# MCS-51™ MACRO ASSEMBLER USER'S GUIDE

Manual Order Number 9800937-01

# MCS-51™ MACRO ASSEMBLER USER'S GUIDE

Manual Order Number 9800937-01


Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

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

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

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

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

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



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

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

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

# PREFACE

This manual describes how to program the MCS-51 single-chip microcomputers in assembly language. It also describes the operating instructions for the MCS-51 Macro Assembler.

The term "MCS-51" refers to an entire family of single-chip microcomputers, all of which have the same basic processor design. They include:

8051—the 8×51 processor with no ROM on-chip.

8351—the 8×51 processor with 4K bytes ROM. It is manufactured by Intel with ROM memory pre-programmed.

8751—the 8×51 processor with 4K bytes EPROM. The 8751 can be programmed and erased many times by the user.

Throughout this manual when we wish to refer to a specific chip, but also point out something that is true for the entire family, we speak of the 8051.

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

"Chapter 1—Introduction," which describes assembly language programming and provides an overview of 8051 hardware.

"Chapter 2—Operands and Assembly-Time Expressions," which describes each operand class and discusses assembly-time expressions.

"Chapter 3—Instruction Set," completely describes the operation of each instruction in alphabetical order.

"Chapter 4—Directives," which describes how to define symbols and describes use of all directives.

"Chapter 5—Macros," which describes the definition and use of the Macro Processing Language.

"Chapter 6—Assembler Operation and Controls," which describes how to invoke the assembler and how to control assembler operation.

"Chapter 7—Assembler Output: Error Messages and Listing File Format," which describes how to interpret error messages and the listing file.

Before you program one of the MCS-51 microcomputers, you should read the *MCS-51 User's Manual*, Order Number 121517.

# CONTENTS

# TABLES



# ILLUSTRATIONS

# CHAPTER 1
# INTRODUCTION

Most lines of source code in an assembly language source program translate into machine instructions. Therefore, the assembly language programmer must be familiar with both the assembly language and the microcomputer for which his program is intended.

The first part of this chapter describes the assembler. The second part describes the features of the MCS-51 single-chip processor from a programmer's point of view—the symbols and instructions that give programmers access to the hardware features.

## 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 may then be programmed into one of the MCS-51 processors and executed. If you have ever written a computer program directly in machine-recognizable form, such as binary or hexadecimal code, you will appreciate the advantages of programming in a symbolic assembly language.

Assembly language operation codes (mnemonics) are easily remembered (MOV for move instructions, ADD for addition). You can also symbolically express addresses and values referenced in the operand field of instructions. Since you assign these names, you can make them as meaningful as the mnemonics for the instructions. For example, if your program must manipulate a date as data, you can assign it the symbolic name DATE. If your program contains a set of instructions used as a timing loop (a set of instructions executed repeatedly until a specific amount of time has passed), you can name the instruction group TIMER.

For your convenience, the assembler has a set of predefined symbols that you may use in your program. They correspond to addressable hardware features described later in this chapter.

## What the Assembler Does

To use the assembler, create a source program with a text file editor. (The text editor is described in the *ISIS-II System User's Guide*, Order Number 9800306.) The source program consists of comments, assembler controls and directives, and assembly language instructions. These instructions are written using mnemonic opcodes and labels as described above.

When you invoke the assembler, specify the ISIS-II filename of your program. The assembler can only be executed under ISIS-II running on an MDS-800 or SERIES-II Model 220, 230 or 240 with 64K of memory and at least one disk or diskette drive.

The assembler's output usually consists of two files:

the object file —containing the translated executable source code,
the listing file —containing a copy of the source and object code in human readable format.

### Object File

The object file is the executable form of the assembler's output. It is recorded in absolute format hex code. This file may then be programmed into an 8751, or it may be executed by an ICE-51 (the In-Circuit Emulator for the MCS-51 microcomputer). The format of this file is described in *Absolute Object File Formats*, Order Number 9800183.

### 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 use only an 8-bit value, the assembler tells you that the value exceeds the permissible range. Chapter 7 describes the format of the listing file.

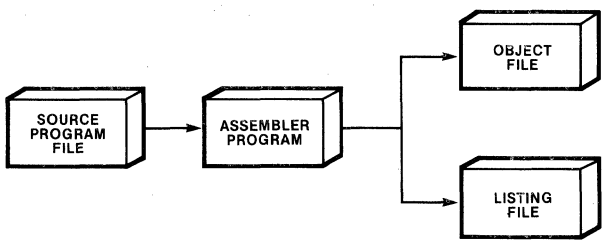



Figure 1-1. Assembler Outputs 937-1

## Writing, Assembling, and Debugging an MCS-51 Program

There are several steps necessary to incorporate an MCS-51 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 *MCS-51 User's Manual*.

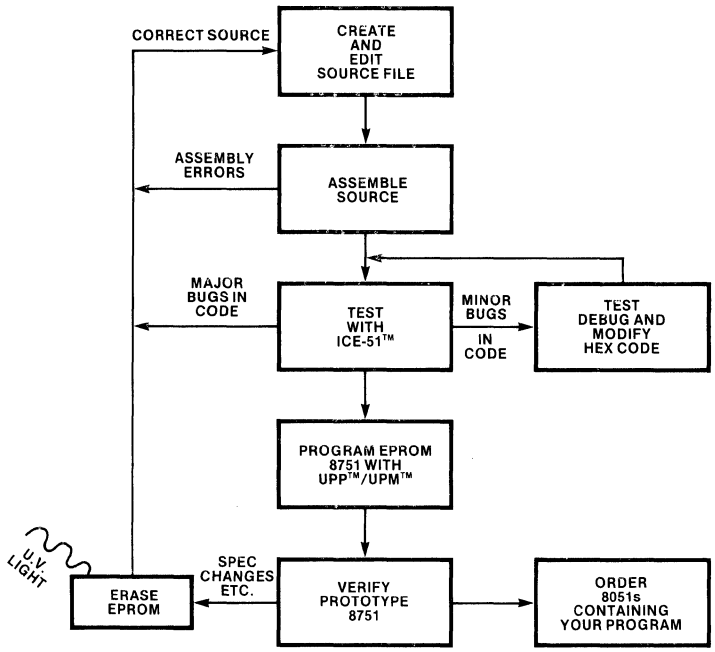



Figure 1-2. MCS-51™ Software Development Flow Chart 937-2

To illustrate the necessary steps let us show how one program was assembled and programmed into an 8751. The program in figure 1-3 was created for use on any member of the MCS-51 family. It is a good starting point to get acquainted with program development in the MCS-51 family. It includes I/O and uses several unique hardware features.

The invocation line and the console output generated by the assembler is shown below. This example assumes two drives in the system. The assembler program is on drive 0 and the source program is on drive 1. The output files will be :F1:TEST.LST (the listing file), and :F1:TEST.HEX (the object file).

```
-ASM51 :F1:TEST.SRC
ISIS-II MCS-51 MACRO ASSEMBLER, V1.0
ASSEMBLY COMPLETE, NO ERRORS FOUND
-
```

Figure 1-3 shows the resulting listing file, :F1:TEST.LST. A complete listing is shown in Appendix G.

The next step in debugging your code is to program it into an EPROM 8751 and test it in a prototype environment. (Further testing could be done via ICE-51.) To program your code into an 8751 you must have a UPP connected to your Intellec system. For a complete description of how to use UPP and UPM see *Universal PROM Programmer Reference Manual*, Order number 9800133 and *Universal PROM Programmer User's Manual*, Order number 9800819.

```
MCS-51 MACRO ASSEMBLER                                                    PAGE    1


ISIS-II MCS-51 MACRO ASSEMBLER V1.0
OBJECT MODULE PLACED IN :F1:TEST.HEX
ASSEMBLER INVOKED BY:  ASM51 :F1:TEST.SRC


LOC  OBJ            LINE     SOURCE

  0032                 1      FIRST_NUMBER  DATA 50  ; STORAGE LOCATION FOR FIRST NUMBER
  003C                 2      SECOND_NUMBER DATA 60  ; STORAGE FOR SECOND NUMBER
0BB8                   3      ORG 3000
                       4      ; These strings will be placed in high memory
                       5      ; They will be used to output messages to the terminal
                       6      ; The 00H byte at the end of each string identifies the end character
0BB8 54595045          7      TYPO:           DB        'TYPE ^X TO RETYPE A NUMBER',00H
0BBC 205E5820
0BC0 544F2052
0BC4 45545950
0BC8 45204120
0BCC 4E554D42
0BD0 4552
0BD2 00
0BD3 54595045          8      F_NUMB:         DB        'TYPE IN FIRST NUMBER: ',00H
0BD7 20494E20
0BDB 46495253
0BDF 54204E55
0BE3 4D424552
0BE7 3A20
0BE9 00
0BEA 54595045          9      S_NUMB:         DB        'TYPE IN SECOND NUMBER: ',00H
0BEE 20494E20
0BF2 5345434F
0BF6 4E44204E
0BFA 554D4245
0BFE 523A20
0C01 00
0C02 54484520         10      SUM:                DB        'THE SUM IS ',00H
0C06 53554D20
0C0A 495320
0C0D 00
0000                  11      ORG 0
                      12      ;  The following instructions prepare the serial port to recieve and
                      13      ;    send data at 110 baud
                      14      ;    Hardware assumptions:
                      15      ;                              Proper power supply
                      16      ;                              Logic to modify TTL signal to current loop
                      17      ;                              Necessary cabeling to connect terminal
                      18      ;
0000 758920           19      MOV TMOD,#00100000B     ; SET TIMER MODE TO AUTO-RELOAD
0003 758D03           20      MOV TH1,#(-253)         ; SET TIMER FOR 110 BAUD
                      21                              ; 110 = 10.7MHz/12*16*2*253
                      22                              ; 110 = desired baud rate
                      23                              ; 10.7MHz = external clock rate
                      24                              ; -253 = timer preset value
                      25                              ; 12*16*2 = conversion constant
0006 7598DA           26      MOV SCON,#11011010B     ; PREPARE SERIAL PORT FOR OPERATION
0009 D28E             27      SETB TR1                ; START CLOCK
                      28      START:
```

Figure 1-3. MCS-51™ Example Program Listing

937-3

```
MCS-51 MACRO ASSEMBLER                                                    PAGE    2

LOC  OBJ            LINE     SOURCE

                      29     ; This part of program starts communication and gets first number
000B 900BB8           30              MOV DPTR,#TYPO
000E 12006C           31              CALL PUT_STRING              ;  OUTPUT HOW TO RECOVER FROM TYPO
0011 120061           32              CALL PUT_CRLF
0014 900BD3           33                   MOV DPTR,#F_NUMB        ; GET ADDRESS OF DB STRING
0017 12006C           34                   CALL PUT_STRING         ; OUTPUT STRING FOR FIRST NUMBER
001A 120061           35                   CALL PUT_CRLF           ; OUTPUT CARRIAGE RETURN LINE FEED
001D 7832             36                   MOV R0,#FIRST_NUMBER
001F 120077           37                   CALL GET_NUMB           ; GET FIRST NUMBER
0022 120061           38                   CALL PUT_CRLF
                      39     ; THIS SECTION GETS SECOND NUMBER FROM CONSOLE
0025 900BEA           40                   MOV DPTR,#S_NUMB        ; OUTPUT STRING FOR SECOND NUMBER
0028 12006C           41                   CALL PUT_STRING
002B 120061           42                   CALL PUT_CRLF
002E 783C             43                   MOV R0,#SECOND_NUMBER
0030 120077           44                   CALL GET_NUMB           ; GET SECOND NUMBER
0033 120061           45                   CALL PUT_CRLF
                      46     ; THIS SECTION OF CODE CONVERTS ASCII NUMBERS TO BINARY
0036 7932             47                   MOV R1,#FIRST_NUMBER
0038 1200BF           48                   CALL ASCBIN             ; TRANSLATE ASCII STRING TO BINARY NUMBER
003B 793C             49                   MOV R1,#SECOND_NUMBER
003D 1200BF           50                   CALL ASCBIN             ; TRANSLATE SECOND ASCII STRING
0040 E532             51                   MOV A,FIRST_NUMBER      ; GET RESULT OF FIRST TRANSLATION
                      52     ; ADD NUMBERS AND CHANGE BINARY SUM TO ASCII STRING
0042 253C             53                   ADD A,SECOND_NUMBER     ; ADD BOTH NUMBERS
0044 F532             54                   MOV FIRST_NUMBER,A
0046 7932             55                   MOV R1,#FIRST_NUMBER     ; PREPARE FOR RETRANSLATION
0048 120099           56                   CALL BINASC             ; TRANSLATE BINARY NUMBER TO ASCII
004B 900C02           57                   MOV DPTR,#SUM
                      58     ; OUTPUT SUM STRING AND CONVERTED ASCII SUM
004E 12006C           59                   CALL PUT_STRING         ; OUTPUT SUM STRING
0051 AA04             60                   MOV R2,4
0053 7932             61                   MOV R1,#FIRST_NUMBER
0055 E7               62     PUT_SUM:      MOV A,@R1
0056 120091           63                   CALL PUT_CHAR
0059 09               64                   INC R1
005A DAF9             65                   DJNZ R2,PUT_SUM
005C 120061           66                   CALL PUT_CRLF
005F 80AA             67                   JMP START
                      68     ;     BEGIN SERVICE ROUTINES
                      69     ;     THIS LISTING DOES NOT DISPLAY I/O SERVICE
                      70     ;     SEE APPENDIX FOR COMPLETE LISTING
  000D                71     CR              EQU             0DH
  000A                72     LF              EQU             0AH
                      73 +1  $  NOLIST
  REG               121     NUMB_PTR    EQU         R1
  0030              122     ZERO        EQU         ('0')
  002D              123     MINUS       EQU         ('-')
  002B              124     PLUS        EQU         ('+')
                    125 +1  $    EJECT
```

Figure 1-3. MCS-51™ Example Program Listing (Cont'd.)

937-4

```
MCS-51 MACRO ASSEMBLER                                                    PAGE    3

LOC  OBJ            LINE     SOURCE

                     126     ;*************************************************************************
                     127     ;
                     128     ; This routine converts a binary 2's complement number to a 4 character
                     129     ; ASCII string.
                     130     ; INPUT:
                     131     ;       The binary value must be located in memory at the address contained
                     132     ;       in register 1.
                     133     ; OUTPUT:
                     134     ;        The 4 charater result is placed in memory with the first character
                     135     ;        at the address contained in register 1.
                     136     ; NOTES:
                     137     ;        The contents of register A and B will be destroyed.
                     138     ;        The contents of the memory location initially addressed by
                     139     ;        register 1 will be replaced with the first charater in the
                     140     ;        resulting character string.
                     141     ;*************************************************************************
                     142     BINASC:
  00E7               143     SIGN    BIT         ACC.7
0099 E7              144             MOV         A,@NUMB_PTR             ; Get number
009A 772B            145             MOV         @NUMB_PTR,#PLUS
009C 30E704          146             JNB         SIGN,VAL                ; Test bit 7 for] sign
009F 772D            147                     MOV         @NUMB_PTR,#MINUS  ;  Insert negative sign
                     148     ;  Change negative number to positive.
00A1 14              149             DEC         A
00A2 F4              150             CPL         A
                     151     ;  Now work on first digit
                     152     VAL:
00A3 09              153                     INC         NUMB_PTR
                     154     ; Factor out first digit
00A4 75F064          155                     MOV         B,#100
00A7 84              156                     DIV AB
00A8 2430            157                     ADD A,#ZERO
00AA F7              158                     MOV @NUMB_PTR,A
00AB 09              159                     INC NUMB_PTR
                     160     ; Factor out second digit from remainder
00AC E5F0            161                     MOV A,B
00AE 75F00A          162                     MOV B,#10
00B1 84              163                     DIV AB
00B2 2430            164                     ADD A,#ZERO
00B4 F7              165                     MOV @NUMB_PTR,A
00B5 09              166                     INC NUMB_PTR
                     167     ;  Get third and final digit
00B6 E5F0            168                     MOV A,B
00B8 2430            169                     ADD A,#ZERO
00BA F7              170                     MOV @NUMB_PTR,A
                     171     ;  restore NUMB_PTR
00BB 19              172                     DEC NUMB_PTR
00BC 19              173                     DEC NUMB_PTR
00BD 19              174                     DEC NUMB_PTR
00BE 22              175                     RET
                     176 +1  $   EJECT
```

Figure 1-3. MCS-51™ Example Program Listing (Cont'd.) 937-5

```
MCS-51 MACRO ASSEMBLER                                                    PAGE    4

LOC  OBJ            LINE     SOURCE

                     177     ; ************************************************************************
                     178     ; This routine takes a 4 character string located in memory and converts
                     179     ; it to a binary 2's complement number.
                     180     ; The number must begin with a sign character ('+' or '-'), and be
                     181     ; between -128 and +127.
                     182     ; INPUT:
                     183     ;       Four ASCII characters a sign character followed a '0' or a '1'
                     184     ;       and the last 2 characters can be any digit.
                     185     ;       The contents of register 1 must point to the sign character.
                     186     ; OUTPUT:
                     187     ;        A binary 2's complement representation of the value of the
                     188     ;        character string.
                     189     ;        Register 1 contains the address of the binary value.
                     190     ; NOTES:
                     191     ;        The contents of the memory location initially
                     192     ;        addressed by register 1 is destroyed.
                     193     ;        The contents of registers 7 and B and the accumulator
                     194     ;        are destroyed.
                     195     ; ************************************************************************
  REG                196     TEMP            EQU  R7
                     197     ASCBIN:
                     198     ; Go right to number compute sign at end
00BF 09              199                     INC NUMB_PTR
00C0 E7              200                     MOV A,@NUMB_PTR
00C1 9430            201                     SUBB A,#ZERO
00C3 75F064          202                     MOV B,#100
00C6 A4              203                     MUL AB
                     204     ;  Store first digit's value and go to next digit
00C7 FF              205                     MOV TEMP,A
00C8 09              206                     INC NUMB_PTR
00C9 E7              207                     MOV A,@NUMB_PTR
00CA 9430            208                     SUBB A,#ZERO
00CC 75F00A          209                     MOV B,#10
00CF A4              210                     MUL AB
                     211     ;  Add first digit value to secon store and go to third digit
00D0 2F              212                     ADD A,TEMP
00D1 FF              213                     MOV TEMP,A
00D2 09              214                     INC NUMB_PTR
00D3 E7              215                     MOV A,@NUMB_PTR
00D4 9430            216                     SUBB A,#ZERO
                     217     ;  Add third digit value to total.  Store and go back for sign
00D6 2F              218                     ADD A,TEMP
00D7 FF              219                     MOV TEMP,A
00D8 19              220                     DEC NUMB_PTR
00D9 19              221                     DEC NUMB_PTR
00DA 19              222                     DEC NUMB_PTR
00DB E7              223                     MOV A,@NUMB_PTR
                     224     ; Test for sign value
00DC B42D04          225                     CJNE A,#MINUS,POS
00DF EF              226                     MOV A,TEMP
00E0 F4              227                     CPL A
00E1 04              228                     INC A
00E2 FF              229                     MOV TEMP,A
00E3 EF              230     POS:                MOV A,TEMP
                     231     ; store result and return


MCS-51 MACRO ASSEMBLER                                                    PAGE    5

LOC  OBJ            LINE     SOURCE

00E4 F7              232                     MOV @NUMB_PTR,A
00E5 22              233                     RET
                     234             END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 1-3. MCS-51™ Example Program Listing (Cont'd.)

937-6

## Hardware Overview

The 8051 is a high density microcomputer on a chip that is upwardly compatible with the 8048. Its major features are:

- resident 4K bytes of ROM or EPROM program memory (no program memory resident on 8051), expandable to 64K bytes
- resident 128 bytes of RAM memory, which includes 4 banks of 8 general-purpose registers and a stack for subroutine and interrupt routine calls
- 64K bytes of external RAM address space
- 16-bit Program Counter giving direct access to 64K bytes of program memory
- 8-bit stack pointer that can be set to any address in on-chip RAM
- two programmable 16-bit timers/counters
- programmable full duplex serial I/O port
- four 8-bit bidirectional parallel I/O ports
- timer and I/O interrupts with 2 levels of priority
- 111 instructions with 51 basic functions (including memory to memory move)
- Boolean functions with 128 software flags and 12-bit address instructions
- one microsecond instruction cycle time
- Arithmetic and Logic Unit that includes add, subtract, multiply, and divide arithmetic functions, as well as *and*, *or*, *exlusive or*, and *complement* logical functions

Figure 1-4 is a block diagram of the MCS-51 processor. It shows the data paths and principal functional units accessible to the programmer.

## Memory Segments

The MCS-51 processors have four separate address segments or spaces:

- Code address space—4K on-chip, and up to 60K may be added off-chip by user.
- Internal Data address space—128 bytes RAM and 128 byte hardware register address space (only 20 addresses used).
- External data address space—up to 64K of off-chip memory added by user.
- Bit address space—shares locations accessible in the data address space.

The code address space, internal data address space, and external data address space correspond to three physically distinct memories, and are addressed by different machine instructions. This is an important distinction that is a key to understanding how to program the 8051.

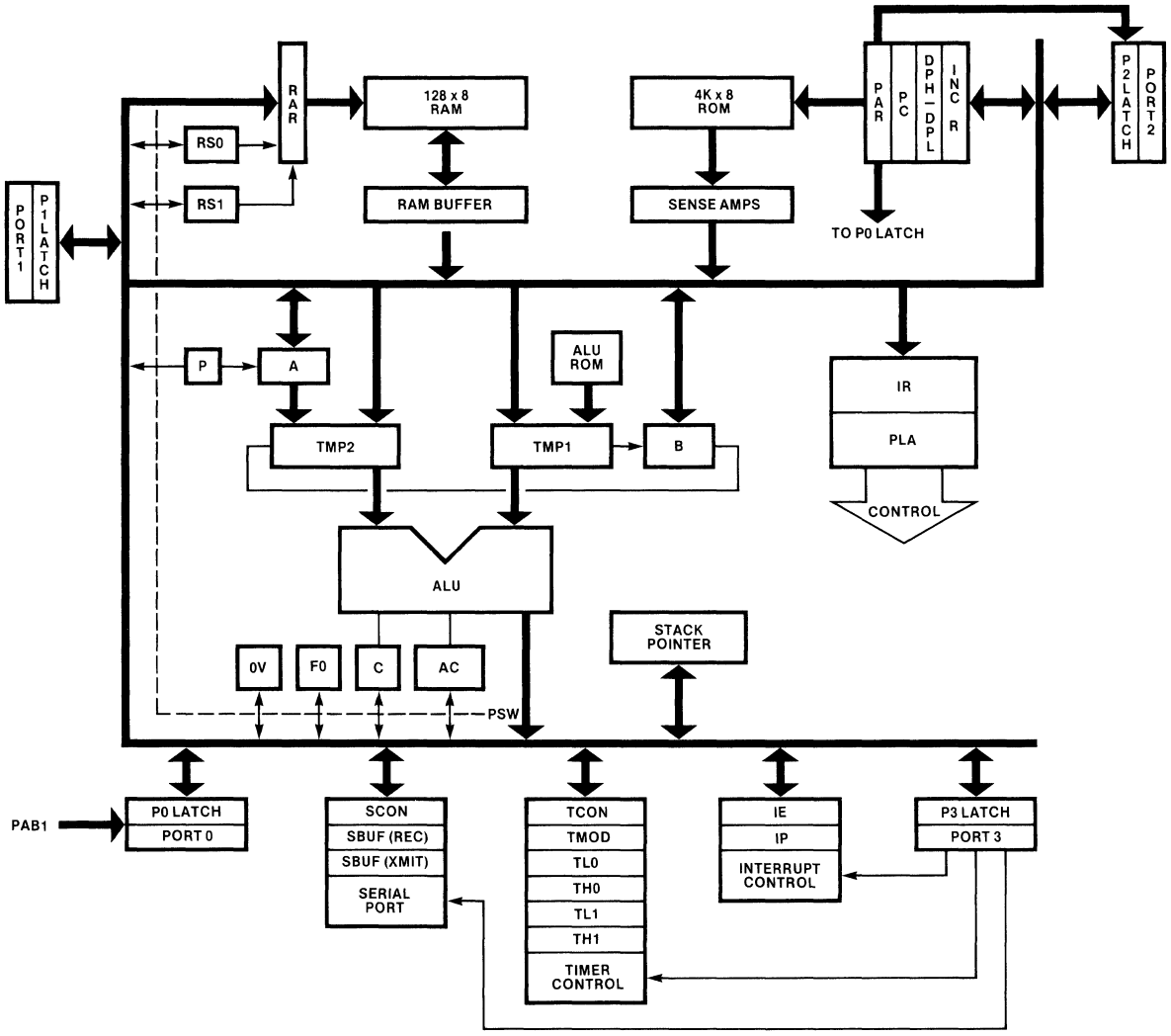



Figure 1-4. MCS-51™ Block Diagram

937-7

To help you keep these segments and their addresses separate, ASM51 assigns a segment type attribute to symbols containing addresses in the various address spaces.

CSEG—Code address space
DSEG—Data address space (on-chip)
XSEG—External data address space (off-chip)
BSEG—Bit address space

When you specify in an operand to an instruction a symbol with the wrong attribute, ASM51 generates an error message to warn you of the inconsistency. Chapters 2 and 3 show what segment type attribute is expected in each instruction, and Chapter 4 describes how to define a symbol with any of the segment type attributes.

Figure 1-5 shows the code address space (usually ROM), and external data address space (usually RAM). Off-chip ROM and RAM can be tailored to use all or part of the address space to better reflect the needs of your application. You can access ROM and off-chip RAM with the MOVC and MOVX instructions respectively.

To the programmer, there is no distinction between on-chip and off-chip code. The 16-bit program counter freely addresses on- and off-chip code memory with no change in instruction fetch time.

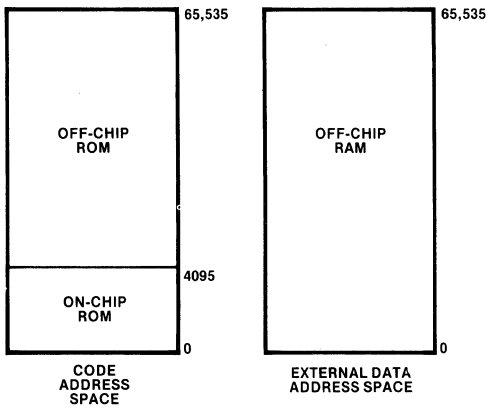



Figure 1-5. MCS-51™ Code Address Space and External Data Address Space

Figure 1-6 shows the data address space containing the bit address space. The data address space contains 4 banks of general-purpose registers in the low 32 bytes (0 - 1FH). In addition to the 128 bytes of RAM, the 8051's hardware registers are mapped to data addresses. The addresses from 128 to 255 are reserved for these registers, but not all of those addresses have hardware registers mapped to them. These reserved addresses are unusable.

The data segment contains two areas that are bit addressable. One is located in RAM in the 16 bytes above the register banks (20H - 2FH). The other bit address area is in the address space reserved for hardware registers. The contents of both bit address areas can be accessed as part of a byte with a data address or as a single bit with a bit address.

A complete description of how to specify all of the addresses and how to access the various address spaces in your program is given in Chapter 2—Operands and Assembly-Time Expressions, and Chapter 3—The Instruction Set.

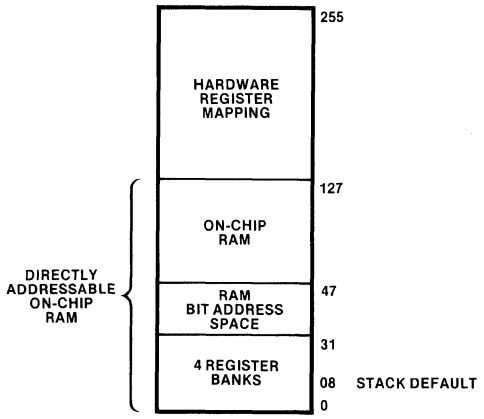



Figure 1-6. MCS-51™Data Address Space and Bit Address Space 937-9

## Data Units

The 8051 manipulates data in four basic units—bits, nibbles (4 bits), bytes, and addresses (16 bits).

The most common data unit used is a byte; all of the internal data paths are 8 bits wide, and the code memory, the data memory, and the external data memory store and return data in byte units. However, there are many instructions that test and manipulate single bits. Bits can be set, cleared, complemented, logically combined with the carry flag, and tested for jumps. The nibble (BCD packed digit) is less commonly used in the 8051, but BCD arithmetic can be performed without conversion to binary representation.

Instructions that use 16-bit addresses deal with the Data Pointer (DPTR a 16-bit register) and the Program Counter (jumps and subroutine calls). However, with the add with carry (ADDC) and subtract with borrow (SUBB) instructions, software implementation of 16-bit arithmetic is relatively easy.

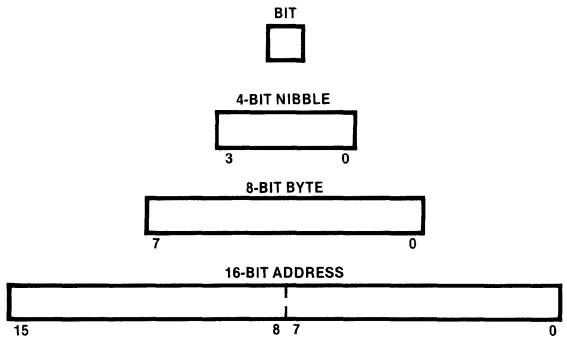



Figure 1-7. MCS-51™Data Units 937-10

## Arithmetic and Logic Functions

The arithmetic functions include:

- ADD—signed 2's complement addition
- ADDC—signed 2's complement addition
- SUBB—signed 2's complement subtraction with borrow
- DA—adjust 2 packed BCD digits after addition
- MUL—unsigned integer multiplication
- DIV—unsigned integer division
- INC—signed 2's complement increment
- DEC—signed 2's complement decrement

The accumulator receives the result of ADD, ADDC, SUBB, and DA functions. The accumulator receives partial result from MUL and DIV. DEC and INC can be applied to all byte operands, including the accumulator.

The logical functions include:

- ANL—logical *and* on each bit between 2 bytes or 2 bits
- CPL—logical *complement* of each bit within a byte or a single bit
- ORL—logical *or* on each bit between 2 bytes or 2 bits
- XRL—logical *exclusive or* on each bit between 2 bytes

The accumulator usually receives the result of the byte functions, and the carry flag usually receives the result of the bit functions, but some instructions place the result in a specified byte or bit in the data address space.

The instructions shown above are described in Chapter 3.

## General-Purpose Registers

The 8051 has four banks of 8 general-purpose registers. They are located in the first 32 bytes of on-chip RAM (00H - 1FH). You can access the registers of the currently active bank through their special assembler symbols (R0, R1, R2, R3, R4, R5, R6, and R7). To change the active bank you modify the register bank select bits (RS0 and RS1) contained in the program status word (PSW, described in table 1-3). Table 1-1 below shows the bank selected for all values of RS0 and RS1.

**Table 1-1. Register Bank Selection**

| RS1 | RS0 | Bank | Memory Locations |
|---|---|---|---|
| 0 | 0 | 0 | 00H—07H |
| 0 | 1 | 1 | 08H—0FH |
| 1 | 0 | 2 | 10H—17H |
| 1 | 1 | 3 | 18H—1FH |

## The Stack

The stack is located in on-chip RAM. It is a last-in-first-out storage mechanism used to hold the Program Counter during interrupts and subroutine calls. You can also use it to store and return data with the POP and PUSH instructions. The Stack Pointer contains the address of the top of the stack.

The Stack Pointer (SP) is an 8-bit register that may contain any address in on-chip RAM memory. However, on the 8051 it should never exceed 127. If it does, all data pushed is lost. *A pop, when the SP is greater than 127, returns invalid data.*

The SP always contains the address of the last byte pushed on the stack. On power-up (Reset) it is set to 07H, so the first byte pushed on the stack after reset will be at location 08H. This location is compatible with the 8048's stack. Most programs developed for the 8051 will reset the bottom of the stack by changing the contents of the SP before using the stack, because 08H-1FH is the area reserved for several of the 8051's general-purpose-register banks. The following instruction causes the next byte pushed on the stack to be placed at location 100.

