development for the iSBC 86/12A board. To facilitate conversion of 8080A/8085A assembly language programs to un on the iSBC 86/12A board CONV-86 is available under the ISIS-II operating system.

In-Circuit Emulator — ICE-86 in-circuit emulator provides the necessary link between the software development environment provided by the Intellec system and the "target" iSBC 86/12A execution system. In addition to providing the mechanism for loading executable code and data into the iSBC 86/12A board, ICE-86 in-circuit emulator provides a sophisticated command set to assist in debugging software and final integration of the user hardware and software. ICE-86 in-circuit emulator maximizes the use of available development resources by allowing Intellec resident resources (e.g., memory and peripherals) to be accessed by software running on the target iSBC 86/12A system. In addition, software can be executed without an iSBC 86/12A execution vehicle, in 2K bytes of RAM resident in the ICE-86 system itself. Symbolic references to instruction and data locations can be made through ICE-86 in-circuit emulator to allow the user to reference memory locations with assigned names.

PL/M-86 — Intel's high level programming language, PL/M-86, is also available as an Intellec Microcomputer Development System option. PL/M-86 provides the capability to program in a natural, algorithmic language and eliminates the need to manage register usage or allocate memory. PL/M-86 programs can be written in a much shorter time than assembly language programs for a given application. PL/M-86 includes byte and word, integer, pointer and floating point (32-bit) data types and also includes conditional compilation and macro features.

SPECIFICATIONS

Word Size

Instruction — 8, 16, 24, or 32 bits
Data — 8, 16 bits

Cycle Time

Basic Instruction Cycle — 1.2 μsec
— 400 nsec (assumes instruction in the queue)

Note:
Basic instruction cycle is defined as the fastest instruction time (i.e., two clock cycles)

Memory Capacity

On-Board Read Only Memory — 16K bytes (sockets only); expandable to 32K bytes with iSBC 340 EPROM/ROM expansion module.

On-Board RAM — 32K bytes; expandable to 64K bytes with iSBC 300 RAM expansion module.

Off-Board Expansion — Up to 1 megabyte in user specified combinations of RAM, ROM, and EPROM.

Note:
Read only memory may be added in 2K, 4K, or 8K-byte increments.

Memory Addressing

On-Board EPROM/ROM — FF000-FFFFF$_H$ (using 2758 EPROMs); FE000-FFFFF$_H$ (using 2716 EPROMs or 2316 ROMs); FC000-FFFFF$_H$ (using 2732 EPROMs or 2332A ROMs); F8000-FFFFF$_H$ (with iSBC 340 EPROM option and four additional 2732 EPROMs).

On-Board RAM — 32K bytes of dual port RAM. Optionally expandable to 64K bytes with iSBC 300 RAM option.

CPU Access — 32K bytes: 00000-07FFF$_H$; 64K bytes: 00000-0FFFF$_H$.

MULTIBUS Access — Jumper selectable for any 8K-byte boundary, but not crossing a 128K-byte boundary. Access for 8K, 16K, 24K or 32K (16K, 32K, 48K, 64K with iSBC 300 option) bytes may be selected for on-board CPU use only.

I/O Capacity

Parallel — 24 programmable lines using one 8255A.
Serial — 1 programmable line using one 8251A.

I/O Addressing

On-Board Programmable I/O

Port	8255A				USART	
	1	2	3	Control	Data	Control
Address	C8	CA	CC	CE	D8 or DC	DA or DE

Serial Communications Characteristics

Synchronous — 5—8 bit characters; internal or external character synchronization; automatic sync insertion.

Asynchronous — 5—8 bit characters; break character generation; 1, 1½, or 2 stop bits; false start bit detection.

Baud Rates

Frequency (kHz) (Software Selectable)	Baud Rate (Hz)		
	Synchronous	Asynchronous	
		÷ 16	÷ 64
153.6	—	9600	2400
76.8	—	4800	1200
38.4	38400	2400	600
19.2	19200	1200	300
9.6	9600	600	150
4.8	4800	300	75
2.4	2400	150	—
1.76	1760	110	—

Note:
Frequency selected by I/O write of appropriate 16-bit frequency factor to baud rate register (8253 Timer 2).

Interrupts

Addresses for 8259A Registers (Hex notation I/O address space)

C0 or C4 Write: Initialization Command Word 1 (ICW1) and Operation Control Words 2 and 3 (OCW2 and OCW3)
Read: Status and Poll Registers

C2 or C6 Write: ICW2, ICW3, ICW4, OCW1 (Mask Register)
Read: OCW1 (Mask Register)

Note:
Several registers have the same physical address; sequence of access and one data bit of control word determine which register will respond.

Interrupt Levels — 8086 CPU includes a non-maskable Interrupt (NMI) and a maskable interrupt (INTR). NMI interrupt is provided for catastrophic events such as power failure. NMI vector address is 00008. INTR interrupt is driven by on-board 8259A PIC, which provides 8-bit identifier of interrupting device to CPU. CPU multiplies identifier by four to derive vector address. Jumpers select interrupts from 17 sources without necessity of external hardware. PIC may be programmed to accommodate edge-sensitive or level-sensitive inputs.

Timers

Register Addresses (Hex notation, I/O address space)

D0 Timer 0
D2 Timer 1
D4 Timer 2
D6 Control register

Note:
Timer counts are loaded as two sequential output operations to same address as given

Input Frequencies

Reference: 2.46 MHz ±0.1% (0.041 μs period, nominal); 1.23 MHz ±0.1% (0.81 μs period, nominal); or 153.60 kHz ±0.1% (6.51 μs period nominal).

Note:
Above frequencies are user selectable.

Event Rate: 2.46 MHz max

Output Frequencies/Timing Intervals

Function	Single Timer/Counter		Dual Timer/Counter (Two Timers Cascaded)	
	Min	Max	Min	Max
Real-time interrupt	1.63 μs	427.1 ms	3.26 s	466.50 min
Programmable one-shot	1.63 μs	427.1 ms	3.26 s	466.50 min
Rate generator	2.342 Hz	613.5 kHz	0.000036 Hz	306.8 kHz
Square-wave rate generator	2.342 Hz	613.5 kHz	0.000036 Hz	306.8 kHz
Software triggered strobe	1.63 μs	427.1 ms	3.26 s	466.50 min
Hardware triggered strobe	1.63 μs	427.1 ms	3.26 s	466.50 min
Event counter	—	2.46 MHz	—	—

Interfaces

MULTIBUS — All signals TTL compatible
Parallel I/O — All signals TTL compatible
Interrupt Requests — All TTL compatible
Timer — All signals TTL compatible
Serial I/O — RS232C compatible, data set configuration

System Clock (8086 CPU)

5.00 MHz ± 0.1%

Auxiliary Power

An auxiliary power bus is provided to allow separate power to RAM for systems requiring battery backup of read/write memory. Selection of this auxiliary RAM power bus is made via jumpers on the board.

Connectors

Interface	Pins (qty)	Centers (in.)	Mating Connectors
Bus	86	0.156	VIKING 3KH43/9AMK12
Parallel I/O	50	0.1	3M 3415-000
Serial I/O	26	0.1	3M 3462-000

Memory Protect

An active low TTL compatible memory protect signal is brought out on the auxiliary connector which, when asserted, disables read/write access to RAM memory on the board. This input is provided for the protection of RAM contents during system power down sequences.

Line Drivers and Terminators

I/O Drivers — The following line drivers are all compatible with the I/O driver sockets on the iSBC 86/12A board.

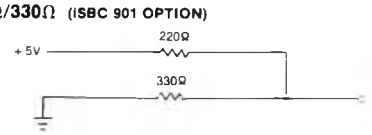
Driver	Characteristic	Sink Current (mA)
7438	I,OC	48
7437	I	48
7432	NI	16
7426	I,OC	16
7409	NI,OC	16
7408	NI	16
7403	I,OC	16
7400	I	16

Note:
I = inverting; NI = non-inverting, OC = open collector.

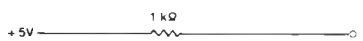
Port 1 of the 8255A has 20 mA totem-pole bidirectional drivers and 1 kΩ terminators.

