



Development Solutions

MCS[®]-96

Macro Assembler User's Guide
for DOS Systems

MCS[®]-96 Macro Assembler User's Guide for DOS Systems

MCS[®]-96 MACRO ASSEMBLER USER'S GUIDE FOR DOS SYSTEMS

Order Number: 122350-003

REV.	REVISION HISTORY	DATE
-001	Original Issue.	7/85
-002	Updated Typeset Version.	5/88
-003	Updated to cover 80C196KC microcontroller, including information on six new 80C196KC-specific instructions, 80C196 interrupt processing, and the 80C196KC peripheral transaction server. Removed references to obsolete ERRORSUMMARY control and to 80C196KA microcontroller.	5/90

CAUTION

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. As temporarily permitted by regulation, it has not been tested for compliance with the limits for Class A Computing Devices pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user, at his own expense, will be required to take whatever measures may be required to correct the interference.

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

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

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

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

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

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ®.)

Above	i860	int _e lBOS	MAPNET	RMX/80
ACE51	^	Intel Certified	MCS®	RUPI
ACE96	i®	InteleVision	Megachassis	Seamless
ACE186	I2ICE	int _e l _i gent Identifier	MICROMAINFRAME	SLD
ACE196	ICE	int _e l _i gent Programming	MULTI CHANNEL	SugarCube
ACE960	iCEL	Intellec®	MULTIMODULE	Tech Direct
Action Media	ICEVIEW	Intellink	MultiSERVER	ToolTALK
BITBUS	iCS	iOSP	ONCE	UPI
COMMPuter	iDBP	iPAT	OpenNET	Visual Edge
CREDIT	iDIS	iPDS	OTP	VLSiCEL
Data Pipeline	iLBX	iPSC®	PRO750	ZapCode
DVI	iMDDX	iRMK	PROMPT	376
ETOX	iMMX	iRMX®	Promware	386
FaxBACK	Inboard	iSBC®	QUEST	387
Genius	Insite	iSBX	QueX	4-SITE
i486	Intel®	iSDM	Quick-Erase	486
i586	int _e l®	iSXM	Quick-Pulse Programming	586
i686	Intel386	Library Manager	Ripplemode	686
i750				

IBM, PC AT, Personal System/2, and PS/2 are registered trademarks and PC XT is a trademark of International Business Machines Corporation.

Copyright © 1985, 1988, 1990, Intel Corporation, All Rights Reserved

Notational Conventions

The following notational conventions are used throughout this manual:

UPPERCASE	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.
<i>italic</i>	Italic indicates a metasymbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
<i>directory-name</i>	Is that portion of a <i>pathname</i> that acts as a file locator by identifying the device and/or directory containing the <i>filename</i> .
<i>filename</i>	Is a valid name for the part of a <i>pathname</i> that names a file.
<i>pathname</i>	Is a valid designation for a file; in its entirety, it consists of a <i>directory</i> and a <i>filename</i> .
<i>pathname1</i> , <i>pathname2</i> , ...	Are generic labels used in examples where one or more user-specified <i>pathnames</i> would actually be printed.
<i>system-id</i>	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
Vx.y	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ } . . .	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.

	The vertical bar separates options within brackets [] or braces { }.
. . .	Ellipses indicate that the preceding argument or parameter may be repeated.
[, . . .]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered.
	<pre>SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')</pre>
<cr>	Indicates a carriage return.
MONOSPACE TYPE	Type of this style represents syntax, filenames, program examples, or computer output. Bold monospace type preceded by a prompt represents user input.

Contents



Preface	xv
---------------	----

Chapter 1 Introduction

1.1	What Is an Assembler?	1-1
1.2	How to Develop a Program	1-2
1.2.1	The Advantages of Modular Programming	1-2
1.2.2	Efficient Program Development	1-2
1.2.3	Multiple Use of Subprograms	1-3
1.2.4	Ease of Debugging and Modifying	1-3
1.3	MCS [®] -96 Modular Program Development Process	1-3
1.3.1	Segments, Modules, and Programs	1-3
1.3.2	Program Entry and Edit	1-4
1.3.3	Assembly	1-4
1.3.3.1	Object File	1-5
1.3.3.2	Listing File	1-5
1.3.4	Relocation and Linkage	1-6
1.3.5	Conversion to Hexadecimal Format	1-6
1.3.6	Keeping Track of Files	1-6
1.4	Writing, Assembling, and Debugging an MCS [®] -96 Program	1-7
1.5	Hardware Overview	1-10
1.5.1	Memory Space	1-15
1.5.2	High-speed Input/Output Unit	1-16
1.5.2.1	Timers	1-19
1.5.2.2	High-speed Inputs	1-19
1.5.2.3	High-speed Outputs	1-20
1.5.2.4	Software Timers	1-22
1.5.3	I/O Ports	1-22
1.5.4	Serial Port	1-24
1.5.5	Pulse Width Modulator	1-24
1.5.6	A/D Converter	1-25
1.5.7	Interrupts	1-25

1.5.8	Watchdog Timer	1-32
1.5.9	Reset	1-32
1.5.10	Special Function Registers	1-32
1.5.10.1	PWM Control Register	1-32
1.5.10.2	Serial Port Control/Status Register	1-33
1.5.10.3	A/D Command Register	1-33
1.5.10.4	A/D Result Register	1-34
1.5.10.5	Interrupt Registers	1-34
1.5.10.6	HSI Mode Register	1-34
1.5.10.7	HSI Status Register	1-35
1.5.10.8	HSO Command Tag	1-35
1.5.10.9	I/O Control Registers	1-37
1.5.10.10	HSIO Status Registers	1-39
1.5.11	Program Status Word	1-40

Chapter 2 Assembly Language Concepts

2.1	Assembly Language Components	2-1
2.1.1	Character Set	2-2
2.1.2	Numbers	2-2
2.1.3	Long Constants	2-2
2.1.4	Real Numbers	2-3
2.1.5	Delimiters	2-6
2.1.6	Symbols	2-7
2.1.7	Reserved Words	2-7
2.1.8	Assembler-generated Symbols	2-8
2.1.9	Location Counter	2-8
2.1.10	Strings	2-8
2.2	Expressions and Basic Operands	2-9
2.2.1	Basic Operands	2-10
2.2.2	Attributes of Expression Operands	2-10
2.2.2.1	Type	2-11
2.2.2.2	Relocatability	2-12
2.2.2.3	Value	2-12
2.2.3	Absolute Expressions	2-12
2.2.4	Relocatable Expressions	2-14
2.3	Statement Format	2-14
2.3.1	Additional Statement Rules	2-16
2.4	Program Format	2-17

2.5	Segments	2-18
2.5.1	Register Segment (Overlayable and Non-overlayable)	2-18
2.5.2	Data Segment	2-19
2.5.3	Stack Segment	2-19
2.5.4	Code Segment	2-19
2.6	Absolute and Relocatable Segments	2-20

Chapter 3 Instruction Set

3.1	Introduction	3-1
3.2	Program Status Word	3-3
3.3	Instruction Operands	3-4
3.3.1	Referring to a Register: breg, wreg, lreg	3-4
3.3.2	Referring to a General Memory Location: baop, waop	3-4
3.3.3	Direct Addressing	3-6
3.3.4	Immediate Addressing	3-7
3.3.5	Indirect Addressing	3-7
3.3.6	Indirect Addressing with Auto Increment	3-8
3.3.7	Based or Indexed Addressing	3-9
3.3.8	Referring to a Code Address: cadd	3-10
3.3.9	Specifying a Count for a Shift: cnt/breg	3-10
3.3.10	Specifying a Bit Operand: bitno	3-11
3.4	Description of Instructions	3-11
	ADD (Two Operands)	3-12
	ADD (Three Operands)	3-13
	ADDB (Two Operands)	3-14
	ADDB (Three Operands)	3-15
	ADDC	3-16
	ADDCB	3-17
	AND (Two Operands)	3-18
	AND (Three Operands)	3-19
	ANDB (Two Operands)	3-20
	ANDB (Three Operands)	3-21
	BBC	3-22
	BBS	3-23
	BC	3-24
	BE	3-25
	BGE	3-26
	BGT	3-27
	BH	3-28

BLE	3-29
BLT	3-30
BMOV	3-31
BMOVI	3-33
BNC	3-34
BNE	3-35
BNH	3-36
BNST	3-37
BNV	3-38
BNVT	3-39
BR (Direct)	3-40
BR (Indirect)	3-41
BR (Direct)	3-42
BV	3-43
BVT	3-44
CALL (Direct)	3-45
CLR	3-46
CLRB	3-47
CLRC	3-48
CLRVT	3-49
CMP	3-50
CMPL	3-52
DBNZ	3-53
DBNZW	3-55
DEC	3-57
DECB	3-58
DI	3-59
DIV	3-60
DIVB	3-61
DIVU	3-62
DIVUB	3-63
DJNZ	3-64
DJNZW	3-65
DPTS	3-66
EI	3-67
EPTS	3-68
EXT	3-69
EXTB	3-70
IDLPD	3-71
INC	3-72

INCB	3-73
JBC	3-74
JBS	3-75
JC	3-76
JE	3-77
JGE	3-78
JGT	3-79
JH	3-80
JLE	3-81
JLT	3-82
JNC	3-83
JNE	3-84
JNH	3-85
JNST	3-86
JNV	3-87
JNVT	3-88
JST	3-89
JV	3-90
JVT	3-91
LCALL	3-92
LD	3-93
LDB	3-94
LDBSE	3-95
LDBZE	3-96
LJMP	3-97
MUL (Two Operands)	3-98
MUL (Three Operands)	3-99
MULB (Two Operands)	3-100
MULB (Three Operands)	3-101
MULU (Two Operands)	3-102
MULU (Three Operands)	3-103
MULUB (Two Operands)	3-104
MULUB (Three Operands)	3-105
NEG	3-106
NEGB	3-107
NOP	3-108
NORML	3-109
NOT	3-110
NOTB	3-111
OR	3-112

ORB	3-113
POP	3-114
POPA	3-115
POPF	3-116
PUSH	3-117
PUSHA	3-118
PUSHF	3-119
RET	3-120
RST	3-121
SCALL	3-122
SETC	3-123
SHL	3-124
SHLB	3-125
SHLL	3-126
SHR	3-127
SHRA	3-128
SHRAB	3-129
SHRAL	3-130
SHRB	3-131
SHRL	3-132
SJMP	3-133
SKIP	3-134
ST	3-135
STB	3-136
SUB (Two Operands)	3-137
SUB (Three Operands)	3-138
SUBB (Two Operands)	3-139
SUBB (Three Operands)	3-140
SUBC	3-141
SUBCB	3-142
TIJMP	3-143
XCH	3-144
XCHB	3-145
XOR	3-146
XORB	3-147

Chapter 4 Assembler Directives

4.1	Module Level	4-3
4.1.1	MODULE	4-3
4.1.2	PUBLIC	4-4
4.1.3	EXTERNAL	4-4
4.1.4	END	4-5
4.2	Location Counter Control	4-5
4.2.1	Segment Selection (CSEG, DSEG, RSEG, and OSEG)	4-5
4.2.2	ORG	4-7
4.3	Symbol Definition	4-8
4.3.1	EQU and SET	4-8
4.4	Code Definition	4-9
4.4.1	DCB	4-9
4.4.2	DCW	4-10
4.4.3	DCL	4-11
4.4.4	DCR	4-11
4.5	Storage Reservation	4-12
4.5.1	DSB, DSW, DSL, and DSR	4-12
4.6	Conditional Assembly	4-13
4.6.1	IF, ELSE, and ENDIF	4-13

Chapter 5 The Macro Processing Language

5.1	Introduction	5-1
5.1.1	The Advantages of Using Macros	5-1
5.1.2	An Example of Macro Use	5-2
5.1.3	Macros and Subroutines	5-4
5.2	Macro Directives and Macro Calls	5-4
5.2.1	Macro Definition	5-5
5.3	MACRO Directives	5-6
5.3.1	MACRO	5-6
5.3.2	ENDM	5-7
5.3.3	LOCAL	5-7
5.3.4	REPT	5-9
5.3.5	IRP	5-10
5.3.6	IRPC	5-11
5.3.7	EXITM	5-12
5.4	Special Macro Operators	5-13
5.5	Nesting Macro Definitions	5-13

5.6	Macro Calls	5-15
5.6.1	Macro Call	5-15
5.6.2	Nested Macro Calls	5-17
5.7	Macro Expansion	5-18
5.8	Null Macros	5-20
5.9	Sample Macros	5-21

Chapter 6 Assembler Operation and Controls

6.1	How to Invoke the MCS®-96 Macro Assembler	6-1
6.2	Specifying Assembler Controls	6-2
6.2.1	Types of Controls	6-5
6.3	Control Processing	6-5
6.4	Description of Controls	6-6
	COND/NOCOND	6-7
	DEBUG/NODEBUG	6-8
	EJECT	6-9
	ERRORPRINT/NOERRORPRINT	6-10
	GEN/NOGEN	6-11
	INCLUDE	6-12
	LIST/NOLIST	6-13
	OBJECT/NOOBJECT	6-14
	PAGELength	6-15
	PAGEWIDTH	6-16
	PRINT/NOPRINT	6-17
	SAVE/RESTORE	6-18
	SYMBOLS/NOSYMBOLS	6-19
	TITLE	6-20
	XREF/NOXREF	6-21

Chapter 7 Assembler Output: Error Messages and Listing File Format

7.1	Console Output	7-1
7.1.1	Sign-on Message	7-1
7.2	Error Messages and Recovery	7-1

7.2.1	Console Error Messages	7-2
7.2.1.1	Invocation Line Error Messages	7-2
7.2.1.2	I/O Error Messages	7-3
7.2.1.3	Insufficient (Host) Memory Error Messages	7-3
7.2.1.4	Internal Synchronization Error Messages	7-3
7.2.2	Sign-off Message	7-4
7.3	Listing File	7-4
7.3.1	Header	7-8
7.3.2	Additional Introductory Lines	7-9
7.3.3	Body	7-9
7.3.3.1	Error Lines	7-11
7.3.4	The Symbol Table	7-11
7.4	The Errorprint File	7-15

Appendix A Machine Instruction Summary

Appendix B Assembler Directives

Appendix C Assembler Controls

Appendix D ASM96 Reserved Words

Appendix E Error Messages

E.1	Console Error Messages	E-1
E.2	Source File Error Messages	E-2

Index

Service Information	Inside Back Cover
---------------------------	-------------------

Figures

1-1	Assembler and Linker/Relocator Outputs	1-5
1-2	MCS®-96 Program Development Process	1-8
1-3	Sample Program Listing	1-9
1-4	Block Diagram of 8096 Microcontroller	1-13
1-5	Block Diagram of 80C196KB Microcontroller	1-14

1-6	Block Diagram of 80C196-KC Microcontroller	1-15
1-7	MCS®-96 Memory Map	1-17
1-8	MCS®-96 Special Function Registers	1-18
1-9	High-speed Output Unit	1-21
1-10	8096 Interrupt Structure	1-26
1-11	80C196 Interrupt Structure	1-29
1-12	PTS Interrupt Response	1-31
2-1	Typical Assembly-unit Program Structure	2-18
7-1	Source Listing Example	7-5
7-2	Symbol Table Example	7-12

Tables

1-1	8096 Microcontrollers	1-10
1-2	8x9y Proliferations	1-11
1-3	80C196 Microcontrollers	1-11
1-4	MCS®-96 State Times	1-13
1-5	Port 2 Pin Functionality	1-23
1-6	PWM and Controlling Pins	1-24
1-7	8096 Interrupt Vector and Priorities	1-27
1-8	8096 Interrupt Enable Bits	1-28
1-9	80C196 Additional Interrupt Vector and Priorities	1-28
1-10	80C196 Interrupt Enable Bits	1-30
2-1	ASM-96 Numbers and Bases	2-3
2-2	ASM96 Delimiters	2-6
2-3	Valid Operators for ASM96 Absolute Expressions Listed by Precedence (High to Low)	2-13
3-1	Abbreviations and Notations Used in Instruction Pages	3-1
3-2	Program Status Word Flags	3-3
5-1	Macro Operators	5-14
6-1	ASM96 Macro Assembler Controls	6-4
A-1	Instructions in Alphabetical Order	A-1
A-2	Instructions in Hexadecimal Order	A-5
A-3	80C196 Instruction Cycle Times	A-10
A-4	8096 Instruction Cycle Times	A-14
C-1	Assembler Controls	C-2

Preface



This manual describes how to program the MCS[®]-96 single-chip microcomputers in assembly language. It also describes the operating instructions for the ASM96 assembler that is used for MCS-96. The term MCS-96 refers to an entire family of single-chip microcomputers, all of which have the same basic processor design.

Throughout this manual when we wish not only to refer to a specific chip, but also to point out something that applies to the entire family, we speak of the 8096, 80C196, or MCS-96 processor. Except where otherwise specified, information in this manual applies to both 8096 and 80C196 processors.

The book is intended as a reference, but it contains some instructional material as well. It is organized as follows:

- Chapter 1, "Introduction", describes assembly language programming and provides an overview of the MCS-96 hardware.
- Chapter 2, "Assembly Language Concepts", describes the MCS-96 character set, numbers, delimiters, symbols, reserved words, basic operands, operations, expressions (relocatable and absolute) and segments.
- Chapter 3, "Instruction Set", completely describes the operation of each instruction, listed in alphabetic order.
- Chapter 4, "Assembler Directives", describes how to specify auxiliary information (such as storage reservation, location counter control, definition of non-executable code, object code relocation, and flow of assembler processing).
- Chapter 5, "Macro Processing Language", defines and describes the use of Macro Processing Language.

- Chapter 6, "Assembler Operation and Controls", describes how to invoke the assembler and how to control its operation through the assembler controls.
- Chapter 7, "Assembler Output: Error Messages and Listing File Format", describes how to interpret error messages and the listing file.

Related Literature

The following manuals are of interest to users of the MCS-96 Macro Assembler on DOS:

- *MCS®-96 Utilities User's Guide for DOS Systems*, order number 122355
- *The 8096 Floating-Point Library for DOS Systems*, order number 122365
- *iC-96 Compiler User's Guide for DOS Systems*, order numbers 481194 and 481195
- *PL/M-96 User's Guide for DOS Systems*, order number 122360
- *80C196KC User's Guide*, order number 270704 (This publication is a data sheet for the 80C196-KC microcontroller.)
- *Automotive Products Handbook*, order number 231792
- *The Embedded Control Applications Handbook*, order number 270648
- *The 16-bit 8096: Programming, Interfacing, Applications*, order number 270646
- *ACE-196*, order number 270779 (This is an on-line tutorial for programming the 80C196 microcontrollers.)
- *16-bit Embedded Controllers*, order number 270646
- *ICE-196PC In-Circuit Emulator User's Guide*, order number 452016

This manual describes the MCS®-96 Macro Assembler (ASM96) and explains the process of developing software in assembly language for the MCS-96 family of processors.

Assembly language programs translate language operation codes directly into machine instructions that instruct the processor as to what operation it should perform. Therefore, the assembly language programmer should be familiar with both the microcomputer architecture and with the assembly language. This chapter presents an overview of ASM96 and how it is used, as well as a brief description of the processor architecture and hardware features.

1.1 What Is an Assembler?

An assembler is a software tool (a program) designed to simplify the task of writing computer programs. It performs the clerical task of translating symbolic code into executable object code. This object code can then be programmed into one of the MCS-96 processors and executed.

Assembly language operation codes (mnemonics) are assigned to each machine code instruction. The mnemonics are selected so that they are more meaningful and thus more easily remembered than the machine code equivalent. Thus, the mnemonic `ST` is used for the store instruction and the mnemonic `ADD` for the addition instruction. Within the operand field of an instruction, you can also refer to addresses and values by names you have assigned to them rather than by their absolute addresses. Assigning meaningful names to addresses and values help you (and others working with your programs) to understand them. For example, if your program must manipulate a date as data, you can assign the symbolic name `DATE` to that data variable. Likewise, if the program executes a given instruction set until a specified time has elapsed, you can assign that section of your program the symbolic name `TIMER_LOOP`.

Any assembly language program consists of three parts:

- machine instructions
- assembler directives
- assembler controls

A machine instruction is a machine code that can be executed by the machine. Each instruction is explained in detail in Chapter 3.

Assembler directives are used to define the program structure and symbols, and to generate non-executable code; for example, data or messages. Each directive is explained in Chapter 4.

Assembler controls set the mode of operation for the assembler and direct the flow of the assembly process. The controls are described in Chapter 6.

1.2 How to Develop a Program

ASM96, the assembly language used for MCS-96 microcontrollers, enables you to program in a modular fashion. The following paragraphs explain the basics of modular program development.

1.2.1 The Advantages of Modular Programming

Many programs are too long or too complex to write as a single unit. Programming becomes much simpler when the overall program is divided into smaller functional units. Modular programs are usually easier to code, to debug, and to change than are monolithic programs.

1.2.2 Efficient Program Development

Programs can be developed more quickly using the modular approach because small subprograms are easier to understand, to design, and to test than are large programs. With the module inputs and outputs defined, the programmer can supply the needed input and verify the correctness of the module by examining the output. The separate modules can then be linked and located into one program module. Finally, the completed module can be tested.

1.2.3 Multiple Use of Subprograms

Code written for one program can often be used in other programs. Modular programming allows such sections to be easily saved and then transferred at a future date. Because the code is relocatable, saved modules can be linked to any program that fulfills the input and output requirements of the given module. In monolithic programs, on the other hand, a given section of code is buried deep within the program and cannot easily be exhumed and transferred to other programs.

1.2.4 Ease of Debugging and Modifying

Modular programs are generally easier to debug than are monolithic programs. Because of the well-defined module interfaces of the program, problems can be isolated to a specific module. Once the faulty module has been identified, fixing the problem becomes significantly simpler. When a modular program must be modified, either a specific module can be modified or a new module can be added. New or debugged modules can easily be linked to the existing program. This is not the case for monolithic programs.

1.3 MCS[®]-96 Modular Program Development Process

This section provides a brief discussion of the program development process with the ASM96 assembler, RL96 utilities, and code conversion programs.

1.3.1 Segments, Modules, and Programs

In the initial design stages, the tasks to be performed by the program are first defined and then partitioned into subprograms. The kinds of subprograms used with ASM96 and RL96 are segments, modules, and programs.

A segment is a block of code or data memory. A segment can be relocatable or absolute, has a type (for example, CODE or DATA) and has other attributes.

A module contains one or more segments. It cannot contain more than one relocatable segment of each type. A module has a name assigned by the user. The module definitions determine the scope of local symbols. An object file contains one or more modules. To create files containing more than one module, use LIB96. Refer to the *MCS®-96 Utilities User's Guide*, listed in the Preface, for further information on LIB96.

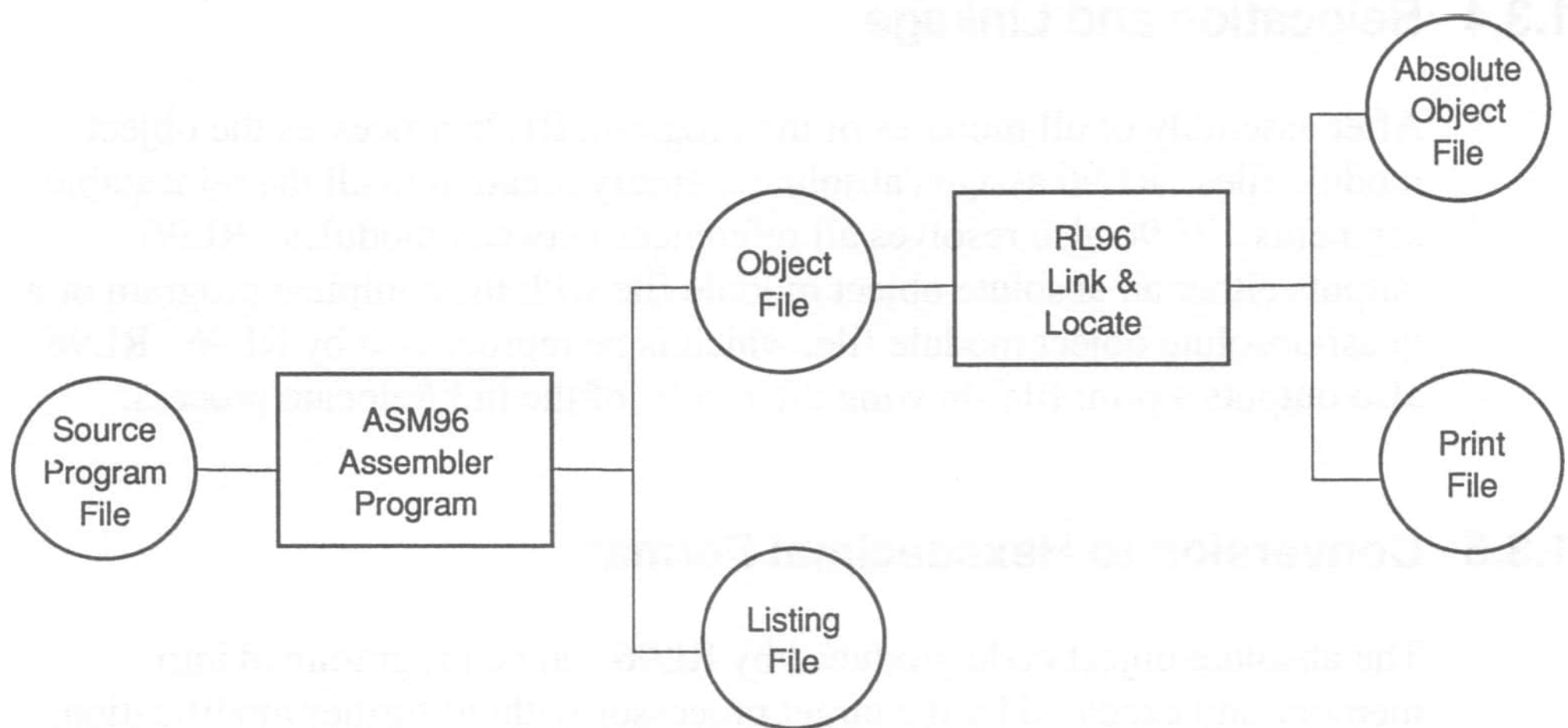
A program consists of a single absolute module, merging all absolute and relocatable segments from all input modules.

1.3.2 Program Entry and Edit

After the design is completed, the source code for each module is entered into a disk file using a text editor. When errors are detected in the development process, the text editor can be used to make corrections in the source code.

1.3.3 Assembly

The ASM96 assembler translates the source code into object code. The assembler produces an object file (relocatable when at least one input segment is relocatable; if not, then absolute), and a listing file showing the results of the assembly. Figure 1-1 summarizes the assembly and the link and relocate outputs. When the ASM96 invocation contains the DEBUG control, the object file also contains the symbol table and other debug information for use in symbolic debugging of the program.



OSD522

Figure 1-1 Assembler and Linker/Relocator Outputs

1.3.3.1 Object File

The object file contains machine language instructions and data that can be loaded into memory for execution or interpretation. In addition, it contains control information governing the loading process.

The assembler can produce object files in relocatable object code format. However, if the module contains only absolute segments and no external references, the object file resulting from assembly is absolute. It can be loaded without the need of the RL96 pass.

1.3.3.2 Listing File

The listing file provides a permanent record of both the source program and the object code. The assembler also provides diagnostic messages in the listing file for syntax and other coding errors. For example, if you specify a 16-bit value for an instruction that can only use an 8-bit value, the assembler tells you that the value exceeds the permissible range. Chapter 7 of this manual describes the format of the listing file. Unless you specify otherwise, the listing file includes the symbol table, which lists all the symbols and their attributes.

1.3.4 Relocation and Linkage

After assembly of all modules of the program, RL96 processes the object module files. RL96 assigns absolute memory location to all the relocatable segments. RL96 also resolves all references between modules. RL96 outputs either an absolute object module file with the complete program or a quasi-absolute object module file, which is be reprocessed by RL96. RL96 also outputs a print file showing the results of the link/relocate process.

1.3.5 Conversion to Hexadecimal Format

The absolute object code produced by RL96 can be programmed into memory and executed by the target processor without further modification. However, certain MCS-96 development products require the hexadecimal object code format. For use with these products, the absolute object file must be processed by the code conversion program OH. Refer to the *MCS[®]-96 Utilities User's Guide*, Chapter 6, for instructions on how to use OH.

1.3.6 Keeping Track of Files

It is convenient to use the extensions of filenames to indicate the stage in the process represented by the contents of each file. Thus, source code files can be marked by extensions such as .SRC or .A96, indicating that the file contains source text or is input for ASM96, respectively. Object code files automatically receive the extension .OBJ unless you explicitly specify some other extension (any combination of valid characters is allowed). Executable files generally have no extension. Listing files are automatically given the extension .LST by the assembler unless you explicitly designate otherwise. RL96 uses .M96 for the default print file extension.

Use caution with the extension .TMP, as some operating system utilities create temporary files with this extension. If you have files with the same name and extension as these, your files are overwritten when the utility creates its temporary file.

1.4 Writing, Assembling, and Debugging an MCS[®]-96 Program

Several steps are necessary to incorporate an MCS-96 microcomputer in your application. The flow chart in Figure 1-2 shows the steps involved in preparing the code. If you are developing hardware for your application in addition to the software, consult the *Embedded Control Applications Handbook* or the *Automotive Products Handbook*, listed in the Preface.

Figure 1-3 shows an assembly listing of a sample program. The assembler was invoked as follows:

```
- ASM96 SAMPLE.A96
system id MCS-96 MACRO ASSEMBLER Vx.y
ASSEMBLY COMPLETE, NO ERROR(S) FOUND.
```

Figure 1-3 shows the listing file of a simple module that is part of a larger program not shown here. A larger example is provided in Chapter 7.

The next step after the program is assembled by ASM96 is to combine all modules into one program using RL96. RL96 produces a print file consisting of a segment map and a symbol table. Refer to the *MCS[®]-96 Utilities User's Guide*, listed in the Preface, for a further description of the segment map and the symbol table.

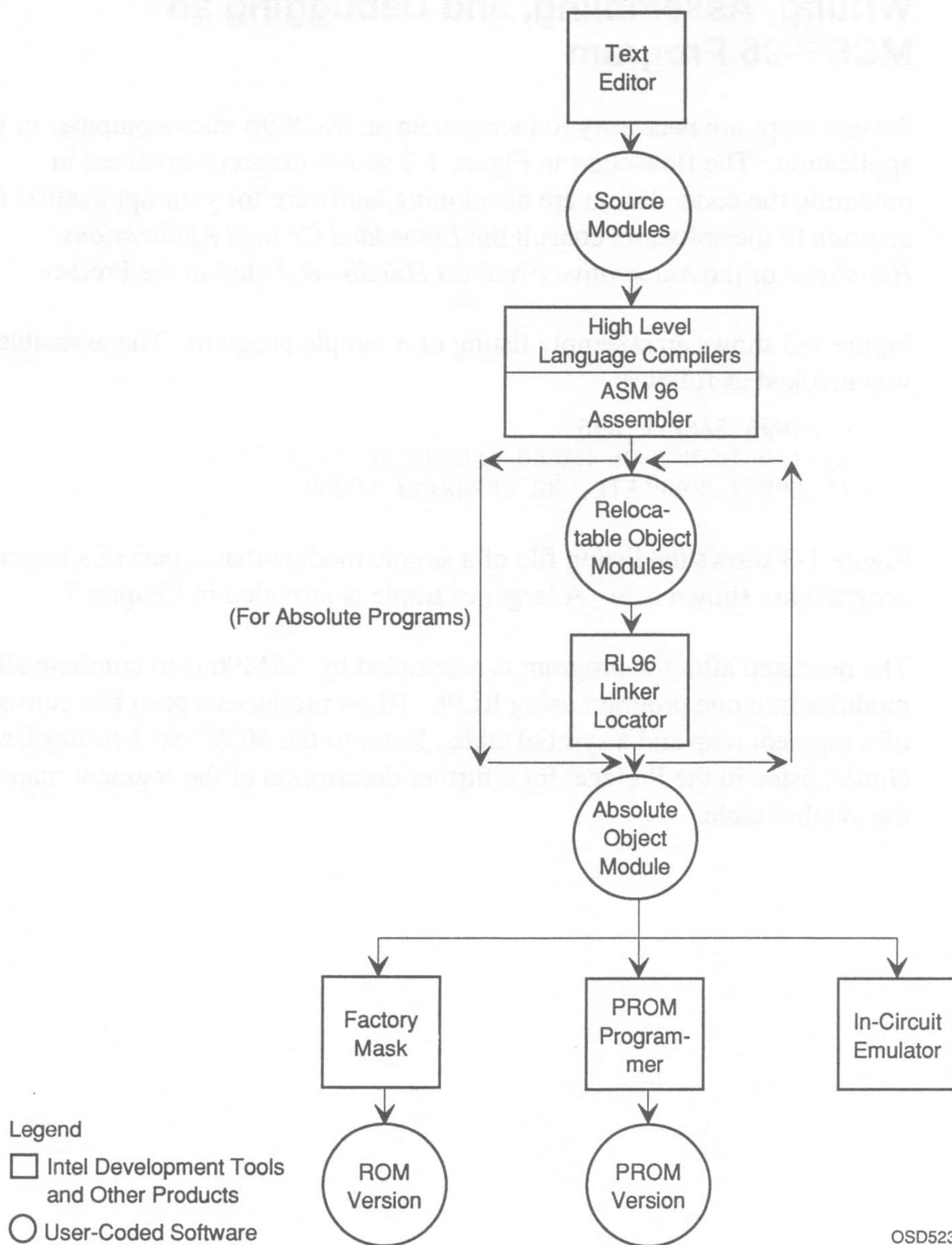


Figure 1-2 MCS®-96 Program Development Process

systemid MCS-96 MACRO ASSEMBLER, Vx.y

SOURCE FILE: f1.a96

OBJECT FILE: f1.obj

CONTROLS SPECIFIED IN INVOCATION COMMAND: debug nosymbols

LOC	OBJECT	STMT	SOURCE STATEMENT
		1	\$TITLE('8096-BASED MONITOR')
		2	;The main module of an 8096-based monitor
		3	
		4	;Symbol definitions
000D		5	CR EQU 13 ;Carriage-return character (ASCII)
000A		6	LF EQU 10 ;Line-feed character (ASCII)
		7	
		8	EXTRN Console_Out:ENTRY, Monitor:ENTRY
		9	
2080		10	CSEG AT 2080H ;Entry point upon reset
2080 C90000	R	11	START: PUSH #Signon ;Print signon message
2083 EF0000	E	12	CALL Console_Out
2086 E70000	E	13	BR Monitor ;Enter the monitor loop
		14	
0000		15	CSEG ;Start a relocatable segment
0000 20383039362D4241		16	Signon: DCB Len, '8096-BASED MONITOR Vx.y . CR, LF'
0020		17	Len EQU \$-Signon-1 ;ASM-time computed string length
		18	
0021		19	END

ASSEMBLY COMPLETED, NO ERROR(S) FOUND.

Figure 1-3 Sample Program Listing

1.5 Hardware Overview

The MCS-96 family of 16-bit microcontrollers consists of two sub-families: the 8096 and the 80C196 microcontrollers. The 8096 microcontrollers are listed in Table 1-1. Each proliferation is listed in Table 1-2. The 80C196 microcontrollers are listed in Table 1-3.

The 8096 microcontroller features include:

- High-speed 16-by-16 multiply (6.25 microseconds with a 12MHz input frequency)
- High-speed 32-by-16 divide (6.25 microseconds with a 12MHz input frequency)
- High-speed subsystem with:
 - Up to 4 time capture inputs
 - Up to 4 time triggered outputs
 - 2 16-bit timer counters
 - 4 software counters
- Pulse-width modulated output
- 16-bit Watchdog timer
- Full duplex serial port with dedicated baud rate generator
- Five 8-bit I/O ports
- 22 interrupt sources with 8 vector locations
- 232-byte register file
- Memory-to-memory (register-to-register) architecture
- 10-bit A/D converter

Table 1-1 8096 Microcontrollers

Microcontroller	Description
8096-90	A 16-bit microcontroller designed for high-speed control functions
8096-BH	An 8096-90 with extended temperature range, two added counter/timers, and other functionality improvements

Table 1-2 8x9y Proliferations

Microcontroller	Pins	On-chip Memory	Bus Width	A/D Converter
8095	48	not present	16 bits	4 analog inputs
8096	68	not present	16 bits	not present
8097	68	not present	16 bits	8 analog inputs
8098	48	8K bytes ROM	8 bits	4 analog inputs
8395	48	8K bytes ROM	16 bits	4 analog inputs
8396	68	8K bytes ROM	16 bits	not present
8397	68	8K bytes ROM	16 bits	8 analog inputs
8795	48	8K bytes EPROM	16 bits	4 analog inputs
8796	68	8K bytes EPROM	16 bits	not present
8797	68	8K bytes EPROM	16 bits	8 analog inputs

Table 1-3 80C196 Microcontrollers

Microcontroller	Description
80C196KB	In addition to the features available on the 8096 microcontrollers, the 80C196KB features include: - high-speed 16-by-16 multiply (2.33 microseconds with a 12MHz input frequency) - high-speed 32-by-16 divide (4.0 microseconds with a 12MHz input frequency) - independent capture of timer2 and up and down counting on timer2 in the high-speed subsystem - 6 additional interrupt sources with 10 additional vectors - 6 additional instructions - faster cycle times on many existing instructions - powerdown and idle modes for power savings
80C196KC	In addition to the features available on the 80C196KB microcontroller, the 80C196KC features include: - 6 additional instructions - Peripheral transaction server for more flexible interrupt handling

Figures 1-4, 1-5, and 1-6 are block diagrams of the 8096, 80C196KB, and 80C196KC microcontrollers. For more hardware design information on the MCS-96 microcontrollers, refer to the *16-Bit Embedded Controller Handbook*.

The MCS-96 processors support byte and word operations. Bits and double-words (32-bits) are supported for a subset of the instruction set. The CPU on the 80C196 is 16 bits wide and is connected to the interrupt controller and the memory controller by a 16-bit bus. In addition, an 8-bit bus is used to transfer opcodes from the memory controller to the CPU. The 16-bit bus between the CPU and memory controller does not exist on the 8096, so the 8-bit bus is used for both data and opcode transfers. All peripheral devices on the 80C196 are connected to the CPU by a 16-bit bus.

Internal operation of the microcontroller is based on the oscillator frequency divided by three (for the 8096) or two (for the 80C196), giving the basic operating time unit known as a state time. The MCS-96 state times are shown in Table 1-4. Either a crystal or an external source can be used to drive the oscillator. Instruction execution times average $1/2$ to 2 microseconds in typical applications.