```
MOV SP,#99          ; Initialize stack to start at location 100
                    ; The hardware increments the SP
                    ; BEFORE a push
```

## Symbolically Addressable Hardware Registers

Each programmable register is accessible through a numeric data address, but the assembler supplies a predefined symbol that should be used instead of the register's numeric address. Table 1-2 identifies each hardware register, its numeric address, and its predefined symbol.

The predefined symbols given in table 1-2 stand for the on-chip data addresses of the hardware registers. In many cases the only access to these registers is through these data addresses. However, some of the registers have an identity both as a special assembler symbol and as a data address symbol (e.g., both "ACC" and "A" stand for the accumulator), but even though these symbols may be semantically the same, they are syntactically different. For example,

```
ADD A,#27
```

is a valid instruction to add 27 to the contents of the accumulator, but

```
ADD ACC,#27
```

is invalid and will cause an error, because there is no form of ADD taking a data address as the destination (ACC specifies a data address). The differences become even more subtle in some assembly instructions where both symbols are valid but assemble into different machine instructions:

```
MOV A,#27           ; assembles into a 2 byte instruction
MOV ACC,#27         ; assembles into a 3 byte instruction
```

Chapter 2 describes the syntax for all instruction operands, and Chapter 3 describes the operands expected in each instruction.

Because the hardware registers are mapped to data addresses, there is no need for special I/O or control instructions. For example,

```
MOV A,P2
```

moves a copy of the input data at Port 2 to the accumulator. To output a character on the Serial I/O port (after preparing SCON), simply move the character into the Serial port buffer (SBUF):

```
MOV SBUF,#'?'
```

**Table 1-2. Symbolically Addressable Hardware Registers**

| Predefined Symbol | Data Address | Meaning |
|---|---|---|
| ACC | E0H | ACCUMULATOR (Data address of A) |
| B | F0H | MULTIPLICATION REGISTER |
| DPH | 83H | DATA POINTER (high byte) |
| DPL | 82H | DATA POINTER (low byte) |
| IE | A8H | INTERRUPT ENABLE |
| IP | B8H | INTERRUPT PRIORITY |
| P0 | 80H | PORT 0 |
| P1 | 90H | PORT 1 |
| P2 | A0H | PORT 2 |
| P3 | B0H | PORT 3 |
| PSW | D0H | PROGRAM STATUS WORD |
| SBUF | 99H | SERIAL PORT BUFFER |
| SCON | 98H | SERIAL PORT CONTROLLER |
| SP | 81H | STACK POINTER |
| TCON | 88H | TIMER CONTROL |
| TH0 | 8CH | TIMER 0 (high byte) |
| TH1 | 8DH | TIMER 1 (high byte) |
| TL0 | 8AH | TIMER 0 (low byte) |
| TL1 | 8BH | TIMER 1 (low byte) |
| TMOD | 89H | TIMER MODE |

## Bit Addressing

Many of the hardware control registers are also bit addressable. The flags contained in them can be accessed with a bit address as well as through the byte address shown above. One way to do this is through the bit selector (.). For example to access the 0 bit in the accumulator, you might specify ACC.0.

Bit addressing allows the same simplicity in testing and modifying control and status flags as was shown above with addressable registers. For example to start Timer 0 running, set the run flag to 1 via its bit address (SETB TCON.4).

Throughout the remainder of this chapter, several programmable features including predefined bit addresses of status and control flags, discussed. To use these features, you simply modify the corresponding address as if it were a RAM location.

## The Program Status Word

The Program Status Word (PSW) contains several status bits that reflect the state of the 8051. Figure 1-8 shows the predefined bit address symbol, the bit position, and meaning of each bit in the PSW.

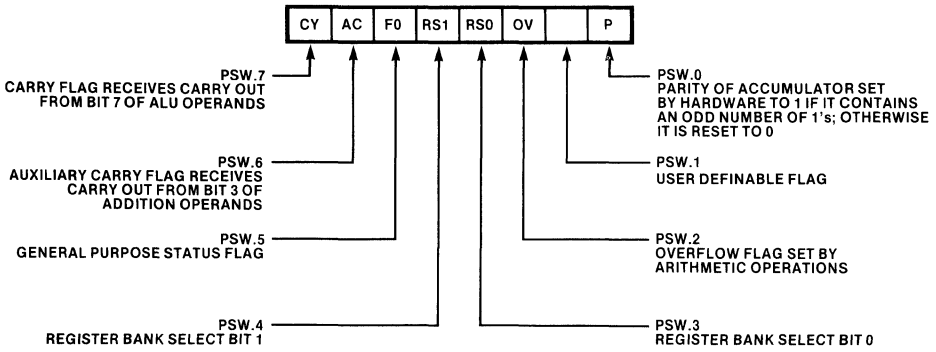



Figure 1-8. Bit Descriptions of Program Status Word 937-11

## Timer and Counter

The 8051 has two independently programmable timers. They feature a 16-bit counter and are controlled by 2 registers, timer mode (TMOD) and timer control (TCON). Figure 1-9 shows the predefined bit address symbols, the positions and meanings of the bits in TCON. (For a complete description of the timer see the *MCS-51 User's Manual.*)

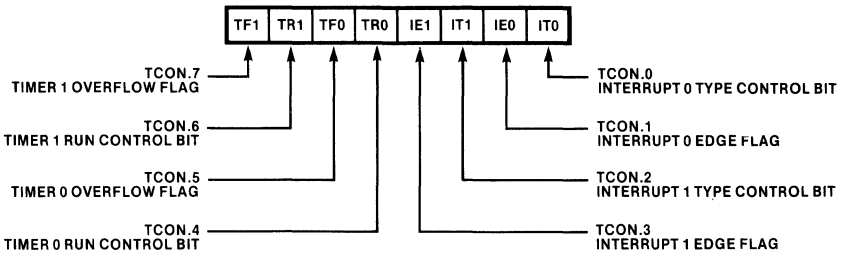



Figure 1-9. Bit Descriptions of TCON 937-12

## I/O Ports

The 8051 has 4 8-bit I/O ports; each bit in the ports corresponds to a specific pin on the chip. All four ports are buffered by a port latch, and they are addressable through a data address (as a byte) or 8 bit addresses (as a set of bits). As noted earlier, this removes the need for special I/O instructions. The numeric data address and the predefined symbol for each port is shown below:

| Port | Predefined Symbol | Data Address |
|---|---|---|
| 0 | P0 | 80H |
| 1 | P1 | 90H |
| 2 | P2 | A0H |
| 3 | P3 | B0H |

Port 0 and Port 2 are used for external program and external data addressing. Port 0 also receives the input data from off-chip addressing. If off-chip memory is not implemented, then ports 0 and 2 are bidirectional I/O ports. Port 1 is a general purpose bidirectional I/O port.

Port 3 contains the external interrupt pins, the external timer, the external data memory read and write enables, and the serial I/O port transmit and receive pins. The bits that correspond to these pins are individually addressable via predefined bit address symbols. Figure 1-10 shows the meaning of each bit, its position in Port 3, and its predefined bit address symbol.

If the external interrupts, external data addressing, and serial I/O features of the 8051 are not used Port 3 can function as a bidirectional I/O port.

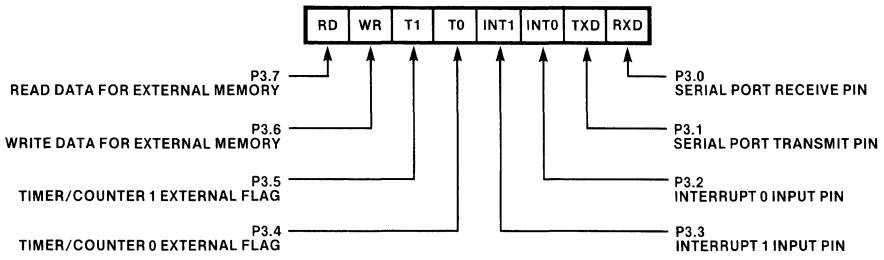



Figure 1-10. Bit Descriptions for Port 3 937-13

## Serial I/O Port

The serial I/O port permits I/O expansion using UART protocols. The serial I/O port is controlled by Serial Port Controller (SCON), a register that is both bit addressable and byte addressable. Figure 1-11 shows the predefined bit address symbols, positions and meanings of the bits in SCON. For complete details of Serial I/O port control see the *MCS-51 User's Manual.*

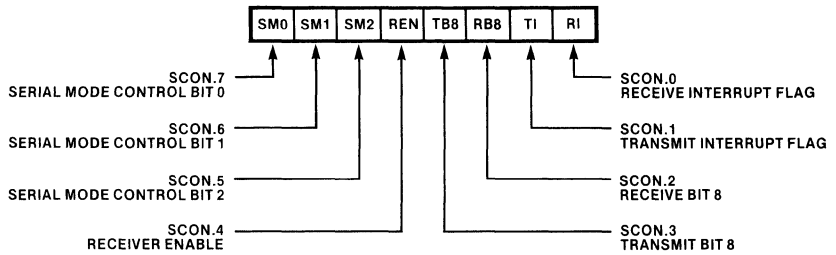



Figure 1-11. Bit Descriptions for Serial Port Control 937-14

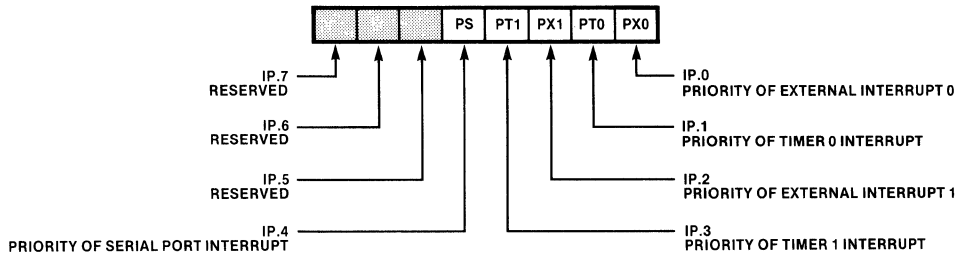
## Interrupt Control

There are two registers that control timer and I/O interrupts and priorities. They are IE (Interrupt Enable) and IP (Interrupt Priority). When the interrupt enable bit for a device is 1, it can interrupt the processor. The 8051 does not respond to an interrupt until the instruction being executed has been completed (this can be as long as 4 cycles).

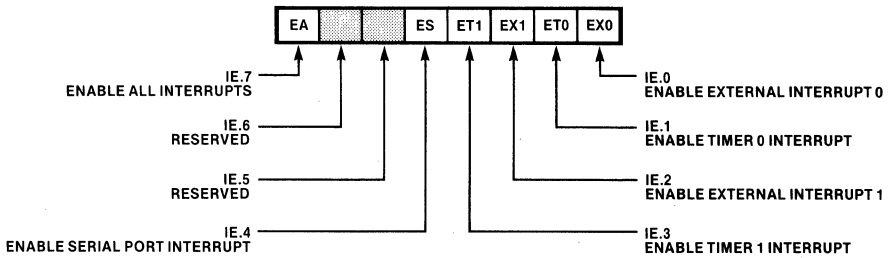
When it does respond, the 8051's hardware disables interrupts of the same or lesser priority and makes a subroutine call to the code location designated for the interrupting device. Typically, that location contains a jump to a longer service routine. The instruction RETI must be used to return from a service routine, in order to reenable interrupts. The reserved locations, the predefined labels, and the associated interrupt devices are listed below. These labels may be used to aid the placement of I/O routines in code memory.

| Predefined Label | Location | Interrupting Device |
|---|---|---|
| RESET | 00H | Power on Reset (First instruction executed on power up.) |
| EXTI0 | 03H | External interrupt 0 |
| TIMER0 | 0BH | Timer 0 |
| EXTI1 | 13H | External interrupt 1 |
| TIMER1 | 1BH | Timer 1 |
| SINT | 23H | Serial I/O port |

The 8051 has two levels of interrupt priority (0 and 1). Figure 1-12 shows the predefined bit address symbol, the position and the device associated with each bit contained in IE and IP. A level 1 priority device can interrupt a level 0 service routine, but a level 0 interrupt will not affect a level 1 service routine. Interrupts on the same level are disabled.




Interrupt Priority




Interrupt Enable

937-15

**Figure 1-12. Bit Descriptions for Interrupt Enable and Interrupt Priority**

## Reset

On reset all of the registers in the 8051 assume an initial value. Table 1-3 shows these initial values. This will always be the state of the chip when your code begins execution. You can use these initial values or reinitialize them as necessary in your program.

**Table 1-3. State of the 8051 after Power-up**

| Register | Value |
|---|---|
| Accumulator | 00H |
| Multiplication Register | 00H |
| Data Pointer | 0000H |
| Interrupt Enable | 00H |
| Interrupt Priority | 00H |
| Port 0 | 0FFH |
| Port 1 | 0FFH |
| Port 2 | 0FFH |
| Port 3 | 0FFH |
| Program Counter | 0000H |
| Program Status Word | 00H |
| Serial Port Control | 00H |
| Serial I/O Buffer | undefined |
| Stack Pointer | 07H |
| Timer Control | 00H |
| Timer Mode | 00H |
| Timer 0 Counter | 0000H |
| Timer 1 Counter | 0000H |

NOTE

The PC is always set to 0 on reset, thus the first instruction executed in a program is at ROM location 0. The contents of RAM memory is unpredictable at reset.

# CHAPTER 2
# OPERANDS AND ASSEMBLY-TIME EXPRESSIONS

This chapter discusses the operand types used by ASM51. It describes their use and some of the ways you can specify them in your program. The latter part of the chapter deals with expressing numbers and using assembly-time expressions.

## Operands

The general form of all instruction lines is as follows:

[*Label*:] *Mnemonic* [*Operand*] [,*Operand*] [,*Operand*] [;*Comment*]

The number of operands and the type of operands expected depend entirely on the mnemonic. Operands serve to further define the operation implied by a mnemonic, and they identify the parts of the machine affected by the instruction.

All operands fall into one of six classes:

- Special Assembler symbols
- Indirect Addresses
- Immediate Data
- Data Addresses (on-chip)
- Bit Addresses
- Code Addresses

A special assembler symbol is a specific reserved word required as the operand in an instruction. Indirect addresses use the contents of a register to specify a data address.

The remaining operand types (immediate data, data addresses, bit addresses, and code addresses) are numeric operands. They may be specified symbolically, but they must evaluate to a number. The range permitted for a numeric operand depends on the instruction with which it is used. The operand can be made up of predefined or user defined symbols, numbers, and Assembly-Time operators.

As described in Chapter 1, the data address space, the bit address space, the external data address space, and the code address space are separate and distinct address areas on the 8051. In many cases the same numeric value is a valid address for all four address spaces (segments). To help avoid logic errors in your program, ASM51 performs type checking in instruction operands (and arguments to assembler directives), that address these segments. The segment type expected in each of these operands is described below. Chapter 4 describes how to define symbols with different segment types.

## Special Assembler Symbols

The assembler reserves several symbols to designate specific registers as operands. A special assembler symbol is encoded in the opcode byte, as opposed to a data address which is encoded in an operand byte. Table 2-1 lists these symbols and describes the hardware register each represents.

**Table 2-1. Special Assembler Symbols**

| Special Symbol | Meaning |
|---|---|
| A | Accumulator |
| R0,R1,R2 R3,R4,R5, R6,R7 | Stands for the 8 general registers in the currently active bank (4 register banks available) |
| DPTR | Data pointer: a 16-bit register used for indexing tables in code address space and external address space |
| PC | Program Counter: a 16-bit register that contains the address of the next instruction to be executed. |
| C | Carry flag receives ALU carry out and borrow from bit 7 of the operands |
| AB | Accumulator/B Register pair used in MUL and DIV instructions |

If the definition of an instruction requires one of these symbols, only that special symbol can be used. However, you can, with the SET and EQU directives, define other symbols to stand for the accumulator (A) or the working registers (R0, R1,...R7). Symbols so defined may not be forward referenced in an instruction operand. You cannot use a special assembler symbol for any other purpose in an instruction operand or directive argument. Several examples of instructions that use these symbols are shown below.

```
INC DPTR        ; Increment the entire 16-bit contents of the Data Pointer by 1

SETB C          ; Set the Carry flag to 1

MOV R6,A        ; Move the contents of the accumulator to working register 6

JMP @A+PC       ; Add the contents of the accumulator to the contents of the
                ;program counter and jump to that address

MUL AB          ; Multiply accumulator by register B and place result in A and B
```

## Indirect Addressing

An indirect address operand identifies a register that contains the address of a memory location to be used in the operation. The actual location affected will depend on the contents of the register when the instruction is executed. In most instructions indirect addresses affect on-chip memory. However, the MOVC and MOVX instructions use an indirect address operand to address code memory and external data memory respectively.

In on-chip indirect addressing either register 0 or register 1 of the active register bank can be specified as an indirect address operand. The commercial at sign (@) followed by the register's special symbol (R0 or R1), or a symbol defined to stand for the register's special symbol, indicates indirect addressing. On the 8051 the address contained in the specified indirect address registers must be between 0 and 127. So, you cannot access hardware registers through indirect addressing. *If an indirect address register contains a value greater than 127 when it is used for on-chip addressing, the program continues with no indication of the error*. If it is a source operand, a byte containing undefined data is returned. If it is a destination operand, the data is lost.

The following examples show several uses of indirect addressing.

```
ADD A,@R1       ; Add the contents of the on-chip RAM location addressed by
                ; register 1 to the accumulator

INC @R0         ; Increment the contents of the on-chip RAM location
                ; addressed by register 0

MOVX @DPTR,A    ; Move the contents of the accumulator to the off-chip memory
                ; location addressed by the data pointer
```

## Immediate Data

An immediate data operand is a numeric expression that, when assembled, is encoded as part of the machine instruction. The pound sign (#) immediately before the expression indicates that is is an immediate data operand. The numeric expression must be a valid assembly-time expression.

The assembler represents all numeric expressions in 16 bits, and converts to the appropriate form for instruction encoding. (Appendix H shows how ASM51 represents positive numbers internally. The 2's complement notation used for negative numbers is shown below.) Most instructions require the value of the immediate data to fit into a byte. The low order byte of the assembler's 16-bit internal representation is used. This permits a numeric expression range of values from −256 to +255. These values all have a homogeneous high order byte (i.e., all ones or all zeros) when represented in 16 bits. The immediate data operands that accept a 16-bit value can use any value representable by the assembler. Immediate data operands do not require any specific segment type.

The following examples show several ways of specifying the immediate data operand.

```
MOV A,#0E0H         ; Place the hex constant E0 in the accumulator

MOV DPTR,#0A14FH    ; This is the only instruction that uses a 16-bit immediate data
                    ; operand

ANL A,#128          ; Mask all but the high order bit of the accumulator
                    ; 128(base 10) = 1000 0000(base 2)
```

## Data Addressing

The memory address operand is a numeric expression that evaluates to one of the 128 on-chip memory addresses or one of the hardware register addresses. The low-order byte of the assembler's 16-bit internal representation is used. This permits a range from −256 to +255, but since the 8-bit value encoded in the instruction has no

sign to the 8051, it is easier to think of its value as only positive (0 to 255). (Appendix H shows how ASM51 represents positive numbers internally. The 2's complement notation used for negative numbers is shown below.) Instructions that use the data address operand require that the symbol or expression specified be of segment type DSEG or have no segment type at all. (Symbols are discussed below under Assembly-Time Expression Evaluation.)

The data addresses from 0 to 127 access the 8051's on-chip RAM space, while the addresses from 128 to 255 access the hardware registers. Not all of the addresses in the hardware register space are defined. The illustration below (figure 2-1) shows the meaningful addresses and their predefined data address names.

If you read from a reserved address, undefined data will be returned. If you write to a reserved address, the data will be lost. Using these peculiarities in your program may result in incompatability with future versions of the chip.

The following examples show several ways of specifying data addresses.

