8089 ASSEMBLER USER'S GUIDE

Manual Order Number: 9800938-01

A97/0879/10K FL

PREFACE

This manual is intended for software engineers who are familiar with assembly language programming. The contents of this manual are meant to:

- Introduce the purpose, features and terminology of the Intel 8089 IOP (I/O Processor)
- Provide reference information on the syntax and semantics of the 8089 Assembly Language, including 8089 assembler controls
- Give examples of the use of the 8089 Assembly Language, including the 8089 assembler controls

The manual is organized as follows:

Chapter 1: An Overview of 8089 Operation and Programming

Description of IOP operation
Introduction to task block programs

Chapter 2: Operands

Description of the types and forms of 8089 Assembly Language operands

Chapter 3: The Instruction Set

Instruction set overview
Alphabetized description of each instruction (for quick reference)

Chapter 4: Assembler Directives

Description and examples of assembler directives

Chapter 5: Assembler Controls and Operation

Assembler invocation and controls

Chapter One presents basic information referred to throughout the manual. It should be read before attempting to write task block programs.

Each of the remaining chapters, Chapters Two through Five, deals with a single element of the 8089 Assembly Language or its assembler, ASM89. More experienced assembly language programmers may find the information in Appendices A, B, C, and D sufficient for their needs, referencing Chapters Two through Five when a more thorough explanation is needed. These chapters provide detailed descriptions and examples, meant to familiarize a programmer with writing 8089 task block programs in the 8089 Assembly Language.

Reference Publications

The following publications provide helpful reference information:

ISIS-II User's Guide, Order No. 9800306, for information on the ISIS-II operating facilities.

MCS-86 User's Manual, Order No. 9800722, for 8089 hardware information and design considerations.

MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, Order No. 9800639, for information on the linkage and relocation utilities LINK86 and LOC86.

MCS-86 Assembly Language Reference Manual, Order No. 9800640, for 8086 Assembly Language information.

MCS-86 Absolute Object File Formats, Order No. 9800821, for MCS-86 absolute object file formats.

CONTENTS

CONTENTS (CONT'D.)

ILLUSTRATIONS

TABLES

CHAPTER 1 AN OVERVIEW OF 8089 OPERATION AND PROGRAMMING

Introduction

This manual is about the 8089 Assembly Language. An 8089 programmer must be familiar with this symbolic language and the operation of the 8089—this chapter provides an introduction to both.

The 8089 I/O Processor

The 8089 brings the mainframe and minicomputer I/O channel to the micro-computer world. I/O operations, which previously required large amounts of CPU supervision and therefore limited its data processing time, can now be independently managed and maintained by the 8089. I/O channels, by relieving the burden of I/O processing from the CPU, significantly improve system throughput.

Figure 1-1 illustrates the advantage of using an I/O channel to handle I/O operations. At the request of the host processor, the I/O channel initializes an I/O device, starts the I/O operation, and checks for a successful completion. In the meantime,

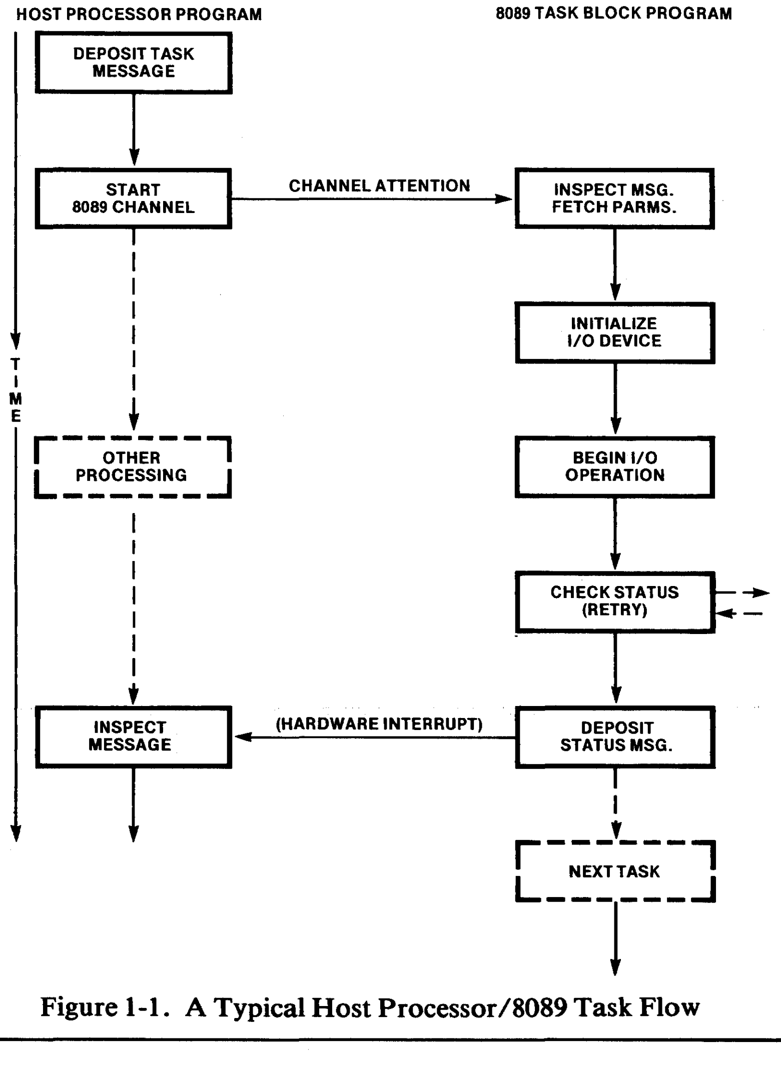


Figure 1-1. A Typical Host Processor/8089 Task Flow

the host processor is free to do other processing. If the operation does not complete successfully, the channel takes corrective action, signalling the host processor when the I/O operation is completed or error correction routines have finished executing.

8089 System Configurations

The 8089 may appear in two system configurations—LOCAL and REMOTE. In the LOCAL configuration, the 8089 shares the system bus with a host processor. In the REMOTE configuration, the 8089 shares the system bus with a host processor and also has its own remote bus, not accessible by the host processor.

Figure 1-2 shows a generalized LOCAL configuration. A common bus interface is shared by the two processors (see shaded area), whose use is controlled by means of the request/grant (RQ/GT) circuitry. The shared system bus can be an 8- or 16-bit bus. All the system's resources are accessible by both processors. The 8089 can address a megabyte of memory and 64k of I/O addresses.The width of the system bus and bus access control via the request/grant circuitry are established during 8089 initialization, discussed later in this chapter.

A generalized REMOTE configuration is shown in figure 1-3. The 8089 has its own remote bus, not accessible by the host processor (see shaded area). This remote bus may be an 8- or 16-bit bus—it need not be the same width as the system bus, e.g., the remote bus could be 8-bits and the system bus could be 16-bits. The 8089 also accesses a shared system bus by means of a MULTIBUS™ interface and an 8289 Bus Arbiter, which controls its access to the system bus. A 64k address space is available to the 8089 over its remote bus. One megabyte of address space is available to the

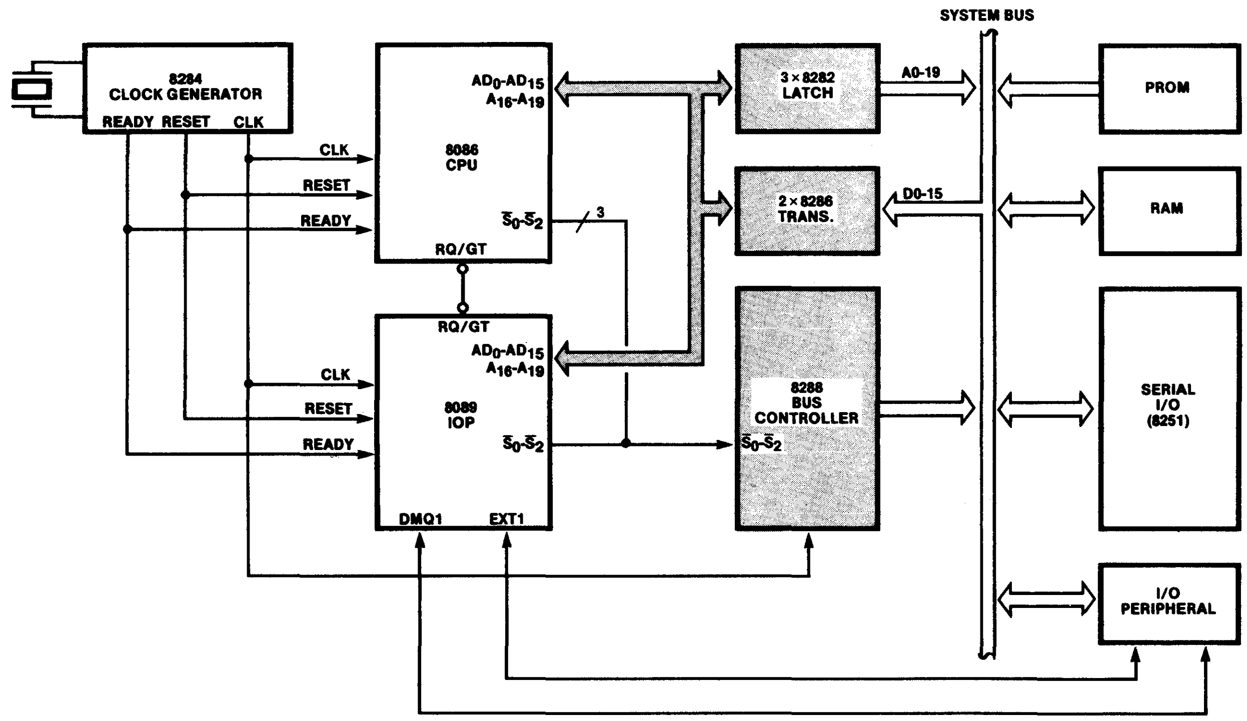


Figure 1-2. Generalized LOCAL Configuration

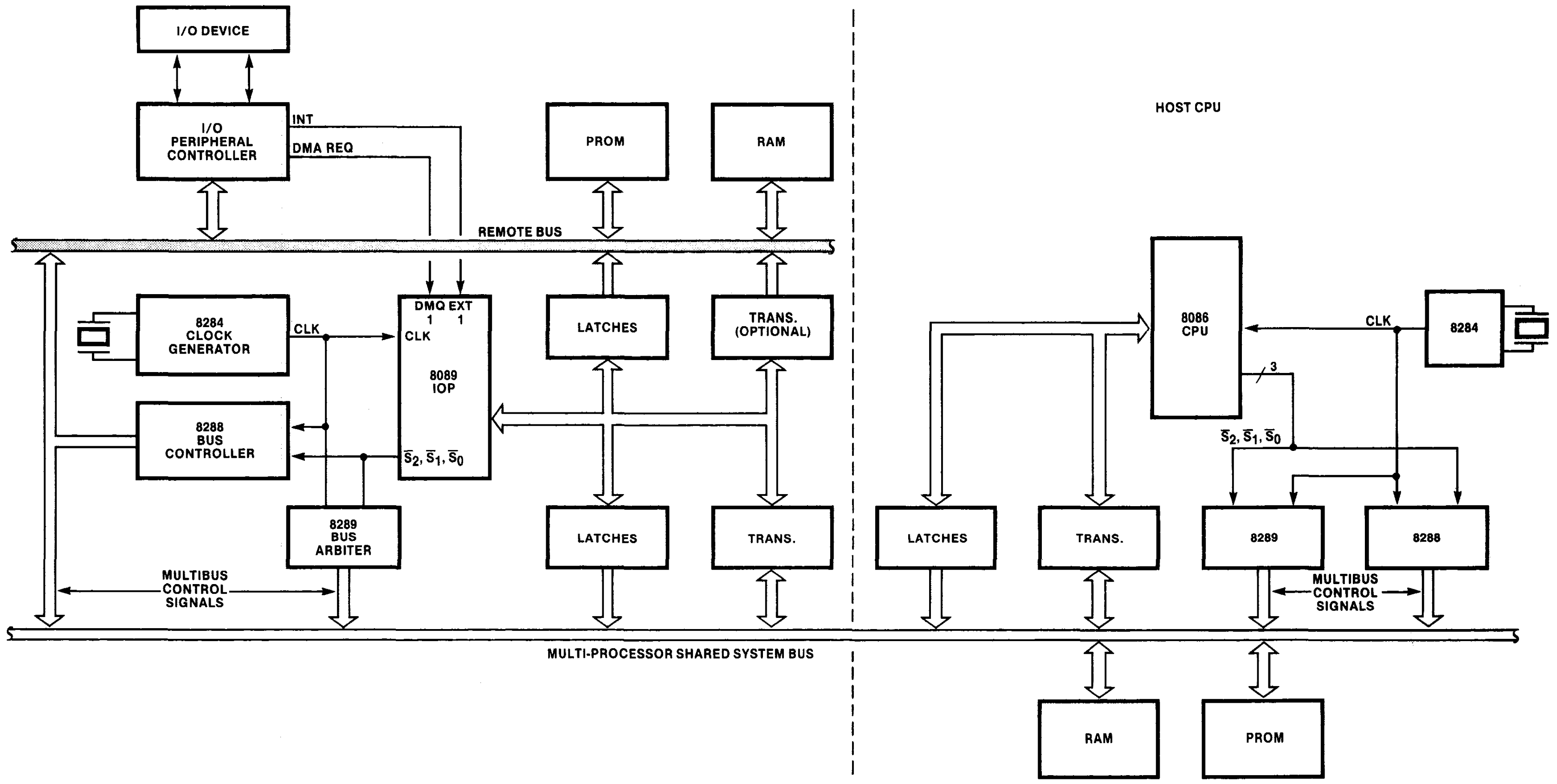


Figure 1-3. Generalized REMOTE Configuration

8089 over the system bus. The 8089 can use its remote bus without affecting the use of the system bus by other processors. If the the remote bus is shared with another processor, the request/grant circuitry may be used to control access to it. The size of the 8089's remote bus is specified during 8089 initialization.

In this manual, the addresses available to the 8089 over its remote bus (in the REMOTE configuration) are referred to as "local space addresses" or addresses in the 64k "local address space" (see figure 1-4A). In LOCAL configurations (the 8089 has no remote bus), this 64k address space is used for I/O addressing (see figure 1-4B). The term "local (I/O) address" in this manual refers to the 64k 8089 address space which can be either addresses on its remote bus (REMOTE configuration) or I/O addresses (LOCAL configuration).

The terms "system space address" or "system address space" refer to the 8089's one megabyte address space. In REMOTE configurations, this is the space addressed over the system bus, which is shared with other processors. In LOCAL configurations, these addresses are used to access memory. The term "system (memory) address" refers to the one megabyte IOP address space—system addresses in a REMOTE configuration (see figure 1-4A) and memory addresses in a LOCAL configuration (see figure 1-4B).

Task Block Programs

The 8089 has two independent I/O channels that operate concurrently. Each channel has a separate set of registers and individual external interrupt, DMA request and external terminate pins. Figure 1-5 shows, conceptually, the 8089's two I/O channels.

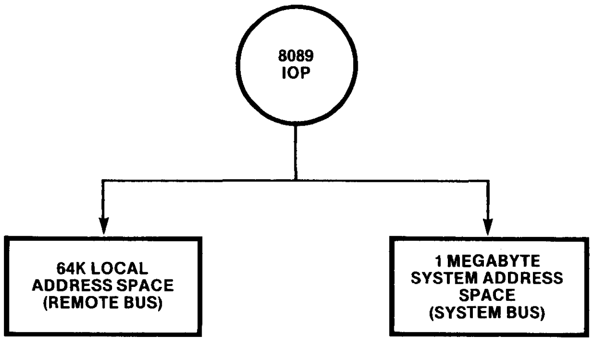

8089
IOP
64K LOCAL ADDRESS SPACE (REMOTE BUS)
1 MEGABYTE SYSTEM ADDRESS SPACE (SYSTEM BUS)

Figure 1-4A. 8089 REMOTE Configuration Address Space

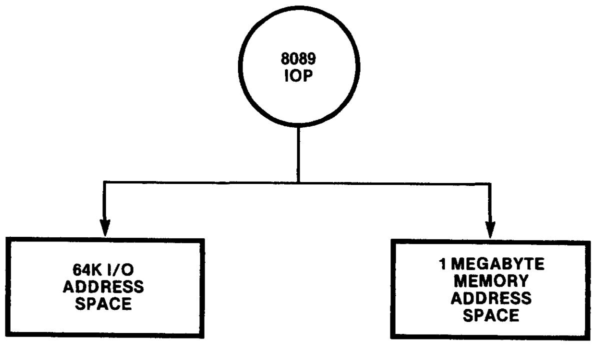


Figure 1-4B. 8089 LOCAL Configuration Address Space

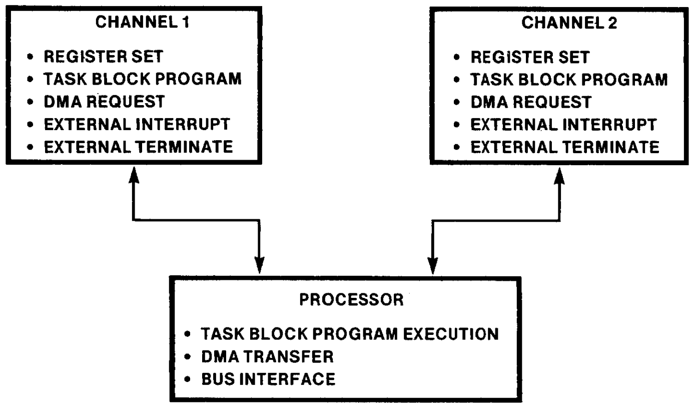


Figure 1-5. A Conceptual View of the 8089 I/O Processor

A task block program, written in 8089 Assembly Language, is executed for each channel. Task block programs manage and control the I/O operations performed by an I/O channel. The 8089 Assembly Language instruction set contains specialized I/O and general-purpose data processing instructions for simple and efficient control of I/O operatons:

- Bit manipulation and test instructions.
- Memory-to-memory, peripheral-to-memory, and peripheral-to-peripheral data transfer operations.
- Simple arithmetic and logical operation instructions.
- Conditional, unconditional, and bit test control transfer instructions.
- Special instructions for interrupt control, DMA initialization, and a semaphore test and set mechanism.

Task block programs vary in size and complexity, depending on I/O system design and the I/O operation being conducted. There is a great deal of flexibility in the use of task block programs to manage and control I/O operations. A modular technique may be employed, using a number of simple, well-defined task block programs, linked in sequence, to perform I/O operations.

The 8089 Assembly Language Assembler—ASM89

ASM89 is the assembler for the 8089 Assembly Language. Its output, shown in Figure 1-6, consists of two possible files:

- An object file containing the source file translated into object code.
- A list file showing the input source statements, the assembler-generated object code, error messages, and (optionally) a symbol table.

Note that the 8089 Assembly Language source file can contain numerous task block programs. The number of task block programs contained in a single 8089 Assembly Language source file is limited by the size of the segment defined in the source file, which cannot exceed 64k consecutive byte addresses.

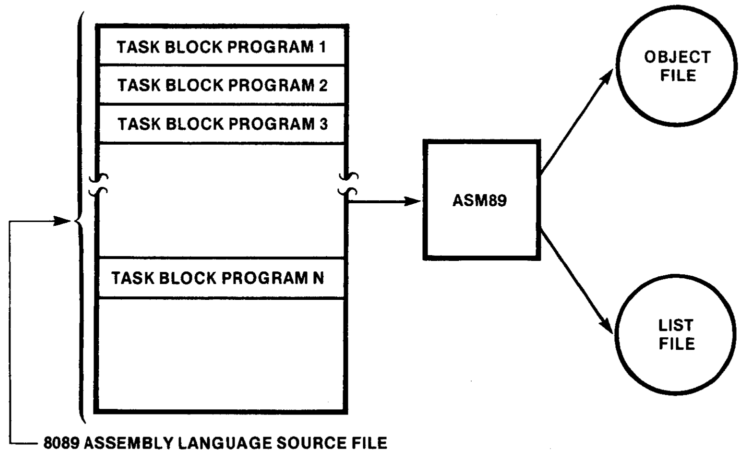


Figure 1-6. ASM89 Output Files

Object File

The assembly of an 8089 Assembly Language source file generates an object module, containing the object code generated by ASM89. A single, relocatable segment must be defined in each object module. This segment has a maximum size of 64k (65,536) consecutive bytes. LINK86 is used to resolve intermodule references; LOC86 is used to assign absolute addresses to the object module. (See *MCS-86 Software Development Utilities Operating Instructions for ISIS-II User's,* Order No. 9800639 for information on LINK86 and LOC86 operation.)

The relocatable segment defined in an ASM89 object module is paragraph aligned, i.e., when located it begins at an address which is divisible by sixteen (the last digit of the address, in hexadecimal, is a zero). This segment is not combinable. Unlike 8086 segments, the segment in an 8089 object module cannot be combined with other segments to form a single segment when linked and located.

List File

The list file provides a record of the source file, the assembler-generated object code, and the assembly process, including the assembler invocation command and error messages issued by the assembler. A symbol table, giving information on user-defined symbols in the source file, may also be included in the list file. (See "Format of Listing File" in Chapter 5 of this manual for more information.)

8089/Host Processor Communication

The 8089 and its host processor communicate through messages placed in blocks of shared memory. The host processor sets up these communication blocks and supplies their addresses to the 8089. There are two such blocks: the Channel Control Block and the Command Parameter Block.

The address of the Channel Control Block (CB) is supplied to the 8089 during system initialization (see "8089 Initialization" later in this chapter). The Channel Control Block contains two identical sets of pre-defined fields, one for each channel (figure 1-7). Each set of fields is composed of six bytes: a one-byte Channel Control Word (CCW) used to issue commands to the I/O channel; a one-byte channel BUSY flag, indicating the activity status of the channel; and two words used to supply the offset and segment address of the channel's Command Parameter Block.

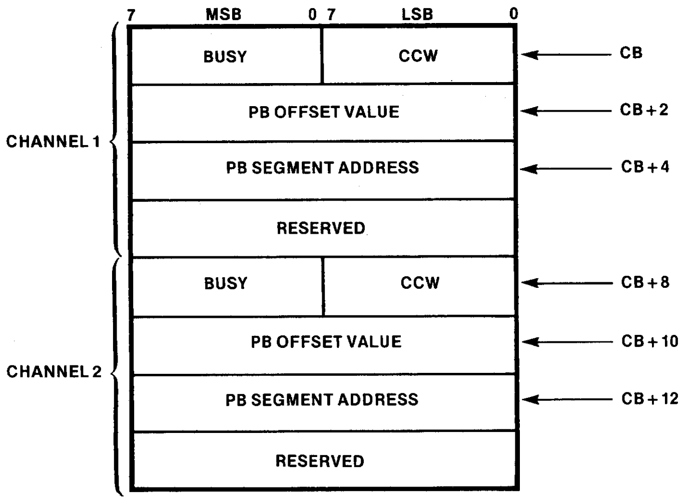


Figure 1-7. The Channel Control Block (CB)

The Channel Control Block is inspected by the appropriate channel, as specified by the SEL (select) input pin, whenever a channel attention (CA) is received by the 8089 (other than the first CA after a reset). Examination of the CCW by a channel is transparent to its operation.

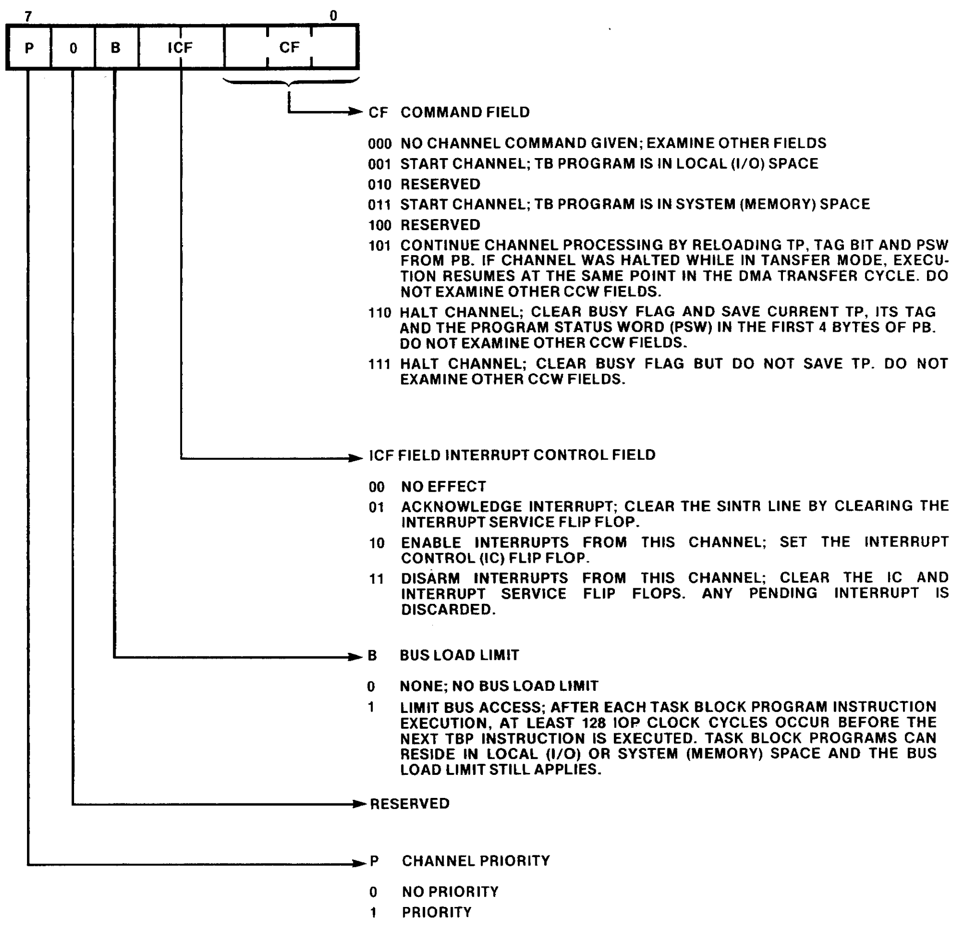
Figure 1-8 shows the CCW. It contains four fields, each controlling some aspect of the I/O channel's operation. The three bit Command Field (CF), bits 0-2, directs the channel's operation, optionally:

- starting task block program execution (from a task block program located in system (memory) or local (I/O) address space)
- suspending channel operation (task block program pointer and Program Status Word (PSW) saved)
- continuing channel operation (stored task block program pointer and PSW restored)
- halting channel operation (task block program pointer and PSW not saved)

The Interrupt Control Field (ICF) is used in conjunction with the task block program SINTR instruction to supply interrupts to the host processor's interrupt hardware. Each channel has its own interrupt pin, SINTR-1 and SINTR-2 respectively, to provide the hardware interrupt signal. The host processor enables, acknowledges, or disables interrupts from a channel through the ICF.

The Bus Load Limit field (B) limits task block program instruction execution for a channel to one instruction every 128 IOP clock cycles. This bus load limit field applies to task block programs residing in either system (memory) or local (I/O) space.

The Priority field (P) of the CCW is used to resolve conflicts that arise when both channels request service for operations of equal priority in the 8089's operation hierarchy. If the P field values are the same for both channels, service cycles alternate between them. If the two channels have different P field values, the channel with P = 1 is serviced first. (See "DMA Transfer" later in this chapter and also the *MCS-86 User's Guide* for more information on 8089 channel priorities.)



CF COMMAND FIELD

000	NO CHANNEL COMMAND GIVEN; EXAMINE OTHER FIELDS
001	START CHANNEL; TB PROGRAM IS IN LOCAL (I/O) SPACE
010	RESERVED
011	START CHANNEL; TB PROGRAM IS IN SYSTEM (MEMORY) SPACE
100	RESERVED
101	CONTINUE CHANNEL PROCESSING BY RELOADING TP, TAG BIT AND PSW FROM PB. IF CHANNEL WAS HALTED WHILE IN TANSFER MODE, EXECUTION RESUMES AT THE SAME POINT IN THE DMA TRANSFER CYCLE. DO NOT EXAMINE OTHER CCW FIELDS.
110	HALT CHANNEL; CLEAR BUSY FLAG AND SAVE CURRENT TP, ITS TAG AND THE PROGRAM STATUS WORD (PSW) IN THE FIRST 4 BYTES OF PB. DO NOT EXAMINE OTHER CCW FIELDS.
111	HALT CHANNEL; CLEAR BUSY FLAG BUT DO NOT SAVE TP. DO NOT EXAMINE OTHER CCW FIELDS.

ICF FIELD INTERRUPT CONTROL FIELD

00	NO EFFECT
01	ACKNOWLEDGE INTERRUPT; CLEAR THE SINTR LINE BY CLEARING THE INTERRUPT SERVICE FLIP FLOP.
10	ENABLE INTERRUPTS FROM THIS CHANNEL; SET THE INTERRUPT CONTROL (IC) FLIP FLOP.
11	DISARM INTERRUPTS FROM THIS CHANNEL; CLEAR THE IC AND INTERRUPT SERVICE FLIP FLOPS. ANY PENDING INTERRUPT IS DISCARDED.

B BUS LOAD LIMIT

0	NONE; NO BUS LOAD LIMIT
1	LIMIT BUS ACCESS; AFTER EACH TASK BLOCK PROGRAM INSTRUCTION EXECUTION, AT LEAST 128 IOP CLOCK CYCLES OCCUR BEFORE THE NEXT TBP INSTRUCTION IS EXECUTED. TASK BLOCK PROGRAMS CAN RESIDE IN LOCAL (I/O) OR SYSTEM (MEMORY) SPACE AND THE BUS LOAD LIMIT STILL APPLIES.

RESERVED

P CHANNEL PRIORITY

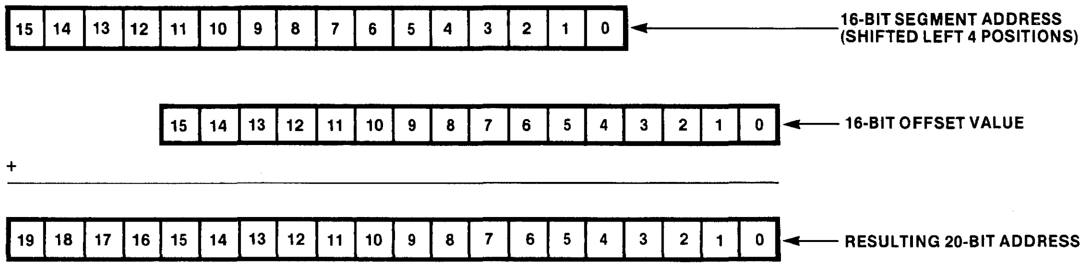
0	NO PRIORITY
1	PRIORITY

Figure 1-8. The Channel Control Word (CCW)

The channel BUSY flag byte indicates a channel's activity status. Following the first CA after reset, during 8089 initialization, "00" (hex) is written to channel one's BUSY flag byte by the 8089 hardware when initialization has been completed. On any subsequent CA, the 8089 hardware sets the BUSY flag byte to "FF" (hex) if the CCW starts or continues a channel; to "00" if the CCW halts or suspends a channel. The BUSY flag byte is also cleared to "00" by a task block program HLT instruction.

The four bytes following the CCW and BUSY flag byte contain the offset and segment address of a channel's Command Parameter Block (bytes 2—5 of the CB for channel 1; bytes 8—11 of the CB for channel 2). When a channel start command is issued through the CCW, the 20-bit address of the Command Parameter Block is formed from the offset and segment address values (see figure 1-9) and stored in the channel's PP register.

The Command Parameter Block (PB) is of variable, user-defined size. It contains two pre-defined fields: bytes 0—1 contain either the 16-bit address of a local (I/O) space task block program or the 16-bit offset value of a system (memory) task block program; bytes 2—3 contain the 16-bit segment address of a task block program located in system (memory) space. These two fields are also used by the 8089 hardware to store a channel's PSW (see below), and its TP pointer/register and tag bit when a channel's operation is suspended.



NOTE:

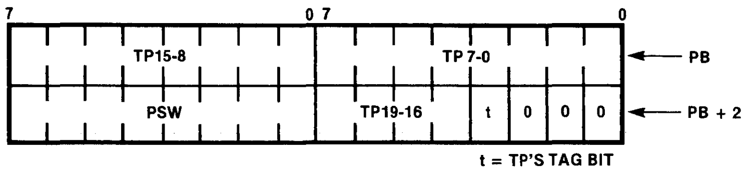
1. THIS METHOD IS IDENTICAL TO THAT USED BY THE 8086 TO FORM 20-BIT ADDRESSES.
2. ALL 20-BIT ADDRESSES ARE FORMED BY THE IOP ACCORDING TO THE ABOVE METHOD. ONCE A 20-BIT ADDRESS HAS BEEN FORMED, IT CANNOT BE DISASSEMBLED INTO ITS 16-BIT OFFSET VALUE AND SEGMENT ADDRESS COMPONENTS. THE 8089 CAN BOTH STORE AND RESTORE 20-BIT ADDRESSES. (SEE THE MOVP INSTRUCTION DESCRIPTION IN CHAPTER 3.)

Figure 1-9. The Formation of 20-Bit Addresses by the 8089 Hardware

The size of the PB following the above four bytes is user-defineable. This area may be used to pass messages between the host processor and the 8089. The STRUC assembler directive creates a template of offset values which can be used to access blocks of parameters and I/O control information in this area, using the PP register as a base address. (See the section "Data Memory Operands" in Chapter 2 and the STRUC assembler directive in Chapter 4.)

When a channel is started by the host processor, the Command Field of the CCW specifies where the channel's task block program is located.. If the task block program is in local (I/O) space, a 16-bit address from the first word (2 bytes) of the PB is loaded into the TP pointer/register. TP's tag bit is set to logical one (see figure 1- 10A). If the task block program is in system (memory) space, a 20-bit address is formed from a 16-bit offset value in the first word of the PB and a segment address contained in the second. TP's tag bit is set to logical zero. (See figure 1-10B.)

When a channel's operation is suspended by a HALT AND SAVE command issued through the CCW (Command Field (CF) contains 110 binary, HALT AND SAVE), the 20-bit TP pointer/register, its tag bit, and the channel's PSW are stored in the first four bytes of the PB:



FORMAT OF THE STORED TP POINTER/REGISTER, TAG BIT AND CHANNEL PSW, SAVED WHEN A HALT COMMAND IS ISSUED THROUGH THE CCW

The Program Status Word (PSW) is a byte containing information on a channel's status. It is continuously updated by the 8089 but is not directly accessible by task block programs. It can only be examined when a channel's operation has been suspended, at which time it is stored in the fourth byte of the channel's PB by the 8089 hardware.

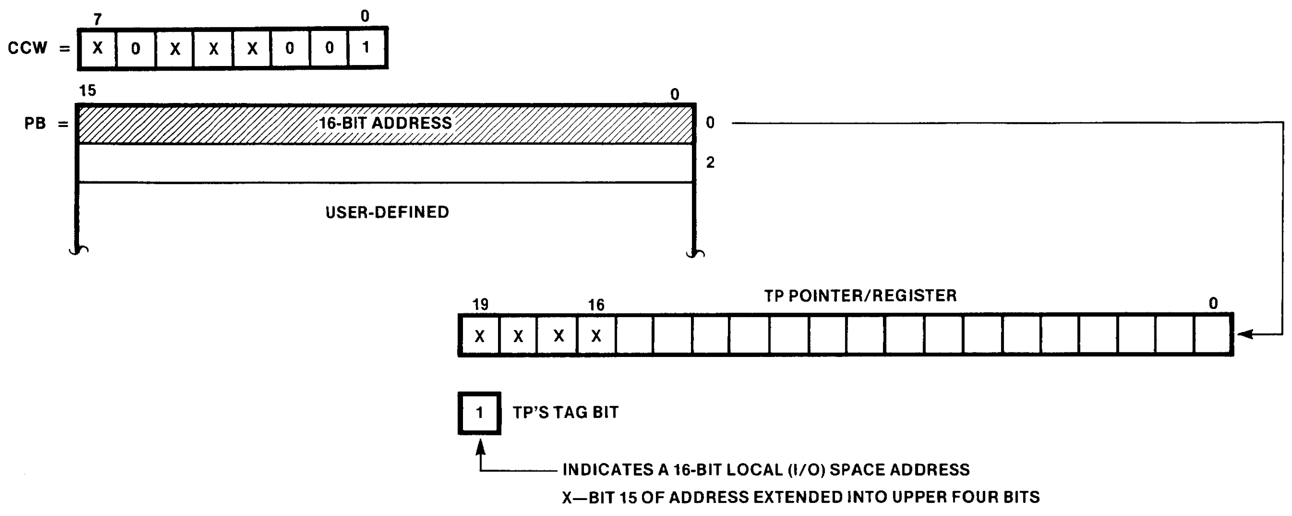


Figure 1-10A. The Loading of a Local (I/O) Space Task Block Program Address Into the Pointer/Register

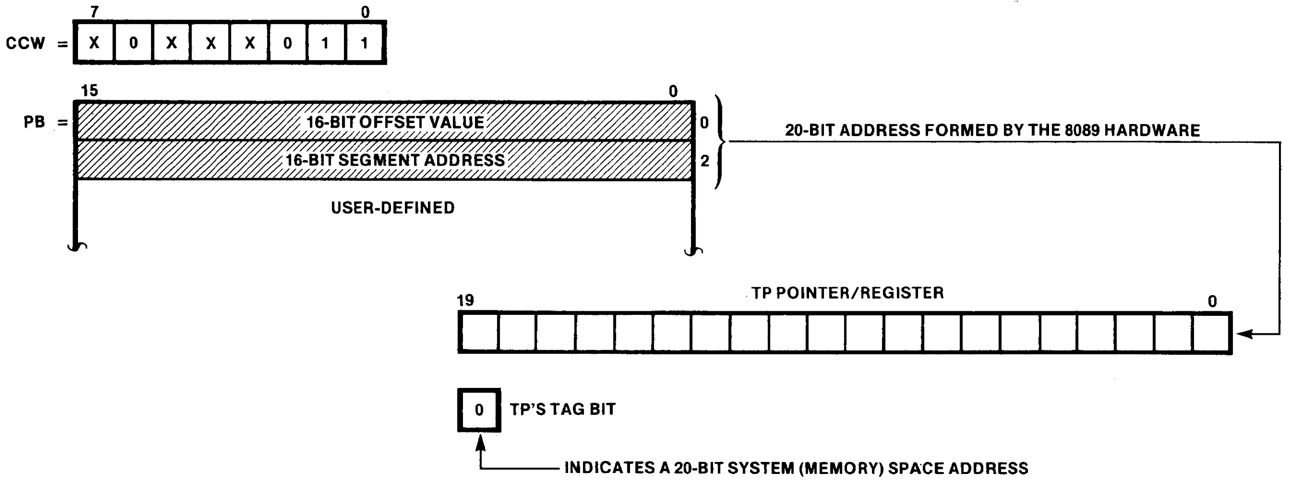
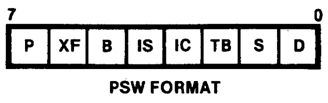


Figure 1-10B. The Loading of a System (MEMORY) Space Task Block Program Address Into the TP Pointer/Register

The PSW contains the following fields:



PSW FORMAT

P:	PRIORITY FIELD (CCW)
XF:	CHANNEL IN ACTIVE TRANSFER STATE
B:	BUS LOAD LIMIT FIELD (CCW)
IS:	INTERRUPT SERVICE (REQUEST) FLIP FLOP
IC:	INTERRUPT CONTROL FLIP FLOP
TB:	CHANNEL EXECUTING TASK BLOCK INSTRUCTIONS
S:	SOURCE WIDTH IS 8/16 (0/1)
D:	DESTINATION WIDTH IS 8/16 (0/1)

0 = SET 1 = NOT SET

When channel operations are resumed following their suspension (101B in the Command Field of the CCW), the stored TP pointer/register and tag bit are restored from the PB by the 8089 hardware. Any changes to the PSW while it was stored will be in effect when channel operation resumes. (See figure 1-11.)

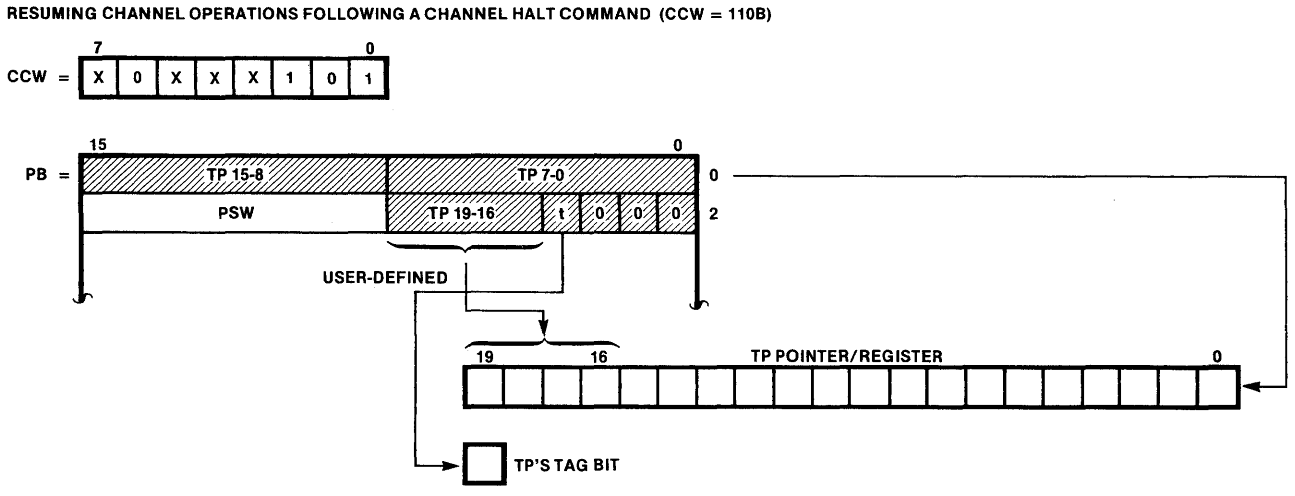


Figure 1-11. Loading a Stored Task Block Program Address Into TP When Channel Operation is Resumed Following a Channel HALT AND SAVE Command (CCW=110B)

The TP Pointer/Register and Task Block Programs

A channel's TP pointer/register functions as the task block program instruction pointer. TP points to the location of the task block program instruction to be executed.

TP is normally loaded by the 8089 hardware from a channel's Command Parameter Block when task block program execution is started. The Command Field of a channel's CCW specifies the location of a task block program and determines the value of TP's tag bit. If a local (I/O) space task block program is specified, the tag bit is set to a logical one and TP is loaded with a 16-bit address from the PB. If a system (memory), task block program is specified, the tag bit is set to a logical zero and TP is loaded with a 20-bit address from the PB.

When a channel's operation is suspended by a command in the CCW, TP and its tag bit are stored in the first three bytes of the channel's PB. A task block program CALL instruction also stores the TP pointer/register and tag bit, at a location specified by a CALL instruction operand.

8089 Initialization

A linked list of data memory blocks is prepared by the host processor in shared data memory and used to initialize the 8089. Each block in the chain specifies certain system parameters and points to the location of the next block in the sequence. Figure 1-12 shows the initialization sequence.

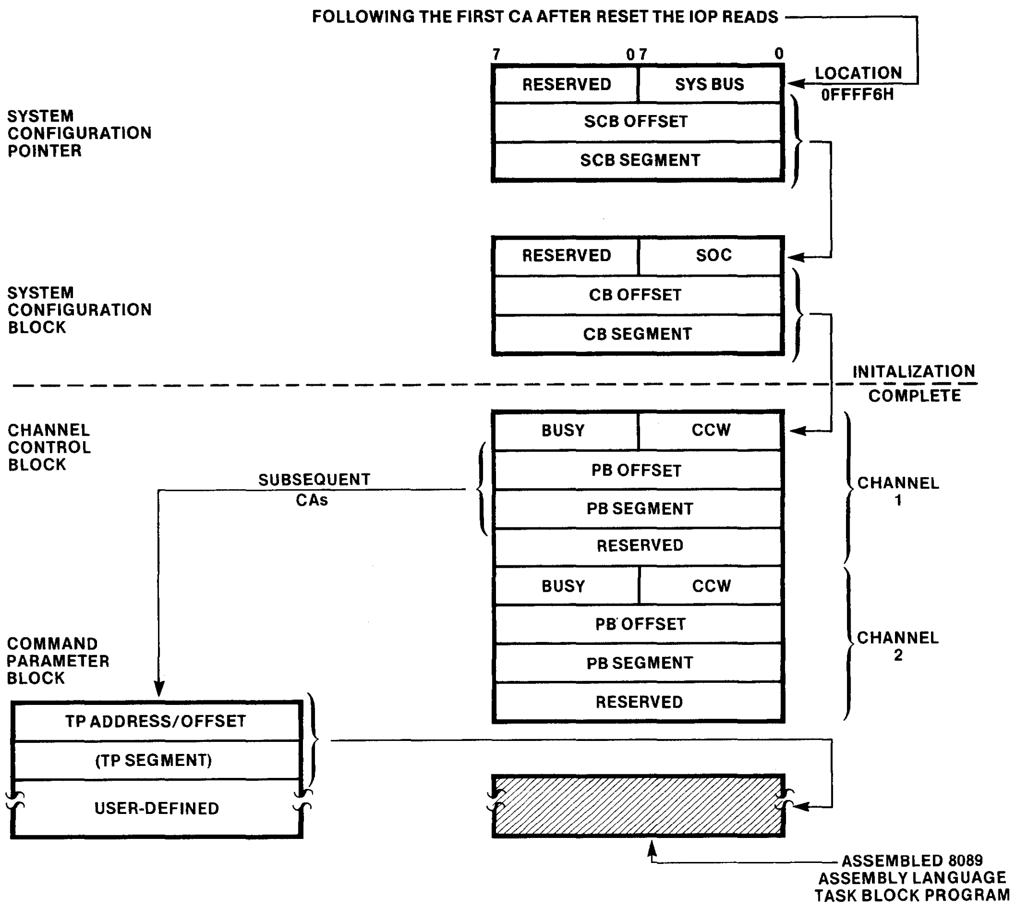
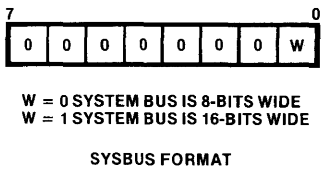


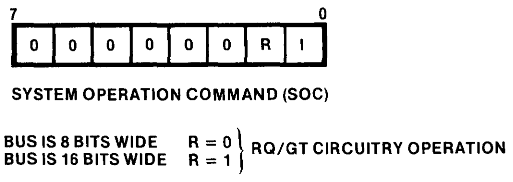
Figure 1-12. 8089 Initialization and Communication Blocks

The first memory block in the sequence, the System Configuration Pointer (SCP), is the only block whose location is fixed. It must be located in system (memory) space at address 0FFFF6H. This block is inspected by the IOP following the first channel attention (CA) it receives after a reset. The first byte of the SCP defines the width of the system (memory) bus to the 8089.


7
0
0 0 0 0 0 0 0 W
W = 0 SYSTEM BUS IS 8-BITS WIDE
W = 1 SYSTEM BUS IS 16-BITS WIDE
SYSBUS FORMAT

The second byte of the SCP is reserved. Bytes two through five point to the location of the System Configuration Block (SCB), the next block in the initialization sequence. The SCB's offset value is contained in the first two bytes; its segment address is contained in the next two bytes. The 20-bit address of the SCB is formed from the offset value and the segment address.