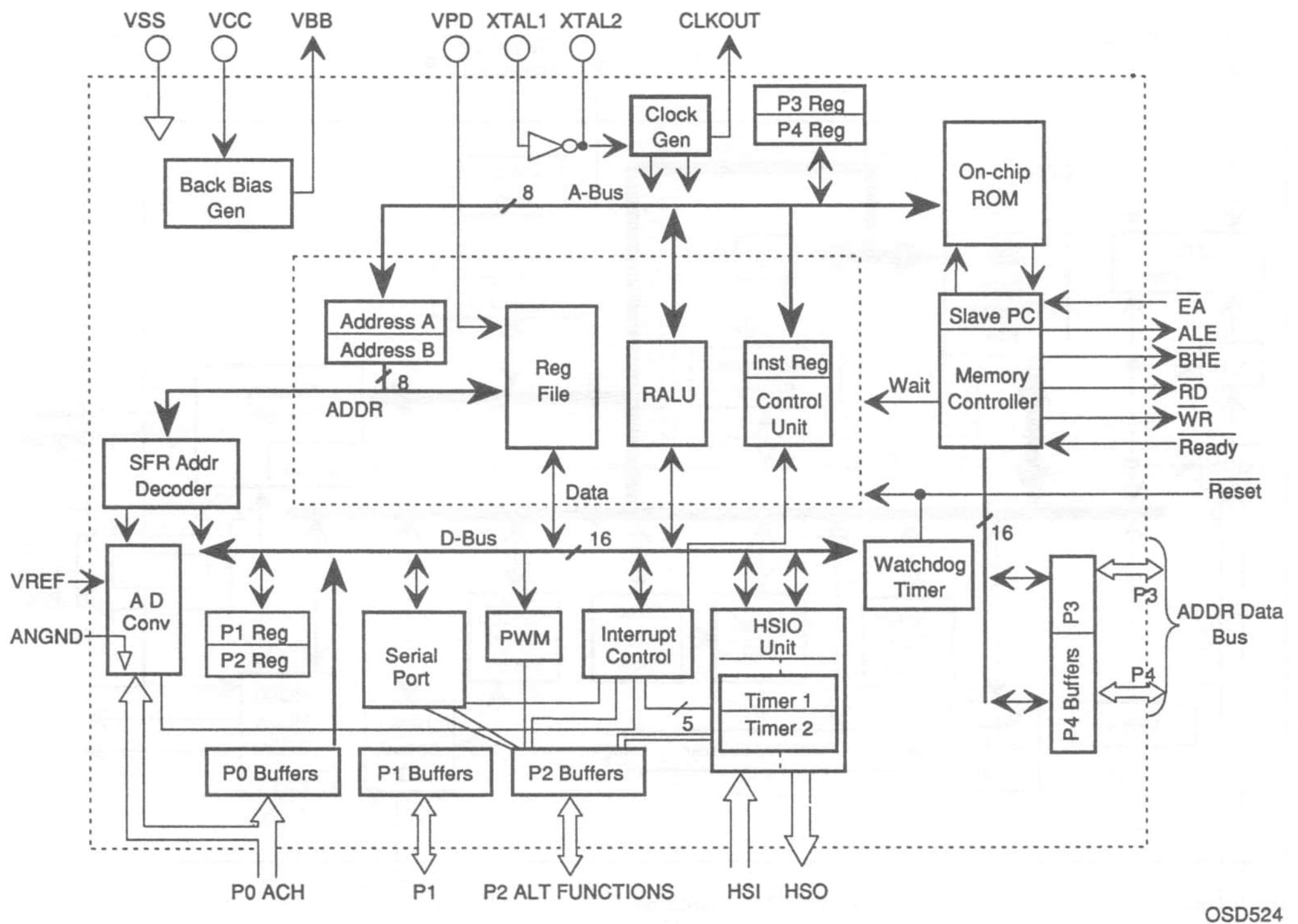


Figure 1-4 Block Diagram of 8096 Microcontroller

Table 1-4 MCS[®]-96 State Times

Microcontroller Family	Oscillator Frequency		
	16 Mhz	12Mhz	8Mhz
8096		250ns	374ns
80C196	125ns	167ns	250ns

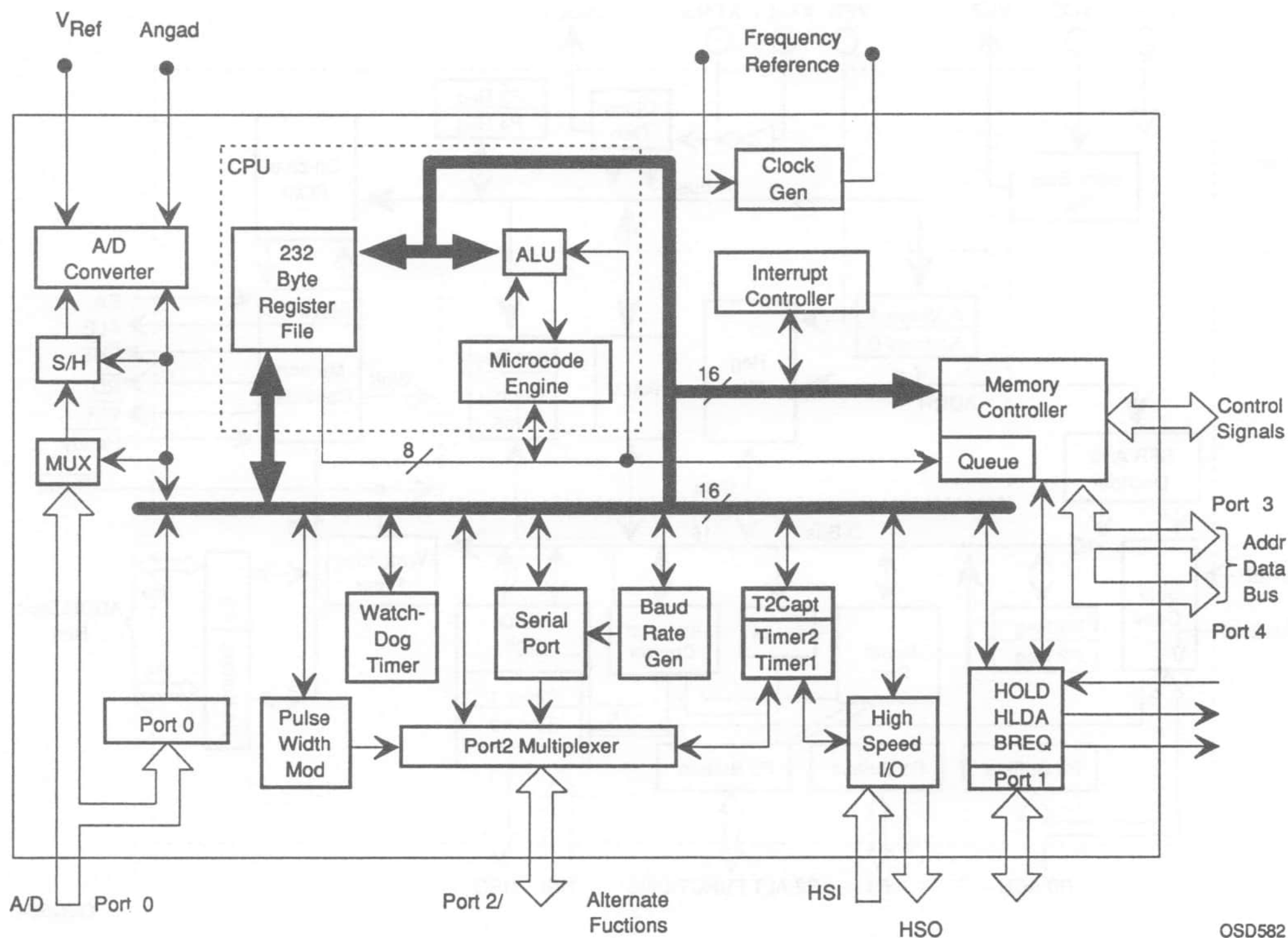


Figure 1-5 Block Diagram of 80C196KB Microcontroller

Four high-speed capture inputs are provided to record the times at which external events occur. Six high-speed pulse generator outputs are provided to trigger external events at preset times. The high-speed output unit can have up to four 16-bit software timers in operation at once.

An operational on-chip A/D converter converts up to four (in the 48-pin version) or eight (in the 68-pin version) analog input channels to 10-bit digital values.

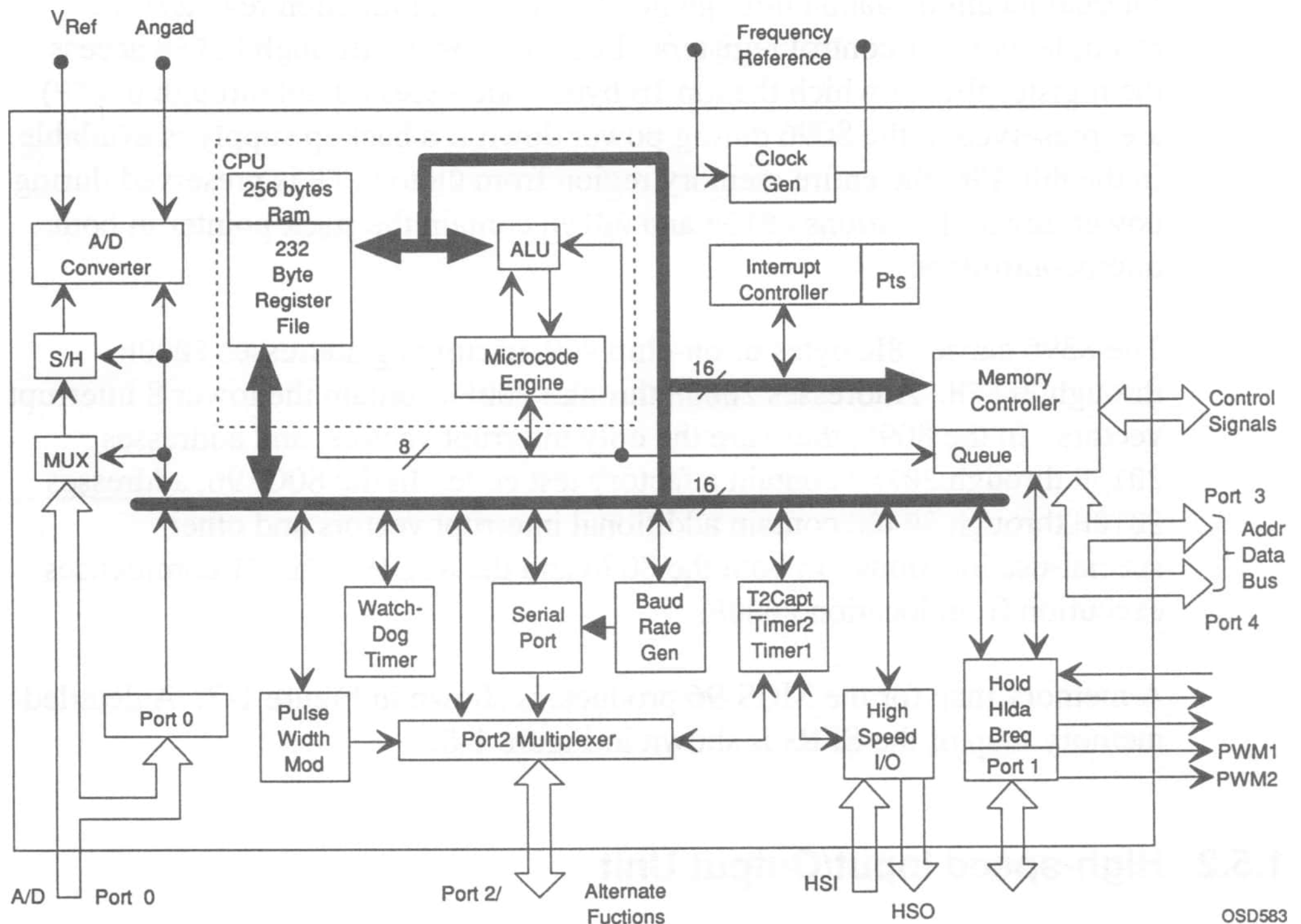


Figure 1-6 Block Diagram of 80C196-KC Microcontroller

1.5.1 Memory Space

The MCS-96 uses the same address space for both code and data memory. They can execute instructions from any memory address. Instructions cannot, however, be fetched from internal RAM. That is, the contents of the special function registers (SFRs) cannot be used as instructions. Instruction fetches from locations 0000H through 00FFH (8096BH and 80C196KB) or through 01FFH (80C196KC) are automatically directed to external memory. This is true even if the processor has onboard ROM or EPROM. This section in external memory is reserved for use by Intel development tools.

Internal locations 0000H through 0017H are special function registers; for example, ports or control registers. Locations 0018H through 00FFH access the register file, of which the top 16 bytes (addresses 00F0H through 00FFH) are preserved in the 8096 during power down if a backup supply is available. In the 80C196, the entire memory region from 0H to 0FFH is preserved during power down. Locations 0018H and 0019H contain the stack pointer in both microcontrollers.

The 8396 carries 8K bytes of on-chip ROM, occupying addresses 2000H through 3FFFH. Addresses 2000H through 200FH contain the lower 8 interrupt vectors. In the 8096, these are the only interrupt vectors, and addresses 2010H through 207FH contain a factory test code. In the 80C196, addresses 2010H through 2040H contain additional interrupt vectors and other special-use locations. In both the 8096 and the 80C196, RESET commences execution from location 2080H.

A memory map for the MCS-96 products is shown in Figure 1-7. A detailed memory map of the SFRs is shown in Figure 1-8.

1.5.2 High-speed Input/Output Unit

Four individual sections make up the high-speed input/output (HSIO) unit and work together to form a very flexible timer/counter-based I/O system. Included in the HSIO are two 16-bit timers (timer 1 and timer 2), a programmable high-speed input unit, and a programmable high-speed output unit. In the 80C196, timer 2 can count both up and down.

With very little CPU overhead the HSIO can measure pulse widths, generate waveforms, and create periodic interrupts. Depending on the application, it can perform the work of up to 18 timer/counters and capture or compare registers.

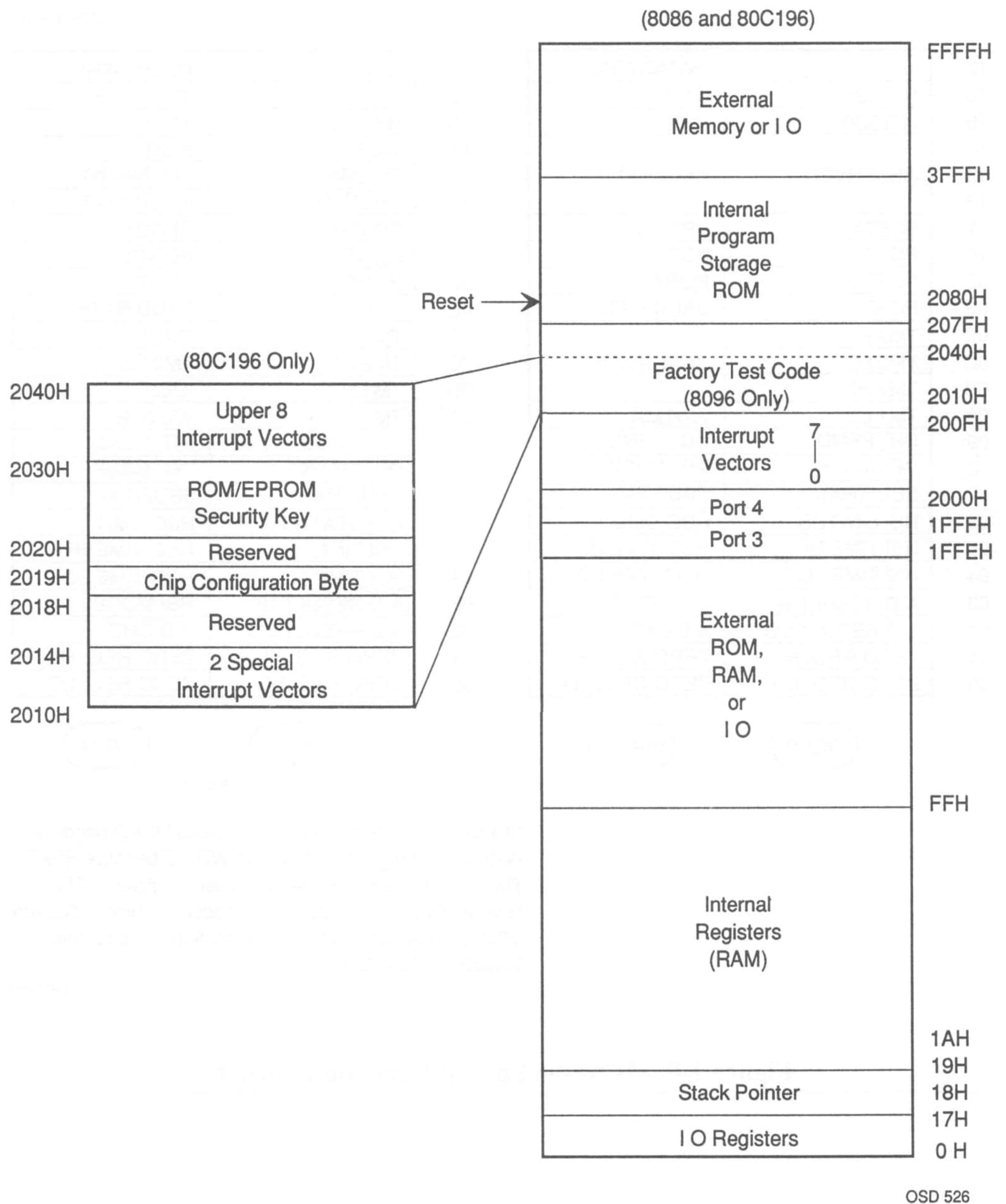


Figure 1-7 MCS®-96 Memory Map

WSR = 0*

17		PWMCNTRL
16	HSIOS1	IOC1
15	HSIOS0	IOC0
14		
13	RESERVED	RESERVED
12		
11	SPSTAT	SPCON
10	PORT2	PORT2
0F	PORT1	PORT1
0E	PORT0	BAUD RATE
0D	TM2 HI	RESERVED
0C	TM2 LO	
0B	TM1 HI	
0A	TM1 LO	WDTMR
09	INT_PEND	INT_PEND
08	INT_MASK	INT_MASK
07	SBUF(RX)	SBUF(TX)
06	HSI STATUS	HSO CMD
05	HSI TIME HI	HSO TIME HI
04	HSI TIME LO	HSO TIME LO
03	A D RESULT HI	HSI MODE
02	A D RESULT LO	A D CMD
01	ZERO REG. HI	ZERO REG. HI
00	ZERO REG. LO	ZERO REG. LO

READ

WRITE

(8096)

17	IOS2	PWMCNTRL
16	HSIOS1	IOC1
15	HSIOS0	IOC0
14	WSR	WSR
13	INT_MASK1	INT_MASK1
12	INT_PEND1	INT_PEND1
11	SPSTAT	SPCON
10	PORT2	PORT2
0F	PORT1	PORT1
0E	PORT0	BAUD RATE
0D	TM2 HI	TM2 HI
0C	TM2 LO	TM2 LO
0B	TM1 HI	IOC2
0A	TM1 LO	WDTMR
09	INT_PEND	INT_PEND
08	INT_MASK	INT_MASK
07	SBUF(RX)	SBUF(TX)
06	HSI STATUS	HSO CMD
05	HSI TIME HI	HSO TIME HI
04	HSI TIME LO	HSO TIME LO
03	A D RESULT HI	HSI MODE
02	A D RESULT LO	A D CMD
01	ZERO REG. HI	ZERO REG. HI
00	ZERO REG. LO	ZERO REG. LO

READ

WRITE

(80C196)

*If WSR = 15, registers shown here as READ become WRITE, and registers shown as WRITE become READ. The exceptions to this are the Timer2 registers. The register TM2 HI, at location 0D, becomes Timer2 Capture (High); the register TM2 LO, at location 0C, becomes Timer2 Capture (Low).

OSD527

Figure 1-8 MCS®-96 Special Function Registers

1.5.2.1 Timers

The high-speed input/output unit contains two 16-bit timer registers. The timer 1 register is a free-running timer which is automatically incremented every eight state times. For the 8096, it can be cleared only by RESET. For the 80C196, however, it can be cleared by writing zero to timer 1 when $WSR = 15$.

The timer 2 register, also 16 bits, can be programmed to count transitions on either $T2CLK$ or $HSI.1$. The selected input to timer 2 is sampled every state time and the count is incremented when the sample train shows a transition (any transition: 0-to-1 or 1-to-0) has occurred. Timer 2 is cleared by RESET or by software, and can be programmed to be cleared by signals from either $T2RST$ or $HSI.0$. Either timer register can be programmed to generate an interrupt on overflow.

I/O register expansion on the 80C196 is done by switching between register windows 0 and 15. This switching is done using the window select register (WSR). The *PUSHA* and *POPA* instructions can be used to push and pop the WSR and second interrupt mask. For more information on the effect of the WSR on register use, see the handbooks listed in the Preface.

1.5.2.2 High-speed Inputs

The HSI unit looks for transitions in any of its input lines, and when it sees one it records the time (from timer 1) and which input line made the transition. This information is stored in an eight-deep *FIFO*. The unit can be programmed to look for either positive and/or negative transitions, or 8 positive transitions. It can also be programmed to activate the HSI data available interrupt either when the first entry to the *FIFO* has been made or when the seventh entry has been made.

The HSI unit has four input lines: $HSI.0$ through $HSI.3$. Two of these lines are shared with the high-speed output unit. Each input line can be individually enabled or disabled to the HSI unit by software.

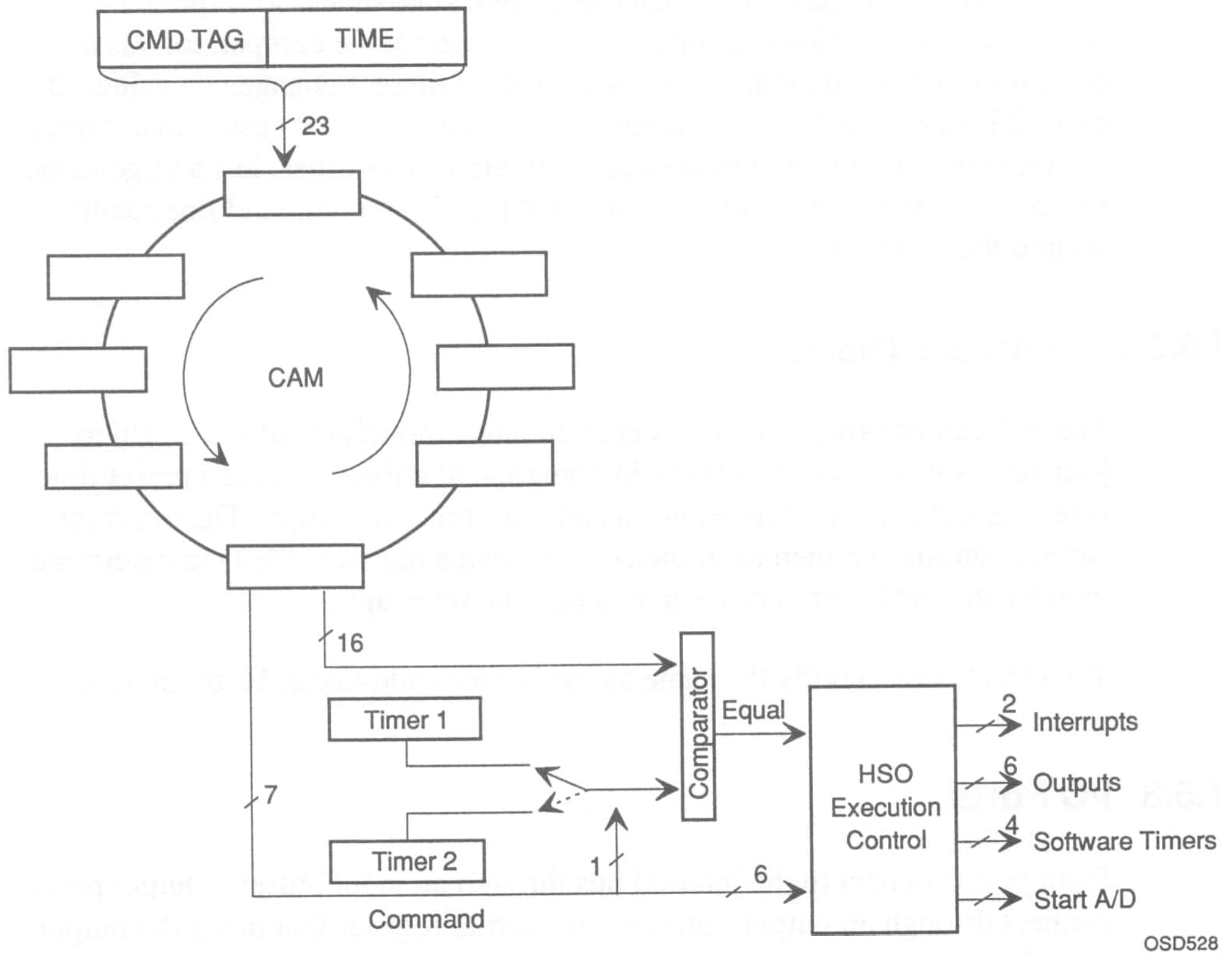
1.5.2.3 High-speed Outputs

The high-speed output (HS0) unit (Figure 1-9) can be programmed to cause transitions on any of its output lines at specified times. The HS0 unit has six output lines, HS0.0 through HS0.5. HS0.4 and HS0.5 are shared with the HSI unit as HSI.2 and HSI.3. These two outputs can be individually enabled or disabled by software.

The HS0 unit can be programmed to set or clear any of its output lines, or reset timer 2, or trigger an A/D conversion at the preset time. Either timer 1 or timer 2 can be selected to be the reference timer for this purpose. Up to eight such preset actions can be in storage at any one time. As each action is carried out at its preset time, its command is deleted from storage.

Storage of HS0 commands is an eight-deep content addressable memory (CAM) file. Each CAM register is 23 bits wide. Sixteen bits specify the time at which the action is to be carried out; seven bits specify the nature of the action and whether timer 1 or timer 2 is the reference.

To enter a command into the CAM file, write the seven-bit command tag into location 0006H and the time at which the action is to be carried out into word address 0004H. Writing to these addresses accesses the HS0 holding register. The command does not actually enter the CAM file until an empty CAM register rotates around the holding register (see Figure 1-9).



OSD528

Figure 1-9 High-speed Output Unit

The CAM file rotates one register position per state time. Thus it takes eight state times for the holding register to have had access to all eight CAM registers. Similarly, it takes eight state times for the comparator to have had access to all eight CAM registers. This defines the time resolution of the HSO unit to be eight state times.

Timer 1 is incremented only once every eight state times, so if timer 1 is being used as the reference timer for an HSO action, the comparator has a chance to look at all eight CAM registers before timer 1 changes its value. If timer 2 is being used as the reference timer, the user must ensure that it does not increment more often than once every eight state times, lest a target-time for an HSO action come and go before that particular command line rotates around the comparator.

1.5.2.4 Software Timers

The HSO can be programmed to generate interrupts at preset times. Up to four such software timers can be in operation at a time. As each preset time is reached, the HSO unit generates a software timer interrupt. The interrupt service routine can then examine an HSI0 status register (IOS1) to determine which software timer expired and caused the interrupt.

The effect is essentially the same as having four additional 16-bit timers.

1.5.3 I/O Ports

Input ports connect to the internal bus through an input buffer. Output ports connect through an output buffer to an internal register that holds the output bits. Bidirectional ports consist of an internal register, an output buffer, and an input buffer. Bidirectional ports are shown in Figures 1-4 and 1-5 as consisting of two separate blocks: the internal register and the I/O buffer.

When an instruction accesses a bidirectional port as a source register, the question often arises as to whether the value that is brought into the CPU comes from the internal port register or from the port pins through the input buffer. In the 8096, the value always comes from the port pins, never from the internal register.

Port 0 is an input-only port that shares its pins with the analog inputs to the A/D converter. One can read port 0 digitally, or, by writing the appropriate control bits to the A/D command register, select one of the lines of this port to be the input to the A/D converter.

Port 1 is a quasi-bidirectional I/O port. Quasi-bidirectional means the port pin has a weak internal pullup that is always active and an internal pulldown that can either be on (to output a 0) or off (to output a 1). If the internal pulldown is left off (by writing a 1 to the pin), the pin's logic level can be controlled by an external pulldown that can either be on (to input a 0) or off (to input a 1). From the user's point of view the main distinction is that a quasi-bidirectional input sources current while being externally held low.

In parallel with the weak internal pullup is a much stronger internal pullup that is activated for one state time when the pin is internally driven from 0 to 1. This is done to speed up the 0-to-1 transition time.

Port 2 is a multi-functional port. Six of its pins are shared with other functions in the 8096, and all eight pins are shared in the 80C196, shown in Table 1-5.

Table 1-5 Port 2 Pin Functionality

Port	Function	Alternate Function
P2.0	output	TXD (serial port transmit)
P2.1	input	RXD (serial port receive)
P2.2	input	EXTINT (external interrupt)
P2.3	input	T2CLK (timer 2 input)
P2.4	input	T2RST (timer 2 reset)
P2.5	output	PWM (pulse-width modulated signal)
P2.6	quasi-bidirectional	Timer2 up/down select (80C196)
P2.7	quasi-bidirectional	Timer2 Capture (80C196)

Ports 3 and 4 are bidirectional with open-drain outputs in all pins. Also, these two ports share their pins with the memory controller.

1.5.4 Serial Port

The serial port is compatible with the MCS[®]-51 serial port. It is full duplex, meaning it can transmit and receive simultaneously. It can also commence reception of a second byte before a previously received byte has been read from the receive register. The serial port registers are both accessed at the same address. Writing to this location accesses the transmit register, and reading accesses a physically separate receive register.

1.5.5 Pulse Width Modulator

The pulse width modulator (PWM) output of the 8096BH and the 80C196KB shares a pin with the P2.5 port bit. Bit 0 of control register IOC1 selects the function of this pin to be either PWM or P2.5. The 80C196KC has three PWM outputs. PWM0 shares P2.5 port bit, and PWM1 and PWM2 share P1.3 and P1.4 port bits, respectively. Setting IOC1.0 selects the PWM0 function, and setting T2CNTC register bit 2 and 3 when WSR = 1 enables PWM1 and PWM2 outputs over the port function. When PWM is selected, this pin outputs a pulse train having a fixed period of 256 state times, and a programmable width of 0 to 255 state times. The width is programmed by loading the desired value, in state times, to the PWM_Control register (8096BH and 80C196KB) and PWM1_Control and PWM2_Control registers (80C196KC only). Table 1-6 shows the PWM outputs of 8096 and 80C196 microcontrollers.

Table 1-6 PWM and Controlling Pins

Microcontroller	Pulse Width Modulator		Pin
	Channel	Controlling Bit = 1	
8096BH	0	IOC1.0	P2.5
80C196KB	0	IOC1.0	P2.5
80C196KC	0	IOC1.0	P2.5
	1	T2CNTC.2	P1.3
	2	T2CNTC.3	P1.4

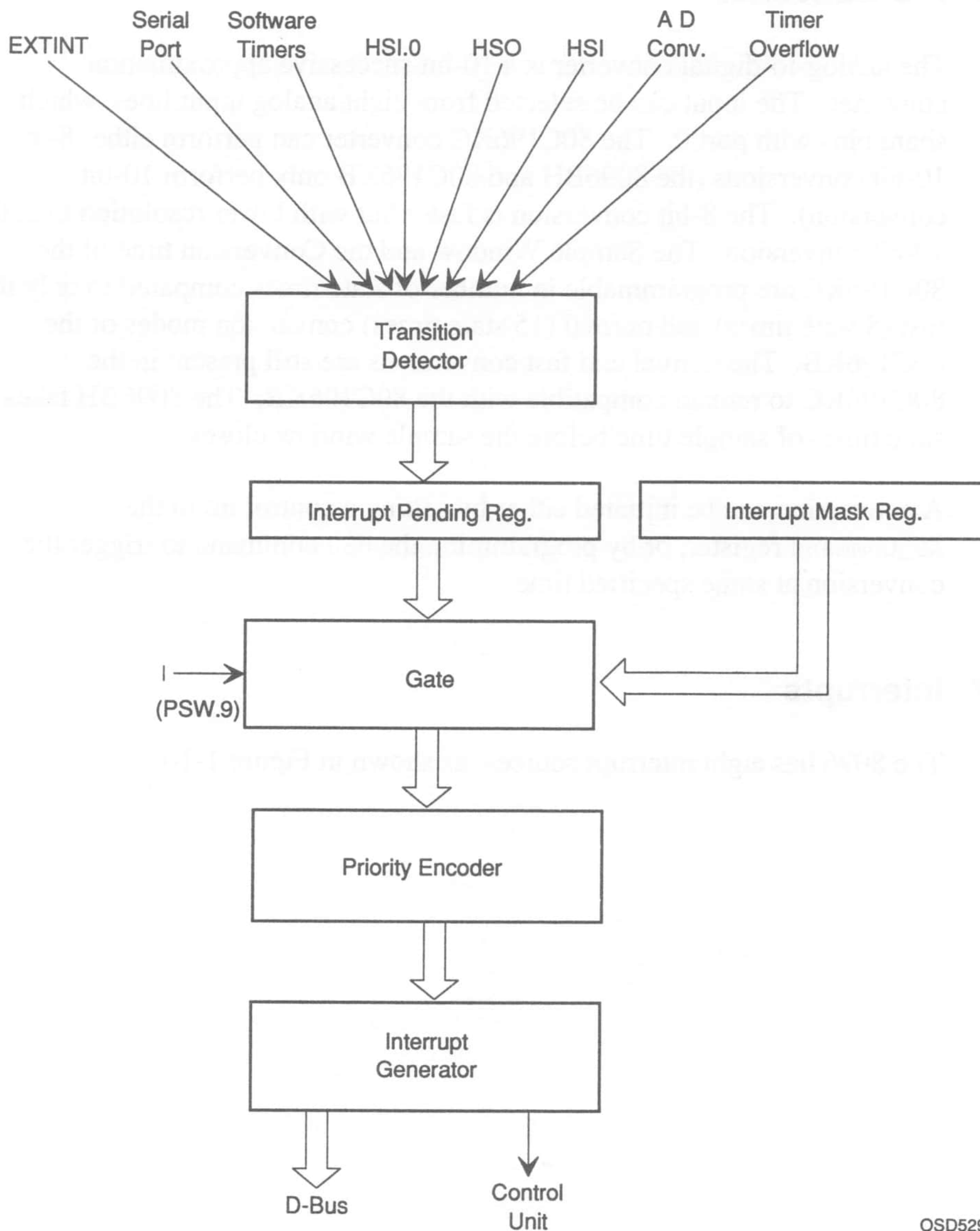
1.5.6 A/D Converter

The analog-to-digital converter is a 10-bit successive approximation converter. The input can be selected from eight analog input lines, which share pins with port 0. The 80C196KC converter can perform either 8- or 10-bit conversions (the 8096BH and 80C196KB only perform 10-bit conversion). The 8-bit conversion is faster but with lower resolution than the 10-bit conversion. The Sample Window and the Conversion time of the 80C196KC are programmable in number of state times compared to only the fast (8 state times) and normal (15 state times) conversion modes of the 80C196KB. The normal and fast conversions are still present in the 80C196KC to remain compatible with the 80C196KB. The 8096BH takes 4 state times of sample time before the sample window closes.

A conversion can be initiated either by setting a control bit in the AD_Command register, or by programming the HS0 command to trigger the conversion at some specified time.

1.5.7 Interrupts

The 8096 has eight interrupt sources, as shown in Figure 1-10.



OSD529

Figure 1-10 8096 Interrupt Structure

A 0-to-1 transition from any of the sources sets a corresponding bit in the interrupt pending register. The content of the interrupt mask register determines whether a pending interrupt is serviced. If it is to be serviced, the CPU pushes the current program counter onto the stack and reloads it with the vector corresponding to the interrupt. The interrupt vectors are located in addresses 2000H through 200FH, as shown in Table 1-7.

Table 1-7 8096 Interrupt Vector and Priorities

Interrupt Source	Vector Location (Low Byte)	Priority
EXTINT	200EH	7 (highest)
serial port	200CH	6
software timers	200AH	5
HSI.0	2008H	4
high speed outputs	2006H	3
HSI data available	2004H	2
A/D conversion complete	2002H	1
timer overflow	2000H	0 (lowest)

Execution then vectors to the interrupt service routine (ISR). At least one instruction in the ISR is always executed before a second interrupt is serviced. Any ISR can itself be interrupted, but not before the first instruction of the current ISR has been executed. Normally, this first instruction is PUSHF (push flags). PUSHF does two things: it pushes the current PSW onto the stack, and it clears the PSW. Clearing the PSW clears the interrupt mask register, in effect blocking further interrupts. In the next instructions, if the programmer so desires, interrupts from selected sources can be re-enabled. In this manner, the software determines the priority structure.

The ISR then would normally terminate with POPF (pop flags), which restores the original PSW, and RET, which restores the original program counter.

While the 8096 has only a single priority level in the sense that any interrupt can be itself interruptible, it does have a priority structure for resolving simultaneous interrupts, as shown in Table 1-7.

Table 1-8 shows all register bits needed to be manipulated in order to enable each 8096 interrupt and other registers needed to use the feature.

Table 1-8 8096 Interrupt Enable Bits

Interrupt	Mask	Control Bits	+	Reg Bits
EXTINT	INT_MASK.7	IOC1.1		
Serial Port	INT_MASK.6			SP_CON, Baud_Rate, IOC1.5=1
Software Timers	INT_MASK.5	HSO_Command		
HSI.0	INT_MASK.4	HSI PIN(Rising edge)		
High Speed Outputs	INT_MASK.3	HSO_Command		
HSI Data Available	INT_MASK.2	IOC1.7		IOC0
A/D Conversion Complete	INT_MASK.1			AD_Command, HSO_Command
Timer Overflow	INT_MASK.0	IOC1.2 or IOC1.3 =1		

The 80C196 has an additional 8 interrupts and 2 special interrupts, listed in Table 1-9. The unimplemented opcode, trap, and non-maskable interrupts cannot be disabled.

Table 1-9 80C196 Additional Interrupt Vector and Priorities

Interrupt Source	Vector Location (Low Byte)	Priority
non-maskable	203EH	15 (highest)
HSI FIFO full	203CH	14
EXTINT Pin	203AH	13
timer2 overflow	2038H	12
timer2 capture	2036H	11
4th entry into HSI FIFO	2034H	10
RI	2032H	9
TI	2030H	8
trap (special interrupt)	2012H	N/A
unimplemented opcode (special interrupt)	2010H	N/A

The 80C196 Interrupt Structure is shown in Figure 1-11. The transition detector looks for 0 to 1 transitions on any of the sources. External sources have a maximum transition speed of one edge every state time. The interrupt line should be held at least one state time, or it may not be detected.

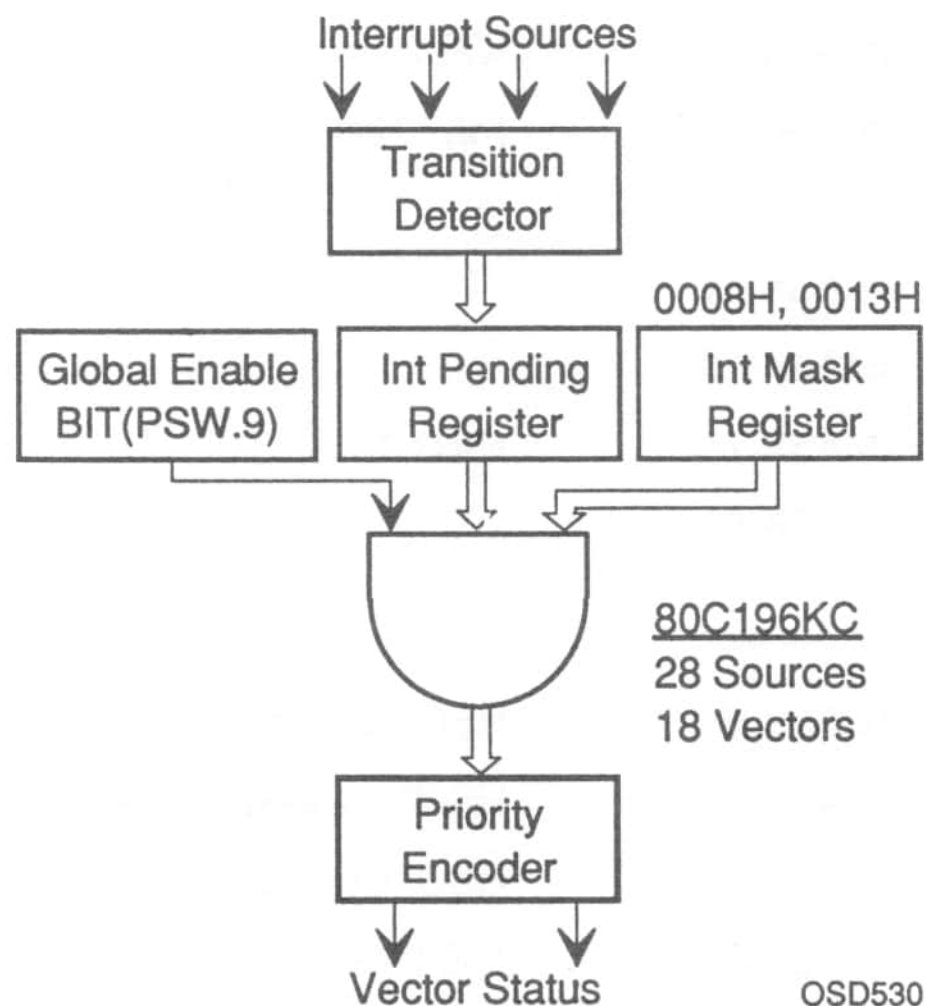


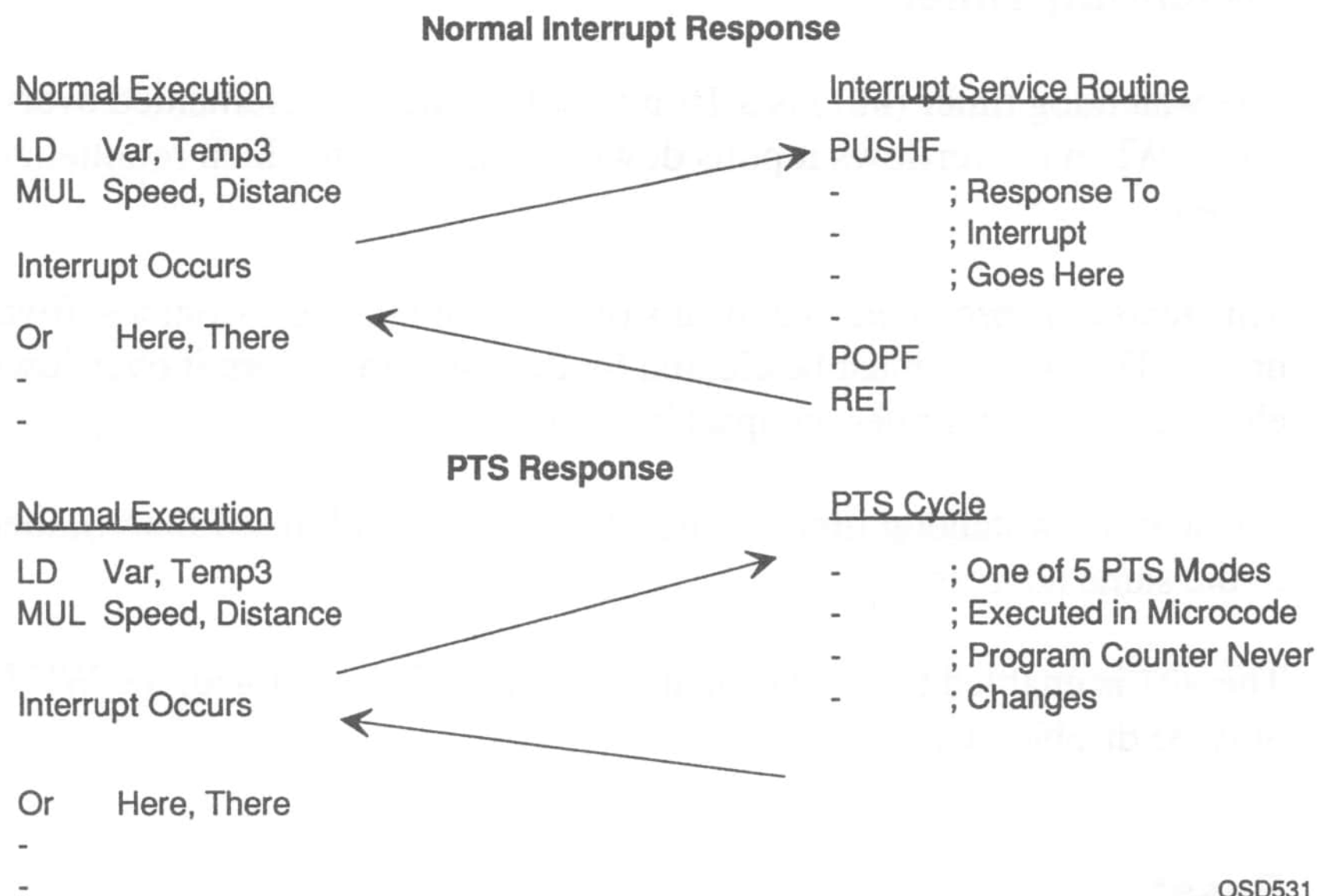
Figure 1-11 80C196 Interrupt Structure

Table 1-10 shows all register bits needed to be manipulated in order to enable each 80C196 interrupt and other registers needed to use the feature.