```
MOV P1,A        ; Move the contents of the accumulator to the predefined data
                ; address 90(base 16) Port 1

ORL A,20*5      ; Logical OR of accumulator with location 100(base 10) uses an
                ; assembly-time operator multiply

INC 32          ; Increment location 32 (base 10) in memory
```

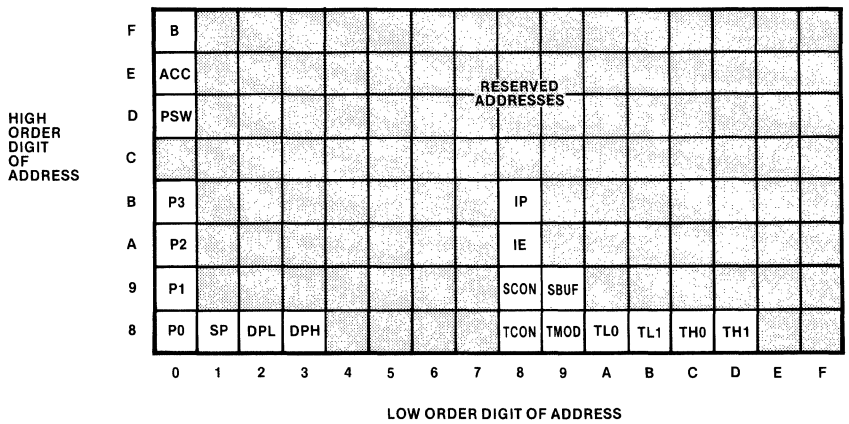



Figure 2-1. Hardware Register Address Area

937-16

## Bit Addressing

A bit address is a numeric value encoded in the instruction by the assembler. There are two ways to represent a bit address in an operand.

1. You can specify the byte that contains the bit with a data address, and single out the particular bit in that byte with the bit selector (“.” period) followed by a bit identifier (0-7). For example, 40.5, 21H.0 and ACC.7 are valid uses of the bit selector. You can use an assembly-time expression to express the byte address or the bit identifier. The assembler will translate this to the correct numeric value. However, only certain bytes in the on-chip address space are bit addressable. (See figure 2-2.)

2. You can do the translation youself, by using a numeric expression that evaluates to a bit address. Like memory addresses, the low order byte of the assembler's 16-bit internal representation is used. This permits a numeric expression range from −256 to +255, but since the 8-bit value encoded in the instruction has no sign, it is easier to think of its value as only positive (0 to 255). (Appendix H shows how ASM51 represents positive numbers internally. The 2's complement notation used for negative numbers is shown below.)

Instructions that use the bit address operand require that symbols or expressions used be of segment type BSEG, or have no segment type at all. (Symbols are discussed below under Assembly-Time Expression Evaluation.) Figures 2-2a and 2-2b show the bits assigned to each numeric bit address.

The following examples show several ways of specifying the same bit.

```
SETB TR1          ; Set the predefined bit address TR1 (Timer 1 Run Flag)

SETB 88H.6        ; Set bit 6 of location 88H (Timer 1 Run Flag)

SETB 8EH          ; Set the bit address 8E(base 16) (Timer 1 run flag)
```

As with data addresses there are several bit addresses that are predefined as symbols that you can use in an operand. Table 2-2 shows these predefined bit addresses. You can also define your own bit address symbols with the BIT directive described in Chapter 4 Assembler Directives.

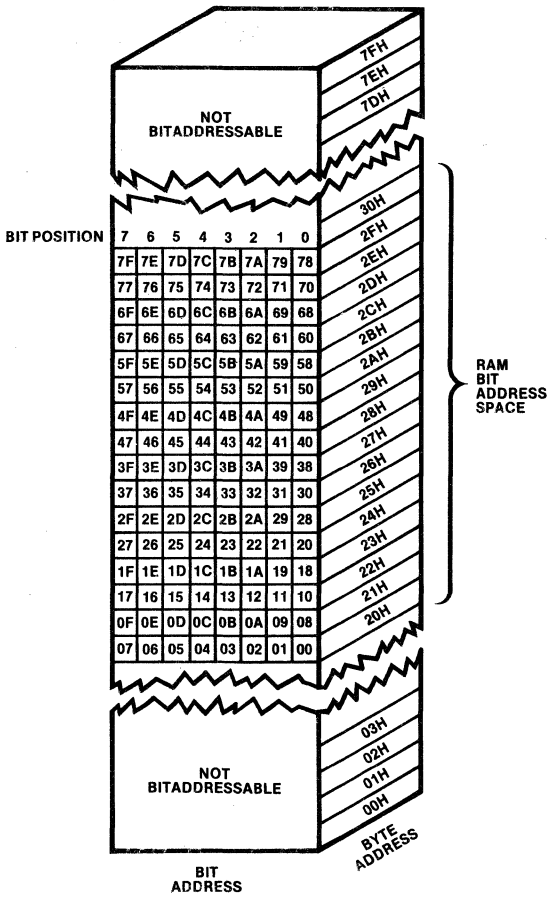
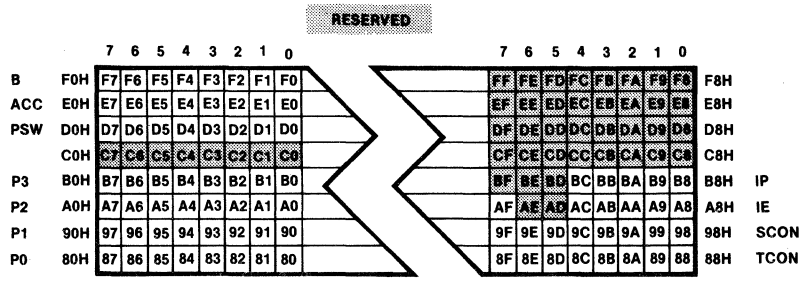



Figure 2-2a. Bit Addressable Bytes in RAM




937-17

Figure 2-2b. Bit Addressable Bytes in Hardware Register Address Area

Table 2-2. Predefined Bit Addresses

| Symbol | Bit Position | Bit Address | Meaning |
|---|---|---|---|
| CY | PSW.7 | D7H | Carry Flag |
| AC | PSW.6 | D6H | Auxiliary Carry Flag |
| F0 | PSW.5 | D5H | Flag 0 |
| RS1 | PSW.4 | D4H | Register Bank Select Bit 1 |
| RS0 | PSW.3 | D3H | Register Bank Select Bit 0 |
| OV | PSW.2 | D2H | Overflow Flag |
| P | PSW.0 | D0H | Parity Flag |
| TF1 | TCON.7 | 8FH | Timer 1 Overflow Flag |
| TR1 | TCON.6 | 8EH | Timer 1 Run Control Bit |
| TF0 | TCON.5 | 8DH | Timer 0 Overflow Flag |
| TR0 | TCON.4 | 8CH | Timer 0 Run Control Bit |
| IE1 | TCON.3 | 8BH | Interrupt 1 Edge Flag |
| IT1 | TCON.2 | 8AH | Interrupt 1 Type Control Bit |
| IE0 | TCON.1 | 89H | Interrupt 0 Edge Flag |
| IT0 | TCON.0 | 88H | Interrupt 0 Type Control Bit |
| SM0 | SCON.7 | 9FH | Serial Mode Control Bit 0 |
| SM1 | SCON.6 | 9EH | Serial Mode Control Bit 1 |
| SM2 | SCON.5 | 9DH | Serial Mode Control Bit 2 |
| REN | SCON.4 | 9CH | Receiver Enable |
| TB8 | SCON.3 | 9BH | Transmit Bit 8 |
| RB8 | SCON.2 | 9AH | Receive Bit 8 |
| TI | SCON.1 | 99H | Transmit Interrupt Flag |
| RI | SCON.0 | 98H | Receive Interrupt Flag |
| EA | IE.7 | AFH | Enable All Interrupts |
| ES | IE.4 | ACH | Enable Serial Port Interrupt |
| ET1 | IE.3 | ABH | Enable Timer 1 Interrupt |
| EX1 | IE.2 | AAH | Enable External Interrupt 1 |
| ET0 | IE.1 | A9H | Enable Timer 0 Interrupt |
| EX0 | IE.0 | A8H | Enable External Interrupt 0 |
| RD | P3.7 | B7H | Read Data for External Memory |
| WR | P3.6 | B6H | Write Data for External Memory |
| T1 | P3.5 | B5H | Timer/Counter 1 External Flag |
| T0 | P3.4 | B4H | Timer/Counter 0 External Flag |
| INT1 | P3.3 | B3H | Interrupt 1 Input Pin |
| INT0 | P3.2 | B2H | Interrupt 0 Input Pin |
| TXD | P3.1 | B1H | Serial Port Transmit Pin |
| RXD | P3.0 | B0H | Serial Port Receive Pin |
| PS | IP.4 | BCH | Priority of Serial Port Interrupt |
| PT1 | IP.3 | BBH | Priority of Timer 1 Interrupt |
| PX1 | IP.2 | BAH | Priority of External Interrupt 1 |
| PT0 | IP.1 | B9H | Priority of Timer 0 |
| PX0 | IP.0 | B8H | Priority of External Interrupt 0 |

## Code Addressing

There are three types of instructions that require a code address in their operands. They are relative jumps, absolute 2K page jumps or calls, and long jumps or calls. The difference between each type is the range of values that the code address operand may assume. All three expect an expression which evaluates to a code address (a numeric expression between 0 and 65535) but if you specify a relative jump or a 2K page jump, only a small subset of all possible code addresses is valid. Instructions that use the code address operand require that the symbol or expression specified be of segment type CSEG or have no segment type at all. (Symbols and labels are discussed below under Assembly-Time Expression Evaluation.)

### Relative Jump (SJMP and Conditional Jumps)

The code address to a relative jump must be close to the relative jump instruction itself. The range is from −128 to +127 bytes from the first byte of the instruction that *follows* the relative jump.

The assembler takes the specified code address and computes a relative offset that is encoded as an 8-bit 2's complement number. That offset is added to the contents of the program counter (PC) when the jump is made, but, since the PC is always incremented to the next instruction before the jump is executed, the range is computed from the succeeding instruction.

When you use a relative jump in your code, you must use a numeric expression that evaluates to the absolute code address of the jump destination. The assembler does all the offset computation. If the address is out of range, the assembler will issue an error message.

### 2K Page Jumps and Calls (AJMP and ACALL)

The code address operand to a 2K page jump or call is a numeric expression that is evaluated and then encoded in the instruction by the assembler. The low order 11 bits of the destination address are placed in the opcode byte and one operand byte. When the jump or call is executed, the 11-bit page address replaces the low order 11 bits of the program counter. This permits a range of 2048 bytes, or anywhere within the current 2K byte page.

If the page jump or call is the last instruction on a 2K page, the high order bits of the PC change when incremented to address the next instruction; thus, the jump will be made within that new 2K page.

### Long Jumps and Calls (LJMP and LCALL)

The code address operand to a long jump or call is a numeric expression that will be evaluated and then encoded as a 16-bit value in the instruction by the assembler. All 16 bits of the program counter are replaced by this new value when the jump or call is executed. Since 16 bits are used, any value representable by the assembler will be acceptable (0-65535).

The following examples show each type of instruction that calls for a code address.

```
SJMP LABEL      ; Jump to LABEL (relative offset LABEL must be within −128
                ; and +127 of instruction that follows SJMP)

ACALL SORT      ; Call subroutine labeled SORT (SORT must be an address to
                ; within the current 2K page)

LJMP EXIT       ; Long jump; the label or symbol EXIT must be defined
                ; somewhere in the program
```

### Generic Jump and Call (JMP or CALL)

The assembler provides two instruction mnemonics that do not represent a specific opcode. They are JMP and CALL. JMP may assemble to any of the unconditional jump instructions (SJMP, AJMP, or LJMP). CALL may assemble to ACALL or LCALL. These generic mnemonics will always evaluate to an instruction that will reach the specified code address operand.

This is an effective tool to use during program development, since sections of code change drastically in size with each development cycle. (See Chapter 3 for a complete description of both generic jumps.)

## Assembly-Time Expression Evaluation

An expression is a combination of numbers, character strings, symbols, and operators that evaluate to a single 16-digit binary number. Except for some directives all expressions can use forward references (symbols that have not been defined at that point in the program) and any of the assembly-time operators.

### Specifying Numbers

You can specify numbers in hexadecimal (base 16), decimal (base 10), octal (base 8), and binary (base 2). The default representation, used when no base designation is given, is decimal. Table 2-3 below shows the digits of each numbering system and the base designation character for each system.

**Table 2-3. Assembly Language Number Representation**

| Number System | Base Designator | Digits in Order of Value |
|---|---|---|
| Binary | B | 0, 1 |
| Octal | O or Q | 0, 1, 2, 3, 4, 5, 6, 7 |
| Decimal | D or (nothing) | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| Hexadecimal | H | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F |

The only limitation to the range of numbers is that they must be representable within 16 binary digits.

Table 2-4 gives several examples of number representation in each of the number systems.

**Table 2-4. Examples of Number Representation**

| base 16 | base 10 | base 8 | base 2 |
|---|---|---|---|
| 50H | 80 | 120O | 01010000B |
| 0ACH* | 172D | 254Q | 10101100B |
| 01H | 1 | 1Q | 1B |
| 10H | 16D | 20Q | 10000B |

* A hexadecimal number must start with a decimal digit; 0 is used here.

## ASM51 Number Representation

Internally, ASM51 represents all numeric values with 16 bits. When ASM51 encounters a number in an expression, it immediately converts it to 16-bit binary representation. Numbers cannot be greater than 65,535. Appendix H describes conversion of positive numbers to binary representation.

Negative numbers (specified by the unary operator "−") are represented in 2's complement notation. There are two steps to converting a positive binary number to a negative (2's complement) number.

```
0000 0000 0010 0000B  = 20H
1111 1111 1101 1111   = Not 20H          1. Complement each bit in the number.
1111 1111 1110 0000   = (Not 20H) +1     2. Add 1 to the complement.
1111 1111 1110 0000B  = −20H
```

To convert back simply perform the same two steps again.

Although 2's complement notation is used, ASM51 does not convert these numbers for comparisons. Therefore, large positive numbers have the same representation as small negative numbers (e.g., −1 = 65535). Table 2-5 shows number interpretation at assembly-time and at program execution-time.

**Table 2-5. Interpretations of Number Representation**

| Number Characteristic | Assembly-Time Expression Evaluation | Program Execution Arithmetic |
|---|---|---|
| Base Representation | Binary, Octal, Decimal, or Hexadecimal | Binary, Octal, Decimal, or Hexadecimal |
| Range | 0-65,535 | User Controlled |
| Evaluates To: | 16 Bits | User Interpretation |
| Internal Notation | Two's Complement | Two's Complement |
| Signed/Unsigned Arithmetic | Unsigned | User Interpretation |

## Character Strings in Expressions

The MCS-51 assembler allows you to use ASCII characters in expressions. Each character stands for a byte containing that character's ASCII code. (Appendix H contains a table of the ASCII character codes.) That byte can then be treated as a numeric value in an expression. In general two characters or less are permitted in a string (only the DB directive accepts character strings longer than two characters). In a one character string the high byte is filled with 0's. With a two character string, the first character's ASCII value is placed in the high order byte, and the second character's value is placed in the low order byte.

All character strings must be surrounded by the single quote character ('). To incorporate the single quote character into the string, place two single quote characters side-by-side in a string. For example, 'z''' is a string of two characters: a lower case "Z" and the single quote character.

The ability to use character strings in an expression offers many possibilities to enhance the readability of your code. Below, there are two examples of how character strings can be used in expressions.

```
TEST: CJNE A,#'X',SKIP  ; If A contains 'X' then fall through
JMP FOUND               ; Otherwise, jump to skip and
SKIP: MOV A,@R1         ; Move next character into accumulator
DEC R1                  ; Change R1 to point to next character
DJNZ R2,TEST            ; JUMP to TEST if there are still more
                        ; characters to test

MOV A,SBUF              ; Move character in serial port buffer
                        ; to accumulator
SUBB A,#'0'             ; Subtract '0' from character just read
                        ; this returns binary value of the digit
```

**NOTE**

A corollary of this notation for character strings is the null string—two single quotes surrounding no characters (side-by-side). When the null character string is used in an expression it evaluates to 0, but when used as an item in the expression list of a DB directive it will evaluate to nothing and will not initiate memory. (See Chapter 4 for an example.)

## Use of Symbols

The assembler has several kinds of symbols available to the programmer. They may stand for code addresses, bit addresses, data addresses, constants, or registers. They allow a programmer to enhance the readability of his code. All symbols are assigned two attributes when they are defined in the program: a numeric value, and a segment type.

Once you have defined a symbol anywhere in your program (some expressions require that no forward references be used), you can use it in any numeric operand in the same way that you would use a constant, providing you respect segment type conventions. The segment type required for each numeric operand is described above. The creation of user-defined symbols is completely described in Chapter 4 "Assembler Directives."

Besides the user-defined symbols there are several predefined bit addresses and data addresses available for commonly used hardware registers and flags. Table 2-6 shows all of the predefined memory address symbols and the values they represent. The bit address symbols have been listed earlier in this chapter. (See Table 2-2.)

Table 2-6. Predefined Data Addresses

| Symbol | Hexadecimal Address | Meaning |
|---|---|---|
| ACC | E0 | Accumulator |
| B | F0 | Multiplication Register |
| DPH | 83 | Data Pointer (high byte) |
| DPL | 82 | Data Pointer (low byte) |
| IE | A8 | Interrupt Enable |
| IP | B8 | Interrupt Priority |
| P0 | 80 | Port 0 |
| P1 | 90 | Port 1 |
| P2 | A0 | Port 2 |
| P3 | B0 | Port 3 |
| PSW | D0 | Program Status Word |
| SBUF | 99 | Serial Port Buffer |
| SCON | 98 | Serial Port Controller |
| SP | 81 | Stack Pointer |
| TCON | 88 | Timer Control |
| TH0 | 8C | Timer 0 (high byte) |
| TH1 | 8D | Timer 1 (high byte) |
| TL0 | 8A | Timer 0 (low byte) |
| TL1 | 8B | Timer 1 (low byte) |
| TMOD | 89 | Timer Mode |

Remember that these symbols evaluate to a data address and cannot be used in instructions that call for a special assembler symbol.

```
ADD A,#5        ; This is a valid instruction A is the special
                ; assembler symbol required for this operand
ADD ACC,#5      ; This is an invalid instruction and will generate
                ; an error message. ACC is an address and not
                ; the special symbol required for the instruction
```

There is an additional symbol that may be used in any numeric operand, the location counter ($). When you are using the location counter in an instruction's operand, it will stand for the address of the first byte of the instruction currently being encoded. You can find a complete description of how to use and manipulate the location counter in Chapter 4, "Assembler Directives."

## Using Operators in Expressions

There are four classes of assembly-time operators: arithmetic, logical, special, and relational. All of them return a 16-bit value. Instruction operands that require only 8 bits will receive the low order byte of the expression (unless the operator HIGH is used). The distinction between each class of operators is loosely defined. Since they may be used in the same expression, they work on the same type of data, and they return the same type of data.

## Arithmetic Operators

Table 2-7 contains a list of all the arithmetic operators:

**Table 2-7. Arithmetic Assembly-Time Operators**

| Operator | Meaning |
|---|---|
| + | Unary plus or add |
| – | Unary minus or subtract |
| * | Multiplication |
| / | Integer division (discard remainder) |
| MOD | Modular division (discard quotient) |

The following examples all produce the same bit pattern in the low order byte (0011 0101B):

```
+53
27+26
-203
65-12
2*25+3      multiplication is always executed before the addition
160/3
153 MOD 100
```

Note that the MOD operator must be separated from its operands by at least one space or tab.

## Logical Operators

Table 2-8 contains a list of all logical operators. The logical operators perform their operation on each bit of their operands.

**Table 2-8. Logical Assembly-Time Operators**

| Operator | Meaning |
|---|---|
| OR | Full 16-bit OR |
| AND | Full 16-bit AND |
| XOR | Full 16-bit exclusive OR |
| NOT | Full 16-bit complement |

The following examples all produce the same 8-bit pattern in the low order byte (0011 0101B):

```
00010001B OR  00110100B
01110101B AND 10110111B
11000011B XOR 11110110B
          NOT 11001010B
```

Note that all logical operators must be separated from their operand by at least one space or tab.

## Special Assembler Operators

Table 2-9 contains a list of all special operators:

**Table 2-9. Special Assembly-Time Operators**

| Operator | Meaning |
|---|---|
| SHR | 16-bit shift right |
| SHL | 16-bit shift left |
| HIGH | Select the high order byte of operand |
| LOW | Select the low order byte of operand |
| ( ) | Evaluate the contents of the parenthesis first |

The following examples all produce the same 8-bit pattern in the low order byte (0011 0101B):

| | |
|---|---|
| 01AFH SHR 3 | Bits are shifted out the right end and 0 is shifted into the left |
| HIGH (1135H SHL 8) | Parenthesis is required since HIGH has a greater precedence than SHL Bits are shiftedout the left and 0 is shifted in the right |
| LOW 1135H | Without using the LOW operator the high order byte would have caused an error in an 8-bit operand. |

Note SHR, SHL, HIGH and LOW must be separated from their operands by at least one space or tab.

## Relational Operators

The relational operators differ from all of the other operators in that the result of a relational operation will always be either 0 (False) or 0FFFFH(True). Table 2-10 contains a list of all the relational operators:

**Table 2-10. Relational Assembly-Time Operators**

| Operator | Meaning |
|---|---|
| EQ = | Equal |
| NE <> | Not equal |
| LT < | Less than |
| LE <= | Less than or equal to |
| GT > | Greater than |
| GE >= | Greater than or equal to |

The following examples all will return TRUE (0FFFFH):

```
27H EQ 39D
27H<>27D
33 LT 34
7>5
16 GE 10H
```

Note that the two-letter (mnemonic) form of the relational operator must be separated from their operands by at least one space or tab; the symbolic form does not.

## Operator Precedence

Every operator is given a precedence in order to define which operator is evaluated first in an expression. For example the expression 3*5+1 could be interpreted as 16 or 18 depending on whether the + or the * is evaluated first. The following list shows the precedence of the operators in descending order.

- Parenthesized expression ( )
- HIGH, LOW
- *, /, MOD, SHL, SHR
- +, – *unary and binary forms*
- EQ, NE, LT, LE, GT, GE, =, <>, <, <=, >, >=
- NOT
- AND
- OR, XOR

All operators on the same precedence level are evaluated from left to right in the expression.

## Segment Typing in Expressions

Most expressions formed with assembly-time operators do not have a segment type, but some operations allow the expression to assume the segment type of a symbol used in the expression. The rules for expressions having a segment type are listed below.

1. Expressions that contain only constants or symbols without a segment type have no segment type.
2. The result of operations performed by the following operators will have no segment type.

| | | | | | |
|---|---|---|---|---|---|
| HIGH | LOW | NOT | OR | XOR | AND |
| EQ | NE | GT | GE | LE | LT |
| * | / | MOD | SHR | SHL | |

- Operations performed with +, − and ( ) can have a segment type. Table 2-11 shows what conditions are necessary for the result to have a segment type.

**Table 2-11. Segment Typing in Operations**

| Operand | Operator | Operand | Segment Type |
|---|---|---|---|
| — | ( ) | Value (S) | Segment type maintained |
| — | + | Value (S) | Segment type maintained |
| — | − | Value (S) | Segment type maintained |
| Value (N) | + | Value (S) | Segment type maintained |
| Value (S) | + | Value (N) | Segment type maintained |
| Value (S) | + | Value (S) | Segment type lost |
| Value (N) | − | Value (S) | Segment type maintained |
| Value (S) | − | Value (N) | Segment type maintained |
| Value (S) | − | Value (S) | Segment type lost |

(S) is a numeric value (symbol or the result of an operation) with a segment type attribute

(N) is a numeric value with no segment type attribute

NOTE

The table above shows the result of simple binary and unary operations. These results are also valid for more complex expressions. Each operation is evaluated according to precedence and the intermediate result will have a numeric value and sometimes a segment type.

# CHAPTER 3
# INSTRUCTION SET

This chapter contains complete documentation for all of the 8051 instructions. The instructions are listed in alphabetical order by mnemonic and operands.

## Introduction

This chapter is designed to be used as a reference. Each instruction is documented using the same basic format. The action performed by an instruction is defined in three ways. First, the operation is given in a short notation; the symbols used and their meanings are listed in the table below. The operation is then defined in a few sentences in the description section. Finally, an example is given showing all of the registers affected and their contents before and after the instruction.

**NOTE**

The only exception is that the program counter (PC) is not always shown. All instructions increment the PC by the number of bytes in the instruction. The "Example:" entry for most instructions do not show this increment by the PC. Only those instructions that directly affect the PC (e.g., JMP, ACALL, or RET) show the contents of the PC before and after execution.

The list of notes that appears at the bottom of some instructions refer to side-effects (flags set and cleared and limitations of operands). The numbers refer to the notes tabulated on page 3-143/3-144. You can unfold that page for easier reference while you are studying the instruction set.

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

With one exception, the operands to 3 byte instructions are encoded in the same order as they appear in the source. Only the "Move Memory to Memory" instruction is encoded with the second operand preceding the first.

The illustration below (figure 3-1) describes the meaning of each section of the instruction documentation.

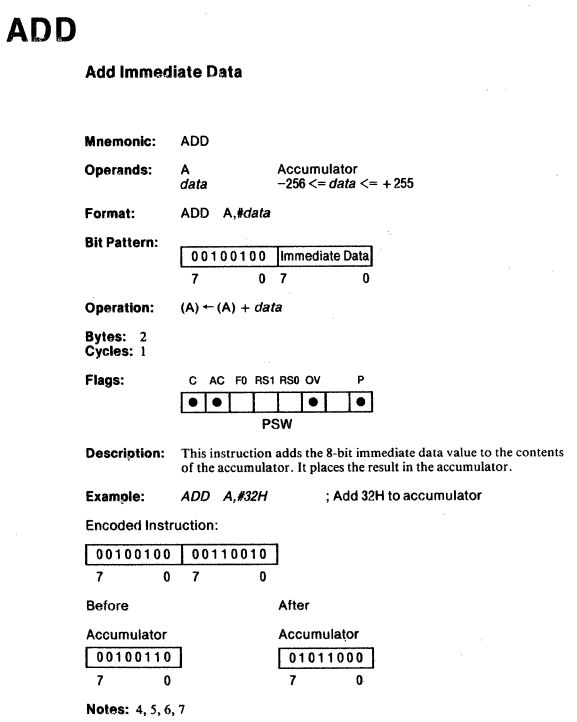



**Figure 3-1. Format For Instruction Definitions** 937-18

**Mnemonic:** shows opcode mnemonic. It is shown in upper case, but upper or lower case characters are permitted.

**Operands:** indicates range and type of operands permitted.

**Format:** shows the format of the instruction, including the order of operands on the source line.

**Bit Pattern:** indicates bit pattern in opcode and position of operands when encoded. Letters in the opcode's bit pattern vary with operand specified.

**Operation:** symbolically defines the operation performed by the instruction. The symbols used in this entry are defined in table 3-1.

**Bytes and Cycles:** shows the number of bytes of code and the number of machine cycles used by the instruction.

**Flags:** indicates any status flag that may be changed during the execution of the instruction.

**Description:** is a brief prose description of the operation performed by the instruction.

**Example:** shows an example instruction as it would appear in the source. It also shows the bit pattern of the encoded instruction, and the contents of all registers affected by the instruction, immediately before and after the instruction is executed.

The PC is incremented by all instructions, but only instructions that affect the PC as part of their operation show its contents in the example.

**Notes:** indicates the notes on page 3-143/3-144 that pertain to the instruction.

**Table 3-1. Abbreviations and Notations Used**

| | |
|---|---|
| A | Accumulator |
| AB | Register Pair |
| B | Multiplication Register |
| *bit address* | 8051 bit address |
| *page address* | 11-bit code address within 2K page |
| *relative offset* | 8-bit 2's complement offset |
| C | Carry Flag |
| *code address* | Absolute code address |
| *data* | Immediate data |
| *data address* | On-chip 8-bit RAM address |
| DPTR | Data pointer |
| PC | Program Counter |
| R*r* | Register(r=0-7) |
| SP | Stack pointer |
| *high* | High order byte |
| *low* | Low order byte |
| *i-j* | Bits i through j |
| *.n* | Bit n |
| | |
| AND | Logical AND |
| NOT | Logical complement |
| OR | Logical OR |
| XOR | Logical exclusive OR |
| + | Plus |
| – | Minus |
| / | Divide |
| * | Multiply |
| (X) | The contents of X |
| ((X)) | The memory location addressed by (X)<br>(The contents of X) |
| = | Is equal to |
| <> | Is not equal to |
| < | Is less than |
| > | Is greater than |
| ← | Is replaced by |

# ACALL

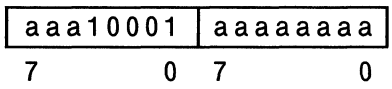
## Absolute Call Within 2K Byte Page

**Mnemonic:** ACALL

**Operands:** *code address*

**Format:** ACALL *code address*

**Bit Pattern:**

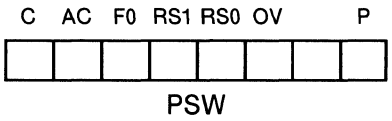



**Operation:**

(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC *low*)
(SP) ← (SP) + 1
((SP)) ← (PC *high*)
(PC) 0-10 ← *page address*

**Bytes:** 2
**Cycles:** 2

**Flags:**




**Description:** This instruction stores the incremented contents of the program counter (the return address) on the stack. The low-order byte of the program counter (PC) is always placed on the stack first. It replaces the low-order 11 bits of the PC with the encoded 11-bit page address. The destination address specified in the source must be within the 2K byte page of the instruction following the ACALL.

The 3 high-order bits of the 11-bit page address form the 3 high-order bits of the opcode. The remaining 8 bits of the address form the second byte of the instruction.

**Example:**