I/O Terminators — 220Ω/330Ω divider or 1 kΩ pullup

220Ω/330Ω (ISBC 901 OPTION)



1 K Ω (ISBC 902 OPTION)



Bus Drivers

Function	Characteristic	Sink Current (mA)
Data	Tri-state	50
Address	Tri-state	50
Commands	Tri-state	32

Physical Characteristics

Width — 12.00 in. (30.48 cm)
Height — 6.75 in. (17.15 cm)
Depth — 0.70 in. (1.78 cm)
Weight — 19 oz. (539 gm)

Electrical Characteristics

DC Power Requirements

Configuration	Current Requirements			
	V_{CC} = +5V ±5% (max)	V_{DD} = +12V ±5% (max)	V_{BB} = −5V ±5% (max)	V_{AA} = −12V ±5% (max)
Without EPROM[1]	5.2A	350 mA	—	40 mA
RAM Only[3]	390 mA	40 mA	1.0 mA	—
With iSBC 530[4]	5.2A	450 mA	—	140 mA
With 4K EPROM[5] (using 2758)	5.5A	350 mA	—	40 mA
With 8K ROM[5] (using 2316E)	6.1A	350 mA	—	40 mA
With 8K EPROM[5] (using 2716)	5.5A	350 mA	—	40 mA
With 16K ROM[5] (using 2732 or 2332A)	5.4A	350 mA	—	40 mA

Notes:
1. Does not include power for optional ROM/EPROM, I/O drivers, and I/O terminators.
2. Does not include power required for optional ROM/EPROM, I/O drivers and I/O terminators.
3. RAM chips powered via auxiliary power bus.
4. Does not include power for optional ROM/EPROM, I/O drivers, and I/O terminators. Power for iSBC 530 is supplied via serial port connector.
5. Includes power required for four ROM/EPROM chips, and I/O terminators installed for 16 I/O lines; all terminator inputs low.

Environmental Characteristics

Operating Temperature — 0°C to 55°C
Relative Humidity — to 90% (without condensation)

Reference Manual

9803074-01 — iSBC 896/12A Single Board Computer Hardware Reference Manual (NOT SUPPLIED)

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

ORDERING INFORMATION

Part Number	Description
SBC 86/12A	Single Board Computer with 32K bytes RAM

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051
Tel: (408) 987-8088**
TWX: 910-338-0026
TELEX: 34-6372

iSBC 957
INTELLEC - iSBC 86/12A INTERFACE AND EXECUTION PACKAGE

Establishes communication between the iSBC 86/12A and the Intellec Development Systems to aid in MCS-86™ software development

Allows full speed execution of MCS-86™ programs

Includes EPROM resident system monitor for iSBC 86/12A

Allows Intellec ISIS-II files to be transferred between iSBC 86/12A and Intellec Microcomputer Development System

Offers "Virtual Terminal" capability which permits the Intellec console to access the iSBC 86/12A Monitor

Provides powerful console commands for software debug

Allows access to all iSBC 86/12A memory, registers, flags and I/O ports

Includes all necessary hardware, software and documentation

The iSBC 957 Intellec-iSBC 86/12A Interface and Execution Package contains all the necessary hardware, software cables and documentation required to interface an iSBC 86/12A Single Board Computer to an Intellec Microcomputer Development System for software development and full speed execution.

FUNCTIONAL DESCRIPTION

Overview

The iSBC 957 Intellec-iSBC 86/12A Interface and Execution Package extends the software development capabilities of the Intellec Microcomputer Development Systems to the iSBC 86/12 and iSBC 86/12A Single Board Computers. It allows software modules developed under the Intellec resident ISIS-II Operating Systems to be down loaded to the iSBC 86/12A for full-speed execution and debug. In addition, the iSBC 957 allows segments of iSBC 86/12A memory to be saved on floppy disk files. Special communication software allows transparent access to the powerful debug commands in the iSBC 86/12A monitor from the Intellec console terminal.

Software Capabilities

The software included in the iSBC 957 package consists of the iSBC 86/12A monitor residing on four Intel EPROMs which are inserted into sockets on the iSBC 86/12A board. A diskette is also included which contains the Intellec resident communications software that links the iSBC 86/12A with the Intellec Microcomputer Development System. The EPROM resident software creates an execution environment in which object modules may be loaded into the iSBC 86/12A memory, executed at full speed, modified if necessary and saved on the Intellec system floppy disk. The monitor provides the ability to execute selected program segments with breakpoints or by single stepping, examine and modify registers and memory, perform port I/O, move a block of memory, compare blocks of memory, search for a word/

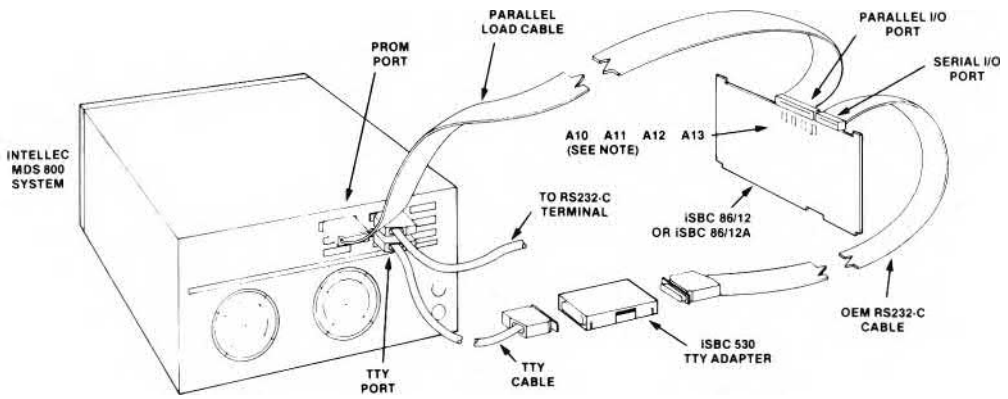
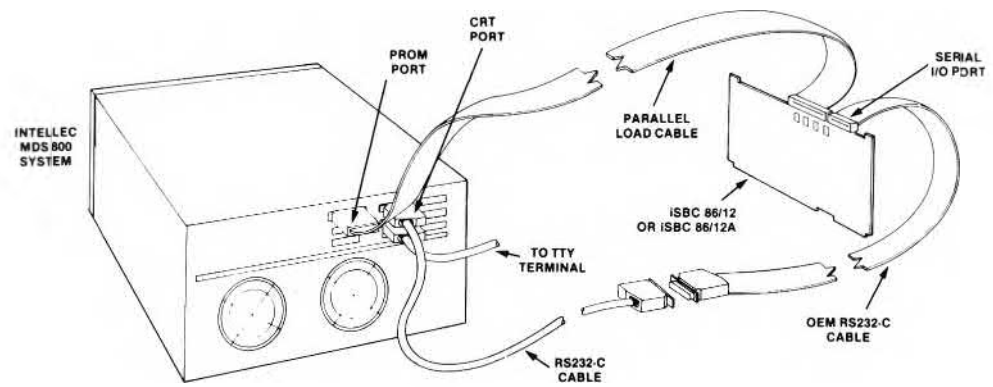


Figure 1a. Intellec MDS-800 Series System Using RS232-C Compatible Terminal



Note: A10, A12, A13 — Insert Terminator Pack (supplied)
A11 — Insert Status Adapter Ass'y (supplied)

Figure 1b. Intellec MDS-800 Series System with TTY Terminal

byte value, and perform hex arithmetic. In addition, the monitor provides for the recognition of interrupts via a user-defined table. The program on the diskette contains communication software which passes appropriate console commands to the iSBC 86/12A resident monitor and also interfaces with the ISIS-II operating system to transfer files between the development system diskettes and the iSBC 86/12A.

System Interfacing

The physical interface between the Intellec Microcomputer Development System and the iSBC 86/12A is accomplished with cables supplied with the iSBC 957 package. The cabling arrangement varies depending on whether the system is a member of the Intellec MDS-800 family or one of the Intellec Series II family.