Table 1-10 80C196 Interrupt Enable Bits

Interrupt	Mask	Control Bits	+	Reg Bits
NMI	IMASK1.7	NMI PIN		
HSI FIFO Full	IMASK1.6	IOC1.7=1		IOC0
EXTINT Pin	IMASK1.5	IOC1.1		
Timer2 overflow	IMASK1.4	IOC1.3=1, IOC2.5		
Timer2 capture	IMASK1.3	P2.7(Rising Edge)		
4th entry into HSI FIFO	IMASK1.2			IOC0
RI	IMASK1.1			
TI	IMASK1.0			IOC1.5
EXTINT	IMASK.7	IOC1.1		
Serial Port	IMASK.6			SP_CON, Baud_Rate, IOC1.5=1
Software Timers	IMASK.5	HSO_Command		
HSI.0	IMASK.4	HSI PIN(Rising edge)		
High Speed Outputs	IMASK.3	HSO_Command		
HSI Data Available	IMASK.2	IOC1.7=0		IOC0
A/D Conversion Complete	IMASK.1			AD_Command HSO_Command
Timer Overflow	IMASK.0	IOC1.2 or IOC1.3 =1		

Besides the 10 additional interrupts, the 80C196KC contains a peripheral transaction server (PTS) that can reduce the overhead involved in handling an interrupt. Figure 1-12 shows the difference between normal interrupt handling and interrupt handling through the PTS. A normal interrupt sequence calls the interrupt handler, pushes (PUSHF) the program status word (PSW), processes the interrupt, pops (POPF) the program status word, and returns (RET) to the program. PTS interrupt handling eliminates the call, PUSHF, POPF, and RET by generating a PTS cycle to be executed in microcode.



OSD531

Figure 1-12 PTS Interrupt Response

A PTS interrupt can do any of the following:

- move data in single or block transfers between memory locations
- read data from the A/D converter
- save the HSI FIFO to a table in either internal or external memory
- load the HSO CAM from a table in either internal or external memory

A PTS interrupt cannot service an unimplemented opcode, trap, or non-maskable interrupt.

To use the PTS, you must use the EPTS instruction to set the peripheral server enable (PSE) bit in the PSW. To disable the PTS, use the DPTS instruction. Chapter 3 describes the EPTS and DPTS instructions. For more information on the structure and operation of the PTS, see the handbooks and the *80C196KC User's Manual*, listed in the Preface.

1.5.8 Watchdog Timer

The watchdog timer (WDT) is a 16-bit counter that is incremented every state time. When it overflows it pulls down the RESET pin which reinitializes the system.

This feature is provided as a means of graceful recovery from a software upset. The counter must be cleared by the software before it overflows, or else the system assumes an upset has occurred and activates RESET.

To clear the watchdog timer, write 1EH to the WDT address (000AH), then 0E1H to the same address.

The WDT is enabled the first time it is cleared. Once enabled, the WDT can only be disabled by a RESET.

1.5.9 Reset

Refer to the *16-Bit Embedded Controller Handbook* for reset information for each specific component.

1.5.10 Special Function Registers

1.5.10.1 PWM Control Register

The width of the PWM output pulse is programmed by writing the desired value in state times (0 to 0FFH) into this register.

1.5.10.2 Serial Port Control/Status Register

Bit:	7	6	5	4	3	2	1	0
	RB8/RPE	RI	TI	TB8	REN	PEN	M2	M1

Where:

M1, M2	specifies the mode.
PEN	enables the parity function (even parity).
REN	enables the receive function.
TB8	programs the 9th data bit (if not parity) on transmission.
TI	is the transmit interrupt flag.
RI	is the receive interrupt flag.
RB8	is the 9th data bit received (if not parity).
RPE	is the parity error indicator (if parity active).

1.5.10.3 A/D Command Register

Bit:	7	6	5	4	3	2	1	0
	X	X	..MODE..	GO		channel #		

Where:

MODE	selects whether to perform 8 or 10 bit conversion. 00 selects 10-bit conversion; 01 selects 8-bit conversion (80C196KC only).
channel #	selects which of the eight analog input channels is to be converted to digital form.
GO	indicates when the conversion is to be initiated. GO = 1 means start now; GO = 0 means the conversion is to be initiated by the HS0 unit at a specified time.

1.5.10.4 A/D Result Register

Bit:	15-8	7-6	5	4	3	2	1	0
	MSB	LSB	...MODE...		ST	 channel #		

(10-bit A/D Result)

Where:

channel # tells which of the eight analog input channels this was converted from.

ST is a STATUS bit. $S = 1$ means an A/D conversion is currently under way. $S = 0$ means the A/D unit is currently idle.

MODE shows whether 8- or 10-bit conversion was performed (80C196KC only).

Because of the way the A/D result register is implemented on the chip, it must be read as two separate bytes.

1.5.10.5 Interrupt Registers

The interrupt pending and mask registers each contain one bit for each of the eight interrupt sources: Bit 7 is for the highest priority source and bit 0 is for the lowest priority source. The interrupt mask register is the low byte of the PSW. The registers IMASK1 and IPEND1 contain the bits for the additional 8 interrupts sources of the 80C196.

1.5.10.6 HSI Mode Register

As previously described, the HSI unit looks for the desired transition in its input lines, and when it sees one it records the time (from timer 1) and in which input line the transition occurred.

The HSI mode register defines what kinds of input transitions cause the contents of timer 1 to be captured into the HSI FIFO, as follows:

Bit:	7	6	5	4	3	2	1	0
	HSI.3 Mode		HSI.2 Mode		HSI.1 Mode		HSI.0 Mode	

Each two-bit mode control field defines one of four possible modes:

- 00 eight positive transitions
- 01 positive transitions
- 10 negative transitions
- 11 any transition

High and low levels need to hold for at least one state time or they can be missed.

1.5.10.7 HSI Status Register

The HSI status register tells which input made a targeted transition and what its current state is (0 or 1), as follows:

Bit:	7	6	5	4	3	2	1	0
	HSI.3		HSI.2		HSI.1		HSI.0	

For each two-bit field, the lower bit indicates whether or not the transition has been made at this input, and the upper bit gives the current state at that pin.

1.5.10.8 HSO Command Tag

The HSO unit can be programmed to cause transitions in any of its output lines at specified times. The specified time can be taken from either timer 1 or timer 2. The unit has six output lines, but two of them (HSO.4 and HSO.5) are shared by the HSI unit. The unit can initiate a number of other actions at specified times: initiate an A/D conversion, reset timer 2, or indicate a software timer time-out. It can initiate these actions with or without an accompanying interrupt.

The HSO Command register is as follows:

Bit:	7	6	5	4	3	2	1	0
	L	T	1/0	I	 channel #			

Where:

channel #	defines the recipient of the action.
I	specifies whether or not the action is to be accompanied by an interrupt.
1/0	specifies whether the action on an output line is to be set to 1 or to 0.
T	selects whether the "specified time" is to be from timer 1 (T = 0) or timer 2 (T = 1).
L	(80C196 only) locks events in CAM if this is enabled by IOC2.6.

The channel # codes are as follows:

Channel #	Recipient of Action
0-5H	output lines HSO.0-HSO.5
6H	output lines HSO.0 and HSO.1 combined
7H	output lines HSO.2 and HSO.3 combined
8H-0BH	software timers ST0-ST3 (the action being to establish it)
0CH	(80C196KC only) HSO pins 0-5 together; otherwise unflagged
0DH	unflagged event
0EH	timer 2 (the action being to reset it)
0FH	A/D converter (the action being to initiate a conversion)

1.5.10.9 I/O Control Registers

The 8096 has two I/O control registers, IOC0 and IOC1, and the 80C196 has an additional register called IOC2. IOC0 conditions timer 2 and the HSI lines. IOC1 selects some pin functions and enables or disables some interrupt sources. IOC2 selects timer 2, A/D and HSO features.

Timer 2 can count one of two selectable clock sources, T2CLK or HSI.1. It can be reset by one of two selectable hardware sources (T2RST or HSI.0) or by timed software (through the HSO unit) or by immediate software (through IOC0 bit 1). The selected hardware reset source pin can be enabled or disabled to this function.

The four HSI lines can be enabled or disabled to the HSI unit by setting or clearing bits in IOC0.

The bits of IOC0 are as follows:

Bit	Control Function
-----	------------------

- | | |
|---|---|
| 0 | Set 1 to enable HSI.0 to the HSI function. Clear 0 to disable HSI.0 to the HSI function. It can still be used to reset timer 2, however. |
| 1 | Writing a 1 to this bit resets timer 2 every time. |
| 2 | Set 1 to enable HSI.1 to the HSI function. Clear to 0 to disable HSI.1 to the HSI function. It can still be used to clock timer 2, however. |
| 3 | Set to 1 to enable the external timer 2 reset source (T2RST or HSI.0). Clear to disable this source. |
| 4 | Set to 1 to enable HSI.2 to the HSI function. |
| 5 | Select external timer 2 reset source: 0 selects T2RST: 1 selects HSI.0. |
| 6 | Set to 1 to enable HSI.3 to the HSI function. |
| 7 | Select timer 2 clock source: 0 selects T2CLK: 1 selects HSI.1. |

IOC1 selects some pin functions and enables or disables some interrupt sources. Port pin P2.5 can be selected to be the PWM output. The external interrupt source can be selected to be either EXTINT (same pin as P2.2) or Analog Channel 7 (ACH7, the same pin as P0.7). Timer 1 and timer 2 overflow interrupts can be individually enabled or disabled. The HSI interrupt can be selected to activate either when there is one FIFO entry or seven. Port pin P2.0 can be selected to be the TXD output. HSO.4 and HSO.5 can be enabled or disabled to the HSO unit.

The bits of IOC1 are as follows:

Bit	Control Function
0	Select pin function: If this bit is 1, P2.5 becomes PWM.
1	Select external interrupt source: 0 selects EXTINT: 1 selects ACH7 (P0.7).
2	Set to 1 to enable timer 1 overflow interrupt. Clear to disable.
3	Set to 1 to enable timer 2 overflow interrupt. Clear to disable.
4	HSI interrupt option: 0 selects to activate interrupt at first entry to FIFO. 1 selects to activate on seventh entry.
5	Select pin function: If this bit is 1, P2.0 becomes TXD.
6	Set to 1 to enable HSO.4 to the HSO function.
7	Set to 1 to enable HSO.5 to the HSO function.

IOC2 selects some of timer 2, A/D, and HSO features. Fast increment mode of timer 2 can be enabled to transition every one state time. Timer 2 can be configured as an up or down counter. The divide by 2 prescaler of the PWM can be enabled. The conversion time mode can be selected to use the 80C197KB fast and slow modes or the 80C196KC mode which takes the state time from the AD_TIME register. A/D clock prescaler can be disabled when using the 80C196KB fast mode. Timer 2 can be set to interrupt at 7FFFH/8000H boundary instead of 0FFFFH/0000H. A lock can be set to keep commands in the CAM. The entire CAM can be cleared.

The bits of IOC2 are as follows:

Bit	Control Function
0	Set to 1 to enable the fast increment mode which transition speed is once every state time.
1	Set to 1 to configure timer 2 to be an up/down counter.
2	Set to 1 to enable the PWM's divide by 2 prescaler.
3-4	Selects conversion time mode: 00 selects 80C196KB slow mode (15 state times), 10 selects 80C196KB fast mode (8 state times), X1 enables the 80C196KC mode in which the state time is taken from AD_TIME register. If using the 80C196KC conversion mode, it does not matter what's in bit 4.
5	Set to 1 enables timer 2 to interrupt on 7FFFH/8000H boundary.
6	Set to 1 to lock CAM entries.
7	Set to 1 to clear entire CAM.

1.5.10.10 HSIO Status Registers

There are two HSIO status registers. One gives the current status of the HSO lines and CAM. The other gives the overflow status of the four software timers, and of timers 1 and 2, and indicates the status of the HSI FIFO. The bits of IOS0 are as follows:

Bit	Function
0	Current state of HSO.0.
1	Current state of HSO.1.
2	Current state of HSO.2.
3	Current state of HSO.3.
4	Current state of HSO.4.
5	Current state of HSO.5.
6	A 0 indicates that the HSO holding register is empty and there is at least one empty register in the CAM.
7	A 0 indicates that the HSO holding register is empty.

The bits of IOS1 are as follows:

Bit	Function
0	a 1 value indicates software timer 0 has expired.
1	a 1 value indicates software timer 1 has expired.
2	a 1 value indicates software timer 2 has expired.
3	a 1 value indicates software timer 3 has expired.
4	a 1 value indicates timer 2 has overflowed.
5	a 1 value indicates timer 1 has overflowed.
6	a 1 value indicates the HSI FIFO contains at least seven entries.
7	a 1 value indicates the HSI FIFO contains at least one entry.

1.5.11 Program Status Word

The low byte of the program status word (PSW) is the interrupt mask register. The high byte is as follows:

Bit:	15	14	13	12	11	10	9	8
	Z	N	V	VT	C	PSE	I	ST
Z	Zero bit. Z = 1 means the result of the previous operation was zero. (ADDC and SUBC can clear Z, but they cannot set it.)							
N	Negative bit. N = 1 means the algebraically correct result is negative. (The actual result as stored can have an incorrect sign due to an overflow, but N reflects the algebraically correct sign.)							
V	Overflow bit. V = 1 means the magnitude of the algebraically correct result is too large to be correctly represented in the available space.							
VT	Overflow Trap bit. Anytime V gets set, VT gets set too. Subsequent operations can clear V, but VT can be cleared only by software.							

- C Carry bit. This is the carry-out of the most significant bit (MSB) of the result of any arithmetic operation. During left-shifts, the MSB moves into C and a 0 moves into the least significant bit (LSB). During right-shifts, the LSB moves into C and a 0 moves into the MSB.
- PSE Peripheral Transaction Server Enable (80C196KC only). PSE = 1 globally sets the PTS. Use the EPTS instruction to set the PSE bit. Reads 0 for the 8096BH and 80C196KB.
- I Interrupt enable bit. I = 0 disables all interrupts.
- ST Sticky bit. During right-shifts the LSB moves into C. If the previous value C was a 1, ST is set (see the SHR and SHRA instructions).

Copy the TSP data to a file named TSP.DAT in the
the root of the hard drive. The file should be
named TSP.DAT and should be in the root of the
hard drive. The file should be in the root of the
hard drive.

Proposed Transducer Model: 800000 (m/s)
The model is the 800000 (m/s) model. The model
is the 800000 (m/s) model.

Proposed Transducer Model: 800000 (m/s)
The model is the 800000 (m/s) model. The model
is the 800000 (m/s) model.

2

Assembly Language Concepts



2.1 Assembly Language Components

Like any language, an assembly language consists of a vocabulary (a collection of terms) and a grammar (a set of rules that specifies how the terms are to be put together to express concepts or actions). The basic unit of the English language is the sentence. In order for a sentence to be meaningful, it must contain at least one of each of two types of terms, a subject and a verb, and these must be placed in a certain order. The basic unit of the assembly language (corresponding to the sentence in English) is the statement. There are three types of source lines: instructions, directives, and controls. Each type is described in a separate chapter of this manual. This chapter explains the types of terms that appear in statements.

In order for an English-language statement to be meaningful, it must contain certain types of terms, and can contain more terms and types of terms. Each must appear in a certain position in the statement relative to the other types of terms. In addition, each term in an English language sentence must be composed of valid elements: a set of the twenty-six letters of the English alphabet. A term containing letters from the Cyrillic alphabet or cuneiform characters would not be meaningful in English. If all the terms are composed of valid elements but the terms are placed in an inappropriate order, the sentence will be meaningless. Thus, the collection of valid English terms "industry spur of the is avarice" is not a meaningful sentence because the terms are placed in an incorrect order. When put in the correct order they represent David Hume's statement, "Avarice is the spur of industry."

Likewise, an assembly language statement consists of fields. Only the appropriate terms (i.e., those of the correct type) can appear in the given field, and each term must be composed of valid elements if the statement is to be meaningful. The statement format defines the correct order of the fields in a statement. The character set defines the set of valid elements that can appear in any term. If any field contains invalid characters, or if the fields are in an incorrect order, the statement will be meaningless. The assembler will indicate this to you by returning an error message. Furthermore, the error message will tell you in what way the statement is in error, and you must then correct it.

2.1.1 Character Set

ASM96 uses the ASCII character set. All ASCII characters are valid, but some of them have certain special semantic meanings for the assembler (unless they appear in a comment or in a string) that affects the way the assembler treats them. Numbers, delimiters, reserved words, and symbols all have special meaning. They are listed and defined as follows.

2.1.2 Numbers

ASM96 supports modulo-64K unsigned arithmetic (i.e., numbers are in the form of unsigned 16-bit integers). Numbers must be presented to the assembler as a series of valid digits followed by a base (also termed radix). The bases and their valid digits are presented in Table 2-1.

2.1.3 Long Constants

Long constants (32-bit unsigned numbers) can be specified by the DCL directive (see Chapter 4).

2.1.4 Real Numbers

Real numbers can be defined by the DCR directive (see Chapter 4). The format for real numbers is:

[sign] int_dec_digit.[frac_dec_digit][E[sign][exp_dec_digit]]

Where:

- sign* is the optional sign (+ or -). If not specified, + is assumed.
- int_dec_digit* are the decimal digits representing the integer part of the real number.
- frac_dec_digit* are the decimal digits representing the fractional part of the real number.
- exp_dec_digit* are the decimal digits representing the exponential part of the real number. The exponential part is optional. If not specified or if no digit is entered, the exponent part is assumed to be equal to 1 (i.e., 10 to the 0 th power). If the sign is not explicitly entered, it is assumed to be positive.

Table 2-1 ASM-96 Numbers and Bases

Number	Base
Binary	the digits 0 to 1, followed by B (or b).
Octal	the digits 0 to 7, followed by O or Q (or o or q).
Decimal	the digits 0 to 9, followed by D (or d) or a null.
Hexidecimal	the digits 0 to 9, the supplementary hexadecimal A to F, followed by H (or h).

NOTE: When no base is explicitly stated, the decimal base is assumed as a default.

The maximum absolute value of a real number is $3.37E38$. The minimum absolute value is $1.17E-38$.

A real value occupies four contiguous memory bytes, which can be viewed as 32 contiguous bits. The bits are divided into fields as follows:

sign (1 bit)	exponent (8 bits)	fraction (23 bits)
-----------------	----------------------	-----------------------

Where:

- the *sign* bit is 0 if the real value is positive or zero, or 1 if the real value is negative.
- the *exponent* field contains a value "offset" by 127 - in other words, the actual exponent can be obtained from the exponent field value by subtracting 127. This field is all 0's if the real value is zero.
- the *fraction* field contains the binary digits of the fractional part of the real value, when it is represented in "binary scientific" notation (see the following). This field is all 0's if the real value is zero.

The byte with the *lowest* address contains the least significant eight bits of the fraction field. The byte with the *highest* address contains the sign bit and the most significant seven bits of the exponent field.

The following examples illustrate these concepts.

Consider the following binary number, equivalent to the decimal value 10.25:

1010.01B

The period (.) in this number is a binary point. The same number can be represented as:

1.01001B * 2³

This is binary scientific notation, with the binary point immediately to the right of the most significant digit. The digits 01001 are the fractional part, and 3 is the exponent. This value is represented as follows:

- The sign bit is 0, since the value is positive.
- The exponent field contains the binary equivalent of $127 + 3 = 130$.
- The leftmost digits of the fraction field is 01001, and the remainder of this field is all 0's.

The complete 32-bit representation is:

0 10000010 010010000000000000000000

and the contents of the four contiguous memory bytes is as follows:

highest address:	01000001
	00100100
	00000000
lowest address:	00000000

Note that the most significant digit is not actually represented, since by definition it is a 1 unless the real value is zero. If the real value is zero, the entire 32-bit representation is all 0's.

For a second example, consider the fraction 1/16, or 0.0625. In binary, this is:

0.0001B

In binary scientific we have:

1.0000B * 2^{-4}

The actual exponent, -4, is represented as 123 (i.e., $127 - 4$), and the fraction field contains all 0's.

The largest possible value for a valid exponent field is 254, which corresponds to an actual exponent of 127. The largest possible absolute value for a positive or negative real value is therefore:

$$1.111111111111111111111111111111B * 2^{127}$$

or approximately 3.37×10^{38}

The lowest permissible exponent field value for a non-zero real value is 1, which corresponds to an actual exponent of -126. The smallest possible absolute value for a positive or negative real value is therefore:

1.0B * 2⁻¹²⁶

or approximately $1.17 \cdot 10^{-38}$

2.1.5 Delimiters

Delimiters serve to separate and to terminate terms in an ASM96 statement just as spaces serve to delimit words in an English sentence. The English language sentences "The future is now here" and "The future is nowhere" are identical except for the positioning of the delimiters, but have very different meanings. Likewise, the position of a delimiter can radically affect the meaning of a statement to the assembler. The complete set of ASM96 delimiters and their definitions are presented in Table 2-2.

Table 2-2 ASM96 Delimiters

Character(s)	Definition
Space	One or more spaces can be entered. Spaces serve as field separators or symbol delimiters.
,	Commas separate operands.
'	Single quotes serve to delimit character strings.
()	Parentheses delimit expressions.
[]	Square brackets enclose base or index registers in instruction operands.
< >	Angle brackets enclose macro arguments.
%	The percent sign indicates that the following macro argument is to be evaluated as an expression before replacing the formal parameter.
!	The exclamation mark is used as a literalizer in macro processing.

Table 2-2 ASM96 Delimiters (continued)

Character(s)	Definition
&	The ampersand is used to separate formal parameters from adjacent text.
LF	The Line Feed (ASCII 0AH) is the line terminator.
CR	The Carriage Return (ASCII 0DH) can optionally precede the Line Feed.
HT	The Horizontal Tab is treated as a space.
;	The semicolon delimits the beginning of a comment.
::	Two successive semicolons mark the start of a comment in a macro definition that is to be suppressed when the macro is expanded.
#	The pound sign prefixes an immediate value in immediate addressing mode.
:	The colon separates the name from the rest of a statement.

2.1.6 Symbols

Symbols are user-defined tags that can be used to represent memory addresses, constants, macros, etc. A symbol contains up to thirty-one (31) characters. The first character must be a letter or a special character. The special characters are the question mark (?) and the underscore (_). The remaining characters can be letters, special characters, or the digits 0 to 9. The case of letters (upper or lower) within symbols is ignored by the assembler. Thus, it treats `io_control` the same as `IO_CONTROL` or `Io_CoNtRoL`. The special characters are very useful in creating more meaningful symbol names, especially since the hyphen is not valid in a symbol name. Thus, `DATA_Buffer_START` or `DATA?Buffer?START` are more easily recognizable than is `DATABufferSTART`.

2.1.7 Reserved Words

Reserved words are the names that have specific meaning in the assembly language and thus are not available for user definition. Instruction mnemonics, assembler and macro directives, and the predefined symbol `STACK` are all reserved words. The complete set of ASM96 reserved words is shown in Appendix D. In the appendix, the reserved words appear in all uppercase. However, the assembler ignores case in determining if a term is a reserved word. Thus, `stack` is a reserved word just as `STACK` is.

2.1.8 Assembler-generated Symbols

These are symbols generated for local macro symbols. They all have the form `??dddd` where *d* is a decimal digit. Since using such a symbol as a user-defined symbol can duplicate an assembler-generated symbol, you should treat assembler-generated symbols as you would reserved words and refrain from creating conflicting names.

2.1.9 Location Counter

The current location within the active segment of the program is referenced in a statement by the dollar sign (\$).

2.1.10 Strings

Strings are a sequence of printable ASCII characters. A string is delimited by a beginning and an ending single quote ('). Characters that appear within the string delimiters are treated literally by the assembler and do not possess their usual special meaning. Thus, 'STACK' is the character string S-T-A-C-K and not the reserved word STACK, 'FFH' is the character string F-F-H and not the hexadecimal number FF, and '\$' is the literal dollar sign and not the location counter.

If the single quote mark is itself to be the initial or the terminal character of the string, two single quote marks must appear. Thus, you would use `''tis the season'` to delimit the string 'tis the season. Entering only one quote mark would not preserve the initial single quote as part of the string.

Within strings case is significant. Thus, 'Bigfoot' is not the same string as 'BIGFOOT'.

Strings can be used as operands to the DCB directive (see Chapter 4) and as operands in expression. When used in a DCB directive, the string length can be anywhere from zero (i.e., a null string) to 255. When a string is used as an operand in an expression, the string can contain only one or two characters.

2.2 Expressions and Basic Operands

Basic operands are numbers, user defined symbols, the predefined symbol (\$) for the active location counter, the predefined variable (STACK) for the bottom of the stack, and strings of one or two characters. An expression consists of one of the following items:

- a single basic operand
- a single expression acted upon by a prefix unary operator
- two expressions by an infix operator

A prefix unary operator is one that is meaningful if it precedes a single basic operand. Thus, -25 is meaningful, as is NOT A. The text /32 and >Limit are not meaningful. In order for the latter two operations to be meaningful, two basic operands are required. Thus, 166/32 is meaningful, as is Total > Limit.

Expressions are used to define constants. Expressions are either defined at assembly time (absolute constants) or at relocation time (termed relocatable expressions). When any expression is defined, each operand in the expression is evaluated as a 16-bit unsigned integer (i.e., an integer in the range of 0 to 0FFFFH).

2.2.1 Basic Operands

There are five kinds of basic operands as follows:

Symbols	User defined symbols can represent a memory address, a constant, or an external name that is defined in some other module as either a constant or an address.
Numbers	Scalar quantities are represented as numbers. They can be expressed in binary, octal, decimal or hexadecimal. When no base is explicitly stated, decimal is assumed.
\$	The dollar sign (\$) is a predefined symbol that represents the present value of the active location counter.
STACK	The predefined variable STACK represents the bottom of the stack.
Strings	Only one- and two-character strings are allowed to serve as operands in expressions. A one-character string is always interpreted by the assembler as a one-byte constant and set equal to the ASCII value of the character. A two-character string is interpreted as a word. The low order byte of the word is set equal to the ASCII value of the first character of the string and the high order byte is set equal to the ASCII value of the second character of the string.

2.2.2 Attributes of Expression Operands

Every expression has three attributes as follows:

- type
- relocatability
- value

2.2.2.1 Type

The type attribute states the type of segment (register, code, data, stack, overlay or null) to which the operand belongs. Thus an operand defined in the code segment is of type code. For each of the five kinds of basic operands and for unary and binary expressions, type is defined according to the following rules:

Symbols	For user defined symbols, the type depends upon how the symbol was defined (i.e., as a label or via a directive such as EQU, SET, or EXTRN). If the symbol was defined as a label or with the EXTRN directive, the type is the type of the active segment. If it was defined via a SET or EQU directive, its type is the type of the right-hand side expression.
Numbers and strings	For numbers and for strings the segment type is null.
The location counter symbol (\$)	For \$, the type is the type of the active segment.
Bottom of the stack (STACK)	For STACK, the type is always stack.
Expressions resulting from unary operations	+ is the only unary operation that can be applied to memory address. The type of the resulting expression is the type of the memory address. When any unary operation is applied to numbers, the type of the resulting expression is always null.
Expression resulting from binary operations	The only valid expressions containing a number and a memory address are the following: memory_address - number memory_address + number number + memory_address. The resulting type is always that of the memory address.

When any binary operation is applied to two numbers, the resulting type is always null.

The only binary operations that can only be applied to two memory addresses are subtraction and comparison, and both memory addresses must belong to the same segment (i.e., they both have the same type and relocatability). The resulting expression is always null.

2.2.2.2 Relocatability

The relocatability attribute defines whether the operand is absolute, relocatable, or external.

2.2.2.3 Value

The value attribute is always a 16-bit two's complement number. If the type is null (i.e., you are dealing with a pure number), the value of the expression is a typeless number. If the segment is anything other than null, the value is a memory address. The precise address cannot be defined until RL- time in the case of relocatable expressions.

2.2.3 Absolute Expressions

Absolute expressions are expressions that are evaluated at assembly time. Absolute expressions can appear anywhere in a program. Table 2-3 lists the valid operators for absolute expressions according to their precedence (i.e., the order in which they are evaluated).

NUL is a unary operator used in macro definition to test for the existence of a macro argument. For more information on macro definitions, refer to Chapter 5 of this manual.

NUL and the relational operators produce a result that is interpreted as being either logical true (represented by 0FFFFH) or logical false (represented by 0H).

**Table 2-3 Valid Operators for ASM96 Absolute Expressions
Listed by Precedence (High to Low)**

Operator(s)	Function (if not conventional)	Precedence
NUL	test argument in Macro	Highest
*, /, MOD, SHR, SHL	multiply, divide, MOD (take quotient only), shift right, shift left	
+, -	addition, subtraction (unary and binary)	
EQ, NE, GT, GE, LT, LE	relational operators equal, not equal, greater than, greater than or equal, less than, less than or equal	
=, <>, >, >=, <, <=	symbolic representations of the relational operators listed above (their precedence is the same)	
NOT	logical inversion (bit by bit)	Lowest
AND	logical conjunction (bit by bit)	
OR, XOR	logical disjunction and exclusive OR (bit by bit)	

An operator composed of letters, such as NOT or XOR, must be separated from an adjacent symbol by at least one space. Thus, for the operator XOR and the symbol SIZE, XOR SIZE or XOR SIZE are both valid, while XORSIZE is not and will result in an error.

As a general rule, in order for an expression to be an absolute expression (i.e., completely computable at assembly time) all operands in the expression must be absolute (i.e., their location attribute must be absolute). However, an expression containing subtraction or any of the relational operators and having both operands relocatable but of the same segment type (i.e., the type attribute is the same) can be evaluated completely at assembly time, and the expression is treated as being absolute and of null type. This is because in the evaluation, the segment types, being the same, cancel out.

2.2.4 Relocatable Expressions

Relocatable expressions are those expressions that are evaluated only at RL time. The only valid relocatable expressions are those that are composed of either one relocatable or one external symbol plus a two's complement constant (i.e., a constant computed at assembly time). Thus, the permissible forms are a relocatable or an external symbol plus or minus a constant or a constant plus a relocatable or an external symbol. A constant minus a relocatable or an external symbol is not permitted. Thus, RELO-5, EXT0+3, and 12+RELO are valid, but 13-EXT0 is not.

Relocatable expressions that are based upon an external symbol can appear only when specifying operands to statements that generate object code, i.e., only in machine instructions and code definition directives.

2.3 Statement Format

The assembly source program statement consists of four fields. Each field is optional. In order from left to right, the fields are the label field, the operation field, the operand field, and the comment field. Thus, the correct syntax for a statement can be represented as

`[Label:|Name] [Operation] [Operand,...] [;Comment] <CR,LF>`

where the brackets and other symbols have the meaning described in the notational conventions of this manual. Each of the fields will now be described.

The label field	contains either a label or a name. Note that structurally, a label differs from a name in that a label must be followed by a colon. Functionally, a label can appear only before machine instructions, macro calls, storage reservation directives, code definition directives, and empty statements. The value of a label is the value of the program counter at the beginning of the instruction, i.e., after achieving the required alignment. The value of a label of an empty statement is the value of the program counter at the beginning of the next instruction. The purpose of a label is to quickly get you to a position within the code. A name is used to associate a set of attributes explicitly with a tag, but without defining a specific storage (e.g., macro definitions and EQU directives).
The operation field	contains a machine instruction mnemonic code, a directive mnemonic, or a user-defined macro name. The contents of the operation field determine which instruction the machine is to execute (e.g., jump, add, etc.).
The operand field	contents and syntax depend on the specific operation in the statement. Some operations take more than one operand. The contents of the operand field specify on what the operation is to be performed. The operand(s) can be simple names, expressions involving names and numeric operators, or an address expression that defines the method (e.g., indirect or indexed) by which another operand is to be accessed. Only certain operands are allowed with certain operations.

The comment field allows you to place an English language (or any language, existing or invented, that can be written using the ASM96 character set) description on each statement. The comment is ignored by the assembler (except that it does appear in the listing file). Therefore, special characters appearing in the comment field do not result in the assembler performing any special function. The comment field must begin with a semicolon. All text appearing between the semicolon and the next line feed is taken as a comment.

2.3.1 Additional Statement Rules

1. There are no continuation lines in ASM96. All parameters in a given statement must be placed on the same line.
2. Each statement must be terminated by line feed. A carriage return can optionally precede the line feed.
3. Blank characters (spaces) can appear anywhere in a statement except within numbers, names, and special symbols. Thus, DATA BUFFER is two names, not one, and ; ; is not the same as ; ;.
4. An empty line (i.e., one containing only blanks, comments, or null) can appear anywhere within a program.
5. A line that contains only a label can appear only within segments. It cannot appear before the first x SEG directive.

2.4 Program Format

Just as English sentences can be put together to form a memorandum directing someone to do something, so assembly language statements are put together to form programs that direct the microcontroller to do something. Just as the standard office memo has a particular format that makes it easily understandable by the recipient, so the assembler has a set of rules that govern the format of its input. The format for assembler-units, the units of source code presented to the assembler for assembly, is presented below. Refer to the notational conventions of this manual for the meaning of the brackets and other symbols used.

```
[ { Primary Control Line(s) } ]  
[ MODULE Directive ]  
[ Statement(s) ]  
END
```

Generally, an assembly unit begins with one or more primary control lines, although it is permissible to have no primary control line. Refer to Chapter 6 of this manual for a detailed discussion of the form and function of ASM96 controls. It can be stated briefly here that the controls control the operation of the assembler as it encounters statements.

The ASCII SUB, or `<Ctrl>Z`, character can appear as the last character in a file. The ASCII value of this character is 1A hexadecimal. This character is used by DOS for an end-of-file marker and all characters following the `<Ctrl>Z` are ignored by the ASM96 assembler.

The `MODULE` directive allows you to assign attributes to the module being assembled. See Chapter 4 for a description of these attributes.

Many statements are meaningful and valid only if they appear in a specific context. Thus, a machine instruction can appear only within a code segment. The rules governing the statements are stated in the description of the statements.

Figure 2-1 presents a graphic representation of a typical assembly-unit.

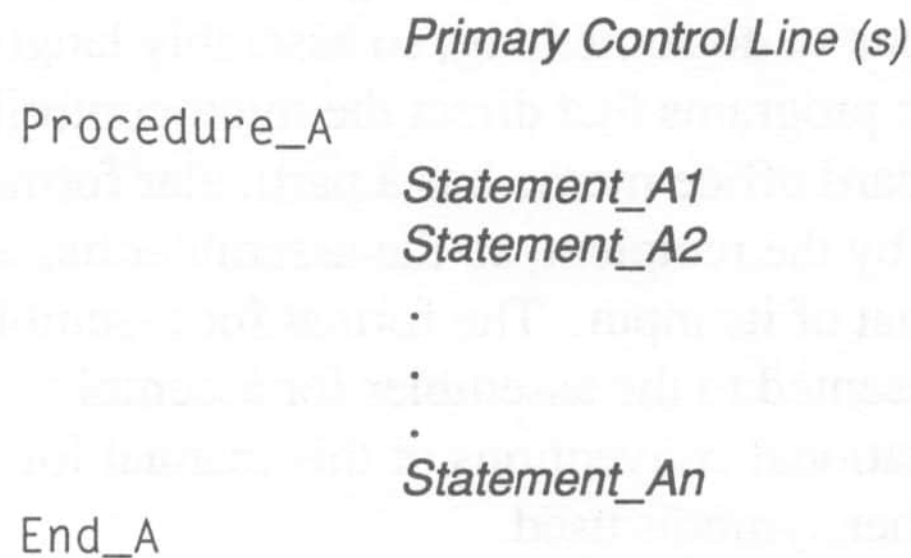


Figure 2-1 Typical Assembly-unit Program Structure

2.5 Segments

A segment is a piece of memory defined within a module or a section of memory used by your programs. Each segment has a specific function. The five segment types in ASM96 are

- the non-overlayable register segment
- the overlayable register segment
- the data segment
- the stack segment
- the code segment

2.5.1 Register Segment (Overlayable and Non-overlayable)

The register segment is a portion of memory allocated in the REGISTER section. Variables belonging to the register segment can be used as registers throughout your program and are referred to as registers. Only storage reservation can be specified within this segment. If a register segment is declared as overlayable (via the OSEG directive, Chapter 4) the RL96 relocater and linker can decide to overlay it with other overlayable segments if they are not active simultaneously. Non-overlayable segments can not be overlaid.

2.5.2 Data Segment

The data segment is a portion of memory allocated in a RAM section.

Variables belonging to this segment can be used as memory-referenced operands (using direct, indirect, indexed, or based addressing modes). Only storage reservation can be specified within this segment.

2.5.3 Stack Segment

The stack segment is a portion of memory allocated in a RAM section.

Variables belonging to this segment can be referenced using based addressing based upon the stack pointer or any other register used as a pointer. Neither code nor storage can be defined within this segment, which is solely for efficient stack use. To reference a static variable within the stack segment, specify the (negative) offset from the bottom of the stack (the initial value of the stack pointer).

2.5.4 Code Segment

The code segment is a portion of memory allocated in a ROM section. This segment is used for storing program constants and code. Variables belonging to the code segment can be used as memory-referenced operands (using direct, indirect, based, or indexed addressing modes) or as the targets of branch and call instructions.

2.6 Absolute and Relocatable Segments

You can define a segment as either absolute or as relocatable. When you define a segment as absolute, a new segment is created. When you define a segment as relocatable, the previous relocatable segment of that type (if any exists) is continued. Thus, the object module contains at most one relocatable segment of each type and an unlimited number of absolute segments. Absolute segments do not require relocation, but they can contain public symbols and references to other segments or modules.

The assembler attaches an alignment attribute to each non-empty relocatable segment. This attribute determines whether the segment starts on a byte, a word, or a double word boundary, thus ensuring proper alignment of all symbols within the given segment. When the object files are linked, the segments are placed in the required section of memory and aligned according to their alignment attribute. Because of the special character of stack segments, all stack segments are placed contiguously in one section of RAM.

3

Instruction Set



3.1 Introduction

This chapter contains complete documentation for all of the 8096 and 80C196 instructions. The instructions are listed in alphabetical order by mnemonic and operands, and an example is given for each instruction. Note that the effect on the program counter (PC) is not always shown. All instructions increment the PC by the number of bytes in the instruction.

The operands entry for each instruction briefly indicates the range of values and segment types permitted in each operand. For a complete description of the limits of any operand see Chapter 2 of this manual. In general, the operand's name will identify what section to consult.

Table 3-1 provides a list of the abbreviations and instructions used throughout the remainder of Chapter 3, and their meanings.