```
        ORG 35H
        ACALL SORT  ; Call SORT (evaluates to page
                    ; address 233H)
          .
          .
          .
        ORG 233H
SORT:   PUSH ACC    ; Store Accumulator
          .
          .
          .
        RET         ; Return from call
```

Encoded Instruction:

| 01010001 | 00110011 |
|---|---|
| 7 0 | 7 0 |

| Before | | After | |
|---|---|---|---|
| Program Counter | | Program Counter | |
| 00000000 | 00110101 | 00000010 | 00110011 |
| 15 8 | 7 0 | 15 8 | 7 0 |
| Stack Pointer | | Stack Pointer | |
| 00100110 | | 00101000 | |
| 7 0 | | 7 0 | |
| (27H) | | (27H) | |
| 00000000 | | 00110111 | |
| 7 0 | | 7 0 | |
| (28H) | | (28H) | |
| 00000000 | | 00000000 | |
| 7 0 | | 7 0 | |

**Notes:** 2, 3

# ADD

## Add Immediate Data

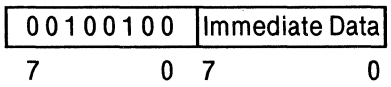
**Mnemonic:** ADD

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | −256 <= *data* <= +255 |

**Format:** ADD A,#*data*

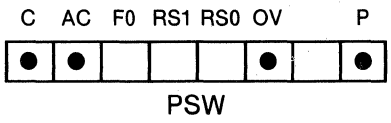
**Bit Pattern:**




**Operation:** (A) ← (A) + *data*

**Bytes:** 2
**Cycles:** 1

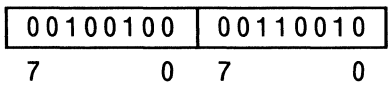
**Flags:**




**Description:** This instruction adds the 8-bit immediate data value to the contents of the accumulator. It places the result in the accumulator.

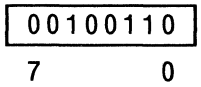
**Example:** *ADD A,#32H* ; Add 32H to accumulator

Encoded Instruction:



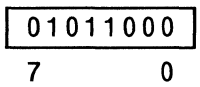

Before

Accumulator




After

Accumulator




**Notes:** 4, 5, 6, 7

## Add Indirect Address

**Mnemonic:** ADD

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** ADD A,@R*r*

**Bit Pattern:**

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|

7 0

**Operation:** (A) ← (A) + ((R*r*))

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | ● | | | | ● | | ● |

PSW

**Description:** This instruction adds the contents of the data memory location addressed by register *r* to the contents of the accumulator. It places the result in the accumulator.

**Example:** *ADD A,@R1* ; Add indirect address to accumulator

Encoded Instruction:

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

7 0

| Before | | After | |
|---|---|---|---|
| Accumulator | | Accumulator | |
| 1 | 0 0 0 0 1 1 0 | 1 | 1 1 0 1 0 0 0 |
| 7 | 0 | 7 | 0 |
| Register 1 | | Register 1 | |
| 0 | 0 0 1 1 1 0 0 | 0 | 0 0 1 1 1 0 0 |
| 7 | 0 | 7 | 0 |
| (1CH) | | (1CH) | |
| 0 | 1 1 0 0 0 1 0 | 0 | 1 1 0 0 0 1 0 |
| 7 | 0 | 7 | 0 |

**Notes:** 5, 6, 7, 15

# ADD

## Add Register

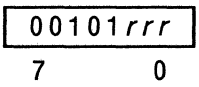
**Mnemonic:** ADD

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 7 |

**Format:** ADD A,R*r*

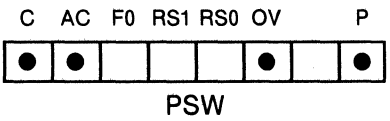
**Bit Pattern:**




**Operation:** (A) ← (A) + (R*r*)

**Bytes:** 1
**Cycles:** 1

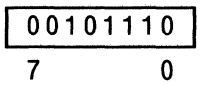
**Flags:**

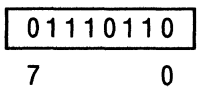
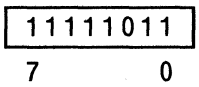
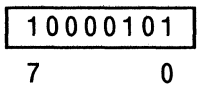
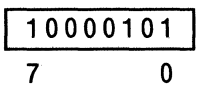



**Description:** This instruction adds the contents of register *r* to the contents of the accumulator. It places the result in the accumulator.

**Example:** *ADD A,R6* ; Add R6 to accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
|  |  |
| Register 6 | Register 6 |
|  |  |






**Notes:** 5, 6, 7

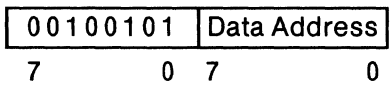
## Add Memory

**Mnemonic:** ADD

**Operands:** A Accumulator
*data address* 0 <= *data address* <= 255

**Format:** ADD A,*data address*

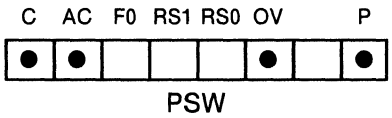
**Bit Pattern:**




**Operation:** (A) ← (A) + (*data address*)

**Bytes:** 2
**Cycles:** 1

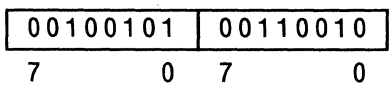
**Flags:**




**Description:** This instruction adds the contents of the specified data address to the contents of the accumulator. It places the result in the accumulator.

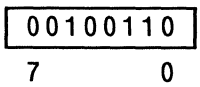
**Example:** *ADD A,32H* ; Add the contents of
; 32H to accumulator

Encoded Instruction:

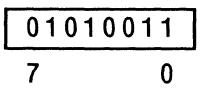



Before

Accumulator

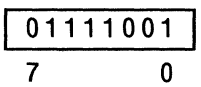



(32H)

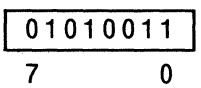



After

Accumulator




(32H)




**Notes:** 5, 6, 7, 8

# ADDC

## Add Carry Plus Immediate Data to Accumulator

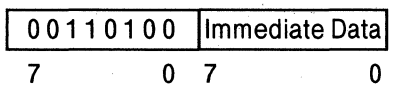
**Mnemonic:** ADDC

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | $-256 <= data <= +255$ |

**Format:** ADDC A,*#data*

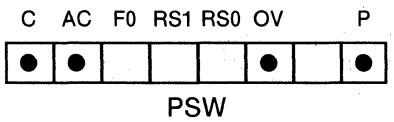
**Bit Pattern:**




**Operation:** (A) ← (A) + (C) + *data*

**Bytes:** 2
**Cycles:** 1

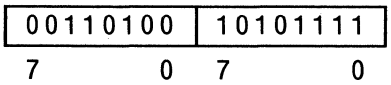
**Flags:**




**Description:** This instruction adds the contents of the carry flag (0 or 1) to the contents of the accumulator. The 8-bit immediate data value is added to that intermediate result, and the carry flag is updated. The accumulator and carry flag reflect the sum of all three values.

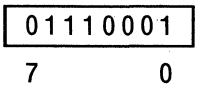
**Example:** *ADDC A,#0AFH* ; Add Carry and 0AFH to accumulator

Encoded Instruction:




Before

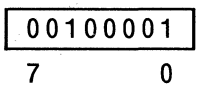
Accumulator




Carry


After

Accumulator




Carry


**Notes:** 4, 5, 6, 7

# ADDC

## Add Carry Plus Indirect Address to Accumulator

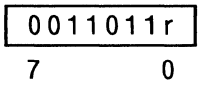
**Mnemonic:** ADDC

**Operands:**

| | |
|---|---|
| A | Accumulator |
| Register | 0 <= *r* <= 1 |

**Format:** ADDC A,@R*r*

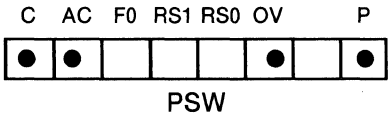
**Bit Pattern:**




**Operation:** (A) ← (A) + (C) + ((R*r*))

**Bytes:** 1
**Cycles:** 1

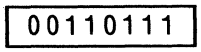
**Flags:**




**Description:** This instruction adds the contents of the carry flag (0 or 1) to the contents of the accumulator. The contents of data memory at the location addressed by register *r* is added to that intermediate result, and the carry flag is updated. The accumulator and carry flag reflect the sum of all three values.

**Example:** *ADDC A,@R1* ; Add carry and indirect address to
; accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 11101000 (7 0) | 00000000 (7 0) |
| Register 1 | Register 1 |
| 01101001 (7 0) | 01101001 (7 0) |
| (69H) | (69H) |
| 00011000 (7 0) | 00011000 (7 0) |
| Carry | Carry |
| 0 | 1 |

**Notes:** 5, 6, 7, 15

# ADDC

## Add Carry Plus Register to Accumulator

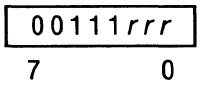
**Mnemonic:** ADDC

**Operands:**

| | |
|---|---|
| A | Accumulator |
| Register | 0 <= *r* <= 7 |

**Format:** ADDC A,R*r*

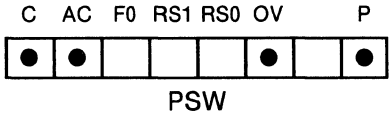
**Bit Pattern:**




**Operation:** (A) ← (A) + (C) + (R*r*)

**Bytes:** 1
**Cycles:** 1

**Flags:**

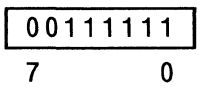



**Description:** This instruction adds the contents of the carry flag (0 or 1) to the contents of the accumulator at bit 0. The contents of register *r* is added to that intermediate result, and the carry flag is updated. The accumulator and carry flag reflect the sum of all three values.

**Example:**

```
ADDC   A,R7          ; Add carry and register 7
                     ; to accumulator
```

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00110000 | 00111011 |
| 7 0 | 7 0 |
| Register 7 | Register 7 |
| 00001010 | 00001010 |
| 7 0 | 7 0 |
| Carry | Carry |
| 1 | 0 |

**Notes:** 5, 6, 7

# ADDC

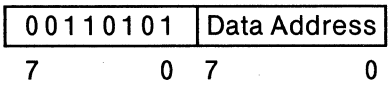
## Add Carry Plus Memory to Accumulator

**Mnemonic:** ADDC

**Operands:** A — Accumulator
*data address* — 0 <= *data address* <= 255

**Format:** ADDC A,*data address*

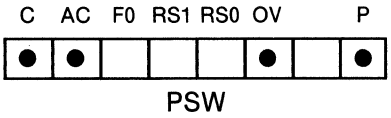
**Bit Pattern:**




**Operation:** (A) ← (A) + (C) + (*data address*)

**Bytes:** 2
**Cycles:** 1

**Flags:**

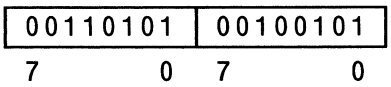



**Description:** This instruction adds the contents of the carry flag (0 or 1) to the contents of the accumulator. The contents of the specified data address is added to that intermediate result, and the carry flag is updated. The accumulator and carry flag reflect the sum of all three values.

**Example:**

```
ADDC   A,25H        ; Add carry and contents of 25H to
                    ; accumulator
```

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 10101110 | 10110101 |
| (25H) | (25H) |
| 00000111 | 00000111 |
| Carry | Carry |
| 0 | 0 |

**Notes:** 5, 6, 7, 8

# AJMP

## Absolute Jump within 2K Byte Page

**Mnemonic:** AJMP

**Operands:** *code address*

**Format:** AJMP *code address*

**Bit Pattern:**

| a a a 0 0 0 0 1 | a a a a a a a a |
|---|---|
| 7 0 | 7 0 |

**Operation:** (PC) ← (PC) + 2
(PC) 0-10 ← *page address*

**Bytes:** 2
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction replaces the low-order 11 bits of the program counter with the encoded 11-bit address. The destination address specified in the source must be within the 2K byte page of the instruction following the AJMP.

The 3 high-order bits of the 11-bit page address form the 3 high-order bits of the opcode. The remaining 8 bits of the address form the second byte of the instruction.

**Example:**

```
        ORG 0E80FH
TOPP:   MOV A,R1
          .
          .
          .
        ORG 0EADCH
        AJMP TOPP     ; Jump backwards to TOPP
                      ; at location 0E80FH
```

Encoded Instruction:

| 0 0 0 0 0 0 0 1 | 0 0 0 0 1 1 1 1 |
|---|---|
| 7 0 | 7 0 |

Before

Program Counter

| 1 1 1 0 1 0 1 0 | 1 1 0 1 1 1 0 0 |
|---|---|
| 15 8 | 7 0 |

After

Program Counter

| 1 1 1 0 1 0 0 0 | 0 0 0 0 1 1 1 1 |
|---|---|
| 15 8 | 7 0 |

**Notes:** None

# ANL

## Logical AND Immediate Data to Accumulator

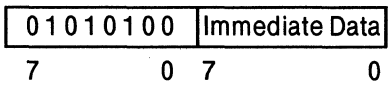
**Mnemonic:** ANL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | −256 <= *data* <= +255 |

**Format:** ANL A,#*data*

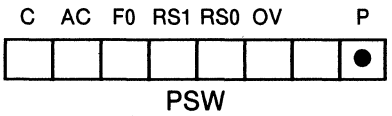
**Bit Pattern:**




**Operation:** (A) ← (A) AND *data*

**Bytes:** 2
**Cycles:** 1

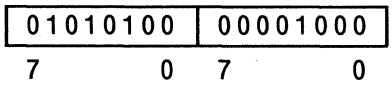
**Flags:**




**Description:** This instruction ANDs the 8-bit immediate data value to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of each operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

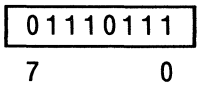
**Example:** *ANL A,#00001000B* ; Mask out all but bit 3

Encoded Instruction:



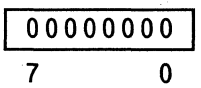

Before

Accumulator




After

Accumulator




**Notes:** 4, 5

## Logical AND Indirect Address to Accumulator

**Mnemonic:** ANL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** ANL A,@R*r*

**Bit Pattern:**

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← (A) AND ((R*r*))

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction ANDs the contents of the memory location addressed by the contents of register *r* to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of each operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

**Example:**

```
ANL   A,@R0        ; AND indirect address with
                   ; accumulator
```

Encoded Instruction:

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | | | | | | | | After | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Accumulator | | | | | | | | Accumulator | | | | | | | |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 | | | | | | | 0 | 7 | | | | | | | 0 |
| Register 0 | | | | | | | | Register 0 | | | | | | | |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 7 | | | | | | | 0 | 7 | | | | | | | 0 |
| (52H) | | | | | | | | (52H) | | | | | | | |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 | | | | | | | 0 | 7 | | | | | | | 0 |

**Notes:** 5, 15

## Logical AND Register to Accumulator

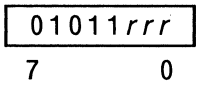
**Mnemonic:** ANL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | 0 <= R*r* <= 7 |

**Format:** ANL A,R*r*

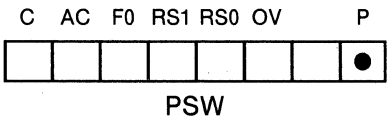
**Bit Pattern:**




**Operation:** (A) ← (A) AND (R*r*)

**Bytes:** 1
**Cycles:** 1

**Flags:**

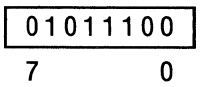



**Description:** This instruction ANDs the contents of register *r* to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of each operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

**Example:**

```
MOV  R4,#10000000B   ; Move mask to R4
ANL  A,R4            ; AND register 4 with accumulator
```

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 10011001 | 10000000 |
| 7 0 | 7 0 |
| Register 4 | Register 4 |
| 10000000 | 10000000 |
| 7 0 | 7 0 |

**Note:** 5

## Logical AND Memory to Accumulator

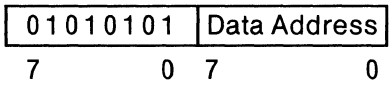
**Mnemonic:** ANL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data address* | 0 <= *data address* <= 255 |

**Format:** ANL A,*data address*

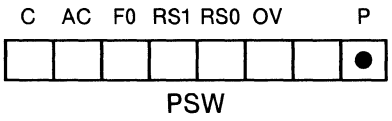
**Bit Pattern:**




**Operation:** (A) ← (A) AND (*data address*)

**Bytes:** 2
**Cycles:** 1

**Flags:**

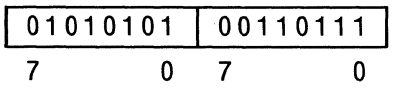



**Description:** This instruction ANDs the contents of the specified data address to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of each operand is also 1; otherwise bit *n* is 0. It places the result in the accumulator.

**Example:**

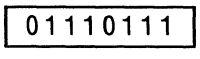
```
ANL   A,37H          ; AND contents of 37H with
                     ; accumulator
```

Encoded Instruction:

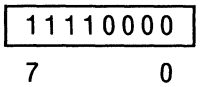



Before

Accumulator

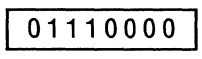



(37H)

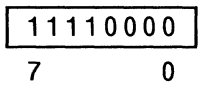



After

Accumulator




(37H)




**Notes:** 5, 8

# ANL

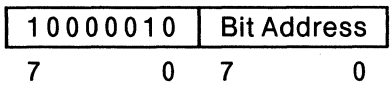
## Logical AND Bit to Carry Flag

**Mnemonic:** ANL

**Operands:** C — Carry Flag
*bit address* — 0 <= *bit address* <= 255

**Format:** ANL C,*bit address*

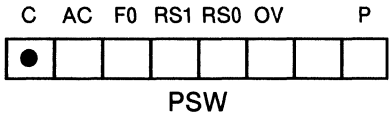
**Bit Pattern:**




**Operation:** (C) ← (C) AND (*bit address*)

**Bytes:** 2
**Cycles:** 1

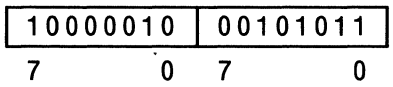
**Flags:**




**Description:** This instruction ANDs the contents of the specified bit address to the contents of the carry flag. If both bits are 1, then the result is 1; otherwise, the result is 0. It places the result in the carry flag.

**Example:** *ANL C,37.3* ; AND bit 3 of byte 37 with Carry

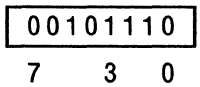
Encoded Instruction:




Before

Carry Flag

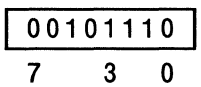

(37)




After

Carry Flag


(37)




**Notes:** None

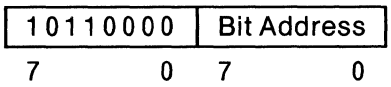
## Logical AND Complement of Bit to Carry Flag

**Mnemonic:** ANL

**Operands:** C Carry Flag
*bit address* 0 <= *bit address* <= 255

**Format:** ANL C,/*bit address*

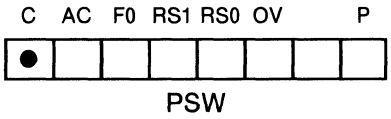
**Bit Pattern:**




**Operation:** (C) ← (C) AND NOT (*bit address*)

**Bytes:** 2
**Cycles:** 2

**Flags:**

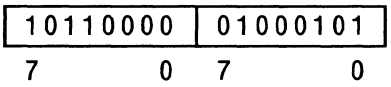



**Description:** This instruction ANDs the complemented contents of the specified bit address to the contents of the carry flag. The result is 1 when the carry flag is 1 and the contents of the specified bit address is 0. It places the result in the carry flag. The contents of the specified bit address does not change.

**Example:**