The SCB is a six byte block that may be located anywhere in system (memory) space. The block contains information regarding request/grant circuitry operation and also specifies the width of a remote bus, if one is present. The first byte of the block contains the system operation command (SOC):


7
0
0
0
0
0
0
0
R
I
I = 0 THE REMOTE BUS IS 8 BITS WIDE
I = 1 THE REMOTE BUS IS 16 BITS WIDE
R = 0
R = 1
RQ/GT CIRCUITRY OPERATION

SYSTEM OPERATION COMMAND (SOC)

where:

"I" defines the width of a remote bus to the 8089. The width of this bus may differ from that of the system (memory) bus. In a LOCAL configuration, where there is no remote bus, 'I' should specify the bus width for the system bus, given in the SCP.

"R" specifies the mode of request/grant circuitry operation when the RQ/GT line is used to control access to a bus shared between two processors. One processor is a MASTER, the other is a SLAVE. The 8089 is designated a MASTER or a SLAVE by a hardware input (the SEL pin) from the host processor during initialization.

A MASTER controls the bus on initialization and grants control to the SLAVE upon request. If the bus is shared with an 8086 or 8088 host processor, the IOP must be a SLAVE and the value of "R" must be logical zero. The IOP, through the RQ/GT circuitry, requests the bus from the MASTER and automatically returns bus control to the MASTER when it is finished.

If two 8089s share a bus, their "R" values must be the same. If "R" is a logical one, the SLAVE requests the bus from the MASTER as above but does NOT relinquish bus control when it is finished. The MASTER must request the bus from the SLAVE if he wishes to use it. Bus control alternates between the IOPs, each requesting the bus if it does not control it.

The SCB contains the offset and segment address of the Channel Control Block (CB). The 16-bit offset value is located in bytes two and three of the SCB. Bytes four and five contain the 16-bit segment address. The 20-bit address of the CB is formed from the offset value and segment address by the 8089 hardware.

After the SCB has been read, the 8089 hardware writes 00H to the BUSY flag byte of channel 1 in the Channel Control Block, indicating the end of IOP initialization. The SCB may now be used to initialize other 8089s in the system, if they are present.

Registers

There are two identical sets of registers in the 8089, one for each channel. The registers are used by 8089 Assembly Language task block programs and DMA transfer operations. Figure 1-13 shows a channel's register set.

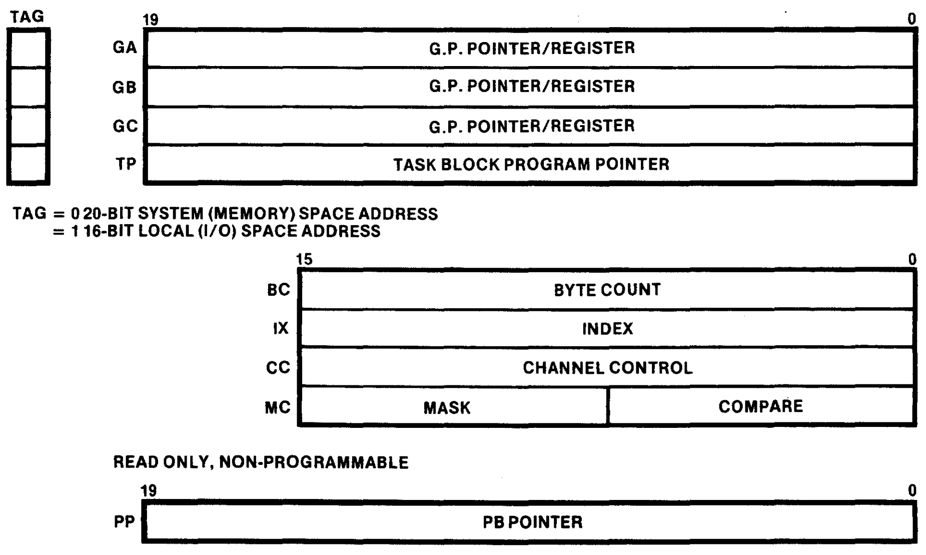


Figure 1-13. An 8089 Channel's Register Set

GA, GB, GC, and TP are 20-bit pointer/registers. Each pointer/register has an associated tag bit and is used, primarily, to address data. The value of the tag bit indicates whether the pointer/register contains a 16-bit local (I/O) space address or a 20-bit system (memory) space address (see "8089 Addressing Scheme" later in this chapter). Pointer/registers can also be used as 16-bit general purpose registers in task block programs. When used as a 16-bit register, the upper four bits of a pointer/register are filled with the sign bit (bit 15 or bit 7) of data.

There are four 16-bit registers: BC, IX, CC, and MC. Registers BC, IX, and MC can be used as general purpose registers. IX and MC have specific uses in the 8089 Assembly Language: IX can supply an index value in data memory operands (see "Data Memory Operands" in Chapter 2); MC supplies mask/compare bytes in JMCE and JMCNE conditional transfer instructions (see Chapter 3). BC, IX, and MC also play special roles in DMA transfer operations. Register CC is only used to control chained task block program instruction execution and DMA transfer operations. The section on DMA transfer later in this chapter describes CC's role in an 8089 channel's operation.

One register, PP, is read only, non-programmable. It contains the address of a channel's Command Parameter Block, which is automatically loaded when the channel is issued a start command through its CCW.

The following lists the features and function of each register:

GA, GB:	GA and GB are 20-bit pointer/registers, each with an associated tag bit. In task block programs, they are used to point to data. In DMA transfers, they provide the source and destination addresses, as specified in register CC. GA and GB also may be used as 16-bit general purpose registers in task block programs.
GC:	A 20-bit pointer/register with an associated tag bit, GC is used to point to data in task block programs. In the translate mode of DMA transfer, GC contains the base address of a 256-byte translation table. It also may be used as a 16-bit general purpose register in task block programs.

PP:	PP is a 20-bit read only, non-programmable register containing the address of a channel's PB. This address is automatically loaded when a channel is started and always points to system (memory) space. PP is used as a base address to access the user-defined portion of the PB.
IX:	IX is a 16-bit general purpose register. In some memory addressing modes, IX is added to a base pointer/register to access data.
BC:	BC is a 16-bit general purpose register, used as a byte counter during DMA transfers. BC is decremented by one after each transfer from an 8-bit soure; by two after each transfer from a 16-bit source.
MC:	A 16-bit general purpose register, MC supplies mask and compare bytes used by the task block program instructions JMCE and JMCNE, and also in DMA transfer mask/compare operations.
TP:	A 20-bit pointer/register with an associated tag bit, TP is equivalent to a conventional program counter in task block program execution, i.e., it points to the location of the next instruction to be executed. TP is loaded from the PB when task block program execution is started or resumed.
CC:	A 16-bit register, CC controls DMA transfers and chained task block program instruction execution.

8089 Addressing Scheme

All data in task block programs, except for instructions using immediate data, is addressed indirectly, i.e., all data is pointed to by a pointer/register containing a base address; offset and index values can optionally be added to this base address. (See "Data Memory Operands" in Chapter 2.)

8089 addresses are physically 20 bits in length. There are two distinct types of addresses:

- 20-bit system (memory) addresses (1 megabyte)
- 16-bit local (I/O) addresses (64k bytes)

In the hardware, these address types correspond to the 20-bit memory and 16-bit I/O addresses of the 8086. However, unlike the 8086, the 8089 does not have separate instructions for memory and I/O operations. Instead, the 8089 uses the pointer/register tag bits to indicate 16-bit local (I/O) addresses (tag bit = 1) and 20-bit system (memory) addresses (tag bit = 0).

Both 20- and 16-bit addresses may be needed in a task block program, whether the 8089 has its own remote bus (REMOTE configuration) or shares a bus with a host processor (LOCAL configuration). In a REMOTE configuration, 16-bit addresses are used to access the 8089's remote bus and 20-bit addresses are used to access the shared system bus. In a LOCAL configuration with an 8086, 16-bit addresses access I/O ports and 20-bit addresses access memory. A programmer must know the type of address (16- or 20-bit) needed when accessing a system's resources.

DMA Transfer

The 8089 is designed to manage and maintain high speed DMA transfers between the following:

- Memory → I/O port
- I/O port → I/O port
- Memory → Memory

DMA transfers are initiated by a special task block program instruction and use some of a channel's registers in their operation. Table 1-1 shows these registers and their role in DMA transfer operations.

Table 1-1. Registers Used by DMA Transfer Operations

REGISTER	ROLE IN DMA TRANSFER
GA, GB	Specify DMA Source and Destination
GC	Provides base address of 256 byte translate table
BC	Byte counter-decremented by byte or word
MC	Contains mask/compare byte for data testing
CC	Specifies DMA transfer control parameters

Register CC specifies control parameters governing DMA transfers. Figure 1-14 shows the fields it contains and the parameters they specify.

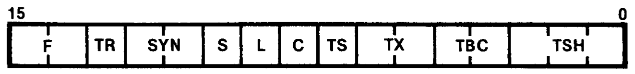
Register CC also controls chained task block program instruction execution by a channel (bit eight). Normally, the 8089 observes the following priorities when servicing the 8089's two channels:

- (highest priority) DMA transfers
- Channel Attentions (CA's)
- Task block program instruction execution
- (lowest priority) Idle cycles

When both channels request service, the channel with the higher priority task is serviced first. In the nonchained mode, no task block program instruction execution occurs on a channel if a DMA transfer is being performed on the other channel. In chained mode, the priority of task block program instruction execution equals that of DMA transfer, possibly allowing a channel's task block program to execute concurrently with DMA transfers on the other channel (depending on "P" in the CCW).

NOTE

The above discussion of priorities in 8089 channel operation is overly-simplified. Caution should be observed when using chained task block program instruction execution. For a complete explanation of channel priorities in the 8089, see the *MCS-86 User's Manual,* Order No. 9800722.



CHANNEL CONTROL REGISTER

F FUNCTION CONTROL

00	PORT TO PORT	GS → GD
01	BLOCK TO PORT	(GS)+ → GD
10	PORT TO BLOCK	GS → (GD)+
11	BLOCK TO BLOCK	(GS)+ → (GD)+

GS AND GD ARE THE SOURCE/DESTINATION POINTERS AS SELECTED BY THE S FIELD. BLOCK (MEMORY) POINTERS ARE POST AUTO-INCREMENTED (BYTE/WORD), INDICATED BY (GS)+ OR (GD)+.

TR TRANSLATE MODE

0 NO EFFECT
1 TRANSLATE INCOMING DATA; THE INCOMING BYTE IS ADDED AS A POSITIVE DISPLACEMENT TO REGISTER GC. THE ADDRESS FORMED IS USED TO FETCH A BYTE WHICH IS TREATED AS THE NORMALLY FETCHED DATA.

TRANSLATE MODE IS ONLY ALLOWED WHEN BOTH SOURCE AND DESTINATION LOGICAL WIDTHS, AS SET BY THE TBP WID INSTRUCTION, ARE EIGHT.

SYN SYNCHRONIZATION CONTROL

00 NONE; TRANSFERS ARE AUTOMATIC
01 SOURCE; TRANSFERS ARE SYNCHRONIZED WITH DMA REQUESTS FROM THE SOURCE.
10 DESTINATION; TRANSFERS ARE SYNCHRONIZED WITH DMA REQUESTS FROM THE SPACE DESTINATION.
11 (RESERVED)

S SOURCE/DESTINATION FIELD

0 GA IS SOURCE POINTER; GB IS DESTINATION
1 GB IS DESTINATION POINTER; GA IS SOURCE

L LOCK CONTROL

0 NO LOCK
1 LOCK ACTIVATED; DURING TRANSFERS, THE IOP'S LOCK PIN IS ACTIVATED UPON THE RECEIPT OF THE FIRST DMA REQUEST UNTIL THE COMPLETION OF THE LAST TRANSFER.

C CHAINING CONTROL

0 NO CHAINING MODE
1 CHAINING MODE; SET THE PRIORITY OF TBP PROCESSING EQUAL TO THE PRIORITY OF DMA PROCESSING.

TS SINGLE TRANSFER

0 NO EFFECT
1 SINGLE BYTE OR WORD TRANFERS, AS SPECIFIED BY THE WID TASK BLOCK PROGRAM INSTRUCTION. DMA TRANSFER IS TERMINATED AFTER EACH TRANSFER. TPB EXECUTION RESUMES AT TP.

IN SINGLE TRANSFER MODE, THE SOURCE AND DESTINATION LOGICAL WIDTHS, AS SET BY THE WID INSTRUCTION MUST BE EQUAL.

TX EXTERNAL TERMINATE

00 NO EFFECT
01 TERMINATE DMA TRANSFERS WHEN THE EXTERNAL TERMINATE PIN IS TRUE; RESUME TBP EXECUTION AT TP.
10 SAME AS 01 ABOVE; RESUME TBP EXECUTION AT TP + 4.
11 SAME AS 01 ABOVE; RESUME TBP EXECUTION AT TP + 8.

TBC BYTE COUNT TERMINATION

00 NO EFFECT
01 TERMINATE DMA TRANSFERS WHEN REGISTER BC = 0; RESUME TBP EXECUTION AT TP.
10 SAME AS 01 ABOVE; RESUME TBP EXECUTION AT TP + 4.
11 SAME AS 01 ABOVE; RESUME TBP EXECUTION AT TP + 8.

TSH MASK/COMPARE TERMINATION

000 SEARCH INCOMING BYTES UNTIL A MATCH IS FOUND. DMA TRANSFER IS TERMINATED AND THE MATCHING BYTE IS THE LAST BYTE TRANSFERRED. RESUME TBP EXECUTION AT TP.
010 SAME AS 001 ABOVE; RESUME TBP EXECUTION AT TP + 4.
001 SAME AS 001 ABOVE; RESUME TBP EXECUTION AT TP + 8.
100 NO EFFECT
101 SEARCH INCOMING BYTES DURING DMA TRANSFERS WHILE MATCHING OCCURS. DMA TRANSFER IS TERMINATED AND THE NON-MATCHING BYTE IS THE LAST BYTE TRANSFERRED. RESUME TBP EXECUTION AT TP.
110 SAME AS 101 ABOVE; RESUME TBP EXECUTION AT TP + 4
111 SAME AS 101 ABOVE; RESUME TBP EXECUTION AT TP + 8.

Figure 1-14. The Channel Control Register

The WID and XFER task block program instructions are directly associated with DMA transfer. WID sets the logical width of the DMA source and destination. These logical widths determine what type of data assembly/disassembly occurs during DMA transfers. (See the *MCS-86 User's Manual* for information on assembly/disassembly operations in the 8089.)

The XFER task block program instruction initiates DMA transfer. DMA transfer mode is entered after the execution of the instruction following the XFER instruction. This allows a task block program to pass information to a peripheral with the channel ready to accept DMA transfer requests immediately. To insure correct DMA transfer operation, the instruction following the XFER instruction must not load the pointer/registers GA or GB, or register CC.

Interrupts

A channel uses the SINTR task block program instruction to generate interrupts to the external system interrupt hardware. Each channel has its own hardware pin, SINTR-1 and SINTR-2, for this function.

The host processor uses the ICF of the CCW in the Channel Control Block to control interrupts from the IOP's channels. Interrupts must be enabled in the ICF for the external system to detect them. Otherwise, SINTR task block program instructions have no effect. The ICF is also used by a host processor to acknowledge or disable channel interrupts.

A Sample Task Block Program

The following example task block program was written to conduct a simple I/O operation in a REMOTE configuration system, i.e., the IOP has its own remote bus. The task block program performs a DMA transfer from data memory in IOP local address space to data memory in system address space.

Figure 1-15 is a copy of the list file from ASM89's processing of the 8089 Assembly Language source program MOVBUF.SRC. The NAME assembler directive assigns the name EXAMPLE__PROGRAM to the object module, which is placed by the assembler in the object file MOVBUF.PRG on device :F1:. The SEGMENT assembler directive assigns the name SEG89 to the segment defined in the object module.

In the beginning of the source file, a section of data memory is reserved for the DMA transfer source data, SOURCE. A double word of data memory is also reserved to contain the offset value and segment address of the external symbol INPUT__DATA, the DMA transfer destination in 20-bit system address space. The offset and segment address values of INPUT__DATA are supplied by LINK86 processing of the object module. The example task block program starts at the location labeled STRT@TB@PRG1.

The following pages trace the execution of this sample task block program through four stages. Note that either IOP channel could execute the program, provided the appropriate preparations are made, i.e., the program's address is present in the channel's Command Parameter Block when the channel is issued a start or resume task block program execution command.

Stage One

A memory map of the host processor/IOP system is shown in figure 1-16. The system is in a REMOTE configuration, i.e., the IOP has its own remote bus, not accessible by the host processor. The one megabyte of system memory is accessed by the IOP over the shared system bus (pointer/register tag bit = 0). The 64k bytes of local IOP memory is accessed over the remote bus (pointer/register tag bit = 1).

The segment defined in the example source program's object module (SEG89) has been located by LOC86 at address 0H in the IOP's local memory. In this example, then, the assembler's location counter values, given in the printed listing, correspond to the absolute addresses assigned by LOC86.

The assembly language source program DD directive reserves four bytes (a double word) for the offset value and segment address of the external symbol INPUT__DATA. LINK86 and LOC86 processing of the assembler-generated object module supply these values (see Stage 2).

```
8089 ASSEMBLER                                                                          PAGE    1

ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE EXAMPLE_PROGRAM
OBJECT MODULE PLACED IN :F1:MOVBUF.PRG
ASSEMBLER INVOKED BY :F1:ASM89 :F1:MOVBUF.SRC PRINT(:F1:MOVBUF.PRT) OBJECT(:F1:MOVBUF.PRG)

                                 1 ;*******************************************************************
                                 2 ;*                                                                 *
                                 3 ;* THE IOP HAS ITS OWN REMOTE BUS IN THIS SYSTEM (REMOTE CONFIGURATION). *
                                 4 ;* THIS TASK BLOCK PROGRAM PERFORMS A DMA TRANSFER OPERATION TO MOVE    *
                                 5 ;* DATA FROM DATA MEMORY ACCESSED BY THE REMOTE BUS, TO DATA MEMORY    *
                                 6 ;* SHARED WITH A HOST PROCESSOR VIA THE SYSTEM BUS.                  *
                                 7 ;*                                                                 *
                                 8 ;*******************************************************************
                                 9 ;
                                10 ;
                                11                NAME   EXAMPLE_PROGRAM      ;ASSIGNS A NAME TO THE OBJECT MODULE.
                                12 ;
0000                            13 SEG89          SEGMENT                     ;THIS SEGMENT DIRECTIVE NAMES THE 64K
                                14                                            ;SEGMENT THAT WILL CONTAIN THE
                                15                                            ;ASSEMBLER-GENERATED OBJECT CODE.
                                16                                            ;THIS SEGMENT NAME IS USED BY LOC86
                                17                                            ;TO LOCATE THE THE OBJECT MODULE.
                                18 ;
                                19                EXTRN  INPUT_DATA           ;IDENTIFY THE SYMBOL INPUT_DATA AS A
                                20                                            ;SYMBOL DEFINED IN ANOTHER ASSEMBLY
                                21                                            ;OR COMPILATION.
                                22 ;
  0080                          23 BUFCNT         EQU    128
                                24 ;
  C408                          25 CHANNEL_CNTRL  EQU    0C408H
                                26 ;
0000                            27 SOURCE:        DS     128                  ;RESERVE 128 BYTES OF DATA MEMORY FOR
                                28                                            ;THE INPUT BUFFER.
                                29 ;
0080   00000000                 30 DESTINATION:   DD     INPUT_DATA           ;DEFINES A DOUBLE WORD CONTAINING
                                31                                            ;THE OFFSET AND SEGMENT ADDRESS
                                32                                            ;OF THE DMA TRANSFER DESTINATION IN
                                33                                            ;SHARED SYSTEM DATA MEMORY.
                                34 ;
0084   5130 8000                35 STRT$TB$PRG1:  MOVI   GC, DESTINATION      ;LOAD THE ADDRESS OF THE DATA MEMORY
                                36                                            ;LOCATION IN LOCAL SPACE THAT
                                37                                            ;CONTAINS THE OFFSET AND SEGMENT
                                38                                            ;ADDRESS OF THE DMA TRANSFER
                                39                                            ;DESTINATION INTO GC.
                                40 ;
0088   018A                     41                LPD    GA, [GC]             ;FORM A 20-BIT ADDRESS FROM THE
                                42                                            ;OFFSET AND SEGMENT ADDRESS STORED
                                43                                            ;AT [GC].  GA'S TAG BIT IS SET TO
                                44                                            ;LOGICAL '0', INDICATING A 20-BIT
                                45                                            ;SYSTEM SPACE ADDRESS.
                                46 ;
008A   3130 0000                47                MOVI   GB, SOURCE           ;LOAD THE 16-BIT ADDRESS OF THE DMA
                                48                                            ;TRANSFER SOURCE INTO GB.
                                49                                            ;GB'S TAG BIT IS SET TO A LOGICAL '1'
                                50                                            ;INDICATING A 16-BIT LOCAL SPACE
                                51                                            ;ADDRESS.
                                52 ;
008E   D130 08C4                53                MOVI   CC, CHANNEL_CNTRL
                                54 ;


8089 ASSEMBLER                                                                          PAGE    2

0092   8000                     55                WID    8, 8                 ;SET DMA TRANSFER SOURCE AND
                                56                                            ;DESTINATION LOGICAL WIDTHS TO 8-
                                57                                            ;BITS. THE LOGICAL WIDTH DETERMINES
                                58                                            ;DATA ASSEMBLY/DISASSEMBLY DURING
                                59                                            ;DMA TRANSFERS.
                                60 ;
0094   6000                     61                XFER                        ;BEGIN DMA TRANSFER OPERATION
                                62                                            ;FOLLOWING THE EXECUTION OF THE NEXT
                                63                                            ;INSTRUCTION.
                                64 ;
0096   6830 80                  65                MOVBI  BC, BUFCNT           ;SET BYTE COUNT TO 128.  THE WID
                                66                                            ;INSTRUCTION SPECIFIES AN 8-BIT
                                67                                            ;SOURCE SO REGISTER BC IS
                                68                                            ;DECREMENTED BY 1 AFTER EACH
                                69                                            ;TRANSFER.  IF WID SPECIFIES A 16-
                                70                                            ;BIT SOURCE, REGISTER BC IS
                                71                                            ;DECREMENTED BY 2 AFTER EACH
                                72                                            ;TRANSFER.
                                73 ;
0099   2048                     74                HLT                         ;TASK BLOCK PROGRAM EXECUTION RESUMES
                                75                                            ;HERE FOLLOWING THE DMA TRANSFER
                                76                                            ;OPERATION.  TASK BLOCK PROGRAM
                                77                                            ;EXECUTION ENDS AND THE CHANNEL BUSY
                                78                                            ;BYTE IN THE CHANNEL CONTROL BLOCK
                                79                                            ;IS CLEARED.
                                80 ;
009B                            81 SEG89          ENDS                        ;THE END OF THE SEGMENT.
                                82 ;
                                83                END                         ;THE END OF THE SOURCE PROGRAM.
```

Figure 1-15. Example Task Block Program List File

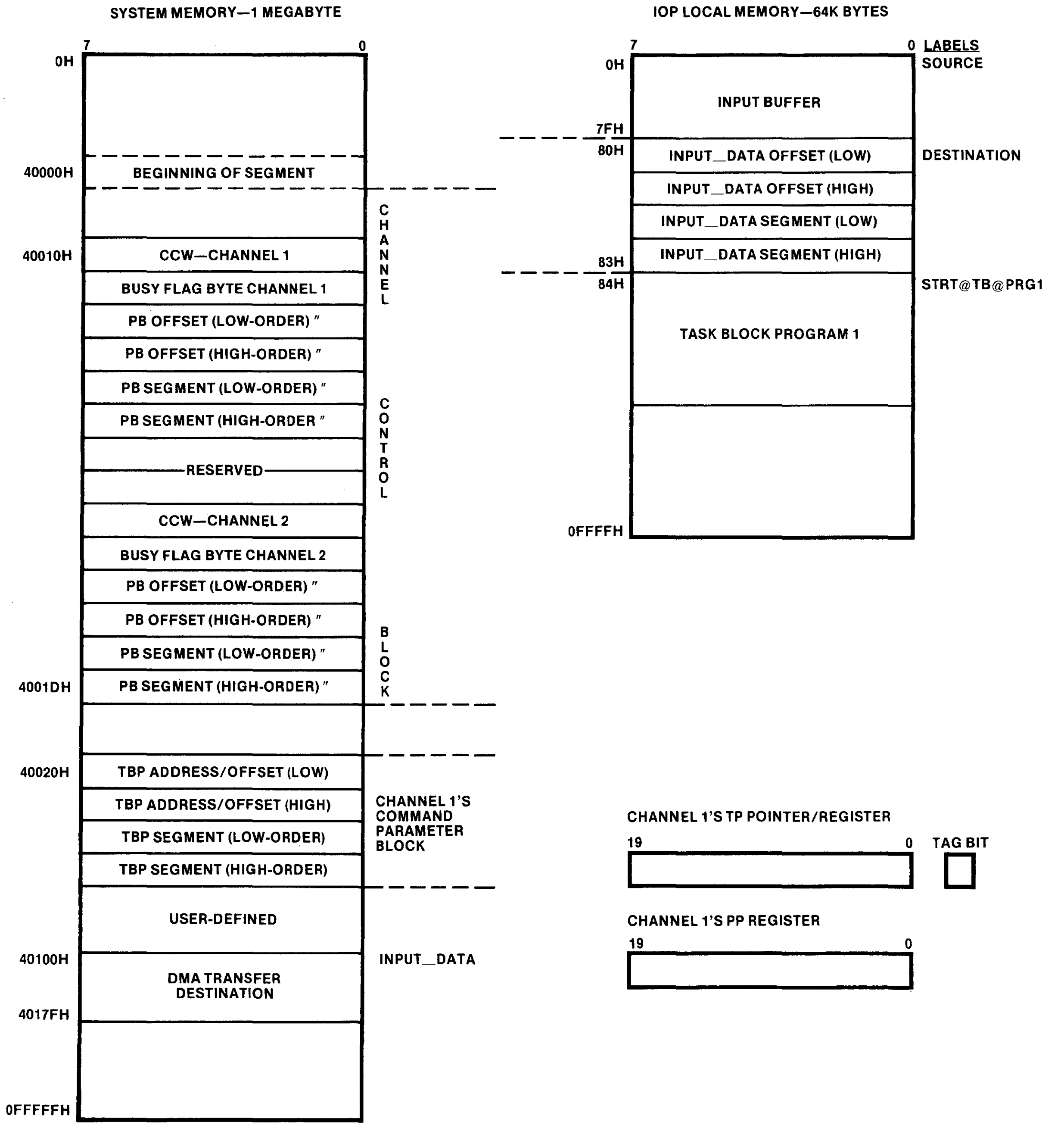


Figure 1-16. Stage One—System Memory Map

The blocks of shared memory for host processor–IOP communications (Channel Control Block and Command Parameter Block) are contained in a segment located at address 40000H in system memory.

This example assumes that the host processor has the address of the task block program to be executed (Task Block Program 1), possibly supplied by LINK86.

Stage Two

Figure 1-17 shows the preparations made by the host processor before task block program execution by channel 1 is started.

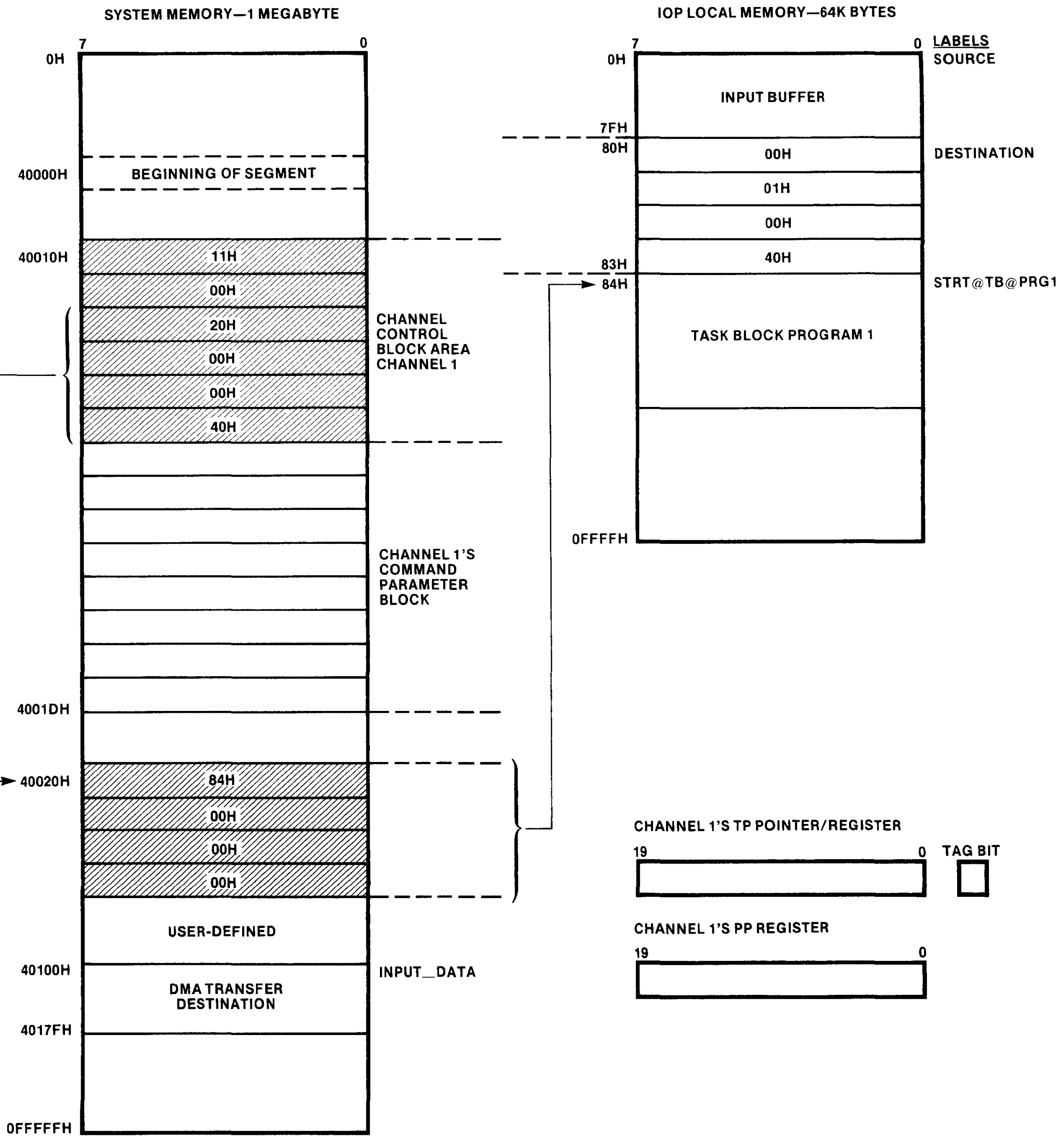
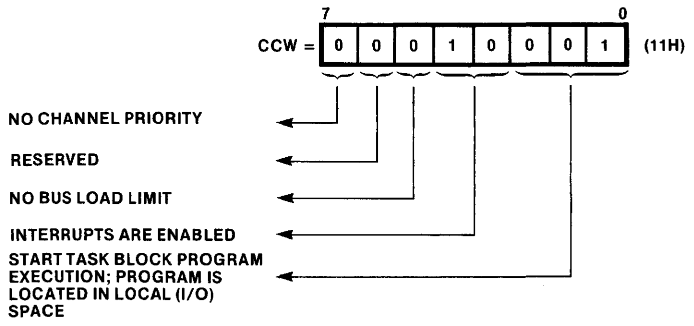


Figure 1-17. Stage Two—Host Processor Preparations

The channel control word (CCW) for channel 1, placed in the Channel Control Block, specifies:


7
0
CCW = 0 0 0 1 0 0 0 1 (11H)
NO CHANNEL PRIORITY
RESERVED
NO BUS LOAD LIMIT
INTERRUPTS ARE ENABLED
START TASK BLOCK PROGRAM EXECUTION; PROGRAM IS LOCATED IN LOCAL (I/O) SPACE

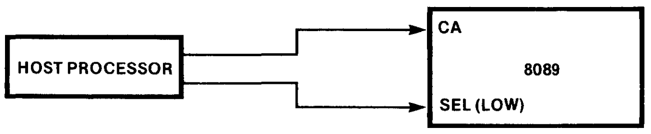
Channel 1's BUSY flag byte contains 00H, indicating that the channel is presently inactive.

The address of channel 1's Command Parameter Block (PB) has been placed in bytes 2-5 of the Channel Control Block. Bytes 2-3 contain the CP's offset value. Bytes 4-5 contain the PB's segment address.

The address of the task block program to be executed by channel 1 has been placed in its Command Parameter Block. Since the CCW specifies a local (16-bit address) task block program location, only the first two bytes are accessed when channel 1 loads the task block program address into its TP pointer/register.

Stage Three

The host processor activates channel 1 via a channel attention and the SEL input pin value:



The 8089 hardware reads channel 1's CCW and:

- Computes the 20-bit address of its Command Parameter Block and stores it in channel 1's PP register
- Loads the task block program address into channel 1's TP pointer/register and sets TP's tag bit to logical 1, indicating a local space task block program, as specified in the CCW
- Writes 0FFH to channel 1's BUSY flag byte in the Channel Control Block.

Task block program execution starts at the instruction beginning at the address in channel 1's TP pointer/register (84H in local IOP memory—see figure 1-18). The address of the data memory location in local IOP space containing the offset and segment address of the DMA transfer destination, INPUT_DATA, is loaded into pointer/register GC. A 20-bit address is formed from the offset and segment data and placed in pointer/register GA by the LPD GA, [GC] instruction. GA's tag bit is set to logical 0, indicating a 20-bit system space address.

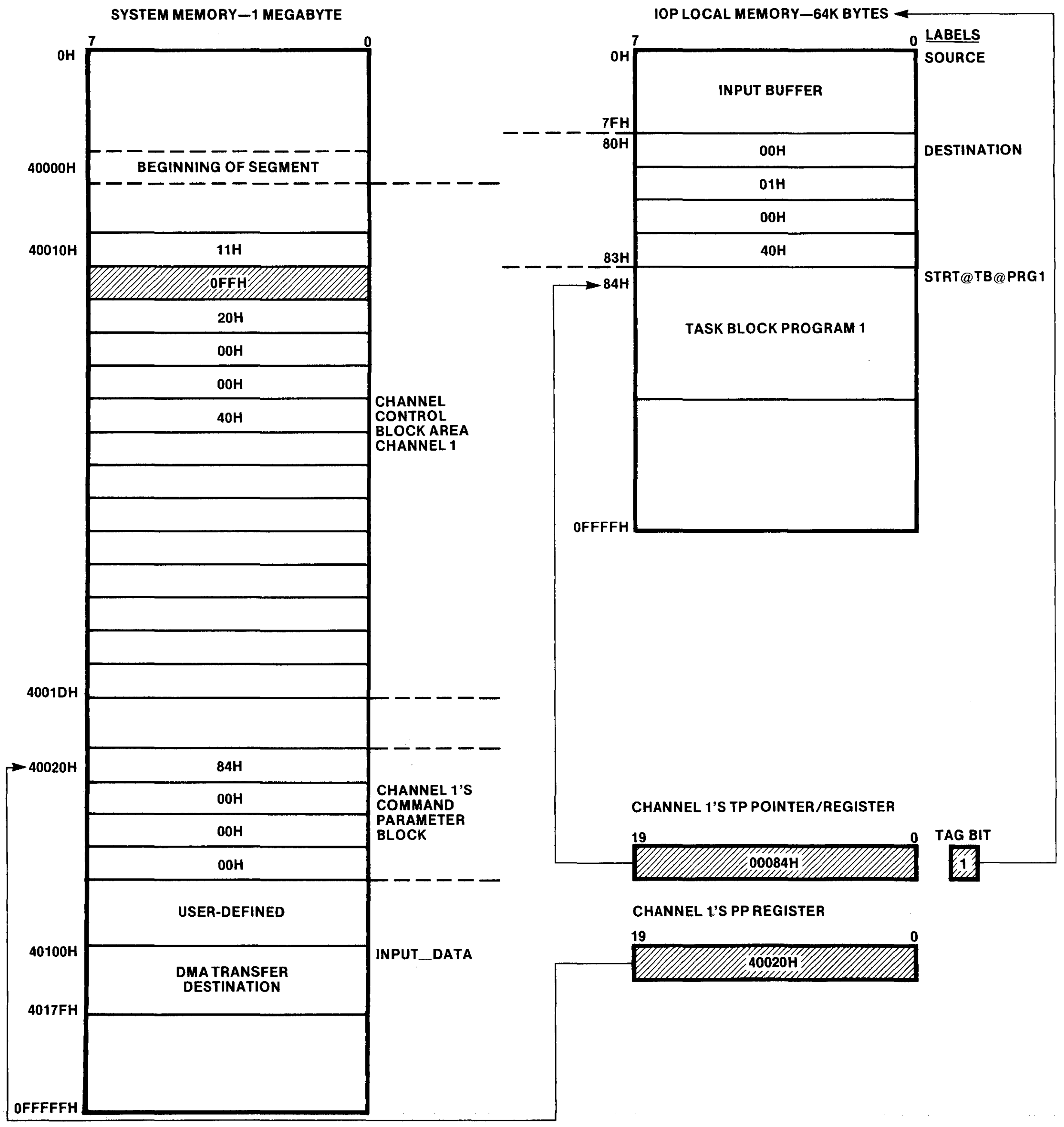
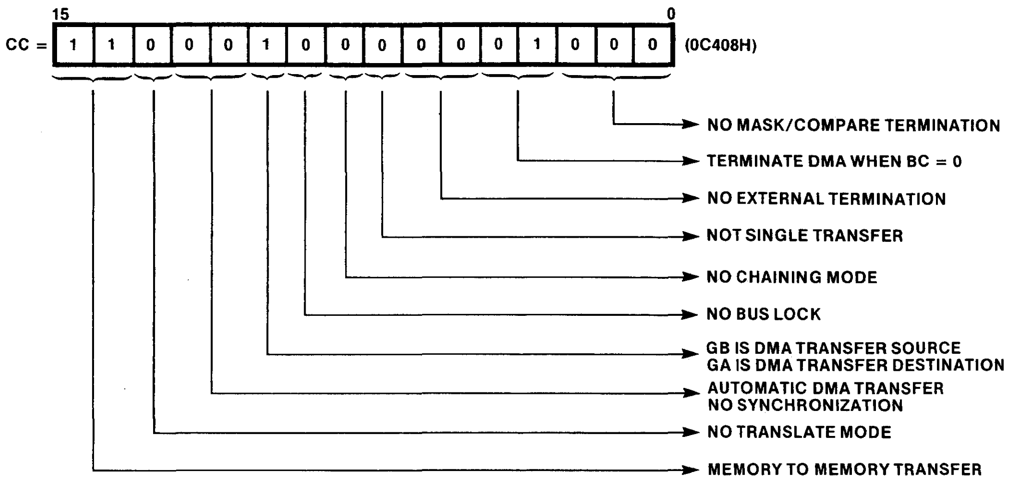


Figure 1-18. Stage Three—Channel 1 Begins Task Block Program Exection

Pointer/register GB is loaded with the 16-bit local space address of the DMA transfer source by the MOVI GB, SOURCE instruction. GB's tag bit is set to logical 1, indicating a 16-bit local IOP space address.

Register CC is loaded with DMA transfer control information by the MOVI CC, CHANNEL__CNTRL instruction. Register CC specifies



The DMA source and destination logical widths are specified by the WID 8, 8 instruction. (The IOP optimizes DMA transfers by data assembly/disassembly operations, depending on the WID instruction values and the source data address [odd or even].)

DMA transfer begins following the execution of the MOVI BC, BUFCNT instruction, the instruction following the XFER instruction. Data in local IOP memory is transferred to system memory, according to the DMA control parameters in register CC. When 128 bytes have been transferred, DMA transfer is terminated (byte count termination—register BC = 0) and task block program execution resumes at the HLT instruction.

Stage Four

Task block instruction execution has ended, following the execution of the task block program HLT instruction (see figure 1-19). The HLT instruction has cleared channel 1's BUSY flag byte to 00H, indicating that channel 1 is now inactive. The TP pointer/register contains the next sequential address following the HLT instruction.

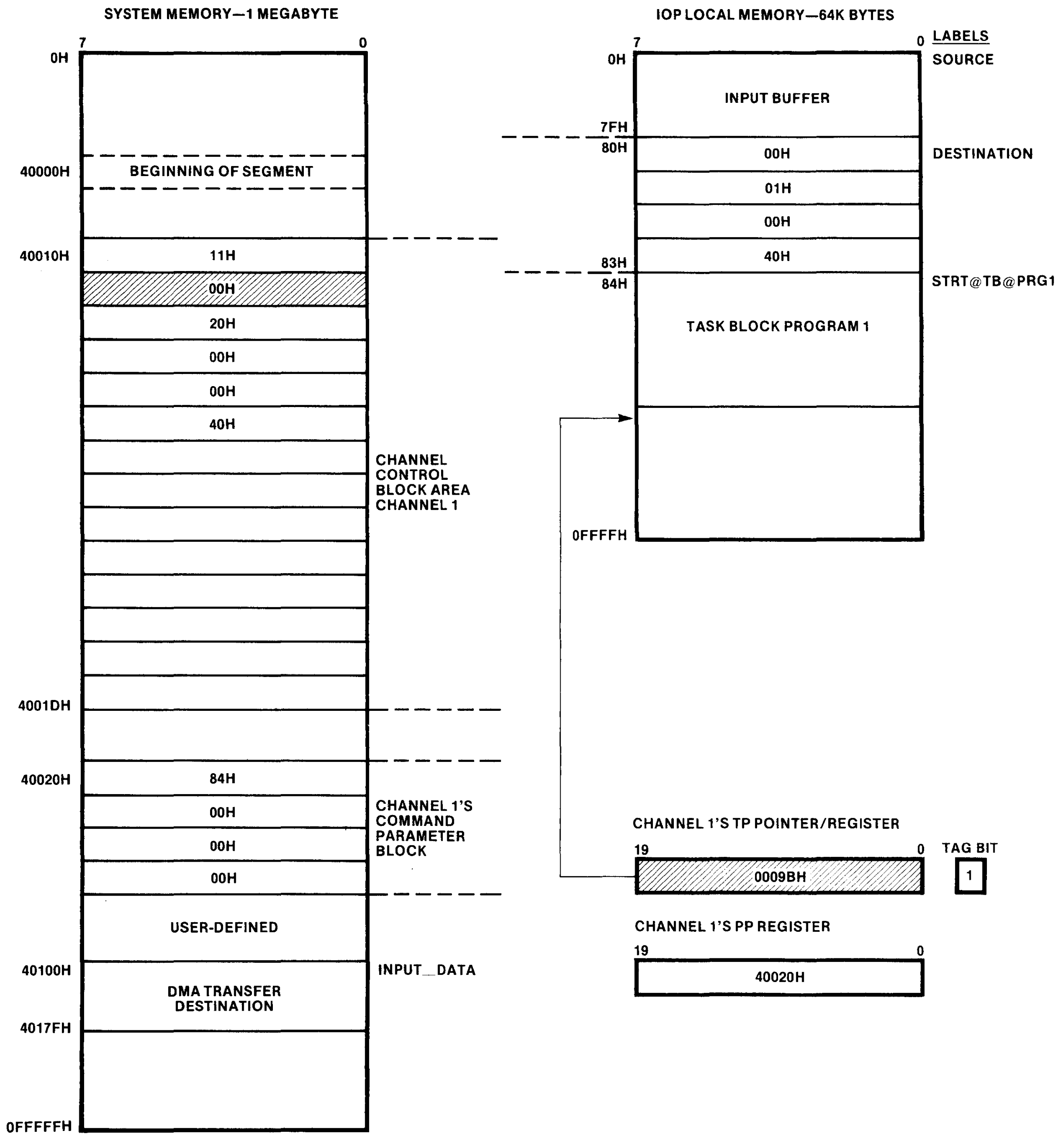


Figure 1-19. Stage Four—Task Block Program Execution Ended

CHAPTER 2
OPERANDS

Introduction

This chapter describes the types and forms of operands for assembly language instructions. Assembly language instructions are dealt with in Chapter Three, "The Instruction Set."

Most assembly language instructions require one or more operands. The most general form of these instructions is:

```
[LABEL]  OPERATION  OPERAND1, OPERAND2, OPERAND3  [COMMENT]
```

where 'OPERATION' is a specific processor activity and 'OPERAND1', 'OPERAND2' and 'OPERAND3' are the items that participate in the activity.

For those already acquainted with an assembly language a more familiar form is:

```
[LABEL]  MNEMONIC  OPERAND1, OPERAND2, OPERAND3  [COMMENT]
```

where mnemonic is the assembler defined symbolic name for some operation.

Suppose we wish to move an item of data from a register to a data memory location. Using the two-operand general form this is expressed as:

```
[LABEL]      MOVE          DATA MEMORY LOCATION,     MACHINE REGISTER
          (OPERATION)          (OPERAND1)               (OPERAND2)
```

or, (again for those familiar with an assembly language)

```
 MEM:        MOV            M,           R          ;Move register to memory
(LABEL)  (MNEMONIC)    (OPERAND1) (OPERAND2)   ;(COMMENT).
```

The mnemonic MOV is the assembler-recognized symbolic name for the operation we desire. M and R are symbols for Memory and Register. By convention the source item for a move is given as the rightmost operand and the destination of a move is given as the leftmost operand. This convention is followed throughout this assembly language.

Operand Overview

8089 machine instructions operate on various kinds of items. Table 2-1 summarizes these items and their associated operand types.

Table 2-1. Operand Types

ITEM	OPERAND TYPE	EXAMPLES
MACHINE REGISTERS	REGISTER	IX, MC, CC
MACHINE POINTER/REGISTERS	POINTER/REGISTER	GA, GB, GC
IMMEDIATE DATA VALUES	IMMEDIATE DATA	0FFH, ADTAB + 4
LOCATIONS WITHIN A PROGRAM	PROGRAM LOCATION	$ + 6, START
DATA IN MEMORY	DATA MEMORY	[GA], [GB].5
BITS OF MEMORY DATA	DATA MEMORY BIT	0, 1, 7

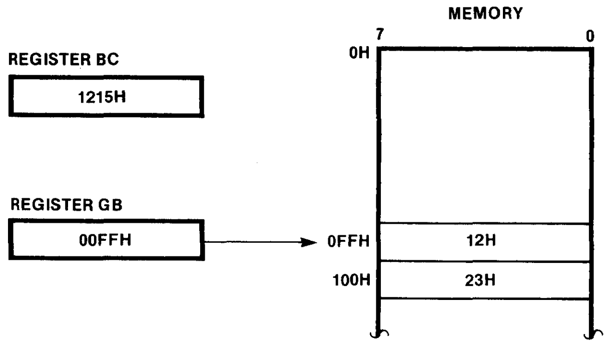
Most instructions require that one or more data items be supplied as operand(s). In the 8089 assembly language, this means that most operation mnemonics require one or more symbolic expressions as operands.

For example, to add the contents of a data memory location to a register we must specify the register and the data memory location—ADD IX, [GA]. Or, to logically AND a register with an immediate value we must again specify the the items to be operated on—AND GC, TOTAL. In these two examples IX, [GA], GC and TOTAL are assembly language instruction operands.