Table 3-1 Abbreviations and Notations Used in Instruction Pages

Abbreviation, Notation	Meaning
baop	byte address moded operand
BEA	bytes of effective address
bitno	bit number
breg	byte register
cadd	code address
CEA	cycles of effective address
cnt	count
DEST	destination
disp	displacement

Table 3-1 Abbreviations and Notations Used in Instruction Pages (continued)

Abbreviation, Notation	Meaning
disp-hi	high order byte of displacement
disp-low	low order byte of displacement
lreg	long word register
offs-hi	high order byte of offset
offs-lo	low order byte of offset
PC	program counter
SP	stack pointer
SRC	source
val-hi	value of high order byte
val-lo	value of low order byte
waop	word address moded operand
wreg	word register
AND	logical AND
NOT	logical complement
OR	logical OR
XOR	logical exclusive OR
+	plus
-	minus
/	divide
*	multiply
X	the contents of X
(X)	the contents of the contents of X
=	equal to
<>	not equal to
<	less than
>	greater than
←	is replaced by

3.2 Program Status Word

The program status word (PSW) contains information on the current status of the MCS-96 microcontroller and the condition codes describing the results of the most recently executed instruction. Status bits 8-15 are not directly addressable but can be accessed by means of special instructions. The PSW bits are defined in Table 3-2. Refer to these definitions for complete information on the flags that are affected by the instructions.

Table 3-2 Program Status Word Flags

Bits	Flag	Description
0-7		Interrupt Mask.
8	ST	The STicky bit is set to indicate that during a right shift a one has been shifted into and then out of the Carry flag. This flag can be used with the Carry flag to determine rounding.
9	I	When cleared, the global Interrupt disable bit disables all interrupts except NMI.
10	PSE	(80C196KC only) When set, it enables the Peripheral Transaction Server. Otherwise, cleared when writing to the PSW for compatibility with future products.
11	C	The Carry flag is set to indicate the state of the arithmetic carry from the most significant bit of the ALU for an arithmetic operation, or the state of the last bit shifted out of an operand for a shift. Arithmetic Borrow after a subtract operation is the complement of the C flag (i.e., if the operation generated a borrow, C=0.)
12	VT	The oVerflow Trap flag is set when the V flag is set, but it is only cleared by the CLRVT, JVT and JNVT instructions. This allows testing for overflows in a group of operations instead of after each operation.
13	V	The oVerflow flag is set to indicate that an operation has generated a result which is outside the range for the destination data type.
14	N	The Negative flag is set to indicate that an operation has generated a negative result. Note that the N flag is in the algebraically correct state even if an overflow occurs. For shift operations, including the normalize operation and all three forms (SHL, SHR, SHRA) of byte, word and double word shifts, the N flag is set to the same value as the most significant bit of the result. This is true even if the shift count is 0.
15	Z	The Zero flag is set to indicate that an operation has generated a result equal to zero. The instructions SUBC(B) and ADDC(B) can clear the Z flag but cannot set it. This makes it easier to perform double word arithmetic, since a zero in the high word does not set the zero flag.

3.3 Instruction Operands

3.3.1 Referring to a Register: breg, wreg, lreg

The register file of the MCS-96 contains 256 byte registers, of which the first 24 are special purpose, mainly memory mapped I/O, two are additional registers that can be used as general purpose registers or as the stack pointer, and the other 230 are general purpose registers. The registers can also be viewed as 128 word registers or 64 long (i.e., double-word) registers. Thus the word register at location 20H occupies the same place as byte registers 20H (which holds the low-order byte of the word), and byte register 21H (which holds the high-order byte of the word). The long register at location 20H overlaps byte registers 20H-23H, with the low-order byte at location 20H, the next higher-order byte at location 21H, and the high-order byte at location 23H.

When an instruction refers to a byte register (acronym breg), the operand can specify any expression in the range of 0-255 that refers either to the register segment or to no segment.

When an instruction refers to a word register (acronym wreg), the operand can specify any even expression in the range of 0-255 that refers either to the register segment or to no segment.

When an instruction refers to a long register (acronym lreg), the operand can specify any expression in the range of 0-255 that is divisible by 4 and that refers either to a register segment or to no segment.

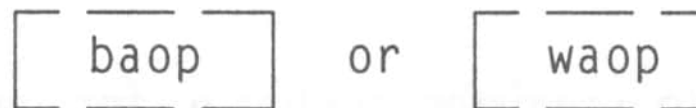
3.3.2 Referring to a General Memory Location: baop, waop

Many instructions allow one of their operands to be a general memory location that is referred to by one of the addressing modes of the 8096.

When an instruction refers to a byte memory location (acronym baop), the operand can specify any expression in the range of 0-65535 that can refer to any segment.

When an instruction refers to a word memory location (acronym waop), the operand specified must be even (all word operands in the 8096 must be on word boundary).

In the description of the instructions, such operands are called



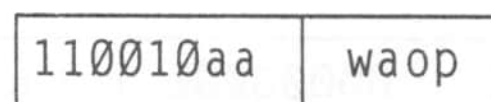
In the opcode byte there are two bits, which appear as aa, which partially determine the addressing mode. Thus the information on the PUSH instruction will appear as follows:

.

Format

PUSH waop

Opcode Pattern



Bytes:	1+BEA
Cycles: Onchip Stack:	8+CEA
Offchip Stack:	12+CEA

.

The number of bytes and the number of cycles needed for the instruction also depends on the addressing mode. The following section describes each addressing mode, the value of the aa field, the width (acronym BEA) and format of the baop or waop field, and the additional number of cycles required to execute the instruction (acronym CEA).

3.3.3 Direct Addressing

To use direct addressing, specify any label or expression as the operand. For a waop, the operand specified must be on word boundary. The assembler chooses among the two possible addressing modes to accomplish direct addressing.

If the operand refers to a register and has no forward reference, register addressing is used:

	baop	waop				
Pattern	<table border="1"><tr><td>xxxxxx00</td><td>breg</td></tr></table>	xxxxxx00	breg	<table border="1"><tr><td>xxxxxx00</td><td>wreg</td></tr></table>	xxxxxx00	wreg
xxxxxx00	breg					
xxxxxx00	wreg					
Additional Bytes:	1	1				
Additional Cycles:	0	0				

Otherwise, long indexing based on register 0 is used:

Pattern	xxxxxx11	00000001	loc-low	loc-high
Additional Bytes:				3
Additional Cycles:	Onchip Memory:			3
	Offchip Memory:			8

Examples

```
PUSH ALPHA
PUSH GAMMA+1
```

NOTE

It is important that such references to registers will not contain forward references; otherwise the longer and slower code will be generated.

3.3.4 Immediate Addressing

To use immediate addressing, specify a number (#), followed by the immediate expression. For baop, the expression must be in the range of -256 to +255. For waop, the expression can be in the range of -65536 to +65535.

baop

Pattern

xxxxxx01	value
----------	-------

Additional Bytes: 1
Additional Cycles: 0

or

waop

Pattern

xxxxxx01	val-lo	val-hi
----------	--------	--------

Additional Bytes: 1
Additional Cycles: 2

Examples

CMPB ALPHA, #2
PUSH #GAMMA

3.3.5 Indirect Addressing

To use indirect addressing, specify a word register within square brackets. The value of that register specifies the address of the required operand. Since a word register has an even address, only seven bits are needed to store its address: the least significant bit is always zero. That bit in the instruction is used to distinguish between indirect addressing with or without auto increment.

For waop, the run-time value of that register must be even.

Pattern

xxxxxx10	wreg 0
----------	--------

Additional Bytes: 1
 Additional Cycles: Onchip Memory: 2
 Offchip Memory: 7

Examples

```
LD  ALPHA,[BETA]
ADD ALPHA,BETA,[GAMMA]
```

3.3.6 Indirect Addressing with Auto Increment

To use indirect addressing with auto increment, specify a word register within square brackets, followed by a + sign. The value of that register specifies the address of the required operand. After the operand is fetched, the register is incremented by 1 (for baop) or 2 (for waop).

Since a word register has an even address, only seven bits are needed to store its address: the least significant bit of the address is always zero. That bit in the instruction is used to distinguish between indirect addressing with or without auto increment, and is set to 1 to indicate auto increment.

For waop, the run-time value of that register must be even.

Pattern

xxxxxx10	wreg 1
----------	--------

Additional Bytes: 1
 Additional Cycles: Onchip Memory: 3
 Offchip Memory: 8

Examples

```
CMPB ALPHA,[BETA]+
SUB  ALPHA,BETA,[GAMMA]+
```

3.3.7 Based or Indexed Addressing

This addressing mode is used when part of the address is known at compile time, and part is computed at run time. This is useful both for indexing into arrays, where the base address is known at compile time and the offset of the desired element is computed dynamically, and to reference based data structure, where the base address is dynamic, but the offset of the desired member in the structure is known at compile time.

To use based or indexed addressing, specify any label or expression as the offset, followed by a word register within square brackets. The specified offset is added to the value of that register to get the address of the required operand.

For waop, the offset must be even, and the run-time value of that register must be even.

Since a word register has an even address, only seven bits are needed to store its address: the least significant bit of the address is always zero. That bit in the instruction is set to zero to indicate short indexing and set to one to indicate long indexing. The assembler automatically chooses between those two type of indexing.

If the offset is not relocatable, has no forward reference, and is in the range of -128 to +127 inclusive, short indexing is used:

Pattern

xxxxxx11	wreg 0	offset
----------	--------	--------

Additional Bytes: 2

Additional Cycles: Onchip Memory: 2

Offchip Memory: 7

Otherwise, long indexing is used:

Pattern

xxxxxx11	wreg 1	offs-low	offs-hi
----------	--------	----------	---------

Additional Bytes: 3

Additional Cycles: Onchip Memory: 3

Offchip Memory: 8

Examples

```
LD    ALPHA,2[BETA]
CMP   ALPHA,BETA[GAMMA]
```

3.3.8 Referring to a Code Address: cadd

All jumps and calls refer to another code address (acronym cadd) as their operand. This can be any expression that refers either to the code segment or to no segment. If the expression is not relocatable, its value must be at least 100H. In the instruction, the destination address is stored as an offset from the end of the current instruction. This offset must be in the range of -128 to +127 in all non-generic conditional jumps; it must be in the range of -1024 to +1023 for SJMP and SCALL; and it can be anything for LJMP, LCALL, and the generic instructions. When using generic instructions, the assembler will choose the proper instruction or instruction sequence depending on the offset to the desired label. If the operand contains forward references, the assembler will use safe but suboptimal heuristics. For external reference, the long form of the branch will be used.

3.3.9 Specifying a Count for a Shift: cnt/breg

A count for all shifts can be specified in one of two ways: either as an immediate operand in the range of 0 to 15, as in

```
SHL   ALPHA,#3
```

or as one of the registers in the range 16 to 255, as in

```
SHR    ALPHA,BETA
```

Both immediate count and the address of the count register are stored immediately following the opcode byte, and the 8096 distinguishes between them by the value of the field. Since the general purpose registers start at location 18H or 24, it is not a real limitation.

3.3.10 Specifying a Bit Operand: `bitno`

Instructions which require bit operands such as BBC, BBS, JBC, and JBS only allow `bitno` to be greater than or equal to zero and less than seven. If `bitno` is greater than seven, the assembler will modify `breg` by calculating the result of `bitno` modulo 8. The quotient is added to the base address of `breg` and the remainder is `bitno`. The assembler will allow a comma or a period after `breg`.

Example:

```
RSEG at 30H
bitfields: DSL 1           ;32 bits for semaphores

CSEG
BBS          bitfields.28,jump_proc
```

With this example, the base address of `bitfields` (`breg`) is 30H. The assembler will calculate 28 modulo 8. The quotient is 3, remainder 4. The assembler then adds the quotient to the base address and treats the remainder as the `bitno` resulting in `breg` referring to address 33H bit 4.

3.4 Description of Instructions

An alphabetical list of instructions and their descriptions make up the remainder of Chapter 3. For information about instruction size and execution times, see Appendix A.

ADD (Two Operands)

Format ADD wreg,waop

Operation The sum of the two word operands is stored into the destination (leftmost) operand.

$$(DEST) \leftarrow (DEST) + (SRC)$$

Opcode Pattern

011001aa	waop	wreg
----------	------	------

Flags Affected C, N, V, VT, Z

Examples

```
ADD DELTA, #500
ADD ALPHA, BETA
ADD ALPHA, [GAMMA]+
```

ADD (Three Operands)

Format

ADD wreg,wreg,waop

Operation

The sum of the second and third word operands is stored into the destination (leftmost) operand.

$$(DEST) \leftarrow (SRC1) + (SRC2)$$

Opcode Pattern

010001aa	waop	wreg	wreg
----------	------	------	------

Flags Affected

C, N, V, VT, Z

Examples

```
ADD    DELTA, TIMER1, #2
ADD    ALPHA, BETA, GAMMA
ADD    ALPHA, DELTA, 10[GAMMA]
```

ADDB (Two Operands)

Format

ADDB breg,baop

Operation

The sum of the two byte operands is stored into the destination (leftmost) operand.

$(\text{DEST}) \leftarrow (\text{DEST}) + (\text{SRC})$

Opcode Pattern

011101aa	baop	breg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

ADDB DELTA, #2
ADDB ALPHA, BETA
ADDB ALPHA, [GAMMA] +

ADDB (Three Operands)

Format

ADDB breg,breg,baop

Operation

The sum of the second and third byte operands is stored into the destination (leftmost) operand.

$$(DEST) \leftarrow (SRC1) + (SRC2)$$

Opcode Pattern

010101aa	baop	breg	breg
----------	------	------	------

Flags Affected

C, N, V, VT, Z

Examples

```
ADDB    DELTA, TIMER1, #2
ADDB    ALPHA, BETA, GAMMA
ADDB    ALPHA, DELTA, 10[GAMMA]
```


ADDC

Format

ADDC wreg,waop

Operation

The sum of the two word operands and the carry flag (0 or 1) is stored into the destination (leftmost) operand.

$$(DEST) \leftarrow (DEST) + (SRC) + C$$

Opcode Pattern

101001aa	waop	wreg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

ADDC DELTA, #500
ADDC ALPHA, BETA
ADDC ALPHA, [GAMMA]+

ADDCB

Format

ADDCB breg,baop

Operation

The sum of the two byte operands and the carry flag (0 or 1) is stored into the destination (leftmost) operand.

$$(DEST) \leftarrow (DEST) + (SRC) + C$$

Opcode Pattern

101101aa	baop	breg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

ADDCB DELTA, #2
ADDCB ALPHA, BETA
ADDCB ALPHA, [GAMMA] +

AND (Two Operands)

Format

AND wreg,waop

Operation

The two word operands are ANDed, the result having a 1 only in those bit positions where both operands had a 1, with 0's in all other bit positions. The result is stored into the destination (leftmost) operand. The carry and overflow flags are reset to 0.

$(\text{DEST}) \leftarrow (\text{DEST}) \& (\text{SRC})$

$C \leftarrow 0$

$V \leftarrow 0$

Opcode Pattern

011000aa	waop	wreg
----------	------	------

Flags Affected

C, N, V, Z

Examples

AND DELTA, #500
AND ALPHA, BETA
AND ALPHA, [GAMMA] +

AND (Three Operands)

Format

AND wreg,wreg,waop

Operation

The second and third word operands are ANDed, the result having a 1 only in those bit positions where both operands had a 1, with 0's in all other bit positions. The result is stored into the destination (leftmost) operand. The carry and overflow flags are reset to 0.

$$(DEST) \leftarrow (SRC1) \& (SRC2)$$
$$C \leftarrow 0$$
$$V \leftarrow 0$$

Opcode Pattern

010000aa	waop	wreg	wreg
----------	------	------	------

Flags Affected

C, N, V, Z

Examples

AND DELTA, TIMER1, #2

AND ALPHA, BETA, GAMMA

AND ALPHA, DELTA, 10[GAMMA]

ANDB (Two Operands)

Format

ANDB breg,baop

Operation

The two byte operands are ANDed, the result having a 1 only in those bit positions where both operands had a 1, with 0's in all other bit positions. The result is stored into the destination (leftmost) operand. The carry and overflow flags are reset to 0.

$(\text{DEST}) \leftarrow (\text{DEST}) \& (\text{SRC})$

$C \leftarrow 0$

$V \leftarrow 0$

Opcode Pattern

011100aa

baop

breg

Flags Affected

C, N, V, Z

Examples

ANDB DELTA, #10100000B

ANDB ALPHA, BETA

ANDB ALPHA, [GAMMA]+

ANDB (Three Operands)

Format

ANDB breg,breg,baop

Operation

The second and third word operands are ANDed, the result having a 1 only in those bit positions where both operands had a 1, with 0's in all other bit positions. The result is stored into the destination (leftmost) operand. The carry and overflow flags are reset to 0.

$$(DEST) \leftarrow (SRC1) \& (SRC2)$$
$$C \leftarrow 0$$
$$V \leftarrow 0$$

Opcode Pattern

010100aa	baop	breg	breg
----------	------	------	------

Flags Affected

C, N, V, Z

Examples

```
ANDB    DELTA, SBUF, #01111111B
ANDB    ALPHA, BETA, GAMMA
ANDB    ALPHA, DELTA, 10[GAMMA]
```

BBC

Format

BBC breg,bitno,cadd

Operation

This instruction is a generic branch on a bit-clear condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JBC breg,bitno,cadd

Medium Sequence

JBS breg,bitno,\$+5
SJMP cadd

Long Sequence

JBS breg,bitno,\$+6
LJMP cadd

Flags Affected

None

Examples

BBC SBUF, 7, EVEN_PARITY

BBS

Format

BBS breg,bitno,cadd

Operation

This instruction is a generic branch on a bit-set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JBS breg,bitno,cadd

Medium Sequence

JBC breg,bitno,\$+5
SJMP cadd

Long Sequence

JBC breg,bitno,\$+6
LJMP cadd

Flags Affected

None

Examples

BBS SBUF, 7, ODD_PARITY

BC

Format

BC cadd

Operation

This instruction is a generic branch on carry set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JC cadd

Medium Sequence

JNC \$+4
SJMP cadd

Long Sequence

JNC \$+5
LJMP cadd

Flags Affected

None

Examples

BC TARGET_LABEL

BE

Format

BE cadd

Operation

This instruction is a generic branch on equal (to 0) condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JE cadd

Medium Sequence

JNE \$+4
SJMP cadd

Long Sequence

JNE \$+5
LJMP cadd

Flags Affected

None

Examples

BE TARGET_LABEL

BGE

Format

BGE cadd

Operation

This instruction is a generic branch on greater or equal condition (i.e., when the negative flag is 0). The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JGE cadd

Medium Sequence

JLT \$+4
SJMP cadd

Long Sequence

JLT \$+5
LJMP cadd

Flags Affected

None

Examples

BGE TARGET_LABEL

BGT

Format

BGT cadd

Operation

This instruction is a generic branch on greater-than condition (i.e., both the negative flag and the zero flag are 0). The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JGT cadd

Medium Sequence

JLE \$+4
SJMP cadd

Long Sequence

JLE \$+5
LJMP cadd

Flags Affected

None

Examples

BGT TARGET_LABEL

BH

Format

BH cadd

Operation

This instruction is a generic branch on higher (i.e., the carry flag is set, but the zero flag is not, which is greater than in unsigned arithmetic) condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JH cadd

Medium Sequence

JNH \$+4
SJMP cadd

Long Sequence

JNH \$+5
LJMP cadd

Flags Affected

None

Examples

BH TARGET_LABEL

BLE

Format

BLE cadd

Operation

This instruction is a generic branch on less or equal condition (i.e., either the negative flag or the zero flag is set). The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JLE cadd

Medium Sequence

JGT \$+4
SJMP cadd

Long Sequence

JGT \$+5
LJMP cadd

Flags Affected

None

Examples

BLE TARGET_LABEL

BLT

Format

BLT cadd

Operation

This instruction is a generic branch on less-than condition (i.e., the negative flag is set). The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JLT cadd

Medium Sequence

JGE \$+4
SJMP cadd

Long Sequence

JGE \$+5
LJMP cadd

Flags Affected

None

Examples

BLT TARGET_LABEL

BMOV

Format

BMOV lreg,wreg

Operation

This instruction is a generic block move used to move a block of word data from one location in memory to another. The leftmost operand contains the source and destination pointers in adjacent word registers. The source and destination addresses are calculated from these pointers using the indirect with auto-increment addressing mode. The rightmost operand contains the word count of the block move; it is not decremented during the block move.

```
COUNT ← (CNTREG)
LOOP: SRCPTR ← (PTRS)
DSTPTR ← (PTRS + 2)
(DSTPTR) ← (SRCPTR)
(PTRS) ← SRCPTR + 2
(PTRS + 2) ← DSTPTR + 2
COUNT ← COUNT - 1
if COUNT <> 0 then go to LOOP
```

Notes

This instruction is specific to the 80C196 family of microcontrollers. This instruction can result in a very long un-interruptible operation. To provide an interruptible version of BMOV on the 80C196KC processor, use BMOV I. To provide an interruptible version of BMOV on the 80C196KB processor, use BMOV with DJNZW/DJNZ (see examples). This is possible because although the pointers to the source and destination locations are modified, the block word count is not.

BMOV (continued)

Opcode Pattern

11000001	wreg	lreg
----------	------	------

Flags Affected

None

Examples

BMOV PLACES,COUNT

The following example moves 10 blocks of 5 words each:

```
LD PTRS, SRC ;Pointer to base of source block
LD PTRS+2, DST ;Pointer to base of destination block
LD CNTREG, #5 ;Number of words moved per block
LD CNTSET, #10 ;Counter for number of blocks moved
MOVE: BMOV PTRS, CNTREG ;Move a block
DJNZW CNTSET, MOVE ;Decrement counter, loop if not zero
```

BMOVI

Format

BMOVI lreg,wreg

Operation

This instruction is a generic block move used to move a block of word data from one location in memory to another. BMOVI is identical to BMOV except that BMOVI is interruptable.

```
COUNT ← (CNTREG)
LOOP: SRCPTR ← (PTRS)
DSTPTR ← (PTRS + 2)
(DSTPTR) ← (SRCPTR)
(PTRS) ← SRCPTR + 2
(PTRS + 2) ← DSTPTR + 2
COUNT ← COUNT - 1
if COUNT <> 0 then go to LOOP
```

Notes

This instruction is specific to the 80C196KC microcontroller. This instruction is interruptible. To provide an un interruptible version of BMOVI, use BMOV.

Opcode Pattern

11001101	wreg	lreg
----------	------	------

Flags Affected

None

Examples

BMOVI PLACES,COUNT

BNC

Format

BNC cadd

Operation

This instruction is a generic branch on carry not set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JNC cadd

Medium Sequence

JC \$+4
SJMP cadd

Long Sequence

JC \$+5
LJMP cadd

Flags Affected

None

Examples

BNC TARGET_LABEL

BNE

Format

BNE cadd

Operation

This instruction is a generic branch on not equal (to 0) condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JNE cadd

Medium Sequence

JE \$+4
SJMP cadd

Long Sequence

JE \$+5
LJMP cadd

Flags Affected

None

Examples

BNE TARGET_LABEL

BNH

Format

BNH cadd

Operation

This instruction is a generic branch on not higher (i.e., either the carry flag is clear , or the zero flag is set, which is less or equal in unsigned arithmetic) condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JNH cadd

Medium Sequence

JH \$+4
SJMP cadd

Long Sequence

JH \$+5
LJMP cadd

Flags Affected

None

Examples

BNH TARGET_LABEL

BNST

Format

BNST cadd

Operation

This instruction is a generic branch on sticky-bit not set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JNST cadd

Medium Sequence

JST \$+4
SJMP cadd

Long Sequence

JST \$+5
LJMP cadd

Flags Affected

None

Examples

BNST TARGET_LABEL

BNV

Format

BNV cadd

Operation

This instruction is a generic branch on overflow not set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JNV cadd

Medium Sequence

JV \$+4
SJMP cadd

Long Sequence

JV \$+5
LJMP cadd

Flags Affected

None

Examples

BNV TARGET_LABEL

BNVT

Format

BNVT cadd

Operation

This instruction is a generic branch on overflow trap not set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JNVT cadd

Medium Sequence

JVT \$+4
SJMP cadd

Long Sequence

JVT \$+5
LJMP cadd

Flags Affected

VT

Examples

BNVT TARGET_LABEL

BR (Direct)

Format

BR cadd

Operation

This instruction is a generic branch instruction. The assembler chooses SJMP or LJMP according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Opcode Pattern

Translated to SJMP or LJMP as needed

Flags Affected

None

Examples

BR TARGET_LABEL

BR (Indirect)

Format

BR [wreg]

Operation

The execution continues at the address specified in the operand word register.

$PC \leftarrow (DEST)$

Opcode Pattern

11100011	wreg
----------	------

Flags Affected

None

Examples

BR [GAMMA]

BST

Format

BST cadd

Operation

This instruction is a generic branch on sticky-bit set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JST cadd

Medium Sequence

JNST \$+4
SJMP cadd

Long Sequence

JNST \$+5
LJMP cadd

Flags Affected

None

Examples

BST TARGET_LABEL

BV

Format

BV cadd

Operation

This instruction is a generic branch on overflow set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JV cadd

Medium Sequence

JNV \$+4
SJMP cadd

Long Sequence

JNV \$+5
LJMP cadd

Flags Affected

None

Examples

BV TARGET_LABEL

BVT

Format

BVT cadd

Operation

This instruction is a generic branch on overflow-trap set condition. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

JVT cadd

Medium Sequence

JNVT \$+4
SJMP cadd

Long Sequence

JNVT \$+5
LJMP cadd

Flags Affected

VT

Examples

BVT TARGET_LABEL

CALL (Direct)

Format

CALL cadd

Operation

This instruction is a generic CALL instruction. The assembler chooses SCALL or LCALL according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Opcode Pattern

Translated to SCALL or LCALL as needed

Flags Affected

None

Examples

CALL TARGET_LABEL

CLR

Format CLR wreg

Operation The value of the word operand is set to 0.
 $(DEST) \leftarrow 0$

Opcode Pattern

00000001	wreg
----------	------

Flags Affected C, N, V, Z

Examples CLR DELTA

CLRB

Format

CLRB breg

Operation

The value of the byte operand is set to 0.

$(\text{DEST}) \leftarrow 0$

Opcode Pattern

00010001	breg
----------	------

Flags Affected

C, N, V, Z

Examples

CLRB ALPHA

CLRC

Format

CLRC

Operation

The value of the carry flag is set to 0.

$C \leftarrow 0$

Opcode Pattern

11111000

Flags Affected

C

Examples

CLRC

CLRVT

Format

CLRVT

Operation

The value of the overflow-trap flag is set to 0.

$VT \leftarrow 0$

Opcode Pattern

11111100

Flags Affected

VT

Examples

CLRVT

NOTE

After the execution of a CLRVT instruction, the overflow flag can be set and the overflow-trap flag cleared.

CMP

Format

CMP wreg, waop

Operation

The source (rightmost) word operand is subtracted from the destination (leftmost) word operand. The flags are altered but the operands remain unaffected. The carry flag is set as complement of borrow.

(DEST) - (SRC)

Opcode Pattern

100010aa	waop	wreg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

CMP TIMER2, #500
CMP ALPHA, BETA
CMP ALPHA, [GAMMA] +

CMPB

Format

CMPB breg,baop

Operation

The source (rightmost) byte operand is subtracted from the destination (leftmost) byte operand. The flags are altered but the operands remain unaffected. The carry flag is set as complement of borrow.

(DEST) - (SRC)

Opcode Pattern

100110aa	baop	breg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

CMPB SBUF, #'A'
CMPB ALPHA, BETA
CMPB ALPHA, [GAMMA]++

CMPL

Format

CMPL lreg, lreg

Operation

This instruction is used to compare the magnitudes of two double word (long) operands. The operands are specified using the direct addressing mode. The source (rightmost) long operand is subtracted from the destination (leftmost) long operand. Five PSW flags are altered but the operands remain unaffected. The carry flag is set as complement of borrow.

DST - SRC

Note

This instruction is specific to the 80C196 family of microcontrollers.

Opcode Pattern

1100101	lreg	lreg
---------	------	------

Flags Affected

C, N, V, VT, Z

Examples

CMPL ALPHA, BETA
CMPL ALPHA, [GAMMA] +

DBNZ

Format

DBNZ breg, cadd

Operation

This instruction is a generic decrement and branch if not 0. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This is not necessarily the most efficient representation when a forward reference is used.

Short Sequence

DJNZ breg, cadd

Medium Sequence

DJNZ breg, \$+5
SJMP \$+4
SJMP cadd

Long Sequence

DJNZ breg, \$+5
SJMP \$+5
LJMP cadd

Flags Affected

None

DBNZ (continued)

Examples

DBNZ ALPHA, START_OF_LOOP

An example showing the distance limits ASM96 uses in determining whether to expand DBNZ to a short, medium, or long sequence is given under the DBNZW entry in this section. The expansion limits are the same for DBNZ and DBNZW. Note the following differences between the two generic instructions:

- DBNZ expansions use the jump instruction DJNZ; DBNZW expansions use the jump instruction DJNZW.
- The leftmost operand for DJNZ is a byte operand (breg); the corresponding operand for DJNZW is a word operand (wreg).

DBNZW

Format

DBNZW wreg,cadd

Operation

This instruction is the same as DBNZ except the count is a word operand instead of a byte operand; thus this instruction expands to DJNZW instead of to DJNZ. The assembler chooses one of the following sequences of instructions, according to the distance to the referenced address. This instruction does not necessarily give the most efficient representation when a forward reference is used.

Short Sequence

DJNZW wreg,cadd

Medium Sequence

DJNZW wreg,\$+4

SJMP \$+4

SJMP cadd

Long Sequence

DJNZW wreg,cadd

SJMP \$+5

LJMP cadd

Note

This instruction is specific to the 80C196 family of microcontrollers.

Flags Affected

None

DBNZW (continued)

Examples

DBNZW COUNTER, START_OF_LOOP

The following example shows the distance limits ASM96 uses in determining whether to expand DBNZW to a short, medium, or long sequence:

```
                dbnzw wreg, 100h                ; long sequence.
DJNZW wreg,$+4
SJMP $+5
LJMP 100h      ; long sequence.
                dbnzw wreg, $-1018              ; long sequence.
DJNZW wreg,$+4
SJMP $+5
LJMP $-1018    ; long sequence.

                dbnzw wreg, $-1017              ; medium sequence.
DJNZW wreg,$+4
SJMP $+4
SJMP $-1017    ; medium sequence.
                dbnzw wreg, $-126               ; medium sequence.
DJNZW wreg,$+4
SJMP $+4
SJMP $-126     ; medium sequence.

                dbnzw wreg, $-125              ; short sequence.
                dbnzw wreg, $                 ; short sequence.
                dbnzw wreg, $+130             ; short sequence.

                dbnzw wreg, $+131              ; medium sequence.
DJNZW wreg,$+4
SJMP $+4
SJMP $+131     ; medium sequence.
                dbnzw wreg, $+1030            ; medium sequence.
DJNZW wreg,$+4
SJMP $+4
SJMP $+1030    ; medium sequence.

                dbnzw wreg, $+1031            ; long sequence.
DJNZW wreg, $+4
SJMP $+5
LJMP $+1031    ; long sequence.
                dbnzw wreg, 0F000h           ; long sequence.
DJNZW wreg,$+4
SJMP $+5
LJMP 0F000h    ; long sequence.
```

DEC

Format

DEC wreg

Operation

The value of the word operand is decremented by 1.

$(\text{DEST}) \leftarrow (\text{DEST}) - 1$

Opcode Pattern

00000101	wreg
----------	------

Flags Affected

C, N, V, VT, Z

Examples

DEC DELTA

DECB

Format

DECB breg

Operation

The value of the byte operand is decremented by 1.

$(\text{DEST}) \leftarrow (\text{DEST}) - 1$

Opcode Pattern

00010101	breg
----------	------

Flags Affected

C, N, V, VT, Z

Examples

DECB ALPHA

DI

Format

DI

Operation

Interrupts are disabled.

$I \leftarrow 0$

Opcode Pattern

11111010

Flags Affected

I

Examples

DI

DIV

Format

DIV 1reg, waop

Operation

This instruction divides the content of the destination double-word operand by the content of the source word operand, using signed arithmetic. The low-order word of the destination (i.e., the word with the lower address) will contain the quotient; the high-order word will contain the remainder.

$(\text{low word DEST}) \leftarrow (\text{DEST}) / (\text{SRC})$

$(\text{high word DEST}) \leftarrow (\text{DEST}) \text{ MOD } (\text{SRC})$

The above two statements are performed concurrently.

Opcode Pattern

11111110	100011aa	waop	1reg
----------	----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

DIV DELTA, #500
DIV ALPHA, BETA
DIV ALPHA, [GAMMA] +

DIVB

Format DIVB wreg,baop

Operation This instruction divides the content of the destination word operand by the content of the source byte operand, using signed arithmetic. The low-order byte of the destination (i.e. the byte with the lower address) will contain the quotient; the high-order byte will contain the remainder.

$$\begin{aligned} (\text{low byte DEST}) &\leftarrow (\text{DEST}) / (\text{SRC}) \\ (\text{high byte DEST}) &\leftarrow (\text{DEST}) \text{ MOD } (\text{SRC}) \end{aligned}$$

The above two statements are performed concurrently.

Opcode Pattern	11111110	100111aa	baop	wreg

Flags Affected C, N, V, VT, Z

Examples

```
DIVB DELTA, #500
DIVB ALPHA, BETA
DIVB ALPHA, [GAMMA]+
```

DIVU

Format

DIVU 1reg, waop

Operation

This instruction divides the content of the destination double-word operand by the content of the source word operand, using unsigned arithmetic. The low-order word of the destination (i.e. the word with the lower address) will contain the quotient; the high-order word will contain the remainder.

$(\text{low word DEST}) \leftarrow (\text{DEST}) / (\text{SRC})$

$(\text{high word DEST}) \leftarrow (\text{DEST}) \text{ MOD } (\text{SRC})$

The above two statements are performed concurrently.

Opcode Pattern

100011aa	waop	1reg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

DIVU DELTA, #500
DIVU ALPHA, BETA
DIVU ALPHA, [GAMMA] +

DIVUB

Format

DIVUB wreg,baop

Operation

This instruction divides the content of the destination word operand by the content of the source byte operand, using unsigned arithmetic. The low-order byte of the destination (i.e., the byte with the lower address) will contain the quotient; the high-order byte will contain the remainder.

$(\text{low byte DEST}) \leftarrow (\text{DEST}) / (\text{SRC})$

$(\text{high byte DEST}) \leftarrow (\text{DEST}) \bmod (\text{SRC})$

The above two statements are performed concurrently.

Opcode Pattern

100111aa	baop	wreg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

DIVUB DELTA, #250
DIVUB ALPHA, BETA
DIVUB ALPHA, [GAMMA] +

DJNZ

Format

DJNZ breg, cadd

Operation

The value of the byte operand is decremented by 1. If the result is not equal to 0, the distance from the end of this instruction to the target label is added to the program counter, effecting the jump.

$(COUNT) \leftarrow (COUNT) - 1$

if $(COUNT) \neq 0$ then

$PC \leftarrow PC + disp$ (sign-extended to 16 bits)

and if

Opcode Pattern

11100000	breg	disp
----------	------	------

Flags Affected

None

Examples

DJNZ COUNTER, START_OF_LOOP

DJNZW

Format

DJNZW wreg, cadd

Operation

This instruction is the same as DJNZ except the count is a word operand instead of a byte operand and DJNZW takes one state time longer than DJNZ. The counter word is decremented by 1; if the result is not equal to 0 the jump occurs. The range of the jump is -128 to +127.

COUNT $\leftarrow$ COUNT - 1
if COUNT $\neq$ 0 then
PC $\leftarrow$ PC + disp (sign extended)

Note

This instruction is specific to the 80C196 family of microcontrollers.

Opcode Pattern

11100001	wreg	cadd
----------	------	------

Flags Affected

None

Examples

DJNZW COUNTER, START_OF_LOOP

DPTS

Format DPTS

Operation This instruction disables the peripheral transaction server (PTS) by clearing the appropriate bit in the program status word (PSW).

Note This instruction is specific to the 80C196KC microcontroller.

Opcode Pattern 11101100

Flags Affected PSE

Examples DPTS

EI

Format

EI

Operation

Interrupts are enabled following the execution of the next statement.

$I \leftarrow 1$

Opcode Pattern

11111011

Flags Affected

I

Examples

EI

EPTS

Format

EPTS

Operation

This instruction enables the peripheral transaction server (PTS) by setting the appropriate bit in the program status word (PSW).

Note

This instruction is specific to the 80C196KC microcontroller.

Opcode Pattern

11101101

Flags Affected

PSE

Examples

EPTS

EXT

Format

EXT 1reg

Operation

The low-order word of the operand is sign-extended throughout the high-order word of the operand.

if (low word DEST) < 8000H then (high word DEST) $\leftarrow$ 0

else (high word DEST) $\leftarrow$ 0FFFFH

C $\leftarrow$ 0

V $\leftarrow$ 0

Opcode Pattern

00000110	1reg
----------	------

Flags Affected

C, N, V, Z

Examples

EXT DELTA

EXTB

Format

EXTB wreg

Operation

The low-order byte of the operand is sign-extended throughout the high-order byte of the operand.

if (low byte DEST) < 80H then (high byte
DEST) $\leftarrow$ 0

else (high byte DEST) $\leftarrow$ 0FFH

C $\leftarrow$ 0

V $\leftarrow$ 0

Opcode Pattern

00010110

wreg

Flags Affected

C, N, V, Z

Examples

EXTB ALPHA

IDLPD

Format

IDLPD #key

Operation

This instruction causes the processor to enter IDLE or POWERDOWN mode or to execute a RESET. The bus controller completes any prefetch cycle in progress before the CPU stops or resets. The mode the processor enters depends on the byte value of the key, as follows:

Key	Processor Response
1	enter IDLE mode
2	enter POWERDOWN mode
any other	execute RESET

```
if KEY = 1 then enter IDLE
else if KEY = 2 then enter POWERDOWN
else execute RESET
```

Note

This instruction is specific to the 80C196 family of microcontrollers.

Opcode Pattern

11110110	key
----------	-----

Flags Affected

Key = 1 or Key = 2:	None
Otherwise:	All reset

INC

Format INC wreg

Operation The value of the word operand is incremented by 1.
 $(\text{DEST}) \leftarrow (\text{DEST}) + 1$

Opcode Pattern

00000111	wreg
----------	------

Flags Affected C, N, V, VT, Z

Examples INC DELTA

INCB

Format

INCB breg

Operation

The value of the byte operand is incremented by 1.

$(\text{DEST}) \leftarrow (\text{DEST}) + 1$

Opcode Pattern

00010111	breg
----------	------

Flags Affected

C, N, V, VT, Z

Examples

INCB ALPHA

JBC

Format

JBC breg,bitno,cadd

Operation

The specified bit is tested. If it is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the bit is set (i.e., 1), control passes to the next sequential instruction.

if (specified bit) = 0 then
 $PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

00110bbb	breg	disp
----------	------	------

where *bbb* is the bit number within the specified register.

Flags Affected

None

Examples

JBC SBUF, 7, EVEN_PARITY

JBS

Format

JBS breg,bitno,cadd

Operation

The specified bit is tested. If it is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the bit is clear (i.e., 0), control passes to the next sequential instruction.

if (specified bit) = 1 then
 $PC \leftarrow PC + \text{disp}$ (sign-extended to 16 bits)

Opcode Pattern

00111bbb	breg	disp
----------	------	------

where *bbb* is the bit number within the specified register.