```
ANL   C,/40.5        ; Complement contents of 40.5
                     ; then AND with Carry
```

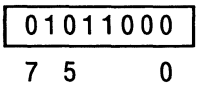
Encoded Instruction:




Before

Carry Flag

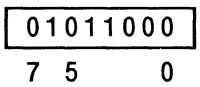

(40)




After

Carry Flag


(40)




**Notes:** None

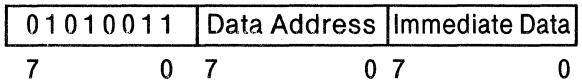
## Logical AND Immediate Data to Memory

**Mnemonic:** ANL

**Operands:** *data address* 0 <=*data address* <= 255
*data* −256 <= *data* <= +255

**Format:** ANL *data address*,#*data*

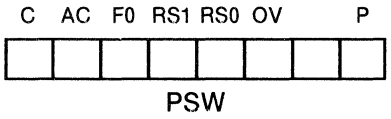
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) AND *data*

**Bytes:** 3
**Cycles:** 2

**Flags:**

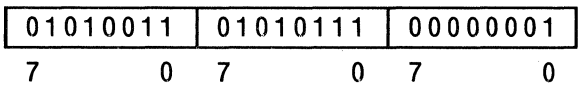



**Description:** This instruction ANDs the 8-bit immediate data value to the contents of the specified data address. Bit *n* of the result is 1 if bit *n* of each operand is also 1; otherwise, bit *n* is 0. It places the result in data memory at the specified address.

**Example:**

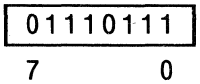
```
MOV 57H,PSW      ; Move PSW to 57H
ANL 57H,#01H     ; Mask out all but parity bit
                 ; to check accumulator parity
```

Encoded Instruction:



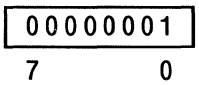

Before

(57H)




After

(57H)




**Notes:** 4, 9

## Logical AND Accumulator to Memory

**Mnemonic:** ANL

**Operands:** *data address* 0 <= *data address* <= 255
A Accumulator

**Format:** ANL *data address*,A

**Bit Pattern:**

| 01010010 | Data Address |
|---|---|
| 7 0 | 7 0 |

**Operation:** (*data address*) ← (*data address*) AND A

**Bytes:** 2
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction ANDs the contents of the accumulator to the contents of the specified data address. Bit *n* of the result is 1 if bit *n* of each operand is also 1; otherwise, bit *n* is 0. It places the result in data memory at the specified address.

**Example:**
```
MOV A,#10000001B    ; Load mask into accumulator
ANL 10H,A           ; Mask out all but bits 0 and 7
```

Encoded Instruction:

| 01010010 | 00010000 |
|---|---|
| 7 0 | 7 0 |

| Before | | After | |
|---|---|---|---|
| Accumulator | | Accumulator | |
| 10000001 | | 10000001 | |
| 7 0 | | 7 0 | |
| (10H) | | (10H) | |
| 00110001 | | 00000001 | |
| 7 0 | | 7 0 | |

**Note:** 9

# CALL

## Generic Call

**Mnemonic:** CALL

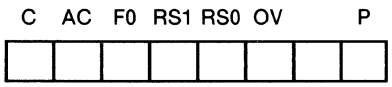
**Operands:** *code address*

**Format:** CALL *code address*

**Bit Pattern:** Translated to ACALL or LCALL as needed

**Operation:** Either ACALL or LCALL

**Flags:**



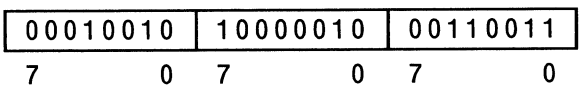

PSW

**Description:** This instruction is translated to ACALL when the specified code address contains no forward references and that address falls within the current 2K byte page; otherwise; it is translated to LCALL. This will not necessarily be the most efficient representation when a forward reference is used. See the description for ACALL and LCALL for more detail.

**Example:**

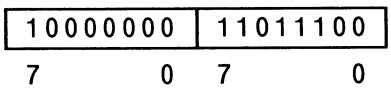
```
        ORG 80DCH
        CALL SUB3     ; Call SUB3 (SUB3 is a forward
          .           ; reference so LCALL is encoded
          .           ; even though ACALL would work in
          .           ; this case.)
SUB3:   POP 55H       ; Address 8233H
```

Encoded Instruction:

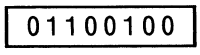



Before

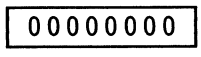
Program Counter



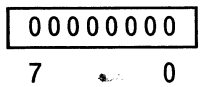

Stack Pointer




(65H)

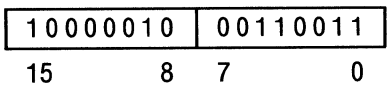



(66H)

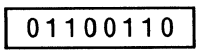



After

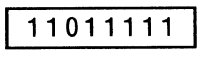
Program Counter



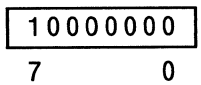

Stack Pointer




(65H)




(66H)




**Notes:** 1, 2, 3

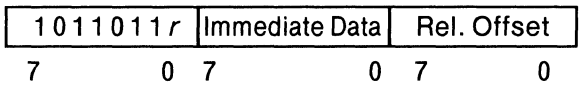
## Compare Indirect Address to Immediate Data, Jump if Not Equal

**Mnemonic:** CJNE

**Operands:** R*r* Register 0 <= *r* <= 1
*data* −256 <= *data* <= +255
*code address*

**Format:** CJNE @R*r*,#*data*,*code address*

**Bit Pattern:**

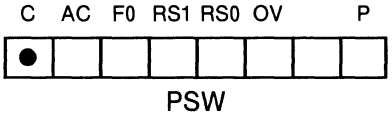



**Operation:**

(PC) ← (PC) + 3
IF ((R*r*)) < > *data*
THEN
(PC) ← (PC) + *relative offset*
IF ((R*r*)) < *data*
THEN
(C) ← 1
ELSE
(C) ← 0

**Bytes:** 3
**Cycles:** 2

**Flags:**




**Description:** This instruction compares the immediate data value with the memory location addressed by register *r*. If they are not equal, control passes to the specified code address. If they are equal, then control passes to the next sequential instruction.

If the immediate data value is greater than the contents of the specified data address, then the carry flag is set to 1; otherwise, it is reset to 0.

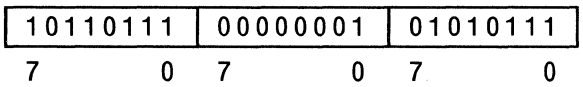
The Program Counter is incremented to the next instruction. If the operands are not equal, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:** *CJNE @R1,#01,SCAB* ; Jump if indirect address
; does not equal 1

.
.
.

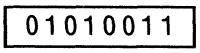
SCAB: MOV C,F0 ; 54H bytes from CJNE

Encoded Instruction:

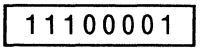



Before

Register 1

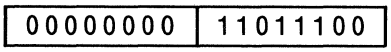



(53H)



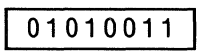

Carry Flag


Program Counter

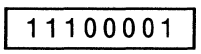



After

Register 1

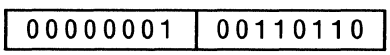



(53H)




Carry Flag


Program Counter




**Notes:** 4, 10, 11, 12, 15

## Compare Immediate Data to Accumulator, Jump if Not Equal

**Mnemonic:** CJNE

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | −256 <= *data* <= +255 |
| *code address* | |

**Format:** CJNE A,#*data*,*code address*

**Bit Pattern:**

| 10110100 | Immediate Data | Rel. Offset |
|---|---|---|
| 7 0 | 7 0 | 7 0 |

**Operation:**

(PC) ← (PC) + 3
IF (A) < > *data*
THEN
(PC) ← (PC) + *relative offset*
IF (A) < *data*
THEN
(C) ← 1
ELSE
(C) ← 0

**Bytes:** 3
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | | | |

PSW

**Description:** This instruction compares the immediate data value with the contents of the accumulator. If they are not equal, control passes to the specified code address. If they are equal, then control passes to the next sequential instruction.

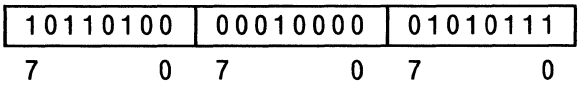
If the immediate data value is greater than the contents of the accumulator, then the carry flag is set to 1; otherwise, it is reset to 0.

The Program Counter is incremented to the next instruction. If the operands are not equal, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

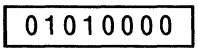
```
        ORG 10DCH
        CJNE A,#10H,NEXT ; Jump if accumulator does not equal
                         ; 10H
                .
                .
                .
NEXT:   INC A            ; Location 1136H
```

Encoded Instruction:



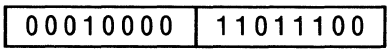

Before

Accumulator



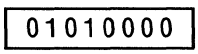

Carry Flag


Program Counter



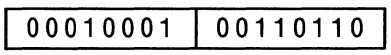

After

Accumulator




Carry Flag


Program Counter




**Notes:** 4, 10, 11, 12

## Compare Memory to Accumulator, Jump if Not Equal

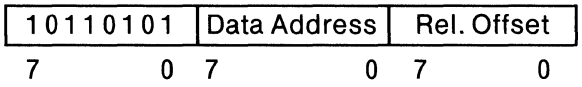
**Mnemonic:** CJNE

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data address* | 0 <= *data address* <= 255 |
| *code address* | |

**Format:** CJNE A,*data address*,*code address*

**Bit Pattern:**

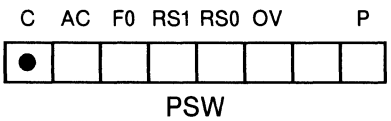



**Operation:**

(PC) ← (PC) + 3
IF (A) < > (*data address*)
THEN
(PC) ← (PC) + *relative offset*
IF (A) < (*data address*)
THEN
(C) ← 1
ELSE
(C) ← 0

**Bytes:** 3
**Cycles:** 2

**Flags:**



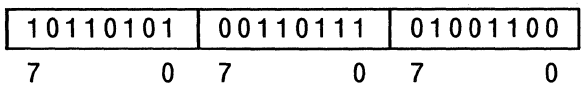

**Description:** This instruction compares the contents of the specified memory location to the contents of the accumulator. If they are not equal, control passes to the specified code address. If they are equal, then control passes to the next sequential instruction.

If the contents of the specified memory location is greater than the contents of the accumulator, then the carry flag is set to 1; otherwise, it is reset to 0.

The Program Counter is incremented to the next instruction. If the operands are not equal, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

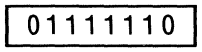
**Example:** *CJNE A,37H,TEST*; Jump if 37H and accumulator
. ; are not equal
.
.
TEST: INC A ; 4FH bytes from CJNE

Encoded Instruction:

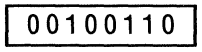



Before

(37H)

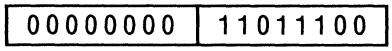



Accumulator



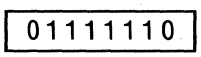

Carry Flag


Program Counter

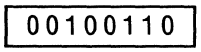



After

(37H)

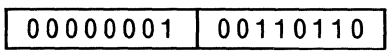



Accumulator




Carry Flag


Program Counter




**Notes:** 8, 10, 11, 12

# CJNE

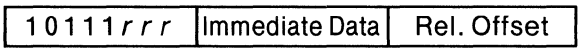
## Compare Immediate Data to Register, Jump if Not Equal

**Mnemonic:** CJNE

**Operands:** R*r* Register $0 <= r <= 7$
*data* $-256 <= data <= +255$
*code address*

**Format:** CJNE R*r*,#*data*,*code address*

**Bit Pattern:**

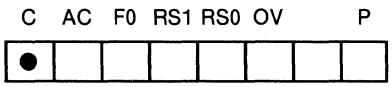



**Operation:**

(PC) ← (PC) + 3
IF (R*r*) < > *data*
THEN
(PC) ← (PC) + *relative offset*
IF (R*r*) < *data*
THEN
(C) ← 1
ELSE
(C) ← 0

**Bytes:** 3
**Cycles:** 2

**Flags:**




**Description:** This instruction compares the immediate data value with the contents of register *r*. If they are not equal, control passes to the specified code address. If they are equal, then control passes to the next sequential instruction.

If the immediate data value is greater than the contents of the specified register, then the carry flag is set to 1; otherwise, it is reset to 0.

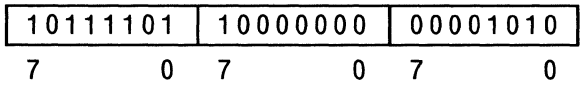
The Program Counter is incremented to the next instruction. If the operands are not equal, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:** *CJNE R5,#32H,SKIP10* ; Jump if register 5 does not
; equal 32H

.
.
.

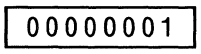
SKIP10: MOV R5,P0 ;13 bytes from CJNE

Encoded Instruction:



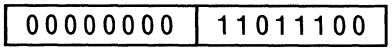

Before

Register 5



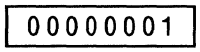

Carry Flag


Program Counter



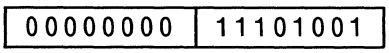

After

Register 5




Carry Flag


Program Counter




**Notes:** 4, 10, 11, 12

# CLR

## Clear Accumulator

**Mnemonic:** CLR

**Operands:** A Accumulator

**Format:** CLR A

**Bit Pattern:**

| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← 0

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction resets the accumulator to 0.

**Example:** *CLR A* ; Set accumulator to 0

Encoded Instruction:

| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Before

Accumulator

| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

After

Accumulator

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Note:** 5

# CLR

## Clear Carry Flag

**Mnemonic:** CLR

**Operands:** C Carry Flag

**Format:** CLR C

**Bit Pattern:**

| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (C) ← 0

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | | | |

PSW

**Description:** This instruction resets the carry flag to 0.

**Example:** *CLR C* ; Set carry flag to 0

Encoded Instruction:

| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | After |
|---|---|
| Carry Flag | Carry Flag |
| 1 | 0 |

**Notes:** None

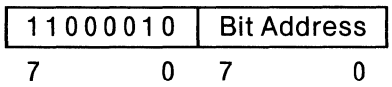
## Clear Bit

**Mnemonic:** CLR

**Operands:** *bit address* 0 <= *bit address* <= *255*

***Format:*** *CLR bit address*

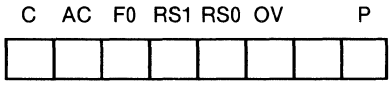
**Bit Pattern:**




**Operation:** (*bit address*) ← 0

**Bytes:** 2
**Cycles:** 1

**Flags:**

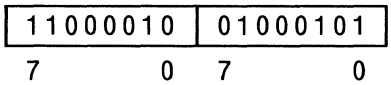



PSW

**Description:** This instruction resets the specified bit address to 0.

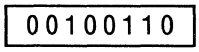
**Example:** *CLR 40.5* ; Set bit 5 of byte 40 to 0

Encoded Instruction:



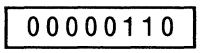

Before

(40)




After

(40)




**Notes:** None

# CPL

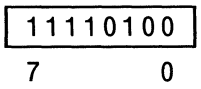
## Complement Accumulator

**Mnemonic:** CPL

**Operands:** A Accumulator

**Format:** CPL A

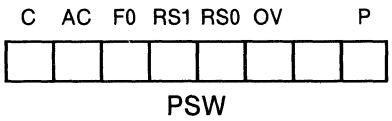
**Bit Pattern:**




**Operation:** (A) ← NOT (A)

**Bytes:** 1
**Cycles:** 1

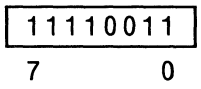
**Flags:**




**Description:** This instruction resets each 1 in the accumulator to 0, and sets each 0 in the accumulator to 1.

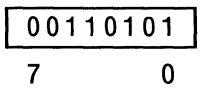
**Example:** *CPL A* ; Complement accumulator

Encoded Instruction:



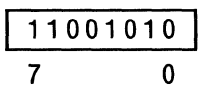

Before

Accumulator




After

Accumulator




**Notes:** None

## Complement Carry Flag

**Mnemonic:** CPL

**Operands:** C Carry flag

**Format:** CPL C

**Bit Pattern:**

| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

7 0

**Operation:** (C) ← NOT (C)

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | | | |

PSW

**Description:** This instruction sets the carry flag to 1 if it was 0, and resets the carry flag to 0 if it was 1.

**Example:** *CPL C* ; Complement Carry flag

Encoded Instruction:

| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

7 0

| Before | After |
|---|---|
| Carry Flag | Carry Flag |
| 1 | 0 |

**Notes:** None

# CPL

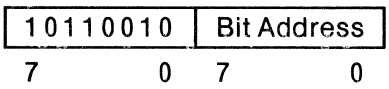
## Complement Bit

**Mnemonic:** CPL

**Operands:** *bit address* 0 <= *bit address* <= 255

**Format:** CPL *bit address*

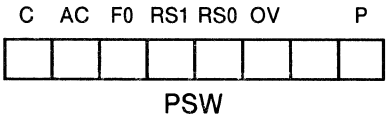
**Bit Pattern:**




**Operation:** (*bit address*) ← NOT (*bit address*)

**Bytes:** 2
**Cycles:** 1

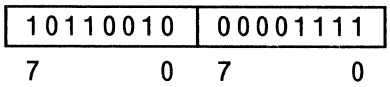
**Flags:**




**Description:** This instruction sets the contents of the specified bit address to 1 if it was 0, and resets the contents of the bit address to 0 if it was 1.

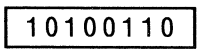
**Example:** *CPL 33.7* ; Set bit 7 of byte 33 to 0

Encoded Instruction:



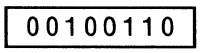

Before

(33)




After

(33)




**Notes:** None

## Decimal Adjust Accumulator

**Mnemonic:** DA

**Operands:** A Accumulator

**Format:** DA A

**Bit Pattern:**

| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

7 0

**Operation:** (See description below.)

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | | | ● |

PSW

**Description:** This instruction adjusts the contents of the accumulator to correspond to packed binary coded decimal (BCD) representation, after an add of two BCD numbers. If the auxiliary carry flag is 1, or the contents of the low order nibble (bits 0—3) of the accumulator is greater than 9, then 6 is added to the accumulator. If the carry flag is set before or after the add or the contents of the high order nibble (bits 4—7) is greater than 9, then 60H is added to the accumulator. The accumulator and the carry flag contain the final adjusted value.

**Example:**
```
ADD A,R1
DA A                    ; Adjust the Accumulator after add
```

Encoded Instruction:

| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

7 0

| Before | After |
|---|---|
| Accumulator | Accumulator |
| 1 0 0 1 1 0 1 1 | 0 0 0 0 0 0 0 1 |
| 7 0 | 7 0 |
| Carry Flag | Carry Flag |
| 0 | 1 |
| Auxiliary Carry Flag | Auxiliary Carry Flag |
| 0 | 0 |

**Notes:** 5, 6

# DEC

## Decrement Indirect Address

**Mnemonic:** DEC

**Operands:** R*r* Register 0 <= *r* <= 1

**Format:** DEC @R*r*

**Bit Pattern:**

| 0 | 0 | 0 | 1 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** ((R*r*)) ← ((R*r*)) −1

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction decrements the contents of the memory location addressed by register *r* by 1. It places the result in the addressed location.

**Example:** *DEC @R0* ; Decrement counter

Encoded Instruction:

| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Before

Register 0

| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

(37H)

| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

After

Register 0

| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

(37H)

| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Note:** 15

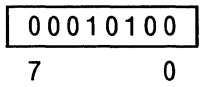
## Decrement Accumulator

**Mnemonic:** DEC

**Operands:** A Accumulator

**Format:** DEC A

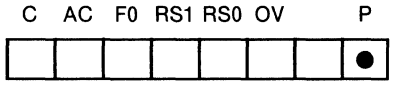
**Bit Pattern:**




**Operation:** (A) ← (A) –1

**Bytes:** 1
**Cycles:** 1

**Flags:**

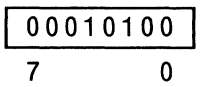



PSW

**Description:** This instruction decrements the contents of the accumulator by 1. It places the result in the accumulator.

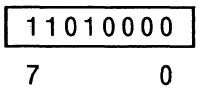
**Example:** *DEC A* ; Decrement accumulator

Encoded Instruction:



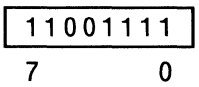

Before

Accumulator




After

Accumulator




**Note:** 5

# DEC

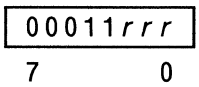
## Decrement Register

**Mnemonic:** DEC

**Operands:** R*r* Register 0 <= *r* <= 7

**Format:** DEC R*r*

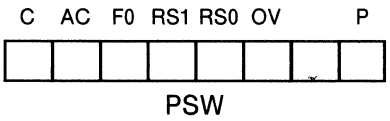
**Bit Pattern:**




**Operation:** (R*r*) ← (R*r*) –1

**Bytes:** 1
**Cycles:** 1

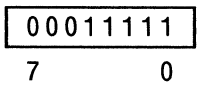
**Flags:**




**Description:** This instruction decrements the contents of register *r* by 1. It places the result in the specified register.

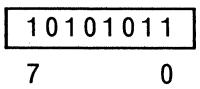
**Example:** *DEC R7* ; Decrement register 7

Encoded Instruction:



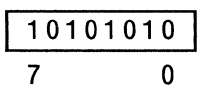

Before

Register 7




After

Register 7




**Notes:** None

# DEC

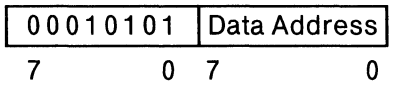
## Decrement Memory

**Mnemonic:** DEC

**Operands:** *data address* 0 <= *data address* <= 255

**Format:** DEC *data address*

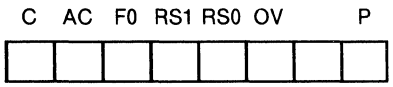
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) –1

**Bytes:** 2
**Cycles:** 1

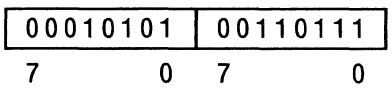
**Flags:**




**Description:** This instruction decrements the contents of the specified data address by 1. It places the result in the addressed location.

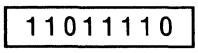
**Example:** *DEC 37H* ; Decrement counter

Encoded Instruction:



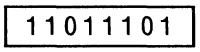

Before

(37H)




After

(37H)




**Note:** 9

# DIV

## Divide Accumulator by B

**Mnemonic:** DIV

**Operands:** AB Register Pair

**Format:** DIV AB

**Bit Pattern:**

| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (AB) ← (A) / (B)

**Bytes:** 1
**Cycles:** 4

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | ● | | ● |

PSW

**Description:** This instruction divides the contents of the accumulator by the contents of the multiplication register (B). Both operands are treated as unsigned integers. The accumulator contains the quotient; the multiplication register contains the remainder.

The carry flag is always cleared. Division by 0 sets the overflow flag; otherwise, it is cleared.

**Example:**
```
MOV B,#5
DIV AB          ; Divide accumulator by 5
```

Encoded Instruction:

| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Before

Accumulator

| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Multiplication Register (B)

| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

After

Accumulator

| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Multiplication Register (B)

| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Note:** 5

# DJNZ

## Decrement Register and Jump if Not Zero

**Mnemonic:** DJNZ

**Operands:** R*r* Register 0 <= *r* <= 7
*code address*

**Format:** DJNZ R*r*,*code address*

**Bit Pattern:**

| 11011*rrr* | Rel. Offset |
|---|---|
| 7 0 | 7 0 |

**Operation:**
(PC) ← (PC) + 2
(R*r*) ← (R*r*) −1
IF (R*r*) < > 0
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 2
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction decrements the contents of register *r* by 1, and places the result in the specified register. If the result of the decrement is 0, then control passes to the next sequential instruction; otherwise, control passes to the specified code address.

The Program Counter is incremented to the next instruction. If the decrement does not result in 0, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

```
LOOP1: ADD  A,R7          ; ADD index to accumulator
                .
                .
                .
       DJNZ R7,LOOP1      ; Decrement register 7 and
       INC A              ; jump to LOOP1 (15 bytes
                          ; backward from INC
                          ; instruction)
```

Encoded Instruction:

| 11011111 | 11110001 |
|---|---|
| 7 0 | 7 0 |

Before

Register 7

| 00000010 |
|---|
| 7 0 |

Program Counter

| 00000100 | 11011100 |
|---|---|
| 15 8 | 7 0 |

After

Register 7

| 00000001 |
|---|
| 7 0 |

Program Counter

| 00000100 | 11001111 |
|---|---|
| 15 8 | 7 0 |

**Notes:** 10, 11, 12

# DJNZ

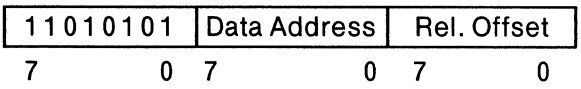
## Decrement Memory and Jump if Not Zero

**Mnemonic:** DJNZ

**Operands:** *data address* 0 <= *data address* <= 255
*code address*

**Format:** DJNZ *data address,code address*

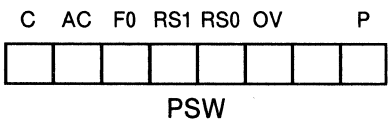
**Bit Pattern:**




**Operation:**
(PC) ← (PC) + 3
(*data address*) ← (*data address*) −1
IF (*data address*) < > 0
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 3
**Cycles:** 2

**Flags:**

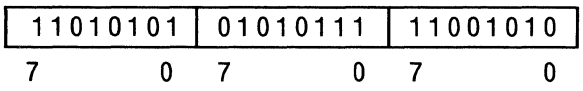



**Description:** This instruction decrements the contents of the specified data address by 1, and places the result in the addressed location. If the result of the decrement is 0, then control passes to the next sequential instruction; otherwise, control passes to the specified code address.

The Program Counter is incremented to the next instruction. If the decrement does not result in 0, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

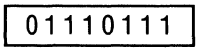
**Example:**
```
LOOP 3: MOV R7,57H        ; Store loop index in register 7
        .
        .
        .
        DJNZ 57H,LOOP3    ; Decrement 57H and jump
        INC A             ; backward to LOOP3 (51 bytes
                          ; backwards from the INC A
                          ; instruction)
```

Encoded Instruction:

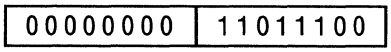



Before

(57H)

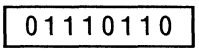



Program Counter

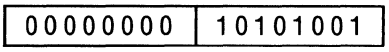



After

(57H)




Program Counter




**Notes:** 9, 10, 11, 12

# INC

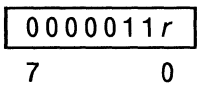
## Increment Indirect Address

**Mnemonic:** INC

**Operands:** R*r* Register 0 <= *r* <= 1

**Format:** INC @R*r*

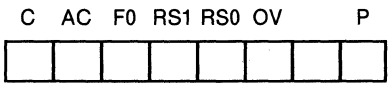
**Bit Pattern:**




**Operation:** ((R*r*)) ← ((R*r*)) + 1

**Bytes:** 1
**Cycles:** 1

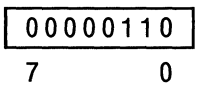
**Flags:**




**Description:** This instruction increments the contents of the memory location addressed by register *r* by 1. It places the result in the addressed location.

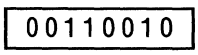
**Example:** *INC @R0* ; Increment counter

Encoded Instruction:

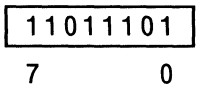



Before

Register 0

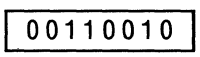



(32H)

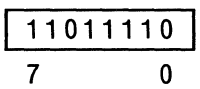



After

Register 0




(32H)




**Note:** 15

## Increment Accumulator

**Mnemonic:** INC

**Operands:** A Accumulator

**Format:** INC A

**Bit Pattern:**

| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← (A) + 1

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction increments the contents of the accumulator by 1. It places the result in the accumulator.

**Example:** *INC A* ; Increment accumulator

Encoded Instruction:

| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Before

Accumulator

| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

After

Accumulator

| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Note:** 5

# INC

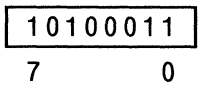
## Increment Data Pointer

**Mnemonic:** INC

**Operands:** DPTR Data Pointer

**Format:** INC DPTR

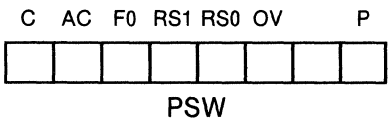
**Bit Pattern:**




**Operation:** (DPTR) ← (DPTR) + 1

**Bytes:** 1
**Cycles:** 2

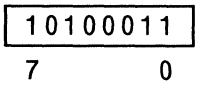
**Flags:**




**Description:** This instruction increments the 16-bit contents of the data pointer by 1. It places the result in the data pointer.

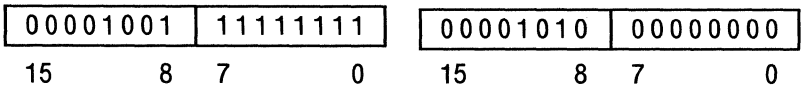
**Example:** *INC DPTR* ; Increment data pointer

Encoded Instruction:




Before After

Data Pointer Data Pointer




**Notes:** None

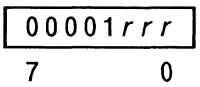
## Increment Register

**Mnemonic:** INC

**Operands:** R*r* Register 0 <= *r* <= 7

**Format:** INC R*r*

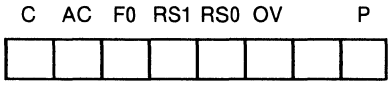
**Bit Pattern:**




**Operation:** (R*r*) ← (R*r*) + 1

**Bytes:** 1
**Cycles:** 1

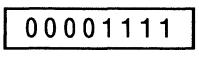
**Flags:**




**Description:** This instruction increments the contents of register *r* by 1. It places the result in the specified register.

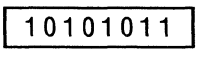
**Example:** *INC R7* ; Increment register 7

Encoded Instruction:



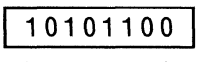

Before

Register 7




After

Register 7




**Notes:** None

# INC

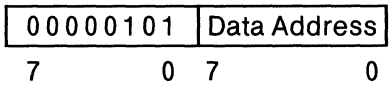
## Increment Memory

**Mnemonic:** INC

**Operands:** *data address* 0 <= *data address* <= 255

**Format:** INC *data address*

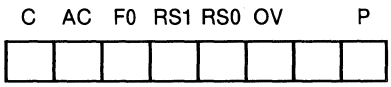
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) + 1

**Bytes:** 2
**Cycles:** 1

**Flags:**

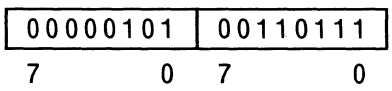



PSW

**Description:** This instruction increments the contents of the specified data address by 1. It places the result in the addressed location.

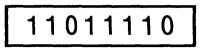
**Example:** *INC 37H* ; Increment 37H

Encoded Instruction:



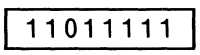

Before

(37H)




After

(37H)




**Note:** 9

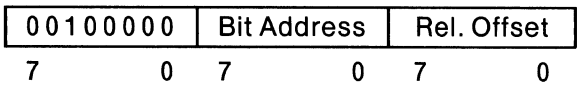
## Jump if Bit Is Set

**Mnemonic:** JB

**Operands:** *bit address* 0 <= *bit address* <= 255
*code address*

**Format:** JB *bit address*,*code address*

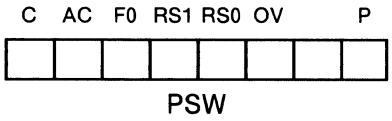
**Bit Pattern:**




**Operation:**
(PC) ← (PC) + 3
IF (*bit address*) = 1
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 3
**Cycles:** 2

**Flags:**



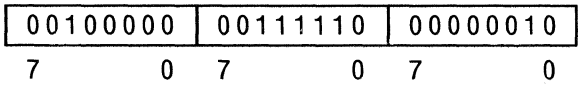

**Description:** This instruction tests the specified bit address. If it is 1, control passes to the specified code address. Otherwise, control passes to the next sequential instruction.

The Program Counter is incremented to the next instruction. If the test was successful, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

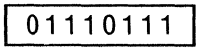
```
        JB 39.6,EXIT    ; Jump if bit 6 of byte 39 is 1
          .
          .
          .
        SJMP TOP
EXIT:   MOV A,39        ; Move 39 to accumulator (EXIT label
                        ; is 5 bytes from jump statement)
```

Encoded Instruction:

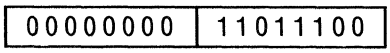



Before

(39)

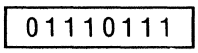



Program Counter

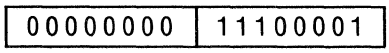



After

(39)




Program Counter




**Notes:** 10, 11, 12

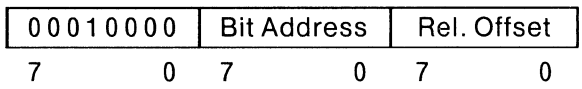
## Jump and Clear if Bit Is Set

**Mnemonic:** JBC

**Operands:** *bit address* 0 <= *bit address* <= 255
*code address*

**Format:** JBC *bit address*,*code address*

**Bit Pattern:**

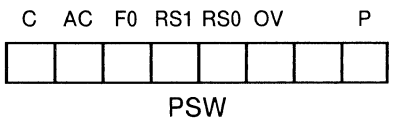



**Operation:**

(PC) ← (PC) + 3
IF (*bit address*) = 1
THEN
(*bit address*) ← 0
(PC) ← (PC) + *relative offset*

**Bytes:** 3
**Cycles:** 2

**Flags:**



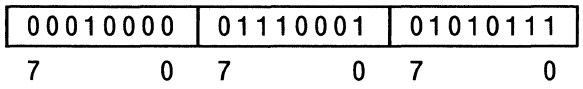

**Description:** This instruction tests the specified bit address. If it is 1, the bit is cleared, and control passes to the specified code address. Otherwise, control passes to the next sequential instruction.

The Program Counter is incremented to the next instruction. If the test was successful, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

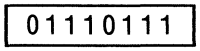
```
        ORG 0DCH
        JBC 46.1,OUT3  ; Test bit 1 of byte 46
            .          ; jump and clear if 1
            .
            .
        ORG136H
OUT3:   INC R7
```

Encoded Instruction:

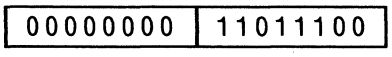



Before

(46)

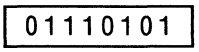



Program Counter

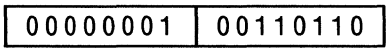



After

(46)




Program Counter




**Notes:** 10, 11, 12

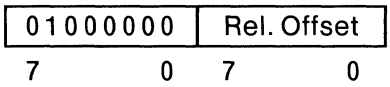
## Jump if Carry Is Set

**Mnemonic:** JC

**Operands:** *code address*

**Format:** JC *code address*

**Bit Pattern:**

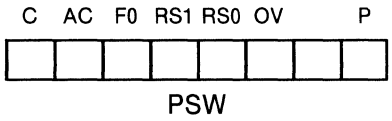



**Operation:**

(PC) ← (PC) + 2
IF (C) = 1
THEN
(PC) ← (PC) + *relative code*

**Bytes:** 2
**Cycles:** 2

**Flags:**



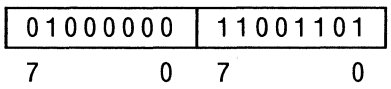

**Description:** This instruction tests the contents of the carry flag. If it is 1, then control passes to the specified code address. Otherwise, control passes to the next sequential instruction.

The Program Counter is incremented to the next instruction. If the test was successful, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