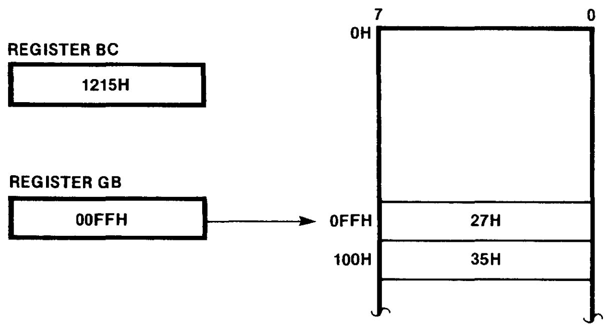
Examples:

1. Suppose we wish to add register BC, containing 1215H (1215 hexadecimal), to a word of data memory containing 2312H.

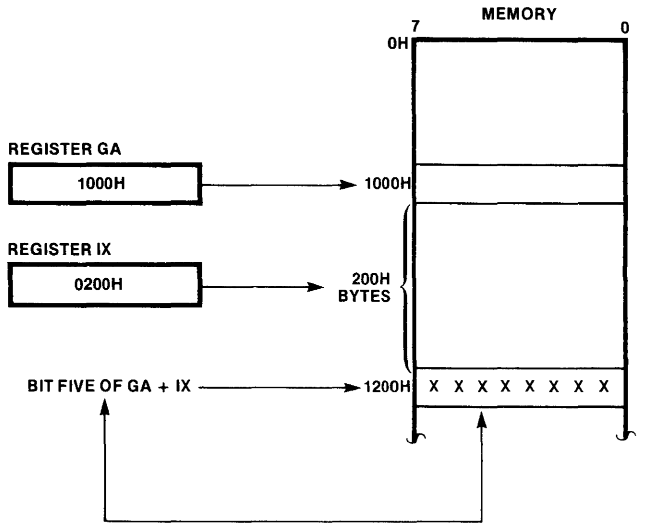
BC	is the assembly language symbol for register BC.
[GB]	is an assembly language expression for the word of data memory beginning at the address contained in pointer/register GB.



INSTRUCTION: ADD [GB], BC
OPERATION: [GB] ← 2312H + 1215H
RESULT:



2. The instruction JBT [GA+IX], 5, ERROR_ROUTINE tests bit five of the data memory byte located at GA + IX and jumps to the instruction labeled ERROR_ROUTINE if the bit is true (equal to logical one).



The remainder of this chapter deals with each operand type individually.

Register Operands

Register operands are a group of symbols recognized by the assembler which represent registers. These symbols are reserved and cannot be redefined. (For a complete list of reserved symbols see Appendix G).

The register operands are:

SYMBOL	REGISTER NAME	SYMBOL	REGISTER NAME
BC	Byte Count	GC	General Purpose C
CC	Channel Control	IX	Index Register
GA	General Purpose A	MC	Mask/Compare
GB	General Purpose B	TP	Task Pointer

PP also is a register symbol, representing the read-only, non-programmable Parameter Block Pointer Register. PP can be used only in data memory operands. (See DATA MEMORY operands later in this chapter).

Certain registers, as indicated by their names, play specific roles in IOP channel operations (see Chapter One and the *MCS-86 User's Manual,* order number 9800722).

Examples:

```
MOVI  MC, 7F00H          ;Move immediate value 7F00H to register MC.

OR    [GA], CC           ;Logically OR register CC to the word of data
                         ;memory beginning (low-order byte) at location
                         ;[GA].

JNZ   BC, REPEAT         ;Jump to program location labeled REPEAT if
                         ;register BC is not zero.
```

It is possible to assign another name to a register through the EQU assembler directive.

Example:

```
SOURCE    EQU   GA              ;Define symbol SOURCE for register
                                ;represented by GA.

          INC   SOURCE          ;Same as INC GA.
```

SOURCE may be used in the same contexts as GA.

Invalid uses of register operands:

```
BC:    DB    1AH                ;Attempts to redefine BC as the label of a data
                                ;memory byte location.

IX:    NOP                      ;Attempts to redefine IX as the label of an
                                ;assembly language instruction.

       JBT   MC, 5, TARGET      ;MC used in an invalid context (memory
                                ;operand required).

       MOVI  [GB], GA + 9       ;GA used in an expression, an invalid context.
```

Pointer/Register Operands

Pointer/register operands represent 20-bit registers and their associated tag bits. They are used to point to data memory and I/O space in a system. (For more detail on the use of pointer/registers see the section entitled "DATA MEMORY OPERANDS" in this chapter and also Chapter One.)

Pointer/registers can also be used as regular 16-bit registers, hence the inclusion of their assembler-recognized symbols under register operands in the previous section.

Pointer/registers are:

SYMBOL	NAME	SYMBOL	NAME
GA	General Purpose A	GC	General Purpose C
GB	General Purpose B	TP	Task Pointer

Like any register symbol, a pointer/register symbol is reserved and cannot be redefined. Also, the EQU assembler directive can be used to assign an alternate name to a pointer/register.

Examples:

```
MOVP  [PP].4, TP          ;Move 20-bit TP pointer/register and tag bit to
                          ;data memory.

LPDI  GA,  ADDR           ;Load pointer/register GA with 20-bit address
                          ;formed from four bytes of immediate data.

LPD   GC, [GB]            ;Load pointer/register GC with 20-bit address
                          ;formed from four bytes of data memory
                          ;beginning at location [GB].
```

Invalid uses of pointer/register operands:

```
GA:     DB  0E2H                    ;Attempts to redefine GA as the label of a data
                                    ;memory byte.

        JMP  GC                     ;Pointer/register operands not allowed in this
                                    ;context.

        MOVI  [GC], TP              ;Invalid context; TP not allowed in immediate
                                    ;data value expressions.
```

Immediate Data Operands

An immediate data operand is an expression representing:

- A data memory location

  Example:

```
DATA@TABLE:   DS    128        ;Reserve 128 bytes of data memory with the
                               ;first byte labeled DATA@TABLE.

MOVI  GB,  DATA@TABLE          ;Move the address of the first byte of data table
                               ;to pointer/register GB.
```

- A program location

  Example:

```
LPDI  TP, SUB1                 ;Load the TP pointer/register with the address
                               ;of the instruction labeled SUB1.
```

- An 8- or 16-bit value

  Example:

```
ORI  GB,  0D5BH                ;OR the contents of pointer/register GB with
                               ;the 16-bit immediate value 0D5BH.
```

Expressions

Expressions are composed of:

- symbols
- numeric constants
- character string constants of one or two characters
- the location counter reference ($)
- the assembly time operators + and −

Symbols

A symbol consists of 1 to 31 alphabetic, numeric or special characters, the first of which must be an alphabetic or special character. The special characters allowed in a symbol are:

? _ @

Symbols longer than 31 characters are truncated to 31 characters and flagged as errors.

VALID SYMBOLS	INVALID SYMBOLS	
INPUT?	INPUT/OUTPUT	"/" invalid special character.
INITIAL__VALUE	THIS ITEM	Embedded space is an invalid character.
POINTER__STORE	752__WILSON__STREET	Symbol cannot begin with a numeric.
ERROR__CODE	STEP__4.1	"." invalid special character.
ROUTINE@1	ANY__SET__OF__VALID__CHARACTERS__THIS__LONG	

Labels and Names

User-defined symbols are one of two types: labels or names. A symbol followed immediately by a colon (:) defines a label. These symbols are assigned the value of the assembler's location counter where they are defined. Labels normally appear in instruction or assembler directive source statements, but they can also appear alone, allowing the same location to be referenced by more than one symbolic name.

Examples:

```
LABEL1:

LABEL2:

LABEL3:    ADD   BC, [GA]            ;LABEL1, LABEL2, and LABEL3 all reference
                                     ;the same location.

START:     MOV   GA, [GB]            ;An instruction label.

DATA__T    DB    0FFH                ;An assembler directive label.
```

A name is defined by the appearance of a symbol, NOT followed by a colon, in the label-field of certain assembler directives. The value of the symbol depends on the assembler directive used.

Examples:

```
ELEVEN     EQU  11

IOP__CODE SEGMENT
```

Numeric Constants

A numeric constant can be specified in one of four number systems: Binary, Decimal, Hexadecimal or Octal. The first character of any numeric constant must be a decimal digit (0, 1, ... 9). The digit '0' is always acceptable for this purpose. Any number not specifically identified as binary, hexadecimal or octal is assumed to be decimal. Negative numbers appear in two's complement form.

Binary Constants	One or more binary digits (0, 1) followed immediately by the letter B.

```
ORBI    GA, 10110111B             ;OR GA with immediate binary value.

ADDBI   [GB], 11011110B           ;ADD immediate binary value to data memory
                                  ;byte at address specified by GB.
```

Decimal Constants	One or more decimal digits (0, 1, ... 9) optionally followed immediately by the letter D.

```
MOVI    BC, 30500                 ;Load register BC with immediate decimal
                                  ;value.

ANDI    CC, 17526D                ;AND register CC with immediate decimal
                                  ;value.
```

Hexadecimal Constants	One or more hexadecimal digits (0, 1, ... 9, A, B, C, D, E, F) followed immediately by the letter H. Note that the first digit must be a decimal digit (0, 1, ... 9).

```
ORI     GA, 0FEH                  ;OR register GA with immediate hexadecimal
                                  ;value.

MOVI    [GB+IX], 271FH            ;Move immediate hexadecimal value to a word
                                  ;of data memory beginning (low-order byte) at
                                  ;[GB+IX].
```

Octal Constants	One or more octal digits (0, 1, ... 7) followed immediately by the letter O or the letter Q.

```
ADDBI   [GA].7, 36O               ;ADD immediate octal value to data memory
                                  ;byte.

MOVI    CC, 1352Q                 ;Move immediate octal value to register CC.
```

The section in this chapter entitled "Permissible Range of Expression Values" describes the maximum numeric values allowed by the assembler.

Invalid Numeric Constants

```
01210B                            ;2 not a binary digit.

F712H                             ;First digit is not a decimal digit (0, 1, ...9).

1A7Q                              ;A is not an octal digit.

0F7                               ;F is not a decimal digit.
```

Character String Constants Containing One or Two Characters

A character string constant consists of one or more printable ASCII characters enclosed in single-quote marks ('). Each single-quote mark within a character string must be represented as two successive single-quote marks ('').

A character string constant consisting of only one or two characters can be used as a numeric constant in an expression.

Examples:

```
        ADDI  GB, 'Eh'          ;ADD immediate value 4568H to register GB.

        MOVI  [PP].7, '*'       ;Move immediate value 2AH to data memory.
```

A character string constant which contains more than two characters can only be used to define character string data with the DB assembler directive.

Location Counter Reference

Within an expression the current (at the beginning of the statement) value of the assembler's location counter can be referenced using the dollar sign ($) special character.

Example:

```
        MOVI  BC, 128           ;Load immediate value 128 (decimal) into
                                ;register BC.

LOOP:   MOV   GB, [GA]          ;Move 16-bits of data memory to register GB.

        DEC   BC                ;Decrement BC.

        JZ    BC, $ + 6         ;Jump around the unconditional jump if
                                ;register BC = 0.

        JMP   LOOP              ;Fall through to here if BC <> 0.

        LPD   GC, [PP].8        ;Instruction executed when BC = 0.
```

Assembly Time Operators

The following assembly time operations can be performed:

OPERATOR	OPERATION
+	Unary or binary addition.
−	Unary or binary subtraction.

The assembler sign-extends (bit 7) 8-bit values to 16-bits. Operations within expressions are performed on 16-bit quantities to yield a 16-bit result. Operators are executed in left to right order; they have equal precedence.

External symbols, which can only appear in expressions used in a DD assembler directive or an LPDI instruction, must be added (not subtracted) within the expression. Only one external symbol is allowed per expression.

Parentheses '( )' are NOT allowed in expressions.

Examples:

```
           EXTRN   OUT_MOD         ;Assembler directive indicating symbol
                                   ;OUT_MOD is defined in some other
                                   ;program.
DATA_1:    DB      7FH             ;Assembler directive defining symbol
                                   ;DATA_1 as the label of a data memory
                                   ;location: (the value of DATA_1 is not 07FH—
                                   ;it is the value of the assembler's location
                                   ;counter at the time DATA_1 is defined).
           LPDI    GB, OUT_MOD-7   ;Load pointer/register with immediate value.

           MOVI    BC, DATA_1 + 4  ;Load register with immediate value.
```

Invalid expressions using assembly time operators:

```
           EXTRN   RECD1           ;Identify RECD1 as a symbol defined in some
                                   ;program.
           LPDI    GB, 4—RECD1     ;External symbols cannot be subtracted within
                                   ;expressions.
           ADDI    MC, (MASK + 2)  ;Parentheses not allowed in expressions.
```

Permissible Range of Expression Values

Hexadecimal values can range from 0H to 0FFFFH or 0 to 65,535 decimal. Negative values are expressed in two's complement form.

All arithmetic operations are performed using two's complement arithmetic. Results are modulo 64K—the assembler performs no overflow detection.

Expressions used as immediate byte operands are evaluated modulo 256 (decimal 256 is equal to zero).

Examples:

```
           ADDI    GA, 65635       ;ADD an immediate word value of 99 or 63H
                                   ;(65635 modulo 64K) to register GA.
           MOVBI   [GC], -4        ;Move 0FCH (two's complement of 4) to data
                                   ;memory byte location specified by
                                   ;pointer/register GC.
           ORBI    CC, 0C7H        ;OR register CC with immediate byte value
                                   ;0C7H.
```

Examples of immediate data operands:

```
           ORBI    [GB], 11

           ADDBI   [GA+IX], TOTAL

           MOVI    BC, INPUT_CNT

           LPDI    GC, MAIN_MEM

           MOVBI   GA, STATUS + 5
```

Program Location Operands

Both conditional and unconditional control transfer instructions require a program location operand to specify the jump target. This operand is an expression (usually a label) representing the jump target's location in the program.

Locations within a program can be specified by three general types of expressions:

- an expression containing an instruction label
- an expression containing only numeric constants
- an expression containing a relative instruction address, i.e. one containing the location counter reference $

Instruction Labels

An instruction label is most commonly used to specify a jump target. In an expression, a label can be combined with an offset value to specify the jump target.

Examples:

```
TARGET:  MOV   GA, [GB]              ;An instruction labeled TARGET.

         JMP   TARGET                ;Unconditional jump to instruction with the
                                     ;label TARGET.

         JMCE  [GA].5, TARGET + 2    ;Conditional jump (mask/compare result equal
                                     ;to zero) to instruction following TARGET.

         JZ    BC, TARGET - 3        ;Conditional jump (register BC equals zero) to
                                     ;instruction 3 bytes before TARGET.
```

Numeric Constants

A numeric constant can be used to specify the jump target. This address is NOT an absolute address; it represents a displacement from the beginning of the (maximum) 64k program segment.

Examples:

```
         JMP   4004H                 ;Unconditional jump to the instruction located
                                     ;a displacement of 4004H from the beginning
                                     ;of the program segment.
```

Relative Instruction Addresses

A relative instruction address expresses the jump target relative to the control transfer instruction's address. The special character dollar sign ($), representing the value of the assembler's location counter at the beginning of the instruction, is used.

Example:

```
         JBT   [GB], 4, $ - 6        ;Conditional jump (bit four equal to a logical
                                     ;one) to the instruction six bytes before the
                                     ;beginning of this instruction.
```

Data Memory Operands

The contents of data memory are always addressed indirectly, that is, through a pointer/register (GA, GB, or GC) or the PP register. Both 20-bit system (memory) space and 16-bit local (I/O) space can be accessed.

When the IOP has its own remote bus (REMOTE configuration), the shared system bus is accessed using 20-bit addresses loaded into GA, GB or GC by the LPD or LPDI instructions. The pointer/register's tag bit is set to logical zero. In systems where the IOP shares the local bus with a host processor (LOCAL configuration), 20-bit addresses, again loaded through LPD or LPDI instructions, may be used to access data memory.

In REMOTE configurations, the IOP accesses its remote bus with 16-bit addresses loaded into GA, GB, or GC by the MOV, MOVB, MOVBI or MOVI instructions. The pointer/register's tag bit is set to logical one. In LOCAL configurations, these 16-bit addresses may be used to access I/O.

The 20-bit PP (parameter pointer) register contains the address of a channel's Command Parameter Block. This address always points to system (memory) space. It is loaded into the PP register automatically, whenever a channel is started. The contents of the register cannot be altered by a task block program. In data memory operands it is used to access the user-defined portions of the Command Parameter Block.

See Chapter One and the *MCS-86 User's Manual* for information on IOP system configurations.

Examples:

```
                LPD    GA, [PP].8          ;Load pointer/register GA with a 20-bit address
                                           ;formed from four bytes of the Command
                                           ;Parameter Block. GA's tag bit is set to logical
                                           ;zero.
                MOV    GC, [GB]            ;Move 16-bits of data memory from the address
                                           ;given by pointer/register GB to
                                           ;pointer/register GC. GC's tag bit is set to
                                           ;logical one.
                . . .
DATA_T:                DS     200          ;Define a label DATA_T, the beginning
                                           ;address of 200 bytes of reserved data memory.
                MOVI   GA, DATA_T          ;Load pointer/register GA with the 16-bit
                                           ;address of the reserved data memory bytes.
                                           ;GA's tag bit is set to logical one.
```

Data memory operands have four forms, as follows:

[PREG] (base address only) PREG can be the pointer/register GA, GB, GC or the PP register. PREG contains the data memory address.

```
MOV    CC, [GB]        ;Move 16-bits of data memory, beginning at the
                       ;address in GB, to register CC.

ADD    [GA], BC        ;Add register BC to the word of data memory
                       ;beginning (low-order byte) at location [GA].

ORB    [PP], MC        ;OR register MC to the first byte of the
                       ;Command Parameter Block.
```

[PREG].d (base address plus an unsigned 8-bit offset) d is an expression evaluated modulo 256 to form an 8-bit offset value. If d is greater than 255 an error message is issued by the assembler.

```
AND     MC, [GA].4      ;AND register MC with the word of data
                        ;memory beginning (low-order byte) at location
                        ;GA + 4.

NOT     [GC].4108       ;Complement the word of data memory
                        ;beginning (low-order byte) at location GC + 12
                        ;(4108 modulo 256). The assembler would flag
                        ;this instruction as an error since d is greater
                        ;than 255.
```

[PREG+IX] (base address plus the Index register) The data memory address is formed by adding the Index register and the base address. The base address and Index register are not changed.

```
MOV     [GB+IX], BC     ;Move register BC to data memory, low-order
                        ;byte at address GB + IX.

NOTB    [PP+IX]         ;Complement the byte PP + IX.
```

[PREG+IX+] (base address plus the Index register; the Index register is post auto-incremented by byte or word (1 or 2)) The data memory address is formed by adding the Index register and the base address. At the end of the instruction the Index register is automatically incremented by the size of the operand (one for byte operands, two for word operands). The base address is unchanged.

```
MOV   [GA], [GB+IX+]  ;Move a word of data memory, beginning at
                      ;GB + IX, to the word of data memory
                      ;beginning at GA. The Index register is post
                      ;auto-incremented by two (a word).

DEC   [GC+IX+]        ;Decrement the word of data memory
                      ;beginning at GC + IX. The Index register is
                      ;post auto-incremented by two (a word).

ORBI  [PP+IX+], 26    ;OR immediate byte value to a location within
                      ;the Command Parameter Block. The Index
                      ;register is post auto-incremented by one (a
                      ;byte).
```

Data Memory Bit Operands

Instructions that set and clear bits (SETB, CLR) or conditional jump instructions that test bits (JBT, JNBT) require operands that specify which bit of a data memory byte is accessed. A data memory bit operand provides this information.

The bits in a data memory byte are numbered, right to left, as follows:

```
MSB      LSB

XXXXXXXX
7 6 5 4 3 2 1 0
```

The bit number is the operand used in an instruction to specify the referenced bit.

Example:

```
D_MEM_BYTE:    DB  0FFH          ;Define a symbol D_MEM_BYTE as the label
                                 ;of a data memory byte with an initial value of
                                 ;0FFH.
```

The data memory byte at D_MEM_BYTE contains:

```
             7           0
            ┌─────────────┐
            │ 11111111    │
            └─────────────┘
```

```
MOVI    GA, D_MEM_BYTE           ;Load address of data memory byte into
                                 ;register GA.

CLR     [GA],5                   ;Clear bit five of the data memory byte
                                 ;located at GA.
```

The data memory byte at D_MEM_BYTE now contains:

```
             7           0
            ┌─────────────┐
            │ 11011111    │     (0DFH)
            └─────────────┘
```

CHAPTER 3
THE INSTRUCTION SET

Introduction

Most of this chapter is an alphabetized collection of instruction mnemonics. For each mnemonic, the coding format and operands of the instruction are given, along with symbolic and prose descriptions of the instruction's operation. An example of the use of each instruction and the format of the assembled instruction are also included. A fold-out page at the end of this chapter contains helpful operand and instruction decoding information.

In cases where the coding format of the operands makes a significant difference in the instruction's operation, separate listings are given for each coding format of the mnemonic. For example, the mnemonic ADDB has two listings: ADDB R, M and ADDB M, R.

The execution time, in clock timings, is listed for each instruction. One clock timing, as obtained from a 5 MHZ clock, is 200 nanoseconds. When 16 bits of data memory are used by an instruction, two execution times are given, reflecting the effect of bus size and odd/even data memory addresses on instruction execution times.

Instruction fetch time must be added to the given instruction execution time to determine the total time required to execute an instruction. Table 3-1 summarizes the instruction fetch times:

Table 3-1. 8089 Instruction Fetch Times (in clocks)

	2		3		4		5		← No. of bytes to be fetched
	Q	NO Q	Q	NO Q	Q	NO Q	Q	NO Q	← Is data in Queue?
E	/	14	/	18	/	22	/	26	Task Block Program on 8-bit bus
O	/	14	/	18	/	22	/	26	
E	/	7	/	14*	/	14	/	18*	Task Block Program on 16-bit bus
O	11*	14*	11	14	15*	18*	15	18	

↑ Even/odd starting boundary

*—Next byte loaded into Queue

The above reference to a queue refers to an internal one byte queue the IOP maintains to minimize instruction fetch time. For further details on IOP instruction fetching, see the *MCS-86 User's Manual,* order number 9800722.

A description of instruction source statements and assembled instruction formats as well as a breakdown of the instruction set by function precedes the instruction set encyclopedia.

Instruction Source Statement Format

The general format of an instruction source statement is:

```
[LABEL]    MNEMONIC    [OPERAND(S)]    [;COMMENT]
```

Items enclosed within brackets ([ ]) are optional. A label is never required but is optional on all instructions. Not all instructions require operands. A comment, any printable ASCII character(s) preceded by an unquoted semicolon (;), is optional on all source lines. All characters from the semicolon to the end of the line are ignored by the assembler but will appear in the assembly listing.

An instruction source statement is made up of one or more source lines terminated by an uncontinued end-of-line. A source line consists of zero or more characters terminated by an end-of-line, indicated by one of the following:

- CR a carriage return (0DH)
- LF a line-feed (0AH)
- CRLF a carriage return followed by a line-feed (0D0AH)

A source statement is continued by placing an ampersand (&) as the first character of the next source line. The sequence end-of-line& is treated like a blank by the assembler. Character string constants cannot be continued to the next source line.

The assembler compresses each source statement as follows: all comments and the final end-of-line are deleted; tabs, and all sequences of unquoted blanks, and end-of-line&'s are reduced to single blanks; all quoted quotes are changed into single quotes. The maximum number of characters in one compressed source statement is 256.

Examples:

```
        NOP

        HLT                     ;This is a comment.
BEGIN:  LPD  GA, [GB]           ;BEGIN: is a label
        MOV                     ;This source statement
&          GA,                  ;is made up of
&              [GC]             ;three source lines.
```

Assembled Instructions

Each 8089 instruction is at least two bytes in length. Up to three additional bytes can also be generated, specifying offset data, displacement, and immediate values. Figure 3-1 shows the general format of an assembled instruction.

If an offset value is used to specify a data memory address (AA field in low order assembled instruction byte = 01), an unsigned 8-bit offset field immediately follows the first two assembled instruction bytes:

```
7            0 7            0 7            0
| b/R/PWBAAW | OPCODEMM     | offset if AA=01 |
(low order byte)            (high order byte)
```

If the instruction source statement includes an immediate byte or word value, an 8- or 16-bit immediate value field follows the first two assembled instruction bytes and the offset field, if it is present:

```
7            0 7            0 7               0 7               0 7               0
| b/R/PWBAAW | OPCODEMM     | offset if AA=01 | i-value (low)   | i-value (high)  |
(low order byte)                                                  (high order byte)
```

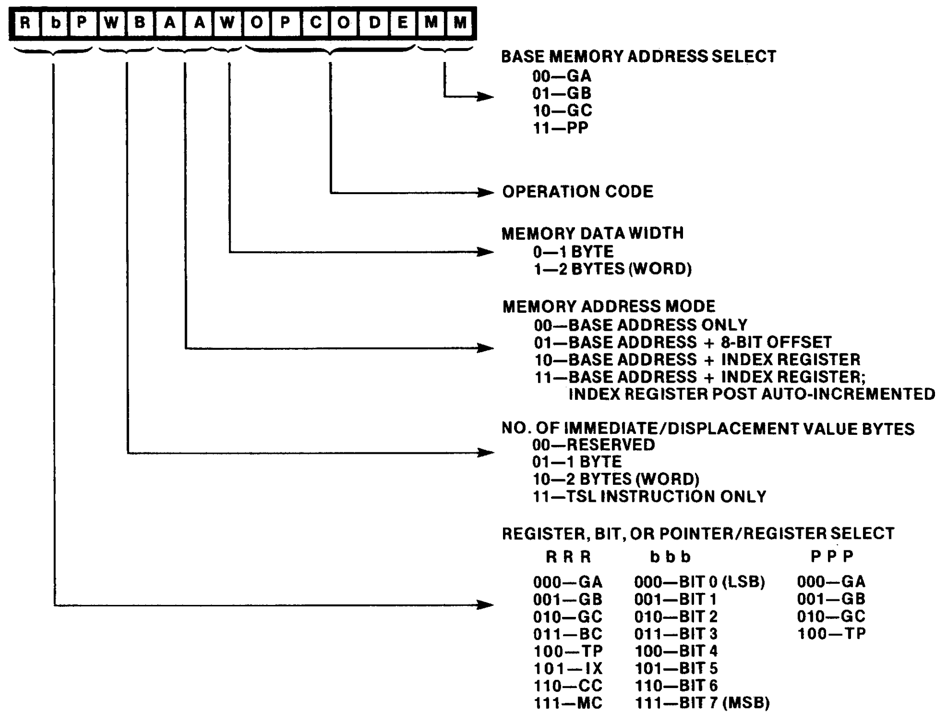


Figure 3-1. 8089 Assembled Instruction Format

Control transfer instructions have a signed one-or-two byte displacement value included in their assembled instructions. An 8- or 16-bit field containing the displacement value follows the first two bytes of the assembled instruction and the offset field if it is present:

7 0	7 0	7 0	7 0	7 0
b/R/PWBAAW	OPCODEMM	offset if AA=01	sdisp-low	sdisp-high
(low order byte)				(high order byte)

Two exceptions to the preceding rules for additional bytes in assembled instructions should be noted. The TSL instruction has an 8-bit immediate value field and an 8-bit signed displacement field. These two fields follow, in the given order, the first two bytes of the assembled instruction and the offset field, if it is present. (See the TSL instruction mnemonic description.)

The assembled instructions for memory to memory move operations are a minimum of four bytes in length. A maximum of six bytes can be generated by the assembler if two offset fields are present. (See the MOV and MOVB instruction mnemonic descriptions.)

Examples:

1. Figure 3-2 shows the assembled instruction ADD IX, [PP].24
2. Figure 3-3 shows the assembled instruction MOVI [GB].8, 4A27H

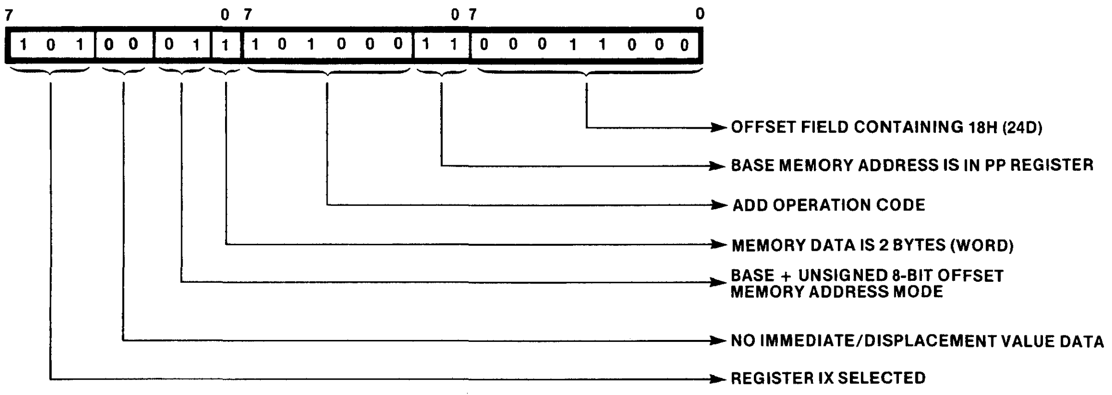

7 0 7 0 7 0
1 0 1 0 0 0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 0 0
OFFSET FIELD CONTAINING 18H (24D)
BASE MEMORY ADDRESS IS IN PP REGISTER
ADD OPERATION CODE
MEMORY DATA IS 2 BYTES (WORD)
BASE + UNSIGNED 8-BIT OFFSET MEMORY ADDRESS MODE
NO IMMEDIATE/DISPLACEMENT VALUE DATA
REGISTER IX SELECTED

Figure 3-2. Assembled Encoding of ADD IX, [PP].24

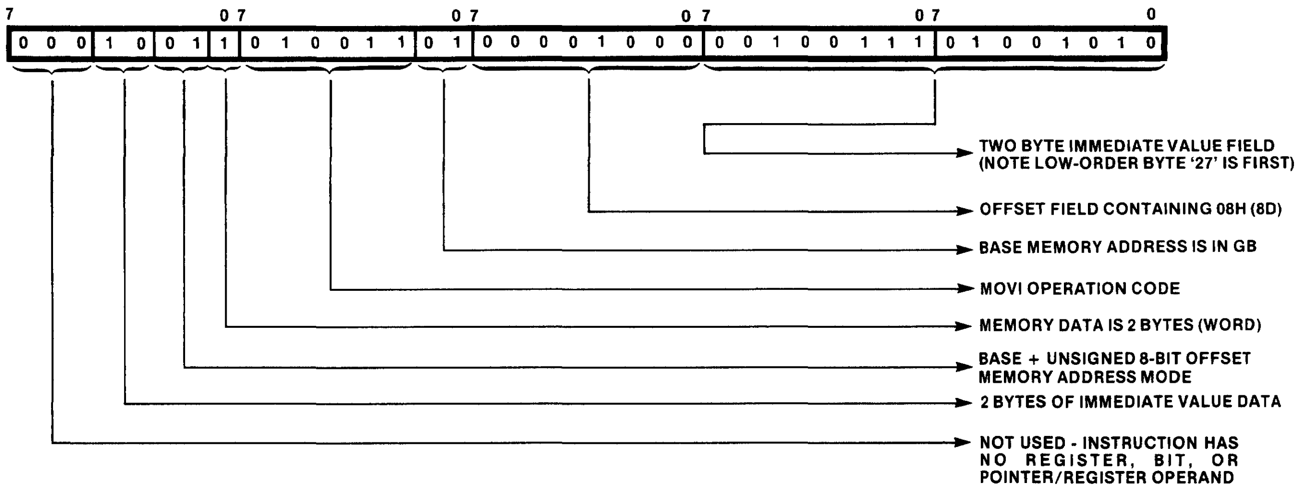

7 0 7 0 7 0 7 0 7 0
0 0 0 1 0 0 1 1 0 1 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 1 0 1 0 0 1 0 1 0
TWO BYTE IMMEDIATE VALUE FIELD (NOTE LOW-ORDER BYTE '27' IS FIRST)
OFFSET FIELD CONTAINING 08H (8D)
BASE MEMORY ADDRESS IS IN GB
MOVI OPERATION CODE
MEMORY DATA IS 2 BYTES (WORD)
BASE + UNSIGNED 8-BIT OFFSET MEMORY ADDRESS MODE
2 BYTES OF IMMEDIATE VALUE DATA
NOT USED - INSTRUCTION HAS NO REGISTER, BIT, OR POINTER/REGISTER OPERAND

Figure 3-3. Assembled Encoding of MOVI [GB].8, 4A27H

Instruction Mnemonics by Functional Group

The instruction mnemonics are described in this section in five functional groups:

Data Transfer

Control Transfer

Arithmetic and Logical

Bit Manipulation and Test

Special and Miscellaneous

Data Transfer Instructions

There are four distinct types of internal (excluding I/O operations) data transfer operations:

- Load/store 20-bit pointer/registers
- Load/store 16-bit registers
- Move immediate data to memory or register
- Move memory-to-memory

20-bit pointer/registers, GA, GB, GC or TP, can be loaded with 20-bit addresses by the LPD and LPDI instructions. LPD loads an address formed from four bytes of data memory; LPDI loads an address formed from four bytes of immediate data. An external symbol can appear in an LPDI instruction. Both of these instructions set the pointer/register's tag bit to logical zero.

A 20-bit pointer/register and its tag bit are stored in or restored from three bytes of data memory via the MOVP instruction. See the MOVP instruction mnemonic description later in this chapter for the format of a stored pointer/register and tag bit.

The 16-bit registers can be loaded with 8- or 16-bit data using the MOV, MOVB, MOVI, and MOVBI instructions. MOV and MOBV load a register from 16 and 8 bits of data memory respectively. MOVI loads a register with 16 bits of immediate data; MOVBI loads a register with 8 bits of immediate data. When a byte (memory or immediate) is loaded into a register, it is sign-extended (bit 7) into the high order byte.

MOV is used to store 16-bit registers in data memory. The MOVB instruction stores the low order byte of a register in data memory.

NOTE

20-bit pointer/registers can be used as registers in the MOV, MOVB, MOVI, and MOVBI instructions. The sign bit (bit 15 or bit 7) is sign- extended into the high order bits. The pointer/register's tag bit is set to logical one by these instructions.

Memory data or immediate data can be moved to a memory location using the MOV, MOVB, MOVI and MOVBI instructions. The assembled instruction for MOV and MOVB in this case is at least four bytes long.

MNEMONIC	OPERATION
LPD	Load 20-bit pointer/register from data memory
LPDI	Load 20-bit pointer/register from immediate data
MOVP	Move 20-bit pointer/register to (store) or from (restore) memory
MOV	Move 16-bits of data memory to/from data memory or register
MOVB	Move 8-bits of data memory to/from data memory or register
MOVI	Move 16-bits of immediate data to data memory or register
MOVBI	Move 8-bits of immediate data to data memory or register

Control Transfer Instructions

Call and jump instructions alter the normal sequential execution of task block program instructions and transfer control to another, non-sequential instruction within the program. This instruction is called the jump target. One operand within a control transfer instruction is an expression specifying the location of the jump target.

Displacements

Jumps are made by adding a signed byte or word displacement value (sign-extended to 20 bits) to the 20-bit TP pointer/register to form the jump target address. Jump targets within −128, +127 bytes of the end of a control transfer instruction can be reached with a signed byte displacement value. Jump targets within −32,768, +32,767 bytes of the end of a control transfer instruction require a signed word displacement value.

All jump targets must be within a −32,768, +32,767 byte range of the end of a control transfer instruction. There is NO wraparound from the end of the (maximum) 64k program instruction space to the beginning. Figure 3-4 shows the range of jump target locations for signed byte and signed word displacement values.

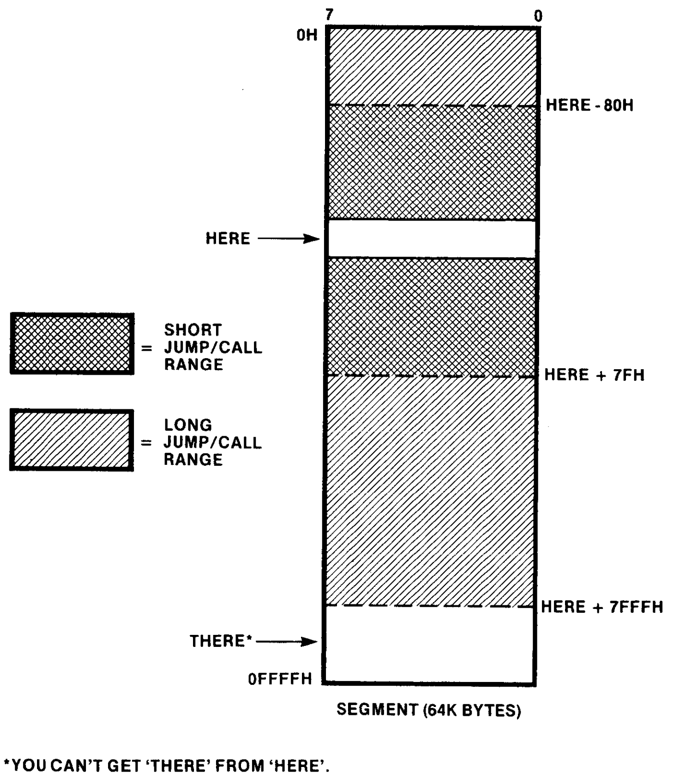


Figure 3-4. Control Transfer Jump Target Range

Short and Long

Control transfer instruction mnemonics have two forms: a short form and a long form. The long form is constructed by adding an 'L' prefix to the short form of the control transfer instruction mnemonic.

Examples:

SHORT	LONG
CALL	LCALL
JBT	LJBT
JMP	LJMP

When the short form of a control transfer instruction mnemonic is coded, the assembler generates a signed byte *or* word displacement value. If the expression specifying the jump target contains only symbols previously defined to the assembler (this includes the special character $, the location counter reference), the minimum size displacement value necessary to reach the jump target is generated.

The long form of a control transfer instruction mnemonic always generates a signed word displacement value, regardless of the actual distance to the jump target.

Short Form Errors

If the short form of a control transfer instruction mnemonic is coded and the jump target address cannot be determined by the assembler on its first pass (i.e., the expression specifying the jump target contains a forward reference), a signed byte displacement value is assumed to be sufficient. If later the assembler determines that a signed word displacement is necessary, the short form instruction will be flagged as an error. The long form of the instruction mnemonic must be coded in its place.

Examples:

```
J_TARGET: MOV  [GA].4, [PP].12      ;An instruction labeled J_TARGET.
   .        (200 bytes of assembled source program)
   .
   .
          JMP  J_TARGET             ;The address of the jump target J_TARGET
                                    ;can be determined by the assembler on its
                                    ;first pass. A signed word displacement value
                                    ;is generated by the assembler.
   .
   .
   .
          JZ   [GB], $ + 16         ;$ + 16 is NOT a forward reference. The
                                    ;expression specifying the jump target
                                    ;contains only symbols defined to the
                                    ;assembler when the JZ instruction is
                                    ;processed on its first pass. A signed byte
                                    ;displacement value is generated.
   .
   .
   .
          CALL [GC].4, SUB_RT       ;A short CALL instruction whose jump target
                                    ;SUB_RT is not yet defined to the assembler
                                    ;on its first pass.
   .        (200 bytes of assembled source program)
   .
   .
SUB_RT:   ADDI MC, 722H             ;The CALL instruction's jump target.
```

The above CALL instruction will be flagged as an error by the assembler, having determined that the jump target requires a signed word displacement value rather than the signed byte displacement value it assumed. An LCALL will have to be coded in place of the CALL mnemonic.

Unconditional Control Transfer Instructions:

MNEMONIC	OPERATION
CALL / LCALL	Store TP pointer/register and tag bit; Jump
JMP / LJMP	Jump

Conditional Control Transfer Instructions:

MNEMONIC	OPERATION
JMCE / LJMCE	Jump on mask/compare equal
JMCNE / LJMCNE	Jump on mask/compare not equal
JNZ / LJNZ	Jump on nonzero register or data memory word
JNZB / LJNZB	Jump on nonzero data memory byte
JZ / LJZ	Jump on zero register or data memory word
JZB / LJZB	Jump on zero data memory byte

Arithmetic and Logical Instructions

Arithmetic and logical operations can be performed on registers and 8- or 16-bit data. The ADDB, ADDBI, ANDB, ANDBI, ORB, and ORBI instructions operate on registers and 8-bit memory or immediate data. DECB, INCB, and NOTB operate on 8-bit memory data only.

All 8-bit immediate or memory data is sign-extended to 16-bits in arithmetic and logical operations. It cannot be assumed that the high order byte of a register is unaffected by an 8-bit operation.

Example:

Register MC contains 8351H:

```
7          0 7          0
| 10000011 | 01010001 |
```

The following instruction is executed:

```
ANDBI   MC, 47H                    ;The immediate byte data is sign-extended
                                   ;(bit 7) to 16-bits. The 16-bit result of the AND
                                   ;operation is placed in register MC.
```

Register MC now contains 41H (not 8341H).

```
7          0 7          0
| 00000000 | 01000001 |
```

To preserve the high order byte of the MC register the 16-bit form of the instruction, ANDI, must be used: ANDI MC, 0FF41H.

The instructions ADD, ADDI, AND, ANDI, DEC, INC, OR, ORI, and NOT operate on registers and 16-bit memory or immediate data.

When 20-bit pointer/registers are used as registers in arithmetic and logical operations, bit 15 of 16-bit quantities and bit 7 of 8-bit quantities are sign-extended into the high-order bits. The upper four bits (bits 16-19) of a pointer/register are undefined following all arithmetic and logical operations except addition. ADD, ADDI, ADDB, ADDBI can carry into the high order bits of a pointer/register.

Example:

Pointer/register GA contains 2E200H. The following instruction adds 32,765 (decimal) to pointer/register GA:

```
ADDI  GA,  32765
```

Pointer/register GA now contains 361FDH.

MNEMONIC	OPERATION
ADD	ADD register and 16-bit memory data
ADDB	ADD register and 8-bit memory data
ADDBI	ADD register or 8-bit memory data and 8-bit immediate data
ADDI	ADD register or 16-bit memory data and 16-bit immediate data
AND	AND register with 16-bit memory data
ANDB	AND register with 8-bit memory data
ANDBI	AND register or 8-bit memory data with 8-bit immediate data
ANDI	AND register or 16-bit memory data with 16-bit immediate data
DEC	Decrement register or 16-bit memory data
DECB	Decrement 8-bit memory data
INC	Increment register or 16-bit memory data
INCB	Increment 8-bit memory data
OR	OR register and 16-bit memory data
ORB	OR register and 8-bit memory data
ORBI	OR register or 8-bit memory data with 8-bit immediate data
ORI	OR register or 16-bit memory data with 16-bit immediate data
NOT	Complement register or 16-bit memory data
NOTB	Complement 8-bit memory data

Bit Manipulation and Test Instructions

These instructions clear, set, or test a particular data memory bit.

The result of a bit test determines whether or not a jump occurs to some other instruction within the task block program. The bit test instructions require three operands: a data memory operand specifying the address of the data memory byte in which the bit to be tested is located; a data memory bit operand specifying the bit to be tested; and a program location operand specifying the jump target. Bit test instructions, since they are control transfer instructions, have both a short and long form. (See "Control Transfer Instructions" in this chapter for more on short and long control transfer instructions.)

Examples:

```
JBT   [GA].4, 3, TARGET              ;Test bit three of the data memory byte at
                                     ;GA + 4. Jump to the instruction labeled
                                     ;TARGET if the tested bit equals a logical
                                     ;one.

LJNBT   [GC+IX], 0, ERROR_FIX        ;Test bit zero of the data memory byte at
                                     ;GC + IX. Jump to the instruction labeled
                                     ;ERROR_FIX if the tested bit does not
                                     ;equal a logical one.
```

MNEMONIC	OPERATION
SETB	Set selected data memory bit to logical one
CLR	Clear selected data memory bit to logical zero
JBT / LJBT	Jump on data memory bit true (bit = logical one)
JNBT / LJNBT	Jump on data memory bit not true (bit < > logical one)

Special and Miscellaneous Instructions

This group contains those instructions that specifically pertain to I/O processing by the 8089. It also includes the NOP (no operation) instruction.

A full understanding of the use of the special IOP instructions requires a knowledge of 8089 operation. The *MCS-86 User's Manual* is the best source for such information. The operation of each of these instructions is explained under its mnemonic in the following encyclopedia.

MNEMONIC	OPERATION
HLT	END task block program instruction execution.
NOP	No operation.
SINTR	Set interrupt service flip flop.
TSL	Test and set data memory byte while system bus is locked.
WID	Set DMA source and destination logical widths.
XFER	Begin DMA transfer following the execution of the next instruction.

Add Memory Word to Register

Add Register to Memory Word

Mnemonic: ADD

Coding Format: ADD R, M
ADD M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (OP1) ← (OP1) + (OP2)

A word of data memory, with low order byte at location 'M', is added to the contents of register 'R'. The 16-bit result is placed in the leftmost operand, 'OP1'.

If 'OP1' is a 20-bit pointer/register (GA, GB, GC or TP) the memory data is sign-extended (bit 15) to 20-bits. A carry can occur into the upper bits, bits 16-19, of the pointer/register.

Examples:

```
ADD  GA, [GB]     ;Register GB points to the first (low order) byte of the word of
                  ;memory data which is added to the contents of register GA.

ADD  [GC], IX     ;The contents of the Index register are added to the word of
                  ;memory data which begins at the address contained in
                  ;register GC.
```

Assembled Instruction:

ADD R, M (ADD TO REGISTER FROM MEMORY WORD)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 1 0 0 0 M M	offset if AA=01

Execution Time:

11 clocks bus width = 16 bits and address is even
15 clocks bus width = 8 bits or bus width = 16 bits and address is odd

ADD M, R (ADD TO MEMORY WORD FROM REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 1	1 1 0 1 0 0 M M	offset if AA=01

Execution Time:

16 clocks bus width = 16 bits and address is even
26 clocks bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

ADDB R, M

Add Memory Byte to Register

Mnemonic: ADDB **Coding Format:** ADDB R, M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (R) ← (R) + sign-extended (M)
two 16-bit operands; 16-bit result

The data memory byte at location 'M' is sign extended (bit 7) to a 16-bit quantity and added to the register, 'R'. The 16-bit result is placed in register 'R'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP) the memory data is sign- extended (bit 7) to 20-bits. A carry can occur into the upper bits, bits 16-19, of the pointer/register.

Example:

```
ADDB   GA, [GB]          ;Add byte at [GB] to register GA.
```

Assembled Instruction:

ADDB R, M (ADD TO REGISTER FROM MEMORY BYTE)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 1 0 0 0 M M	offset if AA=01

Execution Time:

11 clocks

NOTE 1) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

Add Register to Memory Byte

Mnemonic: ADDB **Coding Format:** ADDB M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (M) ← (M) + low-order byte (R)

The data memory byte at location 'M' is added to the low-order byte of register 'R'. The 8-bit result is placed in data memory at location 'M'.

Examples:

```
SOME__OFFSET   EQU  5H
               ADDB  [GC].SOME__OFFSET, BC     ;Add the low-order byte of
                                               ;register BC to data memory
                                               ;byte at [GC] + 5. The 8-bit
                                               ;result is placed in [GC] + 5.
```

Assembled Instruction:

ADDB M, R (ADD TO MEMORY BYTE FROM REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 1	1 1 0 1 0 0 M M	offset if AA=01

Execution Time:

16 clocks

ADDBI R, I

Add Immediate Byte to Register

Mnemonic: ADDBI **Coding Format:** ADDBI R, I
Operands: 'R' is a register symbol
'I' is an expression evaluated modulo 256

Operation: (R) ← (R) + sign-extended (i-value)
two 16-bit operands; 16-bit result

An immediate byte value is sign extended (bit 7) to a 16-bit quantity and added to the contents of the register, 'R'. The 16-bit result is placed in register, 'R'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP) the immediate value is sign-extended (bit 7) to 20-bits. A carry can occur into the upper bits, bits 16-19, of the pointer/register.