Flags Affected

None

Examples

JBS SBUF, 7, ODD_PARITY

JC

Format

JC cadd

Operation

If the carry flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the carry flag is clear (i.e., 0), control passes to the next sequential instruction.

if $C = 1$ then

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11011011

disp

Flags Affected

None

Examples

JC TARGET_ADDRESS

JE

Format

JE cadd

Operation

If the zero flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the zero flag is clear (i.e., 0), control passes to the next sequential instruction.

if $Z = 1$ then

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11011111	disp
----------	------

Flags Affected

None

Examples

JE TARGET_ADDRESS

JGE

Format

JGE cadd

Operation

If the negative flag is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the negative flag is set (i.e., 1), control passes to the next sequential instruction.

if N = 0 then

PC ← PC + disp (sign-extended to 16 bits)

Opcode Pattern

11010110

disp

Flags Affected

None

Examples

JGE TARGET_ADDRESS

JGT

Format

JGT cadd

Operation

If both the negative flag and the zero flag are clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If either the negative flag or the zero flag is set (i.e., 1), control passes to the next sequential instruction.

if $N = 0$ AND $Z = 0$ then
 $PC \leftarrow PC + \text{disp}$ (sign-extended to 16 bits)

Opcode Pattern

11010010	disp
----------	------

Flags Affected

None

Examples

JGT TARGET_ADDRESS

JH

Format

JH cadd

Operation

If the carry flag is set (i.e., 1), but the zero flag is not, the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If either the carry flag is clear (i.e., 0), or the zero flag is set (i.e., 1), control passes to the next sequential instruction.

if $C = 1$ and $Z = 0$ then

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11011001

disp

Flags Affected

None

Examples

JH TARGET_ADDRESS

JLE

Format

JLE cadd

Operation

If either the negative flag or the zero flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If both the negative flag and the zero flag are clear (i.e., 0), control passes to the next sequential instruction.

if $N = 1$ OR $Z = 1$ then

$PC \leftarrow PC + \text{disp}$ (sign-extended to 16 bits)

Opcode Pattern

11011010	disp
----------	------

Flags Affected

None

Examples

JLE TARGET_ADDRESS

JLT

Format

JLT cadd

Operation

If the negative flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the negative flag is clear (i.e., 0), control passes to the next sequential instruction.

if $N = 1$ then

$PC \leftarrow PC + \text{disp}$ (sign-extended to 16 bits)

Opcode Pattern

11011110	disp
----------	------

Flags Affected

None

Examples

JLT TARGET_ADDRESS

JNC

Format

JNC cadd

Operation

If the carry flag is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the carry flag is set (i.e., 1), control passes to the next sequential instruction.

if $C = 0$ then

$PC \leftarrow PC + \text{disp}$ (sign-extended to 16 bits)

Opcode Pattern

11010011	disp
----------	------

Flags Affected

None

Examples

JNC TARGET_ADDRESS

JNE

Format

JNE cadd

Operation

If the zero flag is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the zero flag is set (i.e., 1), control passes to the next sequential instruction.

if $Z = 0$ then

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11010111	disp
----------	------

Flags Affected

None

Examples

JNE TARGET_ADDRESS

JNH

Format

JNH cadd

Operation

If either the carry flag is clear (i.e., 0), or the zero flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the carry flag is set (i.e., 1), but the zero flag is clear (i.e., 0), control passes to the next sequential instruction.

if $C = 0$ OR $Z = 1$ then
 $PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11010001	disp
----------	------

Flags Affected

None

Examples

JNH TARGET_ADDRESS

JNST

Format JNST cadd

Operation If the sticky bit flag is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the sticky bit flag is set (i.e., 1), control passes to the next sequential instruction.

if ST = 0 then
PC ← PC + disp (sign-extended to 16 bits)

Opcode Pattern

11010000	disp
----------	------

Flags Affected None

Examples JNST TARGET_ADDRESS

JNV

Format

JNV cadd

Operation

If the overflow flag is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the overflow flag is set (i.e., 1), control passes to the next sequential instruction.

if $V = 0$ then

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11010101	disp
----------	------

Flags Affected

None

Examples

JNV TARGET_ADDRESS

JNVT

Format

JNVT cadd

Operation

If the overflow trap flag is clear (i.e., 0), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the overflow trap flag is set (i.e., 1), control passes to the next sequential instruction.

if VT = 0 then
 PC ← PC + disp (sign-extended to 16 bits)
VT ← 0

Opcode Pattern

11010100	disp
----------	------

Flags Affected

VT

Examples

JNVT TARGET_ADDRESS

JST

Format

JST cadd

Operation

If the sticky bit flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the sticky bit flag is clear (i.e., 0), control passes to the next sequential instruction.

if ST = 1 then

PC ← PC + disp (sign-extended to 16 bits)

Opcode Pattern

11011000	disp
----------	------

Flags Affected

None

Examples

JST TARGET_ADDRESS

JV

Format

JV cadd

Operation

If the overflow flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the overflow flag is clear (i.e., 0), control passes to the next sequential instruction.

if $V = 1$ then

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

11011101	disp
----------	------

Flags Affected

None

Examples

JV TARGET_ADDRESS

JVT

Format

JVT cadd

Operation

If the overflow trap flag is set (i.e., 1), the distance from the end of this instruction to the target label is added to the program counter, effecting the jump. If the overflow trap flag is clear (i.e., 0), control passes to the next sequential instruction.

```
if VT = 1 then
    PC ← PC + disp (sign-extended to 16 bits)
VT ← 0
```

Opcode Pattern

11011100	disp
----------	------

Flags Affected

VT

Examples

JVT TARGET_ADDRESS

LCALL

Format

LCALL cadd

Operation

The contents of the program counter (the return address) is pushed onto the stack. Then the distance from the end of this instruction to the target label is added to the program counter, effecting the call. The operand can be any address in the entire address space.

$SP = SP - 2$

$(SP) \leftarrow PC$

$PC \leftarrow PC + disp$

Opcode Pattern

11101111	disp-low	disp-hi
----------	----------	---------

Flags Affected

None

Examples

LCALL MY_ROUTINE

LD

Format

LD wreg, waop

Operation

The value of the source (rightmost) word operand is stored into the destination (leftmost) operand.

$(\text{DEST}) \leftarrow (\text{SRC})$

Opcode Pattern

101000aa	waop	wreg
----------	------	------

Flags Affected

None

Examples

LD DELTA, #500
LD ALPHA, BETA
LD ALPHA, [GAMMA]++

LDB

Format

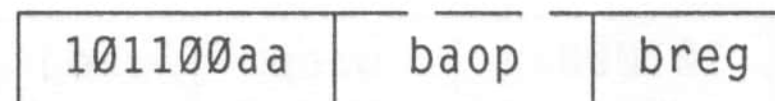
LDB breg,baop

Operation

The value of the source (rightmost) byte operand is stored into the destination (leftmost) operand.

$(\text{DEST}) \leftarrow (\text{SRC})$

Opcode Pattern



Flags Affected

None

Examples

LDB DELTA, #2
LDB ALPHA, BETA
LDB ALPHA, [GAMMA] +

LDBSE

Format

LDBSE wreg,baop

Operation

The value of the source (rightmost) byte operand is sign-extended and stored into the destination (leftmost) word operand.

(low byte DEST) $\leftarrow$ (SRC)

if (SRC) < 80H then (high byte DEST) $\leftarrow$ 0

else (high byte DEST) $\leftarrow$ 0FFH

Opcode Pattern

101111aa	baop	wreg
----------	------	------

Flags Affected

None

Examples

LDBSE DELTA, #2
LDBSE ALPHA, BETA
LDBSE ALPHA, [GAMMA]+

LDBZE

Format

LDBZE wreg,baop

Operation

The value of the source (rightmost) byte operand is zero-extended and stored into the destination (leftmost) word operand.

(low byte DEST) $\leftarrow$ (SRC)

(high byte DEST) $\leftarrow$ $\emptyset$

Opcode Pattern

101011aa	baop	wreg
----------	------	------

Flags Affected

None

Examples

LDBZE DELTA, #2

LDBZE ALPHA, BETA

LDBZE ALPHA, [GAMMA]+

LJMP

Format

LJMP cadd

Operation

The distance from the end of this instruction to the target label is added to the program counter, effecting the jump. The operand can be any address in the entire address space.

$PC \leftarrow PC + \text{disp}$

Opcode Pattern

11100111	disp-low	disp-hi
----------	----------	---------

Flags Affected

None

Examples

LJMP ANY_LABEL

MUL (Two Operands)

Format

MUL Ireg,waop

Operation

The two word operands are multiplied using signed arithmetic and the 32-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$(\text{DEST}) \leftarrow (\text{DEST}) * (\text{SRC})$

Opcode Pattern

11111110	011011aa	waop	Ireg
----------	----------	------	------

Flags Affected

ST

Examples

MUL DELTA, #500
MUL ALPHA, BETA
MUL ALPHA, [GAMMA] +

MUL (Three Operands)

Format

MUL Ireg,wreg,waop

Operation

The second and third word operands are multiplied using signed arithmetic and the 32-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$(\text{DEST}) \leftarrow (\text{SRC1}) * (\text{SRC2})$

Opcode Pattern

11111110	010011aa	waop	wreg	Ireg
----------	----------	------	------	------

Flags Affected

ST

Examples

```
MUL    DELTA, TIMER1, #2
MUL    ALPHA, BETA, GAMMA
MUL    ALPHA, DELTA, 10[GAMMA]
```

MULB (Two Operands)

Format MULB wreg,baop

Operation The two byte operands are multiplied using signed arithmetic and the 16-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$$(DEST) \leftarrow (DEST) * (SRC)$$

Opcode Pattern	<table><tr><td>11111110</td><td>011111aa</td><td>baop</td><td>wreg</td></tr></table>	11111110	011111aa	baop	wreg
11111110	011111aa	baop	wreg		

Flags Affected ST

Examples

MULB DELTA, #2

MULB ALPHA, BETA

MULB ALPHA, [GAMMA]+

MULB (Three Operands)

Format MULB wreg,breg,baop

Operation The second and third byte operands are multiplied using signed arithmetic and the 16-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$$(DEST) \leftarrow (SRC1) * (SRC2)$$

Opcode Pattern

11111110	010111aa	baop	breg	wreg
----------	----------	------	------	------

Flags Affected ST

Examples

MULB DELTA, TIMER1, #2

MULB ALPHA, BETA, GAMMA

MULB ALPHA, DELTA, 10[GAMMA]

MULU (Two Operands)

Format

MULU 1reg,waop

Operation

The two word operands are multiplied using unsigned arithmetic and the 32-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$(\text{DEST}) \leftarrow (\text{DEST}) * (\text{SRC})$

Opcode Pattern

011011aa	waop	1reg
----------	------	------

Flags Affected

ST

Examples

MULU DELTA, #500

MULU ALPHA, BETA

MULU ALPHA, [GAMMA]+

MULU (Three Operands)

Format MULU lreg,wreg,waop

Operation The second and third word operands are multiplied using unsigned arithmetic and the 32-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$$(DEST) \leftarrow (SRC1) * (SRC2)$$

Opcode Pattern

010011aa	waop	wreg	lreg
----------	------	------	------

Flags Affected ST

Examples

MULU DELTA, TIMER1, #2
MULU ALPHA, BETA, GAMMA
MULU ALPHA, DELTA, 10[GAMMA]

MULUB (Two Operands)

Format

MULUB wreg,baop

Operation

The two byte operands are multiplied using unsigned arithmetic and the 16-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$$(DEST) \leftarrow (DEST) * (SRC)$$

Opcode Pattern

011111aa	baop	wreg
----------	------	------

Flags Affected

ST

Examples

MULUB DELTA, #2
MULUB ALPHA, BETA
MULUB ALPHA, [GAMMA] +

MULUB (Three Operands)

Format

MULUB wreg,breg,baop

Operation

The second and third byte operands are multiplied using unsigned arithmetic and the 16-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

$(\text{DEST}) \leftarrow (\text{SRC1}) * (\text{SRC2})$

Opcode Pattern

010111aa	baop	breg	wreg
----------	------	------	------

Flags Affected

ST

Examples

```
MULUB    DELTA, TIMER1, #2
MULUB    ALPHA, BETA, GAMMA
MULUB    ALPHA, DELTA, 10[GAMMA]
```

NEG

Format

NEG wreg

Operation

The value of the word operand is negated.

$(\text{DEST}) \leftarrow -(\text{DEST})$

Opcode Pattern

00000011

wreg

Flags Affected

C, N, V, VT, Z

Examples

NEG DELTA

NEGB

Format

NEGB breg

Operation

The value of the byte operand is negated.

$(\text{DEST}) \leftarrow -(\text{DEST})$

Opcode Pattern

00010011	breg
----------	------

Flags Affected

C, N, V, VT, Z

Examples

NEGB ALPHA

NOP

Format NOP

Operation Absolutely nothing is done. Control passes to the next sequential instruction.

Opcode Pattern 11111101

Flags Affected None

Examples NOP

NORML

Format

NORML lreg,breg

Operation

The leftmost double-word operand is normalized; i.e., it is shifted to the left until its most significant bit is 1. If the most significant bit is still 0 after 31 shifts, the process stops and the zero flag is set. The number of shifts actually performed is stored in the second operand.

```
(COUNT) ← 0
DO WHILE (MSB(DEST) = 0) AND ((COUNT) < 31)
    (DEST) ← (DEST) * 2
    (COUNT) ← (COUNT) + 1
END WHILE
```

Opcode Pattern

00001111	breg	lreg
----------	------	------

Flags Affected

C, N, Z

Examples

NORML ALPHA, RESULTING_COUNT

NOT

Format

NOT wreg

Operation

The value of the word operand is complemented: each 1 is replaced with a 0, and each 0 with a 1.

$(\text{DEST}) \leftarrow \text{NOT } (\text{DEST})$

$C \leftarrow 0$

$V \leftarrow 0$

Opcode Pattern

00000010	wreg
----------	------

Flags Affected

C, N, V, Z

Examples

NOT DELTA

NOTB

Format

NOTB breg

Operation

The value of the byte operand is complemented: each 1 is replaced with a 0, and each 0 with a 1.

$(\text{DEST}) \leftarrow \text{NOT } (\text{DEST})$

$C \leftarrow 0$

$V \leftarrow 0$

Opcode Pattern

00010010	breg
----------	------

Flags Affected

C, N, V, Z

Examples

NOTB ALPHA

OR

Format

OR wreg, waop

Operation

The source (rightmost) word operand is ORed with the destination (leftmost) word operand. Each bit is set to 1 if the corresponding bit in either the source operand or the destination operand was 1. The result replaces the original destination operand.

$(\text{DEST}) \leftarrow (\text{DEST}) \text{ OR } (\text{SRC})$

$C \leftarrow 0$

$V \leftarrow 0$

Opcode Pattern

10000000aa	waop	wreg
------------	------	------

Flags Affected

C, N, V, Z

Examples

OR DELTA, #500
OR ALPHA, BETA
OR ALPHA, [GAMMA] +

ORB

Format

ORB breg,baop

Operation

The source (rightmost) byte operand is ORed with the destination (leftmost) byte operand. Each bit is set to 1 if the corresponding bit in either the source operand or the destination operand was 1. The result replaces the original destination operand.

$(\text{DEST}) \leftarrow (\text{DEST}) \text{ OR } (\text{SRC})$

$C \leftarrow 0$

$V \leftarrow 0$

Opcode Pattern

100100aa	baop	breg
----------	------	------

Flags Affected

C, N, V, Z

Examples

ORB DELTA, #2

ORB ALPHA, BETA

ORB ALPHA, [GAMMA]+

POP

Format

POP waop

Operation

The word on top of the stack is popped and placed at the destination operand.

$(\text{DEST}) \leftarrow (\text{SP})$

$\text{SP} \leftarrow \text{SP} + 2$

Opcode Pattern

110011aa

waop

Flags Affected

None

Examples

POP DELTA

POP [GAMMA] +

NOTE

It is illegal to use an immediate operand in a POP instruction.

POPA

Format

POPA

Operation

This instruction is used instead of POPF to support the 8 additional interrupts in the 80C196 family of microcontrollers. This instruction is similar to POPF, but pops two words instead of one. The first word is popped into the IMASK1/WSR register pair; the second word is popped into the PSW/INT_MASK register pair; and the SP is incremented by 4. An interrupt cannot occur between this instruction and the one following it.

$$\text{IMASK1/WSR} \leftarrow (\text{SP})$$
$$\text{SP} \leftarrow \text{SP} + 2$$
$$\text{PSW/INT_MASK} \leftarrow (\text{SP})$$
$$\text{SP} \leftarrow \text{SP} + 2$$

Note

This instruction is specific to the 80C196 family of microcontrollers.

Opcode Pattern

11110101

Flags Affected

All

Examples

POPA

POPF

Format

POPF

Operation

The word on top of the stack is popped and placed in the PSW. Interrupts are disabled for the next instruction.

$(PSW) \leftarrow (SP)$

$SP \leftarrow SP + 2$

Opcode Pattern

11110011

Flags Affected

All

Examples

POPF

PUSH

Format

PUSH waop

Operation

The specified operand is pushed onto the stack.

$SP \leftarrow SP - 2$
 $(SP) \leftarrow (DEST)$

Opcode Pattern

110010aa	waop
----------	------

Flags Affected

None

Examples

PUSH DELTA
PUSH [GAMMA] +

PUSHA

Format

PUSHA

Operation

This instruction is used instead of PUSHF to support the 8 additional interrupts on the 80C196 family of microcontrollers. This instruction is similar to PUSHF, but pushes two words instead of one. The first word pushed is PSW/INT_MASK. The second word pushed is IMASK1/WSR. The PSW, INT_MASK, and IMASK1 registers are cleared, and the SP is decremented by 4. The WSR is unchanged. An interrupt cannot occur between this instruction and the one following it.

```
SP ← SP - 2
(SP) ← PSW/INT_MASK
PSW/INT_MASK ← 0
SP ← SP - 2
(SP) ← IMASK1/WSR
IMASK1 ← 0
```

Note

This instruction is specific to the 80C196 family of microcontrollers.

Opcode Pattern

11110100

Flags Affected

All are cleared

Examples

PUSHA

PUSHF

Format

PUSHF

Operation

The PSW is pushed on top of the stack, and then set to all 0's. This implies that interrupts are disabled.

$$SP \leftarrow SP - 2$$
$$(SP) \leftarrow PSW$$
$$PSW \leftarrow 0$$

Opcode Pattern

11110010

Flags Affected

All

Examples

PUSHF

RET

Format

RET

Operation

The PC is popped off the top of the stack.

PC ← (SP)
SP ← SP + 2

Opcode Pattern

11110000

Flags Affected

None

Examples

RET

RST

Format

RST

Operation

The PSW is initialized to 0, and the PC is initialized to 2080H. The I/O registers are set to their initial value (see Chapter 1).

PSW ← 0
PC ← 2080H

Opcode Pattern

11111111

Flags Affected

All

Examples

RST

SCALL

Format

SCALL cadd

Operation

The contents counter (the return address) is pushed onto the stack. Then the distance from the end of this instruction to the target label is added to the program counter, effecting the call. The offset from the end of this instruction to the target label should be in the range of -1024 to +1023 inclusive.

$$SP = SP - 2$$
$$(SP) \leftarrow PC$$
$$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$$

Opcode Pattern

00101xxx	disp-low
----------	----------

where xxx holds the three high-order bits of the displacement.

Flags Affected

None

Examples

SCALL MY_ROUTINE

SETC

Format

SETC

Operation

The carry flag is set.

$C \leftarrow 1$

Opcode Pattern

11111001

Flags Affected

C

Examples

SETC

SHL

Format

```
SHL    wreg, #count
      or
SHL    wreg, breg
```

Operation

The destination (leftmost) word operand is shifted left as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. The right bits of the result are filled with 0's. The last bit shifted out is saved in the carry.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← High order bit of (DEST)
    (DEST) ← (DEST)*2
    Temp ← Temp - 1
end while
```

Opcode Pattern

00001001	cnt/breg	wreg
----------	----------	------

Flags Affected

C, N, V, VT, Z

Examples

```
SHL DELTA, #5
```

SHLB

Format

SHLB breg, #count
or
SHLB breg, breg

Operation

The destination (leftmost) byte operand is shifted left as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. The right bits of the result are filled with 0's. The last bit shifted out is saved in the carry.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← High order bit of (DEST)
    (DEST) ← (DEST)*2
    Temp ← Temp - 1
end while
```

Opcode Pattern

00011001	cnt/breg	breg
----------	----------	------

Flags Affected

C, N, V, VT, Z

Examples

SHLB DELTA, #5

SHLL

Format

SHLL 1reg, #count
or
SHLL 1reg, breg

Operation

The destination (leftmost) double-word operand is shifted left as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. The right bits of the result are filled with 0's. The last bit shifted out is saved in the carry.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← High order bit of (DEST)
    (DEST) ← (DEST)*2
    Temp ← Temp - 1
end while
```

Opcode Pattern

00001101	cnt/breg	1reg
----------	----------	------

Flags Affected

C, N, V, VT, Z

Examples

SHLL DELTA, #5

SHR

Format

SHR wreg, #count
 or
SHR wreg, breg

Operation

The destination (leftmost) word operand is shifted right as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. The left bits of the result are filled with 0's. The last bit shifted out is saved in the carry. The sticky bit flag is clear at the beginning of the instruction, and set if at any time during the shift a 1 is shifted first into the carry, and a further shift cycle occurs.

```
Temp ← (COUNT)
do while Temp <> 0
  C ← Low order bit of (DEST)
  (DEST) ← (DEST)/2 where / is unsigned division
  Temp ← Temp - 1
end while
```

Opcode Pattern

00001000	cnt/breg	wreg
----------	----------	------

Flags Affected

C, N, ST, V, VT, Z

Examples

SHR DELTA, #5

SHRA

Format

SHRA wreg, #count
or
SHRA wreg, breg

Operation

The destination (leftmost) word operand is shifted right as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. If the original high order bit value was 0, 0's are shifted in. If that value was 1, 1's are shifted in. The last bit shifted out is saved in the carry. The sticky bit flag is clear at the beginning of the instruction, and set if at any time during the shift a 1 is shifted first into the carry, and a further shift cycle occurs.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← Low order bit of (DEST)
    (DEST) ← (DEST)/2 where / is signed division
    Temp ← Temp - 1
end while
```

Opcode Pattern

00001010	cnt/breg	wreg
----------	----------	------

Flags Affected

C, N, ST, Z

Examples

SHRA DELTA, #5

SHRAB

Format

SHRAB breg, #count
or
SHRAB breg, breg

Operation

The destination (leftmost) byte operand is shifted right as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. If the original high order bit value was 0, 0's are shifted in. If that value was 1, 1's are shifted in. The last bit shifted out is saved in the carry. The sticky bit flag is clear at the beginning of the instruction, and set if at any time during the shift a 1 is shifted first into the carry, and a further shift cycle occurs.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← Low order bit of (DEST)
    (DEST) ← (DEST)/2 where / is signed division
    Temp ← Temp - 1
end while
```

Opcode Pattern

00011010	cnt/breg	breg
----------	----------	------

Flags Affected

C, N, ST, Z

Examples

SHRAB DELTA, #5

SHRAL

Format

SHRAL lreg, #count
or
SHRAL lreg, breg

Operation

The destination (leftmost) double-word operand is shifted right as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. If the original high order bit value was 0, 0's are shifted in. If that value was 1, 1's are shifted in. The last bit shifted out is saved in the carry. The sticky bit flag is clear at the beginning of the instruction, and set if at any time during the shift a 1 is shifted first into the carry, and a further shift cycle occurs.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← Low order bit of (DEST)
    (DEST) ← (DEST)/2 where / is signed division
    Temp ← Temp - 1
end while
```

Opcode Pattern

00001110	cnt/breg	lreg
----------	----------	------

Flags Affected

C, N, ST, Z

Examples

SHRAL DELTA, #5

SHRB

Format

SHRB breg, #count
or
SHRB breg, breg

Operation

The destination (leftmost) byte operand is shifted right as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. The left bits of the result are filled with 0's. The last bit shifted out is saved in the carry. The sticky bit flag is clear at the beginning of the instruction, and set if at any time during the shift a 1 is shifted first into the carry, and a further shift cycle occurs.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← Low order bit of (DEST)
    (DEST) ← (DEST)/2 where / is unsigned division
    Temp ← Temp - 1
end while
```

Opcode Pattern

00011000	cnt/breg	breg
----------	----------	------

Flags Affected

C, N, ST, V, VT, Z

Examples

SHRB DELTA, #5

SHRL

Format

SHRL 1reg, #count
or
SHRL 1reg, breg

Operation

The destination (leftmost) double-word operand is shifted right as many times as specified by the count (rightmost) operand. The count can be specified either as an immediate value in the range of 0 to 15 inclusive, or as the content of any register, the address of which is 16 to 255. The left bits of the result are filled with 0's. The last bit shifted out is saved in the carry. The sticky bit flag is clear at the beginning of the instruction, and set if at any time during the shift a 1 is shifted first into the carry, and a further shift cycle occurs.

```
Temp ← (COUNT)
do while Temp <> 0
    C ← Low order bit of (DEST)
    (DEST) ← (DEST)/2 where / is unsigned division
    Temp ← Temp - 1
end while
```

Opcode Pattern

00001100	cnt/breg	breg
----------	----------	------

Flags Affected

C, N, ST, V, VT, Z

Examples

SHRL DELTA, #5

SJMP

Format

SJMP cadd

Operation

The distance from the end of this instruction to the target label is added to the program counter, effecting the jump. The offset from the end of this instruction to the target label should be in the range of -1024 to +1023 inclusive.

$PC \leftarrow PC + \text{disp (sign-extended to 16 bits)}$

Opcode Pattern

00100xxx	disp-low
----------	----------

where xxx holds the three high order bits of the displacement.

Flags Affected

None

Examples

SJMP LABEL

SKIP

Format SKIP breg

Operation Absolutely nothing is done. This is actually a two-byte NOP, where the second byte can be any value, and is simply ignored. Control passes to the next sequential instruction.

Opcode Pattern

00000000	breg
----------	------

Flags Affected None

Examples SKIP 'A'

ST

Format

ST wreg, waop

Operation

The value of the leftmost word operand is stored into the rightmost operand.

$(\text{DEST}) \leftarrow (\text{SRC})$

Opcode Pattern

110000aa	waop	wreg
----------	------	------

Flags Affected

None

Examples

ST ALPHA, BETA
ST ALPHA, [GAMMA] +

NOTE

It is illegal to use an immediate operand in a ST instruction.

STB

Format

STB breg,baop

Operation

The value of the leftmost byte operand is stored into the rightmost operand.

$(\text{DEST}) \leftarrow (\text{SRC})$

Opcode Pattern

110001aa	baop	breg
----------	------	------

Flags Affected

None

Examples

STB ALPHA, BETA
STB ALPHA, [GAMMA]+

NOTE

It is illegal to use an immediate operand in a STB instruction.

SUB (Two Operands)

Format

SUB wreg, waop

Operation

The source (rightmost) word operand is subtracted from the destination (leftmost) word operand, and the result is stored in the destination. The carry flag is set as complement of borrow.

$(\text{DEST}) \leftarrow (\text{DEST}) - (\text{SRC})$

Opcode Pattern

011010aa	waop	wreg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

```
SUB    DELTA, #500
SUB    ALPHA, BETA
SUB    ALPHA, [GAMMA]+
```

SUB (Three Operands)

Format SUB wreg,wreg,waop

Operation The source (rightmost) word operand is subtracted from the second word operand, and the result is stored in the destination (the leftmost operand). The carry flag is set as complement of borrow.

$(DEST) \leftarrow (SRC1) - (SRC2)$



Flags Affected C, N, V, VT, Z

Examples

SUB DELTA, TIMER1, #2

SUB ALPHA, BETA, GAMMA

SUB ALPHA, DELTA, 10[GAMMA]

SUBB (Two Operands)

Format

SUBB breg,baop

Operation

The source (rightmost) byte operand is subtracted from the destination (leftmost) byte operand, and the result is stored in the destination. The carry flag is set as complement of borrow.

$$(DEST) \leftarrow (DEST) - (SRC)$$

Opcode Pattern

011110aa	baop	breg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

```
SUBB    DELTA, #2
SUBB    ALPHA, BETA
SUBB    ALPHA, [GAMMA]+
```

SUBB (Three Operands)

Format SUBB breg,breg,baop

Operation The source (rightmost) byte operand is subtracted from the second byte operand, and the result is stored in the destination (the leftmost operand). The carry flag is set as complement of borrow.

$(\text{DEST}) \leftarrow (\text{SRC1}) - (\text{SRC2})$

Opcode Pattern



Flags Affected

C, N, V, VT, Z

Examples

```
SUBB DELTA, TIMER1, #2
SUBB ALPHA, BETA, GAMMA
SUBB ALPHA, DELTA, 10[GAMMA]
```


SUBC

Format

SUBC wreg,waop

Operation

The source (rightmost) word operand is subtracted from the destination (leftmost) word operand. If the carry flag was clear, 1 is subtracted from the above result. The result replaces the original destination operand. The carry flag is set as complement of borrow.

$$(DEST) \leftarrow (DEST) - (SRC) - (1-C)$$

Opcode Pattern

101010aa	waop	wreg
----------	------	------

Flags Affected

C, N, V, VT, Z

Examples

SUBC DELTA, #500
SUBC ALPHA, BETA
SUBC ALPHA, [GAMMA]+

SUBCB

Format SUBCB breg,baop

Operation The source (rightmost) byte operand is subtracted from the destination (leftmost) byte operand. If the carry flag was clear, 1 is subtracted from the above result. The result replaces the original destination operand. The carry flag is set as complement of borrow.

$$(DEST) \leftarrow (DEST) - (SRC) - (1-C)$$

Opcode Pattern

101110aa	baop	breg
----------	------	------

Flags Affected C, N, V, VT, Z

Examples

SUBCB	DELTA, #2
SUBCB	ALPHA, BETA
SUBCB	ALPHA, [GAMMA]+

TIJMP

Format

TIJMP wreg1,(wreg2),#ival

Operation

This instruction causes a jump to an address selected out of a table of addresses. The first operand, BASE, is a word register pointing to the base of the address table. The second operand, INDEX, is a word register pointing to an 8-bit index into the address table. The third operand, MASK, is an immediate value used with INDEX to calculate the offset, DEST, into the table, as follows:

$$\text{OFFSET} = [\text{INDEX}] \text{ AND } \text{MASK}$$
$$\text{DEST} = (2 * \text{OFFSET}) + [\text{BASE}]$$

Note

This instruction is specific to the 80C196KC microcontroller.

Opcode Pattern

11100010	wreg2	ival	wreg1
----------	-------	------	-------

Flags Affected

None

Examples

TIJMP 20H,[30H],#2

XCH

Format

XCH wreg, waop

Operation

This instruction exchanges the contents of two 16-bit (word) memory locations.

Note

This instruction is specific to the 80C196KC microcontroller.

Opcode Pattern

00000100	waop	wreg
----------	------	------

Flags Affected

None

Examples

XCH 20H, 30H

NOTE

It is illegal to use an immediate or indirect operand in an XCH instruction.

XCHB

Format

XCHB breg,baop

Operation

This instruction exchanges the contents of two 8-bit (byte) memory locations.

Note

This instruction is specific to the 80C196KC microcontroller.

Opcode Pattern

00010100	baop	breg
----------	------	------

Flags Affected

None

Examples

XCHB 20H,3000H

NOTE

It is illegal to use an immediate or indirect operand in an XCHB instruction.

XOR

Format

XOR wreg,waop

Operation

The source (rightmost) word operand is XORed with the destination (leftmost) word operand. Each bit is set to 1 if the corresponding bit in either the source operand or the destination operand was 1, but not both. The result replaces the original destination operand.

$(\text{DEST}) \leftarrow (\text{DEST}) \text{ XOR } (\text{SRC})$

$C \leftarrow \emptyset$

$V \leftarrow \emptyset$

Opcode Pattern

100001aa	waop	wreg
----------	------	------

Flags Affected

C, N, V, Z

Examples

XOR DELTA, #500
XOR ALPHA, BETA
XOR ALPHA, [GAMMA]+

XORB

Format

XORB breg,baop

Operation

The source (rightmost) byte operand is XORed with the destination (leftmost) byte operand. Each bit is set to 1 if the corresponding bit in either the source operand or the destination operand was 1, but not both. The result replaces the original destination operand.

$(\text{DEST}) \leftarrow (\text{DEST})$

$\text{XOR} \leftarrow (\text{SRC})$

$\text{C} \leftarrow \emptyset$

$\text{V} \leftarrow \emptyset$

Opcode Pattern

100101aa	baop	breg
----------	------	------

Flags Affected

C, N, V, Z

Examples

XORB DELTA, #2

XORB ALPHA, BETA

XORB ALPHA, [GAMMA]+

180X

180X

180X

The source of the information is not stated, but it is stated that the information is reliable. The source of the information is not stated, but it is stated that the information is reliable. The source of the information is not stated, but it is stated that the information is reliable.

180X

180X

180X

180X

180X

180X

180X

180X

180X

180X

4

Assembler Directives



Assembler directives allow you to specify auxiliary information (such as storage reservation, location counter control, definition of non-executable code, object code relocation, and flow of assembler processing) that determines the manner in which the ASM96 assembler generates object code from your source input.

The complete set of assembler directives can be grouped into categories as shown below. Each category and each of its elements is then discussed in detail, defined, and its syntax specified.

Directives resemble instructions in that directives also appear as program statements. Directives can be interspersed between instructions throughout your programs. Only one directive can appear on a given statement line.

The assembler directives are

- **module level:**

- MODULE
 - PUBLIC
 - EXTRN
 - END

- **location counter control:**

- CSEG
 - DSEG
 - RSEG
 - OSEG
 - ORG

- **symbol definition:**

- EQU
 - SET

- **code definition:**

- DCB
 - DCW
 - DCL
 - DCR

- **storage reservation:**

- DSB
 - DSW
 - DSL
 - DSR

- **conditional assembly:**

- IF
 - ELSE
 - ENDIF

4.1 Module Level

Module level directives provide information about the module as a whole, including inter-module information.

4.1.1 MODULE

Syntax

```
module_name MODULE [ attr,[,attr] ]
```

Where:

module_name identifies the module by defining the name under which the module is placed in the object file. If you omit a MODULE directive in your program, the default name for the module is the root of the source filename (i.e., the filename without any extension or device designation).

attr is either MAIN or STACKSIZE (*n*).

MAIN specifies that the module is to be of type MAIN. If you do not enter MAIN, the default is for the module to be of type non-MAIN.

STACKSIZE (*n*) defines the amount of stack in bytes required by the module. *n* must be an even number. The default (if you do not specify a STACKSIZE) is zero.

The MODULE directive can appear at most once in a program. It must appear at the beginning of the program (i.e., before non-control lines).

4.1.2 PUBLIC

Syntax

```
PUBLIC symbol_name [, . . . ]
```

Where:

symbol_name is the name of a symbol to be declared public.

PUBLIC is used to declare a symbol or symbols as public. Public symbols must have been defined somewhere in the program either by a symbol definition directive, or as a label. Declaring a symbol public allows other modules to access the symbol. Macro names, module names, reserved words, and external symbols cannot be declared public.

4.1.3 EXTERNAL

Syntax

```
EXTRN {symbol_name [:data_type] } [, . . . ]
```

Where:

symbol_name is the name of a symbol to be declared as external.

data_type must be BYTE, WORD, LONG, ENTRY, REAL or NULL.

The EXTRN directive declares one or more symbols to be external (i.e., located in other modules but referred from the present module). All such symbols must have been declared as public in the other modules. These symbols cannot be defined in the present program.

The segment type of the symbols is set to that of the active segment. For example, in order to define an external symbol of type code, the EXTRN directive that specifies the symbol must be placed within a segment of type code.

The *data_type* must match the segment type. Thus, BYTE, WORD, LONG, REAL, and NULL are allowed everywhere, while ENTRY is allowed only in a code segment.

In order to define an external symbol with NULL segment type, you must define it before the first segment selection directive in the program. After the first segment selection directive is issued, the segment type has been defined as either code, data, register, or overlay.

4.1.4 END

Syntax

END

Where:

END

must always be the final statement in a program. Any statement placed after the END directive is flagged as an error and ignored by the assembler.

4.2 Location Counter Control

Location counter control directives explicitly set the location counter.

4.2.1 Segment Selection (CSEG, DSEG, RSEG, and OSEG)

Syntax

{ CSEG | DSEG | RSEG | OSEG } [REL | AT *base_address*]

Where:

CSEG designates a code segment.

DSEG designates a data segment.

RSEG designates a non-overlayable register segment.

OSEG	designates an overlayable register segment.
REL	specifies that the segment is relocatable. REL is the default if neither REL nor AT is explicitly specified.
AT	specifies that the segment is absolute, beginning at address <i>base_address</i> .

The segment selection directives allow you to terminate one segment (the deactivated segment) and to initiate a new segment (code, data, register, or overlay) as specified in the directive. The program continues in the specified segment until a subsequent segment selection directive or the end of the program, whichever comes first, is encountered.

Specifying a relocatable segment by using the REL option continues the previous relocatable segment of that type, if any exists. If none exists, a relocatable segment of that type is created. Specifying AT creates an absolute segment that starts at the address given by *base_address*. If the deactivated segment was absolute, the segment selection directive closes that segment. The next selection of an absolute segment must create a new absolute segment. If the deactivated segment was relocatable, then its location counter is recorded so that it can be resumed when that relocatable segment is reselected.

Statements appearing within segments initiated by the segment selection directive are restricted according to the type of the segment as shown below.

Register segments (overlayable or not) and data segments cannot contain code definition directives or machine instructions.

Code segments cannot contain storage reservation directives.

Segment selection directives also specify the segment type of all external symbols specified within their scope (i.e., while the segment selected is active). Thus, the segment type of an external symbol is always that of the currently active segment.

Since there is no default initial value for the active segment, a segment selection directive must be placed before the first piece of memory is defined (i.e., before any machine instruction, code definition directive or storage reservation directive is entered).

4.2.2 ORG

Syntax

`ORG expression`

Where:

expression is an expression that can be evaluated to an address.

The ORG directive sets the value of the location counter equal to the expression evaluated. The expression cannot contain a forward reference. The segment type of the expression must be the same as the type of the active segment or null. For example, if the active segment is of type data, the expression must be of type data or null.

If the active segment is absolute, then the expression must also be absolute and represent some absolute memory address not below the *base_address* of the segment. If the active segment is relocatable, then the expression is interpreted as an offset from the beginning of the given segment and so the expression can be absolute or relocatable.

If the ORG directive points back into a code segment thus causing redefinition of previously defined code or data, unpredictable results can occur. Refrain from making such backward references.

Since the ORG directive does not create a new segment, it can create gaps in the segment. Such gaps are not filled by any other segments and remain after the relocation and linkage process has been executed.

4.3 Symbol Definition

4.3.1 EQU and SET

The directives EQU and SET are used to define symbols.

Syntax

```
symbol_name { EQU | SET } expression [ :data_type ]
```

Where:

symbol_name is the name of the symbol to be defined.

EQU is used for symbols that cannot be redefined.

SET is used for symbols that can be redefined.

expression defines the segment type, relocatability, value, and data type of the symbol.

data_type is BYTE, WORD, LONG, ENTRY, REAL, or NULL.

Both SET and EQU define the attributes (segment type, relocatability, value and data type) of the *symbol* to be that of *expression*. Symbols defined by EQU cannot be redefined elsewhere in the program, while symbols defined by SET can.

expression cannot contain any external symbols, but can contain one level of forward reference (i.e., any forward referenced symbol that appears in the defining *expression* must be defined in itself without any forward reference).

The optional *data_type* must match the segment type of the expression. Thus, BYTE, WORD, LONG, REAL, and NULL are allowed everywhere, while ENTRY is allowed only for a code segment.

4.4 Code Definition

Code definition directives define code to be inserted in the program beginning at the active location counter position. The code to be entered is specified by a list of expressions separated by commas. Code definition directives can appear only in code segments and nowhere else in the program.

4.4.1 DCB

Syntax

```
[label:] DCB {expression | string} [, . . . ]
```

Where:

label is a valid label.

expression is a valid expression.

string can be a string of *m* characters, where *m* can be any non-negative integer.

The DCB directive specifies a list of byte values (each represented by an expression) to be inserted one after the other starting at the position of the active location counter. The expressions can be relocatable or absolute, but must be in the range of -256 to +255 inclusive. Since the higher byte is ignored, sign information is lost for values outside the range of -128 to +127, inclusive. For relocatable expressions, the range check is made at link time.

The DCB directive is the only statement that can accept a null string or a string of more than two characters. A null string (i.e., length = zero) generates no code. A string of *m* characters is coded as a sequence of *m* bytes, each byte having the ASCII value of the respective character in the string.

If the location counter symbol (\$) appears in any of the expressions in the DCB directive, it is interpreted as having its value before the first byte in the list is processed. That value of the location counter is also used in defining the optional label that can be specified in the directive and for defining any previous labels if they are separated from the present statement by statements that do not affect the value of the location counter.

4.4.2 DCW

Syntax

`[label:] DCW expression [, . . . ]`

Where:

label is a valid label.

expression is a valid expression.

The DCW directive specifies a list of word values (each represented by an expression) to be inserted one after the other, starting at the position of the location counter. The location counter is adjusted by incrementing it by 1 if that is necessary in order for correct alignment (i.e., so that the first word begins on a word boundary).

The expressions can be relocatable or absolute. For each expression, its 16-bit value is stored with the low-order byte in the first (i.e., lower-order) address, followed by the high-order byte in the higher order address.

If the location counter symbol (\$) appears in any of the expressions, its value is its value before the first word in the list is processed. That value is also used for defining any previous labels if they are separated from the present statement by statements that do not affect the value of the location counter.

4.4.3 DCL

Syntax

`[label:] DCL / uns_number [, . . . ]`

Where:

`label` is a valid label.

`/ uns_number` is long unsigned number. The permissible range is 0 to $2^{32}-1$.

The DCL directive specifies a list of one or more long, unsigned numbers (not expressions) to be inserted one after the other, starting at the position of the location counter. The location counter is adjusted by incrementing it by one if that is necessary in order to correct alignment (i.e., so that the address of the first element begins on a word boundary). For each number, the four bytes are stored from the lowest-order byte to the highest-order byte in increasing addresses.

4.4.4 DCR

Syntax

`[label:] DCR real_number [, . . . ]`

Where:

`label` is a valid label.

`real_number` is a real number.

The DCR directive specifies a list of one or more real numbers (not expressions) to be inserted one after the other starting at the position of the location counter. The location counter is adjusted by incrementing it by one if that is necessary for correct alignment (i.e., so that the first element begins on a word boundary).

4.5 Storage Reservation

4.5.1 DSB, DSW, DSL, and DSR

Syntax

`[label:]{DSB | DSW | DSL | DSR} expression`

Where:

label is a valid label.

expression is a valid absolute expression of null segment type containing no forward references.

DSB specifies byte variables, no alignment required.

DSW specifies word variables, adjustment to a word boundary required.

DSL specifies long word variables, adjustment to a long word boundary (if in a register segment) or to a word boundary (if in a data segment) required.

DSR specifies real number variables, adjustment to word boundary required.

The storage reservation directives reserve storage for program variables. Specifying DSB defines storage for byte variables and no adjustment of boundaries is performed; DSW defines storage for word variables and adjustment for word boundaries is performed; DSL defines storage for long word variables and adjustment for word or long word boundaries is performed; and DSR defines storage for real numbers and adjustment to word boundaries is performed. The storage reservation directives cannot appear in a code segment.

The label, if specified, is set to the value of the location counter. This value is aligned as required by DSW, DSL, or DSR. That value is also used for defining previous labels if they are separated from this statement by statements that do not affect the value of the location counter.

Storage is reserved by incrementing the active location counter by *expression*n* where $n = 1$ (for DSB), 2 (for DSW), or 4 (for DSR or DSL). The *expression* must be absolute and of null segment type and contain no forward reference.

4.6 Conditional Assembly

The conditional assembly directives (IF, ELSE, and ENDIF) allow you to control the flow of the assembler processing of statements. These directives provide for conditional execution of statements based upon assembly-time evaluation of user defined expressions.

4.6.1 IF, ELSE, and ENDIF

Syntax