Intellec MDS-800 Interface — In the case of the Intellec MDS-800 family, cables connect the serial I/O port of the iSBC 86/12A to the available serial port on the Intellec system (if the TTY port is used for the iSBC 86/12A interface, the iSBC 530 TTY adapter is inserted into the line). (See Figure 1.) This serial port implements the communication link from the Intellec console terminal to the iSBC 86/12A resident monitor via the Intellec based communication software and is used to pass commands to the iSBC 86/12A. Additionally, a cable is run from the Universal PROM Programmer (UPP) port on the Intellec system to the parallel I/O port on the iSBC 86/12A. The necessary terminators/line drivers and a status adapter assembly are also included to complete this parallel interface on the iSBC 86/12A. This interconnection is used for transferring the ISIS-II disk files between the development system and the iSBC 86/12A.

Intellec Series II Interface — For Intellec Series II Development Systems the connection between it and the iSBC 86/12A is accomplished with a single serial line in-

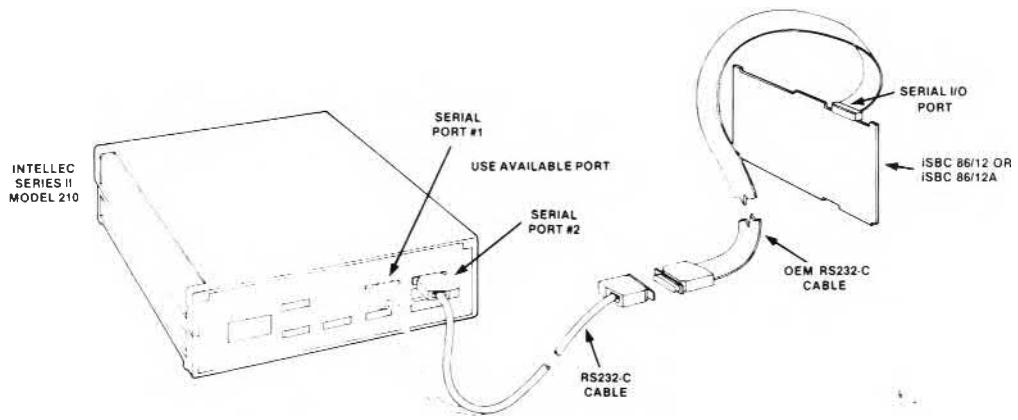


Figure 2a. Intellec Series II Model 210

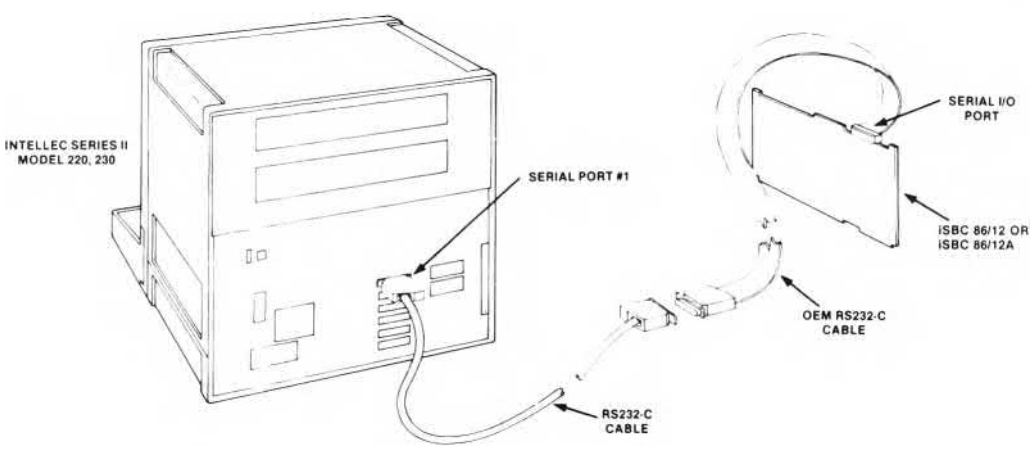


Figure 2b. Intellec Series Models 220, 230

terconnecting the iSBC 86/12A serial port with an available serial port on the Intellec system. All communication including command and data transfer occurs over this serial line. Development systems based on the Intellec Model 210 can use either one of the two available serial ports. (See Figure 2a.) On Models 220 and 230, Serial Port 1 is specified. (See Figure 2b.)

Intellec Environment

An Intellec Microcomputer Development System to be used in conjunction with the iSBC 957 package and an iSBC 86/12A must have the following necessary functionality to support program development and storage:

1. Intellec Development System with 64K bytes of RAM.
2. Console CRT or TTY terminal.
3. Intellec MDS-DDS Dual Double Density Diskette Drive and ISIS-II Operating System or Intellec MDS-2DS Dual Single Density Diskette Drive and ISIS-II Operating System.
4. User-selected language translators.

Note: The Intellec Series II Model 230 Microcomputer Development System and the Intellec MDS-888 Microcomputer Development Center contain all necessary hardware and operating system software to be used with the iSBC 957 package and the iSBC 86/12A.

Execution Environment

A full capability iSBC 86/12A execution environment should include the following components for effective utilization:

1. An iSBC 86/12A Single Board Computer.
2. An iSBC 957 Intellec-iSBC 86/12A Interface and Execution Package.
3. An iSBC 655 or iSBC 660 System Chassis for power and MULTIBUS expansion.
4. One or more iSBC 032, 048, or 064 RAM boards for programs requiring more than 32K bytes of RAM.

Note: The iSBC 86/12A cannot be mounted in the Intellec system and requires a separate operating environment.

Additional memory boards, analog and digital I/O boards, and peripheral controllers can be included in the iSBC 660 System Chassis with the iSBC 86/12A to allow the execution environment to be equivalent to the expected final product configuration.

SPECIFICATIONS

Hardware

Cables

(1) OEM RS232-C cable — Mates with serial I/O port on iSBC 86/12A

(1) RS232-C port cable — Mates with RS232-C port on Intellec system

(1) TTY port cable — Mates with TTY port on Intellec system

(1) Parallel load cable — Mates with UPP port on Intellec system and parallel I/O port on iSBC 86/12A (only used on Intellec MDS-800 series systems)

All cables allow separation of Intellec system and iSBC 86/12A of up to 6 feet.

I/O Drivers and Terminators

(1) 7437 48 mA open collector drivers

(4) iSBC 901 220Ω/330Ω terminator packs

(4) iSBC 902 1 kΩ terminator packs

Drivers and terminators needed when parallel load cable is required

Interface Adapters

(1) iSBC 530 TTY adapter — Used when serial I/O line connects with TTY port on Intellec system

(1) Parallel port status adapter — Mounts on iSBC 86/12A when parallel load cable is required

Miscellaneous — Attachment screws for Intellec mounted connectors

Software

(4) EPROMs with iSBC 86/12A system monitor

(1) Single density floppy diskette with iSBC 86/12A ISIS-II communication software

(1) Double density floppy diskette with iSBC 86/12A ISIS-II communication software

System Monitor

Addresses: RAM: $00000-00180_H$; ROM: $FE000-FFFF_H$

Commands

Basic Commands		
N	(Next)	Single stepped program execution
G	(Go)	Program start with optional breakpoints
S	(Substitute)	Examine and modify memory
X	(Examine)	Examine and modify registers
D	(Display)	Display blocks of memory
M	(Move)	Moves (duplicates) blocks of memory
C	(Compare)	Compare two blocks of memory
F	(Find)	Searches for byte/word value
H	(Hex Arithmetic)	Performs hexadecimal add and subtract
I	(Port Input)	Reads an I/O port
O	(Port Output)	Writes to an I/O port
R	(Read Tape)	Reads and loads paper tape object file
W	(Write Tape)	Writes memory block to paper tape
Intellec Mode Commands		
L	(Load File)	Loads ISIS-II file to iSBC 86/12A
T	(Transfer File)	Writes memory block to ISIS-II file
E	(Exit)	Return to ISIS (Basic Command Mode)

Transfer Rates

Intellec MDS-800 Family

Serial transfer: 110 baud

Parallel transfer: 1K bytes/sec

Intellec Series II Family

Serial transfer: Determined by system console (up to 9600 baud)

Reference Manuals

9800645 — iSBC 86/12 Hardware Reference Manual

9803074-01 — iSBC 86/12A Hardware Reference Manual

9800743 — iSBC 957 Intellec-iSBC 86/12 Interface and Execution Package User's Guide