Example:

```
ADDBI  BC,  37      ;The immediate value '37' (decimal) is added to register BC.
```

Assembled Instruction:

ADDBI R, I (ADD IMMEDIATE BYTE TO REGISTER)

7 0	7 0	7 0
RRR01000	00100000	i-value

Execution Time:

3 clocks

NOTE 1) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

ADDBI M, I

Add Immediate Byte to Memory Byte

Mnemonic: ADDBI **Coding Format:** ADDBI M, I

Operands: 'M' is a data memory expression
'I' is an expression evaluated modulo 256

Operation: (M) ← (M) + i-value

The expression 'I' is evaluated modulo 256 to an immediate signed byte, 'i-value'. This immediate signed byte value is added to the data memory byte at location 'M'. The result is placed in the data memory location 'M'.

Example:

```
ADDBI  [GC], 45H     ;The immediate value '45H' is added to the memory byte at [GC].
```

Assembled Instruction:

ADDBI M, I (ADD IMMEDIATE BYTE TO MEMORY BYTE)

7 0	7 0	7 0	7 0
00001AA0	110000MM	offset if AA=01	i-value

Execution Time:

16 clocks

ADDI

Add Immediate Word to Register

Add Immediate Word to Memory Word

Mnemonic: ADDI

Coding Format: ADDI R, I
ADDI M, I

Operands: 'R' is a register symbol
'M' is a data memory expression
'I' is an expression evaluated modulo 64k

Operation: (OP1) ← (OP1) + i-value

The expression 'I' is evaluated modulo 64k to an immediate signed word value, 'i-value'. This immediate word value is added to the contents of register, 'R', or the word (16 bits) of memory data whose low order byte is located at 'M'. The result is placed in the specified register or memory location, 'OP1'.

If 'OP1' is a 20-bit pointer/register (GA, GB, GC or TP) the immediate value is sign-extended to 20-bits. A carry can occur into the upper bits, bits 16-19, of the pointer/register.

Examples:

```
ADDI  GA,  7F09H       ;The immediate word value '7F09H' is added to the contents of
                       ;register GA.

ADDI  [GB],  57421Q    ;The immediate word value '57421' (Octal) is added to the word
                       ;of memory whose low order byte is at the address contained
                       ;in register GB.
```

Assembled Instruction:

ADDI R, I (ADD IMMEDIATE WORD TO REGISTER)

7 0	7 0	7 0	7 0
RRR10001	00100000	i-value (low)	i-value (high)

Execution Time:

3 clocks

ADDI M, I (ADD IMMEDIATE WORD TO MEMORY WORD)

7 0	7 0	7 0	7 0	7 0
00010AA1	110000MM	offset if AA=01	i-value (low)	i-value (high)

Execution Time:

16 clocks bus width = 16 bits and address is even
26 clocks bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

And Register With Memory Word

And Memory Word With Register

Mnemonic: AND

Coding Format: AND R, M
AND M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (OP1) ← (OP1) AND (OP2)

A word, low order byte at location 'M', is fetched from data memory and logically ANDed with the specified register, 'R'. A logical AND returns a logical '1' in each bit position where both input bits are a logical '1'. Otherwise a logical '0' is returned. The result is placed in the leftmost operand, 'OP1'.

If a 20-bit pointer/register (GA, GB, GC or TP) is used as an operand in this instruction the upper four bits, bits 16-19, are undefined following instruction execution.

Example:

```
AND   GA, [GB+IX]      ;The Index register is added to register GB, forming the
                       ;address of the first (low order) byte of a word of data memory
                       ;which is ANDed with register GA. The result is placed
                       ;in register GA.
```

Assembled Instruction:

AND R, M (AND REGISTER WITH MEMORY WORD)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 1 0 1 0 M M	offset if AA=01

Execution Time:

11 clocks bus width = 16 bits and address is even
15 clocks bus width = 8 bits or bus width = 16 bits and address is odd

AND M, R (AND MEMORY WORD WITH REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 1	1 1 0 1 1 0 M M	offset if AA=01

Execution Time:

16 clocks bus width = 16 bits and address is even
26 clocks bus width = 8 bits or bus width = 16 and address is odd

NOTES 1) A logical AND of two operands examines their corresponding bit positions and returns a logical '1' if both bits are a logical '1'. A logical '0' is returned otherwise.

```
Example:  AND  0101 1110 (5EH) with 0110 0110 (66H)
               0101  1110
          AND  0110  0110
       Result  0100  0110 (46H)
```

2) See ANDB instruction on following page for logical AND with byte data.

3) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

ANDB R,M

And Memory Byte to Register

Mnemonic: ANDB **Coding Format:** ANDB R, M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: 1) The data memory byte located at 'M' is sign-extended to 16-bits

2) (R) ← (R) AND sign-extended (M)
two 16-bit quantities

A byte is fetched from data memory location 'M' and sign-extended (bit 7) to 16 bits. The sign-extended byte is logically ANDed with the register, 'R'. In each bit position a logical '1' is returned if both input bits are a logical '1'. Otherwise, a logical '0' is returned. The result is placed in the register 'R'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP) its upper four bits, bits 16-19, are undefined following instruction execution.

Examples:

```
ANDB   BC, [GA]          ;The data memory byte at location [GA] is ANDed with the
                         ;contents of register BC. The result is placed in register BC.
```

Assembled Instruction:

ANDB R, M (AND MEMORY BYTE TO REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 0	1 0 1 0 1 0 M M	offset if AA=01

Execution Time:

11 clocks

NOTES 1) A logical AND of two operands compares each of their corresponding bit positions and returns a logical '1' if both bits are a logical '1'. A logical '0' is returned otherwise.

```
Example:  AND   1101 1010 (0DAH) with 0111 1010 (7AH)
                1101   1010
          AND   0111   1010
       Result   0101   1010 (5AH)
```

2) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

ANDB M, R

And Register to Memory Byte

Mnemonic: ANDB M, R   **Coding Format:** ANDB M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (M) ← (M) AND low-order byte (R)

A byte is fetched from data memory location 'M' and logically ANDed with the low-order byte of register 'R'. In each bit position, a logical '1' is returned if both input bits are a logical '1'. Otherwise, a logical '0' is returned.

The 8-bit result is placed in data memory location 'M'.

Example:

```
ANDB   [GA], GC          ;The data memory byte at [GA] is ANDed with the low-order
                         ;byte of register GC. The 8-bit result is placed in the data
                         ;memory location [GA].
```

Assembled Instruction:

ANDB M, R (AND REGISTER TO MEMORY BYTE)

7 0	7 0	7 0
R R R 0 0 A A 1	1 1 0 1 1 0 M M	offset if AA=01

Execution Time:

16 clocks

NOTE 1) A logical AND of two operands compares compares each of their corresponding bit positons and returns a logical '1' if both bits are a logical '1'. A logical '0' is returned otherwise.

```
Example:  AND   0010 1010 (2AH) with 1111 0001 (0F1H)
                0010  1010
          AND   1111  0001
       Result   0010  0000 (20H)
```

ANDBI R, I

And Immediate Byte to Register

Mnemonic: ANDBI **Coding Format:** ANDBI R, I

Operands: 'R' is a register symbol
'I' is an expression evaluated modulo 256

Operation: (R) ← (R) AND sign-extended (i-value)
two 16-bit quantities; a 16-bit result

The expression 'I' is evaluated modulo 256 to an immediate signed byte value, 'i-value'. This immediate signed byte value is sign-extended (bit 7) to 16-bits and ANDed with register 'R'. A logical one is output where each input bit is a logical one. A logical zero is output otherwise. The 16-bit result is placed in register 'R'.

If 'R' is a 20-bit pointer/register (GA,GB, GC or TP) the upper four bits, bits 16-19, are undefined following instruction execution.

Example:

```
ANDBI  IX,  0FDH        ;The contents of register IX are ANDed with the immediate byte
                        ;value '0FDH'. The 16-bit result is placed in register IX.
```

Assembled Instruction:

ANDBI R, I (AND IMMEDIATE BYTE TO REGISTER)

7 0	7 0	7 0
RRR01000	00101000	i-value

Execution Time:

3 clocks

NOTE 1) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

ANDBI M, I

And Immediate Byte to Memory Byte

Mnemonic: ANDBI **Coding Format:** ANDBI M, I

Operands: 'M' is a data memory operand
'I' is an expression evaluated modulo 256

Operation: (M) ← (M) AND (i-value)

The expression 'I' is evaluated modulo 256 to an immediate signed byte value, 'i-value'. The data memory byte at location 'M' is ANDed with the immediate signed byte value. A logical one is output when both input bits are a logical one. Otherwise a logical zero is output. The result is placed in the data memory location 'M'.

Example:

```
ANDBI  [GB], 73H        ;The data memory byte at location [GB] is ANDed with the
                        ;immediate byte value 73H.
```

Assembled Instruction:

ANDBI M, I (AND IMMEDIATE BYTE TO MEMORY BYTE)

7 0	7 0	7 0	7 0
00001AA0	110010MM	offset if AA=01	i-value

Execution Time:

16 clocks

And Immediate Word to Register

And Immediate Word to Memory Word

Mnemonic: ANDI

Coding Format: ANDI R, I
ANDI M, I

Operands: 'R' is a register symbol
'M' is a data memory operand
'I' is an expression evaluated modulo 64k

Operation: (OP1) ← (OP1) AND i-value

The expression 'I' is evaluated modulo 64k to an immediate signed word value, 'i-value'. The immediate word value is ANDed with the contents of the specified register 'R', or the word of data memory whose low order byte is located at 'M'. A logical '1' is returned in each bit position where both input bits are a logical '1'. Otherwise, a logical '0' is returned. The result is returned to the leftmost operand, 'OP1'.

If 'OP1' is a 20-bit pointer/register (GA, GB, GC or TP) the upper four bits, bits 16-19, are undefined following instruction execution.

Examples:

```
ANDI  CC,  0FFF7H        ;The contents of register CC are ANDed with the immediate
                         ;word value '0FFF7H'. The result is placed in register CC.

ANDI [GA], 2222H         ;The word of data memory whose low order byte is pointed to
                         ;by register GA is ANDed with the immediate word value
                         ;'2222H'. The result is placed in two bytes of data memory
                         ;beginning at the given memory location. The low order byte of
                         ;the result is placed in the first memory byte; the high order
                         ;byte is placed in the second.
```

Assembled Instruction:

ANDI R, I (AND REGISTER WITH IMMEDIATE WORD)

7 0	7 0	7 0	7 0
R R R 1 0 0 0 1	0 0 1 0 1 0 0 0	i-value (low)	i-value (high)

Execution Time:

3 clocks

ANDI M, I (AND MEMORY WORD WITH IMMEDIATE WORD)

7 0	7 0	7 0	7 0	7 0
0 0 0 1 0 A A 1	1 1 0 0 1 0 M M	offset if AA=01	i-value (low)	i-value (high)

Execution Time:

16 clocks bus width = 16 bits and address is even
26 clocks bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) When the results of an arithmetic or logic operation are placed in a 20-bit pointer/register the upper four bits, bits 16-19, are undefined following the operation, except when addition is performed. In this case, there can be a carry into the upper four bits of the pointer/register.

Call

Mnemonic: CALL **Coding Format:** CALL M, L

Operands: 'L' is an expression representing the jump target
'M' is a data memory expression

Operation: 1) (M) ← (TP) + tag bit

2) (TP) ← (TP) + sdisp

The TP pointer/register, which contains the address of the next sequential instruction following the CALL instruction, and its tag bit, indicating a system or local space task block program, are saved in 3 bytes of data memory beginning at location, 'M'. (See Note 4 below for the format of the stored 20-bit TP pointer/register and tag bit.)

'L' is the jump target, a location within the program. If the address of the jump target can be determined when the assembler processes this instruction on its first pass, a signed byte (−128, +127) or word (−32,768, +32,767) value, 'sdisp', the distance, in bytes, from the end of the CALL instruction to the jump target, is generated. If the address cannot be determined on the first pass (as is the case when 'L' contains a forward reference) the assembler generates a one byte displacement-field, assuming that the jump target address, resolved in a subsequent pass, is within a −128, +127 byte displacement from the end of the instruction (see Note 1 below).

The signed displacement, 'sdisp' is added to the TP pointer/register, which contains the address of the next sequential instruction (the stored TP pointer/register value), to form the jump target address.

Examples:

Suppose the following source lines were assembled:

```
J__TARGET:  MOVI   MC, 1279H
```

. . . (source lines resulting in 191 bytes of object code)

```
            CALL   [PP].12, J__TARGET
```

The address of the jump target, 'J__TARGET', has been determined by the assembler when the 'CALL' instruction is found on its first pass. A displacement outside a range of −128, +127 bytes is required to reach the jump target, so a signed word displacement value is generated, the distance from the end of the 'CALL' instruction to the jump target. In this case the signed word displacement value would be −200, 0FF38H, since the 'CALL' instruction is 5 bytes in length: two bytes followed by a byte containing the address offset value 12, 0CH, followed by the two byte signed displacement value.

The assembled instruction bytes would be: 939F 0C 38FF:

7 0	7 0	7 0	7 0	7 0
10010011	10011111	00001100	00111000	11111111

low order byte high order byte

Note that the low order byte of the signed word displacement value, 38H, comes first in the assembled instruction, followed by 0FFH.

Let's now suppose that the task block program of which the above instruction is a part, is located in local memory space (tag bit therefore equals a logical '1') and that the address at the beginning of the assembled 'CALL' instruction is 7E31H. When the 'CALL' instruction is executed by the IOP, the TP pointer/register, containing the address of the next sequential instruction (7E36), and the tag bit are stored in three bytes of system memory ('PP' always points to system memory space) beginning at address PP + 12 as follows:

7 0	7 0	7 0
00110110	01111110	00001000
low order byte		high order byte

Since the Task block program was located in local memory space (a maximum of 64K in size) bits 4-7 of the third memory byte are a logical '0'. Bit 3 of the third byte is a logical '1', the value of the TP pointer/register's tag bit.

To return instruction execution to the next instruction following the 'CALL' a 'MOVP', not 'MOV', would be required:

```
CALL_RETURN:  MOVP  TP, [PP].12
;restore TP pointer/register and tag bit from memory
```

Assembled Instruction:

7 0	7 0	7 0	7 0
100WBAA1	100111MM	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

17 clocks bus width = 16 bits and address is even
23 clocks bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) If the address of the jump target is known to the assembler when a control transfer instruction is found on the assembler's first pass, a signed byte or word displacement, as required to reach the jump target, will be generated by the assembler. A signed byte displacement is generated if the jump target is within −128, +127 bytes of the end of the control transfer instruction; a signed word displacement, −32,768, +32,767, is generated if the target is outside the byte displacement range. The jump target cannot be outside a range of −32,768, +32,767 bytes of the end of the control transfer instruction.

If the address of a jump target cannot be determined by the assembler on its first pass (the case where 'L' contains a forward reference), the jump target is assumed to be within a −128, +127 byte range of the end of the control transfer instruction and a one byte displacement-field is generated to contain the signed displacement value when it is later determined. However, if it is later determined that a signed word displacement value is necessary to reach the jump target, the assembler flags the control transfer instruction as an error and the long form of the instruction must be coded i.e. an 'L' prefix added to the instruction.

2) A return from a CALL is made via a MOVP instruction where TP is specified as the destination register and the memory location operand is the same as that used in the initial CALL instruction. See MOVP.

3) The memory location where the TP pointer/register and tag bit are to be stored cannot be specified with a post autoincremented Index register, i.e., the AA field of the instruction may not be '11'.

4) Stored Task Pointer Format:

7 0	7 0	7 0
TP (low)	TP (high)	19181716tb 0 0 0

a) The low order byte of the TP pointer/register is stored first, followed by the next sequential byte (high), bits 8-15. The upper 4 bits, 16-19, are stored in the third byte in bit positions 4-7. The tag bit is stored in the third bit position with the unused bits, 0-2, set to logical '0'.

CLR

Clear Selected Bit to Logical Zero

Mnemonic: CLR **Coding Format:** CLR M, b

Operands: 'b' is the bit in the data memory byte (0 <= 'b' <= 7)
'M' is a data memory expression

Operation: Bit 'b' ← 0

The selected bit of a specified data memory byte located at 'M' is cleared to logical '0'.

Examples:

The memory byte located at the address formed by adding 17 to the contents of register GA contains '7DH':

```
7          0
| 01111101 |
```

The following instruction is executed:

```
    CLR  [GA].17, 5
```

The memory byte at GA + 17 now contains '5DH':

```
7          0
| 01011101 |
```

Assembled Instruction:

```
7            0 7            0 7               0
| bbb00AA0 | 111110MM | offset if AA=01 |
```

Execution Time:

16 clocks

NOTES 1) Register bits cannot be cleared using this instruction.

2) 'b' is evaluated modulo 8. If 'b' > 7 or 'b' < 0 the assembler issues an error message.

3) Bit positions within a data memory byte are specified as follows:

```
               MSB      LSB
bit positions | 76543210 |
```

Decrement Register Word

Decrement Memory Word

Mnemonic: DEC **Coding Format:** DEC R
DEC M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (OP1) ← (OP1) − 1

In a 16-bit operation, one is subtracted from the contents of the specified register 'R' or the word of data memory whose low order byte is located at 'M'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP) a 20-bit subtraction is performed. (20000H decrements to 1FFFFH)

Examples:

```
DEC  BC             ;One is subtracted from the contents of register BC.

DEC [GB+IX+]        ;One is subtracted from the word of data memory whose low
                    ;order byte is located at the address formed by adding the Index
                    ;register to GB. Note that the Index register is post
                    ;auto-incremented by two.
```

Assembled Instruction:

DEC R (DECREMENT REGISTER)

7 0	7 0
RRR00000	00111100

Execution Time:

3 clocks

DEC M (DECREMENT MEMORY WORD)

7 0	7 0	7 0
00000AA1	111011MM	offset if AA=01

Execution Time:

16 clocks bus width = 16 bits and address is even
26 clocks bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) To decrement data memory bytes use the DECB instruction.

2) Individual register bytes may NOT be decremented.

3) Decrementing zero returns 0FFFFH unless a pointer/register is operated on. In that case, decrementing zero results in 0FFFFFH.

DECB

Decrement Memory Byte

Mnemonic: DECB **Coding Format:** DECB M

Operands: 'M' is a data memory expression

Operation: (OP1) ← (OP1) − 1

The contents of the data memory byte located at 'M' are reduced by 1.

Examples:

```
DECB  [GA+IX]          ;The contents of the index register are added to register GA
                       ;to form the address of a data memory byte from which
                       ;one is subtracted.
```

Assembled Instruction:

7 0	7 0	7 0
00000AA0	111011MM	offset if AA=01

Execution Time:

16 clocks

NOTES 1) Decrementing a byte value of zero results in 0FFH.

2) Individual register bytes cannot be decremented.

3) To decrement a register or memory word use the DEC instruction.

Halt Channel Program Execution;

Clear Channel Busy Flag in Channel Control Block

Mnemonic: HLT **Coding Format:** HLT

Operands: This instruction has no operands

Operation: None

Task block program execution is stopped and the respective channel BUSY flag byte (channel one or channel two) in the Channel Control Block is cleared.

Examples:

```
HLT                   ;Task block program execution for the channel ceases.
                      ;Channel activity is resumed through a command in the
                      ;channel's CCW.
```

Assembled Instruction:

```
7          0 7          0
| 00100000 | 01001000 |
```

Execution Time:

11 clocks

NOTES 1) A task block program halt instruction must not be confused with a channel halt command issued to a channel through the Channel Control Word (CCW) in the Channel Command Block (CB). Specifically, the task block program halt instruction, 'HLT', does NOT save the TP pointer/register and tag bit or the channel's program status word.

2) By clearing the channel busy flag in the Channel Control Block, the channel indicates that it is now idle. No other activity takes place on the channel until it is restarted through a command in its CCW. The HLT instruction does NOT generate any hardware interrupt signals. Interrupt signals can be generated by a task block program using the SINTR instruction, providing that interrupts have been enabled from the channel in the Channel Control Word (CCW).

INC

Increment Register

Increment Memory Word

Mnemonic: INC

Coding Format: INC R
INC M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (OP1) ← (OP1) + 1

In a 16-bit operation, one is added to the contents of the specified register 'R', or the word of of data memory whose low order byte is located at 'M'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP), a 20-bit increment is performed. An increment can result in a carry into the upper four bits, bits 16-19), of the pointer/register. (1FFFFH increments to 20000H)

Examples:

```
INC  BC             ;One is added to register BC.

INC  [GA]           ;One is added to the word of data memory whose low order
                    ;byte is located at [GA].
```

Assembled Instruction:

INC R (INCREMENT REGISTER)

7 0	7 0
RRR00000	00111000

Execution Time:

3 clocks

INC M (INCREMENT MEMORY WORD)

7 0	7 0	7 0
00000AA1	111010MM	offset if AA=01

Execution Time:

16 clocks bus width = 16 bits and address is even
26 clocks bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) To increment a memory byte use the INCB instruction.

2) Incrementing 0FFFFH results in 0H unless a pointer/register is operated on. In a pointer/register 0FFFFH is incremented to 10000H.

INCB

Increment Memory Byte

Mnemonic: INCB **Coding Format:** INCB M

Operands: 'M' is a data memory expression

Operation: (OP1) ← (OP1) + 1

One is added to the contents of the data memory byte at location 'M'.

Examples:

```
INCB  [GB]          ;One is added to the data memory byte at location [GB].
```

Assembled Instruction:

7 0	7 0	7 0
0 0 0 0 0 A A 0	1 1 1 0 1 0 M M	offset if AA=01

Execution Time:

16 clocks

NOTES 1) Individual register bytes can not be incremented. To increment a register or a memory word use the INC instruction.

2) Incrementing 0FFH results in 00H.

Jump On Bit True

Mnemonic: JBT **Coding Format:** JBT M, b, L

Operands: 'L' is an expression representing the jump target
'b' is the bit in the data memory byte (0 <= 'b' <= 7)
'M' is a data memory expression

Operation:

```
IF bit 'b' = 1
   then (TP) ← (TP) + sdisp

ELSE next instruction
```

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JBT instruction on its first pass, a one or two byte signed displacement value, 'sdisp', is generated. This signed displacement value represents the distance in bytes from the end of the JBT instruction to the jump target. If the jump target is within a range of −128, +127 bytes, a signed byte displacement is generated. Otherwise a signed word displacement, −32,768, +32,767, is generated. Jump targets outside the signed word displacement range are not allowed.

If the address of the jump target cannot be determined when the assembler finds the JBT instruction on its first pass (the case when 'L' contains a forward reference), a signed byte displacement is assumed. Should it later be determined that a signed word displacement is necessary, the JBT instruction is flagged as an error and an LJBT instruction must be coded in its place.

The specified bit, b, of the data memory byte located at 'M', is tested. If the bit is a logical '1', the signed displacement (sign-extended to 20-bits) is added to the contents of the TP pointer/register, forming the jump target address. Program control is passed to the instruction at that address. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the tested bit is not a logical '1' the next sequential instruction is executed.

Example:

The JBT instruction allows a programmer to alter the sequence of task block program instruction execution based upon the value of a specific bit in a data memory byte.

In this example 'COMPLETION_CODE' is the name of a data memory byte in local (16-bit) address space. (If it were in system, space an LPD or LPDI instruction would be necesssary in place of the 'MOVI GB, COMPLETION_CODE' instruction.) An I/O device writes a status code to this byte upon the completion of some task. Bit five of the status code is an error indication bit, set by an abnormal task termination. The task block program checks this bit in 'COMPLETION_CODE' and jumps to an error routine if it is set, i.e., a logical '1'.

```
COMPLETION_CODE: DB 00H         ;Defines the name of a data memory
                                ;byte with an initial value of '00H'.

.

.

.
```

```
                                        ;Device activity initiated;
                                        ;upon completion a status code is
                                        ;written to 'COMPLETION_CODE'.
                                        ;'COMPLETION_CODE' is then
                                        ;examined by the task block program to
                                        ;check for an abnormal termination.


ERROR_CHECK:  MOVI GB, COMPLETION_CODE
                                        ;Move address of
                                        ;COMPLETION_CODE to register GB.
              JBT   [GB], 5, ERROR_ROUTINE
                                        ;Bit five of the data memory byte
                                        ;'COMPLETION_CODE' is tested.
                                        ;If the bit is a logical '1', indicating
                                        ;an error, the program jumps to the
                                        ;program location 'ERROR_ROUTINE'.
                                        ;If the bit is not a logical '1' the next
                                        ;sequential instruction is executed.
```

Assembled Instruction:

7 0	7 0	7 0	7 0
b b b W B A A 0	1 0 1 1 1 1 M M	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

14 clocks

NOTES 1) Register bits cannot be tested.

2) Jump targets cannot be outside a range of −32,768, +32,767 bytes from the end of a control transfer instruction. There is NO wraparound from the end of the 64k program address space to the beginning.

3) The bits in a data memory byte are specified as follows:

```
MSB          LSB
[ 7 6 5 4 3 2 1 0 ]
```

Example:

```
               7             0
              [ 1 0 1 0 0 0 1 0 ]
bit position    7 6 5 4 3 2 1 0
```

JMCE

Jump On Mask Compare Equal

Mnemonic: JMCE **Coding Format:** JMCE M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation:

```
1) (compare-result) ← (low order byte of MC register) XOR (M)

2) (mask-result) ←
      (high order byte of MC register) AND (compare-result)

3) IF (mask-result) = 0
      then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

   ELSE next instruction
```

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JMCE instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JMCE instruction to the jump target. If the jump target is within a range of −128, +127 bytes, a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JMCE instruction on its first pass (the case when 'L' contains a forward reference), a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required the JMCE instruction is flagged as an error and an LJMCE instruction must be coded in its place.

The low order byte of the MC register is used as a compare byte; the high order byte is used as a mask byte. The data memory byte located at 'M' is XORed with the compare byte. The result is then ANDed with the mask byte. If the mask-result is equal to zero, the signed displacement (sign-extended to 20-bits) is added to the TP pointer/register, formimg the jump target address. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.) Task block program execution resumes at the instruction whose address is now in TP.

If the mask-result is not zero the next sequential instruction is executed.

Example:

The JMCE instruction allows a task block program to use the result of a mask compare operation to alter the sequence of task block program instruction execution. This instruction is useful in device control programs, providing a mask and test type operation within a single instruction.

In this example, an unknown number of local data memory bytes are being moved to system memory space. The block of data being moved, however, ends with an ASCII 'ETX' character (03H). The MC register is loaded with a (low order) compare byte and (high order) mask byte to detect the 'ETX' character. Upon detection of the 'ETX' character, data movement ends and a jump is taken to 'NEXT__TASK__BLOCK', where task block program execution resumes.

```
EXTRN   START_OF_DESTINATION                ;Identify 'START_OF_DESTINATION'
                                            ;as a symbol defined in
                                            ;another program.

START_OF_BLOCK_SOURCE: DS 4096D             ;Reserve 4096D bytes of space
                                            ;with name
                                            ;'START_OF_BLOCK_SOURCE'.

        MOVI IX, 00H                        ;Load index register with initial value
                                            ;of 00H.

        MOVI MC, 0FF03H                     ;Load mask and compare bytes into
                                            ;MC register.

        MOVI GA, START_OF_BLOCK_SOURCE      ;Load register GA with starting address
                                            ;of data block to be moved.

        LPDI GB, START_OF_DESTINATION       ;Load GB as a pointer to the
                                            ;destination in system memory space.

LOOP:   JMCE [GA+IX], NEXT_TASK_BLOCK       ;Test the data byte for 'ETX' (03H)
                                            ;and jump to 'NEXT_TASK_BLOCK'
                                            ;if found.

        MOVB [GB+IX], [GA+IX+]              ;Move the data memory byte at location
                                            ;[GA+IX+] to location [GB+IX].
                                            ;The Index Register is post
                                            ;auto-incremented.

        JMP LOOP                            ;Return to JMCE instruction, check
                                            ;next data byte for 'ETX'.

        .
        .

NEXT_TASK_BLOCK:   ...                      ;Instruction where task block program
                                            ;execution resumes when the 'ETX'
                                            ;character is found.
```

Assembled Instruction:

7 0	7 0	7 0	7 0
0 0 0 W B A A 0	1 0 1 1 0 0 M M	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

14 clocks

JMCNE

Jump On Mask Compare Not Equal

Mnemonic: JMCNE **Coding Format:** JMCNE M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation:

```
1) (compare-result) ← (low order byte of MC register) XOR (M)

2) (mask-result) ←
       (high order byte of MC register) AND (compare-result)

3) If (mask-result) <> 0
       then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JMCNE instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JMCNE instruction to the jump target. If the jump target is within a range of −128, +127 bytes a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JMCNE instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required, the JMCNE instruction is flagged as an error and an LJMCNE instruction must be coded in its place.

The low order byte of the MC register is used as a compare byte; the high order byte is used as a mask byte. The data memory byte located at 'M' is XORed with the compare byte. The result is then ANDed with the mask byte. If the mask-result is not equal to zero, the signed displacement (sign-extended to 20-bits) is added to the TP pointer/register, formimg the jump target address. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.) Task block program execution resumes at the instruction whose address is now in TP.

If the mask-result is zero the next sequential instruction is executed.

Example:

The JMCNE instruction allows a task block program to use the result of a mask compare operation to alter the sequence of task block program instruction execution.

In this example the data memory byte 'TERMINATE_CONDITION' contains a completion code. When bit four of 'TERMINATE_CONDITION' is a logical zero and bit seven is a logical one, a catastrophic error is indicated. (Catastrophic only when both conditions are present, i.e. bit four is a logical zero and bit seven is a logical one.) Using the JMCNE instruction the following code tests for the catastrophic error and jumps to 'ANOTHER_BLOCK_OF_CODE' if it is not found. If it is found, the next sequential instruction 'ERROR_ROUTINE' is executed.

```
TERMINATE_CONDITION: DB 00H             ;Define a data memory byte location
                                        ;named 'TERMINATE_CONDITION'.

MOVE GA, TERMINATE_CONDITION            ;Load register GA with address of data
                                        ;memory byte to be tested.

MOVI MC, 0B080H                         ;Load MC register with compare and
                                        ;mask bytes.

JMCNE [GA], ANOTHER_BLOCK_OF_CODE       ;Mask compare data memory byte at
                                        ;location [GA]. Jump to
                                        ;'ANOTHER_BLOCK_OF_CODE' if
                                        ;mask compare result is not equal to
                                        ;zero. If result is zero
                                        ;'ERROR_ROUTINE' is the next
                                        ;instruction executed.

ERROR_ROUTINE:                          ;Label of instruction executed if mask
                                        ;compare result is zero.
.
.

ANOTHER_BLOCK_OF_CODE:   .              ;Label of instruction executed if mask
                                        ;compare result is not zero.
```

Assembled Instruction:

7 0	7 0	7 0	7 0
0 0 0 W B A A 0	1 0 1 1 0 1 M M	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

14 clocks

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes from the end of a control transfer instruction. There is NO wraparound from the end of the 64k range of task block program instruction addresses to the beginning.

JMP

Jump Unconditional

Mnemonic: JMP **Coding Format:** JMP L

Operands: 'L' is an expression representing the jump target

Operation: (TP) ← (TP) + sdisp (sign-extended to 20-bits)

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JMP instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JMP instruction to the jump target. If the jump target is within a range of −128, +127 bytes, a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JMP instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required, the JMP instruction is flagged as an error and an LJMP instruction must be coded in its place.

The signed displacement, 'sdisp', is sign extended to 20-bits and added to the TP pointer/register forming the jump target address. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.) Program control passes to the instruction at that address.

Example:

The JMP instruction unconditionally alters the sequence of task program instruction execution. In this example a JMP instruction is coded at the end of an error routine to pass program control to a statement, 'CONTINUE', where normal processing resumes after execution of the error routine.

```
ERROR_ROUTINE:   . . .                  ;The beginning of a section of code
                                        ;used to correct an error condition
                                        ;detected while processing.

   .
   .

JMP   CONTINUE                          ;Return program control to instruction
                                        ;labeled 'CONTINUE' after executing
                                        ;the error routine.

   .
   .

CONTINUE:   . . .                       ;The instruction executed after JMP
                                        ;instruction.
```

Assembled Instruction

JMP L (SIGNED BYTE DISPLACEMENT)

```
7           0 7          0 7           0
| 10001000   | 00100000   |    sdisp    |
```

Execution Time:

3 clocks

JMP L (SIGNED WORD DISPLACEMENT)

7 0	7 0	7 0	7 0
10010001	00100000	sdisp-low	sdisp-high

Execution Time:

3 clocks

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k instruction address space to the beginning.

JNBT

Jump If Bit Not True

Mnemonic: JNBT **Coding Format:** JNBT M, b, L

Operands: 'L' is an expression representing the jump target
'b' is the bit in the data memory byte (0 <= b <= 7)
'M' is a data memory expression

Operation:

```
If bit 'b' <> 1
   then TP ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JNBT instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JNBT instruction to the jump target. If the jump target is within a range of −128, +127 bytes, a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JNBT instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required the JNBT instruction is flagged as an error and an LJNBT instruction must be coded in its place.

The selected bit, 'b', of the data memory byte at location 'M' is tested. If the bit is not a logical one the signed displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register to form the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the tested bit is a logical one the next sequential instruction is executed.

Example:

The JNBT instruction enables the value of a specified bit in a data memory byte to alter the sequence of task block program instruction execution.

In this example bit four of a data memory byte 'ERROR_?' is tested by the JNBT instruction. If the bit is not a logical one, program control jumps to the statement at 'GOOD_RESULT'. If the bit is a logical one the next sequential instruction, 'BAD_RESULT', is executed.

```
ERROR_?:  DB   00H                  ;Define a data memory byte named
                                    ;'ERROR_?' with an initial value of
                                    ;00H.

MOVI   GA,   ERROR_?                ;Load register GA with adddress of
                                    ;data memory byte 'ERROR_?'.

JNBT   [GA], 4, GOOD_RESULT         ;Test the fourth bit of the data memory
                                    ;byte located at [GA] and jump to
                                    ;'GOOD_RESULT' if it is not a logical
                                    ;one else execute the next sequential
                                    ;instruction, 'BAD_RESULT'.
```

```
BAD_RESULT:     ...                       ;If the fourth bit of 'ERROR_?' is a
                                          ;logical one this instruction is
                                          ;executed.

      .
      .

GOOD_RESULT:     ...                      ;If the fourth bit of 'ERROR_?' is not a
                                          ;logical one, program control jumps to
                                          ;this instruction.
```

Assembled Instruction:

7 0	7 0	7 0	7 0
b b b W B A A 0	1 0 1 1 1 0 M M	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

14 clocks

NOTES 1) Register bits cannot be tested using the JNBT instruction.

2) The jump target of a control transfer instruction must be within a range of −32,768, +32,767 bytes from the end of the instruction. There is NO wraparound from the end of the 64k instruction address range to the beginning.

3) The bits in a data memory byte are specified according to the following format:

```
MSB        LSB
| 7 6 5 4 3 2 1 0 |
```

Example:

```
               | 1 0 1 0 0 0 1 0 |
bit position     7 6 5 4 3 2 1 0
```

JNZ

Jump On Nonzero Register Or Memory Word

Mnemonic: JNZ

Coding Format:
```
JNZ  R, L
JNZ  M, L
```

Operands: 'R' is a register symbol
'M' is a data memory expression
'L' is an expression representing the jump target

Operation:
```
If (OP1) <> 0
   then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JNZ instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JNZ instruction to the jump target. If the jump target is within a range of −128, +127 bytes a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JNZ instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required the JNZ instruction is flagged as an error and an LJNZ instruction must be coded in its place.

The contents of the specified register 'R' or the word of data memory whose low order byte is located at 'M' are examined. If the contents are not logical zero the signed displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register, forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

This instruction performs a 16-bit test. If 'R' is a 20-bit pointer/register (GA, GB, GC, or TP), the contents of its upper four bits, bits 16-19, cannot be determined using this instruction.

If the contents of OP1 are equal to logical zero the next sequential instruction is executed.

Example:

```
JNZ   BC, $ + 17                    ;If register BC is not zero jump ahead
                                    ;17 bytes from the beginning of this
                                    ;instruction.

JNZ   [GC], RETRY                   ;If the word of data memory beginning
                                    ;(low order byte) at location [GC] is
                                    ;not zero jump to instruction labeled
                                    ;'RETRY'.
```

JNZ

Assembled Instruction:

JNZ R, L (JUMP IF REGISTER NOT EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0
RRRWB000	010000MM	sdisp (1-2 bytes)

Execution Time:

5 clocks

JNZ M, L (JUMP IF MEMORY WORD NOT EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0	7 0
000WBAA1	111000MM	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

12 clocks if bus width = 16 bits and address is even
16 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

JNZB

Jump On Nonzero Memory Byte

Mnemonic: JNZB **Coding Format:** JNZB M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation:

```
If (M) <> 0
   then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JNZB instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JNZB instruction to the jump target. If the jump target is within a range of −128, +127 bytes a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JNZB instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required the JNZB instruction is flagged as an error and an LJNZB instruction must be coded in its place.

The contents of the data memory byte at location 'M' are examined. If the contents are not equal to logical zero the signed displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register, forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the contents of the data memory byte are equal to logical zero the next sequential instruction is executed.

Example:

```
JNZB   [GA].4, RECOVERY                 ;If the data memory byte at location
                                        ;[GA] + 4 is not equal to logical zero
                                        ;a jump is made to the instruction
                                        ;labeled 'RECOVERY'.
```

Assembled Instruction:

7 0	7 0	7 0	7 0
0 0 0 W B A A 0	1 1 1 0 0 0 M M	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

12 clocks

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction address space to the beginning.

Jump On Zero Register Or Memory Word

Mnemonic: JZ

Coding Format:
```
JZ  R,  L
JZ  M,  L
```

Operands: 'R' is a register symbol
'M' is a data memory expression
'L' is an expression representing the jump target

Operation:
```
If (OP1) = 0
   then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing some location within the program. If the address of the jump target can be determined by the assembler when it encounters the JZ instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JZ instruction to the jump target. If the jump target is within a range of −128, +127 bytes a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JZ instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required, the JZ instruction is flagged as an error and an LJZ instruction must be coded in its place.

The contents of the specified register 'R' or the word of data memory whose low order byte is located at 'M' are examined. If they equal logical zero the signed displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

This instruction performs a 16-bit test. If 'R' is a 20-bit pointer/register (GA, GB, GC, or TP), the contents of its upper four bits, bits 16-19, cannot be determined using this instruction.

If the contents are not logical zero the next sequential instruction is executed.

Examples:

```
JZ  IX,  MOVE_ROUTINE+5            ;If the contents of the Index register
                                   ;are equal to logical zero a jump is
                                   ;made to the instruction at location
                                   ;MOVE_ROUTINE + 5.

JZ  [PP].12, ALTERNATE             ;If the word of data memory beginning
                                   ;(low order byte) at location [PP] + 12 is
                                   ;zero a jump is made to ALTERNATE.
```

JZ

Assembled Instruction:

JZ R, L (JUMP IF REGISTER EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0
RRRWB000	01000100	sdisp (1-2 bytes)

Execution Time:

5 clocks

JZ M, L (JUMP IF MEMORY WORD EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0	7 0
000WBAA1	111001MM	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

12 clocks if bus width = 16 bits and address is even
16 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

Jump On Zero Memory Byte

Mnemonic: JZB **Coding Format:** JZB M, L

Operand Format: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation: If (M) = 0
then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction

'L', the jump target, is an expression representing a location within the program. If the address of the jump target can be determined by the assembler when it encounters the JZB instruction on its first pass, a one or two byte signed displacement, 'sdisp', is generated. This signed displacement represents the distance in bytes from the end of the JZB instruction to the jump target. If the jump target is within a range of −128, +127 bytes a signed byte displacement results. Otherwise, a signed word displacement, −32,768, +32,767, is generated.

If the address of the jump target cannot be determined when the assembler finds the JZB instruction on its first pass (the case when 'L' contains a forward reference) a signed byte displacement is assumed. Should it later be determined that a signed word displacement is required the JZB instruction is flagged as an error and an LJZB instruction must be coded in its place

If the contents of the data memory byte located at 'M' are a logical zero the signed displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register, forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer /register when the jump target address is formed.)

If the contents are not logical zero the next sequential instruction is executed.

Example:

```
JZB  [GA+IX], NEXT_BLOCK            ;If the data memory byte at the location
                                    ;[GA+IX] is equal to logical zero a jump
                                    ;is made to the instruction labeled
                                    ;'NEXT_BLOCK'.
```

Assembled Instruction

7 0	7 0	7 0	7 0
0 0 0 W B A A 0	1 1 1 0 0 1 M M	offset if AA=01	sdisp (1-2 bytes)

Execution Time:

12 clocks

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes from the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

LCALL

Long Call
(Store TP Pointer/Register and Tag Bit; JUMP)

Mnemonic: LCALL **Coding Format:** LCALL M, L

Operand Format: 'L' is an expression representing the jump target
'M' is a data memory expression

Operation: 1) (M) ← (TP) + tag bit

2) (TP) ← (TP) + sdisp (sign-extended to 20-bits)

The TP pointer/register, containing the address of the next sequential instruction , and the TP pointer/register tag bit, indicating a system or local space task block program location, are saved in 3 bytes of data memory beginning at location 'M'.

'L', the jump target, is an expression representing a location within the program. Unlike the CALL instruction, which can generate a one *or* two byte displacement value, the LCALL instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LCALL instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LCALL instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a CALL instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The signed word displacement, 'sdisp', is sign-extended to 20-bits and added to the contents of the TP pointer/register forming the jump target address. (The TP pointer/register contains the address of the next sequential instruction when the LCALL target address is formed.) Program control passes to the instruction whose address is now in the TP pointer/register (the jump target).

See note 4 below for the format of the stored TP pointer/register and tag bit.

Example:

The LCALL instruction stores the TP pointer/register and tag bit in memory and unconditionally branches to another location within the program. Return is made from the jump by restoring the stored TP pointer/register and tag bit with a MOVP instruction.

In this example a jump is made to an instruction labelled 'SOME_ROUTINE?'. The TP pointer/register and tag bit are stored in three bytes of data memory beginning at the location named 'STORED_POINTER'.

A return is made from the jump to 'SOME_ROUTINE?' via a 'MOVP' instruction. The TP pointer/register and tag bit are restored from 'STORED_POINTER'.

```
STORED_POINTER: DS  3              ;Reserve 3 bytes of data memory
                                   ;named 'STORED_POINTER' in which
                                   ;the TP pointer/register and tag bit
                                   ;are saved.

      .
      .
```

```
MOVI   GC,   STORED_POINTER           ;Load the data memory address of the
                                      ;location where the TP pointer/register
                                      ;and tag bit will be stored into GC.

LCALL   [GC], SOME_ROUTINE?           ;Store TP pointer/register and tag bit
                                      ;at address contained in GC
                                      ;('STORED_POINTER'); branch to
                                      ;instruction at 'SOME_ROUTINE?'

  .
  .

MOVI   GA,   STORED_POINTER           ;Load data memory address of stored
                                      ;TP pointer/register and tag bit into
                                      ;GA.

MOVP   TP,   [GA]                     ;Return from jump, restore TP
                                      ;pointer/register value and tag bit
                                      ;from 'STORED_POINTER'.
```

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
1 0 0 1 0 A A 1	1 0 0 1 1 1 M M	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

17 clocks if bus width = 16 bits and address is even
23 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) A return from an LCALL instruction is made via a MOVP instruction where 'TP' is specified as the destination register and the data memory location is the same as that used in the initial LCALL instruction. See MOVP.

2) Jump targets must be within a −32,768, +32,767 byte range of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

3) The memory location where the TP register and tag bit are stored cannot be specified using a post autoincremented Index register ([PREG+IX+]), i.e., the AA field of the instruction cannot be '11'.

4) Stored Task Pointer Format:

7 0	7 0	7 0
TP (low)	TP (high)	19 18 17 16 tb 0 0 0

a) The low order byte of the TP pointer/register is stored first, followed by the next sequential byte (high), bits 8-15. The upper 4 bits, 16-19, are stored in the third byte in bits 4-7. The tag bit is stored in bit 3 and the unused bits, 0-2, set to logical '0'.

LJBT

Long Jump On Bit True

Mnemonic: LJBT **Coding Format:** LJBT M, b, L

Operands: 'L' is an expression representing the jump target
'b' is the bit in the data memory byte ($0 <= b <= 7$)
'M' is a data memory expression

Operation:

```
If bit 'b' = 1
   then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. Unlike the JBT instruction, which can generate a one *or* two byte displacement value, the LJBT instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJBT instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJBT instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JBT instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The specified bit, 'b', of the data memory byte located at 'M', is tested. If the bit is a logical '1' the signed word displacement, 'sdisp', is sign-extended to 20-bits and added to the contents of the TP pointer/register, forming the address of the jump target, 'L'. Program control is passed to the instruction at that address. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the tested bit is not a logical '1' the next sequential instruction is executed

Example:

The LJBT instruction allows a programmer to alter the sequence of task block program instruction execution based upon the value of a specific bit in a data memory byte. The jump target of the LJBT instruction is within a range of −32,768, +32,767 bytes of the end of the assembled LJBT instruction.

In this example the user defined area of the Parameter Block (PB) contains a parameter byte whose contents are used to direct the IOP channel's operation. Here the task block program checks bit 7 of the parameter byte and jumps to an instruction labeled 'Delay' if the bit is a logical '1'. If the bit is not a logical '1' the instruction labeled 'ALL__SET' is executed.

Note that the LJBT instruction is required in this case since (1) the address of 'DELAY' is not known to the assembler when the LJBT instruction is found on its first pass and (2) a signed word displacement value is required because 'DELAY' is outside a −128, +127 byte range of the end of the instruction.

```
LJBT   [PP].27, 7, DELAY        ;Test bit 7 of parameter byte in user
                                ;defined area of the Parameter Block;
                                ;jump to instruction labeled 'DELAY' if
                                ;bit is a logical '1'.
```

```
ALL_SET:    MOVI   CC,   DMA_INFO          ;This instruction executed if tested bit
                                           ;is not a logical '1'. An immediate word
                                           ;value is loaded into the CC (Channel
                                           ;Control) register.

                 .

                 .    (25,000 bytes of assembled source program statements)

                 .

DELAY:      MOVBI   BC,   TIMER            ;If tested bit is a logical '1' program
                                           ;control jumps to this instruction.
```

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
b b b 1 0 A A 0	1 0 1 1 1 1 M M	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

14 clocks

NOTE 1) Register bits cannot be tested.

2) Jump targets must be within a −32,768, +32,767 byte range of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

3) The bits of a data memory byte are specified as follows:

MSB LSB
7 6 5 4 3 2 1 0

Example:

	1 0 1 0 0 0 1 0
bit positions	7 6 5 4 3 2 1 0

LJMCE

Long Jump On Mask Compare Equal

Mnemonic: LJMCE **Coding Format:** LJMCE M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation:

```
1) (compare-result) ← (low order byte of MC register) XOR (M)

2) (mask-result) ← (high order byte of MC) AND (compare-result)

3) If (mask-result) = 0
       then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

   Else next instruction
```

'L', the jump target, is an expression representing a location within the program. Unlike the JMCE instruction, which can generate a one *or* two byte displacement value, the LJMCE instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJMCE instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJMCE instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JMCE instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The low order byte of the MC register is used as a compare byte; the high order byte is used as a mask byte. The data memory byte at location 'M' is XORed with the compare byte. The result is then ANDed with the mask byte. If the mask-result is equal to zero 'sdisp' is added to the TP pointer/register, forming the jump target address. Task block program execution resumes at the instruction whose address is now in TP (the jump target). The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.