```
IF expression
statement(s)
[ELSE]
statement(s)
ENDIF
```

Where:

expression is a valid absolute expression that contains no forward references.

statement(s) is one or more valid statements.

The *expression* must be a valid absolute expression that contains no forward reference. If the expression in the IF line is evaluated and proves true (least significant bit = 1), the statements between IF and the optional ELSE or the mandatory ENDIF (whichever comes first) are assembled. If the IF line proves false (lsb = 0), any statements between the IF line and the optional ELSE line are ignored, and any statements between the ELSE line and the ENDIF are assembled. If the IF line proves false (lsb = 0), and you have not used the optional ELSE line, the assembler jumps to the ENDIF line, ignoring all statements between the IF line and the ENDIF line.

The statements between an IF directive and its ending ENDIF constitute a conditional assembly block. Conditional assembly directives can appear anywhere in a program, even within macros. However, if a conditional assembly block is initiated in a macro definition, its ENDIF must also lie in that macro.

When statements are not assembled due to conditional assembly directives they are effectively not in the program (though they can be listed by entering the proper assembler control). Therefore, any symbols defined in ignored statements are not available to the rest of the program. Avoid referring to any symbols that are defined in lines that can be ignored due to conditional assembly.

Conditional assembly blocks can be nested within one another (i.e., IF/[ELSE]/ENDIF blocks within another IF/[ELSE]/ENDIF block). Be sure to maintain congruence of IF lines and their respective ENDIF lines. The maximum depth of nesting is 9.

5

The Macro Processing Language



5.1 Introduction

A macro is essentially a facility for replacing one set of parameters with another. In developing your program, you will frequently find that many instruction sequences are repeated several times with only certain parameters changed.

For example, suppose you code a routine that moves five bytes of data from one memory location to another. A little later, you find yourself coding another routine to move four bytes from a different source field to a different destination field. If the two routines use the same coding techniques, you will find that they are identical except for three parameters: the character count, the source field starting address, and the destination field starting address. Certainly, it would be handy if there were some way to regenerate that original routine substituting the new parameters rather than rewrite the code. The macro facility provides this capability and offers several other advantages over simply rewriting the code every time it is needed.

5.1.1 The Advantages of Using Macros

Using macros provides you with four distinct advantages:

1. By eliminating the tedium of rewriting, using the macro minimizes the probability of entry error.
2. Symbols used in macros can be restricted so that they have meaning only within the macro itself. Therefore, as you code your program, you need not worry that you will accidentally duplicate a symbol used in a macro. Also, a macro can be used any number of times in the same program without duplicating any of its own symbols.

3. An error detected in a macro need be corrected only once regardless of how many times the macro appears in the program, thus reducing debugging time.
4. Increased efficiency occurs by reducing the duplication of effort by programmers. Frequently used macros can be made available to all programmers.

Macros also aid the development of structured programming and thereby help produce more readable documentation. Using macros to segment code blocks provides clear program notation and simplifies tracing the flow of the program.

5.1.2 An Example of Macro Use

A macro can be described as a routine defined by a formal sequence of prototype instructions that, when called within a program, results in the replacement of each such call with a code expansion consisting of the actual instructions represented. The following example illustrates macro definition, call, and expansion.

Suppose you desire to compose a typical business form letter that consists of a standard greeting and then lists the specific flight number, departure time, destination, and arrival time for the given specific passenger. A macro, CNFIRM, can be defined that

1. welcomes the passenger, then
2. lists the flight number (FNO), departure time (DTIME), arrival time (ATIME) at the destination (DEST).

The macro CNFIRM has four dummy parameters to be replaced. Thus the macro could be:

```
CNFIRM 13, "1:00", "Bamboolah", "11:00"
```

The other macro, GREET, is the standard message. Its only parameter is the passenger's name. Thus the entire of the text of the message can be placed in a source file in the form:

```
GREET "Mr.  Bozo"  
CNFIRM 13, "1:00", "Bamboolah", "11:00"  
We trust you will enjoy the dream trip of your life.  
Sincerely,  
D.  Warbucks, Manager Air Freight
```

After the sourcefile is passed through a macro processor and the macro calls are expanded, the following letter is produced:

```
Dear Mr. Bozo:  
Peninsula-Panhandle Airlines welcomes you as a passenger.  
We are pleased to confirm your reservation on Flight #13  
which leaves at 1:00 and arrives in Bamboolah at 11:00  
(local time).  
We trust you will enjoy the dream trip of your life.  
Sincerely,  
D.  Warbucks, Manager Air Freight
```

While this example illustrates the substitution of parameters in a macro, it overlooks the relationship of the macro processor to the assembler. The purpose of the macro processor is to generate source code which is then assembled.

5.1.3 Macros and Subroutines

How do macros differ from subroutines invoked by the `CALL` instruction? First, subroutines necessarily branch to another part of your program, while macros generate in-line code. Thus a program contains only one version of a given subroutine, but it contains as many versions of a given macro as there are calls for that macro. Second, the macro does not always generate the same source code; a subroutine does. By changing the parameters in a macro call, you can change the source code that the macro generates. Thus, the macro is more of a general purpose tool used to generate customized source code as required by the particular programming situation. Third, macro expansion and code customization occur at assembly time and at the source code level. Subroutines, on the other hand, reside within the program and require additional execution time for parameter passing and transfer of control.

The choice between using a macro and a subroutine is dependent on the particular situation. The main tradeoff is between the additional space required by the macro on one side, and on the other side, the additional time required to pass parameters and transfer control to a subroutine. In some cases, a single subroutine is more efficient than multiple in-line macros. In situations involving a large number of parameters, the use of macros can prove more efficient. Also keep in mind that macros can call subroutines and subroutines can contain macros. Whereas both subroutines and macros can be used to implement common code, only macros can be used to generate common data structures.

5.2 Macro Directives and Macro Calls

The assembler recognizes the following macro operations:

- `MACRO` directive
- `ENDM` directive
- `LOCAL` directive
- `REPT` directive
- `IRP` directive
- `IRPC` directive

- EXITM directive
- macro call

All of the preceding directives except the macro call are related to macro definition. The macro call initiates the parameter substitution (macro expansion) process.

5.2.1 Macro Definition

Macros must be defined in your program before they can be used. A macro definition is initiated by the `MACRO` directive, which lists the name by which the macro can later be called and the dummy (or formal) parameters to be replaced during macro expansion. The macro definition is terminated by the `ENDM` directive. The prototype instructions, bounded by the `MACRO` and the `ENDM` directives, are called the macro body.

When the label symbols used in a macro body have global scope, multiple defined symbol errors result if the macro is called more than once. A label can be given limited scope using the `LOCAL` directive. This directive assigns a unique value to the symbol each time the macro is called and expanded. Dummy (formal) parameters also have limited scope.

Occasionally you may wish to duplicate a block of code several times either within a macro or in line with other source code. This can be accomplished with minimal coding effort using the `REPT` (repeat block), `IRP` (indefinite repeat), and the `IRPC` (indefinite repeat character) directives. Like the `MACRO` directive, these directives are terminated by `ENDM`. The `EXITM` directive provides an alternate exit from a macro. When encountered, it terminates the current macro just as if `ENDM` had been encountered.

Macro directives cannot be prefixed by a label. However, a label can prefix a macro call statement.

5.3 MACRO Directives

5.3.1 MACRO

Syntax

name MACRO [*dummy_parameters*]

Where:

name

specifies the name of the macro body being defined. Any valid user-defined symbol name can be used as a macro name.

dummy_parameters

can be from zero to sixteen (16) user-defined symbol names. When multiple parameters are used they must be separated by commas. The scope of a dummy parameter is limited to its specific macro definition. Reserved words cannot be used as dummy parameter names. Dummy parameters in a comment are not recognized. No substitution occurs for such parameters. Dummy parameters can, however, appear in a character string. In this case, you must use the ampersand as explained in the Special Macro Operators section in this chapter.

Any machine instruction or applicable assembler directive can be included in the macro body. The distinguishing feature of macro prototype text is that parts of it can be made variable by placing substitutable dummy parameters in instruction fields. These dummy parameters are the same as the symbols in the operand field of the MACRO directive.

Example

```
BIGMAC  MACRO  G1,G2,G3      ; MACRO directive
        LD     Reg1, G1      ; Start of macro body
        LD     Reg2, #G2
        LD     Reg3, G3
LOOP:   LD     Reg4, [Reg2]+
        ST     Reg4, [Reg3]+
        DBNZ   Reg1, LOOP    ; end of macro body
        ENDM                ; ENDM directive
```

This defines a MACRO called BIGMAC that contains the dummy parameters G1, G2, and G3. Note that G1, G2, and G3 cannot be operands of a LOCAL directive. Macro names cannot be redefined.

5.3.2 ENDM

Syntax

```
ENDM
```

The ENDM directive is required to terminate a macro definition. It follows the last prototype instruction. ENDM is also required to terminate code repetition blocks defined by the REPT, IRP, and IRPC directives. If any characters are placed as a label or operand to the ENDM directive, an error is reported. Because nested macro calls are not expanded during macro definition, the ENDM directive closing an outer macro cannot be contained in the expansion of an inner, nested macro call. For further information on nesting macro calls, refer to the Macro Calls section in this chapter.

5.3.3 LOCAL

Syntax

```
LOCAL  local_symbol_name [, . . . ]
```

Where:

local_symbol_name is the name of a label to be defined only within the current macro expansion.

The specified label names are defined and are meaningful only within the current macro expansion. Each time the macro is called or expanded, the assembler assigns each local symbol a unique global symbol in the form `??nnnn`, where `nnnn` is `0001` for the first local symbol in the program, `0002` for the second, and so on. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions. The assembler never duplicates these symbols. You should refrain from defining symbols that begin with `??` to avoid any conflict.

Dummy parameters included in a macro call cannot be operands of a `LOCAL` directive. A dummy parameter is always restricted to its own macro definition. Local symbols can be defined only within a macro definition. A maximum of sixteen local symbols can appear in a macro definition. They must all appear between the `MACRO` directive and the first line of prototype code.

Example

```
BIGMAC  MACRO  G1,G2,G3
        LOCAL  LOOP
        LD     Reg1, G1
        LD     Reg2, #G2
        LD     Reg3, #G3
LOOP:   LD     Reg4, [Reg2]+
        ST     Reg4, [Reg3]+
        DBNZ   Reg1, LOOP
        ENDM
```

By declaring `LOOP` as a local symbol, one eliminates the possibility of error arising from `BIGMAC` being called twice (making `LOOP` multiple-defined) or from the symbol `LOOP` being defined elsewhere in the code.

5.3.4 REPT

Syntax

REPT *expression*

Where:

expression is a valid absolute expression of null segment type containing no forward references.

The REPT directive causes a sequence of source code lines to be repeated *expression* times. All lines appearing between the REPT directive and subsequent ENDM directive constitute the block to be repeated.

The insertion of repeat blocks is performed in-line as the assembler encounters the REPT directive. No explicit call is required to cause the code insertion as the definition is an implied call for expansion.

Example

```
REPT 6
LD   Reg1, [Src_Reg]+
ST   Reg1, [Dst_Reg]+
ENDM
```

This macro moves the 6 words pointed at by Src_Reg to the memory area pointed at by Dst_Reg.

5.3.5 IRP

Syntax

`IRP dummy_parameter, <actual_parameters>`

Where:

dummy_parameter is a single dummy parameter.

actual_parameters is a list of one or more actual parameters, separated by commas. The list of actual parameters must be enclosed within angle brackets (<>).

The IRP directive causes an indefinite repeat. The macro body is executed one time for each actual parameter. Each time the macro body is executed, the successive actual parameters are substituted for the dummy parameter. If no actual parameters are specified, the macro body is executed one time, with a null used for the parameter. If two commas are placed adjacent to each other with no actual parameter intervening, a null is substituted for the actual parameter.

Example

```
LD    Reg1, #STORIT
IRP   X,<FLD1,#3E20H,FLD3>
LD    Reg2, X
ST    Reg2, [Reg1]+
ENDM
```

This macro generates the code:

```
LD    Reg1, #STORIT
LD    Reg2, #FLD1
ST    Reg2, [Reg1]+
LD    Reg2, #3E20H
ST    Reg2, [Reg1]+
LD    Reg2, FLD3
ST    Reg2, [Reg1]+
```

That is, the command set LD and ST is executed successively for each of the actual parameters, FLD1, #3E20H, and FLD3.

5.3.6 IRPC

Syntax

`IRPC dummy_parameter, actual_parameter`

Where:

dummy_parameter is a single dummy parameter.

actual_parameter is sequence of characters.

The IRPC directive causes the macro body to be executed one time for each character of the character list. On each successive execution, a successive character of the list is substituted for the dummy parameter. If the character list is null, one execution of the macro body is performed with a null substituted for the dummy parameter. If you wish to place delimiters in the list to be substituted for the dummy parameter, you must be sure to enclose the list in angle brackets (<>).

Example

```
CLR    Reg1
IRPC   X, 1982
MUL    Reg1, #10
ADD    Reg1, #X
ENDM
```

Generates the code:

```
CLR    Reg1
MUL    Reg1, #10
ADD    Reg1, #1
MUL    Reg1, #10
ADD    Reg1, #9
MUL    Reg1, #10
ADD    Reg1, #8
MUL    Reg1, #10
ADD    Reg1, #2
```

Thus, Reg1 repeatedly accumulates the result of the conversion of 1982 to internal form.

5.3.7 EXITM

Syntax

EXITM

EXITM provides an alternate method of terminating a macro expansion or terminating the repetition of a REPT, IRP, or IRPC code sequence. As soon as the assembler recognizes EXITM, it ignores all macro prototype instructions located between the EXITM and the ENDM directive for this macro. EXITM can be used in addition to but not in place of ENDM.

When used in nested macros, EXITM causes an exit to the previous level of macro expansion. An EXITM within a REPT, IRP, or IRPC terminates not only the current expansion, but all subsequent iterations as well.

Any characters appearing as an operand to the EXITM directive will cause an error.

Example

```
MAC3      MACRO    X,Y
          .
          .
          .
          IF X EQ 0
          EXITM
          .
          .
          .
          ENDM
```

The macro expansion process is terminated as soon as the X equals 0. So long as X is not equal to 0 the line beneath that statement is skipped so the EXITM is ignored. When X equals 0, the EXITM is executed.

5.4 Special Macro Operators

The special macro operators allow you to prevent any ambiguity in specifying parameters to the assembler. Thus, for example, if you wanted to specify three actual parameters of which the third is a comma, entering `PARAM_1 , , , PARAM_2` appears to the assembler as a list of four parameters with the second and third parameters missing. To avoid this ambiguity the comma should be prefixed by an exclamation mark (`PARAM_1 , ! , , PARAM_3`). Placing the comma (or any other character) after the exclamation mark causes the assembler to treat the character as a literal, possessing no special significance.

When a macro is expanded, any ampersand preceding or following a dummy parameter in macro definition is removed and the substitution of the actual parameter occurs at that point. When it is not adjacent to a dummy parameter, the ampersand is not removed and is passed as part of the macro expansion text. Note that the ampersand must be immediately adjacent to the text being concatenated; intervening blanks are not allowed.

If nested macro definitions contain ampersands, the only ampersands removed are those adjacent to dummy parameters belonging to the macro definition currently being expanded. All ampersands must be removed by the time the expansion macro body is performed. Exceptions force illegal character errors. Ampersands are not recognized as operators in comments.

5.5 Nesting Macro Definitions

A macro definition can be contained completely (nested) within the body of another macro definition. The body of a macro consists of all text (including any nested macros) bounded by matching `MACRO` and `ENDM` directives. The assembler allows any number of macro definitions to be nested.

When a higher-level macro is called for expansion, the next lower-level macro cannot be called unless all higher-level macro definitions have already been called and expanded.

Note that since macro names cannot be redefined, the nested macro definition should have the macro name specified as a local symbol (unless the enclosing macro is called only once).

In addition to the exclamation mark, the special macro operators are listed and defined in Table 5-1.

Table 5-1 Macro Operators

Operator	Definition
&	The ampersand specifies concatenation (linking) of text and dummy parameters.
<>	Angle brackets undefine delimiters that you want treated as literal characters instead. Since blanks are delimiters, literal blanks must be enclosed in angle brackets. To pass text to nested macro calls, use one set of angle brackets for each level of nesting.
::	Double semicolons placed before a comment in a macro definition prevent the inclusion of the comment in expansions of the macro and thereby reduce storage requirements. The comment will still appear in the listing of the macro definition.
!	The exclamation mark is placed before a character (usually a delimiter) that is to be passed as literalized text in an actual parameter. To pass a literalized exclamation point, enter two exclamation points (!!). The exclamation point is not preserved while building an actual parameter (e.g., when substitution is used to build another actual parameter). Carriage returns cannot be passed as actual parameters.
NUL	The NUL is used to indicate the omission of a parameter. The omitted (or null) parameter can be represented by two consecutive delimiters (e.g., PARAM_1,,PARAM_3). A null parameter can also be represented by two consecutive single quotes (e.g., ",PARAM_2,PARAM_3). Notice that a null is not a blank. The blank is the ASCII character represented by 20 hex. NUL has no character representation. A typical use of the NUL operator is in IF directives; e.g., <pre> IF NUL X&Y EXITM ENDIF </pre> causes macro expansion to terminate if both the actual arguments of X and Y are null.

Since `IRP`, `IRPC`, and `REPT` blocks constitute macro definitions, they also can be nested within another definition created by `IRP`, `IRPC`, `REPT`, or `MACRO` directives. In addition, an element in an `IRP` or `IRPC` actual parameter list (enclosed in angle brackets) can itself be a list of bracketed parameters; that is, lists of parameters can contain elements that are also lists. For example,

```
LISTS      MACRO    PARAM1,PARAM2
            .
            .
            .
            ENDM
            .
            .
            .
LISTS <A,<B,C>>
```

5.6 Macro Calls

Once a macro has been defined, it can be called any number of times in the program. The call consists of the macro name and any actual parameters that are to replace dummy parameters during macro expansion. During assembly, each macro call is replaced by the macro definition code; dummy parameters are replaced by actual parameters.

5.6.1 Macro Call

Syntax

```
[label:] macro_name [actual_parameters]
```

Where:

label is a user-defined label.

macro_name is a user-defined name for the macro.

actual_parameters is a list of up to 16 actual parameters.

The assembler must encounter the macro definition before the first call for that macro. Otherwise, an error occurs. The assembler inserts the macro body identified by the macro name each time it encounters a call to a previously defined macro in your program.

The positioning of actual parameters in a macro call is critical since the substitution of parameters is based solely on position. The first-listed actual parameter replaces each occurrence of the first-listed dummy parameter; the second actual parameter replaces the second dummy parameter, and so on. When coding a macro call, you must be certain to list actual parameters in the appropriate sequence for that macro.

Notice that blanks are usually treated as delimiters. Therefore, when an actual parameter contains blanks (passing the instruction LD Reg1, [Reg2] for example), the parameter must be enclosed in angle brackets. This is also true for any other delimiter that is to be passed as part of an actual parameter. Carriage returns cannot be passed as actual parameters.

If a macro call specifies more actual parameters than are listed in the macro definition, an error occurs. If fewer parameters appear in the call than in the definition, a null replaces each missing parameter.

The following example shows two calls for the macro BLOCK, defined as follows:

```
BLOCK    MACRO    G1,G2,G3,G4
          LOCAL    LOOP, EOM
          LD        Reg1, G1
          LD        Reg2, #G2
LOOP:    G3        Reg3, [Reg2]+
          G4        EOM
          DBNZ      Reg1, LOOP
EOM:
          ENDM
```

BLOCK operates on the G1 words starting at G2 by the opcode G3. Then it conditionally performs the branch specified by G4.

The first time BLOCK is called, it is used to get the average of the list of words making up the block. The second time BLOCK is called, it is used to find the first word greater than or equal to that average.

MAIN PROGRAM

```
CLR    Reg3
BLOCK  Size, Tab, ADD, \;
```

```
??0001:
```

```
??0002:
```

```
CLR    Reg 3+2
DIV     Reg3, #Size
BLOCK  Size, Tab, CMP, BGE
```

```
??0003:
```

```
??0004:
```

SUBSTITUTION

```
CLR    Reg3
LD      Reg1, Size
LD      Reg2, #Tab
ADD     Reg3, [Reg2]+
;       ??0002
DBNZ    Reg1, ??0001
```

```
CLR    Reg 3+2
DIV     Reg3, #Size
LD      Reg1, #Size
LD      Reg2, #Tab
CMP     Reg3, [Reg2]+
BGE     ??0004
DBNZ    ??0003
```

5.6.2 Nested Macro Calls

Macro calls (including any combination of nested IRP, IRPC, and REPT constructs) can be nested within macro definitions up to nine levels. The macro being called need not be defined when the enclosing macro is defined or called. However, it must be defined before it is called.

A macro definition can also contain nested calls to itself (recursive macro calls) up to nine levels, as long as the recursive macro expansions can be terminated eventually. This operation can be controlled using the conditional assembly directives described in Chapter 4 of this manual.

For example, the macro below, called `RECALL`, calls itself five times after it is first called from elsewhere in the program.

```
PARAM1 SET 5
RECALL MACRO
.
.
.
IF PARAM1 NE 0
PARAM1 SET PARAM1 -1
RECALL ; RECURSIVE CALL
ENDIF
.
.
.
ENDM
```

5.7 Macro Expansion

When a macro is called, the actual parameters to be substituted into the prototype code can be passed in one of two modes. Normally, the substitution of actual parameters for dummy parameters is simply a text substitution. The parameters are not evaluated until the macro is expanded.

If a percent sign (%) precedes the actual parameter in the macro call, however, the parameter is evaluated immediately, before expansion occurs, and is passed as a decimal number representing the value of the parameter. In the case of `IRPC`, a percent sign preceding the actual parameter causes the entire text string to be treated as a single parameter. One `IRPC` iteration occurs for each digit in the decimal string passed as the result of immediate evaluation of the text string.

The normal mechanism for passing actual parameters is adequate for most applications. Using the percent sign to pre-evaluate parameters is necessary only when the value of the parameter is different within the local context of the macro definition as compared to its global value outside the macro definition.

The following macro generates a number of multi-byte arithmetic instructions. The parameters passed in the macro call determine the number of bytes in an operand, and whether an addition or subtraction operation is to be performed. The addresses of the source and destination operand are assumed to be in Reg1 and Reg2, respectively. Some typical calls for this macro follow.

```

MBYTE    'A',5
MBYTE    'S',%COUNT-1

```

Note that the second call shows an expression used as a parameter. This expression is to be evaluated immediately rather than passed simply as text.

The definition of the MBYTE macro follows. This macro uses the conditional IF directive to test the validity of the first parameter. Also, the REPT macro directive is nested within the MBYTE macro.

```

MBYTE    MACRO          X,Y
          IF (X NE 'A') AND (X NE 'S')
              EXITM
          ENDIF
          CLRC
          REPT Y
              LDB          Reg3,[Reg1]
              IF X EQ 'A'
                  ADDCB Reg3,[Reg2]+
              ELSE
                  SUBCB Reg3,[Reg2]+
              ENDIF
              STB          Reg3,[Reg1]+
          ENDM
      ENDM

```

The indentation shown in the definition of the MBYTE macro graphically illustrates the relationship of the IF, ELSE, ENDIF directives and the REPT and ENDM directives. Such indentation is not required in your program, but serves as a helpful documentation aid.

The MBYTE macro generates nothing if the first parameter is neither A nor S. Therefore, the following calls produce no code. The result in the object program is as though the MBYTE macro does not appear in the source program.

```

BYTE     5
MBYTE    'B',5

```

The following is another form of the MBYTE macro. In this definition, notice that concatenation is used to form the ADDCB or SUBCB operation code. If a call to the MBYTE macro specifies a string other than ADD or SUB, illegal operation codes are generated. The assembler flags all operation codes as undefined symbols.

```

MBYTE    MACRO    X,Y
          CLRC
          REPT Y
          LDB      Reg3,[Reg1]
          X&CB    Reg3,[Reg2]+
          STB      Reg3,[Reg1]+
          ENDM
        ENDM

```

5.8 Null Macros

A macro can legally comprise only the MACRO and ENDM directives. Thus, the following is a legal macro definition:

```

NADA     MACRO    P1,P2,P3,P4
          ENDM

```

A call to this macro produces no source code and therefore has no effect on the program.

Although there is no reason to write such a macro, the null (or empty) macro body has a practical application. For example, all the macro prototype instruction might be enclosed with IF-ENDIF conditional directives. When none of the specified conditions is satisfied, all that remains of the macro is the MACRO directive and the ENDM directive.

5.9 Sample Macros

The following examples further demonstrate the use of macro directives and operators.

Example 1: Nesting of IRPC Within a Macro

The following macro definition contains a nested IRPC directive. Notice that the third operand of the outer macro becomes the character string for the IRPC.

```
MOVE      MACRO  X,Y,Z
           IRPC   PARAM,Z
           LD     Reg1,X&PARAM
           ST     Reg1,Y&PARAM
           ENDM
           ENDM
```

Assume that the program contains the call `MOVE SRC,DST,123`. The third parameter of this call is passed to the IRPC. This has the same effect as coding `IRPC PARAM,123`. When expanded, the `MOVE` macro generates the following source code:

```
LD     Reg1,SRC1
ST     Reg1,DST1
LD     Reg1,SRC2
ST     Reg1,DST2
LD     Reg1,SRC3
ST     Reg1,DST3
```

Notice the use of concatenation to form labels in this example.

Example 2: Nested Macros Used to Generate DCB Directives

This example generates a number of DCB directives, each with its own label. Two macros are used for this purpose: GENLAB and BLOCK. The GENLAB macro is defined as follows:

```
      GENLAB    MACRO    F1,F2
$  SAVE          GEN
      F1&F2:    DCB      Ø      ; GENERATE LABELS & DCB'S
$  RESTORE
      ENDM
```

The BLOCK macro, which accepts the number of DCB's to be generated (NUMB) and a label prefix (PREFIX), is defined as follows:

```
      BLOCK    MACRO    NUMB,PREFIX
$  SAVE          NOGEN
      COUNT    SET      Ø
      REPT     NUMB
      COUNT    SET      COUNT+1
      GENLAB   PREFIX,%COUNT ; NESTED MACRO CALL
      ENDM
$  RESTORE
      ENDM
```

The macro call BLOCK 3, LAB generates the following source code:

```
      BLOCK    3,LAB
LAB1:    DCB      Ø
LAB2:    DCB      Ø
LAB3:    DCB      Ø
```

The assembler controls specified in these two macros (the lines beginning with \$) are used to clean up the assembly listing for easier reading. The source code shown for the call BLOCK 3, LAB is what appears in the assembly listing when the controls are used. Without the controls, the assembly listing appears as follows:

```

COUNT    BLOCK    3, LAB
          SET      0
          REPT     3
COUNT    SET      COUNT+1
          GENLAB   LAB, %COUNT
          ENDM
COUNT    SET      COUNT+1
          GENLAB   LAB, %COUNT
LAB1:     DCB      0
COUNT    SET      COUNT+1
          GENLAB   LAB, %COUNT
LAB2:     DCB      0
COUNT    SET      COUNT+1
          GENLAB   LAB, %COUNT
LAB3:     DCB      0

```

Example 3: A Macro that Converts Itself into Subroutine

In some cases, the in-line coding substituted for each macro call imposes an unacceptable memory requirement. The next two examples show two different methods for converting a macro call into a subroutine call. The first time the SBMAC macro is called, it generates a full line substitution that defines the SUBR subroutine. Each subsequent call to SBMAC generates only a CALL instruction to the SUBR subroutine.

Within the following examples, notice that the label SUBR must be global so that it can be called from outside the first expansion. This is possible only when that part of the macro definition containing the global label is called only once in the entire program.

Method #1 - Conditional Assembly. This method of altering the expansion of the sample macro, SBMAC, uses conditional assembly. The variable FIRST is used as a switch based upon whether the value of FIRST when tested in TRUE or FALSE. FIRST is set TRUE just before the first call for SBMAC. SBMAC is defined as follows:

```

TRUE      EQU      0FFH
FALSE     EQU      0
FIRST     SET      TRUE
SBMAC     MACRO
          CALL SUBR
          IF      FIRST
FIRST     SET      FALSE
          BR      DUN
SUBR:     .
          .
          .
          RET
DUN:
          ENDIF
          ENDM

```

The first call to SBMAC expands the full definition, including the call to and definition of SUBR:

```

          SBMAC
          CALL SUBR
          IF      FIRST
          BR      DUN
SUBR:     .
          .
          .
          RET
DUN:
          ENDIF

```

Because FIRST is TRUE when encountered during the first expansion of SBMAC, all the statements between IF and ENDIF are assembled into the program. In subsequent calls, the conditionally-assembled code is skipped so that the subroutine is not regenerated. Only the following expansion is produced:

```

          SUBMAC
          CALL SUBR
          IF      FIRST

```

Method #2 - Conditional Assembly with EXITM. The other method of altering the expansion of SBMAC also used conditional assembly, but used the EXITM directive to suppress unwanted macro expansion after the first call. EXITM is effective when FIRST is FALSE, which it is after the first call to SBMAC.

```

TRUE      EQU      0FFH
FALSE     EQU      0
FIRST     SET      TRUE
SBMAC     MACRO
           CALL     SUBR
           IF       NOT FIRST
           EXITM
           ENDIF
FIRST     SET      FALSE
SUBR:     BR        DUN
           .
           .
           .
           RET
DUN:
           ENDM

```

Example 4: Computed GOTO Macro

This sample macro presents an implementation of a computed GOTO for the 8096. The computed GOTO, a common feature of many high-level languages, allows the program to jump to one of a number of different locations depending on the value of a variable. For example, if the variable has the value 0, the program jumps to the first item in the list; if the variable has the value 3, the program jumps to the fourth item, and so on. In this example, the first parameter is the address of the branch table (specified via a sequence of DCW's), and the second is the table index.

```

TJUMP     MACRO    I,T                ; JUMP TO I-TH ADDR IN TABLE T
           LD       Reg1,I             ; COMPUTE OFFSET OF TABLE ENTRY
           ADD      Reg1, Reg1         ; "
           LD       Reg1, T[Reg1]      ; GET ADDRESS OF ENTRY
           BR       [Reg1]            ; INDIRECT BRANCH
           ENDM

```


Example 5: Using IRP to Define the Jump Table

The TJUMP macro becomes even more useful when a second macro (GOTO) is used to define the jump table, load the address of the table, and then call TJUMP. The GOTO macro is defined as

```
GOTO      MACRO      INDEX,LIST
          LOCAL      JTAB
          TJUMP      INDEX,JTAB      ; LD ACCUM WITH INDEX
JTAB:
          IRP      FORMAL,<LIST>
          DCW      FORMAL      ; SET UP TABLE
          ENDM
          ENDM
```

A typical call to the GOTO macro would be as follows:

```
GOTO      CASE,<COUNT,TIMER,DATE,PTR>
```

This call to the GOTO macro builds a table of DCW directives for the labels COUNT, TIMER, DATE, and PTR. It then calls the TJUMP macro. If the value of the variable CASE is 2 when the GOTO macro is called, the GOTO and the TJUMP macros together cause a jump to the address of the DATE routine.

Notice that any number of addresses can be specified in the list for the GOTO routine as long as they all fit on a single source line.

6

Assembler Operation and Controls



This chapter describes how to invoke the MCS[®]-96 Macro Assembler from the DOS operating system. It also describes the assembler controls in detail.

6.1 How to Invoke the MCS[®]-96 Macro Assembler

The general syntax of the command to invoke the assembler is

```
[directory | device]ASM96 source_file [controls]
```

Where:

source_file is the pathname of the file containing the ASM96 source module.

controls is an optional sequence of assembler controls. The controls are described in detail later in this chapter.

ASM96 normally produces two output files. One contains a formatted listing of your source code. Unless you specify a particular filename with the PRINT control, the formatted listing resides on the same device and has the same name as the source file but with the extension .LST. For more information about the listing file, see Chapter 7. Similarly, unless you specify a particular filename with the OBJECT control, the object file produced by the assembler resides on the same disk and has the same name as the source file but with the extension .OBJ.

To illustrate the construction and function of the invocation line, consider the following example:

```
>ASM96 A:PROG.SRC
```

Here a sourcefile, PROG.SRC on device A:, is to be assembled by ASM96. Since no controls have been specified, the listing file has the name PROG.LST and the object file has the name PROG.OBJ, the default conditions. Both the listing and the object file reside on device A:. Default conditions are also in effect for all the other control options.

You can continue the invocation line on one or more additional lines by entering the ampersand (&) before you enter the carriage return/line feed. The next line then automatically appears with the continuation prompt. Comments can also be entered on the invocation line by placing the comments after the ampersand because the compiler ignores all characters that appear after the ampersand but before the carriage return/line feed that terminates the line, as in the following example:

```
> ASM96 PROG.SRC & The assembler is being run
<< TITLE('PROJECT REVIEW') & for the Project Review file.
<< GEN
```

Refer to your DOS User's Guide for information on submitting batch file commands.

If errors are detected in the invocation line they are fatal and the assembler aborts without processing the source program.

6.2 Specifying Assembler Controls

Assembler controls can be entered in the invocation line as described previously or on a control line in your source code as described below. The general format for a control line is

```
$Control_List [;Comment]
```

The dollar sign (\$) must be the first character of any control line.

Control-List is a list of one or more controls separated by one or more spaces or tabs. Comments are optional. Control lines must always be terminated by carriage return/line feed. Control lines cannot be continued. Control lines can contain comments. The comments begin with a semicolon (;) and continue to the end of the line.

For example,

```
$ PAGEWIDTH (80) PAGELENGTH(72)
$TITLE ('SECTION EIGHT') EJECT; Section 8 listing follows
```

sets up a listing file with a pagewidth of 80 characters/line and a pagelength of 72 lines per page to be titled SECTION EIGHT and to be started on a new page (because of EJECT). Note that parameters to commands, such as 80 for PAGEWIDTH or 'SECTION EIGHT' for TITLE, are placed inside parentheses.

Some controls specify conditions that are either positive or negative (on or off). The negative conditions are specified by prefixing NO to the positive form of the control. Thus the negative form of XREF is NOXREF and the negative form of PRINT is NOPRINT.

All positive forms of controls have two letter abbreviations by which they can be specified. Thus the abbreviation for XREF is XR and the abbreviation for PRINT is PR. The abbreviations for the negative forms of the controls consist of the two-letter abbreviations for the positive form of the command prefixed by NO. For example, NOPR is the abbreviation for NOPRINT as NOXR is the abbreviation for NOXREF.

All primary and some general controls have default settings. These defaults are automatically used by the assembler unless you specify otherwise. Thus it is necessary for you to enter controls only if you desire assembly conditions different from the default conditions.

Controls and control parameters, both in the invocation line and in source lines, can be entered either in uppercase or in lowercase letters.

Table 6-1 lists all the controls and their abbreviations. The default settings are shown where applicable. Following the table is a detailed description of each of the controls.

Table 6-1 ASM96 Macro Assembler Controls

Control Name	Abbreviation	Default
Listing Primary Controls		
ERRORPRINT [(<i>pathname</i>)]/NOERRORPRINT	EP/NOEP	NOEP
PAGELength (<i>n</i>)	PL	PL(60)
PAGewidth (<i>n</i>)	PW	PW(120)
PRINT [(<i>pathname</i>)]/NOPRINT	PR/NOPR	PR(<i>source.LST</i>)
SYMBOLS/NOSYMBOLS	SB/NOSB	SB
XREF/NOXREF	XR/NOXR	NOXR
Object Primary Controls		
DEBUG/NODEBUG	DB/NODB	NODB
OBJECT [(<i>pathname</i>)]/NOOBJECT	OJ/NOOJ	OJ(<i>source.OBJ</i>)
General Controls		
COND/NOCOND	CO/NOCO	CO
EJECT	EJ	—
GEN/NOGEN	GE/NOGE	NOGE
INCLUDE(<i>pathname</i>)	IC	—
LIST/NOLIST	LI/NOLI	LI
SAVE/RESTORE	SA/RS	—
TITLE('string')	TT	TT(<i>modulename</i>)

6.2.1 Types of Controls

Controls can be classified into primary controls (further divided into listing primary controls and object primary controls) and general controls.

Primary controls are set at the beginning of the assembly process and cannot be changed during the assembly. Primary controls must be specified either in the invocation line or in primary control lines. A primary control line is any control line that appears before the first non-control line of the source program. Blank lines and comment lines are considered to be non-control lines. A primary control can appear only once in a given program (the presence of either the positive or the negative form of a given control is counted as the first appearance). Listing primary controls determine the construction of the listing file, while object primary controls determine the construction of the object file.

General controls can be specified in the invocation line or on control lines anywhere in the source file and can appear any number of times in the program (except for the `INCLUDE` control, which can appear at most once in a line and only as the rightmost entry of the line). General controls either cause an immediate action (e.g., `EJECT`, `INCLUDE`) or an immediate change of conditions (e.g., `LIST`, `GEN`). In the latter case, the condition specified by the general control remains in effect until another general control causes it to change. In either case, the immediacy of the result is constrained to mean a general control takes effect at the end of the control line. General controls specified in the invocation line take effect before the first source line is read.

6.3 Control Processing

The assembler processes controls in the following manner. Initially, the assembler is set for the default value for each control. The program text is then processed from left to right, starting at the invocation line and then going from the first source line to the final source line.

Each primary control that is encountered is then set to the condition specified. Any further occurrence of the same primary control produces an error message to that effect and the subsequent occurrence does not affect the assembler processing.

Except for LIST and EJECT, general controls are set as they are scanned and take the last condition specified as the assembler scans left to right along the line. Thus, the condition used by the assembler is the last one it encounters (i.e., the rightmost one), unless a RESTORE is inserted in the middle. If a control line contained (reading from left to right) the following, NOGEN is in effect:

```
$GEN NOGEN GEN NOGEN GEN NOGEN
```

Keep this in mind for controls such as GEN that have negative forms should you enter them more than once on the line. LIST and EJECT become effective only after the current line is printed.

6.4 Description of Controls

An alphabetical listing of assembler controls and their descriptions appear on the following pages.

COND/NOCOND

Syntax

COND
NOCOND

Type

General

Abbreviation

CO/NOCO

Default

COND

Description

COND specifies that conditionally skipped source lines (i.e., those lines that were "skipped over" and not assembled because of IF/ELSE directives) nonetheless appear in the listing (provided that LIST is in effect).

DEBUG/NODEBUG

Syntax

DEBUG
NODEBUG

Type

Object Primary

Abbreviation

DB/NODB

Default

NODEBUG

Description

DEBUG specifies that user symbol table information is to be output to the object file (provided that OBJECT is in effect). NODEBUG suppresses the output of information when OBJECT is in effect. DEBUG cannot be specified in the same assembly with NOOBJECT.

EJECT

Syntax

EJECT

Type

General

Abbreviation

EJ

Default

not applicable

Description

EJECT causes the assembler to start a new listing page. The page ejection is performed after the current line is listed.

ERRORPRINT/NOERRORPRINT

Syntax

ERRORPRINT[(*pathname*)]
NOERRORPRINT

Type

Listing Primary

Abbreviation

EP/NOEP

Default

NOERRORPRINT

Description

ERRORPRINT specifies that lines containing errors and the resultant error message be sent to either the console (if no *pathname* is specified) or to the specified *pathname*. The output file for the ERRORPRINT control cannot be the same as for the print file.

GEN/NOGEN

Syntax

GEN
NOGEN

Type

General

Abbreviation

GE/NOGE

Default

NOGEN

Description

Specifying GEN causes the original source lines to be interspersed with all of the following:

- macro expansion lines
- lines containing object code that cannot fit onto one line (such as a long DCB list)
- sequences of machine instructions resulting from translation of a generic instruction (e.g., DBNZ to a far target).

If NOGEN is specified, only the original source text will appear in the listing.

INCLUDE

Syntax

INCLUDE(*pathname*)

Type

General

Abbreviation

IC

Default

not applicable

Description

The INCLUDE control causes the assembler to pick up and include the specified file in its processing. After the invocation line or the current control line (whichever specified INCLUDE) is processed by the assembler, the current point in the input (source) file is stacked, and the file specified in the INCLUDE control is used as input for the assembler. When the include file has been completely processed, the previous input (source) file is retrieved and processing commences at the point where it was interrupted. The INCLUDE control must always be the rightmost control of the control line in which it appears. The nesting of included files cannot exceed a depth of nine.

LIST/NOLIST

Syntax

LIST
NOLIST

Type

General

Abbreviation

LI/NOLI

Default

LIST

Description

Specifying NOLIST causes the assembler to suppress the listing of any output until the next LIST control is encountered. Lines containing errors are still output, however. When a subsequent LIST control causes the resumption of listing after its suppression due to NOLIST, the listing settings originally in effect (such as PAGEWIDTH and PAGELENGTH) are obeyed.

OBJECT/NOOBJECT

Syntax

OBJECT [(*pathname*)]
NOOBJECT

Type

Object Primary

Abbreviation

OJ/NOOJ

Default

OBJECT(*sourcefile*.OBJ)

Description

The OBJECT control allows you to set the name of the object file produced by the assembler to the *pathname* specified in the control. If you do not enter the optional *pathname*, the default name *sourcefile*.OBJ is used to the object file. Specifying NOOBJECT suppresses the production of an object file. If you do not specify either OBJECT or NOOBJECT, the default is to OBJECT(*sourcefile*.OBJ).

PAGELength

Syntax

PAGELength(*n*)

Type

Listing Primary

Abbreviation

PL

Default

PAGELength(60)

Description

PAGELength sets the maximum number of lines per page of the listing to *n*. The header lines are counted toward the total. The minimum pagelength is 10 lines per page. Specifying a number less than 10 causes an error default to the minimum number of 10. If no pagelength is specified, the default is 60 lines per page.

PAGEWIDTH

Syntax

PAGEWIDTH(*n*)

Type

Listing Primary

Abbreviation

PW

Default

PAGEWIDTH(120)

Description

PAGEWIDTH sets the maximum number of characters per line of the listing to *n*. The minimum pagewidth is 72 characters per line. Specifying a number less than 72 causes an error default to the minimum number of 72. The maximum pagewidth is 132 characters per line. Specifying a number greater than 132 causes an error default to the maximum number of 132. Lines that exceed the right margin setting are continued by wrapping to column 39 of the line directly below. Source lines of more than 255 characters appear in the listing as truncated to 255 characters and are accompanied by an error message to that effect.

PRINT/NOPRINT

Syntax

PRINT [(*pathname*)]
NOPRINT

Type

Listing Primary

Abbreviation

PR/NOPR

Default

PRINT (*sourcefile*.LST)

Description

The PRINT control allows you to set the name of the listing file to the pathname specified in the control. If you do not enter the optional pathname the default pathname, *sourcefile*.LST, is used for the listing file. Specifying NOPRINT suppresses the production of a listing file. If you do not specify either PRINT or NOPRINT, the default is to PRINT(*sourcefile*.LST).

SAVE/RESTORE

Syntax

SAVE
RESTORE

Type

General

Abbreviation

SA/RS

Default

not applicable

Description

Specifying SAVE saves the current settings of the LIST, COND, and GEN flags on a stack of depth 9. You can then change the settings of these controls. Specifying RESTORE causes the original settings to be made operative again. If you specify RESTORE before any settings have been saved, or if you specify more RESTORES than SAVES, an error message to this effect is issued. Unlike other general controls, SAVE and RESTORE take effect at their position in the control line. Thus,

```
$LIST  
$SAVE NOLIST      ; This line is listed  
$RESTORE          ; This line is not listed  
$NOLIST SAVE      ; This line is listed  
$RESTORE LIST     ; This line is not listed  
                  ; This line is listed
```


SYMBOLS/NOSYMBOLS

Syntax

SYMBOLS
NOSYMBOLS

Type

Listing Primary

Abbreviation

SB/NOSB

Default

SYMBOLS

Description

Specifying SYMBOLS causes symbol table information to be placed on the listing file. Therefore, you cannot specify both SYMBOLS and NOPRINT for the same assembly. NOSYMBOLS suppresses the listing of symbol table information on the list file.

TITLE

Syntax

TITLE(*"string"*)

Type

General

Abbreviation

TT

Default

TITLE(*module name*)

Description

The TITLE control allows you to assign the string enclosed in quotation marks as the title of each page of the listing. If you do not explicitly specify a title by the TITLE control, the default condition is for the listing to take the module name for its title. When you do use the TITLE control, the string you wish to use as a title must be enclosed in either single (' ') or double (" ") quotation marks. You must use the same type of quotation mark on both sides of the string. Thus ('string') is not a valid entry. If you want literal quotation marks to appear in the title, you must use the other type of mark as string delimiters. Thus, if you want the title Rosie O'Grady, you must enclose it in double quotation marks so that the control line is:

```
$TITLE("Rosie O'Grady")
```

XREF/NOXREF

Syntax

XREF
NOXREF

Type

Listing Primary

Abbreviation

XR/NOXR

Default

NOXREF

Description

Specifying XREF causes a cross-reference listing to be included in the listing file. You cannot specify XREF in the same assembly with either NOSYMBOLS or NOPRINT. Specifying NOXREF suppresses the inclusion of a cross-reference table in the listing file.

7

Assembler Output: Error Messages and Listing File Format



7.1 Console Output

During assembly ASM96 produces the following output to the console: the sign-on message, fatal error messages, and the sign-off message.

7.1.1 Sign-on Message

The format for the sign-on message is:

```
system_id MCS-96 MACRO ASSEMBLER, Vx.y
```

Where:

system_id is the identification of the system under which the assembler is running.

x.y designates the version of the assembler.

7.2 Error Messages and Recovery

All error messages issued by ASM96 are either displayed on the console or listed in the listing file. Invocation line errors, internal errors, and I/O errors are console errors. Console errors are fatal errors that cause the assembly to abort and a message to this effect to be produced at the console. Errors detected in the source code (source file error messages) are not fatal and do not cause the assembler to terminate. An informative message is produced in the listing file or the errorprint file, and the assembly continues.

7.2.1 Console Error Messages

Upon detecting certain catastrophic conditions within the system hardware or in the invocation line, ASM96 prints an informative message at the console, aborts assembler processing, and returns control to the invoking system.

Console error messages can be divided into four categories:

1. invocation line error messages
2. I/O error messages
3. insufficient (host) memory error messages
4. internal synchronization error messages

7.2.1.1 Invocation Line Error Messages

Invocation line errors are in the form:

```
#INVOCATION LINE ERROR  
      message  
ASSEMBLY TERMINATED
```

Where:

message is an English language message describing the cause of the error.

7.2.1.2 I/O Error Messages

I/O error messages are in the form:

```
#I/O ERROR
    FILE: file type
    NAME: file name
    EXCEPTION nnnnH: message
ASSEMBLY TERMINATED
```

Where:

<i>file type</i>	is the type of the file (SOURCE, INCLUDE, PRINT, OBJECT, or ERRORPRINT).
<i>file name</i>	is the name of the file.
<i>nnnnH</i>	is the hexadecimal code number.
<i>message</i>	is an English language message describing the cause of the error.

7.2.1.3 Insufficient (Host) Memory Error Messages

Insufficient memory messages are in the form:

```
#INSUFFICIENT MEMORY
ASSEMBLY TERMINATED
```

7.2.1.4 Internal Synchronization Error Messages

Internal synchronization errors are in the form:

```
#SYNCHRONIZATION ERROR
ASSEMBLY TERMINATED
```

7.2.2 Sign-off Message

The sign-off message displayed on the console (and in the listing file) when the assembly is terminated is:

```
ASSEMBLY COMPLETED, nnnn ERROR(S) FOUND
```

Where:

nnnn is the number of errors found (NO if no errors are found).

7.3 Listing File

The listing file provides you with important information on the assembly of your program. It contains both user-generated information and assembler-generated information. Errors detected by ASM96 within your source code are also reported in the listing file. Unless you use the controls described in Chapter 6 to specify differently, the assembler produces a listing file that has a pagewidth of 120 characters and a page length of 60 lines. Each page includes a page header, two empty lines followed by the column heading, another empty line, and then the assembly output. Lines that exceed the right margin setting (either the default margin or the margin you have set) are continued on the line directly below the original line commencing at column 35. Refer to Figure 7-1 for an example of the format of the listing file.

system-id (046-N) MCS-96 MACRO ASSEMBLER, Vx.y

SOURCE FILE: sample.a96

OBJECT FILE: sample.obj

CONTROLS SPECIFIED IN INVOCATION COMMAND: xref

LOC	OBJECT	STMT	SOURCE STATEMENT
		1	\$TITLE('--> HAND CALCULATOR: MAIN <--')
		2	
		3	HC_MAIN MODULE MAIN,STACKSIZE(8)
		4	;
		5	;The program simulates a common hand calculator.
		6	;Available commands (DR the display register, MR the memory register):
		7	;
		8	; [operand1] op operand2 - where 'operand is a decimal number.
		9	; 'op' can be +, -, * and /.
		10	; M+ - add DR to MR.
		11	; RM - transfer MR to DR.
		12	; M- - subtract DR from MR (result in MR).
		13	; CM - clear MR.
		14	; ? or H - a help info is displayed.
		15	; EX - exit.
		16	;Commands can be written in upper or lower case letters.
		17	
		18	EXTRN ER_Invalid_Command
		19	PUBLIC MN_Result, MN_Exit_Flag
		20	;
		21	;Runtime conventions:
		22	; 1) Paramters are transferred on the stack.
		23	; 2) Word and byte funcs return their result in the global reg
		24	; MN_Result.

Figure 7-1 Source Listing Example

LOC	OBJECT	STMT	SOURCE STATEMENT
		25	\$INCLUDE(8096.INC)
		-1 26	\$nolist
		86	::::::::::::::::::::::::::
		87	;
		88	;Common definitions
0000		89	CR EQU 0DH
000A		90	LF EQU 0AH
0020		91	BLANK EQU 20H
0009		92	TAB EQU 09H
		93	STRING MACRO Str ;Defines a length-prefixed string
		94	LOCAL Len
		95	Start_LC SET \$
		96	DCB Len, Str
		97	Len EQU \$-Start_LC-1
		98	ENDM
		99	
0000		100	RSEG
0000		101	MN_Result: DSW 1 ;Result of byte or word functions
0002		102	MN_Exit_Flag: DSB 1 ;Exit flag, only bit 0 is used
		103	
2080		104	CSEG at 2080H
		105	EXTRN IO_Put_String, IO_Put_Char
		106	EXTRN RD_Get_Line, RD_Get_Chagr, RD_This_Char
		107	EXTRN RD_Skip_Blanks, RD_Check_EPL
		108	EXTRN UT_FindB, UT_Help_Com
		109	EXTRN ER_Put_Message
		110	EXTRN RG_Eval, RG_Mem_Op, RG_Clear_MR, RG_Recall_MR,
***			^
***			ERROR #10 IN STMT 110 (LINE 52), SYNTAX ERROR
		111	EXTRN RG_Print_DR
2080		112	Start: ;... of main
program			
2080 A1000018	R	113	LD SP, #STACK ;Init StackPointer
2084 C9DC20		114	PUSH #STR_signon ;Print Signon
2087 EF0000	E	115	CALL IO_Put_String
208A 1102	R	116	CLRB MN_Exit_Flag ;Reset exit flag
208C		117	Main_Loop:
208C 300203E78E00	R	118	! BBS MN_Exit_Flag, 0, Exit ;Check for exit
2092 EF0000	E	119	CALL RG_Print_DR

Figure 7-1 Source Listing Example (continued)

LOC	OBJECT	STMT	SOURCE STATEMENT
2095	C93A00	120	PUSH #' : ' ;Print prompt character
2098	EF0000	E 121	CALL IO_Put_Char
209B	EF0000	E 122	CALL RD_Get_Line ;Line is placed in RD_Line
209E	EF0000	E 123	CALL RD_Skip_Blanks
		124	
20A1	EF0000	E 125	CALL RD_This_Char ;Find command type
20A4	C9C720	126	PUSH #STR_1st_Chars
20A7	C800	R 127	PUSH MN_Result
20A9	EF0000	E 128	CALL UT_FindB
20AC	89FFFF00	R 129	CMP MN_Result, #0FFFFH
20B0	D708	130	BNE Command_OK
20B2	C90000	E 131	PUSH #ER_Invalid_Command
20B5	EF0000	E 132	CALL ER_Put_Message
20B8	27D2	133	BR Main_Loop
20BA		134	Command_OK:
20BA	640000	R 135	ADD MN_Result, MN_Result ;Do case MN_Result
20BD	A300FC2000	R 136	LD MN_Result, Com_Tab[MN_Result]
20C2	C98C20	137	PUSH #Main_Loop ;Simulate indirect call
20C5	E300	R 138	BR [MN_Result]
		139	;
		140	;Tables and constants used for the above logic
20C7		141	STR_1st_Chars: STRING <'0123456789','+-*/','MRC?HE'>
20DC		145	STR_Signon: STRING <CR,LF,'8096-BASED CALCULATOR, V1.0',CR,LF>
20FC		149	Com_Tab:
20FC	0000000000000000	E 150	DCW RG_Eval, RG_Eval, RG_Eval, RG_Eval; 0-4
2104	0000000000000000	E 151	DCW RG_Eval, RG_Eval, RG_Eval, RG_Eval; 5-9
210C	0000000000000000	E 152	DCW RG_Eval, RG_Eval, RG_Eval, RG_Eval; +-*/
2114	0000	E 153	DCW RG_Mem_Op ;M+ or M-
2116	0000	E 154	DCW RG_Recall_MR ;RM
2118	0000	E 155	DCW RG_Clear_MR ;CM
211A	00000000	E 156	DCW UT_Help_com, UT_Help_com ;? or H
211E	0000	157	DCW UT_Exit_com ;EX
*** ERROR #38 IN STMT 157 (LINE 93), UNDEFINED SYMBOL			
		158	;
		159	; Exit loop
2120		160	exit:

Figure 7-1 Source Listing Example (continued)

LOC	OBJECT	STMT	SOURCE STATEMENT
2120	C92821	161	PUSH #STR_Signoff ;Print signoff
2123	EF0000	162	Call IO_Put_String
2126	27FE	163	BR \$;Infinite loop
		164	\$GEN
2128		165	STR_Signoff: STRING <CR,LF,' S T O P',CR,LF>
2128		166+1	Start_LC SET \$
2128	0F0D0A2020202053	167+1	DCB ??0003, CR,LF,' S T O P',CR,LF
2130	2054204F20500D0A		
000F		168+1	??0003 EQU \$-Start_LC-1
2138		169	Start_LC SET \$
2138	0F0D0A2020205320	170	DCB ??0003,CR,LF,' S T O P',CR,LF
2140	54204F20500D0A		
		171	
2147		172	END

Figure 7-1 Source Listing Example (continued)

7.3.1 Header

Header information appears at the top of each page of the listing file and contains the title, the date, and the page number as follows:

MCS-96 MACRO ASSEMBLER *title* *date* PAGE *n*

Where:

title is either the title you have specified by the TITLE control or the default title of the module name. The title always starts four character positions to the right of the word ASSEMBLER.

date is the system date. The date always begins 27 character positions to the left of the right margin (either the default margin or one you have specified by the PAGEWIDTH control).

n is the page number. The word PAGE preceding *n* always begins 9 character positions to the left of the right margin (default or specified by PAGEWIDTH).

7.3.2 Additional Introductory Lines

The first page of the listing file also contains the following introductory lines:

```
system id MCS-96 MACRO ASSEMBLER Vx.y
SOURCE FILE: source_pathname
OBJECT FILE: object_pathname
CONTROLS SPECIFIED IN INVOCATION COMMAND: control_list
```

Where:

<i>system id</i>	is the name of your development system.
<i>x.y</i>	specifies the version number of the assembler.
<i>source_pathname</i>	is the pathname of your source file.
<i>object_pathname</i>	is the pathname of your object file, or if you specified NOOBJECT, the entry <none>.
<i>control_list</i>	is a list of all the controls you specified in the invocation line.

7.3.3 Body

The body of the listing file consists of columns of information (fields). The fields are:

- LOC (location counter field, columns 1-4) -- This field contains the hexadecimal value of the location counter, i.e., where the associated byte of code or of data will be placed. The value displayed is adjusted for any required alignment (e.g., in the case of DCW or DSL directives) or (in the case of CSEG, DSEG, RSEG, OSEG, or ORG directives) reflects the change of the location counter.

For absolute segments, the LOC field contains an absolute address. For relocatable segments, the LOC field contains the offset from the beginning of the segment currently being assembled.

The LOC field is displayed for all machine instructions, the code definition directives (DCB, DCW, DCL, and DCR), the storage reservation directives (DSB, DSW, DSL, and DSR), the segment selection directives (CSEG, DSEG, RSEG, and OSEG), the ORG directive, and for empty statements and macro calls that are preceded by labels. Otherwise, the field is blank.

- SET or EQU (columns 3-6) value -- If the statement is a SET or an EQU directive, columns 7-10 display the value of the expression in hexadecimal. Since this value is not part of either the location counter or the object code fields, the value is not aligned with either of them.
- OBJ (object code, columns 6-21) -- This field contains the object code in hexadecimal generated by the assembler for the given line. If NOGEN is in effect, only the first eight bytes of object code will be displayed. Under GEN, the entire object code will be displayed in the following lines, with eight bytes per line (except possibly for the last line).
- fixup indicator (column 23) -- When no fixup is required, this field is blank. If the code contains a relocatable reference, this field contains R. If the code contains an external reference, this field contains E. If the code contains both an external and a relocatable reference, only E is displayed.
- INCLUDE indicator (columns 25-26) -- All source lines that are from an INCLUDE file contains an equal sign (=) in column 29. Column 30 contains the INCLUDE nesting indicator. Thus, a second INCLUDE file nested within an original INCLUDE file contains a 2 in column 30, while the lines from the original INCLUDE file contains 1 in column 30.
- LINE (line number, columns 23-31) -- This field contains the line number. The first line is given the number 1. The line number is then sequentially incremented by 1 for every source line and macro expansion line but not incremented for conditionally skipped lines (i.e., lines not assembled because of an IF statement). The line number does increment, however, if a given source line or macro expansion is assembled but not listed.
- macro expansion indicator (columns 32-33) -- All source lines that are macro expansion lines are marked as such by a plus sign (+) appearing in line 36. The level of macro nesting is recorded in column 37. Thus, 1 indicates the original macro, 2 indicates a macro nested within the original macro.

- expanded generic instruction indicator (column 34) -- If the source line contains a generic instruction that is expanded into several machine instructions or any of the machine instructions resulting from that expansion, column 38 is marked with an exclamation mark (!).
- SOURCE (columns 35-end of line) -- The source line is reproduced exactly as you entered it (except that tabs are expanded to blanks) beginning in column 39. If a generic instruction results in more than one machine instruction and the GEN control is in effect, the instructions resulting from the expansion are displayed, starting in column 39, one instruction per line.

7.3.3.1 Error Lines

When an error occurs, an error line appears in the listing immediately after the source statement which caused the error. The error line format is as follows:

```
*** ERROR #d IN STMT #n (LINE s [OF FILE f ], <error-text>)
```

Where:

- d* is the corresponding error number.
- n* is the assembly statement number from column 28-31 of the listing file.
- s* is the source line number from the user's source file. If the source line is from the INCLUDE file the source line number is the line number within file *f*.

If the error is a "SYNTAX ERROR," a line preceding the error message line will indicate which token caused the error. The line consists of three asterisks (***) followed by underscores (_) extending to the point in error. The caret symbol (^) will "point" to the token in error.

7.3.4 The Symbol Table

The symbol table is a list of all symbols defined in the program along with status information about the symbols. The symbol table is produced only if the SYMBOLS control has been specified. If XREF has also been specified, cross-reference information will be included in the symbol table.

The format of the symbol table is shown in Figure 7-2. A page eject always occurs between the end of the source listing and the start of the symbol table.

MCS-96 MACRO ASSEMBLER		--> HAND CALCULATOR: MAIN <--		01/01/83	
SYMBOL TABLE AND CROSS-REFERENCE LISTING					

N A M E		VALUE	ATTRIBUTES AND REFERENCES		
??0001.		0014H	NULL ABS		
			82	83#	
??0002.		001FH	NULL ABS		
			86	87#	
??0003.		000FH	NULL ABS		
			107	108#	
BLANK		0020H	NULL ABS		
			31#		
COM_TAB		20FCH	CODE ABS WORD		
			75	88#	
COMMAND_OK		20BAH	CODE ABS ENTRY		
			69	73#	
CR.		000DH	NULL ABS		
			29#	86	86 107 107
ER_INVALID_COMMAND.		-----	NULL EXTERNAL		
			18#	34	70
ER_PUT_MESSAGE.		-----	CODE EXTERNAL		
			19#	71	
EXIT.		2126H	CODE ABS ENTRY		
			57	100#	
HC_MAIN		-----	MODULE MAIN STACKSIZE(8)		
			3#		
IO_PUT_CHAR		-----	CODE EXTERNAL		
			46#	60	
IO_PUT_STRING		-----	CODE EXTERNAL		
			46#	54	102
LF.		000AH	NULL ABS		
			30#	86	86 107 107

Figure 7-2 Symbol Table Example

[illegible]

Figure 7-2 Symbol Table Example (continued)

STRING.		-----	MACRO				
				34#	80	84	105
TAB		0009H	NULL ABS				
				32#			
UT_EXIT_COM		-----	UNDEFINED				
				96			
UT_FINDB		-----	CODE EXTERNAL				
				48#	67		
UT_HELP_COM		-----	CODE EXTERNAL				
				48#	95	95	

Figure 7-2 Symbol Table Example (continued)

Each page of the symbol table has the same header lines as do the pages of the source listing. The page numbers continue from the source listing into the symbol table. In addition, the first page of the symbol table contains the header:

SYMBOL TABLE LISTING

If the XREF control is specified, the header reads:

SYMBOL TABLE AND CROSS-REFERENCE LISTING

The symbol table, like the source listing, is divided into fields. The fields are NAME, VALUE, ATTRIBUTES, and REFERENCES (if XREF has been specified along with SYMBOLS). The symbols are then listed in alphabetic order, except for the underscore (_) coming first. Fields and symbols are listed as follows:

- NAME (columns 1-31) -- The name of the symbol is listed. Alternating periods and blanks (. . .) are used to fill the field to column 31.
- VALUE (columns 35-39) -- The value of the symbol (i.e., its offset from the segment in the case of relocatable symbols or the absolute address/value in the case of absolute symbols) is given in hexadecimal form. For undefined symbols, external symbols, macros, and the module symbol, the value field contains four hyphens (- - - -).
- ATTRIBUTES (column 43-end of line) -- The attributes associated with each name are listed. The attributes are separated by spaces. The attributes are:

REG, OVERLAY, CODE, DATA, or STACK is the segment type of the symbol.

NULL appears for symbols that do not belong to any segment.

BYTE, WORD, LONG, REAL, or ENTRY is the data type of the symbol.

ABS appears for absolute symbols.

REL appears for relocatable symbols.

PUBLIC appears for public symbols.

EXTERNAL appears for external symbols.

MACRO appears for macros.

MODULE appears only for the module name symbol. There is only one module per assembly unit.

MAIN indicates that the module is the main module of the application.

STACKSIZE gives the size of the stack required by the module.

UNDEFINED appears for symbols that were not defined in the program.

- CROSS-REFERENCES (columns 43-end of line) -- If the XREF control is also specified, the symbol table listing also contains a list of the line numbers of every statement that use the given symbol. If the value of the symbol is defined (or redefined) in the given statement, the line number of the statement is followed by the number sign (#). Statement line numbers for a given symbol are separated from one another by spaces. The first statement line number begins in column 43, the second six character positions to the right in column 49, the third in column 55, and so on.

7.4 The Errorprint File

The errorprint file consists of all listing lines that contain errors. The listing of erroneous lines is identical to the listing of these lines in the print file. Each error line consists of three asterisks (***) followed by the error number, the statement line number where the error occurred, and the error message. Refer to Appendix E of this manual for a list of all the errors.

2.1.2. When a user is logged in, the system will check the user's profile for any pending tasks.

2.1.3. The system will display a list of pending tasks for the user.

2.1.4. The system will allow the user to mark a task as completed.

2.1.5. The system will update the user's profile with the status of the task.

2.1.6. The system will send a notification to the user when a task is completed.

2.1.7. The system will allow the user to delete a task from the list.

2.1.8. The system will allow the user to filter the list of pending tasks by status.

2.1.9. The system will allow the user to sort the list of pending tasks by priority.

2.1.10. The system will allow the user to search for a specific task in the list.

2.1.11. The system will allow the user to add a new task to the list.

2.1.12. The system will allow the user to edit an existing task in the list.

2.1.13. The system will allow the user to delete a task from the list.

2.1.14. The system will allow the user to mark a task as completed.

2.1.15. The system will allow the user to filter the list of pending tasks by status.

2.1.16. The system will allow the user to sort the list of pending tasks by priority.

2.1.17. The system will allow the user to search for a specific task in the list.

2.1.18. The system will allow the user to add a new task to the list.

2.1.19. The system will allow the user to edit an existing task in the list.

2.1.20. The system will allow the user to delete a task from the list.

2.4 The Feedback File

The feedback file is a text file that contains the user's feedback on the system. The feedback is entered by the user in the feedback form, which is located on the user's profile page. The feedback is then saved to the feedback file. The feedback file is used by the system to generate reports on user feedback. The feedback file is also used by the system to send notifications to the user when their feedback has been received. The feedback file is a text file that contains the user's feedback on the system. The feedback is entered by the user in the feedback form, which is located on the user's profile page. The feedback is then saved to the feedback file. The feedback file is used by the system to generate reports on user feedback. The feedback file is also used by the system to send notifications to the user when their feedback has been received.

A

Machine Instruction Summary

intel®

Table A-1 Instructions in Alphabetical Order

Opcode	Number of Bytes	Mnemonic	Operands
64-67	2+BEA	ADD	wreg,waop
44-47	3+BEA	ADD	wreg,wreg,waop
74-77	2+BEA	ADDB	breg,baop
54-57	3+BEA	ADDB	breg,breg,baop
A4-A7	2+BEA	ADDC	wreg,waop
B4-B7	2+BEA	ADDCB	breg,baop
60-63	2+BEA	AND	wreg,waop
40-43	3+BEA	AND	wreg,wreg,waop
70-73	2+BEA	ANDB	breg,baop
50-53	3+BEA	ANDB	breg,breg,baop
C1	3	BMOV*	lreg,wreg
CD	3	BMOVI**	lreg,wreg
E3	2	BR	[wreg]
01	2	CLR	wreg
11	2	CLRB	breg
F8	1	CLRC	
FC	1	CLRVT	
88-8B	2+BEA	CMP	wreg,waop
98-9B	2+BEA	CMPB	breg,baop
C5	3	CMPL*	lreg,lreg
05	2	DEC	wreg
15	2	DECB	breg

* This instruction is specific to the 80C196 processors

** This instruction is specific to the 80C196KC processor

Table A-1 Instructions in Alphabetical Order (continued)

Opcode	Number of Bytes	Mnemonic	Operands
FA	1	DI	
FE 8C-8F	2+BEA	DIV	lreg,waop
FE 9C-9F	2+BEA	DIVB	wreg,baop
8C-8F	2+BEA	DIVU	lreg,waop
9C-9F	2+BEA	DIVUB	wreg,baop
E0	3	DJNZ	breg,cadd
E1	3	DJNZW*	wreg,cadd
EC	1	DPTS**	
FB	1	EI	
ED	1	EPTS**	
06	2	EXT	lreg
F6	2	IDLPD*	baop
16	2	EXTB	wreg
07	2	INC	wreg
17	2	INCB	breg
30-37	3	JBC	breg,bitno,cadd
38-3F	3	JBS	breg,bitno,cadd
DB	2	JC	cadd
DF	2	JE	cadd
D6	2	JGE	cadd
D2	2	JGT	cadd
D9	2	JH	cadd
DA	2	JLE	cadd
DE	2	JLT	cadd
D3	2	JNC	cadd
D7	2	JNE	cadd
D1	2	JNH	cadd
D0	2	JNST	cadd
D5	2	JNV	cadd

* This instruction is specific to the 80C196 processors

** This instruction is specific to the 80C196KC processor

Table A-1 Instructions in Alphabetical Order (continued)

Opcode	Number of Bytes	Mnemonic	Operands
D4	2	JNVT	cadd
D8	2	JST	cadd
DD	2	JV	cadd
DC	2	JVT	cadd
EF	3	LCALL	cadd
A0-A3	2+BEA	LD	wreg,waop
B0-B3	2+BEA	LDB	breg,baop
BC-BF	2+BEA	LDBSE	wreg,baop[xn]
AC-AF	2+BEA	LDBZE	wreg,baop
E7	3	LJMP	cadd
FE 6C-6F	3+BEA	MUL	lreg,waop
FE 4C-4F	4+BEA	MUL	lreg,wreg,waop
FE 7C-7F	3+BEA	MULB	wreg,baop
FE 5C-5F	4+BEA	MULB	wreg,breg,baop
6C-6F	2+BEA	MULU	lreg,waop
4C-4F	3+BEA	MULU	lreg,wreg,waop
7C-7F	2+BEA	MULUB	wreg,baop
5C-5F	3+BEA	MULUB	wreg,breg,baop
03	2	NEG	wreg
13	2	NEGB	breg
FD	1	NOP	
0F	3	NORML	lreg,breg
02	2	NOT	wreg
12	2	NOTB	breg
80-83	2+BEA	OR	wreg,waop
90-93	2+BEA	ORB	breg,baop
CC-CF	1+BEA	POP	waop
F5	1	POPA*	
F3	1	POPF	
C8-CB	1+BEA	PUSH	waop

* This instruction is specific to the 80C196 processors

Table A-1 Instructions in Alphabetical Order (continued)

Opcode	Number of Bytes	Mnemonic	Operands
F4	1	PUSHA*	
F2	1	PUSHF	
F0	1	RET	
28-2F	2	SCALL	cadd
F9	1	SETC	
09	3	SHL	wreg,#count or breg
19	3	SHLB	breg,#count or breg
0D	3	SHLL	lreg,#count or breg
08	3	SHR	wreg,#count or breg
0A	3	SHRA	wreg,#count or breg
1A	3	SHRAB	breg,#count or breg
0E	3	SHRAL	lreg,#count or breg
18	3	SHRB	breg,#count or breg
0C	3	SHRL	lreg,#count or breg
20-27	2	SJMP	cadd
00	2	SKIP	breg
C0-C3	2+BEA	ST	wreg,waop
C4-C7	2+BEA	STB	breg,baop
68-6B	2+BEA	SUB	wreg,waop[xn]
48-4B	3+BEA	SUB	wreg,wreg,waop
78-7B	2+BEA	SUBB	breg,baop
58-5B	3+BEA	SUBB	breg,breg,baop
A8-AB	2+BEA	SUBC	wreg,waop
B8-BB	2+BEA	SUBCB	breg,baop
E2	4	TIJMP**	wreg,(wreg),#imm
14/1B	3	XCH**	wreg,wreg/offset
04/0B	3	XCHB**	breg,breg/offset
84-87	2+BEA	XOR	wreg,waop
94-97	2+BEA	XORB	breg,baop

* This instruction is specific to the 80C196 processors

** This instruction is specific to the 80C196KC processor

Table A-2 Instructions in Hexadecimal Order

Opcode	Number of Bytes	Mnemonic	Operands
00	2	SKIP	breg
01	2	CLR	wreg
02	2	NOT	wreg
03	2	NEG	wreg
04	3	XCHB*	breg,breg
05	2	DEC	wreg
06	2	EXT	lreg
07	2	INC	wreg
08	3	SHR	wreg,#count or breg
09	3	SHL	wreg,#count or breg
0A	3	SHRA	wreg,#count or breg
0B	5	XCHB**	breg,offset
0C	3	SHRL	lreg,#count or breg
0D	3	SHLL	lreg,#count or breg
0E	3	SHRAL	lreg,#count or breg
0F	3	NORML	lreg,breg
11	2	CLRB	breg
12	2	NOTB	breg
13	2	NEGB	breg
14	5	XCH**	wreg,wreg
15	2	DECB	breg
16	2	EXTB	wreg
17	2	INCB	breg
18	3	SHRB	breg,#count or breg
19	3	SHLB	breg,#count or breg
1A	3	SHRAB	breg,#count or breg
1B	3	XCH**	wreg,offset
20-27	2	SJMP	cadd
28-2F	2	SCALL	cadd

* This instruction is specific to the 80C196 processors

** This instruction is specific to the 80C196KC processor

Table A-2 Instructions in Hexadecimal Order (continued)

Opcode	Number of Bytes	Mnemonic	Operands
30-37	3	JBC	breg,bitno,cadd
38-3F	3	JBS	breg,bitno,cadd
40-43	3+BEA	AND	wreg,wreg,waop
44-47	3+BEA	ADD	wreg,wreg,waop
48-4B	3+BEA	SUB	wreg,wreg,waop
4C-4F	3+BEA	MULU	lreg,wreg,waop
50-53	3+BEA	ANDB	breg,breg,baop
54-57	3+BEA	ADDB	breg,breg,baop
58-5B	3+BEA	SUBB	breg,breg,baop
5C-5F	3+BEA	MULUB	wreg,breg,baop
60-63	2+BEA	AND	wreg,waop
64-67	2+BEA	ADD	wreg,waop
68-6B	2+BEA	SUB	wreg,waop
6C-6F	2+BEA	MULU	lreg,waop
70-73	2+BEA	ANDB	breg,baop
74-77	2+BEA	ADDB	breg,baop
78-7B	2+BEA	SUBB	breg,baop
7C-7F	2+BEA	MULUB	wreg,baop
80-83	2+BEA	OR	wreg,waop
84-87	2+BEA	XOR	wreg,waop
88-8B	2+BEA	CMP	wreg,waop
8C-8F	2+BEA	DIVU	lreg,waop
90-93	2+BEA	DRB	breg,baop
94-97	2+BEA	XORB	breg,baop
98-9B	2+BEA	CMPB	breg,baop
9C-9F	2+BEA	DIVUB	wreg,baop
A0-A3	2+BEA	LD	wreg,waop
A4-A7	2+BEA	ADDC	wreg,waop
A8-AB	2+BEA	SUBC	wreg,waop
AC-AF	2+BEA	LDBZE	wreg,baop

Table A-2 Instructions in Hexadecimal Order (continued)

Opcode	Number of Bytes	Mnemonic	Operands
B0-B3	2+BEA	LDB	breg,baop
B4-B7	2+BEA	ADDCB	breg,baop
B8-BB	2+BEA	SUBCB	breg,baop
BC-BF	2+BEA	LDBSE	wreg,baop
C1	3	BMOV*	lreg,wreg
C0,C2,C3	2+BEA	ST	wreg,waop
C5	3	CMPL*	lreg,lreg
C4,C6,C7	2+BEA	STB	breg,baop
C8-CB	1+BEA	PUSH	waop
CD	3	BMOVI**	lreg,wreg
CC,CE,CF	1+BEA	POP	waop
D0	2	JNST	cadd
D1	2	JNH	cadd
D2	2	JGT	cadd
D3	2	JNC	cadd
D4	2	JNVT	cadd
D5	2	JNV	cadd
D6	2	JGE	cadd
D7	2	JNE	cadd
D8	2	JST	cadd
D9	2	JH	cadd
DA	2	JLE	cadd
DB	2	JC	cadd
DC	2	JVT	cadd
DD	2	JV	cadd
DE	2	JLT	cadd
DF	2	JE	cadd
E0	3	DJNZ	breg,cadd
E1	3	DJNZW*	wreg,cadd

* This instruction is specific to the 80C196 processors

** This instruction is specific to the 80C196KC processor

Table A-2 Instructions in Hexadecimal Order (continued)

Opcode	Number of Bytes	Mnemonic	Operands
E2	4	TIJMP**	wreg,(wreg),#imm
E3	2	BR	[wreg]
E7	3	LJMP	cadd
EC	1	DPTS**	
ED	1	EPTS**	
EF	3	LCALL	cadd
F0	1	RET	
F2	1	PUSHF	
F3	1	POPF	
F4	1	PUSHA*	
F5	1	POPA*	
F6	2	IDLPD*	baop
F8	1	CLRC	
F9	1	SETC	
FA	1	DI	
FB	1	EI	
FC	1	CLRVT	
FD	1	NOP	
FE 4C-4F	4+BEA	MUL	lreg,wreg,waop
FE 5C-5F	4+BEA	MULB	wreg,breg,baop
FE 6C-6F	3+BEA	MUL	lreg,waop
FE 7C-7F	3+BEA	MULB	wreg,baop
FE 8C-8F	2+BEA	DIV	lreg,waop
FE 9C-9F	2+BEA	DIVB	wreg,baop

* This instruction is specific to the 80C196 processors

** This instruction is specific to the 80C196KC processor

The following notes apply to Table A-3 and Table A-4:

1. Indirect and Indexed cycle times are shown as i/e, indicating cycle times for instructions with internal/external operands specified
2. The assembler does not accept the mnemonic TRAP.
3. Cycle times listed are the minimum number of state times required for execution, assuming a 16-bit bus and no wait states. A state time is 2 oscillator periods. The cycle time can increase if wait states are used or if the opcode and its operands are not prefetched and residing in the instruction queue when they are needed.

The instruction queue is almost never empty when the processor is running in the 16-bit bus mode without wait states, so the minimum number of state times is almost always the correct execution time. When adding wait states, multiply the number of wait states by the number of instruction fetches and data accesses and add the result to the cycle time listed for the instruction. This gives the worst-case timing for an instruction sequence.

Instruction execution in the 8-bit mode typically takes 20 to 30 percent longer than in the 16-bit mode.

4. Cycle times for jumps, calls, and returns include the time required to flush the instruction queue and fetch the opcode at the destination address.

Table A-3 80C196 Instruction Cycle Times

Mnemonic (# of ops)	Direct	Immed	Indirect		Indexed	
			Normal	Auto-Inc	Short	Long
ADD (2)	4	5	6/8	7/9	6/8	7/9
ADD (3)	5	6	7/9	8/10	7/9	8/10
ADDB (2)	4	4	6/8	7/9	6/8	7/9
ADDB (3)	5	5	7/9	8/10	7/9	8/10
ADDC	4	5	6/8	7/9	6/8	7/9
ADDCB	4	4	6/8	7/9	6/8	7/9
AND (2)	4	5	6/8	7/9	6/8	7/9
AND (3)	5	6	7/9	8/10	7/9	8/10
ANDB (2)	4	4	6/8	7/9	6/8	7/9
ANDB (3)	5	5	7/9	8/10	7/9	8/10
BMOV	6 + 8 per word			6 + 11/14 per word		
BMOVI	7 + 8 per word + 14 per interrupt			7 + 11/14 per word + 14 per interrupt		
BR [indirect]	7					
CLR	3					
CLRB	3					
CLRC	2					
CLRVT	2					
CMP	4	5	6/8	7/9	6/8	7/9
CMPB	4	4	6/8	7/9	6/8	7/9
CMPL	7					
DEC	3					
DECB	3					
DI	2					
DIV	26	27	28/31	29/32	29/32	30/33
DIVB	18	18	20/23	21/24	21/24	22/25
DIVU	24	25	26/29	27/30	27/30	28/31
DIVUB	16	16	18/21	19/22	19/22	20/23
DJNZ	5 [jump not taken], 9 [jump taken]					
DJNZW	5 [jump not taken], 9 [jump taken]; one state time longer on the 80C196KC					

Table A-3 80C196 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect Normal	Auto-Inc	Indexed Short	Long
DPTS	2					
EPTS	2					
EI	2					
EXT	4					
EXTB	4					
IDLDP	8 [proper key], 25 [improper key]					
INC	3					
INCB	3					
JBC	5 [jump not taken], 9 [jump taken]					
JBS	5 [jump not taken], 9 [jump taken]					
JC	4 [jump not taken], 8 [jump taken]					
JE	4 [jump not taken], 8 [jump taken]					
JGE	4 [jump not taken], 8 [jump taken]					
JGT	4 [jump not taken], 8 [jump taken]					
JH	4 [jump not taken], 8 [jump taken]					
JLE	4 [jump not taken], 8 [jump taken]					
JLT	4 [jump not taken], 8 [jump taken]					
JNE	4 [jump not taken], 8 [jump taken]					
JNH	4 [jump not taken], 8 [jump taken]					
JNST	4 [jump not taken], 8 [jump taken]					
JNC	4 [jump not taken], 8 [jump taken]					
JNV	4 [jump not taken], 8 [jump taken]					
JNVT	4 [jump not taken], 8 [jump taken]					
JST	4 [jump not taken], 8 [jump taken]					
JV	4 [jump not taken], 8 [jump taken]					
JVT	4 [jump not taken], 8 [jump taken]					
LCALL						
internal stack	11					
external stack	13					
LD	4	5	5/7	6/8	6/8	7/9

Table A-3 80C196 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect		Indexed	
			Normal	Auto-Inc	Short	Long
LDB	4	5	5/7	6/8	6/8	7/9
LDBSE	4	4	5/7	6/8	6/8	7/9
LDBZE	4	4	5/7	6/8	6/8	7/9
LJMP	7					
MUL (2)	16	17	18/21	19/22	19/22	20/23
MUL (3)	16	17	18/21	19/22	19/22	20/33
MULB (2)	12	12	14/17	15/18	15/18	16/19
MULB (3)	12	12	14/17	15/18	15/18	16/19
MULU (2)	14	15	16/19	17/20	17/20	18/21
MULU (3)	14	15	16/19	17/20	17/20	18/21
MULUB (2)	10	10	12/15	12/16	12/16	14/17
MULUB (3)	10	10	12/15	12/16	12/16	14/17
NEG	3					
NEGB	3					
NOP	2					
NORML	8 + 1 per shift [9 for 0 shift]					
NOT	3					
NOTB	3					
OR (20)	4	5	6/8	7/9	6/8	7/9
ORB (2)	4	4	6/8	7/9	6/8	7/9
POP						
internal stack	8	--	10/12	11/13	11/13	12/14
external stack	11	--	13/15	14/16	14/16	15/17
POPA						
internal stack	12					
external stack	18					
POPF						
internal stack	7					
external stack	10					

Table A-3 80C196 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect Normal	Auto-Inc	Indexed Short	Long
PUSH						
internal stack	6	7	9/12	10/13	10/13	11/14
external stack	8	9	11/14	12/15	12/15	13/16
PUSHA						
internal stack	12					
external stack	18					
PUSHF						
internal stack	6					
external stack	8					
RET						
internal stack	11					
external stack	14					
RST	15 [includes fetch of configuration byte]					
SCALL						
internal stack	11					
external stack	13					
SETC	2					
SHL	6 + 1 per shift [7 for 0 shift]					
SHLB	6 + 1 per shift [7 for 0 shift]					
SHLL	7 + 1 per shift [8 for 0 shift]					
SHR	6 + 1 per shift [7 for 0 shift]					
SHRA	6 + 1 per shift [7 for 0 shift]					
SHRAB	6 + 1 per shift [7 for 0 shift]					
SHRAL	7 + 1 per shift [8 for 0 shift]					
SHRB	6 + 1 per shift [7 for 0 shift]					
SHRL	7 + 1 per shift [8 for 0 shift]					
SJMP	7					
SKIP	3					
ST	4	5	5/7	6/8	6/8	7/9
STB	4	5	5/7	6/8	6/8	7/9

Table A-3 80C196 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect		Indexed	
			Normal	Auto-Inc	Short	Long
SUB (2)	4	5	6/8	7/9	6/8	7/9
SUB (3)	5	6	7/9	8/10	7/9	8/10
SUBB (2)	4	4	6/8	7/9	6/8	7/9
SUBB (3)	5	5	7/9	8/10	7/9	8/10
SUBC	4	5	6/8	7/9	6/8	7/9
SUBCB	4	4	6/8	7/9	6/8	7/9
TIJMP	15 + 3 for each memory controller reference					
TRAP						
internal stack	16					
external stack	18					
XCH	5					
XCHB	5					
XOR	4	5	6/8	7/9	6/8	7/9
XORB	4	4	6/8	7/9	6/8	7/9

Table A-4 8096 Instruction Cycle Times

Mnemonic (# of ops)	Direct	Immed	Indirect		Indexed	
			Normal	Auto-Inc	Short	Long
ADD (2)	4	5	6/11	7/12	6/11	7/12
ADD (3)	5	6	7/12	8/13	7/12	8/13
ADDB (2)	4	4	6/11	7/12	6/11	7/12
ADDB (3)	5	5	7/12	8/13	7/12	8/13
ADDC	4	5	6/11	7/12	6/11	7/12
ADDCB	4	4	6/11	7/12	6/11	7/12
AND (2)	4	5	6/11	7/12	6/11	7/12
AND (3)	5	6	7/12	8/13	7/12	8/13
ANDB (2)	4	4	6/11	7/12	6/11	7/12
ANDB (3)	5	5	7/12	8/13	7/12	8/13
BR [indirect]	8					

Table A-4 8096 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect Normal	Auto-Inc	Indexed Short	Long
CLR	4					
CLRB	4					
CLRC	4					
CLRVT	4					
CMP	4	5	6/11	7/12	6/11	7/12
CMPB	4	4	6/11	7/12	6/11	7/12
DEC	4					
DECB	4					
DI	4					
DIV	29	30	32/36	33/37	32/36	33/37
DIVB	21	21	24/28	25/29	24/28	25/29
DIVU	25	26	28/32	29/33	28/32	29/33
DIVUB	17	17	20/24	21/25	20/24	21/25
DJNZ	5 [jump not taken], 9 [jump taken]					
EI	4					
EXT	4					
INC	4					
INCB	4					
JBC	5 [jump not taken], 9 [jump taken]					
JBS	5 [jump not taken], 9 [jump taken]					
JC	4 [jump not taken], 8 [jump taken]					
JE	4 [jump not taken], 8 [jump taken]					
JGE	4 [jump not taken], 8 [jump taken]					
JGT	4 [jump not taken], 8 [jump taken]					
JH	4 [jump not taken], 8 [jump taken]					
JLE	4 [jump not taken], 8 [jump taken]					
JLT	4 [jump not taken], 8 [jump taken]					
JNE	4 [jump not taken], 8 [jump taken]					
JNH	4 [jump not taken], 8 [jump taken]					
JNST	4 [jump not taken], 8 [jump taken]					

Table A-4 8096 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect		Indexed	
			Normal	Auto-Inc	Short	Long
JNC	4 [jump not taken], 8 [jump taken]					
JNV	4 [jump not taken], 8 [jump taken]					
JNVT	4 [jump not taken], 8 [jump taken]					
JST	4 [jump not taken], 8 [jump taken]					
JV	4 [jump not taken], 8 [jump taken]					
JVT	4 [jump not taken], 8 [jump taken]					
LCALL						
internal stack	13					
external stack	16					
LD	4	5	6/11	7/12	6/11	7/12
LDB	4	5	6/11	7/12	6/11	7/12
LDBSE	4	4	6/11	7/12	6/11	7/12
LDBZE	4	4	6/11	7/12	6/11	7/12
LJMP	7					
MUL (2)	29	30	31/36	32/37	31/36	32/37
MUL (3)	30	31	32/37	33/38	32/37	33/38
MULB (2)	21	21	23/28	24/29	23/28	24/29
MULB (3)	22	22	24/29	25/30	24/29	25/30
MULU (2)	25	26	27/32	28/33	27/32	28/33
MULU (3)	26	27	28/33	29/34	28/33	29/34
MULUB (2)	17	17	19/24	20/25	19/24	20/25
MULUB (3)	18	18	20/25	21/26	20/25	21/26
NEG	4					
NEGB	4					
NOP	4					
NORML	11 + 1 per shift					
NOT	4					
NOTB	4					
OR (20)	4	5	6/11	7/12	6/11	7/12
ORB (2)	4	4	6/11	7/12	6/11	7/12

Table A-4 8096 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct	Immed	Indirect		Indexed	
			Normal	Auto-Inc	Short	Long
POP						
internal stack	12	--	14/18	14/18	14/18	14/18
external stack	14	--	16/20	16/20	16/20	16/20
POPF						
internal stack	9					
external stack	13					
PUSH						
internal stack	8	8	11/15	12/16	11/15	12/16
external stack	12	12	15/19	16/20	15/19	16/20
PUSHF						
internal stack	8					
external stack	12					
RET						
internal stack	12					
external stack	16					
RST	166					
SCALL						
internal stack	13					
external stack	16					
SETC	4					
SHL	7 + 1 per shift [8 for 0 shift]					
SHLB	7 + 1 per shift [8 for 0 shift]					
SHLL	7 + 1 per shift [8 for 0 shift]					
SHR	7 + 1 per shift [8 for 0 shift]					
SHRA	7 + 1 per shift [8 for 0 shift]					
SHRAB	7 + 1 per shift [8 for 0 shift]					
SHRAL	7 + 1 per shift [8 for 0 shift]					
SHRB	7 + 1 per shift [8 for 0 shift]					
SHRL	7 + 1 per shift [8 for 0 shift]					
SJMP	8					

Table A-4 8096 Instruction Cycle Times (continued)

Mnemonic (# of ops)	Direct		Immed		Indirect		Indexed	
	Short	Long	Short	Long	Normal	Auto-Inc	Short	Long
SKIP	4							
ST	4	--	--		7/11	8/12	7/11	8/12
STB	4	--	--		7/11	8/12	7/11	8/12
SUB (2)	4	5	5		6/11	7/12	6/11	7/12
SUB (3)	5	6	6		7/12	8/13	7/12	8/13
SUBB (2)	4	4	4		6/11	7/12	6/11	7/12
SUBB (3)	5	5	5		7/12	8/13	7/12	8/13
SUBC	4	5	5		6/11	7/12	6/11	7/12
SUBCB	4	4	4		6/11	7/12	6/11	7/12
TRAP								
internal stack	21							
external stack	24							
XOR	4	5	5		6/11	7/12	6/11	7/12
XORB	4	4	4		6/11	7/12	6/11	7/12

B

Assembler Directives



CSEG

Category: Location Counter Control

Syntax: CSEG [REL | AT *base address*]

Function: Selects a code segment, either relocatable or beginning at absolute address specified.

DCB

Category: Code Definition

Syntax: [*label*:] DCB {*expression* | *string*} [, . . .]

Function: Specifies a list of zero or more byte values to be inserted sequentially starting at the position indicated by the location counter.

DCL

Category: Code Definition

Syntax: [*label*:] DCL *long-unsigned-number* [, . . .]

Function: Specifies a list of one or more long unsigned numbers to be inserted sequentially starting at the position indicated by the location counter.

DCR

Category: Code Definition

Syntax: [*label*:] DCR *real number* [, . . .]

Function: Specifies a list of one or more real numbers to be inserted sequentially starting at the position indicated by the location counter.

DCW

Category: Code Definition

Syntax: `[label:] DCW expression[ , . . . ]`

Function: Specifies a list of one or more word values to be inserted sequentially starting at the position indicated by the location counter.

DSB

Category: Storage Reservation

Syntax: `[label:] DSB expression`

Function: Reserves storage for byte program variables starting at position of location counter.

DSEG

Category: Location Counter Control

Syntax: `DSEG [REL | AT base address]`

Function: Selects a data segment either relocatable or beginning at absolute address specified.

DSL

Category: Storage Reservation

Syntax: `[label:] DSL expression`

Function: Reserves storage for long word program variables starting at (aligned) position of location counter.

DSR

Category: Storage Reservation

Syntax: `[label:] DSR expression`

Function: Reserves storage of real number program variables starting at (aligned) position of location counter.

DSW

Category: Storage Reservation

Syntax: `[label:] DSW expression`

Function: Reserves storage for word program variables starting at (aligned) position of location counter.

ELSE	
Category:	Conditional Assembly
Syntax:	[ELSE] [<i>statements</i>]
Function:	Provides alternate in conditional assembly block.
END	
Category:	Module Level
Syntax:	END
Function:	Marks the end of program.
ENDIF	
Category:	Conditional Assembly
Syntax:	ENDIF
Function:	Ends conditional assembly block.
EQU	
Category:	Symbol Definition
Syntax:	<i>symbol name</i> EQU <i>expression</i> [: <i>data type</i>]
Function:	Defines symbols that may not be redefined.
EXTRN	
Category:	Module Level
Syntax:	EXTRN { <i>symbol name</i> [: <i>data type</i>] } [, . . .]
Function:	Declares one or more symbols as external.
IF	
Category:	Conditional Assembly
Syntax:	IF <i>expression</i> [<i>statements</i>]
Function:	Starts conditional assembly block.
MODULE	
Category:	Module Level
Syntax:	<i>module name</i> MODULE [<i>attr</i> [, <i>attr</i>]] where <i>attr</i> can be either MAIN or STACKSIZE(<i>n</i>)
Function:	Defines a module and its attributes.

ORG

Category: Location Counter Control
Syntax: ORG *expression*
Function: Sets value of location counter.

OSEG

Category: Location Counter Control
Syntax: OSEG [REL | AT *base address*]
Function: Selects an overlayable register segment either relocatable or beginning at absolute address specified.

PUBLIC

Category: Module Level
Syntax: PUBLIC *symbol name* [, . . .]
Function: Declares one or more symbols public.

RSEG

Category: Location Counter Control
Syntax: RSEG [REL | AT *base address*]
Function: Selects a non-overlayable register segment either relocatable or beginning at absolute address specified.

SET

Category: Symbol Definition
Syntax: *symbol name* SET *expression* [:*data type*]
Function: Defines symbols that may be redefined.

C

Assembler Controls



Table C-1 provides the following five types of information about each assembler control:

- The Function column indicates the general effect of each control.
- The Name column gives the full name of each control. You can suppress many of the control functions by prefacing the control or its abbreviation with NO.
- The Abbr column shows the two-letter abbreviation for each control. If the control can be prefaced with NO, the two-letter abbreviation can also be prefaced with NO.
- The Default column gives the default setting the assembler assigns for each control.
- The P,G column indicates whether each control is primary or general (see Chapter 6).

Table C-1 Assembler Controls

Function	Name	Abbr	Default	P,G
Input Processing	INCLUDE	IC	-	G
Object Generation	[NO]DEBUG	DB	NODEBUG	P
	[NO]OBJECT	OJ	OBJECT(source.OBJ)	P
Listing Configuration	[NO]COND	CO	COND	G
	EJECT	EJ	-	G
	[NO]ERRORPRINT	EP	NOERRORPRINT	P
	[NO]GEN	GE	NOGEN	G
	[NO]LIST	LI	LIST	G
	PAGELength	PL	PAGELength(60)	P
	PAGEWIDTH	PW	PAGEWIDTH(120)	P
	[NO]PRINT	PR	PRINT(source.LST)	P
	RESTORE	RS	-	G
	SAVE	SA	-	G
	[NO]SYMBOLS	SB	SYMBOLS	P
	TITLE	TT	TITLE(modulename)	G
	[NO]XREF	XR	NOXREF	P

D

ASM96 Reserved Words



ADD
ADDB
ADDC
ADDCB
AND
ANDB
AT
BBC
BBS
BC
BE
BGE
BGT
BH
BLE
BLT
BMOV*
BMOVI*
BNC
BNE
BNH
BNST
BNV
BNVT
BR
BST
BV
BVT
BYTE
CALL
CLR
CLRB
CLRC
CLRVT
CMP
CMPB
CMPL*

CSEG
DBNZ
DBNZW
DCB
DCL
DCR
DCW
DEC
DECB
DI
DIV
DIVB
DIVU
DIVUB
DJNZ
DJNZW*
DPTS*
DSB
DSEG
DSL
DSR
DSW
EI
ELSE
END
ENDIF
ENDM
ENTRY
EPTS*
EQ
EQU
EXITM
EXT
EXTB
EXTRN
GE
GT

IDLDP*
IF
INC
INCB
IRP
IRPC
JBC
JBS
JC
JE
JGE
JGT
JH
JLE
JLT
JNC
JNE
JNH
JNST
JNV
JNVT
JST
JV
JVT
LCALL
LD
LDB
LDBSE
LDBZE
LE
LJMP
LOCAL
LONG
LT
MACRO
MAIN
MOD

MULU
MULUB
NE
NEG
NEGB
NOP
NORML
NOT
NOTB
NUL
NULL
OR
ORB
ORG
OSEG
POP
POPA*
POPF
PUBLIC

PUSH
PUSHA*
PUSHF
REL
REPT
RET
RSEG
RST
SCALL
SET
SETC
SHL
SHLB
SHLL
SHR
SHRA
SHRAB
SHRAL
SHRB

SHRL
SJMP
SKIP
ST
STACK
STACKSIZE
STB
SUB
SUBB
SUBC
SUBCB
WORD
TIJMP*
XCH*
XCHB*
XOR
XORB

* This instruction is specific to 80196.

E

Error Messages



Error messages are of two types: console error messages that are fatal and cause the assembly to abort and source file error messages that do not cause the assembly to abort but merely cause an informative message to appear in the listing file after the offending statement. However, even though the assembly will continue, the object file will probably not be executable. Be sure to correct the error and then reassemble your program.

E.1 Console Error Messages

Console error messages can be divided into four categories: (1) invocation line error messages, (2) I/O error messages, (3) insufficient (host) memory error messages, and (4) internal synchronization error messages.

Invocation line errors are in the form:

```
INVOCATION LINE ERROR
      message
ASSEMBLY TERMINATED
```

Where:

message is an English language message describing the cause of the error.

I/O error messages are in the form:

```
# I/O ERROR
    FILE: file type
    NAME: pathname
    EXCEPTION nnnnH: message
ASSEMBLY TERMINATED
```

Where:

<i>file type</i>	is the type of the file (SOURCE, INCLUDE, PRINT, OBJECT, or ERROR PRINT).
<i>pathname</i>	is the pathname of the file.
<i>nnnnH</i>	is the hexadecimal code number.
<i>message</i>	is an English language message describing the cause of the error.

Insufficient memory messages are in the form:

```
# INSUFFICIENT MEMORY
ASSEMBLY TERMINATED
```

Internal synchronization errors are in the form:

```
# SYNCHRONIZATION ERROR
ASSEMBLY TERMINATED
```

E.2 Source File Error Messages

Source file error messages are not catastrophic in that they do not cause the assembly to abort. An error line appears in the listing following the source statement that caused the error. The error is indicated by three asterisks (***), a number sign (#) followed by the error number and error message. While not catastrophic, source file errors can cause the object code to be unable to execute. Correct any source file errors and reassemble your program before attempting to execute it.

The messages listed by their decimal codes are:

#10 SYNTAX ERROR

The line contains a syntax error.

#11 UNRECOGNIZED CONTROL OR MISPLACED PRIMARY

#12 PRIMARY CONTROL SPECIFIED MORE THAN ONCE

The control can be specified at most once.

#13 MISSING PARAMETER FOR CONTROL

A parameter is omitted for a control that requires a parameter.

#14 NO PARAMETER ALLOWED WITH CONTROL

A parameter is specified for a parameterless control.

#15 INVALID NUMBER

A number contains illegal characters for its base, or its value exceeds 65535 ($2^{16}-1$).

#16 BAD PARAMETER TO CONTROL

The parameter of one of the controls in the line is illegal (e.g., out of allowed range).

#18 CONTRADICTING CONTROLS

XREF specified with NOSYMBOLS or NOPRINT, or
SYMBOLS/PAGEWIDTH/PAGELength specified with NOPRINT, or DEBUG
specified with NOOBJECT.

#19 SAVE-RESTORE STACK OVERFLOW

Too many SAVES without corresponding RESTORES. The maximum nesting is nine.

#20 SAVE-RESTORE STACK UNDERFLOW

A RESTORE control without a corresponding SAVE.

#25 INCLUDE NOT RIGHTMOST CONTROL

#26 INCLUDE STACK OVERFLOW

INCLUDE nesting too deep. The maximum nesting is nine.

#27 INPUT LINE TOO LONG

The maximum length is 255 characters including the terminating line feed.

#28 INVALID CHARACTER

A control character was imbedded in the input. The character is replaced in the listing line by a #.

#29 MISSING APOSTROPHE

A string was not terminated.

#30 INVALID LINE TERMINATOR

End of file was encountered before a line feed.

#31 PARSE STACK OVERFLOW

The statement is too complicated.

#32 TEXT FOUND BEYOND END STATEMENT - IGNORED

#33 PREMATURE END OF FILE (NO END STATEMENT)

#34 STATEMENT NOT ALLOWED IN THIS CONTEXT

LOCAL and EXITM are allowed only within macro definitions. ELSE is allowed only between IF and ENDIF. ENDM is allowed only when terminating a macro definition. ENDIF is allowed only when matching a preceding IF directive. MODULE directive can be preceded only by empty statements and control lines. ORG cannot appear before the first segment selection directive. Machine instruction and code definition directives must appear in code segments. Storage reservation directives can appear only within register, overlay, or data segments.

The statement is ignored.

#35 EXPRESSION STACK OVERFLOW

Expression nesting is too deep.

#36 INVALID RELOCATABLE EXPRESSION

The relocatable expression contains an illegal operation or operand.

#37 ATTEMPT TO DIVIDE BY ZERO

The expression contains a zero divider.

#38 UNDEFINED SYMBOL

The line contains an undefined symbol.

#39 ILLEGAL BINARY OPERATION IN EXPRESSION

Binary operation is illegal if both operands have a segment type other than null, except for subtraction and comparing operands that belong to the same segment.

#40 INVALID CODE ADDRESS

Code address must have a code (or null) segment type and be above 255.

#41 INVALID REGISTER ADDRESS

Register address must have a register (or null) segment type and be below 256.

#42 DATA TYPE NOT COMPATIBLE WITH SEGMENT TYPE

ENTRY attribute can be assigned only to a code segment type or null expression over 256.

#43 PUBLIC ATTRIBUTE NOT ALLOWED FOR THIS SYMBOL

Public attribute cannot be attached to a MACRO, MODULE, or to an external symbol.

#44 EXTERNAL REFERENCE NOT ALLOWED IN THIS CONTEXT

External references are allowed only in code generation statements (machine instructions and the DCB and DCW directives).

#45 EXPRESSION WITH FORWARD REFERENCE NOT ALLOWED

Forward reference is not allowed for this statement (e.g., DSB directive).

#46 ABSOLUTE EXPRESSION EXPECTED

The directive requires an absolute expression as an operand.

#47 SYMBOL ALREADY DEFINED

Symbol cannot be redefined unless it has been defined by a SET directive and it is redefined by a SET directive.

#48 INVALID BIT NUMBER

A bit number should be an expression in range of 0 to 7.

#49 ALIGNMENT ERROR

The address or offset is not properly aligned.

#50 REFERENCE NOT TO CURRENT SEGMENT

The ORG directive requires that the expression will refer to the current segment (or be absolute and neutral).

#51 LOCATION COUNTER TOO LOW

The operand of a CSEG AT directive should be at least 256. The operand of an ORG directive must not point below the base address of the segment (applicable only within an absolute segment).

#52 LOCATION COUNTER OVERFLOW

The location counter is incremented beyond 65535, or beyond 255 for register segment.

#53 DECISION LIST OVERFLOW

Too many generic jumps. Further generic jumps are expanded in their long form.

#54 EXPRESSION NOT WITHIN RANGE

A register expression is not in the range of 0 to 255. An immediate count in a shift is not in the range of 0 to 15. A register count in a shift is not in the range of 16 to 255. An immediate operand in byte instructions or a DCB constant is not in the range of -256 to +255. The offset in JCOND, JBS, JBC, or DJNZ is not in the range of -128 to +127. The offset in SJMP or SCALL is not in the range of -1024 to +1023.

#55 CONDITIONAL ASSEMBLY STACK OVERFLOW

Too many IF statements without corresponding ENDIFs. The maximum nesting is nine.

#56 MISSING ENDIF

An IF block was not terminated until the end of the expansion of the current macro, until the END directive, or until the end of the file.

#57 TOO MANY FORMAL PARAMETERS

A macro can have no more than 16 formal parameters.

#58 TOO MANY LOCAL PARAMETERS

A macro can have no more than 16 local parameters.

#59 MACRO STACK OVERFLOW

Macro call nesting too deep. The maximum nesting is nine.

#60 TOO MANY ACTUAL PARAMETERS

A macro is called with more actual parameters than formal parameter in its definition.

#61 MISSING ENDM

A macro definition was not terminated until the END directive, or until the end of the file.

#62 STACKSIZE NOT EVEN

#63 FLOATING POINT CONSTANT UNDERFLOW

Absolute value of floating point constants must be above 1.17E-38.

#64 FLOATING POINT CONSTANT OVERFLOW

Absolute value of floating point constants must be below 3.37E38.

\$ (dollar sign), 2-8, 2-10
% (percent sign), 5-18
& (ampersand), 5-13, 6-2

A

Absolute

Expression, 4-7

Segment, 4-7

Absolute values, 2-3

A/D conversion, 1-20

A/D converter, 1-25

Capacity, 1-14

Interrupts, 1-31

Pins, 1-23

ADD (Two Operands), 3-12

ADD (Three Operands), 3-13

ADDB (Two Operands), 3-14

ADDB (Three Operands), 3-15

ADDC, 3-16

ADDCB, 3-17

Address space, 1-15

Alignment, 2-20

Expressions, 4-10

Label, 4-12

Operands, 3-5

Program variables, 4-12

Real number, 4-11

Unsigned number, 4-11

Ampersand (&), 5-13, 6-2

AND (Two Operands), 3-18

AND (Three Operands), 3-19

ANDB (Two Operands), 3-20

ANDB (Three Operands), 3-21

Attributes

Assigning to module, 2-17

Defining symbols, 4-8

Location, 2-13

Type, 2-14

B

Batch files, 6-2

BBC, 3-22

BBS, 3-23

BC, 3-24

BE, 3-25

BGE, 3-26

BGT, 3-27

BH, 3-28

BLE, 3-29

BLT, 3-30

BMOV, 3-31

BMOVI, 3-33

BNC, 3-34

BNE, 3-35

BNH, 3-36

BNST, 3-37

BNV, 3-38

BNVT, 3-39

BR (Direct), 3-40, 3-42

BR (Indirect), 3-41

Bus, 1-12

BV, 3-43

BVT, 3-44

Byte operations, 1-12

C

CALL (Direct), 3-45

Calls

Conditional, 3-10

External, 3-10

Forward, 3-10

Generic, 3-10

Carriage return, 2-16

Case sensitivity

Controls, 6-3

Reserved words, 2-7

Symbols, 2-7

Case significance, 2-8

CLR, 3-46

CLRB, 3-47

CLRC

CLRVT, 3-49

CMP, 3-50

CMPB, 3-51

CMPL, 3-52

Comment

Statement structure, 2-16

Computed addressing, 3-9

COND/NOCOND, 6-7

Conditional jumps, 3-10

Constants

Expressions, 2-14

Content addressable memory, 1-20, 1-22

Continuation lines, 2-16, 6-2

Control lines, 2-17

Controls, 1-2

D

Data, 1-5

Data representation

Alignment, 4-11

Numbers, 2-4 thru 2-6

Operands, 2-10

Real number, 4-11

Strings, 2-8

Unsigned number, 4-11

Word values, 4-10

Data structures

Macros, 5-4

Subroutines, 5-4

Data transfers

Interrupts, 1-31

DBNZ, 3-53

DBNZW, 3-55

DEBUG/NODEBUG, 6-8

DEC, 3-57

DECB, 3-58

Defaults

Controls, 6-3

Delimiters, 5-16

DI, 3-59

Diagnostics, 1-5

Directives, 1-2

DIV, 3-60

DIVB, 3-61

DIVU, 3-62

DIVUB, 3-63

DJNZ, 3-64

DJNZW, 3-65

Dollar sign (\$), 2-8, 2-10

DPTS, 3-66

E

Efficiency

Macros, 5-2, 5-4

Subroutines, 5-4

EI, 3-67

EJECT, 6-9

End-of-file, 2-17

EPTS, 3-68

ERRORPRINT/NOERRORPRINT, 6-10

Errors

In listing file, 7-11

EXT, 3-69

EXTB, 3-70

External events, 1-14

External memory, 1-15

External reference, 3-10, 7-10

F

FIFO, 1-19

Filename extensions, 1-6

Files

Identified, 7-9

Include, 6-12

Listing, 6-1, 6-17

Object, 6-1, 6-14

Source, 6-1, 6-12

Fixup, 7-10

Forward references, 3-10

G

GEN/NOGEN, 6-11

General controls, 6-5

Generic instructions

In listing file, 7-11

H

Hardware development, 1-7

High-speed input, 1-14, 1-16, 1-19

Interrupts, 1-31

High-speed output, 1-14, 1-16, 1-20

Interrupts, 1-31

I

IDLDP, 3-71

INC, 3-72

INCB, 3-73
INCLUDE, 6-12
In-line code, 5-4
Instructions, 1-2, 1-5
Internal memory, 1-16
Interrupt mask, 1-27, 1-28, 1-30
Interrupt priorities, 1-27, 1-28
Interrupt vectors, 1-16, 1-27, 1-28
Interrupts, 1-19
 Software timers, 1-22
I/O ports, 1-22

J

JBC, 3-74
JBS, 3-75
JC, 3-76
JE, 3-77
JGE, 3-78
JGT, 3-79
JH, 3-80
JLE, 3-81
JLT, 3-82
JNC, 3-83
JNE, 3-84
JNH, 3-85
JNST, 3-86
JNV, 3-87
JNVT, 3-88
JST, 3-89
Jumps
 Conditional, 3-10
 External, 3-10
 Forward, 3-10
 Generic, 3-10
JV, 3-90
JVT, 3-91

L

Labels, 2-16
 Definition, 2-14, 4-12, 5-8
 Macros, 5-5, 5-8
 Scope, 5-5
 Statement structure, 2-14
LCALL, 3-92
LD, 3-93

LDB, 3-94
LDBSE, 3-95
LDBZE, 3-96
LIB96, 1-4
Libraries, 1-4
Line feed, 2-16
Line numbers, 7-10
Linking, 2-18
LIST/NOLIST, 6-13
Listing file, 6-1
LJMP, 3-97
Loading, 1-5
Location attribute, 2-13
Location counter, 4-12, 7-9
Long indexing, 3-6, 3-9

M

Macro expansion
 In listing file, 7-10
Macros
 Arguments, 2-12
 Definition, 2-12
 Operators, 2-12
MUL (Two Operands), 3-98
MUL (Three Operands), 3-99
MULB (Two Operands), 3-100
MULB (Three Operands), 3-101
MULU (Two Operands), 3-102
MULU (Three Operands), 3-103
MULUB (Two Operands), 3-104
MULUB (Three Operands), 3-105

N

Named
 Statement structure, 2-15
Names
 Assembler-generated, 2-8
 Attributes, 2-15
 Case sensitivity, 2-7
 Macros, 2-8
 Reserved words, 2-7
 Symbols, 2-8
Names, case sensitivity, 2-7
Names character set, 2-7
NEG, 3-106

Negating controls, 6-3

NEGB, 3-107

NOP, 3-108

NORML, 3-109

NOT, 3-110

NOTB, 3-111

O

Object code listing, 7-10

Object file, 6-1

OBJECT/NOOBJECT, 6-14

OH, 1-6

Operands

Absolute, 2-13

Alignment, 3-5

Bit, 3-11

Definition, 2-9

Expression structure, 2-9

Expressions, 3-4

Registers, 3-4

Relocatability, 2-12

Size, 1-12

Statement structure, 2-15

Type attribute, 2-11

Value, 2-12

Operations

Type attribute, 2-12

Operators

Delimiters, 2-13

Expression structure, 2-9

Macros, 2-12, 5-13

Prefix unary, 2-9

Relational, 2-12, 2-13

Unary, 2-12

Operators infix, 2-9

OR, 3-112

ORB, 3-113

Oscillator, 1-12

Overlayable registers, 4-6

Overlaying, 2-18

P

PAGELength, 6-15

PAGEWIDTH, 6-16

Parameters

Controls, 6-3

Macros, 5-2 thru 5-6, 5-13

Re-evaluation, 5-18

Scope, 5-5

Special characters, 5-13

Percent sign (%), 5-18

Peripheral transaction server, 1-30

Peripherals, 1-12

POP, 3-114

POPA, 3-115

POPF, 3-116

Power down, 1-16

Preserving memory, 1-16

Primary controls, 6-5

PRINT/NOPRINT, 6-17

Program variables, 4-12

Pulse width modulator, 1-24

PUSH, 3-117

PUSHA, 3-118, 3-119

Q

Quote marks

" (double quote), 2-8

' (single quote), 2-8

R

Register addressing, 3-6

Register expansion, 1-19

Relocatable

Expression, 4-7

Segment, 4-7

Relocatable reference, 7-10

Reserved memory, 1-15

RESET, 1-16, 1-32

RET, 3-120

RL96, 1-6, 1-7

RST, 3-121

S

SAVE/RESTORE, 6-18

SCALL, 3-122

Scope

Labels, 5-5

Macros, 5-5

- Scope (continued)
 - Parameters, 5-5
- Segment
 - Absolute, 4-6
 - Relocatable, 4-6
 - Type attribute, 2-11, 2-12
- Serial port, 1-24
- SETC, 3-123
- Shift
 - Count, 3-10
- SHL, 3-124
- SHLB, 3-125
- SHLL, 3-126
- Short indexing, 3-9
- SHR, 3-127
- SHRA, 3-128
- SHRAB, 3-129
- SHRAL, 3-130
- SHRB, 3-131
- SHRL, 3-132
- SJMP, 3-133
- SKIP, 3-134
- Source file, 6-1
- Source listing, 7-11
- Space, 2-16
- Special characters
 - Macro parameters, 5-13
- Special function registers, 1-15, 1-32
- ST, 3-135
- Stack
 - Symbol, 2-10
- Stack pointer, 1-16
- State time, 1-12
- Statement structure, 2-1
 - Character set, 2-2
 - Delimiters, 2-6, 2-8
 - Expressions, 2-9
 - Numbers, 2-2
 - Operands, 2-9, 2-10
 - Operators, 2-9
 - Strings, 2-8
 - Symbols, 2-7
- STB, 3-136

- Strings
 - Defining, 4-9
 - Expressions, 2-10
- SUB (Two Operands), 3-137
- SUB (Three Operands), 3-138
- SUBB (Two Operands), 3-139
- SUBB (Three Operands), 3-140
- SUBC, 3-141
- SUBCB, 3-142
- Symbolic debugging, 1-4
- Symbols
 - Expressions, 2-14
 - External, 2-14
 - Macros, 5-1, 5-5
 - Relocatable, 2-14
 - Scope, 5-5
- SYMBOLS/NOSYMBOLS, 6-19

T

- TIJMP, 3-143
- Timers, 1-19
- TITLE, 6-20
- Toggling controls, 6-18
- Type attribute, 2-14

W

- Watchdog timer, 1-32
- Window select register, 1-19
- Word operations, 1-12

X

- XCH, 3-144
- XCHB, 3-145
- XOR, 3-146
- XORB, 3-147
- XREF/NOXREF, 6-21

1. The first part of the report is a general introduction to the subject of the study. It discusses the importance of the study and the objectives of the research.

2. The second part of the report is a detailed description of the methodology used in the study. It includes information about the sample size, the data collection methods, and the statistical analysis techniques.

3. The third part of the report is a discussion of the results of the study. It presents the findings of the research and compares them with the previous studies in the field.

4. The fourth part of the report is a conclusion and a list of references. The conclusion summarizes the main findings of the study and provides recommendations for future research.

5. The fifth part of the report is an appendix containing additional information related to the study, such as raw data, detailed calculations, and supplementary figures.

6. The sixth part of the report is a bibliography listing all the sources used in the study. It includes books, journal articles, and other relevant literature.

7. The seventh part of the report is a glossary of terms used in the study. It provides definitions for key concepts and variables.

8. The eighth part of the report is a list of abbreviations and acronyms used throughout the document. It helps to clarify the meaning of these terms and makes the text easier to read.

9. The ninth part of the report is a list of figures and tables. It provides a brief description of each figure and table and indicates where they can be found in the report.

10. The tenth part of the report is a list of footnotes. It contains additional information that is not directly related to the main text but is still relevant to the study.

11. The eleventh part of the report is a list of appendices. It provides a brief description of each appendix and indicates where it can be found in the report.



READER RESPONSE CARD

MCS®-96 Macro Assembler
User's Guide for DOS Systems
122350-003

We'd Like Your Opinion

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future versions.

To order publications, contact the Intel Literature Distribution Center (see page ii of this manual).

Fill in the squares below with a rating of 1 through 10:

POOR

AVERAGE

EXCELLENT

1

2

3

4

5

6

7

8

9

10

- ☐ Readability
- ☐ Technical depth
- ☐ Technical accuracy
- ☐ Usefulness of material for your needs
- ☐ Comprehensibility of material
- ☐ OVERALL QUALITY OF THIS MANUAL

If you gave a 4 or less (in any category), please explain here:

What suggestions would you have for improving this manual:

If you would like us to call you for more specific suggestions about this book, please additionally fill in your phone number below.

Name

Phone Number (

)

Address

Thanks for taking the time to fill out this form.



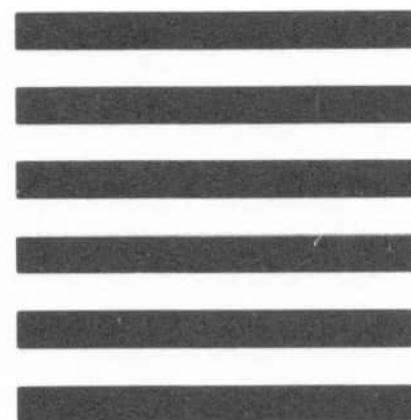
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

**INTEL CORPORATION
DTO TECHNICAL PUBLICATIONS JF1-59
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978**



Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.



International Sales Offices

AUSTRALIA

Intel Australia Pty. Ltd.
Spectrum Building
200 Pacific Hwy., Level 6
Crows Nest, NSE, 2065

BRAZIL

Intel Semicondutores do Brazil LTDA
Av. Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.

CANADA

Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
British Columbia

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Ontario

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Ontario

Intel Semiconductor of Canada, Ltd.
620 St. Jean Boulevard
Pointe Claire H9R 3K2
Quebec

CHINA/HONG KONG

Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC

Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central
Hong Kong

DENMARK

Intel Denmark A/S
Glentevej 61, 3rd Floor
2400 Copenhagen NV

FINLAND

Intel Finland OY
Ruosilantie 2
00390 Helsinki

FRANCE

Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines
Cedex

WEST GERMANY

Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

Intel Semiconductor GmbH
Hohenzollern Strasse 5
3000 Hannover 1

Intel Semiconductor GmbH
Abraham Lincoln Strasse 16-18
6200 Wiesbaden

Intel Semiconductor GmbH
Zettachring 10A
7000 Stuttgart 80

INDIA

Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001

ISRAEL

Intel Semiconductor Ltd.
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY

Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20090 Assago
Milano

JAPAN

Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

Intel Japan K.K.
Daiichi Mitsugi Bldg.
1-8889 Fuchu-cho
Fuchu-shi, Tokyo 183

Intel Japan K.K.
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360

Intel Japan K.K.
Mitsui-Seimei Musashi-kosugi Bldg.
915 Shinmaruko, Nakahara-ku
Kawasaki-shi, Kanagawa 211

Intel Japan K.K.
Nihon Seimei Atsugi Bldg.
1-2-1 Asahi-machi
Atsugi-shi, Kanagawa 243

Intel Japan K.K.
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 450

KOREA

Intel Technology Asia, Ltd.
16th Floor, Life Bldg.
61 Yoido-Dong, Youngdeungpo-Ku
Seoul 150-010

NETHERLANDS

Intel Semiconductor B.V.
Postbus 84130
3099 CC Rotterdam

NORWAY

Intel Norway A/S
Hvamveien 4-PO Box 92
2013 Skjetten

SINGAPORE

Intel Singapore Technology, Ltd.
101 Thomson Road #21-05/06
United Square
Singapore 1130

SPAIN

Intel Iberia S.A.
Zurbaran, 28
28010 Madrid

SWEDEN

Intel Sweden A.B.
Dalvagen 24
171 36 Solna

SWITZERLAND

Intel Semiconductor A.G.
Zuerichstrasse
8185 Winkel-Rueti bei Zuerich

TAIWAN

Intel Technology Far East Ltd.
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ

SERVICE INFORMATION



For service or assistance with Intel products, call:

- **1-800-INTEL-4-U** (1-800-468-3548) - in the United States and Canada
- Your local Intel sales office - in Europe or Japan
- Your Intel distributor - in any other area

Intel stands behind its products with a world-wide service and support organization. If you have problems, need assistance, or have a question, Intel can provide:

- On-site or carry-in service for hardware products
- Phone support for all Intel products
- On-site consulting for designing with Intel products or using Intel products in your designs
- Customer training workshops
- Updates to software products

To get more information on these services or to take advantage of them, call the INTEL-4-U number above.

All Intel products have a minimum warranty of 90 days, and all warranties include one or more of the services listed above. Talk with your Intel salesperson or call the INTEL-4-U number to determine the warranty services available for this product and how to register for them.

Intel is committed to continuing service for all its products.



INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95052-8126 (408) 987-8080

Printed in U.S.A.

DEVELOPMENT TOOLS AND SOFTWARE