```
FIXUP: CLR C        ; Clear carry
       .
       .
       .
       JC FIXUP     ; If carry is 1 go to FIXUP
                    ; 49 bytes backwards from the JC
                    ; instruction
```

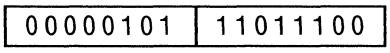
Encoded Instruction:




Before

Carry Flag

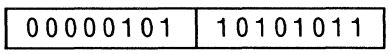

Program Counter




After

Carry Flag


Program Counter




**Notes:** 10, 11, 12

## Generic Jump

**Mnemonic:** JMP

**Operands:** *code address* 0 <= *code address* <= 65,535

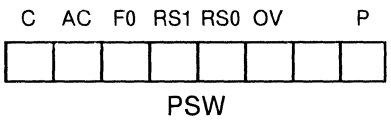
**Format:** JMP *code address*

**Bit Pattern:** Translated to AJMP, LJMP, or SJMP, as needed

**Operation:** Either AJMP, SJMP or LJMP

**Bytes:**
**Cycles:**

**Flags:**

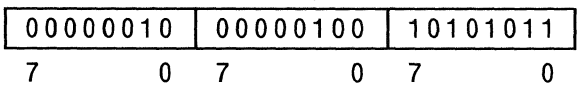



**Description:** This instruction will be translated to SJMP if the specified code address contains no forward references and that address falls within −128 and +127 of the address of the next instruction. It will be translated to AJMP if the code address contains no forward references and the specified code address falls within the current 2K byte page. Otherwise, the JMP instruction is translated to LJMP. If forward references are used to specify the jump destination, then it will not necessarily be the most efficient representation. See the descriptions for SJMP, AJMP, and LJMP for more detail.

**Example:**

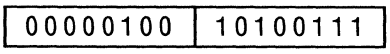
```
        JMP SKIP      ; Jump to SKIP
FF:     INC A         ; Increment A
SKIP:   INC R5        ; Increment register 5
```

Encoded Instruction:



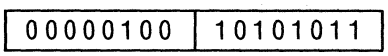

Before

Program Counter




After

Program Counter




**Notes:** None

# JMP

## Jump to Sum of Accumulator and Data Pointer

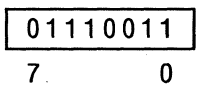
**Mnemonic:** JMP

**Operands:**

| | |
|---|---|
| A | Accumulator |
| DPTR | Data Pointer |

**Format:** JMP @A + DPTR

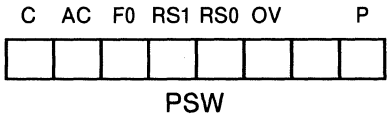
**Bit Pattern:**




**Operation:** (PC) ← (A) + (DPTR)

**Bytes:** 1
**Cycles:** 2

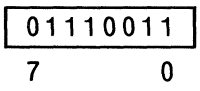
**Flags:**




**Description:** This instruction adds the contents of the accumulator with the contents of the data pointer. It transfers control to the code address formed by that sum.

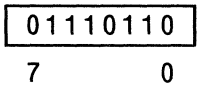
**Example:** *JMP @A + DPTR* ; Jump relative to the accumulator

Encoded Instruction:

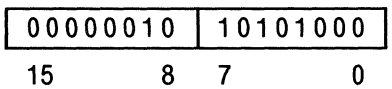



Before

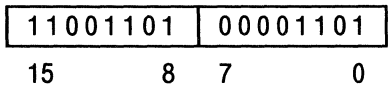
Accumulator




Data Pointer

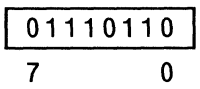



Program Counter

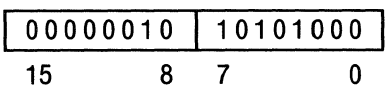



After

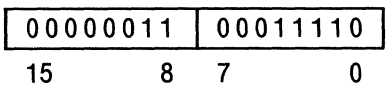
Accumulator




Data Pointer




Program Counter




**Notes:** None

## Jump if Bit Is Not Set

**Mnemonic:** JNB

**Operands:** *bit address*
*code address*

**Format:** JNB *bit address,code address*

**Bit Pattern:**

| 00110000 | Bit Address | Rel. Offset |
|---|---|---|
| 7 0 | 7 0 | 7 0 |

**Operation:**
(PC) ← (PC) + 3
IF (*bit address*) = 0
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 3
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

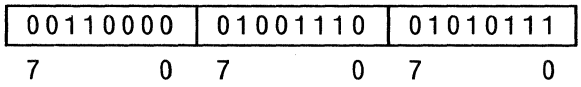
**Description:** This instruction tests the specified bit address. If it is 0, control passes to specified code address. Otherwise, control passes to the next sequential instruction.

The Program Counter is incremented to the next instruction. If the test was successful, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

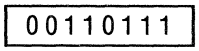
```
        ORG 0DCH
        JNB 41.6,EXIT    ; If bit 6 of byte 41 is 0 go to EXIT
          .
          .
          .
EXIT:   ADD A,41         ; At location 136H
```

Encoded Instruction:

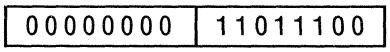



Before

(41)

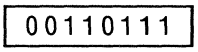



Program Counter

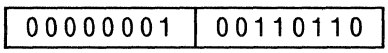



After

(41)




Program Counter




**Notes:** 10, 11, 12

# JNC

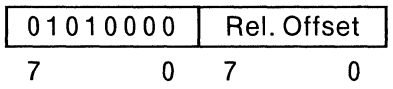
## Jump if Carry Is Not Set

**Mnemonic:** JNC

**Operands:** *code address*

**Format:** JNC *code address*

**Bit Pattern:**

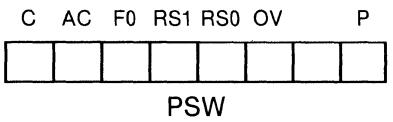



**Operation:**

(PC) ← (PC) + 2
IF (C) = 0
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 2
**Cycles:** 2

**Flags:**




**Description:** This instruction tests the contents of the carry flag. If it is 0, control passes to the specified code address. Otherwise, control passes to the next sequential instruction.

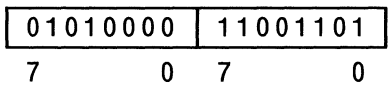
The Program Counter is incremented to the next instruction. If the test was successful, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:** FIXUP: MOV A,R5

.
.
.

*JNC FIXUP* ; Jump to FIXUP if carry is 0
; (51 bytes backwards)

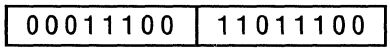
Encoded Instruction:




Before

Carry Flag

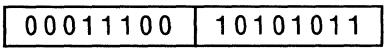

Program Counter




After

Carry Flag


Program Counter




**Notes:** 10, 11, 12

# JNZ

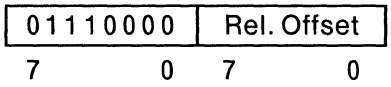
## Jump if Accumulator Is Not Zero

**Mnemonic:** JNZ

**Operands:** *code address*

**Format:** JNZ *code address*

**Bit Pattern:**

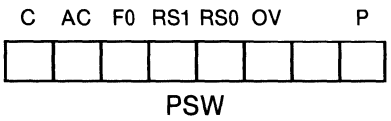



**Operation:**

(PC) ← (PC) + 2
IF (A) < > 0
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 2
**Cycles:** 2

**Flags:**



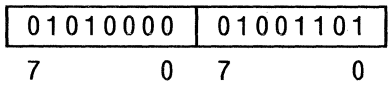

**Description:** This instruction tests the accumulator. If it is not equal to 0, control passes to the specified code address. Otherwise, control passes to the next sequential instruction.

The Program Counter is incremented to the next instruction. If the accumulator is not 0, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

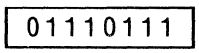
```
        JNZ TEST    ; Jump if accumulator is not 0
          .         ; 77 bytes forward
          .
          .
TEST:   MOV R3,A
```

Encoded Instruction:

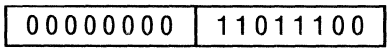



Before

Accumulator

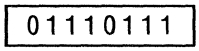



Program Counter

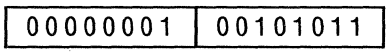



After

Accumulator




Program Counter




**Notes:** 10, 11, 12

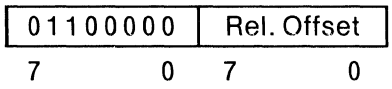
## Jump if Accumulator Is Zero

**Mnemonic:** JZ

**Operands:** *code address*

**Format:** JZ *code address*

**Bit Pattern:**

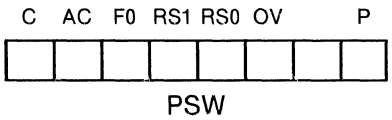



**Operation:**

(PC) ← (PC) + 2
IF (A) = 0
THEN
(PC) ← (PC) + *relative offset*

**Bytes:** 2
**Cycles:** 2

**Flags:**




**Description:** This instruction tests the accumulator. If it is 0, control passes to the specified code address. Otherwise, control passes to the next sequential instruction.

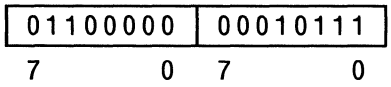
The Program Counter is incremented to the next instruction. If the accumulator is 0, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:** *JZ EMPTY* ; Jump to EMPTY if accumulator is 0

.
.
.

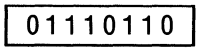
EMPTY: INC A ; 25 bytes from JZ instruction

Encoded Instruction:




Before

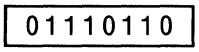
Accumulator



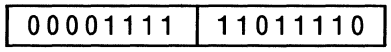

Program Counter


After

Accumulator




Program Counter




**Notes:** 10, 11, 12

# LCALL

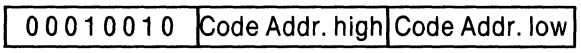
## Long Call

**Mnemonic:** LCALL

**Operands:** *code address* 0 <= *code address* <= 65,535

**Format:** LCALL *code address*

**Bit Pattern:**

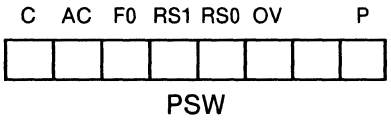



**Operation:**

(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC *low*)
(SP) ← (SP) + 1
((SP)) ← (PC *high*)
(PC) ← *code address*

**Bytes:** 3
**Cycles:** 2

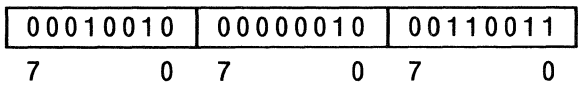
**Flags:**




**Description:** This instruction stores the contents of the program counter (the return address) on the stack, then transfers control to the 16-bit code address specified as the operand.

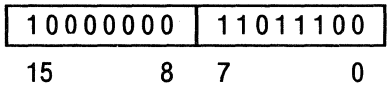
**Example:**
```
SERVICE:  INC A        ; Resides at location 233H
          .
          .
          .
          RETI
          .
          .
          .
          ORG 80 DCH
          LCALL SERVICE   ; Call SERVICE
```

Encoded Instruction:

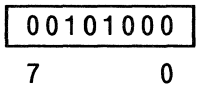



Before

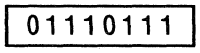
Program Counter



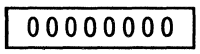

Stack Pointer




(29H)

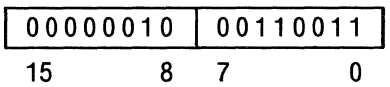



(2AH)

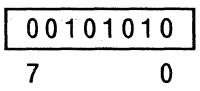



After

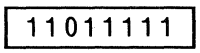
Program Counter



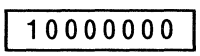

Stack Pointer




(29H)




(2AH)




**Notes:** 1, 2, 3

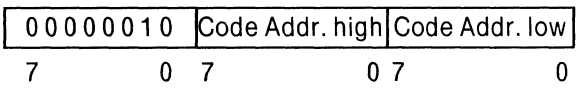
## Long Jump

**Mnemonic:** LJMP

**Operands:** *code address* 0 <= *code address* <= 65,535

**Format:** LJMP *code address*

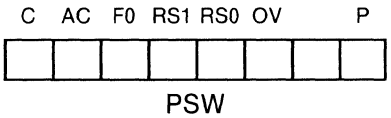
**Bit Pattern:**




**Operation:** (PC) ← *code address*

**Bytes:** 3
**Cycles:** 2

**Flags:**

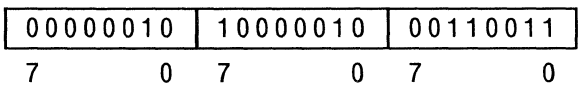



**Description:** This instruction transfers control to the 16-bit code address specified as the operand.

**Example:**

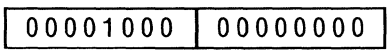
```
        ORG 800H
        LJMP FAR        ; Jump to FAR
          .
          .
          .
FAR:    INC A           ; Current code location (8233H)
```

Encoded Instruction:



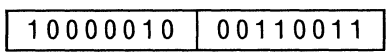

Before

Program Counter




After

Program Counter




**Notes:** None

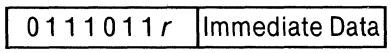
## Move Immediate Data to Indirect Address

**Mnemonic:** MOV

**Operands:** R*r* Register 0 <= *r* <= 1
*data* −256 <= *data* <= +255

**Format:** MOV @R*r*,#*data*

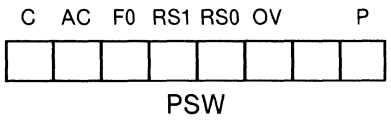
**Bit Pattern:**




**Operation:** ((R*r*)) ← *data*

**Bytes:** 2
**Cycles:** 1

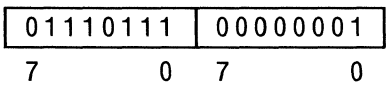
**Flags:**




**Description:** This instruction moves the 8-bit immediate data value to the memory location addressed by the contents of register *r*.

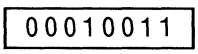
**Example:** *MOV @R1,#01H* ; Move 1 to indirect address

Encoded Instruction:

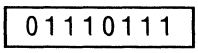



Before

Register 1

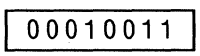



(13H)

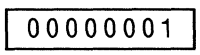



After

Register 1




(13H)




**Notes:** 4, 15

## Move Accumulator to Indirect Address

**Mnemonic:** MOV

**Operands:**

| | |
|---|---|
| R*r* | Register 0 <= *r* <= 1 |
| A | Accumulator |

**Format:** MOV @R*r*,A

**Bit Pattern:**

| 1 | 1 | 1 | 1 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** ((R*r*)) ← (A)

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction moves the contents of the accumulator to the memory location addressed by the contents of register *r*.

**Example:** *MOV @R0,A* ; Move accumulator to indirect
; address

Encoded Instruction:

| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | | After | |
|---|---|---|---|
| Register 0 | | Register 0 | |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 7 | 0 | | | | | | | 7 | 0 | | | | | | |
| (38H) | | (38H) | |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | | | | | | | 7 | 0 | | | | | | |
| Accumulator | | Accumulator | |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | | | | | | | 7 | 0 | | | | | | |

**Note:** 15

# MOV

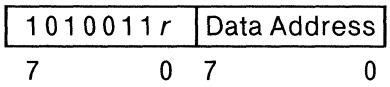
## Move Memory to Indirect Address

**Mnemonic:** MOV

**Operands:** R*r* Register 0 <= *r*<= 1
*data address* 0 <= *data address* <= 255

**Format:** MOV @R*r*,*data address*

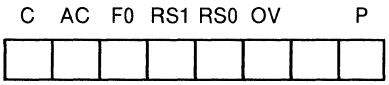
**Bit Pattern:**




**Operation:** ((R*r*)) ← (*data address*)

**Bytes:** 2
**Cycles:** 2

**Flags:**

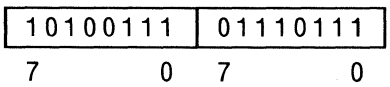



PSW

**Description:** This instruction moves the contents of the specified data address to the memory location addressed by the contents of register *r*.

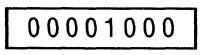
**Example:** *MOV @R1,77H* ; Move the contents of 77H to indirect
; address

Encoded Instruction:

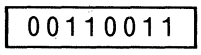



Before

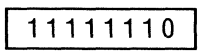
Register 1




(08H)

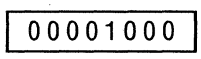



(77H)

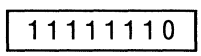



After

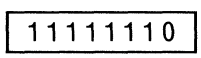
Register 1




(08H)




(77H)




**Notes:** 8, 15

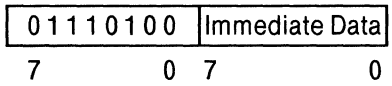
## Move Immediate Data to Accumulator

**Mnemonic:** MOV

**Operands:** A — Accumulator
*data* — $-256 <= data <= +255$

**Format:** MOV A,*#data*

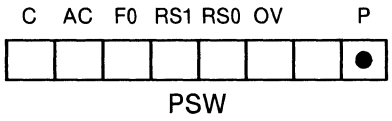
**Bit Pattern:**




**Operation:** (A) ← *data*

**Bytes:** 2
**Cycles:** 1

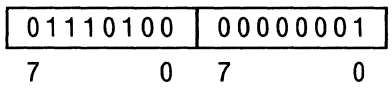
**Flags:**




**Description:** This instruction moves the 8-bit immediate data value to the accumulator.

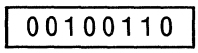
**Example:** *MOV A,#01H* ; Initialize the accumulator to 1

Encoded Instruction:



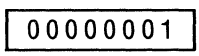

Before

Accumulator




After

Accumulator




**Notes:** 4, 5

## Move Indirect Address to Accumulator

**Mnemonic:** MOV

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** MOV A,@R*r*

**Bit Pattern:**

| 1 | 1 | 1 | 0 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← ((R*r*))

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction moves the contents of the data memory location addressed by register *r* to the accumulator.

**Example:**