If the mask-result is not zero the next sequential instruction is executed.

Example:

The LJMCE instruction allows a task block program to use the result of a mask compare operation to alter the sequence of task block program instruction execution. The jump target of the LJMCE instruction is within a range of −32,768, +32,767 bytes of the end of the instruction.

In this example an I/O device writes a status code to a data memory byte labeled 'OK?'. The following bit pattern in 'OK?' indicates to the task block program that an error has occured in the device's operation and corrective action must be taken:

```
7         0
┌─────────┐
│1X01X1X0 │
└─────────┘
```

An 'X' in a bit position indicates that the bit can be either a logical '1' or a logical '0' in other words, the program doesn't care what value is present when checking for an error. In the remaining bit positions an error is indicated only if the indicated values are present. If any of the values is not as specified no error has occured.

The task block program loads the MC register with a compare and a mask value to detect the above error code. Using the LJMCE instruction the program is able to jump to a routine labeled 'FIX_IT' when an error has occured.

```
OK?:            DB      00H             ;Define a byte of data memory with the name
                                        ;'OK?' and an initial value of 00H.
                .

                MOVI  GC,  OK?          ;Load register GC with the address of the data
                                        ;memory byte containing the device status.

                MOVI  MC,  0B594H       ;Load MC register with mask and compare
                                        ;values to detect the error code.

PROCESS_LOOP:   LJMCE  [GC], FIX_IT     ;Check device status—if no error indicated
                                        ;instruction labeled 'OUT_STEP_1'
                                        ;is executed.

OUT_STEP_1:     MOV  GA, [PP].22        ;Load register GA with 16-bits of data from the
                                        ;user-defined portion of the Parameter Block.

                .   (start I/O device operation)

                JMP  PROCESS_LOOP       ;The end of I/O device operation. Assuming
                                        ;that the I/O device has written its error code in
                                        ;data memory at 'OK?' and that register GC still
                                        ;contains the address of the data memory byte,
                                        ;the task block program jumps to the LJMCE
                                        ;instruction to check for an error. This
                                        ;processing loop continues until either an error
                                        ;occurs or the channel is interrupted/halted by
                                        ;a channel command in the Channel Control
                                        ;Word (CCW).

                .   (14,000 bytes of assembled program instructions)

FIX_IT:         SINTR                   ;The interrupt service flip-flop for the channel
                                        ;is set indicating to the main system hardware
                                        ;the occurance of the I/O device error.
                                        ;(Assuming channel interrupts have been
                                        ;enabled.)
```

Note that the LJMCE instruction must be coded in this case since (1) the address of the jump target 'FIX_IT' is not known by the assembler when it encounters the LJMCE instruction on its first pass and (2) the jump target is outside a −128, +127 byte range from the end of the LJMCE instruction. If a JMCE instruction is coded here it will be flagged as an error by the assembler since it assumes a one byte signed displacement when the jump target address is not known on the assembler's first pass and a two byte (word) displacement is required here.

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
00010AA0	101100MM	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

14 clocks

NOTE 1) Jump targets must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

LJMCNE

Long Jump On Mask Compare Not Equal

Mnemonic: LJMCNE **Coding Format:** LJMCNE M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation:

```
1) (compare-result) ← (low order byte of MC register) XOR (M)

2) (mask-result) ← (high order byte of MC) AND (compare-result)

3) If (mask-result) <> 0
     then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

   Else next instruction
```

'L', the jump target, is an expression representing a location within the program. Unlike the JMCNE instruction, which can generate a one *or* two byte displacement value, the LJMCNE instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJMCNE instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJMCNE instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JMCNE instruction is found on its first pass. (2) The required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The low order byte of the MC register is used as a compare byte; the high order byte is used as a mask byte. The data memory byte at location 'M' is XORed with the compare byte. The result is then ANDed with the mask byte. If the mask-result is not equal to zero, 'sdisp' is added to the TP pointer/register, forming the jump target address. Task block program execution resumes at the instruction whose address is now in TP (the jump target). (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the mask-result is equal to zero, the next sequential instruction is executed.

Example:

The LJMCNE instruction allows a task block program to use the result of a mask compare operation to alter the sequence of task block program instruction execution. The jump target of the LJMCNE instruction is within a range of −32,768, +32,767 bytes.

In this example, each source byte is inspected for a logical '1' in bit position seven and a logical '0' in bit position zero before it is processed. If the byte does not conform to the above format, a jump occurs to the instruction labeled 'ALT_PROCESS'. If the byte does conform to the format, the instruction labeled 'NML_PROCESS' is executed.

```
MOVI   MC,  8180H            ;Load mask and compare bytes into
                             ;register MC.
```

```
              LJMCNE   [GB], ALT__PROCESS  ;The byte to be tested is at the address
                                           ;contained in register GB. If the byte has
                                           ;a logical '1' in bit position seven and a
                                           ;logical zero in bit position zero, the
                                           ;instruction labeled 'NML__PROCESS' is
                                           ;executed. If the byte is not in the above
                                           ;format a jump is made to the instruction
                                           ;labeled 'ALT__PROCESS'.

NML__PROCESS: MOVB   [GA+IX+], [GB]        ;Move the byte at address GB to the
                                           ;location addressed by GA + IX (post
                                           ;auto-increment IX).
              .

              .

              .   (200 bytes of assembled program instructions)

ALT__PROCESS: NOTB   [GB]                  Form the one's complement of the byte
                                           ;addressed by GB.
```

Note that the LJMCNE instruction is required here since (1) the address of the jump target, 'ALT__PROCESS' is not known by the assembler when it finds the LJMCNE instruction on its first pass and (2) the jump target is outside a −128, +127 byte range of the end of the instruction. A JMCNE instruction would be flagged as an error if coded here because the assembler would assume a displacement within a −128, +127 byte range on its first pass when the jump target is unknown. Later the displacement is found to be outside the assumed range, resulting in an error.

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
00010AA0	101101MM	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

14 clocks

NOTE 1) A jump target must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

LJMP

Long Jump Unconditional

Mnemonic: LJMP **Coding Format:** LJMP L

Operands: 'L' is an expression representing the jump target

Operation: (TP) ← (TP) + sdisp (sign-extended to 20-bits)

'L', the jump target, is an expression representing a location within the program. Unlike the JMP instruction, which can generate a one *or* two byte displacement value, the LJMP instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJMP instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJMP instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JMP instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The signed word displacement, 'sdisp', is added to the TP pointer/register, forming the jump target address. Program control passes to the instruction at that address. (The TP pointer register contains the address of the next sequential instruction when the jump target address is formed.)

Example:

```
            LJMP   ERR__TYPE + 3    ;Unconditional jump to an instruction three
                                    ;bytes beyond an instruction labeled
                                    ;'ERR__TYPE'.
             .

             .

             .     (1,253 bytes of assembled source program statements)

ERR__TYPE:  ADD    BC, [PP].12      ;Jump target is three bytes beyond this
                                    ;instruction.
```

Note that the LJMP instruction is required here since (1) the address of the jump target, 'ERR__TYPE' is not known by the assembler when it finds the LJMP instruction on its first pass and (2) the jump target is outside a −128, +127 byte range of the end of the instruction. A JMP instruction would be flagged as an error if coded here because the assembler would assume a displacement within a −128, +127 byte range on its first pass when the jump target is unknown. Later the displacement is found to be outside the assumed range, resulting in an error.

Assembled Instruction:

7 0	7 0	7 0	7 0
10010001	00100000	sdisp-low	sdisp-high

Execution Time:

3 clocks

NOTE 1) A jump target must be within a −32,768, +32,767 byte range of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

Long Jump If Bit Not True

Mnemonic: LJNBT **Coding Format:** LJNBT M, b, L

Operands: 'L' is an expression representing the jump target
'b' is the bit in the data memory byte (0 <= b <= 7)
'M' is a data memory expression

Operation: If bit 'b' <> 1
then TP ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction

'L', the jump target, is an expression representing a location within the program. Unlike the JNBT instruction, which can generate a one *or* two byte displacement value, the LJNBT instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJNBT instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJNBT instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JNBT instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The selected bit, 'b', of the data memory byte located at 'M' is tested. If the bit is not a logical one, 'sdisp' is sign-extended to 20-bits and added to the TP pointer/register to form the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the tested bit is a logical one, the next sequential instruction is executed.

Example:

The LJNBT instruction enables the value of a specified bit in a data memory byte to alter the sequence of task block program instruction execution. The jump target of the LJNBT instruction is within a range of −32,768, +32,767 bytes.

```
        LJNBT  [PP].STATUS, 3, MAX     ;Bit three of a byte located at offset value
                                       ;'STATUS' from the beginning of the Parameter
                                       ;Block is tested. If the bit is not a logical one, a
                                       ;jump is made to the statement labeled 'MAX';
                                       ;otherwise the next sequential instruction,
                                       ;'MIN', is executed.

MIN:    MOVIB  BC,  100                ;Load register BC with immediate byte value of
                                       ;100 (decimal).

        .

        .      (15,000 bytes of assembled source program statements)

MAX:    MOVI   BC,  10000              ;Load register BC with immediate word value of
                                       ;10,000 (decimal).
```

LJNBT

Note that the LJNBT instruction is required here since (1) the address of the jump target, 'MAX', is not known by the assembler when it finds the LJNBT instruction on its first pass, and (2) the jump target is outside a −128, +127 byte range of the end of the instruction. A JNBT instruction would be flagged as an error if coded here because the assembler would assume a displacement within a −128, +127 byte range on its first pass when the jump target is unknown. Later the displacement is found to be outside the assumed range, resulting in an error.

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
b b b 1 0 A A 0	1 0 1 1 1 0 M M	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

14 clocks

NOTES 1) Register bits cannot be tested using the LJNBT instruction.

2) A jump target must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

3) The bits in a data memory byte are specified as follows:

```
MSB       LSB
| 76543210 |
```

Example:

```
               7        0
               | 10100010 |
bit positions    76543210
```

Long Jump On Nonzero Register Or Memory Word

Mnemonic: LJNZ **Coding Format:** LJNZ R, L
LJNZ M, L

Operands: 'R' is a register symbol
'M' is a data memory expression
'L' is an expression representing the jump target

Operation:

```
If (OP1) <> 0
   then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. Unlike the JNZ instruction, which can generate a one *or* two byte displacement value, the LJNZ instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJNZ instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJNZ instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JNZ instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The contents of the specified register 'R' or the word of data memory whose low order byte is located at 'M' are examined. If the contents are not logical zero, the signed word displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register, forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

This instruction performs a 16-bit test. If 'R' is a 20-bit pointer/register (GA, GB, GC, or TP), the contents of its upper four bits, bits 16-19, cannot be determined using this instruction.

If the contents of OP1 are a logical zero, the next sequential instruction is executed.

Examples:

```
LJNZ  IX, FAR_AHEAD          ;If the IX register does not equal zero a jump is
                             ;made to the instruction labeled
                             ;'FAR_AHEAD'.

LJNZ  [GB], NEXT_1           ;If the word of data memory beginning (low
                             ;order byte) at address contained in GB is not
                             ;zero, a jump is made to the instruction labeled
                             ;'NEXT_1'.
```

Assembled Instruction:

LJNZ R,L (JUMP IF REGISTER NOT EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0	7 0
RRR10000	01000000	sdisp-low	sdisp-high

LJNZ

Execution Time:

5 clocks

LJNZ M, L (JUMP IF MEMORY WORD NOT EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0	7 0	7 0
00010AA1	111000MM	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

12 clocks if bus width = 16 bits and address is even
16 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) A jump target must be within a range of −32,768, +32,767 bytes of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

Long Jump on Nonzero Memory Byte

Mnemonic: LJNZB **Coding Format:** LJNZB M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation: If (M) <> 0
then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction

'L', the jump target, is an expression representing a location within the program. Unlike the JNZB instruction, which can generate a one *or* two byte displacement value, the LJNZB instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJNZB instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJNZB instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JNZB instruction is found on its first pass, and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The contents of the data memory byte located at 'M' are examined. If the contents are not equal to logical zero, the signed word displacement, 'sdisp', is sign-extended to 20-bits and added to the TP pointer/register, forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the contents of the memory byte are equal to logical zero, the next sequential instruction is executed.

Example:

```
COUNT:     DB   25               ;Define a byte of data memory labeled
                                 ;'COUNT' with an initial value of 25 (decimal).
           .

           .

           .

PROCESS1:  MOVI  IX, 300H        ;Move immediate word value to register IX.

           .

           .

           .     (150 bytes of assembled source program statements)

           MOVI  GC, COUNT       ;load address of data memory byte into register
                                 ;GC
```

```
                LJNZB   [GC], AGAIN     ;If the data memory byte addressed by GC
                                        ;('COUNT') is not zero, a jump is made to the
                                        ;location represented by the expression
                                        ;'AGAIN'.

        AGAIN   EQU     PROCESS1        ;Define a symbol 'AGAIN' as a synonym for the
                                        ;label 'PROCESS1'.
```

Note that the LJNZB instruction is required here: (1) the address of the jump target, represented by the expression 'AGAIN', is not known to the assembler on its first pass, and (2) the assembler assumes a displacement within a −128, +127 byte range of the end of the instruction if a JNZB instruction is coded; the displacement is later determined to be outside the −128, +127 byte range, resulting in the flagging of the JNZB instruction as an error.

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
00010AA0	111000MM	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

12 clocks

NOTE 1) A jump target must be within a −32,768, +32,767 byte range of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

Long Jump on Zero Register Or Memory Word

Mnemonic: LJZ

Coding Format:
```
LJZ  R,  L
LJZ  M,  L
```

Operands: 'R' is a register symbol
'M' is a data memory expression
'L' is an expression representing the jump target

Operation:
```
If (OP1) = 0
   then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction
```

'L', the jump target, is an expression representing a location within the program. Unlike the JZ instruction, which can generate a one *or* two byte displacement value, the LJZ instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJZ instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJZ instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JZ instruction is found on its first pass and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

The contents of the specified register 'R' or the word of data memory whose low order byte is located at 'M' are examined. If they equal logical zero, the signed word displacement, 'sdisp', is sign-extended to 20 bits and added to the TP pointer/register forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

This instruction performs a 16-bit test. If 'R' is a 20-bit pointer/register (GA, GB, GC, or TP), the contents of its upper four bits, bits 16-19, cannot be determined using this instruction.

If the contents of OP1 are not logical zero, the next sequential instruction is executed.

Examples:

```
           LJZ  BC,  CNCLUDE        ;If register BC equals zero, a jump is made
                                    ;to the instruction labeled 'CNCLUDE'.

           LJZ  [PP].16, CNCLUDE    ;If the word of data memory beginning (low
                                    ;order byte) at PP + 16 is zero, a jump is
                                    ;made to the instruction labeled
                                    ;'CNCLUDE'.
           .

           .

           .    (200 bytes of assembled source program statements)

CNCLUDE:   MOVBI [PP].12, 0FFH      ;The jump target.
```

Note that the LJZ instruction is required in both of the above instructions: (1) the address of the jump target 'CNCLUDE' is not known to the assembler when it encounters the LJZ instruction on its first pass, and (2) the displacement to the jump target is outside a −128, +127 byte range. A JZ instruction would be flagged as an error if it were coded here since the assembler assumes a −128, +127 byte displacement range when the jump target address is not known.

Assembled Instruction:

LJZ R, L (JUMP IF REGISTER EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0	7 0
RRR10000	01000100	sdisp-low	sdisp-high

Execution Time:

5 clocks

LJZ M, L (JUMP IF MEMORY WORD EQUAL TO LOGICAL ZERO)

7 0	7 0	7 0	7 0	7 0
00010AA1	111001MM	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

12 clocks if bus width = 16 bits and address is even
16 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) A jump target must be within a −32,768, +32,767 byte range of the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

Long Jump on Zero Memory Byte

Mnemonic: LJZB **Coding Format:** LJZB M, L

Operands: 'M' is a data memory expression
'L' is an expression representing the jump target

Operation: If (M) = 0
then (TP) ← (TP) + sdisp (sign-extended to 20-bits)

Else next instruction

'L', the jump target, is an expression representing a location within the program. Unlike the JZB instruction, which can generate a one *or* two byte displacement value, the LJZB instruction forms a signed word displacement value, regardless of the size of the displacement necessary to reach the jump target. This signed word displacement, 'sdisp', is the distance in bytes from the end of the LJZB instruction to the jump target. A displacement in the range −128, +127 bytes results in a signed word displacement value whose high order byte is 00H or 0FFH.

The LJZB instruction must be coded only when: (1) the address of the jump target cannot be determined by the assembler when a JZB instruction is found on its first pass, and (2) the required displacement to the jump target is outside a range of −128, +127 bytes from the end of the assembled instruction.

If the contents of the specified memory byte, M, are a logical zero, the signed word displacement, 'sdisp', is sign-extended to 20 bits and added to the TP pointer/register, forming the address of the jump target, 'L'. (The address of the next sequential instruction is in the TP pointer/register when the jump target address is formed.)

If the contents of the data memory byte are not logical zero, the next sequential instruction is executed.

Example:

```
LOOP1:    MOVI  CC, UNIT1_INIT     ;An instruction labeled 'LOOP1' which loads an
                                   ;immediate word value (the value of the symbol
                                   ;'UNIT1_INIT') into register CC.

          .

          .

          .    (305 bytes of assembled source program statements)

          .

          .

          .
          LJZB [PP].9, REPEAT      ;If the byte located nine bytes from the
                                   ;beginning of the Parameter Block is zero, a
                                   ;jump is made to the jump target represented
                                   ;by the expression 'REPEAT'.

REPEAT    EQU   LOOP1              ;Define a symbol 'REPEAT' with the value of
                                   ;'LOOP1'. 'REPEAT' references the same
                                   ;instruction as 'LOOP1'.
```

Note that the LJZB instruction is required in the above instruction: (1) the address of the jump target represented by the expression 'REPEAT' is not known to the assembler when it encounters the LJZB instruction on its first pass and (2) the displacement to the jump target is outside a −128, +127 byte. A JZB instruction would be flagged as an error if it were coded here since the assembler assumes a −128, +127 byte displacement range when the jump target address is not known.

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
00010AA0	111001MM	offset if AA=01	sdisp-low	sdisp-high

Execution Time:

12 clocks

NOTE 1) A jump target must be within a range of −32,7678, +32,767 bytes from the end of a control transfer instruction. There is NO wraparound from the end of the 64k program instruction space to the beginning.

Load Pointer From Memory

Mnemonic: LPD **Coding Format:** LPD P, M

Operands: 'P' is a pointer/register symbol
'M' is a data memory expression

Operation: 1) 20-bit address ← (M)
low order word offset; high order word segment

2) (P) ← 20-bit address

3) P's tag bit ← 0

A 20-bit address is formed from two consecutive words of data memory beginning at 'M'. The first memory word, an offset value is added to the second (segment) word, which is shifted left four bit positions, in the same manner a 20-bit address is formed from a 16-bit offset and a 16-bit segment address by the 8086. The 20-bit address is loaded into pointer/register 'P'.

The pointer/register's tag bit is cleared to zero, indicating a 20-bit system (memory) space address.

Example:

In this example, the pointer/register GA is loaded with a 20-bit address formed from two consecutive words of data memory located in the Parameter Block and pointed to by an offset from the PP register.

```
LPD   GA, [PP].12                ;Four consecutive bytes beginning at location
                                 ;[PP] + 12 are used to form a 20-bit address that
                                 ;is loaded into GA (GA's tag bit is
                                 ;cleared to zero).
```

Assembled Instruction:

7 0	7 0	7 0
P P P 0 0 A A 1	1 0 0 0 1 0 M M	offset if AA=01

Execution Time:

20 clocks if address is even
28 clocks if address is odd

NOTES 1) The LPD instruction is used to form a 20-bit address from a 16-bit offset value and a 16-bit segment address. Once the 20-bit address has been created, it cannot be disassembled into the two 16-bit values used to create it.

2) Twenty bit addresses can be stored in and restored from memory using the 'MOVP' instruction.

LPDI

Load Pointer From Immediate Data

Mnemonic: LPDI **Coding Format:** LPDI P, I

Operands: 'P' is a pointer/register symbol
'I' is an expression which may contain external symbol

Operation: 1) 20-bit address ← (I) + 16-bit segment address

2) (P) ← 20-bit address

3) P's tag bit ← 0

'I' is an expression which can contain an external symbol. An external symbol appearing in 'I' must be added (not subtracted) in the expression.

The expression 'I' is evaluated modulo 641 and supplies a 16-bit offset value . This offset value is added to a 16-bit segment address, which is shifted left four bit positions, in the same manner that a 20-bit address is formed by the 8086.

If 'I' contains an external symbol, the 16-bit offset value and segment address are resolved by relocate and link (LOC86, LINK86) processing of the object module. If 'I' does not contain any external symbols, the 16-bit segment address, supplied by LOC86, is the load origin of the 8089 program.

Note that the assembler allocates four bytes for the offset and segment data when the LPDI instruction is processed. The contents of these four bytes are not defined until the object module has been linked, if necessary, and located.

The pointer/register's tag bit is cleared to logical '0', indicating a 20-bit system (memory) space address.

Examples:

```
EXTRN     DATA_TABLE              ;Assembler directive identifying
                                  ;DATA_TABLE ;as a symbol defined as
                                  ;public in another module.

.

.

.

LPDI  GB, DATA_TABLE              ;A 20-bit address formed from 16-bit offset and
                                  ;segment data provided by relocate and link
                                  ;processing of the external symbol
                                  ;'DATA_TABLE' is loaded into
                                  ;pointer/register GB.

LPDI  GC, 237FH                   ;Load pointer/register GC with a 20-bit address
                                  ;formed using 237FH as the offset value and the
                                  ;load origin of the 8089 program as the
                                  ;segment address.
```

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0	7 0
PPP10001	00001000	offset (low)	offset (high)	segment (low)	segment (high)

Execution Time:

12 clocks if instruction begins on even address
16 clocks if instruction begins on odd address

NOTES 1) Once a 20-bit address has been formed it cannot be disassembled again into its two 16-bit components.

2) A 20-bit pointer/register and tag bit can be stored in, or restored from, data memory using the 'MOVP' instruction.

Move Register to Memory Word
Move Memory Word to Register
Move Memory Word to Memory Word

Mnemonic: MOV	**Coding Format:**	MOV M, R	
		MOV R, M	
		MOV M, M	

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: a) (OP1) ← (OP2)

b) If OP1 = GA, GB, GC or TP *pointer/registers*
then (OP1) ← sign-extended (OP2) *two 20-bit quantities*

OP1's tag bit ← 1

A word (16-bits) is copied from OP2 to OP1. The source data, (OP2), remains unchanged.

If a pointer/register (GA, GB, GC, or TP) is used as the destination operand, OP1, the sign bit, bit-15, is extended into the upper four bits (bits 16–19) of the pointer/register. The pointer/register's tag bit is also set to a logical one, indicating a local (I/O) space, 16-bit address.

If a 20-bit pointer/register is used as a source operand, 'OP2', only bits 0-15 are copied to memory. The high order bits, bits 16-19, are ignored.

Examples:

```
MOV  GB, [GC].2              ;Move the word of data memory beginning
                             ;(low-order byte) at [GC] + 2 to pointer/register
                             ;GB.

MOV  [GC], IX                ;Move the contents of the Index register to the
                             ;memory location pointed to by the contents of
                             ;[GC].

MOV  [GB+IX+], [GA+IX]       ;Move the word of data memory beginning
                             ;(low-order byte) at the location specified by
                             ;register GA + the Index register to the
                             ;location specified by register GB + the Index
                             ;register; Index register post auto-incremented
                             ;by 2 (word operation).
```

Assembled Instruction:

MOV M, R (MOVE REGISTER TO MEMORY WORD)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 0 0 0 1 M M	offset if AA=01

Execution Time:

10 clocks if bus width = 16 bits and address is even
16 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

MOV R, M (MOVE MEMORY WORD TO REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 0 0 0 0 M M	offset if AA=01

Execution Time:

8 clocks if bus width = 16 bits and address is even
12 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

MOV M, M (MOVE MEMORY WORD TO MEMORY WORD)

7 0	7 0	7 0	7 0	7 0	7 0
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
(SOURCE)			(DESTINATION)		

Execution Time:

18 clocks if bus width = 16 bits and address is even
28 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) 20-bit pointer/registers and their tag bits can be stored in, or restored from, memory using the 'MOVP' instruction.

MOVB M, R

Move Register to Memory Byte

Mnemonic: MOVB **Coding Format:** MOVB M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (M) ← truncated (R) *high order register byte truncated*

The high order byte of register 'R' (high order byte plus four bits in the case of pointer/registers GA, GB, GC or TP) is truncated and the least significant byte is placed in the data memory byte at location 'M'.

Example:

```
MOVB [GB], BC          ;Move least significant byte of register BC to
                       ;data memory byte pointed at by GB.
```

Assembled Instruction:

MOVB M, R (MOVE REGISTER TO MEMORY BYTE)

7 0	7 0	7 0
R R R 0 0 A A 0	1 0 0 0 0 1 M M	offset if AA=01

Execution Time:

10 clocks

NOTES 1) Use the 'MOV' instruction for 16-bit data.

2) 20-bit pointer/registers and their tag bits can be stored in or restored from memory using the 'MOVP' instruction.

Move Memory Byte to Register

Mnemonic: MOVB **Coding Format:** MOVB R, M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation:) (R) ← sign-extended (M)

b) If OP1 = GA, GB, GC, or TP *pointer/registers*
then (OP1) ← sign-extended (OP2) *two 20-bit quantities*

OP1's tag bit ← 1

The data memory byte located at 'M' is sign-extended (bit 7) to 16 bits. The sign-extended quantity is copied to the specified register 'R'.

If 'R' is a 20-bit pointer/register, the data is sign-extended to 20 bits and copied to 'R'. The pointer/register's tag bit is set to logical one, indicating a 16-bit local (I/O) space address.

Example:

```
MOVB   MC, [GC+IX]                 ;Register MC is loaded with a sign-extended
                                   ;copy of the byte at location [GC+IX].
```

Assembled Instruction:

MOVB R, M (MOVE MEMORY BYTE TO REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 0	1 0 0 0 0 0 M M	offset if AA=01

Execution Time:

8 clocks

NOTES 1) Use the 'MOV' instruction for 16-bit data.

2) 20-bit pointer/registers and their tag bits can be stored in or restored from memory using the 'MOVP' instruction.

MOVB M, M

Move Memory Byte to Memory Byte

Mnemonic: MOVB **Coding Format:** MOVB M, M

Operands: 'M' is a data memory expression

Operation: (OP1) ← (OP2)

The contents of the data memory byte source, OP2, are copied to the data memory byte destination, OP1.

Example:

```
MOVB  [GB], [GC+IX]          ;The data memory byte at [GC+IX] is copied to
                             ;the data memory location [GB].
```

Assembled Instruction:

MOVB M, M (MOVE MEMORY BYTE TO MEMORY BYTE)

7 0	7 0	7 0	7 0	7 0	7 0
00000AA0	100100MM	offset if AA=01	00000AA0	110011MM	offset if AA=01
(SOURCE)			(DESTINATION)		

Execution Time:

18 clocks

NOTES 1) Use the 'MOV' instruction for 16-bit data.

2) 20-bit pointer/registers and their tag bits can be stored in or restored from memory using the 'MOVP' instruction.

MOVBI R, I

Move Immediate Byte to Register

Mnemonic: MOVBI **Coding Format:** MOVBI R, I

Operand Format: 'R' is a register symbol
'I' is an expression evaluated modulo 256

Operation: 1) (R) ← sign-extended (i-value)

```
2) If OP1 = GA, GB, GC, TP    *pointer/registers*
     then (OP1) ← sign-extended (OP2)    *two 20-bit quantities*

   OP1's tag bit ← 1
```

The expression 'I' is evaluated modulo 256 to an immediate signed byte value, 'i-value'. This value is sign-extended (bit 7) to 16-bits, or, if 'R' is a pointer/register (GA, GB, GC or TP), to 20-bits. The sign extended value is placed in the specified register, 'R'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP), its tag bit is set to a logical one, indicating a 16-bit local (I/O) space address.

Example:

```
MOVBI   BC, -128                  ;Place 80H (-128 decimal in two's complement
                                  ;form) in register BC.
```

Assembled Instruction:

MOVBI R, I (MOVE IMMEDIATE BYTE TO REGISTER)

7 0	7 0	7 0
RRR01000	00110000	i-value

Execution Time:

3 clocks

NOTE 1) Use the 'MOVI' instruction for 16-bit immediate values.

MOVBI M, I

Move Immediate Byte to Memory Byte

Mnemonic: MOVBI **Coding Format:** MOVBI M, I

Operands: 'M' is a data memory expression
'I' is an expression evaluated modulo 256

Operation: (M) ← i-value

The expression 'I' is evaluated modulo 256 to an immediate signed byte value, 'i-value'. This value is placed in the data memory byte located at 'M'.

Example:

```
MOVBI  [GC].7, 15                ;0FH is placed in the data memory byte at
                                 ;location [GC] + 7.
```

Assembled Instruction:

MOVBI M, I (MOVE IMMEDIATE BYTE TO MEMORY BYTE)

7 0	7 0	7 0	7 0
00001AA0	010011MM	offset if AA=01	i-value

Execution Time:

12 clocks

NOTE 1) Use the 'MOVI' instruction for 16-bit immediate values.

Move Immediate Word to Register
Move Immediate Word to Memory Word

Mnemonic: MOVI

Coding Format:
```
MOVI  R,  I
MOVI  M,  I
```

Operands: 'R' is a register symbol
'M' is a data memory expression
'I' is an expression evaluated modulo 64k

Operation: a) (OP1) ← i-value

b) If OP1 is a pointer/register (GA, GB, GC or TP)
(OP1) ← sign-extended (i-value) *sign-extended to 20-bits*

OP1's tag bit ← 1

The expression 'I' is evaluated modulo 64k to an immediate signed word value, 'i-value'. The immediate signed word value is placed in the specified register 'R' or the word of data memory beginning (low-order byte) at location 'M'.

If 'OP1' is a 20-bit pointer/register, (GA, GB, GC or TP), the 'i-value' is sign extended (bit 15) into the upper four bits (16-19). The pointer/register's tag bit is set to a logical one, indicating a 16-bit local (I/O) space address.

Examples:

```
INPUT_COUNT    EQU    1500H              ;Define an 'INPUT_COUNT' and assign
                                         ;it a value of 1500H.
               .
               .
               .
               MOVI   BC, INPUT_COUNT    ;Move the value 1500H into register BC.

               MOVI   [GB].4, 32555      ;Move the value 32555 into the word of
                                         ;data memory beginning (low-order byte)
                                         ;at [GB] + 4.
```

Assembled Instruction:

MOVI R, I (MOVE IMMEDIATE WORD TO REGISTER)

7 0	7 0	7 0	7 0
RRR10001	00110000	i-value (low)	i-value (high)

Execution Time:

3 clocks

MOVI M, I (MOVE IMMEDIATE WORD TO MEMORY WORD)

7 0	7 0	7 0	7 0	7 0
00010AA1	010011MM	offset if AA=01	i-value (low)	i-value (high)

Execution Time:

12 clocks if bus width = 16 bits and address is even
18 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTE 1) Use the 'MOVBI' instruction for immediate byte values.

MOVP M, P

Move Pointer to Memory (Store)

Mnemonic: MOVP **Coding Format:** MOVP M, P

Operands: 'P' is a pointer/register symbol
'M' is a data memory expression

Operation: 1) (M) ← (P)

2) (M) ← P's tag bit

The contents of the specified 20-bit pointer/register and its tag bit are stored in three consecutive data memory bytes beginning at the given memory location, 'M'. (See NOTES below for the format of the stored pointer/register).

Example:

```
POINTER_STORE:  DS  3                      ;Reserve three bytes of data memory
                                           ;with the name 'POINTER_STORE'.

                .

                .

                .

                MOVI  GA, POINTER_STORE    ;Load location of 'POINTER_STORE'
                                           ;into register GA.

                MOVP  [GA], TP             ;Move 'TP' to [GA].
```

Assembled Instruction:

MOVP M, P (MOVE POINTER/REGISTER TO MEMORY)

7 0	7 0	7 0
P P P 0 0 A A 1	1 0 0 1 1 0 M M	offset if AA=01

Execution Time:

16 clocks if bus width = 16 bits and address is even
22 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) The pointer/register and tag bit are stored in the following format:

7 0	7 0	7 0
pointer-low	pointer-high	19181716tb 0 0 0

The low order byte of the pointer/register, 'pointer-low', is stored in the first memory byte. The next byte of the pointer/register, 'pointer-high', is stored in the second memory byte. The four high order bits of the pointer/register, bits 16-19, are stored in bits 4-7 of the third memory byte. The tag bit is stored in bit 3 of the third memory byte. Bits 0-2 of the third memory byte are cleared to zero.

Move Memory to Pointer (Restore)

Mnemonic: MOVP **Coding Format:** MOVP P, M

Operands: 'P' is a pointer/register symbol
'M' is a data memory expression

Operation: 1) (P) ← (M)

2) P's tag bit ← stored tag bit

A stored 20-bit pointer/register and tag bit value are restored to pointer/register 'P' from three consecutive bytes of data memory beginning at memory location 'M'.(See NOTES below for the format of the stored pointer/register).

Example:

```
STORE_POINTER DS  3                      ;Reserve three bytes of data memory named
                                         ;'STORE_POINTER'.
              .
              .
              .
              MOVI   GB, STORE_POINTER   ;Load GB with address of three data memory
                                         ;bytes named 'STORE_POINTER'.
              MOVP   [GB], GA            ;Store 20-bit pointer/register GA and tag bit
                                         ;in three bytes of data memory beginning at
                                         ;location [GB].
              .
              .
              .
              MOVP   GA, [GB]            ;Restore pointer/register GA and tag bit
                                         ;from three bytes of data memory beginning
                                         ;at location [GB].
```

Assembled Instruction:

MOVP P, M (MOVE MEMORY TO POINTER/REGISTER)

7 0	7 0	7 0
P P P 0 0 A A 1	1 0 0 0 1 1 M M	offset if AA=01

Execution Time:

19 clocks if even address
27 clocks if odd address

MOVP P, M

NOTES 1) The pointer/register and tag bit are stored in the following format:

7 0	7 0	7 0
pointer-low	pointer-high	19 18 17 16 tb 0 0 0

The low order byte of the pointer/register, ‘pointer-low’, is stored in the first memory byte. The next byte of the pointer/register, ‘pointer-high’, is stored in the second memory byte. The four high order bits of the pointer/register, bits 16-19, are stored in bits 4-7 of the third memory byte. The tag bit is stored in bit 3 of the third memory byte. Bits 0-2 of the third memory byte are cleared to zero.

No Operation

Mnemonic: NOP **Coding Format:** NOP

Operands: This instruction has no operands.

Operation: None

This instruction takes four clock cycles but performs no operation.

Example:

```
NOP                          ;No operation performed, four clock cycles are used.
```

Assembled Instruction:

```
7          0 7          0
| 00000000 | 00000000 |
```

Execution Time:

4 clocks

NOT

Complement Register
Complement Memory Word
Complement Memory Word; Put Result in Register

Mnemonic: NOT

Coding Format:
```
NOT  R
NOT  M
NOT  R,  M
```

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: a) (OP1) ← NOT (OP1)
OR
b) (R) ← NOT (M)

The contents register 'R' or the word of data memory beginning (low-order byte) at location 'M' are complemented. Any logical '1' becomes a logical '0'. Any logical '0' becomes a logical '1'.

The result of complementing a data memory word may be placed in a register rather than returned to the original memory location. Two operands are then required: a register operand 'R', the destination (OP1), and a data memory operand 'M', (OP2).

If 'R' is a 20-bit pointer/register the upper four bits, bits 16-19, of the result are undefined following its complement. Any data placed in a pointer/register is sign-extended to 20 bits.

Examples:

```
NOT  IX               ;Complement register 'IX'.

NOT  [GB]             ;Complement word of data memory beginning
                      ;(low-order byte) at location [GB].

NOT  GA, [GC+IX]      ;Complement the word of data memory
                      ;beginning (low-order byte) at [GC+IX] and
                      ;put result in register GA.
```

Assembled Instruction:

NOT R (COMPLEMENT REGISTER)

7 0	7 0
RRR00000	00101100

Execution Time:

3 clocks

NOT M (COMPLEMENT MEMORY WORD)

7 0	7 0	7 0
00000AA1	110111MM	offset if AA=01

Execution Time:

16 clocks if bus width = 16 bits and address is even
26 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOT R, M (COMPLEMENT MEMORY WORD; PUT RESULT IN REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 1 0 1 1 M M	offset if AA=01

Execution Time:

11 clocks if bus width = 16 bits and address is even
15 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) The complement operation sets any logical zero in the input data to a logical one. Any logical one in the input data is cleared to a logical zero.

Example:

Complement 0ADH

Before complement:

7 0
1 0 1 0 1 1 0 1

After complement:

7 0	
0 1 0 1 0 0 1 0	(52H)

2) The two's complement of a register or a word of data memory can be formed by adding '1' to the result of a NOT instruction.

3) The ability to complement a word of memory data and place the result in a register can save bus cycles, especially when doing two's complement arithmetic, since one instruction can be eliminated.

Example:

```
OPERAND:  DW    2314H          ;Define a word of data memory which will
                               ;supply an operand in a two's
                               ;complement operation.

          MOVI  GA, OPERAND    ;Load address of data memory operand
                               ;into GA.

          NOT   GC, [GA]       ;Form one's complement of operand in
                               ;memory.

          INC   GC             ;GC now contains two's complement of
                               ;memory operand.
```

NOTB

Complement Memory Byte
Complement Memory Byte; Put Result In Register

Mnemonic: NOTB

Coding Format: NOTB M
NOTB R, M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: a) (M) ← NOT (M)
OR
b) (R) ← sign-extended NOT (M)

The data memory byte located at 'M' is complemented. Any logical one is cleared to logical zero. Any logical zero is set to logical one.

The result of the complement can be put in a register, 'R', rather than returned to the original memory location. The complement result is sign extended (bit 7) to 16-bits, or, if 'R' is a pointer/register, to 20-bits, and placed in the specified register.

Examples:

```
NOTB  [PP].8          ;Complement data memory byte at
                      :location [PP] + 8.

NOTB  MC, [GA]        ;Complement data memory byte at
                      :[GA]; put result in register MC.
```

Assembled Instruction:

NOTB M (COMPLEMENT MEMORY BYTE)

7 0	7 0	7 0
00000AA0	110111MM	offset if AA=01

Execution Time:

16 clocks

NOTB R, M (COMPLEMENT MEMORY BYTE; PUT RESULT IN REGISTER)

7 0	7 0	7 0
RRR00AA0	101011MM	offset if AA=01

Execution Time:

11 clocks

NOTES 1) The complement operation sets any logical zero in the input data to a logical one. Any logical one in the input data is cleared to a logical zero.

Example:

Complement 3BH

Before complement:

```
7        0
┌──────────┐
│ 00111011 │
└──────────┘
```

After complement:

```
7        0
┌──────────┐
│ 11000100 │  (0C4H)
└──────────┘
```

2) The two's complement of a data memory byte can be formed by adding '1' to the result of a NOTB instruction.

3) Use the 'NOT' instruction to complement a register or a word of data memory.

4) The ability to complement a byte of memory data and place the result in a register can save bus cycles, especially when doing two's complement arithmetic since one instruction can be eliminated.

Example:

```
OPERAND:  DB    0B7H          ;Define a byte of data memory which will
                              ;supply an operand in a two's
                              ;complement operation.

          MOVI  GA, OPERAND   ;Load address of data memory operand
                              ;into GA.

          NOTB  GC, [GA]      ;Form one's complement of operand in
                              ;memory.

          INC   GC            ;GC now contains two's complement of
                              ;memory operand.
```

OR Memory Word to Register
OR Register to Memory Word

Mnemonic: OR

Coding Format: OR R, M
OR M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (OP1) ← (OP1) OR (OP2)

The corresponding bit positions of the 16-bit input data are logically ORed. A logical '1' results if either or both input bit positions are a logical '1'. A logical '0' results if neither input bit position contains a logical '1'. The result is placed in the leftmost operand, OP1.

If the destination, OP1, is a 20-bit pointer/register (GA, GB, GC or TP) the upper four bits, bits 16-19, of the result are undefined following this operation.

Examples:

```
OR   MC, [GB]                 ;OR register MC with the word of data memory
                              ;beginning (low-order byte) at [GB]. The result
                              ;is placed in register MC.

OR   [GA].12, IX              ;OR the word of data memory beginning
                              ;(low-order byte) at [GA] + 12 with the IX
                              ;register. The result is placed in data memory
                              ;beginning (low-order byte) at location [GA]
                              ;+ 12.
```

Assembled Instruction:

OR R, M (OR MEMORY WORD TO REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 1	1 0 1 0 0 1 M M	offset if AA=01

Execution Time:

11 clocks if bus width = 16 bits and address is even
15 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

OR M, R (OR REGISTER TO MEMORY WORD)

7 0	7 0	7 0
R R R 0 0 A A 1	1 1 0 1 0 1 M M	offset if AA=01

Execution Time:

16 clocks if bus width = 16 bits and address is even
26 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) A logical OR of two binary digits outputs a logical one if either or both input binary digits is a logical one. A logical zero is output if neither input binary digit is a logical one.

Example:

OR 0EBH and 91H

0EBH	$\boxed{11101011}$	
OR		
91H	$\boxed{10010001}$	
RESULT	$\boxed{11111011}$	0FBH

2) See ORB instruction for logical OR with byte data.

ORB R, M

OR Memory Byte to Register

Mnemonic: ORB **Coding Format:** ORB R, M

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (R) ← (R) OR sign-extended (M)

The data memory byte located at 'M' is sign-extended (bit 7) to 16-bits. The sign-extended memory byte is ORed with the specified register 'R'. A logical one is output where one or both input bits are a logical one. A logical zero is output if both input bits are a logical zero. The 16-bit result is placed in register 'R'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP) the upper four bits (bits 16-19) are undefined following this operation.

Examples:

```
OR   MC, [GB].4              ;OR register MC with data memory byte at [GB] + 4.
```

Assembled Instruction:

ORB R, M (OR MEMORY BYTE TO REGISTER)

7 0	7 0	7 0
R R R 0 0 A A 0	1 0 1 0 0 1 M M	offset if AA=01

Execution Time:

11 clocks

NOTES 1) A logical OR of two binary digits outputs a logical one if either or both input binary digits is a logical one. A logical zero is output if neither input binary digit is a logical one.

Example:

OR 1DH and 24H

1DH	00011101	
OR		
24H	00100100	
RESULT	00111101	3DH

2) See 'OR' instruction for logical OR with a register and 16-bit memory data.

OR Register To Memory Byte

Mnemonic: ORB **Coding Format:** ORB M, R

Operands: 'R' is a register symbol
'M' is a data memory expression

Operation: (M) ← (M) OR low-order byte (R)

The data memory byte located at 'M' is ORed with the low-order byte of 'R'. A logical one is output where either or both input bits are a logical one. A logical zero is output if both input bits are a logical zero. The 8-bit result is placed in data memory at location 'M'.

Examples:

```
ORB  [GC], IX     ;OR data memory byte at [GC] with the low-order byte of register IX.
```

Assembled Instruction:

ORB M, R (OR REGISTER TO MEMORY BYTE)

7 0	7 0	7 0
R R R 0 0 A A 0	1 1 0 1 0 1 M M	offset if AA=01

Execution Time:

16 clocks

NOTES 1) A logical OR of two binary digits outputs a logical one if either or both input binary digits is a logical one. A logical zero is output if neither input binary digit is a logical one.

Example:

OR 5CH and 8BH

5CH	0 1 0 1 1 1 0 0	
OR		
8BH	1 0 0 0 1 0 1 1	
RESULT	1 1 0 1 1 1 1 1	0DFH

2) See OR instruction for logical OR with a register and a 16-bit memory data.

ORBI R, I

OR Immediate Byte To Register

Mnemonic: ORBI **Coding Format:** ORBI R, I

Operands: 'R' is a register symbol
'I' is an expression evaluated modulo 256

Operation: (R) ← (R) OR sign-extended (i-value)

The expression 'I' is evaluated modulo 256 to an immediate signed byte value, 'i-value', (−128 <= i-value <= +127). 'i-value' is sign-extended (bit 7) to 16-bits and ORed with the specified register 'R'. A logical one is output where one or both input bits are a logical one. A logical zero is output if both input bits are a logical zero. The 16-bit result is placed in register 'R'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP) the upper four bits, bits 16-19, are undefined following this operation.

Example:

```
ORBI   CC, 7FH                          ;OR register CC with 7FH.
```

Assembled Instruction:

ORBI R, I (OR IMMEDIATE BYTE TO REGISTER)

7 0	7 0	7 0
RRR01000	00100100	i-value

Execution Time:

3 clocks

NOTES 1) A logical OR of two binary digits outputs a logical one if either or both input binary digits is a logical one. A logical zero is output if neither input binary digit is a logical one.

Example:

OR 51H and 4AH

51H	01010001	
OR		
4AH	01001010	
RESULT	01011011	5BH

2) See 'ORI' instruction for logical OR with 16-bit immediate values.

OR Immediate Byte to Memory Byte

Mnemonic: ORBI **Coding Format:** ORBI M, I

Operands: 'M' is a data memory expression
'I' is an expression evaluated modulo 256

Operation: (M) ← (M) OR i-value

The expression 'I' is evaluated modulo 256 to an immediate signed byte value 'i-value', (−128 <= i-value <= +127). 'i-value' is ORed with the data memory byte located at 'M'. A logical one is output where one or both input bits are a logical one. A logical zero is output if both input bits are a logical zero. The result is placed in the data memory byte at location 'M'.

Examples:

```
ORBI  [GA], 25                    ;OR the data memory byte at [GA] with 25.
```

Assembled Instruction:

ORBI M, I (OR IMMEDIATE BYTE TO MEMORY BYTE)

7 0	7 0	7 0	7 0
00001AA0	110001MM	offset if AA=01	i-value

Execution Time:

16 clocks

NOTES 1) A logical OR of two binary digits outputs a logical one if either or both input binary digits is a logical one. A logical zero is output if neither input binary digit is a logical one.

Example:

OR 95H and 17H

95H	10010101	
OR		
17H	00010111	
RESULT	10010111	97H

2) See 'ORI' instruction for logical OR with 16-bit immediate data.

ORI

OR Immediate Word to Register
OR Immediate Word to Memory Word

Mnemonic: ORI **Coding Format:** ORI R, I
ORI M, I

Operands: 'R' is a register symbol
'M' is a data memory expression
'I' is an expression evaluated modulo 64k

Operation: (OP1) ← (OP1) OR i-value

The expression 'I' is evaluated modulo 64k to an immediate signed word value, 'i-value', (−32,768 <= i-value <= +32,767). 'i-value' is ORed with the register 'R' or the word of data memory beginning (low-order byte) at 'M'. A logical one is output where one or both input bits are a logical one. A logical zero is output if both input bits are a logical zero. The result is placed in OP1, the register 'R' or the word of data memory beginning (low-order byte) at 'M'.

If 'R' is a 20-bit pointer/register (GA, GB, GC or TP), the upper four bits (bits 16-19) are undefined following this operation.

Examples:

```
ORI  BC,  2D4EH            ;OR register BC with 2D4EH.