9800640 — 8086 Assembly Language Manual

ORDERING INFORMATION

Part Number	Description
SBC 957	Intellec-iSBC 86/12A Interfacing and Execution Package

intel

iSBC 300 32K-BYTE RAM EXPANSION MODULE
iSBC 340 16K-BYTE EPROM/ROM EXPANSION MODULE

On-board memory expansion for iSBC 86/12A Single Board Computer

iSBC 300 module provides 32K bytes of dual port dynamic RAM and plugs directly into the iSBC 86/12A board

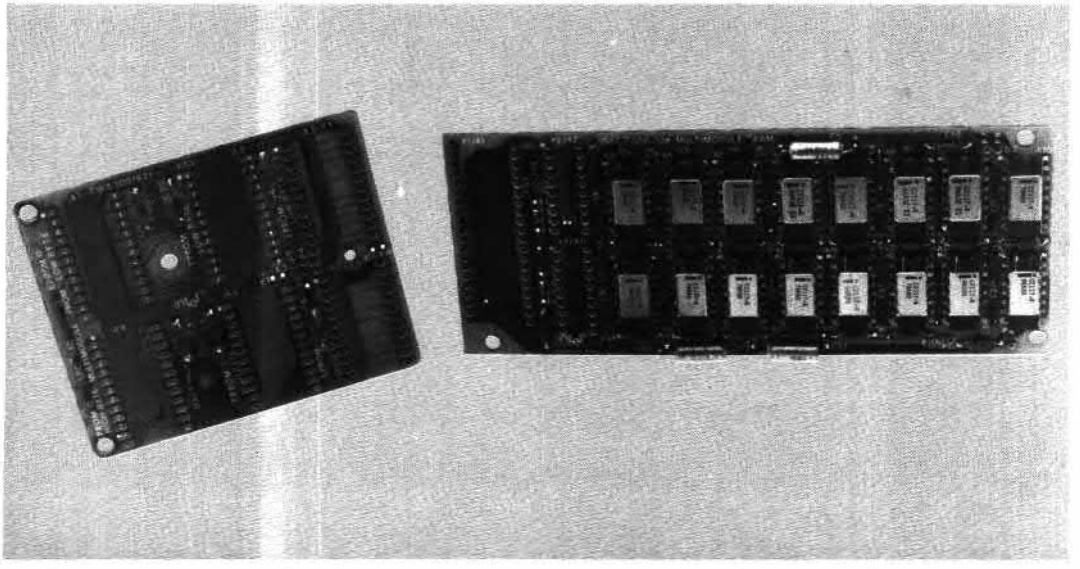
iSBC 340 module provides sockets for up to 16K bytes of additional EPROM/ROM and plugs directly into the iSBC 86/12A board

On-board memory expansion eliminates MULTIBUS system bus latency and increases system throughput

Low power requirements

Simple, reliable mechanical and electrical interconnection

The iSBC 300 32K-byte RAM expansion module and the iSBC 340 16K-byte EPROM/ROM expansion module provide simple, low cost expansion of the memory complement available on the iSBC 86/12A single board computer. Each module utilized individually or together can double the iSBC 86/12A board's on-board RAM and EPROM memory capacity. The iSBC 300 32K-byte RAM expansion module and the iSBC 340 16K-byte EPROM/ROM expansion module options for the iSBC 86/12A board offer system designers a new level of flexibility in defining and implementing Intel® single board computer systems. These options allow the systems designer to double the memory complement of an iSBC 86/12A board with a minimum of system implications. Because they expand the memory configuration on-board, they can be accessed as quickly as the existing iSBC 86/12A memory by eliminating the need for accessing the additional memory via the MULTIBUS system bus. With the iSBC 86/12A board mounted in the top slot of an iSBC 604 or iSBC 614 cardcage, sufficient clearance exists for mounting both the iSBC 300 and/or the iSBC 340 expansion module option(s). If the iSBC 86/12A board is inserted into some other slot, the combination of boards will physically (but not electrically) occupy two cardcage slots. Incremental power required by the options is minimal; for instance, only 305 mW is needed for the iSBC 300 RAM expansion module.



FUNCTIONAL DESCRIPTION

iSBC 300 32K-Byte Expansion Module

The iSBC 300 board measures 7.75" by 2.35" and mounts above the RAM area on the iSBC 86/12A single board computer. It expands the iSBC 86/12A board's on-board dual port RAM capacity from 32K bytes to 64K bytes. The iSBC 300 module contains sixteen 16K-byte dynamic RAM devices, sockets for the Intel® 8202 Dynamic RAM Controller and memory interface latching. To install the iSBC 300 module, the latches and controller from the iSBC 86/12A board are removed and inserted into the sockets on the iSBC 300 module. The add-on board is then mounted onto the iSBC 86/12A board. Pins extending from the controller's and latches' sockets mate with the devices' sockets underneath (see Figure 1). Additional pins mate to supply power and other signals to complete the electrical interface. The module is then secured at three additional points with nylon hardware to insure the mechanical security of the assembly.

To complete the installation, two socketed PROMs are replaced on the iSBC 86/12A board with those supplied with the iSBC 300 kit. These are the on-board memory and MULTIBUS address decode PROMs which allow the iSBC 86/12A board logic to recognize its expanded on-board memory complement.

iSBC 340 16K-Byte Expansion Module

The iSBC 340 module expands the iSBC 86/12A Single Board Computer's on-board EPROM capacity from 16K bytes to 32K bytes. It measures 3.3" by 2.8" and consists of a PC board with six 24-pin special sockets. Two of the sockets have extended pins which mate with two of the EPROM sockets on the iSBC 86/12A board. Two of the EPROMs which would have been inserted on the iSBC 86/12A board are then reinserted in the iSBC 340 module. Additional pins also mate for bringing chip selects for the remaining EPROM devices (see Figure 2). The mechanical interface is similar to that used on the iSBC 300 RAM module and consists of two additional mounting holes and the necessary mounting hardware.

The iSBC 340 module supports Intel® 2732 EPROM or 2332A ROMs as supplied by Intel. One section of the iSBC 86/12A on-board memory and MULTIBUS address decode PROMs (the same decode PROMs mentioned for the iSBC 300 module) is already preprogrammed to support the iSBC 340 module with Intel® 2732 EPROMs. This section is selected through the EPROM configuration switches on the iSBC 86/12A board. The iSBC 340 board can optionally be configured by the user to support Intel® 2758 or 2716 EPROMs or 2316E ROMs by programming new iSBC 86/12A decode PROMs to support these devices. Necessary documentation and PROM map listings are in the iSBC 86/12A Harware Reference Manual (order number 9803074-01).

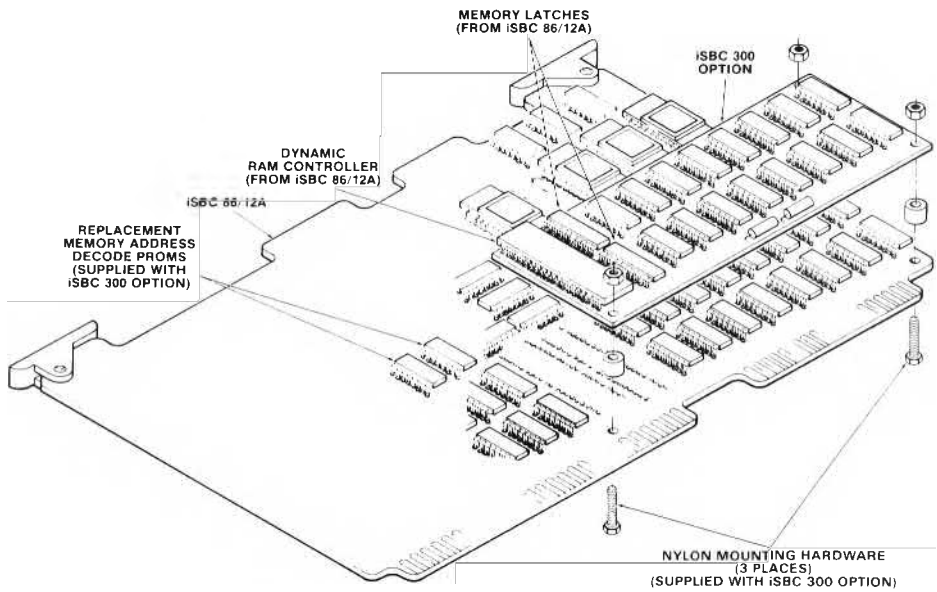


Figure 1. Installation of iSBC 300 RAM Expansion Module on iSBC 86/12A Single Board Computer

SPECIFICATIONS

Word Size
8 or 16 bits (16-bit data paths)

Memory Size
iSBC 300 Module — 32,768 bytes of RAM
iSBC 340 Module — 16,384 bytes (max) of EPROM/ROM

Access Time
iSBC 300 Module — Read: 1 μsec, write: 1.2 μsec
iSBC 340 Module — Standard EPROMs (450 nsec): 1 μsec, fast EPROMs (350 or 390 nsec): 800 nsec

Interface
The interface for the iSBC 300 and iSBC 340 module options is designed only for Intel's iSBC 86/12A Single Board Computer.