```
MOV A,@R1          ; Move indirect address to
                   ; accumulator
```

Encoded Instruction:

| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | | After | |
|---|---|---|---|
| Accumulator | | Accumulator | |
| 1 0 0 0 0 1 1 0 | | 1 1 1 0 1 0 0 0 | |
| 7 | 0 | 7 | 0 |
| Register 1 | | Register 1 | |
| 0 0 0 1 1 1 0 0 | | 0 0 0 1 1 1 0 0 | |
| 7 | 0 | 7 | 0 |
| (1CH) | | (1CH) | |
| 1 1 1 0 1 0 0 0 | | 1 1 1 0 1 0 0 0 | |
| 7 | 0 | 7 | 0 |

**Notes:** 5, 15

## Move Register to Accumulator

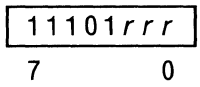
**Mnemonic:** MOV

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 7 |

**Format:** MOV A,R*r*

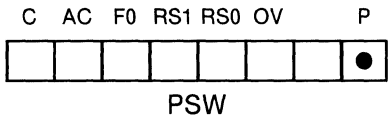
**Bit Pattern:**




**Operation:** (A) ← (R*r*)

**Bytes:** 1
**Cycles:** 1

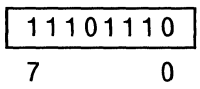
**Flags:**




**Description:** This instruction moves the contents of register *r* to the accumulator.

**Example:** *MOV A,R6* ; Move R6 to accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00101110 | 10000101 |
| 7 0 | 7 0 |
| Register 6 | Register 6 |
| 10000101 | 10000101 |
| 7 0 | 7 0 |

**Note:** 5

# MOV

## Move Memory to Accumulator

**Mnemonic:** MOV

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data address* | 0 <= *data address* <= 255 |

**Format:** MOV A,*data address*

**Bit Pattern:**

| 11100101 | Data Address |
|---|---|
| 7 0 | 7 0 |

**Operation:** (A) ← (*data address*)

**Bytes:** 2
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction moves the contents of data memory at the specified address to the accumulator.

**Example:** *MOV A,P1* ; Move the contents of Port 1 to
; accumulator

Encoded Instruction:

| 11100101 | 10010000 |
|---|---|
| 7 0 | 7 0 |

| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00100110 | 01111001 |
| 7 0 | 7 0 |
| Port I (90H) | Port I (90H) |
| 01111001 | 01111001 |
| 7 0 | 7 0 |

**Notes:** 5, 8

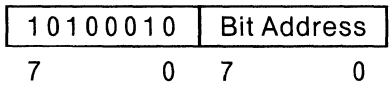
## Move Bit to Carry Flag

**Mnemonic:** MOV

**Operands:** C — Carry Flag
*bit address* — 0 <= *bit address* <= 255

**Format:** MOV C,*bit address*

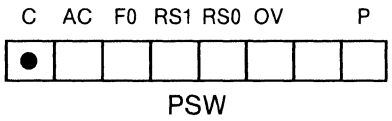
**Bit Pattern:**




**Operation:** (C) ← (*bit address*)

**Bytes:** 2
**Cycles:** 1

**Flags:**

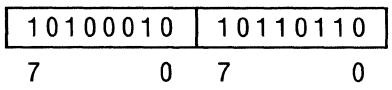



**Description:** This instruction moves the contents of the specified bit address to the carry flag.

**Example:**

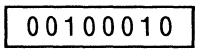
```
MOV C,TXD        ; Move the contents of TXD to Carry
                 ; flag
```

Encoded Instruction:




Before

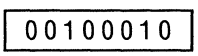
Port 3 (B0H)




Carry Flag


After

Port 3 (B0H)




Carry Flag


**Notes:** None

## Move Immediate Data to Data Pointer

**Mnemonic:** MOV

**Operands:** Data Pointer
*data* 0 <= *data* <= 65,535

**Format:** MOV DPTR,#*data*

**Bit Pattern:**

| 10010000 | Imm. Data high | Imm. Data low |
|---|---|---|
| 7 0 | 7 0 | 7 0 |

**Operation:** (DPTR) ← *data*

**Bytes:** 3
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction moves the 16-bit immediate data value to the data pointer.

**Example:** *MOV DPTR,#0F4FH* ; Initialize the data pointer to 0F4FH

Encoded Instruction:

| 10010000 | 00001111 | 01001111 |
|---|---|---|
| 7 0 | 7 0 | 7 0 |

Before

Data Pointer

| 00000000 | 11011100 |
|---|---|
| 15 8 | 7 0 |

After

Data Pointer

| 00001111 | 01001111 |
|---|---|
| 15 8 | 7 0 |

**Notes:** None

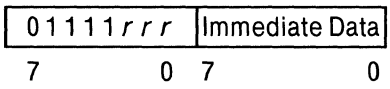
## Move Immediate Data to Register

**Mnemonic:** MOV

**Operands:** R*r* Register 0 <= *r* <= 7
*data* −256 <= *data* <= +255

**Format:** MOV R*r*,#*data*

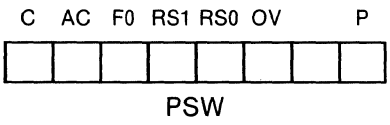
**Bit Pattern:**




**Operation:** (R*r*) ← *data*

**Bytes:** 2
**Cycles:** 1

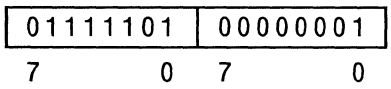
**Flags:**




**Description:** This instruction moves the 8-bit immediate data value to register *r*.

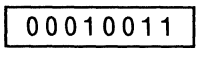
**Example:** *MOV R5,#01H* ; Initialize register 1

Encoded Instruction:



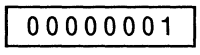

Before

Register 5




After

Register 5




**Note:** 4

## Move Accumulator to Register

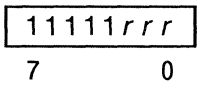
**Mnemonic:** MOV

**Operands:**

| | |
|---|---|
| R*r* | Register 0 <= *r* <= 7 |
| A | Accumulator |

**Format:** MOV R*r*,A

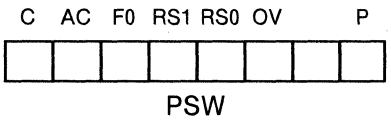
**Bit Pattern:**




**Operation:** (R*r*) ← (A)

**Bytes:** 1
**Cycles:** 1

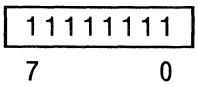
**Flags:**




**Description:** This instruction moves the contents of the accumulator to register *r*.

**Example:** *MOV R7,A* ; Move accumulator to register 7

Encoded Instruction:




| Before | After |
|---|---|
| Register 7 | Register 7 |
| 11011100 | 00111000 |
| 7 0 | 7 0 |
| Accumulator | Accumulator |
| 00111000 | 00111000 |
| 7 0 | 7 0 |

**Notes:** None

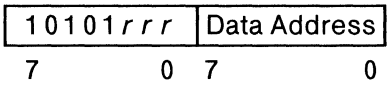
## Move Memory to Register

**Mnemonic:** MOV

**Operands:** R*r* Register 0 <= *r* <= 7
*data address* 0 <= *data address* <= 255

**Format:** MOV R*r*,*data address*

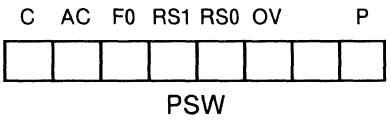
**Bit Pattern:**




**Operation:** (R*r*) ← (*data address*)

**Bytes:** 2
**Cycles:** 2

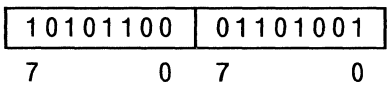
**Flags:**




**Description:** This instruction moves the contents of the specified data address to register *r*.

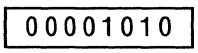
**Example:** *MOV R4,69H* ; Move contents of 69H to register 4

Encoded Instruction:

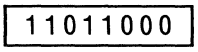



Before

Register 4

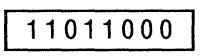



(69H)

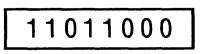



After

Register 4




(69H)




**Note:** 8

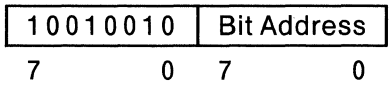
## Move Carry Flag to Bit

**Mnemonic:** MOV

**Operands:** *bit address* 0 <= *bit address* <= 255
C Carry Flag

**Format:** MOV *bit address*,C

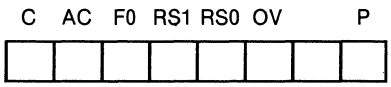
**Bit Pattern:**




**Operation:** (*bit address*) ← (C)

**Bytes:** 2
**Cycles:** 2

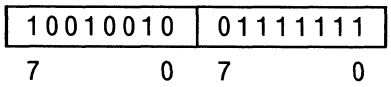
**Flags:**




**Description:** This instruction moves the contents of the carry flag to the specified bit address.

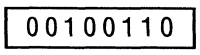
**Example:** *MOV 2FH.7,C* ; Move C to bit address 7FH

Encoded Instruction:




Before

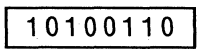
(2FH)




Carry Flag


After

(2FH)




Carry Flag


**Notes:** None

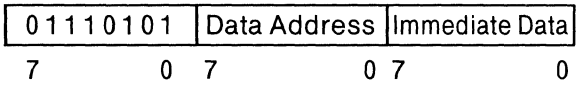
## Move Immediate Data to Memory

**Mnemonic:** MOV

**Operands:** *data address* 0 <= *data address* <= 255
*data* −256 <= *data* <= +255

**Format:** MOV *data address*,#*data*

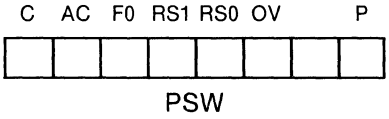
**Bit Pattern:**




**Operation:** (*data address*) ← *data*

**Bytes:** 3
**Cycles:** 2

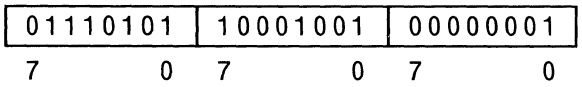
**Flags:**




**Description:** This instruction moves the 8-bit immediate data value to the specified data address.

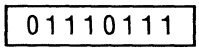
**Example:** *MOV TMOD,#01H* ; Initialize Timer Mode to 1

Encoded Instruction:



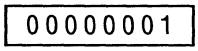

Before

TMOD (89H)




After

TMOD (89H)




**Notes:** 4, 9

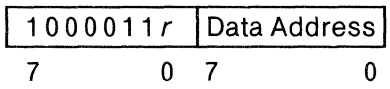
## Move Indirect Address to Memory

**Mnemonic:** MOV

**Operands:** *data address* 0 <= *data address* <= 255
Rr Register 0 <= *r* <= 1

**Format:** MOV *data address*,@R*r*

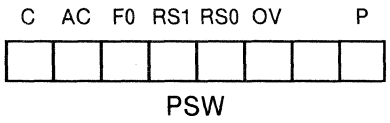
**Bit Pattern:**




**Operation:** (*data address*) ← ((R*r*))

**Bytes:** 2
**Cycles:** 2

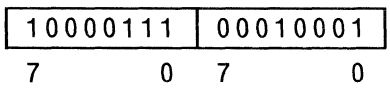
**Flags:**




**Description:** This instruction moves the contents of memory at the location addressed by register *r* to the specified data address.

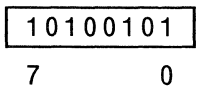
**Example:** *MOV 11H,@R1* ; Move indirect address to 11H

Encoded Instruction:

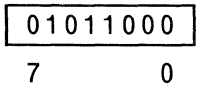



Before

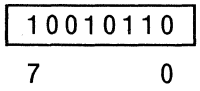
(11H)




Register 1

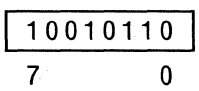



(58H)

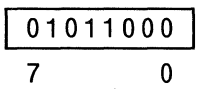



After

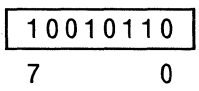
(11H)




Register 1




(58H)




**Notes:** 9, 15

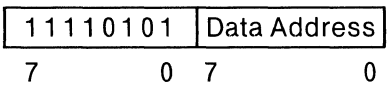
## Move Accumulator to Memory

**Mnemonic:** MOV

**Operands:** *data address* 0 <= *data address* <= 255
A Accumulator

**Format:** MOV *data address*,A

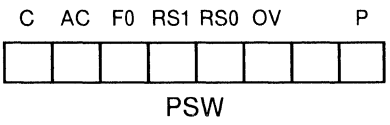
**Bit Pattern:**




**Operation:** (*data address*) ← (A)

**Bytes:** 2
**Cycles:** 1

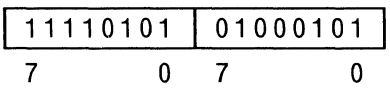
**Flags:**




**Description:** This instruction moves the contents of the accumulator to the specified data address.

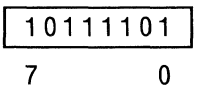
**Example:** *MOV 45H,A* ; Move accumulator to 45H

Encoded Instruction:

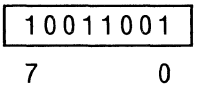



Before

(45H)

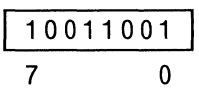



Accumulator

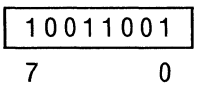



After

(45H)




Accumulator




**Note:** 9

# MOV

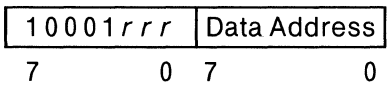
## Move Register to Memory

**Mnemonic:** MOV

**Operands:** *data address* 0 <= *data address* <= 255
Rr Register 0 <= *r* <= 7

**Format:** MOV *data address*,Rr

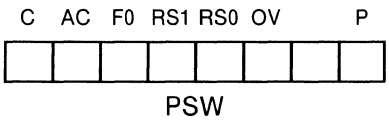
**Bit Pattern:**




**Operation:** (*data address*) ← (Rr)

**Bytes:** 2
**Cycles:** 2

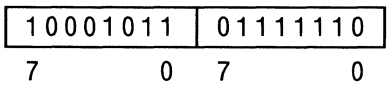
**Flags:**




**Description:** This instruction moves the contents of register *r* to the specified data address.

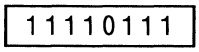
**Example:** *MOV 7EH,R3* ; Move R3 to location 7EH

Encoded Instruction:

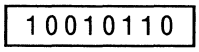



Before

(7EH)

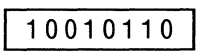



Register 3

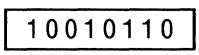



After

(7EH)




Register 3




**Note:** 9

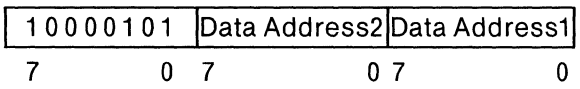
## Move Memory to Memory

**Mnemonic:** MOV

**Operands:** *data address1* 0 <= *data address1* <= 255
*data address2* 0 <= *data address2* <= 255

**Format:** MOV *data address1*,*data address2*

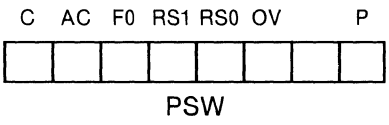
**Bit Pattern:**




**Operation:** (*data address1*) ← (*data address2*)

**Bytes:** 3
**Cycles:** 2

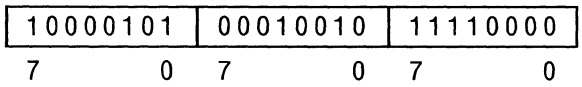
**Flags:**




**Description:** This instruction moves the contents of the source data address (*data address2*) to the destination data address (*data address1*).

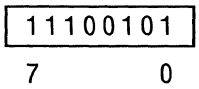
**Example:** *MOV B,12H* ; Move the contents of 12H to B (F0H)

Encoded Instruction:

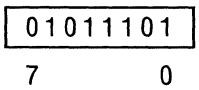



Before

(12H)

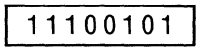



(F0H)

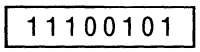



After

(12H)




(F0H)




**Note:** 16

# MOVC

## Move Code Memory Offset from Data Pointer to Accumulator

**Mnemonic:** MOVC

**Operands:**

| | |
|---|---|
| A | Accumulator |
| DPTR | Data Pointer |

**Format:** MOVC A,@A+DPTR

**Bit Pattern:**

| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← ((A) + (DPTR))

**Bytes:** 1
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction adds the contents of the data pointer with the contents of the accumulator. It uses that sum as an address into code memory and places the contents of that address in the accumulator.

The high-order byte of the sum moves to Port 2 and the low-order byte of the sum moves to Port 0.

**Example:** *MOVC A,@A+DPTR* ; Look up value in table

Encoded Instruction:

| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Before

Accumulator

| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Data Pointer

| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | | 8 | 7 | | | | | | | 0 |

(0302H)

| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

After

Accumulator

| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Data Pointer

| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | | 8 | 7 | | | | | | | 0 |

(0302H)

| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Notes: 5

# MOVC

## Move Code Memory Offset from Program Counter to Accumulator

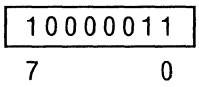
**Mnemonic:** MOVC

**Operands:**

| | |
|---|---|
| A | Accumulator |
| PC | Program Counter |

**Format:** MOVC A,@A+PC

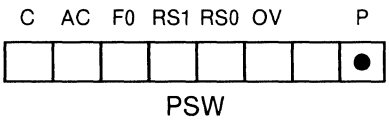
**Bit Pattern:**




**Operation:**
(PC) ← (PC) + 1
(A) ← ((A) + (PC))

**Bytes:** 1
**Cycles:** 2

**Flags:**

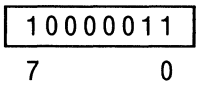



**Description:** This instruction adds the contents of the incremented program counter with the contents of the accumulator. It uses that sum as an address into code memory and places the contents of that address in the accumulator.

The high-order byte of the sum moves to Port 2 and the low-order byte of the sum moves to Port 0.

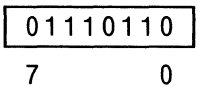
**Example:** *MOVC A,@A+PC* ; Look up value in table

Encoded Instruction:

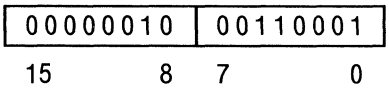



Before

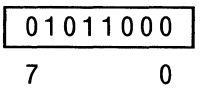
Accumulator




Program Counter

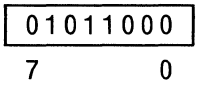



(02A8H)

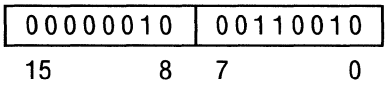



After

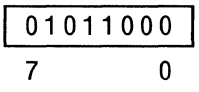
Accumulator




Program Counter




(02A8H)




**Notes:** 5, 12

## Move Accumulator to External Memory Addressed by Data Pointer

**Mnemonic:** MOVX

**Operands:**

| | |
|---|---|
| DPTR | Data Pointer |
| A | Accumulator |

**Format:** MOVX @DPTR,A

**Bit Pattern:**

| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** ((DPTR)) ← (A)

**Bytes:** 1
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction moves the contents of the accumulator to the off-chip data memory location addressed by the contents of the data pointer.

The high-order byte of the Data Pointer moves to Port 2, and the low-order byte of the Data Pointer moves to Port 0.

**Example:** *MOVX @DPTR,A* ; Move accumulator at data pointer

Encoded Instruction:

| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Before

Data Pointer

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | 8 | 7 | | | | | | | | 0 |

(3033H)

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Accumulator

| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

After

Data Pointer

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | | | | | | 8 | 7 | | | | | | | | 0 |

(3033H)

| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

Accumulator

| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Notes:** None

# MOVX

## Move Accumulator to External Memory Addressed by Register

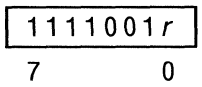
**Mnemonic:** MOVX

**Operands:**

| | |
|---|---|
| R*r* | Register 0 <= *r* <= 1 |
| A | Accumulator |

**Format:** MOVX @R*r*,A

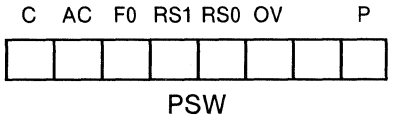
**Bit Pattern:**




**Operation:** ((R*r*)) ← (A)

**Bytes:** 1
**Cycles:** 2

**Flags:**

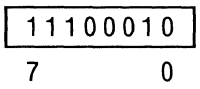



**Description:** This instruction moves the contents of the accumulator to the off-chip data memory location addressed by the contents of register *r*.

The contents of the specified register moves to Port 0. The contents of Port 2 is unaffected, but its previous value will be used in the address to off-chip data memory.

**Example:** MOV P2#0
*MOVX @R0,A* ; Move accumulator to indirect
; address

Encoded Instruction:




| Before | After |
|---|---|
| Register 0 | Register 0 |
| 10111000 | 10111000 |
| 7 0 | 7 0 |
| (00B8H) | (00B8H) |
| 10011001 | 01001100 |
| 7 0 | 7 0 |
| Accumulator | Accumulator |
| 01001100 | 01001100 |
| 7 0 | 7 0 |

**Notes:** None

# MOVX

## Move External Memory Addressed by Data Pointer to Accumulator

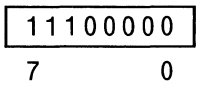
**Mnemonic:** MOVX

**Operands:**

| | |
|---|---|
| A | Accumulator |
| DPTR | Data Pointer |

**Format:** MOVX A,@DPTR

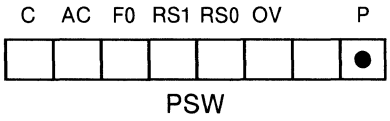
**Bit Pattern:**




**Operation:** (A) ← ((DPTR))

**Bytes:** 1
**Cycles:** 2

**Flags:**



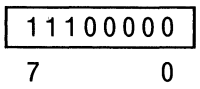

**Description:** This instruction moves the contents of the off-chip data memory location addressed by the data pointer to the accumulator.

The high-order byte of the Data Pointer moves to Port 2, and the low-order byte of the Data Pointer moves to Port 0.

**Example:**

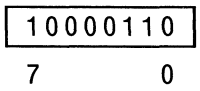
```
MOVX A,@DPTR      ; Move memory at DPTR to
                  ; accumulator
```

Encoded Instruction:

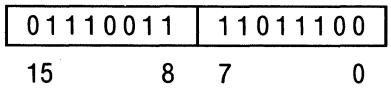



Before

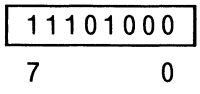
Accumulator




Data Pointer

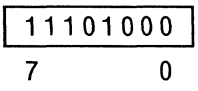



(73DCH)

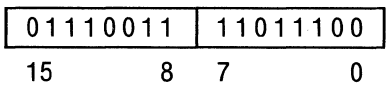



After

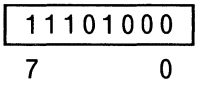
Accumulator




Data Pointer




(73DCH)




**Notes:** 5

# MOVX

## Move External Memory Addressed by Register to Accumulator

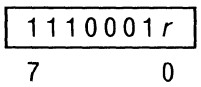
**Mnemonic:** MOVX

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** MOVX A,@R*r*

**Bit Pattern:**

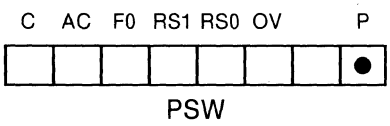



**Operation:** (A) ← ((R*r*))

**Bytes:** 1

**Cycles:** 2

**Flags:**




**Description:** This instruction moves the contents of the off chip data memory location addressed by register *r* to the accumulator.

The contents of the specified register moves to Port 0. The contents of Port 2 is unaffected, but its previous value will be used in the address to off-chip data memory.

**Example:** MOV P2, #55H
*MOVX A,@R1* ; Move memory at R1 to accumulator

Encoded Instruction:

| 1 1 1 0 0 0 1 1 |
|---|
| 7 0 |

| Before | After |
|---|---|
| Accumulator | Accumulator |
| 0 1 0 1 0 1 0 0 | 0 0 0 0 1 0 0 0 |
| 7 0 | 7 0 |
| Register 1 | Register 1 |
| 0 0 0 1 1 1 0 0 | 0 0 0 1 1 1 0 0 |
| 7 0 | 7 0 |
| (551CH) | (551CH) |
| 0 0 0 0 1 0 0 0 | 0 0 0 0 1 0 0 0 |
| 7 0 | 7 0 |

**Notes:** 5

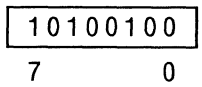
## Multiply Accumulator by B

**Mnemonic:** MUL

**Operands:** AB Multiply/Divide operand

**Format:** MUL AB

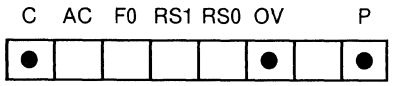
**Bit Pattern:**




**Operation:** (AB) ← (A) * (B)

**Bytes:** 1
**Cycles:** 4

**Flags:**



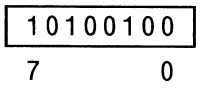

**Description:** This instruction multiplies the contents of the accumulator by the contents of the multiplication register (B). Both operands are treated as unsigned values. It places the low-order byte of the result in the accumulator, and places the high-order byte of the result in the multiplication register.

The carry flag is always cleared. If the high-order byte of the product is not 0, then the overflow flag is set; otherwise, it is cleared.

**Example:**

```
MOV B,#10        ; Move 10 to multiplication register
MUL AB           ; Multiply accumulator by 10
```

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00011111 (7 0) | 00110110 (7 0) |
| Multiplication Register (B) | Multiplication Register (B) |
| 00001010 (7 0) | 00000001 (7 0) |
| Overflow Flag | Overflow Flag |
| 0 | 1 |

**Notes:** 5

# NOP

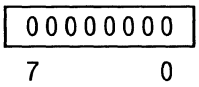
## No Operation

**Mnemonic:** NOP

**Operands:** None

**Format:** NOP

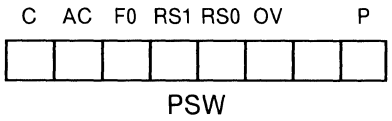
**Bit Pattern:**




**Operation:** No operation

**Bytes:** 1
**Cycles:** 1

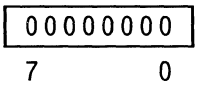
**Flags:**




**Description:** This instruction does absolutely nothing for one cycle. Control passes to the next sequential instruction.

**Example:** *NOP* ; Pause one cycle

Encoded Instruction:




**Notes:** None

# ORL

## Logical OR Immediate Data to Accumulator

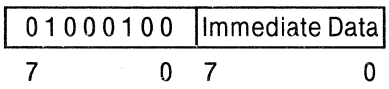
**Mnemonic:** ORL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | $-256 <= data <= +255$ |

**Format:** ORL A,#*data*

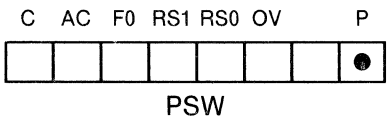
**Bit Pattern:**




**Operation:** (A) ← (A) OR *data*

**Bytes:** 2
**Cycles:** 1

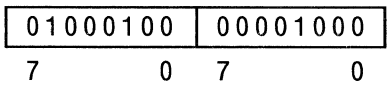
**Flags:**




**Description:** This instruction ORs the 8-bit immediate data value to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of either operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

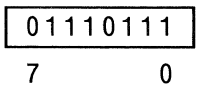
**Example:** *ORL A,#00001000B* ; Set bit 3 to 1

Encoded Instruction:



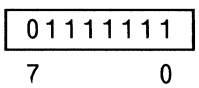

Before

Accumulator




After

Accumulator




**Notes:** 4, 5

## Logical OR Indirect Address to Accumulator

**Mnemonic:** ORL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** ORL A,@R*r*

**Bit Pattern:**

| 0 | 1 | 0 | 0 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← (A) OR ((R*r*))

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction ORs the contents of the memory location addressed by the contents of register *r* to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of either operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

**Example:** *ORL A,@R0* ; Set bit 0 to 1

Encoded Instruction:

| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | | After | |
|---|---|---|---|
| Accumulator | | Accumulator | |
| 0 | 0 1 0 1 0 0 0 | 0 | 0 1 0 1 0 0 1 |
| 7 | 0 | 7 | 0 |
| Register 0 | | Register 0 | |
| 0 | 1 0 1 0 0 1 0 | 0 | 1 0 1 0 0 1 0 |
| 7 | 0 | 7 | 0 |
| (52H) | | (52H) | |
| 0 | 0 0 0 0 0 0 0 1 | 0 | 0 0 0 0 0 0 0 1 |
| 7 | 0 | 7 | 0 |

**Notes:** 5, 15

# ORL

## Logical OR Register to Accumulator

**Mnemonic:** ORL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 7 |

**Format:** ORL A,R*r*

**Bit Pattern:**

| 01001*rrr* |
|---|
| 7 0 |

**Operation:** (A) ← (A) OR (R*r*)

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction ORs the contents of register *r* to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of either operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

**Example:** *ORL A,R4* ; Set bits 7 and 3 to 1

Encoded Instruction:

| 01001100 |
|---|
| 7 0 |

Before

Accumulator

| 10010001 |
|---|
| 7 0 |

Register 4

| 10001000 |
|---|
| 7 0 |

After

Accumulator

| 10011001 |
|---|
| 7 0 |

Register 4

| 10001000 |
|---|
| 7 0 |

**Note:** 5

## Logical OR Memory to Accumulator

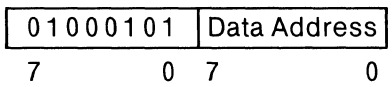
**Mnemonic:** ORL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data address* | 0 <= *data address* <= 255 |

**Format:** ORL A,*data address*

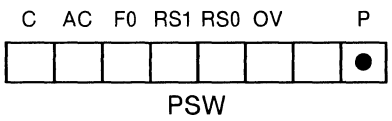
**Bit Pattern:**




**Operation:** (A) ← (A) OR (*data address*)

**Bytes:** 2
**Cycles:** 1

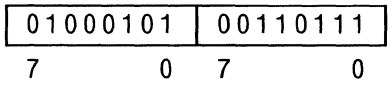
**Flags:**




**Description:** This instruction ORs the contents of the specified data address to the contents of the accumulator. Bit *n* of the result is 1 if bit *n* of either operand is 1; otherwise bit *n* is 0. It places the result in the accumulator.

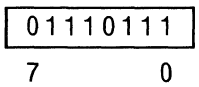
**Example:** *ORL A,37H* ; OR 37H with accumulator

Encoded Instruction:

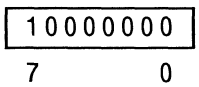



Before

Accumulator

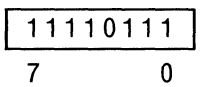



(37H)

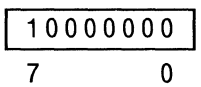



After

Accumulator




(37H)




**Notes:** 5, 8

# ORL

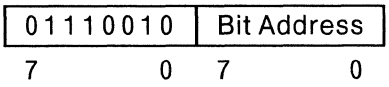
## Logical OR Bit to Carry Flag

**Mnemonic:** ORL

**Operands:** C Carry Flag
*bit address* 0 <= *bit address* <= 255

**Format:** ORL C,*bit address*

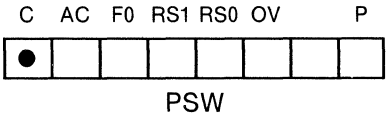
**Bit Pattern:**




**Operation:** (C) ← (C) OR (*bit address*)

**Bytes:** 2
**Cycles:** 2

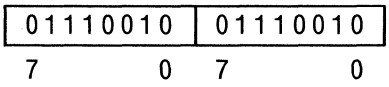
**Flags:**




**Description:** This instruction ORs the contents of the specified bit address with the contents of the carry flag. The carry flag becomes 1 when either the carry flag or the specified bit address is 1; otherwise, it is 0. It places the result in the carry flag.

**Example:** *ORL C,46.2* ; OR bit 2 of byte 46 with Carry

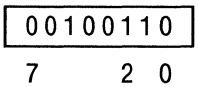
Encoded Instruction:




Before

Carry Flag

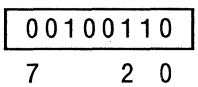

(46)




After

Carry Flag


(46)




**Notes:** None

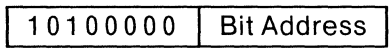
## Logical OR Complement of Bit to Carry Flag

**Mnemonic:** ORL

**Operands:** C Carry Flag
*bit address* 0 <= *bit address* <= 255

**Format:** ORL C/*bit address*

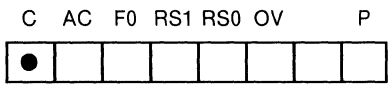
**Bit Pattern:**




**Operation:** (C) ← (C) OR NOT *bit address*

**Bytes:** 2
**Cycles:** 2

**Flags:**

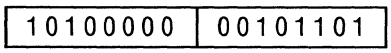



**Description:** This instruction ORs the complemented contents of the specified bit address to the contents of the carry flag. The carry flag is 1 when either the carry flag is already 1 or the specified bit address is 0. It places the result in the carry flag. The contents of the specified bit address is unchanged.

**Example:**