ORI  [GB], 9091H           ;OR word of data memory beginning (low-order
                           ;byte) at [GB] with 9091H.
```

Assembled Instruction:

ORI R, I (OR IMMEDIATE WORD TO REGISTER)

7 0	7 0	7 0	7 0
RRR10001	00100100	i-value (low)	i-value (high)

Execution Time:

3 clocks

ORI M, I (OR IMMEDIATE WORD WITH MEMORY WORD)

7 0	7 0	7 0	7 0	7 0
00010AA1	110001MM	offset if AA=01	i-value (low)	i-value (high)

Execution Time:

16 clocks if bus width = 16 bits and address is even
26 clocks if bus width = 8 bits or bus width = 16 bits and address is odd

NOTES 1) A logical OR of two binary digits outputs a logical one if either or both input binary digits is a logical one. A logical zero is output if neither input binary digit is a logical one.

Example:

OR 09H and 42H

09H $\boxed{00001001}$

OR

42H $\boxed{01000010}$

RESULT $\boxed{01001011}$ 4BH

2) See 'ORBI' instruction for logical OR with immediate byte data.

SETB

Set Selected Bit to Logical 1

Mnemonic: SETB **Coding Format:** SETB M, b

Operands: 'b' is the bit position in the data memory byte (0 <= b <= 7)
'M' is a data memory expression

Operation: b ← 1

The selected bit of a data memory byte located at 'M' is set to logical one.

Examples:

```
SETB  [PP].14, 5                    ;Set bit 5 of [PP] + 14 to logical one.
```

Assembled Instruction:

7 0	7 0	7 0
b b b 0 0 A A 0	1 1 1 1 0 1 M M	offset if AA=01

Execution Time:

16 clocks

NOTES 1) Bit positions within a data memory byte are specified as follows:

	MSB LSB
bit positions	7 6 5 4 3 2 1 0

Set Interrupt Service Flip-Flop

Mnemonic: SINTR **Coding Format:** SINTR

Operands: This instruction has no operands.

Operation: Interrupt service flip-flop ← 1

Set the interrupt service flip-flop. If interrupts from this channel are enabled, the external SINTR space pin for the channel (SINTR 1 or SINTR 2) is activated. Channel interrupts are enabled through the Interrupt Control Field (ICF) in the Channel Control word (CCW), located in the Channel Control Block.

Example:

In conjunction with the Interrupt Control Field of the Channel Control Word (CCW), located in the Channel Control Block, the SINTR instruction can be used to indicate to the main system interrupt hardware the occurence of user defined events.

In this example a status byte is set to '0FFH' by an I/O device upon the suuccessful completion of an operation. The task block program checks the status byte for '00H' indicating the unsuccessful completion of an operation by the I/O device. If '00H' is detected, a jump is made to an error routine which places an error message in a byte located in the user-defined area of the Parameter Block and, using the SINTR instruction, sets the channel's interrupt service flip-flop. (This example assumes that interrupts for the channel have been enabled.)

```
GOOD??:         DB   00H              ;Define a byte in data memory named
                                      ;'GOOD??' where the I/O device will place its
                                      ;completion status.

ERROR           EQU  7FH              ;Define a name 'ERROR' with a value of 7FH.

                .

                .

                .         (A status byte is written to data memory
                           location named 'GOOD??' by an I/O device
                           upon the completion of some operation.)

                MOVI  GB, GOOD??      ;Load address of data memory byte named
                                      ;'GOOD??' into register GB.

                LJZB  [GB], E_ROUT    ;Test status byte for '00H'; jump to instruction
                                      ;labeled 'E_ROUT' if '00H' found.

                .

                .         (12,000 bytes of assembled source program statements.)

                .
```

```
E_ROUT:     MOVBI   [PP].18, ERROR    ;Place 7FH in Parameter Block byte at [PP]
                                      ;+ 18.

            SINTR                     ;Set interrupt service flip-flop; the error
                                      ;message in the Parameter Block can be read
                                      ;by the interrupt service routine and the
                                      ;necessary action taken.
```

Assembled Instruction:

```
7          0 7          0
| 01000000 | 00000000 |
```

Execution Time:

4 clocks

Test and Set While Locked

Mnemonic: TSL **Coding Format:** TSL M, I, L

Operands: 'M' is a data memory expression
'I' is an expression evaluated modulo 256
'L' is an expression representing the jump target which is within a range of −128, +127 bytes of the next instruction

Operation: 1) System bus remains locked during instruction execution

```
2) If (M) = 0
     then (M) ← i-value

   Else (TP) ← (TP) + sdisp (sign-extended to 20-bits)
```

'L', the jump target, is an expression representing a location within the program. 'L' is converted to a signed byte displacement, 'sdisp', the distance (in bytes) from the end of this instruction to the jump target. The value of 'sdisp' ranges from −128 to +127.

The expression 'I' is evaluated module 256 to an immediate signed byte value, 'i-value', (−128 <= i-value <= +127).

The contents of a data memory byte located at 'M' are examined. If equal to logical zero, the immediate value, 'i-value', is placed in the data memory byte location, 'M'. If the contents of the byte are not equal to logical zero, a jump is made to 'L' by adding the signed byte displacement, 'sdisp', to the TP register, forming the jump target address. (The address of the next sequential instruction is in the TP register when the jump target address is formed.)

The system bus remains locked throughout the entire instruction execution. A simple semaphore mechanism can be implemented using this instruction.

Example:

In systems with shared resources, mechanisms for controlling access to these resources are necessary. Such a mechanism can be provided using the TSL instruction to implement a simple semaphore. The following is an example of how such a mechanism might function.

Two I/O channels share a data table containing blocks of control parameters read and updated by each channel. To prevent one channel from reading the control parameter blocks while another is updating them, a data memory byte is used to signal when the data table is being used (0FFH in data memory byte) or is free (00H in data memory byte). Before accessing the data table, each channel tests the data memory byte. If it is in use, the channel loops until the data table is free. When the data table is found free, i.e. 00H is in the data memory byte, 0FFH is written to the data memory byte and the data table is accessed. By locking the system bus, the TSL instruction insures that the other channel will not begin to use the data table between the time it is found free and the time the in-use condition is signalled.

```
BUSY:          DB   00H                 ;Define a data memory byte named 'BUSY'
                                        ;used to signal the availability of the data table.

DATA_TABLE:    DS   200                 ;Reserve 200 bytes of data memory named
                                        ;'DATA_TABLE'.

               .

               .

               .

               MOVI   GB, BUSY          ;Load register GB with address of data memory
                                        ;byte.

FREE?:         TSL   [GB], 0FFH, LOOP   ;Test data memory byte; if equal to 00H (free)
                                        ;move 0FFH to the data memory byte,
                                        ;otherwise jump to instruction labeled 'LOOP'.

               .

               .

               .

LOOP:          JMP   FREE?              ;Retry test of data memory byte.
```

Assembled Instruction:

7 0	7 0	7 0	7 0	7 0
0 0 0 1 1 A A 0	1 0 0 1 0 1 M M	offset if AA=01	i-value	sdisp

Execution Time:

14 clocks if the data memory byte, located at 'M', does not equal zero
16 clocks if the data memory byte, located at 'M', does equal zero

NOTE 1) There is NO wraparound from the end of the 64k program instruction space to the beginning.

Set Source and Destination Logical Widths

Mnemonic: WID **Coding Format:** WID S, D

Operands: 'S' is a value indicating the DMA source logical width (8 or 16)
'D' is a value indicating the DMA destination logical width (8 or 16)

Operation: Source Logical Width ← (OP1)
Destination Logical Width ← (OP2)

The WID instruction specifies the source and destination logical widths (in bits) for DMA transfer. The 8089 optimizes DMA transfers by assembling or disassembling transferred bytes depending upon these logical widths (and also even/odd address boundaries). Logical widths and even/odd address boundaries determine the number of bytes transferred during a DMA transfer cycle.

In the assembled instruction a '1' for 'S' or for 'D' indicates a 16-bit device width is specified. A '0' for 'S' or for 'D' indicates an 8-bit device width is specified.

Example:

```
WID  16,  8                    ;Source logical width for DMA transfer is
                               ;16-bits; destination logical width is 8-bits
```

Assembled Instruction:

```
7        0 7        0
|1SD00000 | 00000000|
```

Execution Time:

4 clocks

NOTE 1) If any value other than '8' or '16' is used for 'S' or 'D' in this instruction, the value '8' is assumed and an error message is issued by the assembler.

Example:

```
WID  0,  0                     ;The logical source and destination widths are
                               ;both 8-bits. The assembly flags this instruction
                               ;as an error.
```

XFER

Enter DMA Transfer Mode After Execution of Next Instruction

Mnemonic: XFER **Coding Format:** XFER

Operands: This instruction has no operands.

Operation: None

DMA transfer mode is entered following the execution of the next instruction. To ensure the correct operation of the DMA transfer mode, the next instruction must not load the GA, GB or CC registers.

Example:

It is important to ensure that the channel is ready to transfer data as soon as a peripheral is granted permission to issue DMA requests. Some 8080 type peripherals may start issuing DMA requests upon receipt of their last parameter. The XFER instruction is designed to handle such situations by forcing the channel into the transfer mode after the execution of the next sequential instruction. This allows the program to supply the last parameter to the peripheral immediately before entering DMA transfer mode.

Assembled Instruction:

```
7         0 7         0
| 01100000 | 00000000 |
```

Execution Time:

4 clocks

ASSEMBLED INSTRUCTION DECODING INFORMATION

RRR

000—GA	100—TP
001—GB	101—IX
010—GC	110—CC
011—BC	111—MC

bbb

000—Bit 0	(LSB)
001—Bit 1	
010—Bit 2	
011—Bit 3	
100—Bit 4	
101—Bit 5	
110—Bit 6	
111—Bit 7	(MSB)

PPP

000—GA
001—GB
010—GC
100—TP

WB

00—Reserved
01—One immediate/displacement value byte
10—Two immediate/displacement value bytes
11—TSL Instruction only

AA Memory Address Mode

00—Base Address only [PREG]
01—Base Address + 8-bit offset [PREG].d
10—Base Address + Index Register [PREG + IX]
11—Base Address + Index Register;
Index Register post auto-incremented [PREG + IX +]

MM Base Memory Address

00—GA
01—GB
10—GC
11—PP

OPERANDS

REGISTER SYMBOLS	DATA MEMORY BIT SYMBOLS	POINTER/REGISTER SYMBOLS
	0 (LSB)	
BC GC	1	GA GC
CC IX	1	GB TP
GA MC	3	
GB TP	4	
	5	
	6	
	7 (MSB)	

DATA MEMORY EXPRESSIONS

[PREG]	— Base Address only PREG can be GA, GB, GC, or PP
[PREG].d	— 'd' is an expression, evaluated modulo 256 PREG + d = address
[PREG + IX]	— Base Address plus the Index Register PREG + IX = address
[PREG + IX +]	— Base Address plus the Index Register PREG + IX = address IX is post auto-incremented by 1 (byte) or 2 (word)

CHAPTER 4 ASSEMBLER DIRECTIVES

Introduction

This chapter describes the directives used to control the 8089 assembler in its generation of object code. The assembler directives discussed in this chapter are grouped as follows:

- Symbol Definition
  - EQU
- Data Definition and Memory Reservation
  - DB
  - DW
  - DD
  - DS
- Structure Definition
  - STRUC / ENDS
- Location Counter Control
  - ORG
  - EVEN
- Program Linkage
  - NAME
  - SEGMENT / ENDS
  - PUBLIC
  - EXTRN
- Assembler Termination
  - END

Assembler Directive Source Statement Format

Assembler directive source statements have the following general format:

```
[LABEL]  MNEMONIC    [OPERAND(S)]    [;COMMENT]
```

Items within brackets are not valid or required in every assembler directive. The description of each directive, found in the following sections, shows its required and optional elements, with optional items appearing in brackets. Comments are optional on any source line.

Assembler directive source statements, like instruction source statements, are made up of one or more source lines. A comment is optional on all source lines. An assembler directive source statement can be continued by placing an ampersand (&) as the first character of the next source line. Character string constants cannot be continued on another source line.

The assembler compresses each source statement as follows: all comments and the final end-of-line are deleted; tabs, and all sequences of unquoted blanks and end-of-line&'s are reduced to single blanks; all quoted quotes are changed into single quotes. The maximum number of characters in one compressed source statement is 256.

Examples:

```
DATA_TABLE:     DS     128          ;DATA_TABLE is a label.

IOP_CODE        SEGMENT             ;IOP_CODE is a name.

ELEVEN                              ;This assembler
&               EQU                 ;directive covers
&                      11           ;three source lines.
```

The assembler directive mnemonics are symbolic names for the various operations the assembler can be directed to perform. These mnemonics are reserved symbols and cannot be redefined. (For a complete list of reserved symbols see Appendix G.)

The following lists the assembler directive mnemonics and the operations they perform:

MNEMONIC	**OPERATION**
EQU	Defines a symbol and assigns a value to it.
DB	Defines byte(s) of data memory with 8-bit value(s).
DW	Defines word(s) of data memory with 16-bit values.
DD	Defines double word(s) of data memory for 20-bit address loading.
DS	Reserves bytes of data memory.
STRUC	Creates a template of offset values; no storage is allocated.
ORG	Sets the assembler's location counter to a specified integer value.
EVEN	Insures that the next instruction/directive begins on an even address boundary.
NAME	Assigns a name to the assembler-generated object module.
SEGMENT	Assigns a name to the segment (≤64k) containing the object code generated by the assembler.
PUBLIC	Identifies symbols defined in this program that are available to separately assembled or compiled programs.
EXTRN	Identifies symbols within this program which are defined and declared PUBLIC in separately assembled or compiled programs.
ENDS	Indicates the end of a SEGMENT or STRUC assembler directive.
END	Indicates the end of a source program.

Symbol Definition Directives

Symbols are often defined by appearing as a label to an assembly language instruction or assembler directive. The value of the assembler's location counter when the instruction or directive is assembled is automatically assigned to these symbols by the assembler. The assembler's location counter begins with a value of zero and is automatically incremented by the length of each instruction or the number of data memory bytes used by each data definition or memory reservation assembler directive.

The EQU assembler directive allows a programmer to define symbols and assign them values, which may differ from the assembler's location counter.

EQU Directive

The EQU assembler directive allows a user to define symbols and assign them values. Its format is:

```
name    EQU    expression
```

A name is *required* in the EQU directive. It must not be previously defined and cannot be redefined in the program.

The expression in an EQU directive cannot contain a forward reference; i.e., all symbols must be defined (in the lexical sense) when the directive is processed on the first assembler pass. Note that the special location counter reference symbol ($) is predefined to the assembler and is not a forward reference.

External symbols are not allowed in EQU expressions.

Examples:

```
TEN       EQU   10               ;Define a symbol TEN with a value of ten
                                 ;(decimal).

RECORD    EQU   DATA_BUFF        ;Define a symbol RECORD with the same value
                                 ;as the symbol DATA_BUFF.

RECORD2   EQU   DATA_BUFF + 2    ;Define a symbol RECORD2 with the value of
                                 ;symbol DATA_BUFF + 2.

START     EQU   $                ;Define a symbol START whose value is the
                                 ;current value of the assembler's location
                                 ;counter (equivalent to the statement START:).

ASCII_V   EQU   'AL'             ;Define a symbol ASCII_V with the ASCII value
                                 ;of AL (414CH) as its value.
```

The EQU directive can also be used to define a synonym for a register name. Symbols defined as synonyms for register names can only appear in the same contexts that the register name is allowed.

Examples:

```
SOURCE    EQU   GA               ;Define a symbol SOURCE synonymous with
                                 ;pointer/register symbol GA.

PARAM_B   EQU   PP               ;Define a symbol PARAM_B synonymous with
                                 ;register symbol PP.
```

Assembly time evaluation of EQU expressions is modulo 64k. Negative values are expressed in two's complement form. Values range from 0 to 0FFFFH or 0 to 65,535 decimal.

Examples:

```
MINUS_1   EQU  -1                   ;Define a symbol MINUS_1 with a value of
                                    ;0FFFFH (two's complement form of -1).

LARGEST   EQU  65535                ;Define a symbol LARGEST with a value of
                                    ;0FFFFH.

MOD_64k   EQU  122421               ;Define a symbol MOD_64k with the value
                                    ;0DE35H (122421 modulo 64k).
```

Data Definition and Memory Reservation Directives

The DB, DW and DD directives initialize data memory. The DS directive reserves data memory but does not initialize it.

A label is optional on all data definition and memory reservation directives.

DB Directive

The DB (define byte) directive stores the specified 8-bit values in consecutive data memory locations, starting at the current value of the location counter. It has the form:

```
[symbol:] DB                  d1[, d2, . . . , dn]
```

where 'd' is an expression or a character string constant. More than one expression or character string constant can be specified; each must be separated by a comma.

If the optional label is present, it is assigned the value of the assembler's location counter where the DB directive begins. It thus references the first byte stored by the DB directive.

Expressions are evaluated modulo 256. Negative values are expressed in two's complement form. Values range from 0 to 0FFH or 0 to 255 decimal.

The size of a character string constant is limited only by the size of the compressed source statement.

Examples:

Label (optional)		Operands	Assembled Code (Hex)
DATA_TABLE:	DB	1, 24Q, 15	01140F
	DB	'CHAR_string'	434841525F737472696E67
MARGIN:	DB	RATE + 10	(value of symbol RATE + 10)
NEGATIVE:	DB	-12	F4 (two's complement of -12)
MOD_256:	DB	1000	E8 (1000 decimal modulo 256)

NOTES:

1. The label DATA_TABLE references the first data memory byte stored by the DB directive, the data memory byte containing 01 (hexadecimal). DATA_TABLE + 1 references the data memory byte containing 14 (hexadecimal), the value of 24 octal.

2. The expression in the second DB directive contains a character string constant. Eleven bytes of data memory are initialized, each containing (in sequence) the ASCII code for a character. The assembler only distinguishes between upper- and lower-case letters within a character string. At all other times, upper- and lower-case letters are not differentiated.

DW Directive

The DW (define word) directive stores the 16-bit values specified by an expression list in fields of two consecutive bytes, starting at the current value of the location counter. The format of the DW directive is as follows:

```
[symbol:]   DW                          d1[, d2, . . . dn]
```

where 'd' is an expression. Expressions in an expression list must be separated by a comma.

If the optional label is present, it is assigned the value of the assembler's location counter where the DW directive begins. It thus references the low-order byte of the first 16-bit value stored by the DW directive.

Expressions in DW directives are evaluated modulo 64k. Negative values are represented in two's complement form. Values range from 0 to 0FFFFH or 0 to 65,535 decimal.

Character string constants containing one or two printable ASCII characters can appear in an expression list. The ASCII code for two characters is stored in reverse order (see example below).

The least significant byte (8 bits) of a 16-bit value is stored in the first data memory location. The most significant byte is stored in the next higher data memory location. If an expression evaluates to a single byte value it is assumed to be the low-order byte of a 16-bit value whose high-order byte is all zeros.

A sixteen bit local (I/O) address is stored low-order byte followed by high-order byte in data memory by the MOV instruction. The DW directive can be used to define a 16-bit address constant to be loaded into a pointer/register with the MOV instruction.

Examples:

LABEL (OPTIONAL)	OPERANDS	ASSEMBLED CODE
LARGE_COUNT:	DW 5280H	8052
SOME?VALUE:	DW 31	1F00
ZERO:	DW 65536	0000 (65,536 modulo 64k)
COMPLEMENT:	DW −1	FFFF (two's complement of −1)
TWO@CHARACTERS:	DW 'AB'	4241 (ASCII values of characters)

NOTES:

1. The label LARGE_COUNT references the first memory byte stored by the DW directive. In this example LARGE_COUNT references the data memory byte containing 80H, the low-order byte of the 16-bit value 5280H.

2. The DW directive above labeled TWO@CHARACTERS has an expression containing a character string constant of two characters. Note the reverse order in which the ASCII values are stored for AB: 42H is the ASCII code for B; 41H is the ASCII code for A.

DD Directive

The DD (define double-word) directive initializes four consecutive bytes (a double-word) of data memory, starting at the current value of the location counter. It has the form:

[symbol:] DD d1[, d2, . . . , dn]

where 'd' is an expression.

If the optional label is present, it is assigned the value of the assembler's location counter when the DD directive is assembled. The label thus references the low-order byte of the first of two words stored by the DD directive.

The DD directive defines four bytes of data which can be used to load a pointer/register (GA, GB, GC or TP) with a 20-bit system (memory) address via the LPD instruction. The first word of data stored is a 16-bit offset value. The second word is a 16-bit segment address.

An external symbol may appear in a DD directive expression, alone or with other (non-external) symbols and numeric constants. The external symbol must be added, NOT subtracted, in the expression. The expression is evaluated modulo 64k, with the external symbol valued at zero. The 16-bit result is stored in the first word of data memory. The value 00H is stored in the second word.

LINK86 must process the assembler's object module to resolve the external reference. When LOC86 assigns absolute addresses to the LINK86 output module, the external symbol's offset value is added to the the contents of the first word defined by the DD directive; its segment address is placed in the second word.

Example:

Label (optional)		Operands	Assembled Code (Hex)
	EXTRN	EXTERNAL	(identify EXTERNAL as a symbol defined in some other program)
EXTERNAL@SYMBOL:	DD	EXTERNAL + 10	0A000000

After the assembler's object module has been processed by LINK86, LOC86 adds the offset value of EXTERNAL to the word containing 10(0A00H), and places EXTERNAL's segment address in the next word. EXTERNAL's 20-bit address, formed from the 16-bit offset value and the 16-bit segment address, can now be loaded into a pointer/register via the LPD instruction.

DS Directive

The DS directive reserves bytes of data memory. Its format is:

```
[SYMBOL:]                    DS           expression
```

The assembler's location counter is incremented by the value of the expression, thereby reserving space in memory. There is no initialization of the data memory bytes reserved by the DS directive. Their contents are unknown when program execution begins.

Any symbol appearing in the expression must be defined, in the lexical sense, to the assembler when the DS directive is processed. A forward reference, i.e., a reference to an as yet undefined symbol, is flagged as an error.

Expressions are evaluated modulo 64k. Negative values are expressed in two's complement form. Values range from 0H to 0FFFFH, or 0 to 65,535 decimal. An expression value of zero reserves no memory space but does assign the value of the location counter to the optional label if it present.

Note that

```
RESERVE:  DS    128
```

is equivalent to (see definition of ORG below)

```
RESERVE   EQU  $
ORG       $+    128
```

The optional label, if present, is assigned the value of the assembler's location counter when the DS directive is assembled. It thus references the first data memory byte reserved.

Example:

```
DATA_BUFFER:       DS        122                ;Reserves 122 bytes of
                                                ;memory.
```

The label DATA_BUFFER references the first reserved byte; DATA_BUFFER + 1 references the second. The contents of the reserved memory bytes are unknown at the start of program execution.

Structure Definition

The STRUC/ENDS Directives

The STRUC and ENDS directives define a template of offset values, used in conjunction with the address mode "[PREG].d" (base plus unsigned 8-bit offset). This template provides a convenient means for addressing blocks of data memory. A structure does not reserve data memory or generate object code.

A structure is defined as follows:

```
name     STRUC

         ...

         ...

         ...

name     ENDS
```

A name is required and must be the same in both the STRUC and concluding ENDS directive. This name is defined as a symbol whose value is zero. It must not have been previously defined and may not be subsequently redefined.

Any instruction or assembler directive, with the exception of PUBLIC, EXTRN, EVEN, NAME, STRUC, ENDS and END, can appear between the STRUC and ENDS directives.

A STRUC directive stores the value of the assembler's location counter and sets it to zero. The following directives and instructions cause the location counter to be incremented in the normal fashion, but no object code is generated.

The ENDS directive restores the saved value of the location counter and normal assembler operation resumes. Once closed, a structure cannot be redefined or extended.

Example of the use of a structure:

The following structure creates a template of offset values to access a block of I/O control parameters written into data memory by a host processor.

STRUCTURE DEFINITION STATEMENTS			OFFSET VALUE
I?O__INFO__BLOCK	STRUC		
CONTROL__PARAMETERS:	DB	0	0000
NEW__STATUS:	DB	0	0001
INPUT__ADDRESS:	DD	0	0002
OUTPUT__ADDRESS:	DD	0	0006
RESULT__CODE:	DB	0	000A
RETRY__COUNT:	DS	0	000B
I?O__INFO__BLOCK	ENDS		

The control information can now be accessed using the pointer/registers GA, GB, or GC, loaded with the control paramter block's base address, and the template offset values:

```
MOV    GA, [GB].INPUT__ADDRESS     ;The 16-bits of data beginning at GB + 2
                                   ;are moved to GA (GA's tag bit is set
                                   ;to logical one).

MOVB   IX, [GC].RETRY__COUNT       ;The byte at GC + 11 is moved to the
                                   ;index register.
```

If the block of control parameters is written into the channel's Command Parameter Block, the PP register can be used as the base address to access the block:

```
MOVBI [PP].RESULT__CODE, 0FFH      ;Here information is being written into the
                                   ;control block at the address PP + 10.
```

Location Counter Control Directive

The assembler's location counter begins with a value of zero and is automatically incremented by the length of each instruction or the number of data memory bytes used by each data definition or memory reservation assembler directive.

ORG Directive

The location counter can be set to a specific integer value by the ORG directive:

```
        ORG        expression
```

The assembler's location counter is set to the value (in hexadecimal) of the expression. The expression is evaluated modulo 64k and negative values are expressed in two's complement form. Expressions are defined in Chapter 2 under "Immediate Data Operands."

Symbols in the expression must be defined, in the lexical sense, to the assembler when the ORG directive is processed. Forward references cause the directive to be flagged as an error.

Example:

```
ORG        1000H        ;The location counter is set to 1000.

ORG        16           ;The location counter is set to 0010.
```

EVEN Directive

System performance can be improved by placing some data and some instructions on even address boundaries. The EVEN assembler directive insures that the assembly language instruction or data memory initialization/reservation directive immediately following it begins at an even value of the assembler's location counter.

If the value of the assembler's location counter is odd when the assembler finds an EVEN directive, a three-byte no-op is generated by the assembler. If the location counter's value is even when an EVEN directive is found, the assembler takes no action and continues on to the next source statement.

The EVEN directive has the following form:

```
            EVEN
```

Example:

```
            EVEN
IN__BUFF:   DS         128
```

The value of IN__BUFF, the address of the first reserved data memory byte, is even.

Program Linkage Directives

The assembler produces a single segment, a maximum size of 64k bytes, origined at zero. This segment can be relocated using the relocation tool LOC86. The segment is aligned on a paragraph boundary; i.e., it begins at an address whose value in hexadecimal has a last digit of zero. The SEGMENT/ENDS directives define this segment and assign it a name. This name is used by LOC86 to relocate the segment.

8089 programs can share symbol table entries with other programs through the use of the PUBLIC and EXTRN directives. LINK86 and LOC86 are used to resolve such external references.

The NAME directive allows a unique name to be assigned to each object module generated by the assembler.

Refer to the publication *MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users,* order number 9800639, for details of LOC86 and LINK86.

NAME Directive

The NAME directive assigns a name to the object module generated by an assembly. It has the form:

```
NAME       module-name
```

The module-name must conform to the rules for forming a symbol; i.e., it can have 1 to 31 alphabetic, numeric or special characters (? __ @), the first of which must be alphabetic or special.

A program can contain at most one NAME directive. If there is no NAME directive, the default name assigned by the assembler is the source file name without any extension.

The module-name appears in the header lines of the listing banner of the list file.

Example:

```
NAME       DEVELOPMENT__PROGRAM__V001
```

SEGMENT/ENDS Directives

The object code generated by ASM89 is contained in a single segment, a maximum of 64k consecutive bytes in size, defined as follows:

```
name       SEGMENT
             .
             .
             .
name       ENDS
```

A name is required and must be the same in both the SEGMENT and ENDS directives.

Every source program must define exactly one segment with the SEGMENT/ENDS directives. If a segment is not defined, no object file is generated by the assembler.

All assembly language instructions and assembler directives which affect the assembler's location counter or define labels, as well as the EQU directive, must follow the SEGMENT directive and precede the ENDS directive.

Example:

```
IOP_CODE   SEGMENT
                .
                .
                .
IOP_CODE   ENDS
```

PUBLIC Directive

The PUBLIC directive makes symbols defined in this program available for access by other separately assembled or compiled programs. It has the form:

```
PUBLIC     symbol1[symbol2, . . . , symboln]
```

Symbols in a list must be separated by a comma. A symbol can be declared PUBLIC only once in a program. Reserved and external symbols cannot be declared PUBLIC.

Symbols declared PUBLIC but not defined in a source program are flagged as errors by the assembler. PUBLIC directives may appear before the SEGMENT directive and anywhere else within the program, except within a structure.

Example:

```
PUBLIC     DATA_LIST, PARM@BLOCK, I/O?DEVICE
```

EXTRN Directive

The EXTRN directive provides the assembler with a list of symbols referenced in this program but defined in other separately assembled or compiled programs. It has the form:

```
EXTRN      symbol1[, symbol2, . . . , symboln]
```

Symbols in a list must be separated by a comma.

A symbol can be declared EXTRN only once in a program. It cannot be defined within the program nor can it be declared PUBLIC.

The EXTRN directive can appear before the program's SEGMENT directive and anywhere else in the program, except in a structure.

Example:

```
EXTRN      DEVICE1, DEVICE2, DATA_TABLE
```

Assembler Termination

END Directive

The END directive identifies the end of the source program and terminates each pass of the assembler. It has the form:

```
END
```

Only one END directive may appear in a source program and it must be the last source statement. The END directive must not appear in an INCLUDE file. Any source statements following the END directive are ignored by the assembler and cause an error message to be issued to the assembler.

CHAPTER 5 ASSEMBLER CONTROLS AND OPERATION

Introduction

This chapter describes the following aspects of ASM89, the ISIS-II 8089 assembler:

- Source file format
- Invocation command, controls, and defaults
- Output files—program list file and object file

A complete list of Error Messages and corresponding user actions (where applicable) appears in Appendix J.

Source File Format

The source file input to ASM89 must reside on a random access device. INTELLEC development systems include a text editor that can be used to create and maintain 8089 Assembly Language source files as diskette files. The ASCII horizontal tab character (09H) is replaced by sufficient blank characters (always at least one) to position to the next tab stop. Tab stops are preset at columns 9, 17, 25, ...

Source files contain three elements:

- 8089 Task Block Programs, composed of 8089 assembly language instructions, described in Chapter 3 of this manual.
- Assembler directives, described in Chapter 4 of this manual.
- Assembler controls lines, described later in this chapter.

Table 5-1 summarizes important source file parameters.

Table 5-1. 8089 Assembly Language Source File Parameters

ITEM	LIMIT
Characters/compressed* source statement	256 characters.
Characters/symbolic name	31; symbolic names greater than 32 characters are flagged as errors.
Symbols/module	300 (approximately); relative to the length of the symbolic names used.
INCLUDE'd files	No assembler imposed limit on the number of INCLUDEd files, but nested INCLUDEs (INCLUDE controls in an INCLUDEd file) are not allowed. INCLUDEd files must not contain an END directive.
Segment definition	A single segment, a maximum of 64k bytes in size, must be defined via the SEGMENT/ ENDS directives.
END directive	A single END directive must appear in a source file.

* The assembler compresses each source statement by deleting all comments, and the final end-of-line, changing all unquoted sequences of blanks and tabs into single blanks, changing unquoted end-of-line&'s into single blanks, and changing all quoted quotes into single characters.

Invocation Command, Controls, and Defaults

You can invoke ASM89 from ISIS-II by entering the command:

:F*n*:ASM89 *source controls*

where:

:F*n*:

designates the drive on which ASM89 resides. If *n*=0, you can omit the drive designation.

source

designates the drive and file (for example, :F1:PROG.SRC) containing the source statements to be assembled.

controls

is a (possibly empty) list of controls, separated by blanks. This field of the invocation command is called the command tail.

You can continue the invocation command to one or more additional lines by entering an unquoted ampersand (&) in place of a blank. Since anything following the ampersand on that line is echoed, but otherwise ignored, you can thus comment your invocation lines; they are echoed in the listing. On subsequent lines, ASM89 prompts you for the remainder of the invocation command by issuing a double asterisk followed by a blank (**). Refer to Example 5-3, "Continuation Lines and Prompting," in this chapter.

Summary of Controls

Table 5-2 provides a summary of ASM89 controls and defaults. There are two classes of controls: Primary (P) and General (G). Both classes of controls can be specified in the command tail and in separate control lines within the source file, except the general controls EJECT and INCLUDE, which can only appear in source file control lines. A control line is an assembler source line having a dollar sign ($) as its first character.

Primary and general controls differ as follows:

- Primary controls establish global modes of operation, and if specified must appear in the command tail or prior to any non-control lines in the source file. If conflicting primary controls are specified (e.g. PRINT and NOPRINT), the last valid control is used.
- General controls may appear in the command tail or in any line of the source file. General controls may be respecified at any time.

Table 5-2. ASM89 Controls and Defaults

CONTROL	P/G	DEFAULT	PURPOSE
OBJECT(file)	P	OBJECT(file.OBJ)	Name and/or place the object file
NOOBJECT	P	OBJECT(file.OBJ)	Don't create object file
PRINT(file)	P	PRINT(file.LST)	Name the listing file

Table 5-2. ASM89 Controls and Defaults (Cont'd.)

CONTROL	P/G	DEFAULT	PURPOSE
NOPRINT	P	PRINT(file.LST)	Don't create listing file
SYMBOLS	P	SYMBOLS	List symbol table
NOSYMBOLS	P	SYMBOLS	Don't list symbol table
PAGEWIDTH(n)	P	PAGEWIDTH(120)	Chars/line in listing
PAGELENGTH(n)	P	PAGELENGTH(62)	Lines/page in listing
PAGING	P	PAGING	Separate pages in listing
NOPAGING	P	PAGING	Continuous listing
DATE('dddddddd')	P	DATE(' ')	Appears in header
TITLE('t...t')	P	TITLE(' ')	Appears in header
LIST	G	LIST	Turn on listing
NOLIST	G	LIST	Turn off listing
EJECT	G		Start new listing page
INCLUDE(file)	G		Assemble a side file here

Primary Control Descriptions

OBJECT(filename)

Specifies that an object file is to be created and gives the location and name of the file. If the file specification is missing, the object file is placed in a file with the same device and name as the source file, and with the extension OBJ.

NOOBJECT

Specifies that no object file is to be produced.

PRINT(filename)

Specifies that a listing file is to be created and names the file. If the file specification is missing, the listing file is placed in a file with the same device and file name as the source file, and with the extension LST.

NOPRINT

Specifies that no listing file is to be created.

SYMBOLS

Specifies that a formatted listing of the symbol table is to be created and appended to the listing file.

NOSYMBOLS

Specifies that a formatted listing of the symbol table is not to be created.

PAGEWIDTH(n)

Specifies the width of the listing page in number of characters per line. The range for n is from 72 - 132 inclusively.

PAGELENGTH(n)

Specifies the length of the listing page in number of lines per page. The range for n is 10 - 255 inclusively.

PAGING

Specifies that the listing is to be formatted as separate pages.

NOPAGING

Specifies that the listing is not to be formatted as separate pages; that is, the listing is continuous.

DATE('ddddddddd')

Supplies a field of up to 9 characters in the header of each listing page for the user-specified date (or other information).

TITLE('t...t')

Supplies a variable length field of characters to appear in the header of each page in the listing. The length of the title field depends on the PAGEWIDTH and the presence or absence of a DATE control. Titles exceeding the field width are truncated.

General Control Descriptions

LIST

Turns on the source statement listing mechanism.

NOLIST

Turns off the source statement listing mechanism. Statements in error and error messages are still listed if PRINT is specified.

EJECT

Causes an eject (by issuing a form-feed to the listing file) to a new page.

INCLUDE(filename)

Specifies that the named file is to be included for assembly. When ASM89 encounters the INCLUDE control, the source input is switched to the specified file and remains there until an end-of-file condition is encountered. The included file(s) must not contain either another INCLUDE control (that is, no nesting of included files is permitted) or an END directive. The end-of-file condition is the only terminator recognized for the included file, regardless of the presence of carriage-returns, line-feeds, or continued lines.

Examples

Example 5-1. Full Default

Suppose the following:

1. ASM89 resides on disk drive 0
2. Your source file, CHAN.TST, resides on disk drive 1

Then the invocation command:

```
    ASM89 :F1:CHAN.TST
```

calls the assembler into operation and results in the following:

- The object file is placed in :F1:CHAN.OBJ
- The listing file is placed in :F1:CHAN.LST
- A formatted listing of the symbol table is placed in the listing file.
- No line in the listing file exceeds 120 characters.
- The listing file is paged; no page in it exceeds 62 lines.
- The Title and Date fields in the listing file header are blank.

Example 5-2. Partial Default

If, in Example 5-1, the invocation command is replaced by:

```
    ASM89 :F1:CHAN.TST OBJECT(:F1:NETCAT.DRV) PRINT(:TO:) DATE('6/21/79')
```

then the results differ as follows:

- The object file is placed in :F1:NETCAT.DRV
- The listing file is printed on the teletypewriter, provided one is attached, powered ON, and set to "LINE" mode.
- The string 6/21/79 (without quotes) appears in the DATE field in the header on each page of the listing.

Example 5-3. Continuation Lines and Prompting

You can continue the invocation line using an unquoted ampersand. Since ASM89 ignores characters appearing between the ampersand and the end of the line, you can use this field to document your invocation line. ASM89 prompts you for more information by issuing a double-asterisk followed by a blank, as follows:

```
ASM89 :F1:CHN3N4.TST          & ISIS-II 8089 Assembly of source file CHN3N4.TST
** OBJECT(:F3:LINK34.001)     & Object file
** PRINT(:F4:LINK34.DOC)      & Listing file
** NOSYMBOLS                  & No symbol table printout this time
** PAGEWIDTH(132)             & Max. line length is 132 chars.
** NOPAGING                   & No form feeds; continuous print-out
** DATE('8/15/79')            & 1st day network integration
** TITLE('Fire Up N3-N4')     & Physical link checkout between nodes 3 and 4
**
```

Processing begins following your carriage-return after the last prompt. The invocation command and its comments are echoed in the listing file, in this case :F4:LINK34.DOC.

Format of the Listing File

Each page of the assembler-generated list file begins with a header:

8089 ASSEMBLER [title] [date] PAGE *X*

Items enclosed in brackets, [ ], are optional. The TITLE control places a user- defined title in the header; the DATE control adds a user-specified date. The page number, beginning with one, is included in the header.

On the first page of the listing file, the header is followed by the listing banner:

ISIS-II 8089 ASSEMBLER *version* ASSEMBLY OF MODULE *module-name*
OBJECT MODULE PLACED IN *object file name*
ASSEMBLER INVOKED BY *invocation command*

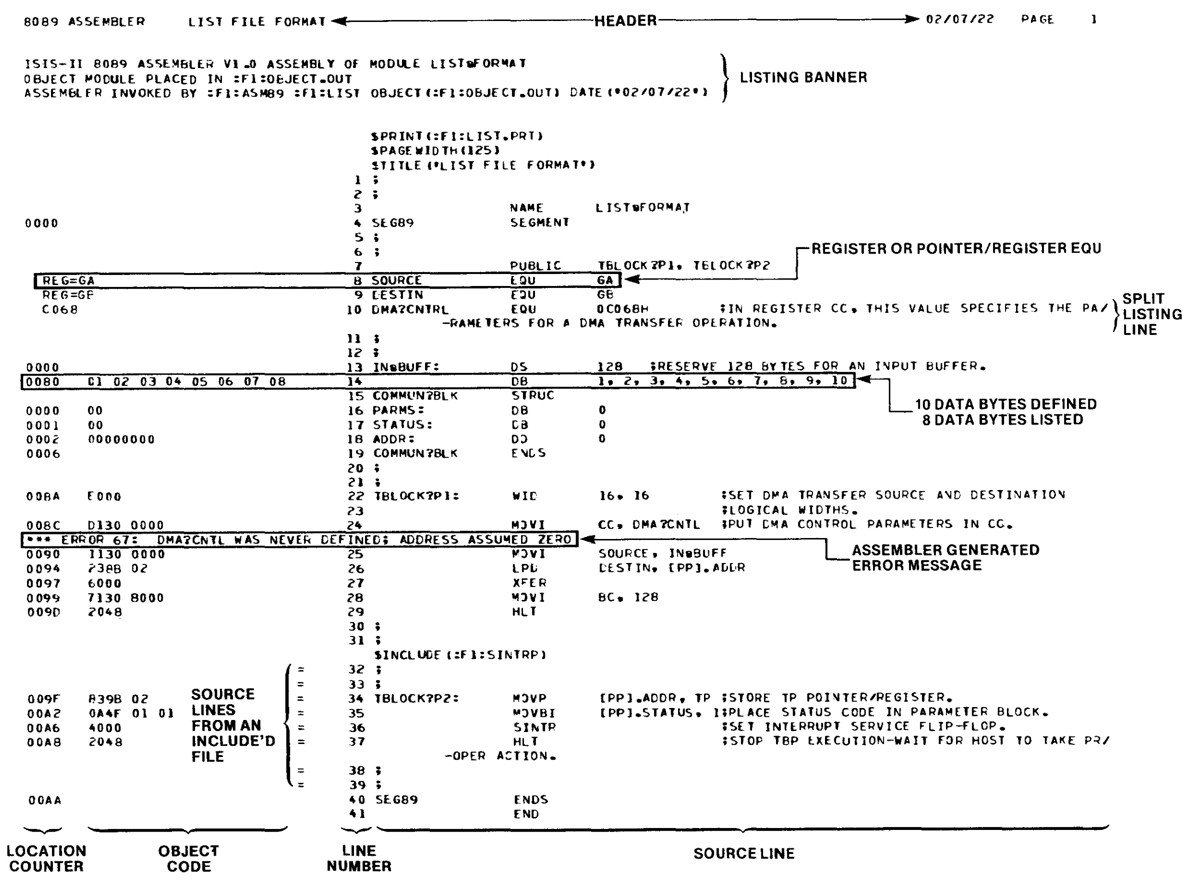
The body of the list file contains the following four fields of information:

Location Counter Object Code Line Number Source Line

All source lines appear, in order, in the body of the list file.

EQU directive values are indented two positions from the first location counter digit. When registers or pointer/registers are assigned alternate names through an EQU directive, the following appears as the EQU value in the list file (see figure 5-1):

REG = *register or pointer/register*



```
8089 ASSEMBLER    LIST FILE FORMAT                                        02/07/22   PAGE    1

ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE LIST_FORMAT
OBJECT MODULE PLACED IN :F1:OBJECT.OUT
ASSEMBLER INVOKED BY :F1:ASM89 :F1:LIST OBJECT(:F1:OBJECT.OUT) DATE('02/07/22')

                                  $PRINT(:F1:LIST.PRT)
                                  $PAGEWIDTH(125)
                                  $TITLE('LIST FILE FORMAT')
                                1 ;
                                2 ;
                                3              NAME      LIST_FORMAT
0000                            4 SEG89        SEGMENT
                                5 ;
                                6 ;
                                7              PUBLIC    TBLOCK?P1, TBLOCK?P2
  REG=GA                        8 SOURCE       EQU       GA
  REG=GB                        9 DESTIN       EQU       GB
  C068                         10 DMA?CNTRL    EQU       0C068H      ;IN REGISTER CC, THIS VALUE SPECIFIES THE PA/
                                     -RAMETERS FOR A DMA TRANSFER OPERATION.
                               11 ;
                               12 ;
0000                           13 IN_BUFF:     DS        128   ;RESERVE 128 BYTES FOR AN INPUT BUFFER.
0080    01 02 03 04 05 06 07 08 14             DB        1, 2, 3, 4, 5, 6, 7, 8, 9, 10
                               15 COMMUN?BLK   STRUC
0000    00                     16 PARMS:       DB        0
0001    00                     17 STATUS:      DB        0
0002    00000000               18 ADDR:        DD        0
0006                           19 COMMUN?BLK   ENDS
                               20 ;
                               21 ;
008A    E000                   22 TBLOCK?P1:   WID       16, 16      ;SET DMA TRANSFER SOURCE AND DESTINATION
                               23                                    ;LOGICAL WIDTHS.
008C    D130 0000              24              MOVI      CC, DMA?CNTL ;PUT DMA CONTROL PARAMETERS IN CC.
*** ERROR 67:  DMA?CNTL WAS NEVER DEFINED; ADDRESS ASSUMED ZERO
0090    1130 0000              25              MOVI      SOURCE, IN_BUFF
0094    238B 02                26              LPD       DESTIN, [PP].ADDR
0097    6000                   27              XFER
0099    7130 8000              28              MOVI      BC, 128
009D    2048                   29              HLT
                               30 ;
                               31 ;
                                  $INCLUDE(:F1:SINTRP)
                      =        32 ;
                      =        33 ;
009F    839B 02       =        34 TBLOCK?P2:   MOVP      [PP].ADDR, TP ;STORE TP POINTER/REGISTER.
00A2    0A4F 01 01    =        35              MOVBI     [PP].STATUS, 1;PLACE STATUS CODE IN PARAMETER BLOCK.
00A6    4000          =        36              SINTR                 ;SET INTERRUPT SERVICE FLIP-FLOP.
00A8    2048          =        37              HLT                   ;STOP TBP EXECUTION-WAIT FOR HOST TO TAKE PR/
                                     -OPER ACTION.
                      =        38 ;
                      =        39 ;
00AA                           40 SEG89        ENDS
                               41              END
```

Figure 5-1. List File Format

```
8089 ASSEMBLER     LIST FILE FORMAT                                              02/07/22    PAGE    2

SYMBOL TABLE
------------

 DEFN VALUE TYPE  NAME
 ----- ----- ----  ----

    18  0002  SYM  ADDR
    15  0000  STR  COMMUN?BLK
     9    GB  REG  DESTIN
 -----  0000  SYM  DMA?CNTL  <---------------- SOURCE FILE SYMBOL NOT DEFINED IN THE FILE
    10  C068  SYM  DMA?CNTRL
    13  0000  SYM  IN@BUFF
    16  0000  SYM  PARMS
     4  0000  SYM  SEG89
     8    GA  REG  SOURCE
    17  0001  SYM  STATUS
    22  008A  PUB  TBLOCK?P1
    34  009F  PUB  TBLOCK?P2

ASSEMBLY COMPLETE; 1 ERROR FOUND <----------- LAST LIST FILE LINE CONTAINING ERROR COUNT
```

Figure 5-1. List File Format (Cont'd.)

Figure 5-1 shows the listing file of a sample program coded in 8089 assembly language.

The object field contains the assembler-generated object code for each source file instruction. The data generated by data-generating source file directives also appears in the object code field. Note that while data-generating directives can generate any number of data bytes, only the first eight bytes generated appear in the listing. (See figure 5-1.)

Source lines that do not fit on a single list file line are split. A '/' at the end of a list file line indicates a split source line. A '—' at the beginning of a list file line indicates that the line is a continuation of the previous list file line. (See figure 5-1.) Source lines from an INCLUDEd file are masked by an '=' character, which appears before the line number and list file line.

Error messages generated by the assembler are placed in the list file immediately following the source statement which provokes the error. (See figure 5-1.) A complete list of error messages is given in Appendix J.

The list file may also include a symbol table. The symbol table appears at the end of the list file, under the heading:

```
SYMBOL TABLE
```

Symbol information appears under the following headings:

```
DEFN VALUE TYPE NAME
```

DEFN	Contains the list file line number where file symbol is defined. '-----' under DEFN indicates that the symbol was found in the source file input but never defined.
VALUE	Indicates the value assigned to the symbol by the assembler. Symbols defined as an alternate name for a register or pointer/register have the Register Symbol listed as their value. External symbols are numbered, starting with one, in the symbol table. This number appears in the value field.

TYPE	Indicates the kind of symbols defined:
	SYM — A user-defined symbol (label or name).
	REG — An alternate name for a register or a pointer register.
	PUB — A symbol declared PUBLIC in the source file.
	EXT — A symbol declared EXTRN in the source file.
	STR — The name of a structure defined in the source file.
NAME	The user-defined symbol.

The list file concludes with the following line, listing the number of errors found by the assembler:

ASSEMBLY COMPLETE; *number of errors found*

GLOSSARY

This glossary contains terms specifically related to the operation of the Intel 8089 I/O Processor.

ASM89—the assembler for the 8089 Assembly Language.

BC—a predefined symbol for the general purpose 16-bit register that is used as a byte counter during DMA transfers.

Bus Load Limit—an 8089 control, specified in the Channel Control Word, that limits task block program instruction execution for a channel.

BUSY flag byte—a byte in the Channel Control Block (CB+1 for channel one; CB+9 for channel two) indicating the activity status of a channel.

CC—a predefined symbol for the 16-bit register used to specify controls for a channel's I/O operations.

Chained task block program instruction execution—the priority of task block program instruction execution is equal to that of DMA transfer; task block program instruction execution on one channel may interleave with DMA transfer operations on the other channel, depending on the P value in the CCW of both channels.

Channel attention—a hardware input to the 8089 used to begin 8089 initialization and initiate communication between a host processor and the 8089's two I/O channels.

Channel Control Block (CB)—a block of shared system memory used for communication between a host processor and the 8089's two I/O channels.

Channel Control Word (CCW)—a byte in the Channel Control Block (CB for channel one; CB+8 for channel two) used to issue commands and specify operation parameters for an 8089 channel.

Command Field (CF)—a three-bit field in the CCW used to issue commands to an 8089 channel.

Command Parameter Block (PB)—a block of shared system memory used for communication between a host processor and an 8089 channel. The address of a channel's task block program is contained in PB.

DMA transfer—a high-speed direct memory access data transfer operation.

GA, GB—predefined symbols for the 20-bit general purpose pointer/registers and their associated tag bits, used in task block programs to access data memory and, in DMA transfers, to specify source/destination addresses.

GC—a predefined symbol for the 20-bit general purpose pointer/register and its associated tag bit, used in task block programs to access data memory and, in DMA tranfers in the translate mode, to specify the base address of a 256 byte translation table.

Indirect addressing—a data memory location is accessed via a pointer/register containing the address of the desired data memory location.

Interrupt Control Field (ICF)—a two-bit field in the CCW used to control interrupts from an 8089 channel.

IX—a predefined symbol for the 16-bit general purpose register used in some data memory expression forms to provide an index value which is added to a base pointer/register; in the data memory expression from [PREG+IX+], IX is post auto-incremented by 1 (byte datum) or 2 (word datum).

Jump target—a location containing the instruction to which program control is passed as a result of a control transfer instruction.

LINK86—an MCS-86 software development utility which resolves inter-module references.

LOC86—an MCS-86 software development utility which assigns absolute addresses to object modules.

LOCAL configuration—an 8089 and a host processor share a single bus.

Local (I/O) space—the 64k byte address space which accesses an 8089's remote bus in a REMOTE configuration or I/O addresses in a LOCAL configuration.

Logical width—the width, in bits, of the DMA transfer source or destination. Logical widths, specified by a task block program WID instruction, may differ from a system's physical bus widths. For example, a DMA transfer source or destination on a 16-bit bus can have a logical width of eight bits. Certain logical widths are required by the 8089 during DMA transfers for data translation and testing operations.

Long jump or call—an "L" prefix is attached to the short form of a control transfer instruction. A signed word displacement (−32,768, +32,767), used to form the jump target's address, is generated by the assembler.

Mask/Compare—an exclusive OR is performed on a data byte and a compare byte. The result is logically ANDed with a mask byte. The result of the logical AND is checked for zero (mask/compare is equal).

MASTER—when the RQ/GT circuitry is used to control access to a bus shared by two processors, one processor is designated a MASTER and controls the bus following system initialization.

MC—a predefined symbol for the 16-bit general purpose register that provides mask/compare bytes for certain 8089 Assembly Language instructions and DMA transfer operations.

Offset, offset value—a 16-bit value added to a 16-bit segment address (shifted left four bit positions) to form a 20-bit address. (See *MCS-86 Assembly Language Reference Manual,* Order Number 9800640, for more information.)

Paragraph aligned—the segment in an ASM89 object-module is located by LOC86 on a paragraph boundary, i.e., it begins at an address divisible by sixteen. (See *MCS-86 Assembly Language Reference Manual,* Order Number 9800640, for more information.)

Pointer/Register—a 20-bit register with an associated tag bit used to point to 16-bit local (I/O) space or 20-bit system (memory) space.

PP—a predefined symbol for the read-only, non-programmable 20-bit register which contains the address of a channel's Command Parameter Block (PB).

Program Status Word (PSW)—an 8-bit value stored in the fourth byte of a channel's PB (PB+3) when a channel's operation is suspended by a HALT AND SAVE command in the CCW. The PSW contains channel status information.

Remote bus—the bus in a REMOTE configuration not accessible by a host processor, accessed by the 8089 with 16-bit local (I/O) addresses.

REMOTE configuration—the 8089 has its own remote bus, inaccessible to a host processor and accessed by 16-bit local (I/O) space addresses. The 8089 also accesses a shared system bus via 20-bit system (memory) space addresses.

RQ/GT—a hardware pin and its associated circuitry used to control access to a bus shared by two processors.

Segment, Segment address—a 16-bit value shifted left four bit positions and added to a 16-bit offset value to form a 20-bit address. (See the *MCS-86 Assembly Language Reference Manual,* Order Number 9800640, for more inforation.)

Short jump or call—a control transfer instruction without an "L" prefix. A signed byte (−128, +127) or a signed word (−32,768, +32,767) displacement value can be generated by a short control transfer instruction. If a forward reference is used in the expression specifying the jump target, the assembler assumes a signed byte displacement value is needed.

SLAVE—when the RQ/GT circuitry is used to control access to a bus shared by two processors, one processor is designated a SLAVE. A SLAVE requests the bus from the MASTER following system initialization. The "R" value in the System Operation Command specifies the way in which the bus is shared between a MASTER and a SLAVE processor.

SYSBUS—the first byte in the System Configuration Pointer, SYSBUS specifies the width of the system bus.

System bus—the bus in a REMOTE configuration accessed by the 8089 using 20-bit addresses. In LOCAL configurations this is the bus shared by the 8089 and a host processor.

System Configuration Block (SCB)—the second block in a linked list of shared data memory blocks used to initialize the 8089. The SCB is pointed to by the System Configuration Pointer and contains the SOC and the Channel Control Block address.

System Configuration Pointer (SCP)—the first block in a linked list of shared data memory blocks used to initialize the 8089. The SCP must begin at address 0FFFF6H. It contains the SYSBUS byte and points to the System Configuration Block.

System (memory) space—the one-megabyte address space which accesses the system bus in a REMOTE configuration and data memory in a LOCAL configuration.

System Operation Command (SOC)—the first byte in the System Configuration Block, the SOC specifies the width of the remote bus, if one is present. It also specifies the mode of RQ/GT circuitry operation.

Tag bit—a bit associated with a 20-bit pointer/register. A tag bit's value indicates whether the pointer/register contains a 16-bit local (I/O) address (tag bit=1) or a 20-bit system (memory) address (tag bit=0).

Task block program (TBP)—a program written in 8089 Assembly Language which manages and controls a channel's I/O operations.

TP—a predefined symbol for the 20-bit pointer/register and its associated tag bit, used as an instruction pointer for a channel's task block programs.

APPENDIX A
OPERAND SUMMARY

8089 Assembly Language instruction operands specify the various kinds of items used in each operation. Table A-1 summarizes these items and their associated operand types:

Table A-1. Data Items and Associated Operand Types

ITEM	OPERAND TYPE	EXAMPLES
Machine registers	Register	IX, MC, BC
Machine Pointer/Registers	Pointer/Register	GA, GB, GC
Immediate Data Values	Immediate Data	0FFH, ADTAB + 4
Locations Within a Program	Program Location	$ + 6, START
Data in Memory	Data Memory	[GA], [GB].5
Bits of Memory Data	Data Memory Bit	0, 1, 7

Register Operands

SYMBOL	REGISTER NAME	SYMBOL	REGISTER NAME
BC	Byte Count	GC	General Purpose C
CC	Channel Control	IX	Index Register
GA	General Purpose A	MC	Mask/Compare
GB	General Purpose B	TP	Task Pointer

Pointer/Register Operands

SYMBOL	REGISTER NAME	SYMBOL	REGISTER NAME
GA	General Purpose A	GC	General Purpose C
GB	General Purpose B	TP	Task Pointer

Immediate Data Operands

Immediate data operands are expressions composed of:

- Symbols
- Numeric constants
- Character string constants of one or two characters
- The special location counter reference symbol $
- The assembly time operators + and −

Immediate data operands can represent a data memory location, an instruction location, or an 8- or 16-bit value.

Program Location Operands

Locations within a program can be specified by three general types of expressions:

- An expression containing an instruction label (e.g. ROUTINE1)
- An expression containing only numeric constants (a displacement from the beginning of the program segment—NOT an absolute address)
- An expression containing a relative instruction address (i.e., an expression containing the special location counter reference symbol $)

Data Memory Operands

Data memory is accessed indirectly, using the contents of the pointer/registers GA, GB, or GC or the PP register as a base address. Data memory operands have four forms:

[PREG] — Base address only

'PREG' can be the pointer/register GA, GB, GC, or the PP register. 'PREG' contains the data memory address.

[PREG].d — Base address plus an unsigned 8-bit offset

'd' is an expression evaluated modulo 256.

[PREG+IX] — Base address plus the Index register.

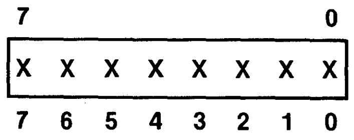
The data memory address is formed by adding the contents of the Index register and the base address. The contents of the Index register and the base address are not changed.

[PREG+IX+] — Base address plus the Index register; Index register post auto-incremented

The data memory address is formed by adding the contents of the Index register and the base address. At the end of the instruction, the Index register is automatically incremented by the size of the memory data (by one for byte data, by two for word data). The base address is unchanged.

Data Memory Bit Operands

The bits in a data memory byte are numbered, right to left, as follows:



The bit number is the operand used in an 8089 Assembly Language instruction, where applicable, to specify the referenced bit.

APPENDIX B
INSTRUCTION SET SUMMARY

Decoding information:

R—a register symbol	P—a pointer/register symbol
M—a data memory expression	b—a data memory bit symbol

I—an expression specifying an immediate value

L—an expression specifying a program location (e.g., a label)

See Appendix A, "Operand Summary," for a description of each of the above items.

R8 —Specifies the low-order byte of a 16-bit register. When 'R8' is the destination (left-most) operand of a data transfer instruction, the data is sign-extended (bit 7) to 16 bits. If 'R' is a 20-bit pointer/register, the data is sign extended to 20 bits and the pointer/register's tag bit is set to logical one. All data is sign-extended to 16 bits when arithmetic and logical operations are performed. The high-order byte of 'R' is, therefore, affected by 8-bit operations. If 'R' is a 20-bit pointer/register, the upper four bits (bits 16-19) are undefined following all arithmetic and logical operations, except addition. Addition to a pointer/register can result in a carry into its upper four bits.

R16 —The entire 16-bit register is used in the operation. When a 20-bit pointer/register is the destination (left-most) operand of a data transfer instruction, the data is sign-extended (bit 15) to 20 bits. The pointer/register's tag bit is set to logical one. If 'R' is a 20-bit pointer/register, the upper four bits (16-19) are undefined following all arithmetic and logical operations, except addition. Addition to a pointer/register can result in a carry into the upper four bits.

M8	—a byte (8 bits) of data memory
M16	—a word (16 bits) of data memory
M24	—three bytes of data memory
M32	—four bytes of data memory
I8	—an 8-bit immediate value
I16	—a 16-bit immediate value

NOTE

A label is optional on *all* assembly language instructions.

Data Transfer Instructions

INSTRUCTION FORMAT		OPERATION
LPD	P, M32	Load 20-bit pointer/register from data memory
LPDI	P, I16	Load 20-bit pointer/register from immediate data
MOVP	M24, P P, M24	Move 20-bit pointer/register to (store) or from (restore) memory
MOV	R16, M16 M16, R16 M16, M16	Move 16-bits of data memory to/from data memory or register

MOVB	R8, M8 M8, R8 M8, M8	Move 8-bits of data memory to/from data memory or register
MOVI	R16, I16 M16, I16	Move 16-bits of immediate data to data memory or register
MOVBI	R8, I8 M8, I8	Move 8-bits of immediate data to data memory or register

Control Transfer Instructions

Unconditional Control Transfer Instructions:

INSTRUCTION FORMAT		OPERATION
CALL LCALL	M24, L	Store TP pointer/register and tag bit; Jump
JMP LJMP	L	Jump

Conditional Control Transfer Instructions:

INSTRUCTION FORMAT		OPERATION
JMCE LJMCE	M8, L	Jump on mask/compare equal
JMCNE LJMCNE	M8, L	Jump on mask/compare not equal
JNZ LJNZ	R16, L M16, L	Jump on nonzero register or data memory word
JNZB LJNZB	M8, L	Jump on nonzero data memory byte
JZ LJZ	R16, L M16, L	Jump on zero register or data memory word
JZB LJZB	M8, L	Jump on zero data memory byte

Arithmetic and Logical Instructions

INSTRUCTION FORMAT		OPERATION
ADD	R16, M16 M16, R16	ADD register and 16-bit memory data
ADDB	R8, M8 M8, R8	ADD register and 8-bit memory data
ADDBI	R8, I8 M8, I8	ADD register or 8-bit memory data and 8-bit immediate data
ADDI	R16, I16 M16, I16	ADD register or 16-bit memory data and 16-bit immediate data
AND	R16, M16 M16, R16	AND register with 16-bit memory data
ANDB	R8, M8 M8, R8	AND register with 8-bit memory data

ANDBI	R8, I8 M8, I8	AND register or 8-bit memory data with 8-bit immediate data
ANDI	R16, I16 M16, I16	AND register or 16-bit memory data with 16-bit immediate data
DEC	R16 M16	Decrement register or 16-bit memory data
DECB	M8	Decrement 8-bit memory data
INC	R16 M16	Increment register or 16-bit memory data
INCB	M8	Increment 8-bit memory data
OR	R16, M16 M16, R16	OR register and 16-bit memory data
ORB	R8, M8 M8, R8	OR register and 8-bit memory data
ORBI	R8, I8 M8, I8	OR register or 8-bit memory data with 8-bit immediate data
ORI	R16, I16 M16, I16	OR register or 16-bit memory data with 16-bit immediate data
NOT	R16 M16 R16, M16	Complement register or 16-bit memory data; (optionally place complemented memory data in register)
NOTB	M8 R8, M8	Complement 8-bit memory data; (optionally place complemented memory data in register)

Bit Manipulation and Test Instructions

INSTRUCTION FORMAT		OPERATION
SETB	M8, b	Set selected data memory bit to logical one
CLR	M8, b	Clear selected data memory bit to logical zero
JBT LJBT	M8, b, L	Jump on data memory bit true (bit = logical one)
JNBT LJNBT	M8, b, L	Jump on data memory bit not true (bit <> logical one)

Special and Miscellaneous Instructions

INSTRUCTION FORMAT		OPERATION
HLT		Halt task block program execution; channel's BUSY flag byte in the CB cleared to 00H
NOP		No operation
SINTR		Set interrupt service flip flop
TSL	M8, I8, L	Test and set data memory byte while system bus is locked
WID	S, D	Set DMA source and destination logical widths
XFER		Begin DMA transfer following the execution of the next instruction

APPENDIX C ASSEMBLER DIRECTIVES SUMMARY

NOTE

Items enclosed in brackets, [ ], are optional.

Symbol Definition

DIRECTIVE FORMAT			OPERATION
name	EQU	expression	Defines a symbol and assigns it a value.

Data Definition and Memory Reservation

DIRECTIVE FORMAT			OPERATION
[symbol:]	DB	d1*[, d2, ... dn]	Defines byte(s) of data memory with 8-bit values.
[symbol:]	DW	d1[, d2, ... dn]	Defines word(s) of data memory with 16-bit values.
[symbol:]	DD	d1[, d2, ... dn]	Defines double word(s) of data memory for 20-bit address loading.
[symbol:]	DS	expression	Reserves bytes of data memory.

Structure Definition

DIRECTIVE FORMAT			OPERATION
name	STRUC		Creates a template of offset values.
	.		
	.		
	.		
name	ENDS		

Location Counter Control

DIRECTIVE FORMAT			OPERATION
	ORG	expression	Sets the assembler's location counter to a specified integer value.
	EVEN		Insures that the next instruction or directive begins at an even assembler location counter value.

*d*x* is an expression, evaluated modulo 256 in DB directives and modulo 64k in DW, DD, and DS directives.

*s*x* is a symbol.

Program Linkage

	DIRECTIVE FORMAT		OPERATION
	NAME	module-name	Assigns a name to the assembler-generated object module.
name	SEGMENT		Assigns a name to the segment containing the assembler-generated object code.
	.		
	.		
	.		
name	ENDS		
	PUBLIC	s1**[, s2, ... sn]	Identifies symbols defined in this source program that can be referenced by separately assembled or compiled programs.
	EXTRN	s1[, s2, ... sn]	Identifies symbols within this source program which are defined and declared PUBLIC in separately assembled or compiled programs.

Assembler Termination

DIRECTIVE FORMAT	OPERATION
END	Indicates the end of a source program.

APPENDIX D
ASSEMBLER CONTROLS SUMMARY

Table D-1. ASM89 Controls and Defaults

CONTROL	P/G	DEFAULT	PURPOSE
OBJECT(file)	P	OBJECT(file.OBJ)	Name and/or place the object file
NOOBJECT	P	OBJECT(file.OBJ)	Don't create object file
PRINT(file)	P	PRINT(file.LST)	Name the listing file
NOPRINT	P	PRINT(file.LST)	Don't create listing file
SYMBOLS	P	SYMBOLS	List symbol table
NOSYMBOLS	P	SYMBOLS	Don't list symbol table
PAGEWIDTH(n)	P	PAGEWIDTH(120)	Chars/line in listing
PAGELENGTH(n)	P	PAGELENGTH(62)	Lines/page in listing
PAGING	P	PAGING	Separate pages in listing
NOPAGING	P	PAGING	Continuous listing
DATE('dddddddd')	P	DATE(' ')	Appears in header
TITLE('t...t')	P	TITLE(' ')	Appears in header
LIST	G	LIST	Turn on listing
NOLIST	G	LIST	Turn off listing
EJECT	G		Start new listing page
INCLUDE(file)	G		Assemble a side file here

APPENDIX E
ASCII CHARACTER SET CHART

ASCII CODES

The 8089 assembler uses the seven bit ASCII code, with the high-order eighth bit (parity bit) always reset.

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
NUL	00
SOH	01
STX	02
ETX	03
EOT	04
ENQ	05
ACK	06
BEL	07
BS	08
HT	09
LF	0A
VT	0B
FF	0C
CR	0D
SO	0E
SI	0F
DLE	10
DC1 (X-ON)	11
DC2 (TAPE)	12
DC3 (X-OFF)	13
DC4 (~~TAPE~~)	14
NAK	15
SYN	16
ETB	17
CAN	18
EM	19
SUB	1A
ESC	1B
FS	1C
GS	1D
RS	1E
US	1F
SP	20
!	21
″	22
#	23
$	24
%	25
&	26
′	27
(	28
)	29
*	2A
+	2B
,	2C
–	2D
.	2E
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47
H	48
I	49
J	4A
K	4B
L	4C
M	4D
N	4E
O	4F
P	50
Q	51
R	52
S	53
T	54
U	55
V	56
W	57
X	58
Y	59
Z	5A
[	5B
\	5C
]	5D
∧ (↑)	5E
– (←)	5F
‘	60
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A
{	7B
|	7C
} (ALT MODE)	7D
~	7E
DEL (RUB OUT)	7F

APPENDIX F DECIMAL/HEXADECIMAL CONVERSION

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

POWERS OF 16 (IN BASE 10)

16^n	n	16^{-n}	
1	0	0.10000 00000 00000 00000	x 10
16	1	0.62500 00000 00000 00000	x 10^{-1}
256	2	0.39062 50000 00000 00000	x 10^{-2}
4 096	3	0.24414 06250 00000 00000	x 10^{-3}
65 536	4	0.15258 78906 25000 00000	x 10^{-4}
1 048 576	5	0.95367 43164 06250 00000	x 10^{-6}
16 777 216	6	0.59604 64477 53906 25000	x 10^{-7}
268 435 456	7	0.37252 90298 46191 40625	x 10^{-8}
4 294 967 296	8	0.23283 06436 53869 62891	x 10^{-9}
68 719 476 736	9	0.14551 91522 83668 51807	x 10^{-10}
1 099 511 627 776	10	0.90949 47017 72928 23792	x 10^{-12}
17 592 186 044 416	11	0.56843 41886 08080 14870	x 10^{-13}
281 474 976 710 656	12	0.35527 13678 80050 09294	x 10^{-14}
4 503 599 627 370 496	13	0.22204 46049 25031 30808	x 10^{-15}
72 057 594 037 927 936	14	0.13877 78780 78144 56755	x 10^{-16}
1 152 921 504 606 846 976	15	0.86736 17379 88403 54721	x 10^{-18}

POWERS OF 10 (IN BASE 16)

10^n	n	10^{-n}	
1	0	1.0000 0000 0000 0000	
A	1	0.1999 9999 9999 999A	
64	2	0.28F5 C28F 5C28 F5C3	x 16^{-1}
3E8	3	0.4189 374B C6A7 EF9E	x 16^{-2}
2710	4	0.68DB 8BAC 710C B296	x 16^{-3}
1 86A0	5	0.A7C5 AC47 1B47 8423	x 16^{-4}
F 4240	6	0.10C6 F7A0 B5ED 8D37	x 16^{-4}
98 9680	7	0.1AD7 F29A BCAF 4858	x 16^{-5}
5F5 E100	8	0.2AF3 1DC4 6118 73BF	x 16^{-6}
3B9A CA00	9	0.44B8 2FA0 9B5A 52CC	x 16^{-7}
2 540B E400	10	0.6DF3 7F67 SEF6 EADF	x 16^{-8}
17 4876 E800	11	0.AFEB FF0B CB24 AAFF	x 16^{-9}
E8 D4A5 1000	12	0.1197 9981 2DEA 1119	x 16^{-9}
918 4E72 A000	13	0.1C25 C268 4976 81C2	x 16^{-10}
5AF3 107A 4000	14	0.2D09 370D 4257 3604	x 16^{-11}
3 8D7E A4C6 8000	15	0.480E BE7B 9D58 566D	x 16^{-12}
23 8652 6FC1 0000	16	0.734A CA5F 6226 F0AE	x 16^{-13}
163 4578 5D8A 0000	17	0.B877 AA32 36A4 B449	x 16^{-14}
DE0 B6B3 A764 0000	18	0.1272 5DD1 D243 ABA1	x 16^{-14}
8AC7 2304 89E8 0000	19	0.1D83 C94F B6D2 AC35	x 16^{-15}

HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0331	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0301	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0212	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
510	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
590	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	4761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	3851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2866	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

APPENDIX G RESERVED SYMBOLS

The following symbols are predefined and cannot be used as user symbols.

ADD
ADDB
ADDBI
ADDI
AND
ANDB
ANDBI
ANDI
BC
CALL
CC
CLR
DB
DD
DEC
DECB
DS
DW
END
ENDS
EQU
EVEN
EXTRN
GA
GB
GC
HLT
INC
INCB
IX
JBT
JMCE
JMCNE
JMP
JNBT
JNZ
JNZB
JZ
JZB
LCALL
LJBT
LJMCE
LJMCNE
LJMP
LJNBT
LJNZ
LJNZB
LJZ
LJZB
LPD
LPDI
MC
MOV
MOVB
MOVBI
MOVI
MOVP
NAME
NOP
NOT
NOTB
OR
ORB
ORBI
ORG
ORI
PP
PUBLIC
SEGMENT
SETB
SINTR
STRUC
TP
TSL
WID
XFER

APPENDIX H
SAMPLE PROGRAM

The following pages show a complete 8089-8086 family program example. The execution vehicles used are an 86/12 Single Board Computer and an 8089 Prototype Board interfaced via the Intel Multibus. In this example, the 8089 unburdens the 8086 by handling message transfers to a CRT and processing message requests. Five messages and a menu (which shows all the message titles) are available for display and selection.

The program listings are shown, the 8086 code compiled in PLM86 and the 8089 code assembled by ASM89. The combination of both these programs should fully explain the initialization and communication protocol between the 8086 and the 8089. Note that the 86/12 Dual Port RAM was set up to appear as upper memory to the 8089 on the Multibus while to the 8086 it appears as lower memory. Further operation is explained throughout the two program listings.

```
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                    PAGE   1


ISIS-II PL/M-86 V1.1 COMPILATION OF MODULE PROTOTYPE89
OBJECT MODULE PLACED IN :F1:PROT89.OBJ
COMPILER INVOKED BY:  PLM86 :F1:PROT89.SRC



               $TITLE('8089 PROTOTYPE DEMO') LARGE OPTIMIZE(2)
   1           PROTOTYPE$89:   DO;
               /****************************************************************************/
               /*                                                                          */
               /*                   DEMO FOR 8089 PROTOTYPE KIT                            */
               /*                                                                          */
               /****************************************************************************/

               /****************************************************************************/
               /*                                                                          */
               /*                      LITERAL DECLARATIONS                                */
               /*                                                                          */
               /****************************************************************************/

               /* 8259A LITERALS */

   2   1           DECLARE

                       INT$STAT$PORT     LITERALLY      '0C0H',
                       INT$MASK$PORT     LITERALLY      '0C2H',
                       INT$ICW1          LITERALLY      '13H' ,
                       INT$ICW2          LITERALLY      '50H' ,
                       INT$ICW4          LITERALLY      '0FH' ,
                       INT$MASK          LITERALLY      '0FEH';


               /*  RAM LOCATIONS FOR THE 8086 */

   3   1           DECLARE

                       SINT$BASE   LITERALLY     '7FF6H' , /* SYSTEM INITIALIZATION BLOCK */
                       SCB$BASE    LITERALLY     '7FE0H' , /* SYSTEM CONTROL BLOCK */
                       CB$BASE     LITERALLY     '7FD0H' , /* COMMAND BLOCK */
                       PB$BASE     LITERALLY     '7000H' , /* PARAMETER BLOCK */
                       TB$BASE     LITERALLY     '70F0H' , /* TASK BLOCK */
                       MSG$BASE    LITERALLY     '7200H' , /* DISPLAY MESSAGE BUFFER */
                       INTR$TYPE   LITERALLY     '0140H' ; /* INTERRUPT VECTOR TABLE */


               /*    RAM LOCATIONS FOR THE 8089 */

   4   1           DECLARE

                       SCB$89     LITERALLY      '0FFFE0H' , /* SYSTEM CONTROL BLOCK */
                       CB$89      LITERALLY      '0FFFD0H' , /* COMMAND BLOCK */
                       PB$89      LITERALLY      '0FF000H' , /* PARAMETER BLOCK */
                       TB$89      LITERALLY      '0FF0F0H' , /* TASK BLOCK */
                       MSG$89     LITERALLY      '0FF200H' ; /* DISPLAY MESSAGE BUFFER */
```

```
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                              PAGE   2



              /*     8089 CCW'S */

   5   1           DECLARE

                      RST$CCW          LITERALLY     '10H' , /* RESET CCW     */
                                                             /* ENABLE INTERRUPTS */
                      INIT$CCW         LITERALLY     '13H' , /* I/O INITIALIZATION CCW */
                                                             /* ENABLE INTERRUPTS      */
                                                             /* EXECUTE TASK BLOCK IN */
                                                             /* SYSTEM MEMORY         */
                      DSP$CCW          LITERALLY     '0BH' ; /* DISPLAY MESSAGE CCW */
                                                             /* RESET INTERRUPT     */
                                                             /* EXECUTE TASK BLOCK  */
                                                             /* IN SYSTEM MEMORY    */


              /*     8089 INITIALIZATION COMMANDS */

   6   1           DECLARE

                      SOC$CMD          LITERALLY     '00H' , /* 8 BIT I/O BUS */
                      SYSBUS$CMD       LITERALLY     '01H' ; /* 16 BIT SYSTEM BUS */


              /*     8089 CHANNEL ATTENTION  */

   7   1           DECLARE

                      CHAN$ATT         LITERALLY     '00H' ;


              /*  MISCELLANEOUS DECLARATIONS  */

   8   1           DECLARE

                      BUSYSTATUS   LITERALLY    '0FFH',
                      TRUE         LITERALLY    '0FFH',
                      FALSE        LITERALLY    '00H',
                      NMBR$MSK     LITERALLY    '07H',
                      CR           LITERALLY    '0DH',
                      LF           LITERALLY    '0AH',
                      ESC          LITERALLY    '1BH',
                      E            LITERALLY    '45H',
                      EOT          LITERALLY    '04H';
```

```
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                              PAGE  3


               /*************************************************************************/
               /*                                                                       */
               /*                          RAM DECLARATIONS                             */
               /*                                                                       */
               /*************************************************************************/

  9   1        DECLARE    SINT    STRUCTURE (SYSBUS WORD, SCB$PTR POINTER) AT (SINT$BASE);

                              /**************************************************/
                              /*                     \    SYSBUS COMMAND        */
                              /**************************************************/
                              /*            SCB          OFFSET                 */
                              /**************************************************/
                              /*            SCB          SEGMENT                */
                              /**************************************************/


 10   1        DECLARE    SCB    STRUCTURE (SOC WORD, CB$PTR POINTER) AT (SCB$BASE);

                               /******************************************************/
                               /*                     \    SOC  COMMAND              */
                               /******************************************************/
                               /*       COMMAND  BLOCK  OFFSET                       */
                               /******************************************************/
                               /*       COMMAND  BLOCK  SEGMENT                      */
                               /******************************************************/


 11   1        DECLARE    CB(2) STRUCTURE (CCW BYTE, BUSY BYTE, PB$PTR POINTER,
                                DUMMY WORD) AT (CB$BASE);

                               /******************************************************/
                               /*     BUSY FLAG          \           CCW             */
                               /******************************************************/
                               /*            PARAMETER  BLOCK  OFFSET                */
                               /******************************************************/
                               /*            PARAMETER  BLOCK  SEGMENT               */
                               /******************************************************/
                               /*                   DUMMY    WORD                    */
                               /******************************************************/

               /*         THE ABOVE COMMAND BLOCK FORMAT IS THE STRUCTURE FORMAT         */
               /*         THE CB ARRAY CONTAINS TWO STRUCTURES; ONE FOR EACH             */
               /*         CHANNEL OF THE 8089.                                           */
```

```
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                          PAGE   4


 12   1      DECLARE     PB     STRUCTURE (TB$PTR POINTER, MSG$PTR POINTER,
                                         LEVEL BYTE, CI BYTE) AT (PB$BASE);

                                /**********************************************************/
                                /*              TASK   BLOCK   OFFSET                   */
                                /**********************************************************/
                                /*              TASK   BLOCK   SEGMENT                  */
                                /**********************************************************/
                                /*          MESSAGE  BUFFER  OFFSET                     */
                                /**********************************************************/
                                /*          MESSAGE  BUFFER  SEGMENT                    */
                                /**********************************************************/
                                /*  CHARACTER FROM CRT  \ DISPLAY LEVEL CMD TO IOP     */
                                /**********************************************************/


 13   1      DECLARE     TB     (512) BYTE AT (TB$BASE);

                                /**********************************************************/
                                /*      RAM BUFFER FOR TASK BLOCK PROGRAM               */
                                /**********************************************************/


 14   1      DECLARE     MSG$BUF (512) BYTE AT (MSG$BASE);

                                 /*********************************************************/
                                 /*        DISPLAY    MESSAGE    BUFFER                  */
                                 /*********************************************************/



 15   1      DECLARE    INTR$VEC$80  POINTER AT (INTR$TYPE);

 16   1      DECLARE    INTR$IP$80   WORD    AT (INTR$TYPE);



             /****************************************************************************/
             /*                                                                        */
             /*                   ROM DECLARATION AND INITIALIZATION                   */
             /*                                                                        */
             /****************************************************************************/

 17   1          DECLARE      MENUE(*) BYTE  DATA (CR,LF,ESC,E,
             '                    ************************************',CR,LF,
             '                    *                                  *',CR,LF,
             '                    *  8086/8089  PROTOTYPE KIT DEMO  *',CR,LF,
             '                    *                                  *',CR,LF,
             '                    ************************************',CR,LF,
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                  PAGE   5


            CR,LF,LF,
            '                SELECTION             TOPIC',CR,LF,LF,
            '                    1           WHAT IS THE 8089 IOP',CR,LF,LF,
            '                    2           WHAT IS THE 8289 BUS ARBITER',CR,LF,LF,LF,
            '                    3           ABOUT THIS DEMONSTRATION',CR,LF,LF,
            '                    4           8089 INITALIZATION PROTOCOL',CR,LF,LF,
            '                    5           8089 COMMUNICATION PROTOCOL',CR,LF,LF,
            LF,LF,LF,
            '              FOR ADDITIONAL INFORMATION ON THE ABOVE TOPICS',CR,LF,
            '              PLEASE SELECT THE APPROPRIATE ENTRY (1,2,3,4,5) - ',EOT);


  18   1    DECLARE    MSG1(*) BYTE DATA(CR,LF,ESC,E,
            '                      8089 I/O PROCESSOR',
            CR,LF,LF,LF,
            '     THE 8089 I/O PROCESSOR IS A FIRST OF ITS KIND SYSTEMS COMPONENT. IT',
            CR,LF,LF,
            'USES THE CONCEPT OF A CHANNEL CONTROLLER, COMMON IN MAINFRAMES, TO SOLVE',
            CR,LF,LF,
            'THE I/O PROCESSING AND HIGH PERFORMANCE DMA REQUIREMENTS OF MICROPROCESSOR',
            CR,LF,LF,
            'SYSTEMS. THE 8089 CAN BE USED IN CONJUNCTION WITH THE 8086 (16 BIT BUS)',
            CR,LF,LF,
            'OR 8088 (8 BIT BUS) AND 8 OR 16 BIT PERIPHERALS TO SIGNIFICANTLY ENHANCE',
            CR,LF,LF,
            'SYSTEM PERFORMANCE. THE 8089 OFFLOADS I/O FROM THE HOST CPU AND PROCESSES',
            CR,LF,LF,
            'CONCURRENTLY WITH CPU ACTIVITY.  ALSO, THE 8089 ADDS INTELLIGENCE TO THE',
            CR,LF,LF,
            'PERIPHERAL SUBSYSTEM WHILE MODULARIZING AND SIMPLIFING THE SYSTEM I/O.',
            CR,LF,LF,
            'EACH IOP HAS TWO I/O CHANNELS THAT CAN PROVIDE DMA AT 1.25 MEGABYTE/SEC,',
            CR,LF,LF,
            'PROCESS INDEPENDENT PROGRAMS, AND HANDLE MULTIPLE I/O DEVICES.',
            CR,LF,LF,
              '               TO SELECT ANOTHER MESSAGE TYPE Y-',EOT);


  19   1    DECLARE     MSG2(*) BYTE DATA (CR,LF,ESC,E,
            '                     THE 8289 BUS ARBITER',
            CR,LF,LF,LF,
            '       THE 8289 BUS ARBITER PROVIDES THE HARDWARE MECHANISMS FOR INTER-',
            CR,LF,LF,
            'PROCESSOR COMMUNICATION AND SHARED RESOURCES IN A MULTIPLE CPU SYSTEM. THE',
            CR,LF,LF,
            '8289 FEATURES SEVERAL USER DEFINABLE PRIORITIZATION AND BUS CONFIGURATIONS.',
            CR,LF,LF,
            'DEMONSTRATED HERE, THE RESB MODE SEPERATES 86/12 PRIVATE RESOURCES FROM',
            CR,LF,LF,
            'SYSTEM BUS SHARED RESOURCES, WHILE THE IOB MODE DIVIDES THE 8089 I/O BUS',
            CR,LF,LF,
            'FROM THE SYSTEM BUS. IN BOTH CASES THE 8289 COMPLETELY ARBITRATES SYSTEM',
            CR,LF,LF,
            'BUS USAGE TO MANAGE MULTIPLE PROCESSOR CONTENTION.',
            CR,LF,LF,
            '       THE 8086 FAMILY AND MULTIBUS CONCEPT ALLOWS PARTITIONING APPLICATIONS',
            CR,LF,LF,
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                    PAGE   6


               'INTO SMALLER MORE MANAGEABLE TASKS.  THUS, ADDING NEW FUNCTIONS OR UPGRADING',
               CR,LF,LF,
               'EXISTING ONES WILL HAVE MINIMAL EFFECT ON THE ORIGINAL DESIGN.',
               CR,LF,LF,
               '                  TO SELECT ANOTHER MESSAGE TYPE Y--',EOT);


  20    1      DECLARE      MSG3(*) BYTE  DATA(CR,LF,ESC,E,
                 '                      ABOUT THIS DEMONSTRATION',
               CR,LF,LF,LF,
               '    TO DEMONSTRATE THE 8086 FAMILY CPU-IOP CONCEPT, AN SBC 86/12 AND AN 8089',
               CR,LF,LF,
               'PROTOTYPE BOARD ARE INTERFACED VIA THE INTEL MULTIBUS.  IN THIS DEMO THE 8089',
               CR,LF,LF,
               'UNBURDENS THE 8086 BY HANDLING MESSAGE TRANSFERS TO THE CRT AND PROCESSING',
               CR,LF,LF,
               'MESSAGE REQUESTS.  OPERATION IS AS FOLLOWS: USING A CHANNEL ATTENTION (CA) THE',
               CR,LF,LF,
               '8086 INITIALIZES THE 8089 AND CAUSES IT TO EXECUTE A TASK BLOCK TO PROGRAM',
               CR,LF,LF,
               'THE PERIPHERAL DEVICES ON ITS LOCAL BUS.  THE 8089 THEN INTERRUPTS THE 8086',
               CR,LF,LF,
               'TO REQUEST A MESSAGE FOR DISPLAY.  RESPONDING, THE 8086 SETS UP LINKAGE TO',
               CR,LF,LF,
               'THE TASK BLOCK PROGRAM AND ISSUES A CA TO THE 8089.  AFTER EACH CA THE 8089',
               CR,LF,LF,
               'DISPLAYS THE MESSAGE, POLLS THE CRT TERMINAL FOR A VALID MESSAGE REQUEST AND',
               CR,LF,LF,
               'THEN INTERRUPTS THE 8086.  HENCEFORTH THE CYCLE IS REPEATED.',
               CR,LF,LF,
                  '                  TO SELECT ANOTHER MESSAGE TYPE Y--',EOT);


  21    1      DECLARE     MSG4(*) BYTE       DATA (CR,LF,ESC,E,
               '                      8089  INITIALIZATION PROTOCAL',
               CR,LF,LF,LF,
               'SYSTEM INITALIZATION      ++++++++++++++++++++++++++++++++++++++++++++',
               CR,LF,
               '                          +                     +  SYSBUS COMMAND     +',
               CR,LF,
               '                          ++++++++++++++++++++++++++++++++++++++++++++',
               CR,LF,
               '                          +      SYSTEM CONTROL BLOCK ADDRESS         +',
               CR,LF,
               '                          ++++++++++++++++++++++++++++++++++++++++++++',
               CR,LF,
               'SYSTEM CONTROL BLOCK      ++++++++++++++++++++++++++++++++++++++++++++',
               CR,LF,
               '                          +                      +  SOC  COMMAND      +',
               CR,LF,
               '                          ++++++++++++++++++++++++++++++++++++++++++++',
               CR,LF,
               '                          +         COMMAND  BLOCK  ADDRESS           +',
               CR,LF,
               '                          ++++++++++++++++++++++++++++++++++++++++++++',
               CR,LF,LF,LF,
               '       ON THE FIRST CHANNEL ATTENTION AFTER RESET, THE IOP READS THESE',
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                                   PAGE   7


              CR,LF,LF,
              'CONTROL BLOCKS TO DETERMINE THE WIDTH OF THE SYSTEM BUS (8 OR 16), THE',
              CR,LF,LF,
              'I/O BUS WIDTH (8 OR 16), PRIORITY INFORMATION, AND WHERE TO FIND INFORMATION',
              CR,LF,LF,
              'DEFINING SUBSEQUENT CHANNEL ATTENTIONS (THE COMMAND BLOCK).',
              CR,LF,LF,
                   '              TO SELECT ANOTHER MESSAGE TYPE Y-',EOT);


  22   1      DECLARE     MSG5(*) BYTE
                       DATA(CR,LF,ESC,E,
              '                      8089 TASK COMMUNICATION PROTOCOL',
              CR,LF,LF,LF,
              '                       ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '  COMMAND BLOCK        +   BUSY FLAG          +   CHANNEL COMMAND WORD  +',
              CR,LF,
              '   (ONE PER CHANNEL)   ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '                       +           PARAMETER  BLOCK  ADDRESS            +',
              CR,LF,
              '                       ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '  PARAMETER BLOCK      ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '                       +              TASK  BLOCK  ADDRESS              +',
              CR,LF,
              '                       ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '                       +           USER DEFINED MESSAGE AREA            +',
              CR,LF,
              '                       ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '  TASK BLOCK           ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,
              '                       +    TASK PROGRAM TO BE EXECUTED BY THE 8089     +',
              CR,LF,
              '                       ++++++++++++++++++++++++++++++++++++++++++++++++++',
              CR,LF,LF,
              '     AFTER A CHANNEL ATTENTION, THE 8089 READS THESE BLOCKS TO SEE WHAT THE',
              CR,LF,LF,
              'CPU WANTS (CHANNEL COMMAND WORD) AND WHERE TO FIND ADDITIONAL INFORMATION',
              CR,LF,LF,
              '(PARAMETER BLOCK).  THE PARAMETER BLOCK GIVES THE TASK PROGRAM ADDRESS AND',
              CR,LF,LF,
              'PARAMETERS TO BE PASSED.       TO SELECT ANOTHER MESSAGE TYPE Y-',EOT);


  23   1      DECLARE     INITTB(60) BYTE EXTERNAL;     /* TB TO INITIALIZE       */
                                                        /* 8251A & 8253           */

  24   1      DECLARE     PROGTB(120) BYTE EXTERNAL;    /* TB FOR MESSAGE DISPLAY */
```

```
              /*****************************************************************************/
              /*    THIS IS THE MAIN PROGRAM WHICH INITALIZES THE 8089 FROM RESET AND      */
              /* THEN ISSUES THE 89 A CA TO EXECUTE A TASK BLOCK WHICH INITALIZES THE      */
              /* 8251A AND THE 8253.  AFTER ALL INITALIZATION IS COMPLETE, THE PROGRAM     */
              /* IS TOTALLY INTERRUPT DRIVEN FROM THE 8089.  THE 8089 INTERRUPTS THE       */
              /* 8086 TO REQUEST A NEW MESSAGE FOR DISPLAY.  TO SERVICE THE INTERRUPT,     */
              /* THE 8086 TRANSFERS THE NEW MESSAGE FROM ROM TO THE MESSAGE BUFFER, SETS */
              /* UP THE APPROPRIATE TASK BLOCK PROGRAM AND ISSUES A NEW CA TO THE IOP TO */
              /* ALLOW IT TO DISPLAY THE NEW MESSAGE.  THE 8086 WILL HALT AFTER ISSUEING */
              /* THE CHANNEL ATTENTION AND WAIT FOR THE NEXT MESSAGE REQUEST.              */
              /*    AFTER EACH CA, THE 8089 WILL DISPLAY THE REQUESTED MESSAGE THEN POLL */
              /* FOR A NEXT MESSAGE REQUEST ENTERED AT THE CRT.  UPON RECEIVING A VALID  */
              /* REQUEST THE 8089 RETURNS THE REQUEST TO THE 8086, ISSUES AN INTERRUPT   */
              /* TO THE 8086 AND HALTS ITS CURRENT TB EXECUTION.  THE 8089 PERFORMS NO   */
              /* OTHER ACTIVITIES UNTIL AWAKENED BY THE CA FROM THE 8086 TO DISPLAY THE  */
              /* NEXT MESSAGE.                                                             */
              /*****************************************************************************/


  25   1      MSGDSPL:     PROCEDURE INTERRUPT 80 PUBLIC;

  26   2          IF  PB.CI='Y' THEN

  27   2              DO;
  28   3                  CALL MOVB(@MENUE, @MSG$BUF,SIZE(MENUE));
  29   3                  PB.LEVEL = FALSE;
  30   3              END;

  31   2          ELSE DO;

  32   3              PB.LEVEL = TRUE;
  33   3              DO CASE (PB.CI AND NMBR$MSK)-1;
  34   4                  CALL MOVB (@MSG1, @MSG$BUF, SIZE (MSG1));
  35   4                  CALL MOVB (@MSG2, @MSG$BUF, SIZE (MSG2));
  36   4                  CALL MOVB (@MSG3, @MSG$BUF, SIZE (MSG3));
  37   4                  CALL MOVB (@MSG4, @MSG$BUF, SIZE (MSG4));
  38   4                  CALL MOVB (@MSG5, @MSG$BUF, SIZE (MSG5));
  39   4              END;
  40   3          END;

  41   2          CALL MOVB (@PROGTB,@TB, SIZE (PROGTB));

  42   2          PB.TB$PTR = TB$89;
  43   2          PB.MSG$PTR = MSG$89;

  44   2              CB(0).CCW = DSP$CCW;
  45   2              CB(0).PB$PTR = PB$89;

  46   2          OUTPUT(CHAN$ATT)=00H;

  47   2          RETURN;

  48   2          END MSGDSPL;
```

```
PL/M-86 COMPILER    8089 PROTOTYPE DEMO                                                    PAGE   9


 49   1       START:    DISABLE;

 50   1             INTR$VEC$80 = @MSGDSPL;
 51   1             INTR$IP$80  = INTR$IP$80 - 27;

 52   1             OUTPUT(INT$STAT$PORT) = INT$ICW1;
 53   1             OUTPUT(INT$MASK$PORT) = INT$ICW2;
 54   1             OUTPUT(INT$MASK$PORT) = INT$ICW4;
 55   1             OUTPUT(INT$MASK$PORT) = INT$MASK;


 56   1             SINT.SYSBUS = SYSBUS$CMD;
 57   1             SINT.SCB$PTR = SCB$89;


 58   1             SCB.SOC = SOC$CMD;
 59   1             SCB.CB$PTR = CB$89;

 60   1             CB(0).CCW = RST$CCW;
 61   1             CB(0).BUSY = BUSYSTATUS;

 62   1             OUTPUT(CHAN$ATT) = 0;

 63   1             DO WHILE CB(0).BUSY = BUSYSTATUS;
 64   2             END;