Memory Addressing
On-board RAM

CPU Access

iSBC 86/12A board only (32K bytes) — 00000-07FFFH.
iSBC 86/12A board + iSBC 300 module (64K bytes) — 00000-0FFFFH.
MULTIBUS Access — Jumper selectable for any 8K-byte boundary, but not crossing a 128K-byte boundary.

On-board EPROM/ROM

iSBC 86/12A board only (16K-bytes max.) — FF000-FFFFFH (using 2758 EPROMs); FE000-FFFFFH (using 2316E ROMs or 2716 EPROMs); and FC000-FFFFFH (using 2332A ROMs or 2732 EPROMs).

iSBC 86/12A board + iSBC 340 module (32K-bytes max.) — FE000-FFFFFH (using 2758 EPROMs); FC000-FFFFFH (using 2316E ROMs or 2716 EPROMs); F8000-FFFFFH (using 2332A ROMs or 2732 EPROMs).

On-board EPROM/ROM is not accessible via the MULTIBUS interface.

Auxiliary Power/Memory Protection
The low power memory protection option included on the iSBC 86/12A boards supports the iSBC 300 RAM module.

"Local Only" Memory Protection
The iSBC 86/12A Single Board Computer supports dedication of on-board RAM for on-board CPU access only in 8K, 16K, 24K, or 32K-byte segements. Installation of the iSBC 300 option allows protection of 16K, 32K, 48K, or 64K-byte segments.

Physical Characteristics

	iSBC 300	iSBC 340
Width	5.75"	3.3"
Length	2.35"	2.8"
Height of iSBC 86/12A plus mounted option	.718	.718*
Weight	13 oz.	5 oz.

*Includes EPROM/ROM's

All necessary mounting hardware (nylon, screws, spacers, nuts) are supplied with each kit.

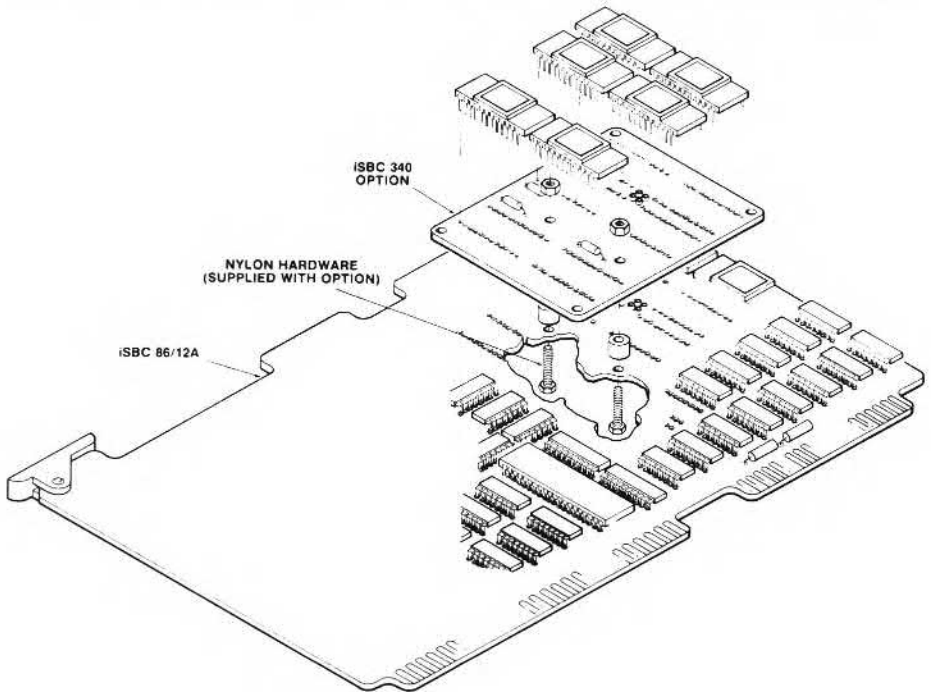


Figure 2. Installation of iSBC 340 EPROM/ROM Option on iSBC 86/12A Single Board Computer

Electrical Characteristics

DC power requirements:

Voltage	iSBC 300	iSBC 340
+5 ±5%	1 mA	120 mA[1]
+12 ±5%	24 mA	—
-12 ±5%	1 mA	—

Note:
1. Loaded with Intel 2732 EPROMs

Environmental Characteristics

Operating Temperature — 0° to +55°C
Relative Humidity — to 90% (without condensation)

Reference Manuals

All necessary documentation for the iSBC 300 module and iSBC 340 module is included in the iSBC 86/12A Hardware Reference Manual; order #9803074-01. (NOT SUPPLIED)

Manuals may be ordered from any Intel sales representative distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, CA 95051

ORDERING INFORMATION

Part Number	Description
SBC 300	32K-byte RAM Expansion Module
SBC 340	16K-byte EPROM Expansion Module

intel®

SDK-86
MCS-86 SYSTEM DESIGN KIT

Complete single board microcomputer system including CPU, memory, and I/O

Easy to assemble kit form

High performance 8086 16-bit CPU

Interfaces directly with TTY or CRT

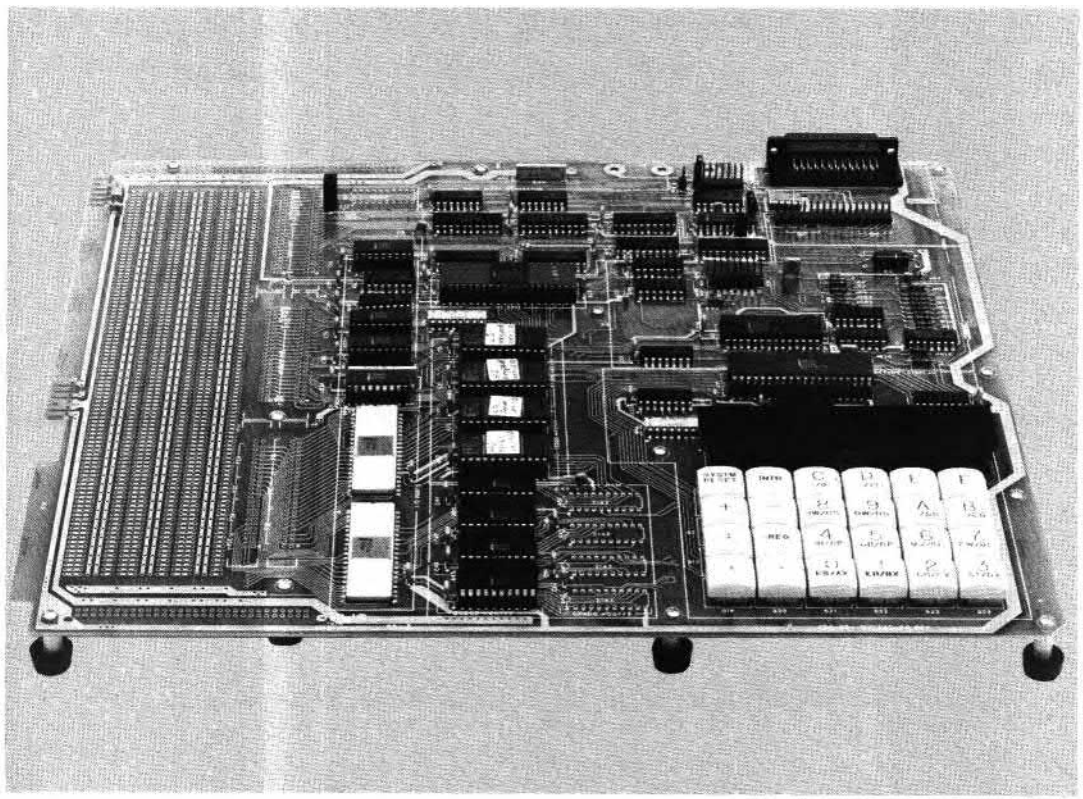
Interactive LED display and keyboard

Wire wrap area for custom interfaces

Extensive system monitor software in ROM

Comprehensive design library included

The SDK-86 MCS-86 System Design Kit is a complete single board 8086 microcomputer system in kit form. It contains all necessary components to complete construction of the kit, including LED display, keyboard, resistors, caps, crystal, and miscellaneous hardware. Included are preprogrammed ROMs containing a system monitor for general software utilities and system diagnostics. The complete kit includes an 8-digit LED display and a mnemonic 24-key keyboard for direct insertion, examination, and execution of a user's program. In addition, it can be directly interfaced with a teletype terminal, CRT terminal, or the serial port of an Intellec system. The SDK-86 is a high performance proto type system with designed-in flexibility for simple interface to the user's application.



FUNCTIONAL DESCRIPTION

The SDK-86 is a complete MCS-86 microcomputer system on a single board, in kit form. It contains all necessary components to build a useful, functional system. Such items as resistors, caps, and sockets are included. Assembly time varies from 4 to 10 hours, depending on the skill of the user. The SDK-86 functional block diagram is shown in Figure 1.

8086 Processor

The SDK-86 is designed around Intel's 8086 microprocessor. The Intel 8086 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor features attributes of both 8-bit and 16-bit microprocessors in that it addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance. Additional features of the 8086 include the following:

- Direct addressing capability to one megabyte of memory
- Assembly language compatibility with 8080/8085
- 14 word x 16-bit register set with symmetrical operations
- 24 operand addressing modes
- Bit, byte, word, and block operations
- 8 and 16-byte signed and unsigned arithmetic in binary or decimal mode, including multiply and divide
- 5 MHz clock rate
- MULTIBUS compatible system interface

A block diagram of the 8086 microprocessor is shown in Figure 2.

System Monitor

A compact but powerful system monitor is supplied with the SDK-86 to provide general software utilities and system diagnostics. It comes in preprogrammed read only memories (ROMs).

Communications Interface

The SDK-86 communicates with the outside world through either the on-board light emitting diode (LED) display/keyboard combination or the user's TTY or CRT terminal (jumper selectable), or by means of a special mode in which an Intellec development system transports finished programs to and from the SDK-86. Memory may be easily expanded by simply soldering in additional devices in locations provided for this purpose. A large area of the board (22 square inches) is laid out as general purpose wire-wrap for the user's custom interfaces.

Assembly

Only a few simple tools are required for assembly: soldering iron, cutters, screwdriver, etc. The SDK-86 assembly manual contains step-by-step instructions for easy assembly with a minimum of mistakes. Once construction is complete, the user connects his kit to a power supply and the SDK-86 is ready to go. The monitor starts immediately upon power-on or reset.

Commands — Keyboard mode commands, serial port commands, and Intellec slave mode commands are summarized in Table 1, Table 2, and Table 3, respectively. The SDK-86 keyboard is shown in Figure 3.

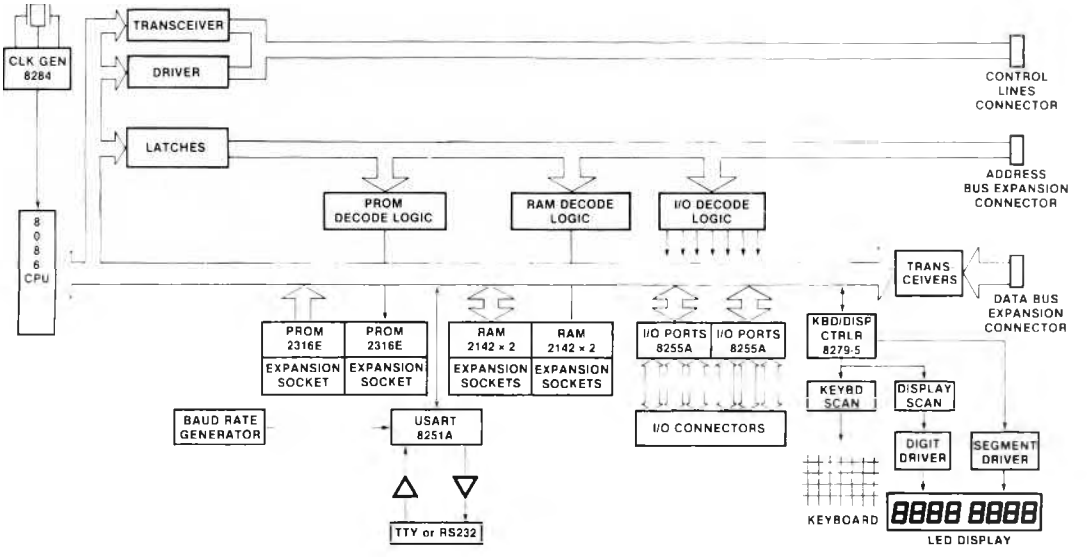


Figure 1. SDK-86 System Design Kit Functional Block Diagram

Documentation

In addition to detailed information on using the monitors, the SDK-86 user's manual provides circuit diagrams, a monitor listing, and a description of how the system works. The complete design library for the SDK-86 is shown in Figure 4 and listed in the specifications section under Reference Manuals.

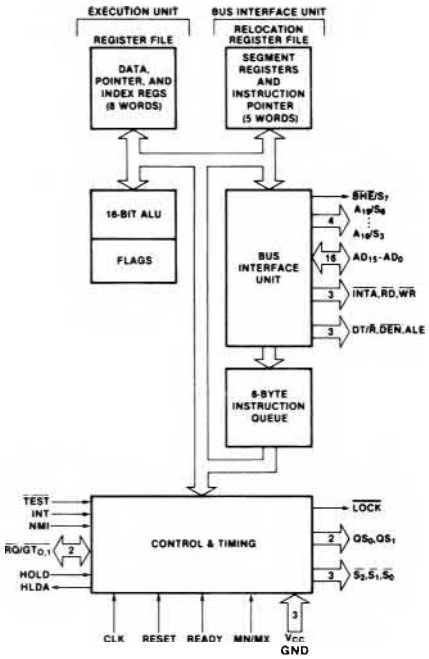


Figure 2. 8086 Microprocessor Block Diagram

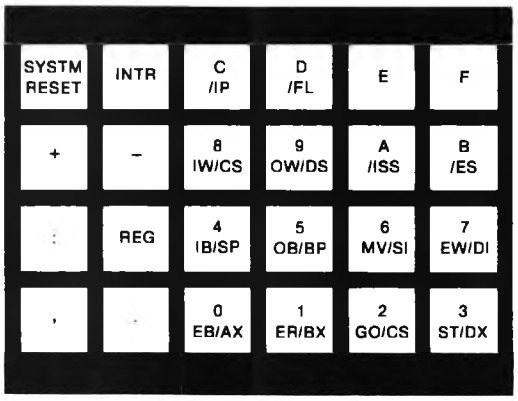


Figure 3. SDK-86 Keyboard

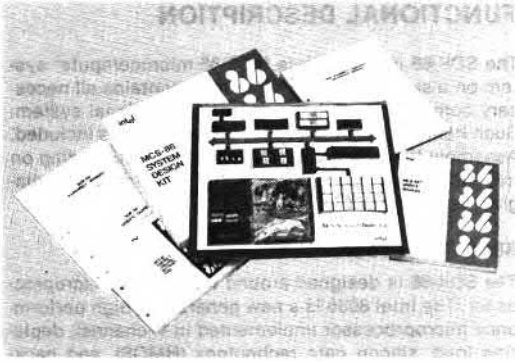


Figure 4. SDK-86 Design Library

Command	Operation
Reset	Starts monitor.
Go	Allows user to execute user program, and causes it to halt at predetermined program stop. Useful for debugging.
Single step	Allows user to execute user program one instruction at a time. Useful for debugging.
Substitute memory	Allows user to examine and modify memory locations in byte or word mode.
Examine register	Allows user to examine and modify 8086 register contents.
Block move	Allows user to relocate program and data portions in memory.
Input or output	Allows direct control of SDK-86 I/O facilities in byte or mode.