 65   1             CALL MOVB(@INITTB,@TB,SIZE(INITTB));
 66   1             CB(0).CCW = INIT$CCW;
 67   1             CB(0).PB$PTR = PB$89;
 68   1             PB.TB$PTR = TB$89;
 69   1             OUTPUT(CHAN$ATT) = 0;

 70   1             ENABLE;
 71   1             DO WHILE TRUE <> FALSE;
 72   2             END;


 73   1       END PROTOTYPE$89;



MODULE INFORMATION:

     CODE AREA SIZE     = 1932H   6450D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0022H     34D
     488 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
8089 ASSEMBLER                                                                PAGE    1


ISIS-II 8089 ASSEMBLER V 1.0 ASSEMBLY OF MODULE DEMO89
OBJECT MODULE PLACED IN :F1:89DEMO.OBJ
ASSEMBLER INVOKED BY ASM89 :F1:89DEMO.SRC PAGELENGTH(63)


                              ;*********************************************************************
                              ;*                                                                   *
                              ;*                        8089 DEMO PROGRAM                          *
                              ;*                                                                   *
                              ;*********************************************************************
                              ;
                              NAME    DEMO89
0000                          DEMO    SEGMENT
                              ;
                              PUBLIC  INITTB
                              PUBLIC  PROGTB
                              ;
                              ;EQUATES
                              ;
 C000                         DADDRESS_8251    EQU     0C000H          ;
 C001                         CADDRESS_8251    EQU     0C001H          ;
 00CA                         MODE_8251        EQU     0CAH            ;
 0040                         RST_8251         EQU     40H             ;
 0025                         COMMAND_8251     EQU     25H             ;
 E003                         MADDRESS_8253    EQU     0E003H          ;
 0037                         MODE_8253        EQU     37H             ;
 E000                         CADDRESS_8253    EQU     0E000H          ;
 0065                         COUNT0LSB_8253   EQU     65H             ;
 0000                         COUNT0MSB_8253   EQU     0               ;
 0059                         Y                EQU     59H             ;
 0009                         CI               EQU     9H              ;
 0004                         MSG_POINTER      EQU     4H              ;
 FF04                         EOT_COMPARE      EQU     0FF04H          ;
 0008                         LEV              EQU     8H              ;
 FF59                         Y_COMPARE        EQU     0FF59H          ;
 F837                         MSG_COMPARE      EQU     0F837H          ;
 FE37                         SIX_SEV_COMPARE  EQU     0FE37H          ;
 FF30                         ZERO_COMPARE     EQU     0FF30H          ;
                              ;
                              ;TASK1 - INTIALIZATION
                              ;
0000   3130 01C0              INITTB: MOVI    GB, CADDRESS_8251       ;INITIALIZE 8251
0004   084D CA                        MOVBI   [GB], MODE_8251         ;
0007   0000                           NOP
0009   0000                           NOP
000B   084D 40                        MOVBI   [GB], RST_8251          ;SOFTWARE RESET
000E   0000                           NOP
0010   0000                           NOP
0012   084D CA                        MOVBI   [GB], MODE_8251         ;2 STOP, CHAR LENGTH 7, X16
0015   0000                           NOP
0017   0000                           NOP
0019   084D 25                        MOVBI   [GB], COMMAND_8251      ;REC AND TRAN ENABLED
001C   084D 37                        MOVBI   [GB],MODE_8253          ;CNT 0, MODE 3, BCD
001F   3130 00E0                      MOVI    GB, CADDRESS_8253       ;
0023   084D 65                        MOVBI   [GB], COUNT0LSB_8253    ;LSB = 65
0026   084D 00                        MOVBI   [GB], COUNT0MSB_8253    ;MSB = 0
0029   0A4F 09 59                     MOVBI   [PP].CI, Y              ;Y TO CI BYTE IN PARAMETER BLOCK
                                                                       ;TO SELECT MENUE FOR DISPLAY
002D   4000                           SINTR                           ;INTERRUPT 8086
002F   2048                           HLT                             ;WAIT FOR CA
```

```
8089 ASSEMBLER                                                                          PAGE    2

                                     ;
                                     ;TASK2 - SEND MESSAGE AND MONITOR CONSOLE
                                     ;
0031   5130 01C0                     PROGTB: MOVI    GC, CADDRESS_8251     ;8251 STATUS ADDR
0035   3130 00C0                             MOVI    GB, DADDRESS_8251     ;8251 DATA ADDR
0039   038B 04                       SEND:   LPD     GA, [PP].MSG_POINTER  ;SEND MESSAGE TO CRT UNTIL EOT
003C   B130 0000                             MOVI    IX, 0                 ;
0040   F130 04FF                             MOVI    MC,EOT_COMPARE        ;MASK COMPARE FOR EOT
0044   0CB0 0A                       EOTCOM: JMCE    [GA+IX], LEVEL        ;EOT ?
0047   08BA FD                       TXRDY2: JNBT    [GC], 0, TXRDY2       ;TRANSMIT READY ?
004A   0690 00CD                             MOVB    [GB], [GA+IX+]        ;SEND CHARACTER TO 8251
004E   8820 F3                               JMP     EOTCOM                ;
0051   0AE7 08 14                    LEVEL:  JZB     [PP].LEV, MSGSEL      ;CHECK LEVEL BYTE IN PARAMETER BLOCK,
                                                                            ;MENUE OR MESSAGE ?
0055   F130 59FF                     MENSEL: MOVI    MC, Y_COMPARE         ;MASK COMPARE FOR Y
0059   28BA FD                       RXRDY1: JNBT    [GC], 1, RXRDY1       ;RECEIVE READY ?
005C   08B5 FA                               JMCNE   [GB], RXRDY1          ;Y ?
005F   0A4F 09 59                            MOVBI   [PP].CI, Y            ;Y TO CI BYTE IN PARAMETER BLOCK
0063   084D 59                               MOVBI   [GB], Y               ;ECHO
0066   8820 25                               JMP     INTR86                ;
0069   F130 37F8                     MSGSEL: MOVI    MC, MSG_COMPARE       ;MASK COMPARE FOR MESSAGE SELECT
006D   28BA FD                       RXRDY2: JNBT    [GC], 1, RXRDY2       ;RECEIVE READY ?
0070   0091 02CF 09                          MOVB    [PP].CI, [GB]         ;MESSAGE SELECTION TO CI BYTE
                                                                            ;IN PARAMETER BLOCK
0075   0AB7 09 F4                            JMCNE   [PP].CI, RXRDY2       ;0 THRU 7 ?
0079   F130 37FE                             MOVI    MC, SIX_SEV_COMPARE   ;MASK COMPARE FOR 6 OR 7
007D   0AB3 09 E8                            JMCE    [PP].CI, MSGSEL       ;6 OR 7 ?
0081   F130 30FF                             MOVI    MC, ZERO_COMPARE      ;MASK COMPARE FOR 0
0085   0AB3 09 E0                            JMCE    [PP].CI, MSGSEL       ;0 ?
0089   0293 09 00CD                          MOVB    [GB], [PP].CI         ;ECHO
008E   4000                          INTR86: SINTR                         ;INTERRUPT 8086
0090   2048                                  HLT                           ;WAIT FOR CA
                                     ;
0092                                 DEMO    ENDS
                                     END
```

```
8089 ASSEMBLER                                                   PAGE   3

SYMBOL TABLE
------------

 DEFN VALUE TYPE  NAME
 ---- ----- ----  ----

   22  E000  SYM  C0ADDRESS_8253
   16  C001  SYM  CADDRESS_8251
   26  0009  SYM  CI
   19  0025  SYM  COMMAND_8251
   23  0065  SYM  COUNT0LSB_8253
   24  0000  SYM  COUNT0MSB_8253
   15  C000  SYM  DADDRESS_8251
    8  0000  SYM  DEMO
   64  0044  SYM  EOTCOM
   28  FF04  SYM  EOT_COMPARE
   37  0000  PUB  INITTB
   86  008E  SYM  INTR86
   29  0008  SYM  LEV
   68  0051  SYM  LEVEL
   20  E003  SYM  MADDRESS_8253
   70  0055  SYM  MENSEL
   17  00CA  SYM  MODE_8251
   21  0037  SYM  MODE_8253
   76  0069  SYM  MSGSEL
   31  F837  SYM  MSG_COMPARE
   27  0004  SYM  MSG_POINTER
   59  0031  PUB  PROGTB
   18  0040  SYM  RST_8251
   71  0059  SYM  RXRDY1
   77  006D  SYM  RXRDY2
   61  0039  SYM  SEND
   32  FE37  SYM  SIX_SEV_COMPARE
   65  0047  SYM  TXRDY2
   25  0059  SYM  Y
   30  FF59  SYM  Y_COMPARE
   33  FF30  SYM  ZERO_COMPARE

ASSEMBLY COMPLETE; NO ERRORS FOUND
```

APPENDIX J ASSEMBLER ERROR MESSAGES/USER ACTIONS

ASM89 error messages are numbered according to the following general scheme:

- 1 - 120 User-provoked errors–Nonfatal
- 121 - 150 Command tail/control line errors–Fatal/Nonfatal
- 151 - 200 Source statement errors–Statement processing abandoned
- 201 - 240 Assembler errors–Not user-provoked
- 241 - 255 Fatal errors–Assembly terminated

Nonfatal errors place an error message or error messages in the list file immediately following the source statement which provoked the error. The format of nonfatal error messages is:

```
*** ERROR <n>: <error text>
```

where "n" is the error number. The assembly of subsequent source statements is not affected by nonfatal errors.

Fatal errors terminate the assembler's processing of the source file and return system control to ISIS. There are two types of fatal errors:

- Fatal I/O errors
- All other fatal errors

Fatal I/O errors provoke the following console message:

```
ASM89 I/O ERROR—

    FILE:   <filename>

    ERROR: <description>

ASSEMBLY TERMINATED
```

All other fatal errors provoke the console message:

```
ASM89 FATAL ERROR—<description>
```

Assembler errors should never occur. If you get one of these error messages, please notify Intel Corporation via a Problem Report Form (Part Number 9800035).

The construct (X) in any message is replaced by a statement-dependent error construct; it may be a number, a quoted string, a register–almost anything. Error constructs in the same error message may differ if the message is provoked by two different source statements.

Most assembler error messages are self-explanatory. Where necessary, a brief error explanation and a description of the action to be taken by the user follows the error message.

*** ERROR 1: PASS ONE ENCOUNTERED (X) FURTHER ERRORS IN THIS STMT

This error message is issued after eight errors are found in a source statement on the assembler's first pass. Pass two errors are listed before pass one errors for a given statement.

*** ERROR 2: PASS TWO ENCOUNTERED (X) FURTHER ERRORS IN THIS STMT

This error message is issued after eight errors are found in a source statement on the assembler's second pass. Pass two errors are listed before pass one errors for a given statement.

*** ERROR 3: (X) WAS DECLARED PUBLIC, BUT NEVER DEFINED; NOT WRITTEN TO OBJECT

The symbol X is declared public in a PUBLIC directive but not defined in the source file. Information normally written to the object file for public symbols is not written for X. A source statement defining X should be added to the source file or X should be deleted from the PUBLIC directive it appears in.

*** ERROR 4: SOURCE TEXT FOLLOWS "END" STATEMENT; IGNORED

Any source file statements following the END directive are ignored by the assembler. To be processed by the assembler, such statements must be placed before the END directive.

*** ERROR 5: NO SEGMENT WAS DEFINED; NO OBJECT FILE WILL BE PRODUCED

Every 8089 Assembly Language source file must define exactly one segment, using the SEGMENT/ENDS assembler directives. If such a segment is not defined in the source file, no object code is generated by the assembler. Any existing object files are retained.

*** ERROR 6: "END" STATEMENT IN INCLUDED FILE

An INCLUDEd file contains an END directive. The assembler accepts the statement and all source statements following the END directive are ignored by the assembler. Only one END directive is allowed per source file; INCLUDEd files are terminated by an end-of-file condition.

*** ERROR 7: STATEMENT TOO COMPLEX; OPERANDS IGNORED STARTING WITH #(X)

The expression list for a DB, DW, or DD assembler directive contains more expressions than the assembler can process. The directive should be broken up into two or more statements. Should this error message be generated by a single expression, a simpler expression must be coded in its place.

*** ERROR 11: SEGMENT (X) IS LONGER THAN 64K BYTES

The segment contained in an ASM89 object module can be a maximum of 64k contiguous byte addresses in length. This error message indicates that the 8089 Assembly Language source program attempts to generate an object module which exceed this limit. The following source file is an example:

```
SEG89   SEGMENT

        ORG     0FFFFH

DATA:   DS      128

SEG89   ENDS

        END
```

The user should check ORG directives for errors. If more than 64k contiguous byte addresses are neccessary, two 8089 Assembly Language source files, a different segment defined in each, must be created.

*** ERROR 12: NAME/LABEL IS FORBIDDEN

A label or name precedes an assembler directive which cannot be labeled or named. For example:

```
FINISHED:  END
```

The END, ORG, EVEN, NAME, PUBLIC, and EXTRN directives cannot be labeled or named.

*** ERROR 13: LABEL USED IN NAME CONTEXT; NAME ASSUMED

*** ERROR 14: NAME USED IN LABEL CONTEXT; LABEL ASSUMED

*** ERROR 15: (X) IS DECLARED BOTH PUB AND EXT; ORIGINAL DEFN USED

The symbol X appears in both a PUBLIC and an EXTRN assembler directive. The first directive is used; the second is ignored. For example:

```
PUBLIC     FOO

EXTRN      FOO
```

The symbol FOO is assumed to be public by the assembler. Symbols cannot be declared both public (PUBLIC) and external (EXTRN).

*** ERROR 16: (X) HAS ALREADY BEEN DECLARED PUBLIC

A symbol can be declared public (PUBLIC) only once in a source file. Additional public declarations of (X) should be deleted.

*** ERROR 17: (X) HAS ALREADY BEEN DECLARED EXTERNAL

A symbol can be declared external (EXTRN) only once in a source file. Additional external declarations of (X) should be deleted.

*** ERROR 18: (X) HAS ALREADY BEEN DECLARED LOCAL; EXT IGNORED

This message appears after an EXTRN directive which includes a symbol already defined as a label or a name in the source file. The external declaration is ignored.

*** ERROR 19: NAME MISMATCH WHEN CLOSING <construct>

The <construct> is either SEGMENT (X) or STRUCTURE (X). The wrong name in an ENDS statement, or trying to close a SEGMENT directive while a STRUCTURE directive is still open will provoke this message. For example:

```
THIS       STRUCTURE

THAT       ENDS
```

The second statement is assumed to read "THIS ENDS".

*** ERROR 20: "ENDS" ASSUMED TO CLOSE <construct>

The <construct> is SEGMENT (X), STRUCTURE (X), or UNNAMED STRUCTURE. This error message follows an ENDS directive which has no name.

*** ERROR 21: <construct> IS ASSUMED TO CLOSE AT "END"

The <construct> is SEGMENT (X), or STRUCTURE (X), or UNNAMED STRUCTURE. An END directive was found before the ENDS closing an active segment or structure.

*** ERROR 24: BAD PARAMETER TO PSEUDO-OP; IGNORED

Provoked by undefined or invalid operands to DS and ORG assembler directives. For example:

```
DS        GA

ORG       'ABCDEF'

DS        ZZZ
```

In the last example, this error is provoked if ZZZ has not been defined to the assembler when the DS directive is processed.

*** ERROR 25: TOO MANY OPERANDS; IGNORED BEGINNING WITH #(X)

An 8089 Assembly Language source statement contains too many operands. For example:

```
JMP       TARGET, ANOTHER
```

The JMP instruction only requires one operand.

*** ERROR 26: "EQU" DOES NOT ALLOW REGISTER EXPRESSIONS; FIRST REG IS USED

Provoked by such things as the following:

```
REG       EQU       GA+GB

REG2      EQU       GB-1
```

Everything following the first register is ignored. The above statements are equivalent to:

```
REG       EQU       GA

REG2      EQU       GB
```

*** ERROR 27: OPERAND OF "EQU" IS AS YET UNDEFINED; ASSUMED ZERO

The operand of an EQU directive is undefined when the EQU is found on the assembler's first pass. The operand's value is assumed to be zero. For example:

```
ENDJ      EQU       LAST

 LAST:    HLT
```

The value of LAST is assumed to be zero when the EQU directive is processed. ENDJ is assigned the value zero.

*** ERROR 28: MODULE NAME IS ALREADY (X); STATEMENT IGNORED

A source file contains two NAME directives. Only one NAME directive is allowed per source file.

*** ERROR 29: ILLEGAL OPERAND TO PUBLIC/EXTRN

*** ERROR 30: NULL OPERAND IS ASSUMED ZERO

An instruction requires more operands than are contained in the source statement. For example,

```
ADD       GA,
```

The missing operand is assumed to be zero.

*** ERROR 31: (X) IS AN INVALID BASE-(X) DIGIT; (X) IS ASSUMED ZERO

This error message is provoked by such source statements as the following:

```
DB          0F7
```

0F7 is assumed to be decimal and F is an invalid decimal digit. The digit in error must be changed or the correct suffix for the desired number system must be added to the number.

*** ERROR 32: SYMBOL IS LONGER THAN 31 CHARACTERS; TRUNCATED TO 31

Symbols can be a maximum of 31 characters in length. Symbols which exceed this limit are truncated by the assembler. The entire symbol does, however, appear in the list file.

*** ERROR 33: TOKEN IS LONGER THAN 255 CHARACTERS; TRUNCATED TO 255

*** ERROR 34: OPERATION DOES NOT ALLOW AN EXTERNAL SYMBOL; EXTERNAL ASSUMED ZERO

External symbols are only allowed in DD assembler directives and LPDI instructions.

*** ERROR 35: ILLEGAL EXPRESSION; ZERO USED

Assembler error—contact Intel Corporation.

*** ERROR 36: NO "END" STATEMENT

The source file does not contain an END directive. The assembler acts as if an END directive immediately precedes the end of the source file.

*** ERROR 37: ILLEGAL OPERAND TO DATA-GENERATING OP; IGNORED

This error message is provoked by invalid operands to DB, DW, DD, and DS assembler directives. For example:

```
DB          [GA]
```

The invalid operand must be changed or deleted.

*** ERROR 38: STRINGS LONGER THAN 2 CHARS ARE FORBIDDEN; IGNORED

*** ERROR 39: BIT SELECTOR IS OUT OF RANGE; VALUE MOD 8 IS USED

The value of a data memory bit operand in an instruction ranges from 0–7. Values outside this range are taken modulo eight by the assembler. For example:

```
SETB        [GA], 11
```

The assembler assumes bit 3 (11 modulo eight) is specified.

*** ERROR 40: UNRECOGNIZED MEMORY REFERENCE IS ASSUMED REGISTER DIRECT

Assembler error—contact Intel Corporation.

*** ERROR 41: NON-REGISTER (X) IS ASSUMED TO BE REGISTER GA

Nonregister symbols used in place of register operands provoke this error message. For example:

```
OR          GD, [PP].CNTRL
```

GD is assumed by the assembler to be GA.

*** ERROR 42: NON-POINTER REGISTER (X) IS ASSUMED TO BE REGISTER GA

This error message is provoked when an instruction requires a pointer register operand and a non-pointer register operand is coded. For example:

```
LPD        BC, [PP].ADDRESS
```

BC is assumed to be GA by the assembler, so the above is equivalent to:

```
LPD        GA, [PP].ADDRESS
```

*** ERROR 43: ILLEGAL SOURCE WIDTH; ASSUMED 8

The source operand in the WID instruction can be 8 or 16. Any other value is assumed by the assembler to be 8. The destination operand in the WID instruction is checked separately by the assembler, so two incorrect logical width operands generate two error messages. Example:

```
WID        12, 16
```

The above statement is treated as WID 8, 16 (not WID 8, 8).

*** ERROR 44: ILLEGAL DESTINATION WIDTH; ASSUMED 8

The destination operand in the WID instruction can be 8 or 16. Any other value is assumed by the assembler to be 8. The source operand is checked separately by the assembler, so two incorrect logical width operands generate two error messages. Example:

```
WID        16, 18
```

The assembler assumes the above to be WID 16, 8 (not WID 8, 8).

*** ERROR 45: JUMP TARGET IS OUTSIDE 1-BYTE WINDOW; WRAPAROUND

The one-byte window is the range of the jump target's address from the end of a control transfer instruction (next instruction address − 128, next instruction address + 127). When the short form of a control transfer instruction is coded, this error occurs when the assembler cannot determine the address of the jump target on its first pass (i.e., the expression giving the jump target's location contains a forward reference). The assembler assumes a signed byte displacement value (of the above range) is required to reach the jump target. If it later determines that a signed word displacement is needed, the short form of the control transfer instruction is flagged as an error.

The user must either: code the long form of the control transfer instruction in place of the short form or eliminate the forward reference in the expression specifying the jump target's location.

NOTE: WRAPAROUND means that the required displacement value has wrapped around within the signed byte value. Thus, the value generated by the assembler is incorrect. For example, if a displacement value of +140 is required the assembler generates a value −116.

*** ERROR 46: JUMP TARGET IS OUTSIDE 2-BYTE WINDOW; WRAPAROUND

The two-byte window is the range of the jump target's address from the end of a control transfer instruction (next instruction address − 32,768, next instruction address + 32,767). All 8089 Assembly Language control transfer instruction jump targets must be in the above range.

The user must move the location of the jump target inside the above range (next instruction − 32,768, next instruction + 32,767). If, in the control transfer instruction, the expression specifying the jump target's location does not contain a forward reference, the short form of the control transfer instruction can be coded and the assembler will generate a signed byte or word displacement as is necessary. (Note that $ + 7 is not a forward reference.) If the expression does contain a forward reference and the jump target is outside a −128, +127 byte range, the long form of the instruction is required.

NOTE: WRAPAROUND means that the displacement value wraps around within a signed word. The assembler does not generate the correct displacement value. For example, a displacement of +65000 generates a displacement value of −536.

*** ERROR 47: MEMORY REFERENCE OFFSET IS > 255; VALUE MOD 256 IS USED

The value of 'd' in the data memory expression form [PREG].d cannot be greater than 255. Example:

```
        MOV      GA, [PP].300
```

The offset value 300 is evaluated modulo 256 and the above expression is treated as:

```
        MOV      GA, [PP].44
```

*** ERROR 48: (X) IS ALREADY DEFINED; REDEFINITION IS IGNORED

This message is provoked when a symbol is defined more than once in a source file. Example:

```
        FOO      EQU      0FFH

        FOO:     DB       8
```

The second use of FOO (as a label) provokes this error. This error might also occur if an INCLUDEd file defines a symbol already defined in the main source file (e.g., FOO is used as an instruction label in both the main source file and an INCLUDEd file). Additional definitions of (X) must be eliminated.

*** ERROR 49: EXPRESSION HAS MORE THAN ONE EXTERNAL; (X) IS ASSUMED ZERO

A single external symbol can appear in an expression used in an LPDI instruction or DD directive. Example:

```
         EXTRN   DOG, CAT

             DD  DOG + CAT
```

The assembler assumes the value of CAT, and any other external symbols in the expression, to be zero.

Note that the following is valid:

```
         EXTRN   DOG, CAT

             DD  DOG, CAT
```

In this case, the external symbols appear in two different expressions.

*** ERROR 50: STATEMENT BEGINS WITH CONTINUATION

A source statement cannot begin in an INCLUDEd file and continue in the main source file, i.e., the first source line following an INCLUDE control line cannot begin with an &. The source statement must be contained in either the INCLUDEd file or the main source file. It cannot be continued from one to the other.

*** ERROR 51: END-OF-FILE WITHIN QUOTED STRING

This error message is provoked by source files ending with the following statement (no end-of-line at end of statement):

```
DB          'ABC
```

The quoted string is assumed to end at the end-of-file.

*** ERROR 52: END-OF-FILE DOES NOT OCCUR ON A LINE BOUNDARY

This error message is generated by an END statement not followed by an end-of-line.

*** ERROR 53: LINE ENDS BEFORE QUOTED STRING

A quoted string cannot contain an end-of-line (a single carriage-return (CR), a single linefeed, or a CR/LF sequence).

*** ERROR 54: ILLEGAL CHARACTER ENCOUNTERED

The assembler accepts all printing characters of the standard ASCII character set. The non-printing characters horizontal tab (09H), carriage-return (0DH) and line-feed (0AH) may also be used with assembler-defined meanings (tab and end-of-line). Invalid characters are treated as a blank by the assembler.

*** ERROR 55: LINE/STATEMENT ENDS BEFORE QUOTED STRING

The quoted string is assumed to close at the end-of-line or end-of-statement.

*** ERROR 56: (X) IS NOT A MEMORY REFERENCE REGISTER; REF BECOMES [GA]; SKIP TO COMMA OR END-OF-LINE

Pointer/registers GA, GB, or GC and the PP register can be used in memory reference expressions. This error is provoked by the following kind of statement:

```
NOT         [BC]
```

BC must be replaced with GA, GB, GC or PP.

*** ERROR 57: INDEXING ASSUMED VIA IX, NOT (X); SKIP TO COMMA OR END-OF-LINE

Expressions of the form:

```
MOV         GA, [PP + BC]
```

provoke this error. The second operand is assumed to read [PP+IX].

*** ERROR 58: VALUE OF REGISTER (X) IN EXPRESSION SET TO ZERO

The following type of expression provokes this error:

```
ADD         MC, [GB].IX
```

IX is not a valid offset. The assembler assumes a zero offset value.

*** ERROR 59: NOT ENOUGH OPERANDS IN AN EXPRESSION

This error message is provoked by the following kind of expression:

```
GOO         EQU         $ +
```

The assembler expects an operand following the + sign. An operand should be provided or the + sign removed from the statement.

*** ERROR 60: OPERATOR OR DELIMITER EXPECTED BEFORE (X);
SKIP TO COMMA OR END-OF-LINE

An operator, + or −, or a delimiter, , or ;, has been forgotten or mistyped. This error message is provoked by statements of the form:

```
JMP        TARGET   5

AND        GA, [GC]   THIS IS AN AND INSTRUCTION.
```

The assembler skips to the next comma or end-of-line.

*** ERROR 63: (X) (ILLEGAL IN EXPRESSION) IS ASSUMED TO BE ZERO

*** ERROR 64: DOT IS ILLEGAL IN THIS CONTEXT; SKIP TO COMMA OR END-OF-LINE

*** ERROR 65: "STRUCTURE" EXPECTS A NAME; UNNAMED STRUCTURE GENERATED

*** ERROR 66: OPERATION (X) IS ILLEGAL AFTER AN OPERATION;
SKIP TO COMMA OR END-OF-LINE

*** ERROR 67: (X) WAS NEVER DEFINED; ADDRESS ASSUMED ZERO

*** ERROR 68: "(X)" IS ILLEGAL IN THIS CONTEXT; SKIP REST OF STMT

While the assembler does accept all printing ASCII characters, they are not valid in all contexts. For example:

```
STOO       EQU        ($ + 5)
```

The open parenthesis character is not allowed in this context and provokes this error message. The remainder of the source statement is skipped by the assembler.

*** ERROR 69: INCOMPLETE MEMORY REFERENCE IS ASSUMED TO BE [GA]

*** ERROR 70: INCOMPLETE MEMORY REFERENCE IS ASSUMED TO BE [REGISTER]

*** ERROR 71: INCOMPLETE MEMORY REFERENCE IS ASSUMED TO BE [REGISTER + IX]

*** ERROR 72: INCOMPLETE MEMORY REFERENCE IS ASSUMED TO BE [REGISTER + IX +]

*** ERROR 73: (X) IS ILLEGAL IN A MEMORY REFERENCE; REF BECOMES [GA];
SKIP TO COMMA OR END-OF-LINE

*** ERROR 74: (X) IS ILLEGAL IN A MEMORY REFERENCE; "]" ASSUMED TO PRECEDE IT;
SKIP TO COMMA OR END-OF-LINE

*** ERROR 75: (X) IS ILLEGAL IN A MEMORY REFERENCE AFTER "]";
SKIP TO COMMA OR END OF LINE

*** ERROR 76: (X) IS ILLEGAL IN A MEMORY REFERENCE AFTER "+";
INDEXED REF ASSUMED; SKIP TO COMMA OR END-OF-LINE

*** ERROR 77: (X) IS ILLEGAL IN A MEMORY REFERENCE AFTER "+IX";
"]" ASSUMED TO PRECEDE IT; SKIP TO COMMA OR END-OF-FILE

*** ERROR 78: (X) IS ILLEGAL IN A MEMORY REFERENCE AFTER "+IX+";
"]" ASSUMED TO PRECEDE IT; SKIP TO COMMA OR END-OF-LINE

*** ERROR 79: OPENING "]" ASSUMED TO BE [GA]; SKIP TO COMMA OR END-OF-LINE

*** ERROR 80: "(X) EQU $" IS ASSUMED ((X) IS ALREADY GLOBAL)

Public symbols cannot be equated to a register symbol. For example:

```
PUBLIC    REG

REG       EQU       GA
```

The above EQU statement is assumed by the assembler to be:

```
REG       EQU       $
```

*** ERROR 81: DELIMITER EXPECTED BEFORE (X); SKIP TO COMMA OR END-OF-LINE

A comma or end-of-line sequence is missing before (X). Everything following (X), until the next delimiter, is ignored. A delimiter must be inserted before (X).

*** ERROR 82: OPERAND (X) FAILS IN PASS 2; ZERO USED

Assembler error—contact Intel Corporation.

*** ERROR 83: ZERO INSERTED BEFORE (X)

The assembler turns the sequences ++, +−, −+, and −− into +0+, +0−, −0+, and −0−. This message reports that this has occurred.

*** ERROR 84: MAXIMUM "INCLUDE" NESTING EXCEEDED

Nested INCLUDEs are not allowed by the assembler. For example:

```
SEG89     SEGMENT

          $INCLUDE(:F1:PROG1)

SEG89     ENDS

          END
```

The above included file (PROG1) cannot contain any INCLUDE controls.

*** ERROR 85: PRIMARY CONTROL FOLLOWS A NON-CONTROL STATEMENT

A control line containing a primary control follows a non-control statement. The primary control, and any controls following it in the control line, are ignored. The primary control must be placed before the first non-control line in the source file.

*** ERROR 86: STRUCTURE (X) IS LONGER THAN 64K BYTES

*** ERROR 87: (X) (ILLEGAL IN EXPRESSION) IS ASSUMED TO BE ZERO;
SKIP TO COMMA OR END-OF-LINE

*** ERROR 88: NON-PROGRAMMABLE REGISTER (X) IS ASSUMED TO BE GA

The PP register is non-programmable and can only be used in data memory expressions. This error message is provoked by the following kind of statements:

```
MOVI      PP, 1234H
```

The assembler assumes the above to read MOVI GA, 1234H.

*** ERROR 89: NO OPERAND PRESENT; STATEMENT IGNORED

A DB, DW, DD, DS, NAME, ORG, PUBLIC, or EXTRN directive has no operands. An operand should be added to the source statement or the statement should be deleted.

*** ERROR 90: SOURCE STATEMENT IS TOO LONG; ADDITIONAL CHARACTERS IGNORED

The maximum size of a compressed 8089 Assembly Language source statement is 256 characters. Additional characters are ignored but do appear in the list file.

*** ERROR 91: ILLEGAL USE OF EXTERNAL; VALUE ASSUMED ZERO

This error message is provoked by an external symbol appearing in the operand field of an EQU directive:

```
EXTRN    PARM

CNTRL    EQU      PARM
```

A value of zero is assigned to the symbol CNTRL by the assembler.

*** ERROR 92: EXTERNAL SYMBOL (X) IS ILLEGAL IN THIS CONTEXT; ASSUMED ZERO

An external symbol appears in an expression in a statement other than an LPDI instruction or DD directive. The value of the external symbol is assumed to be zero. For example:

```
EXTRN    SUM

ADDI     GA, SUM + 22
```

The assembler assumes the value of SUM to be zero and generates an immediate value of 22.

*** ERROR 93: ILLEGAL POST-AUTO-INCREMENT IS IGNORED

A CALL instruction cannot have a data memory expression which uses the post auto-increment form. For example:

```
CALL     [GA+IX+], TARGET
```

The data memory expression form [GA+IX+] is not allowed. Another data memory expression form must be used in its place.

*** ERROR 94: FORWARD REFERENCE TO REGISTER SYMBOL (X) IS ASSUMED ZERO

Symbols created as alternate register names are only allowed in the same contexts that the register symbol is allowed in. This error message is provoked by the following kind of statement:

```
         DB       X

    X    EQU      BC
```

The value of X in the DB directive is assumed to be zero.

*** ERROR 95: ILLEGAL OPERAND #(X) IS ASSUMED ZERO

Operand number (X) in a DB, DW, DD, or EQU directive is a data memory expression or a register symbol.

*** ERROR 121: INVALID DIGIT IN CONTROL FIELD

*** ERROR 122: LINE ENDS BEFORE QUOTED STRING IN CONTROL

*** ERROR 123: CONTROL REQUIRES PARENTHESIZED VALUE

*** ERROR 124: CONTROL REQUIRES QUOTED STRING

*** ERROR 125: RIGHT PARENTHESIS EXPECTED

*** ERROR 126: CONTROL STRING IS TOO LONG

*** ERROR 127: CONTROL VALUE IS TOO LARGE

*** ERROR 128: CONTROL VALUE IS TOO SMALL

*** ERROR 129: UNRECOGNIZED CONTROL

*** ERROR 130: CONTROL REQUIRES NUMERIC VALUE

*** ERROR 131: (X) IS USED ILLEGALLY

*** ERROR 151: NAME REQUIRED; STATEMENT IGNORED

*** ERROR 152: LABEL REQUIRED; STATEMENT IGNORED

*** ERROR 153: ILLEGAL OUTSIDE SEGMENT; STATEMENT IGNORED

*** ERROR 154: ILLEGAL INSIDE STRUCTURE; STATEMENT IGNORED

*** ERROR 155: SYMBOL EXPECTED; TWO NO-OPS GENERATED

*** ERROR 156: TOO MANY EXTERNALS; BALANCE IGNORED

A maximum of 32,767 external symbols may be declared in a source file, provided there is sufficient room in the dictionary. Two separate source files must be created if more than 32,767 external symbols are needed.

*** ERROR 157: "ENDS" HAS NO ANTECEDENT; STATEMENT IGNORED

*** ERROR 158: ATTEMPTED 1-BYTE BRANCH TO 2-BYTE TARGET;
TWO NO-OPS GENERATED

The jump target of a TSL instruction is outside the range next instruction −128, next instruction + 127. The jump target must be relocated inside this range.

*** ERROR 159: ILLEGAL COMBINATION OF OPERANDS; TWO NO-OPS GENERATED

*** ERROR 160: "NAME" DOES NOT ALLOW EXPRESSIONS; STATEMENT IGNORED

*** ERROR 161: SEGMENT (X) IS ALREADY DEFINED; STATEMENT IGNORED

*** ERROR 162: "SEGMENT" REQUIRES A NAME; STATEMENT IGNORED

*** ERROR 163: STRUCTURES MAY NOT BE NESTED; STATEMENT IGNORED

*** ERROR 164: UNRECOGNIZED OPERATION (X); STATEMENT IGNORED

*** ERROR 201: FAILURE DURING STATEMENT SCAN (REMAP)

*** ERROR 202: SYNTAX FAILURE AFTER INITIAL EVALUATION

*** ERROR 203: FAILURE DURING OPERAND CLASSIFICATION

*** ERROR 204: POINTER FAILURE IN PASS 2; GA ASSUMED

*** ERROR 205: DESTINATION LOST BETWEEN PASSES; WIDTH ASSUMED 8

*** ERROR 206: ATTEMPT TO SKIP TO NONEXISTENT OPERAND

*** ERROR 207: OPERAND #(X) FAILS IN PASS ONE; STATEMENT IGNORED

*** ERROR 208: (X) WAS PREVIOUSLY MADE A NON-SYMBOL

*** ERROR 209: UNRECOGNIZED CONSTRUCT WHILE EMPTYING META-TEXT

*** ERROR 210: REWRITTEN EXPRESSION FAILURE

*** ERROR 211: META POINTER IS PAST END OF META TEXT

*** ERROR 212: META POINTER IS BEFORE START OF META TEXT

*** ERROR 213: META NOTE OVERFLOW

*** ERROR 214: META NOTE UNDERFLOW

*** ERROR 215: ATTEMPT TO PLANT UNRECOGNIZED META CHARACTER

*** ERROR 216: ATTEMPT TO PLANT UNRECOGNIZED OBJECT CONSTRUCT

*** ERROR 217: UNRECOGNIZED CONSTRUCT WHILE SKIPPING IN META-TEXT

*** ERROR 218: FAILURE OF OPEN/CLOSE QUOTE META

*** ERROR 220: INVALID META FOUND IN INTERMEDIATE TEXT

*** ERROR 221: UNRECOGNIZED TOKEN TYPE; SKIP TO COMMA OR END-OF-LINE

*** ERROR 222: CONTROL FAILURE IN PASS 2

*** ERROR 247: USED ILLEGALLY

*** ERROR 248: CONTROL IS INVALID IN COMMAND TAIL

*** ERROR 249: INVOCATION DOES NOT END WITH <CR><LF>

*** ERROR 250: INVOCATION LINE IS TOO LONG

*** ERROR 251: INPUT MUST BE FROM A RANDOM-ACCESS FILE

*** ERROR 252: TYPE <n>: <concise message for ISIS error <n> >

*** ERROR 253: LENGTH ERROR ON READ

*** ERROR 254: NOT ENOUGH SPACE FOR ERROR CONSTRUCTS

*** ERROR 255: PASS FAILURE DURING STATEMENT ABANDON

*** ERROR <m>: INTERNAL PROCESSING ERROR

Assembler failure—contact Intel Corporation.

*** ERROR <n>: UNKNOWN ERROR TYPE

Assembler failure—contact Intel Corporation.

APPENDIX K
8089 INSTRUCTIONS IN HEXADECIMAL ORDER

Each 8089 instruction generates a minimum of two bytes of object code. The following lists the hexadecimal values for the second assembled instruction byte, containing the operation code and the base memory address fields.

A "B" appearing in brackets in an instruction mnemonic is coded for the byte form of the instruction.

For example:

20H is generated by both ADDI R, I and ADDBI R, I. An "L" appearing in brackets in a control transfer instruction mnemonic is coded for the long form of the instruction.

For example:

40H is generated by both JNZ R, L and L-JNZ R, L.

See Chapter 3 for the format of the first assembled instruction byte.

HEX	BINARY	INSTRUCTION	BASE ADDRESS
00	00000000	NOP	
00	00000000	SINTR	
00	00000000	WID S, D	
00	00000000	XFER	
01	00000001		
02	00000010		
03	00000011		
04	00000100		
05	00000101		
06	00000110		
07	00000111		
08	00001000	LPDI P, I	
09	00001001		
0A	00001010		
0B	00001011		
0C	00001100		
0D	00001101		
0E	00001110		
0F	00001111		
10	00010000		
11	00010001		
12	00010010		
13	00010011		
14	00010100		
15	00010101		
16	00010110		
17	00010111		
18	00011000		
19	00011001		
1A	00011010		
1B	00011011		
1C	00011100		
1D	00011101		
1E	00011110		
1F	00011111		
20	00100000	ADD[B]I R, I	
20	00100000	[L]JMP L	
21	00100001		
22	00100010		

HEX	BINARY	INSTRUCTION	BASE ADDRESS
23	00100011		
24	00100100	OR[B]I R, I	
25	00100101		
26	00100110		
27	00100111		
28	00101000	AND[B]I R, I	
29	00101001		
2A	00101010		
2B	00101011		
2C	00101100	NOT R	
2D	00101101		
2E	00101110		
2F	00101111		
30	00110000	MOV[B]I R, I	
31	00110001		
32	00110010		
33	00110011		
34	00110100		
35	00110101		
36	00110110		
37	00110111		
38	00111000	INC R	
39	00111001		
3A	00111010		
3B	00111011		
3C	00111100	DEC R	
3D	00111101		
3E	00111110		
3F	00111111		
40	01000000	[L]JNZ R, L	
41	01000001		
42	01000010		
43	01000011		
44	01000100	[L]JZ R, L	
45	01000101		
46	01000110		
47	01000111		
48	01001000	HLT	
49	01001001		
4A	01001010		
4B	01001011		
4C	01001100	MOV[B]I M, I	GA
4D	01001101	MOV[B]I M, I	GB
4E	01001110	MOV[B]I M, I	GC
4F	01001111	MOV[B]I M, I	PP
50	01010000		
51	01010001		
52	01010010		
53	01010011		
54	01010100		
55	01010101		
56	01010110		
57	01010111		
58	01011000		
59	01011001		
5A	01011010		
5B	01011011		
5C	01011100		
5D	01011101		
5E	01011110		
5F	01011111		
60	01100000		
61	01100001		
62	01100010		
63	01100011		
64	01100100		
65	01100101		
66	01100110		
67	01100111		
68	01101000		

HEX	BINARY	INSTRUCTION	BASE ADDRESS
69	01101001		
6A	01101010		
6B	01101011		
6C	01101100		
6D	01101101		
6E	01101110		
6F	01101111		
70	01110000		
71	01110001		
72	01110010		
73	01110011		
74	01110100		
75	01110101		
76	01110110		
77	01110111		
78	01111000		
79	01111001		
7A	01111010		
7B	01111011		
7C	01111100		
7D	01111101		
7E	01111110		
7F	01111111		
80	10000000	MOV[B] R, M	GA
81	10000001	MOV[B] R, M	GB
82	10000010	MOV[B] R, M	GC
83	10000011	MOV[B] R, M	PP
84	10000100	MOV[B] M, R	GA
85	10000101	MOV[B] M, R	GB
86	10000110	MOV[B] M, R	GC
87	10000111	MOV[B] M, R	PP
88	10001000	LPD P, M	GA
89	10001001	LPD P, M	GB
8A	10001010	LPD P, M	GC
8B	10001011	LPD P, M	PP
8C	10001100	MOVP P, M	GA
8D	10001101	MOVP P, M	GB
8E	10001110	MOVP P, M	GC
8F	10001111	MOVP P, M	PP
90	10010000	MOV[B] M, M	GA
91	10010001	MOV[B] M, M	GB
92	10010010	MOV[B] M, M	GC
93	10010011	MOV[B] M, M	PP
94	10010100	TSL M, I, L	GA
95	10010101	TSL M, I, L	GB
96	10010110	TSL M, I, L	GC
97	10010111	TSL M, I, L	PP
98	10011000	MOVP M, P	GA
99	10011001	MOVP M, P	GB
9A	10011010	MOVP M, P	GC
9B	10011011	MOVP M, P	PP
9C	10011100	[L]CALL M, L	GA
9D	10011101	[L]CALL M, L	GB
9E	10011110	[L]CALL M, L	GC
9F	10011111	[L]CALL M, L	PP
A0	10100000	ADD[B] R, M	GA
A1	10100001	ADD[B] R, M	GB
A2	10100010	ADD[B] R, M	GC
A3	10100011	ADD[B] R, M	PP
A4	10100100	OR[B] R, M	GA
A5	10100101	OR[B] R, M	GB
A6	10100110	OR[B] R, M	GC
A7	10100111	OR[B] R, M	PP
A8	10101000	AND[B] R, M	GA
A9	10101001	AND[B] R, M	GB
AA	10101010	AND[B] R, M	GC
AB	10101011	AND[B] R, M	PP
AC	10101100	NOT[B] R, M	GA
AD	10101101	NOT[B] R, M	GB
AE	10101110	NOT[B] R, M	GC

HEX	BINARY	INSTRUCTION	BASE ADDRESS
AF	10101111	NOT[B] R, M	PP
B0	10110000	[L]JMCE M, L	GA
B1	10110001	[L]JMCE M, L	GB
B2	10110010	[L]JMCE M, L	GC
B3	10110011	[L]JMCE M, L	PP
B4	10110100	[L]JMCNE M, L	GA
B5	10110101	[L]JMCNE M, L	GB
B6	10110110	[L]JMCNE M, L	GC
B7	10110111	[L]JMCNE M, L	PP
B8	10111000	[L]JNBT M, b, L	GA
B9	10111001	[L]JNBT M, b, L	GB
BA	10111010	[L]JNBT M, b, L	GC
BB	10111011	[L]JNBT M, b, L	PP
BC	10111100	[L] JBT M, b, L	GA
BD	10111101	[L] JBT M, b, L	GB
BE	10111110	[L] JBT M, b, L	GC
BF	10111111	[L] JBT M, b, L	PP
C0	11000000	ADD[B]I M, I	GA
C1	11000001	ADD[B]I M, I	GB
C2	11000010	ADD[B]I M, I	GC
C3	11000011	ADD[B]I M, I	PP
C4	11000100	OR[B]I M, I	GA
C5	11000101	OR[B]I M, I	GB
C6	11000110	OR[B]I M, I	GC
C7	11000111	OR[B]I M, I	PP
C8	11001000	AND[B]I M, I	GA
C9	11001001	AND[B]I M, I	GB
CA	11001010	AND[B]I M, I	GC
CB	11001011	AND[B]I M, I	PP
CC	11001100		
CD	11001101		
CE	11001110		
CF	11001111		
D0	11010000	ADD[B] M, R	GA
D1	11010001	ADD[B] M, R	GB
D2	11010010	ADD[B] M, R	GC
D3	11010011	ADD[B] M, R	PP
D4	11010100	OR[B] M, R	GA
D5	11010101	OR[B] M, R	GB
D6	11010110	OR[B] M, R	GC
D7	11010111	OR[B] M, R	PP
D8	11011000	AND[B] M, R	GA
D9	11011001	AND[B] M, R	GB
DA	11011010	AND[B] M, R	GC
DB	11011011	AND[B] M, R	PP
DC	11011100	NOT[B] M	GA
DD	11011101	NOT[B] M	GB
DE	11011110	NOT[B] M	GC
DF	11011111	NOT[B] M	PP
E0	11100000	[L]JNZ[B] M, L	GA
E1	11100001	[L]JNZ[B] M, L	GB
E2	11100010	[L]JNZ[B] M, L	GC
E3	11100011	[L]JNZ[B] M, L	PP
E4	11100100	[L]JZ[B] M, L	GA
E5	11100101	[L]JZ[B] M, L	GB
E6	11100110	[L]JZ[B] M, L	GC
E7	11100111	[L]JZ[B] M, L	PP
E8	11101000	INC[B] M	GA
E9	11101001	INC[B] M	GB
EA	11101010	INC[B] M	GC
EB	11101011	INC[B] M	PP
EC	11101100	DEC[B] M	GA
ED	11101101	DEC[B] M	GB
EE	11101110	DEC[B] M	GC
EF	11101111	DEC[B] M	PP
F0	11110000		
F1	11110001		
F2	11110010		
F3	11110011		
F4	11110100	SETB M, b	GA

HEX	BINARY	INSTRUCTION	BASE ADDRESS
F5	11110101	SETB M, b	GB
F6	11110110	SETB M, b	GC
F7	11110111	SETB M, b	PP
F8	11111000	CLR M, b	GA
F9	11111001	CLR M, b	GB
FA	11111010	CLR M, b	GC
FB	11111011	CLR M, b	PP
FC	11111100		
FD	11111101		
FE	11111110		
FF	11111111		

INDEX

The entries in this index are shown as they appear in the text of the book, i.e., lowercase words are lowercase in the text, uppercase words are uppercase in the text. When more than one reference is given for an entry, the primary reference is listed first.

intel®

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. ______

NAME ______ DATE ______

TITLE ______

COMPANY NAME/DEPARTMENT ______

ADDRESS ______

CITY ______ STATE ______ ZIP CODE ______

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.