Table 1. Keyboard Mode Commands

Command	Operation
Dump memory	Allows user to print or display large blocks of memory information in hex format than amount visible on terminal's CRT display.
Start/continue display	Allows user to display blocks of memory information larger than amount visible on terminal's CRT display.
Punch/read paper tape	Allows user to transmit finished programs into and out of SDK-86 via TTY paper tape punch.

Table 2. Serial Mode Commands

8086 INSTRUCTION SET

Table 4 contains a summary of processor instructions used for the 8086 microprocessor.

Mnemonic and Description	Instruction Code			
Data Transfer				
MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w=1
Immediate to register	1 0 1 1 w reg	data	data if w=1	
Memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push:				
Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			
POP = Pop:				
Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			
XCHG = Exchange:				
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with accumulator	1 0 0 1 0 reg			
IN = Input:				
Fixed port	1 1 1 0 0 1 0 w	port		
Variable port	1 1 1 0 1 1 0 w			
OUT = Output:				
Fixed port	1 1 1 0 0 1 1 w	port		
Variable port	1 1 1 0 1 1 1 w			
XLAT=Translate byte to AL	1 1 0 1 0 1 1 1			
LEA=Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS=Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES=Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF=Load AH with flags	1 0 0 1 1 1 1 1			
SAHF=Store AH into flags	1 0 0 1 1 1 1 0			
PUSHF=Push flags	1 0 0 1 1 1 0 0			
POPF=Pop flags	1 0 0 1 1 1 0 1			
Arithmetic				
ADD = Add:				
Reg./memory with register to either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s:w=01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w=1	
ADC = Add with carry:				
Reg./memory with register to either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s:w=01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w=1	
INC = Increment:				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA=ASCII adjust for add	0 0 1 1 0 1 1 1			
DAA=Decimal adjust for add	0 0 1 0 0 1 1 1			
SUB = Subtract:				
Reg./memory and register to either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	25 data	data if s:w=01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w=1	
SBB = Subtract with borrow				
Reg./memory and register to either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1	
DEC = Decrement:				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG=Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		

Mnemonic and Description	Instruction Code			
CMP = Compare:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s:w=01
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w=1	
AAS=ASCII adjust for subtract	0 0 1 1 1 1 1 1			
DAS=Decimal adjust for subtract	0 0 1 0 1 1 1 1			
MUL=Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL=Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM=ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV=Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV=Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD=ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW=Convert byte to word	1 0 0 1 1 0 0 0			
CWD=Convert word to double word	1 0 0 1 1 0 0 1			
Logic				
NOT=Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL=Shift logical/arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR=Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR=Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL=Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR=Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL=Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR=Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND = And:				
Reg./memory and register to either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w=1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w=1	
TEST = And function to flags, no result:				
Register/memory and register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w=1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data	data if w=1	
OR = Or:				
Reg./memory and register to either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w=1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w=1	
XOR = Exclusive or:				
Reg./memory and register to either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w=1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w=1	
String Manipulation				
REP=Repeat	1 1 1 1 0 0 1 z			
MOVS = Move byte/word	1 0 1 0 0 1 0 w			
CMPS = Compare byte/word	1 0 1 0 0 1 1 w			
SCAS = Scan byte/word	1 0 1 0 1 1 1 w			
LODS = Load byte/wd to AL/AX	1 0 1 0 1 1 0 w			
STOS = Stor byte/wd frm AL/A	1 0 1 0 1 0 1 w			
Control Transfer				
CALL = Call:				
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high	
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m		
Direct intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high	
		seg-low	seg-high	
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m		

continued

Mnemonic and Description	Instruction Code		
JMP - Unconditional Jump:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within segment-short	1 1 1 0 1 0 1 1	disp	
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	
RET - Return from CALL:			
Within segment	1 1 0 0 0 0 1 1		
Within seg adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ-Jump on equal/zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE-Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG-Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE-Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA-Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp	
JP/JPE-Jump on parity/parity even	0 1 1 1 1 0 1 0	disp	
JO-Jump on overflow	0 1 1 1 0 0 0 0	disp	
JS-Jump on sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ-Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE-Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG-Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp	
JNB/JAE-Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp	
JNBE/JA-Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp	
JNP/JPO-Jump on not par/par odd	0 1 1 1 1 0 1 1	disp	
JNO-Jump on not overflow	0 1 1 1 0 0 0 1	disp	

Mnemonic and Description	Instruction Code	
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNS Jump on not sign	0 1 1 1 1 0 0 1	disp
LOOP Loop CX times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE Loop while zero/equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE Loop while not zero/equal	1 1 1 0 0 0 0 0	disp
JCXZ Jump on CX zero	1 1 1 0 0 0 1 1	disp
INT - Interrupt		
Type specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO-Interrupt on overflow	1 1 0 0 1 1 1 0	
IRET Interrupt return	1 1 0 0 1 1 1 1	
Processor Control		
CLC Clear carry	1 1 1 1 1 0 0 0	
CMC Complement carry	1 1 1 1 0 1 0 1	
STC Set carry	1 1 1 1 1 0 0 1	
CLD Clear direction	1 1 1 1 1 1 0 0	
STD Set direction	1 1 1 1 1 1 0 1	
CLI Clear interrupt	1 1 1 1 1 0 1 0	
STI Set interrupt	1 1 1 1 1 0 1 1	
HLT Halt	1 1 1 1 0 1 0 0	
WAIT Wait	1 0 0 1 1 0 1 1	
ESC Escape (to external device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK Bus lock prefix	1 1 1 1 0 0 0 0	

Notes

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low

if s:w = 01 then 16 bits of immediate data form the operand
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1, if v = 0 then "count" in (CL)
x = don't care
if v = 0 then "count" = 1, if v = 1 then "count" in (CL) register
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

Mnemonics © Intel, 1978

Table 4. 8086 Instruction Set Summary

SPECIFICATIONS

Central Processor

CPU — 8086-4

Note

May be operated at 2.5 MHz or 5 MHz, jumper selectable, for use with 8086.

Memory

ROM — 8K bytes 2316/2716

RAM — 2K bytes (expandable to 4K bytes) 2142

Addressing

ROM — FE000-FFFFF

RAM — 0-7FF (800-FFF available with additional 2142's)

Note

The wire-wrap area of the SDK-86 PC board may be used for additional custom memory expansion.

Input/Output

Parallel — 48 lines (two 8255A's)

Serial — RS232 or current loop (8251A)

Baud Rate — selectable from 110 to 4800 baud

Interfaces

Bus — All signals TTL compatible

Parallel I/O — All signals TTL compatible

Serial I/O — 20 mA current loop TTY or RS232

Note
The user has access to all bus signals which enable him to design custom system expansions into the kit's wire-wrap area.

Interrupts (256 vectored)

Maskable
Non-maskable
TRAP

DMA

Hold Request — Jumper selectable. TTL compatible input.

Software

System Monitor — Preprogrammed 2716 or 2316 ROMs

Addresses — FE000–FFFFF

Monitor I/O — Keyboard/display or TTY or CRT (serial I/O)

Physical Characteristics

Width — 13.5 in. (34.3 cm)

Height — 12 in. (30.5 cm)

Depth — 1.75 in. (4.45 cm)

Weight — approx. 24 oz. (3.3 kg)

Electrical Characteristics

DC Power Requirement
(Power supply not included in kit)

Voltage	Current
V_{CC} 5V ± 5%	3.5A
V_{TTY} – 12V ± 10%	0.3A
	(V_{TTY} required only if teletype is connected)

Environmental Characteristics

Operating Temperature — 0–50°C

Reference Manuals

9800697A — SDK-86 MCS-86 System Design Kit Assembly Manual

9800722 — MCS-86 User's Manual

9800640A — 8086 Assembly Language Programming Manual

8086 Assembly Language Reference Card

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

ORDERING INFORMATION

Part Number	Description
SDK-86	MCS-86 system design kit

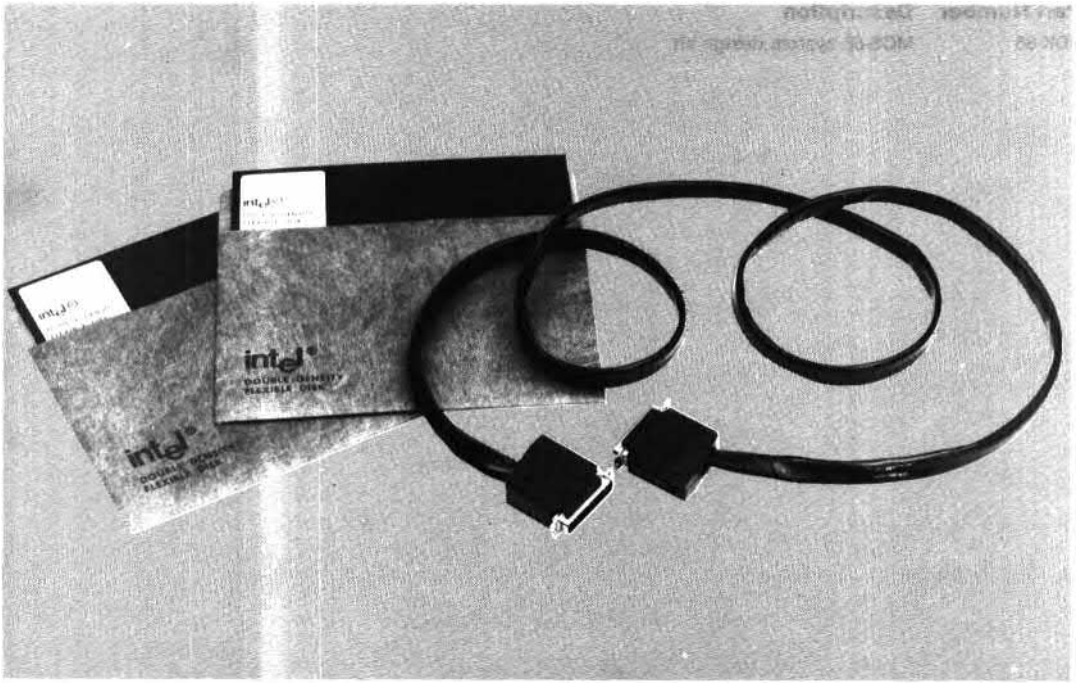
intel®

SDK-C86

MCS-86™ SYSTEM DESIGN KIT SOFTWARE AND CABLE INTERFACE TO INTELLEC® DEVELOPMENT SYSTEM

- **Provides the Software and Hardware Communications Link Between an Intellec® Development System and the SDK-86**
- **Intellec® System Files can be Accessed and Down loaded to the SDK-86 Resident Memory**
- **Data in SDK-86 Memory can be Uploaded and Saved in Intellec® System Files**
- **Enhances and Extends the Power and Usefulness of the SDK-86**
- **Allows the SDK-86 to Become an Execution Vehicle for ISIS-II Developed 8086 Object Code Using the MDS-311 Software Cross Development Package**
- **All SDK-86 Serial Port Mode Commands Become Available at Console of the Intellec® System**

The SDK-C86 product provides the software and hardware link for using the SDK-86 monitor in conjunction with an Intellec® Development System while adding features of data transfer between SDK-86 memory and Intellec® System files. The user may enter programs and data into the SDK-86 and then save them on a diskette. Also, programs and data may be created on the Intellec® System using the MDS-311 cross development software package, then loaded into the SDK-86 for testing and checkout. This provides a real time execution environment of the SDK-86 as a peripheral to the Intellec® System.



HARDWARE

There are two serial ports on the Intellec® System back panel, TTY and CRT. Assuming that one of the ports is used for the Intellec® console, the SDK-C86 cable can plug into the unused port. The SDK-86 is jumper selectable to accept either the CRT (RS232) or TTY (20mA current loop) signals.

The edge connector on the SDK-86 has the MULTIBUS™ form factor. No signals are connected to the fingers except the power supply traces. Therefore, the SDK-86 can plug directly into the Intellec® motherboard to obtain power while using the SDK-C86 cable as the communication link.

SOFTWARE

Two programs must be invoked to operate in the SDK-86 slave mode. One program runs on the SDK-86, and another runs in any ISIS-II environment that includes a diskette drive.

The serial I/O monitor is installed on the SDK-86 and operates as though it was talking to a terminal. The software in the Intellec® allows the Intellec®, with a console device, to behave as if it were a terminal to the SDK-86.

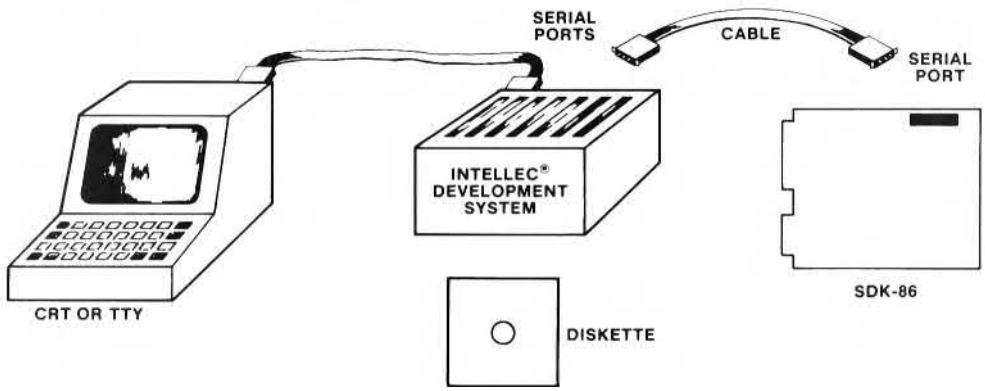
The SDK-C86 software program in the Intellec reads the console input device, then passes the character to the SDK-86 through the serial port. It also receives the characters from the SDK-86 and displays them at the console output device. Besides the basic transfer function, this program also recognizes and performs the Upload and Download functions.

COMMAND MODES

- Transparent: In this mode, the SDK-C86 software passes all characters through without any processing. All the commands of the SDK-86 monitor (except paper tape commands) are available and will function in exactly the same manner as if the terminal were attached directly to the serial port of the SDK-86.
- Upload/Download: In this mode the SDK-C86 software, in the Intellec®, recognizes the mnemonic for Upload or Download from the terminal. It "translates" the Download command to an R (Read hexadecimal tape) command and the Upload command to a W (Write hexadecimal tape). The R and W commands are then passed on to the SDK-86 monitor. Using these paper tape commands allows for a checksummed transfer of data between the Intellec® and the SDK-86 memory.

COMMAND SUMMARY

- Reset — starts the SDK-86 monitor.
- Execute with Breakpoint (G) — Allows you to execute a user program and cause it to halt at a predetermined program step — useful for debugging.
- Single Step (N) — allows you to execute a user program one instruction at a time — useful for debugging.
- Substitute Memory (S, SW) — allows you to examine and modify memory locations in byte or word mode.
- Examine Register (X) — allows you to examine and modify the 8086's register contents.
- Block Move (M) — allows you to relocate program and data portions in memory.
- Input or Output (I, IW, O, OW) — allows direct control of the SDK-86's I/O facilities in byte or word mode.
- Display Memory (D) — allows you to print or display large blocks of memory information in HEX format.
- Load (L) — allows you to load hex format object files into SDK-86 memory from an Intellec.
- Transfer (T) — allows you to save contents of SDK-86 memory in a hex format object file in the Intellec.



SDK-86/Intellec® Slave Mode Configuration