```
ORL C/25H.5        ; Complement contents of 5 in byte
                   ; 25H then OR with Carry
```

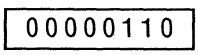
Encoded Instruction:




Before

Carry Flag

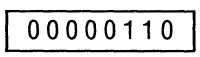

(25H)




After

Carry Flag


(25H)




**Notes:** None

# ORL

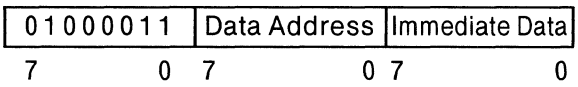
## Logical OR Immediate Data to Memory

**Mnemonic:** ORL

**Operands:** *data address* 0 <= *data address* <= 255
*data* −256 <= *data* <= +255

**Format:** ORL *data address*,#*data*

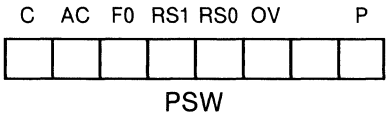
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) OR *data*

**Bytes:** 3
**Cycles:** 2

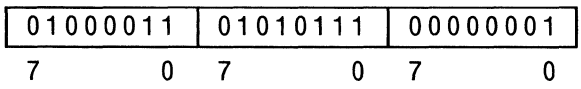
**Flags:**




**Description:** This instruction ORs the 8-bit immediate data value to the contents of the specified data address. Bit n of the result is 1 if bit n of either operand is 1; otherwise bit n is 0. It places the result in memory at the specified address.

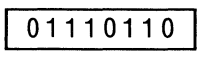
**Example:** *ORL 57H,#01H* ; Set bit 0 to 1

Encoded Instruction:



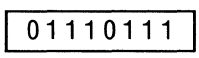

Before

(57H)




After

(57H)




**Notes:** 4, 9

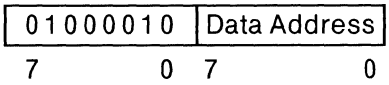
## Logical OR Accumulator to Memory

**Mnemonic:** ORL

**Operands:** *data address* 0 <= *data address* <= 255
A Accumulator

**Format:** ORL *data address*,A

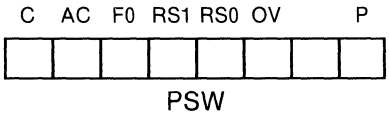
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) OR A

**Bytes:** 2
**Cycles:** 1

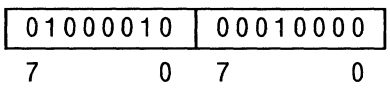
**Flags:**




**Description:** This instruction ORs the contents of the accumulator to the contents of the specified data address. Bit n of the result is 1 if bit n of either operand is 1; otherwise bit n is 0. It places the result in memory at the specified address.

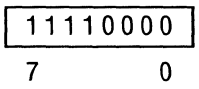
**Example:** *ORL 10H,A* ; OR accumulator with the contents
; of 10H

Encoded Instruction:

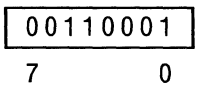



Before

Accumulator

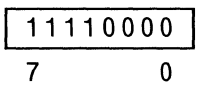



(10H)

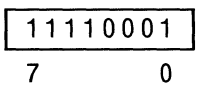



After

Accumulator




(10H)




**Note:** 9

# POP

## Pop Stack to Memory

**Mnemonic:** POP

**Operands:** *data address* 0 <= *data address* <= 255

**Format:** POP *data address*

**Bit Pattern:**

| 11010000 | Data Address |
|---|---|
| 7 0 | 7 0 |

**Operation:** (*data address*) ← ((SP))
(SP) ← (SP) –1

**Bytes:** 2
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction places the byte addressed by the stack pointer at the specified data address. It then decrements the stack pointer by 1.

**Example:** *POP PSW* ; Pop PSW parity is not affected.

Encoded Instruction:

| 11010000 | 11010000 |
|---|---|
| 7 0 | 7 0 |

| Before | After |
|---|---|
| Accumulator | Accumulator |
| 11010101 | 11010101 |
| 7 0 | 7 0 |
| PSW (0D0H) | PSW (0D0H) |
| 10101011 | 11110011 |
| 7 0 | 7 0 |
| Stack Pointer | Stack Pointer |
| 00010000 | 00001111 |
| 7 0 | 7 0 |
| (10H) | (10H) |
| 11110010 | 11110010 |
| 7 0 | 7 0 |

**Notes:** 2, 8, 17

# PUSH

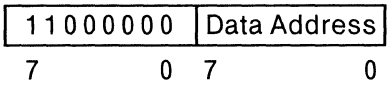
## Push Memory onto Stack

**Mnemonic:** PUSH

**Operands:** *data address* 0 <= *data address* <= 255

**Format:** PUSH *data address*

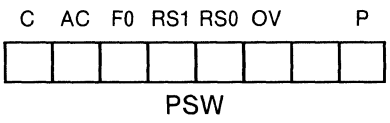
**Bit Pattern:**




**Operation:** (SP) ← (SP) + 1
((SP)) ← (*data address*)

**Bytes:** 2
**Cycles:** 2

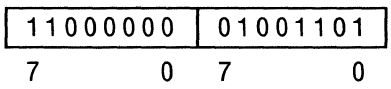
**Flags:**




**Description:** This instruction increments the stack pointer, then stores the contents of the specified data address at the location addressed by the stack pointer.

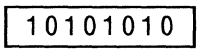
**Example:** *PUSH 4DH* ; Push one byte to the stack

Encoded Instruction:

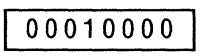



Before

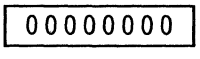
(4DH)




Stack Pointer

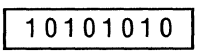



(11H)

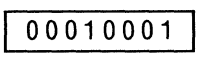



After

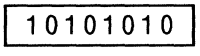
(4DH)




Stack Pointer




(11H)




**Notes:** 2, 3, 8

# RET

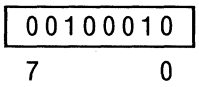
## Return from Subroutine (Non-interrupt)

**Mnemonic:** RET

**Operands:** None

**Format:** RET

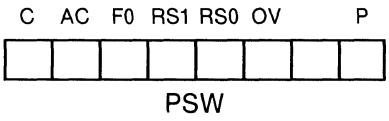
**Bit Pattern:**




**Operation:**
(PC *high*) ← ((SP))
(SP) ← (SP) –1
(PC *low*) ← ((SP))
(SP) ← (SP) –1

**Bytes:** 1
**Cycles:** 2

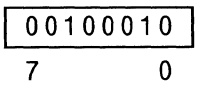
**Flags:**




**Description:** This instruction returns from a subroutine. Control passes to the location addressed by the top two bytes on the stack. The high-order byte of the return address is always the first to come off the stack. It is immediately followed by the low-order byte.

**Example:** *RET* ; Return from subroutine

Encoded Instruction:




| Before | After |
|---|---|
| Program Counter | Program Counter |
| 00000010 01010101 | 00000000 01110011 |
| 15 8 7 0 | 15 8 7 0 |
| Stack Pointer | Stack Pointer |
| 00001010 | 00001000 |
| 7 0 | 7 0 |
| (0AH) | (0AH) |
| 00000000 | 00000000 |
| 7 0 | 7 0 |
| (09H) | (09H) |
| 01110011 | 01110011 |
| 7 0 | 7 0 |

**Notes:** 2, 17

# RETI

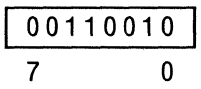
## Return from Interrupt Routine

**Mnemonic:** RETI

**Operands:** None

**Format:** RETI

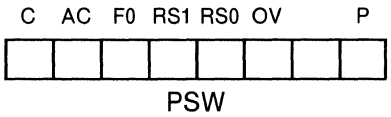
**Bit Pattern:**




**Operation:**
(PC *high*) ← ((SP))
(SP) ← (SP) –1
(PC *low*) ← ((SP))
(SP) ← (SP) –1

**Bytes:** 1
**Cycles:** 2

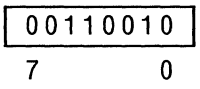
**Flags:**




**Description:** This instruction returns from an interrupt service routine, and reenables interrupts of equal or lower priority. Control passes to the location addressed by the top two bytes on the stack. The high-order byte of the return address is always the first to come off the stack. It is immediately followed by the low-order byte.

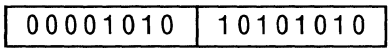
**Example:** *RETI* ; Return from interrupt routine

Encoded Instruction:

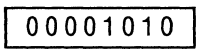



Before

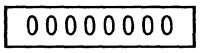
Program Counter



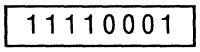

Stack Pointer




(0AH)

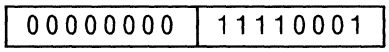



(09H)

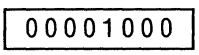



After

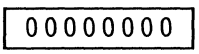
Program Counter



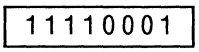

Stack Pointer




(0AH)




(09H)




**Notes:** 2, 17

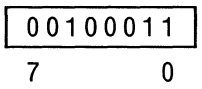
## Rotate Accumulator Left

**Mnemonic:** RL

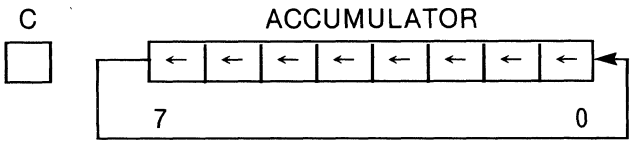
**Operands:** A Accumulator

**Format:** RL A

**Bit Pattern:**

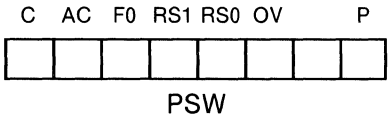



**Operation:**




**Bytes:** 1
**Cycles:** 1

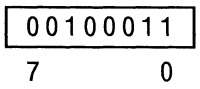
**Flags:**




**Description:** This instruction rotates each bit in the accumulator one position to the left. The most significant bit (bit 7) moves into the least significant bit position (bit 0).

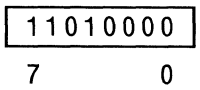
**Example:** *RL A* ; Rotate accumulator left one positon.

Encoded Instruction:



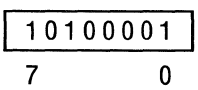

Before

Accumulator




After

Accumulator




**Notes:** None

## Rotate Accumulator and Carry Flag Left

**Mnemonic:** RLC

**Operands:** A Accumulator

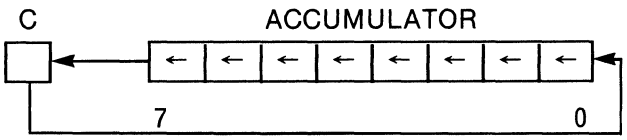
**Format:** RLC A

**Bit Pattern:**

| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

7 0

**Operation:**




**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | | | ● |

PSW

**Description:** This instruction rotates each bit in the accumulator one position to the left. The most significant bit (bit 7) moves into the Carry flag, while the previous contents of Carry moves into the least significant bit (bit 0).

**Example:** *RLC A* ; Rotate accumulator and carry left
; one positon.

Encoded Instruction:

| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

7 0

| Before | After |
|---|---|
| Accumulator | Accumulator |
| 0 0 0 1 1 0 0 1 | 0 0 1 1 0 0 1 1 |
| 7 0 | 7 0 |
| Carry Flag | Carry Flag |
| 1 | 0 |

**Note:** 5

# RR

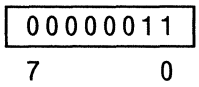
## Rotate Accumulator Right

**Mnemonic:** RR

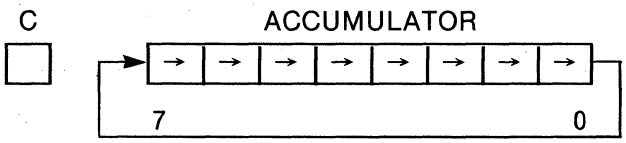
**Operands:** A Accumulator

**Format:** RR A

**Bit Pattern:**

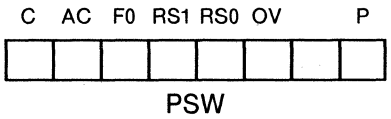



**Operation:**




**Bytes:** 1
**Cycles:** 1

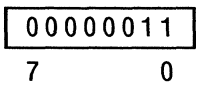
**Flags:**




**Description:** This instruction rotates each bit in the accumulator one position to the right. The least significant bit (bit 0) moves into the most significant bit position (bit 7).

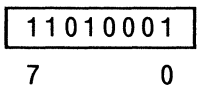
**Example:** *RR A* ; Rotate accumulator right one
; positon.

Encoded Instruction:



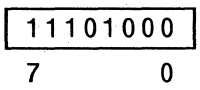

Before

Accumulator




After

Accumulator




**Notes:** None

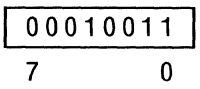
## Rotate Accumulator and Carry Flag Right

**Mnemonic:** RRC

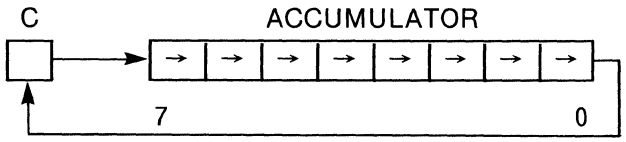
**Operands:** A Accumulator

**Format:** RRC A

**Bit Pattern:**

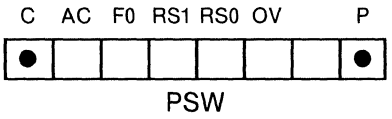



**Operation:**




**Bytes:** 1
**Cycles:** 1

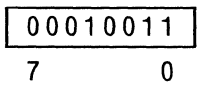
**Flags:**




**Description:** This instruction rotates each bit in the accumulator one position to the right. The least significant bit (bit 0) moves into the Carry flag, while the previous contents of Carry moves into the most significant bit (bit 7).

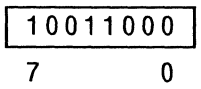
**Example:** *RRC A* ; Rotate accumulator and carry right
; one positon.

Encoded Instruction:




Before

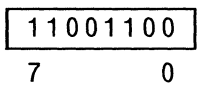
Accumulator




Carry Flag


After

Accumulator




Carry Flag


**Note:** 5

# SETB

## Set Carry Flag

**Mnemonic:** SETB

**Operands:** C Carry Flag

**Format:** SETB C

**Bit Pattern:**

| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (C) ← 1

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| ● | | | | | | | |

PSW

**Description:** This instruction sets the carry flag to 1.

**Example:** *SETB C* ; Set Carry to 1

Encoded Instruction:

| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | After |
|---|---|
| Carry Flag | Carry Flag |
| 0 | 1 |

**Notes:** None

# SETB

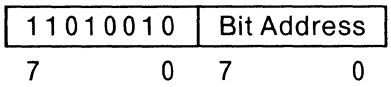
## Set Bit

**Mnemonic:** SETB

**Operands:** *bit address* 0 <= *bit address* <= 255

**Format:** SETB *bit address*

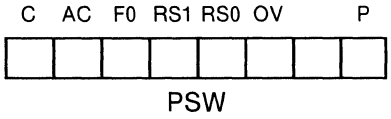
**Bit Pattern:**




**Operation:** (*bit address*) ← 1

**Bytes:** 2
**Cycles:** 1

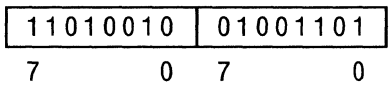
**Flags:**




**Description:** This instruction sets the contents of the specified bit address to 1.

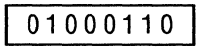
**Example:** *SETB 41.5* ; Set the contents of bit 5 in byte 41
; to 1

Encoded Instruction:



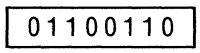

Before

(41)




After

(41)




**Notes:** None

# SJMP

## Short Jump

**Mnemonic:** SJMP

**Operands:** *code address*

**Format:** SJMP *code address*

**Bit Pattern:**

| 10000000 | Rel. Offset |
|---|---|
| 7 0 | 7 0 |

**Operation:** (PC) ← (PC) + 2
(PC) ← (PC) + *relative offset*

**Bytes:** 2
**Cycles:** 2

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

PSW

**Description:** This instruction transfers control to the specified code address. The Program Counter is incremented to the next instruction, then the relative offset is added to the incremented program counter, and the instruction at that address is executed.

**Example:**

```
        SJMP BOTTOM     ; Jump to BOTTOM
FF:INC A
        .
        .
        .
BOTTOM: RR A            ; (15 bytes ahead from the INC
                        ; instruction)
```

Encoded Instruction:

| 10000000 | 00001111 |
|---|---|
| 7 0 | 7 0 |

| Before | | After | |
|---|---|---|---|
| Program Counter | | Program Counter | |
| 11101000 | 11011100 | 11101000 | 11101101 |
| 15 8 | 7 0 | 15 8 | 7 0 |

**Notes:** 10, 11, 12

## Subtract Immediate Data from Accumulator with Borrow

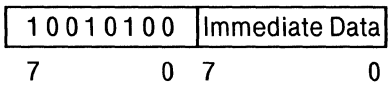
**Mnemonic:** SUBB

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | −256 <= *data* <= +255 |

**Format:** SUBB A,#*data*

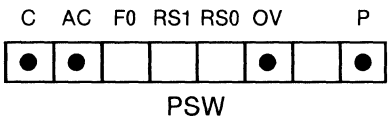
**Bit Pattern:**




**Operation:** (A) ← (A) − (C) − *data*

**Bytes:** 2
**Cycles:** 1

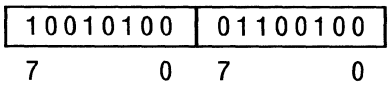
**Flags:**




**Description:** This instruction subtracts the contents of the Carry flag and the immediate data value from the contents of the accumulator. It places the result in the accumulator.

**Example:** *SUBB A,#0C1H* ; Subtract 0C1H from accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00100110 | 01100100 |
| 7 0 | 7 0 |
| Carry Flag | Carry Flag |
| 1 | 1 |
| Auxiliary Carry Flag | Auxiliary Carry Flag |
| 0 | 1 |
| Overflow Flag | Overflow Flag |
| 1 | 0 |

**Notes:** 4, 5, 6, 13, 14

# SUBB

## Subtract Indirect Address from Accumulator with Borrow

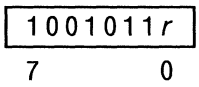
**Mnemonic:** SUBB

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** SUBB A,@R*r*

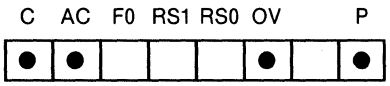
**Bit Pattern:**




**Operation:** (A) ← (A) – (C) – ((R*r*))

**Bytes:** 1
**Cycles:** 1

**Flags:**




PSW

**Description:** This instruction subtracts the Carry flag and the memory location addressed by the contents of register *r* from the contents of the accumulator. It places the result in the accumulator.

**Example:** *SUBB A,@R1* ; Subtract the indirect address from
; accumulator

Encoded Instruction:

| 1 0 0 1 0 1 1 1 |
|---|
| 7 0 |

| Before | After |
|---|---|
| Accumulator | Accumulator |
| 1 0 0 0 0 1 1 0 | 0 0 1 0 0 1 0 0 |
| 7 0 | 7 0 |
| Register 1 | Register 1 |
| 0 0 0 1 1 1 0 0 | 0 0 0 1 1 1 0 0 |
| 7 0 | 7 0 |
| (1CH) | (1CH) |
| 0 1 1 0 0 0 1 0 | 0 1 1 0 0 0 1 0 |
| 7 0 | 7 0 |
| Carry Flag | Carry Flag |
| 0 | 0 |
| Auxiliary Carry Flag | Auxiliary Carry Flag |
| 0 | 1 |
| Overflow Flag | Overflow Flag |
| 0 | 1 |

**Notes:** 5, 6, 13, 14, 15

# SUBB

## Subtract Register from Accumulator with Borrow

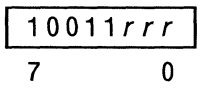
**Mnemonic:** SUBB

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 7 |

**Format:** SUBB A,R*r*

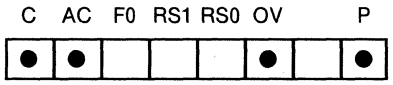
**Bit Pattern:**




**Operation:** (A) ← (A) – (C) – (R*r*)

**Bytes:** 1
**Cycles:** 1

**Flags:**

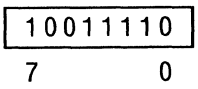



PSW

**Description:** This instruction subtracts the contents of the Carry flag and the contents of register *r* from the contents of the accumulator. It places the result in the accumulator.

**Example:** *SUBB A,R6* ; Subtract R6 from accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 01110110 | 11110000 |
| 7 0 | 7 0 |
| R6 | R6 |
| 10000101 | 10000101 |
| 7 0 | 7 0 |
| Carry Flag | Carry Flag |
| 1 | 1 |
| Auxiliary Carry Flag | Auxiliary Carry Flag |
| 0 | 1 |
| Overflow Flag | Overflow Flag |
| 0 | 1 |

**Notes:** 5, 6, 13, 14

# SUBB

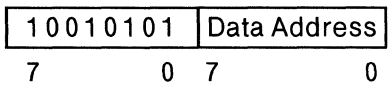
## Subtract Memory from Accumulator with Borrow

**Mnemonic:** SUBB

**Operands:** A Accumulator
*data address* 0 <= *data address* <= 255

**Format:** SUBB A,*data address*

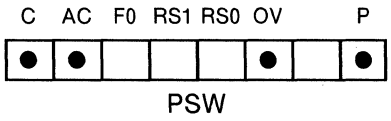
**Bit Pattern:**




**Operation:** (A) ← (A) – (C) – (*data address*)

**Bytes:** 2
**Cycles:** 1

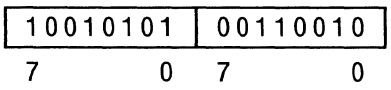
**Flags:**




**Description:** This instruction subtracts the contents of the Carry flag and the contents of the specified address from the contents of the accumulator. It places the result in the accumulator.

**Example:** *SUBB A,32H* ; Subtract 32H in memory from
; accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00100110 | 11010010 |
| 7 0 | 7 0 |
| (32H) | (32H) |
| 01010011 | 01010011 |
| 7 0 | 7 0 |
| Carry Flag | Carry Flag |
| 1 | 1 |
| Auxiliary Carry Flag | Auxiliary Carry Flag |
| 0 | 1 |
| Overflow Flag | Overflow Flag |
| 0 | 0 |

**Notes:** 5, 6, 8, 13, 14

# SWAP

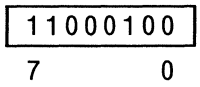
## Exchange Nibbles in Accumulator

**Mnemonic:** SWAP

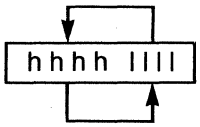
**Operands:** A Accumulator

**Format:** SWAP A

**Bit Pattern:**

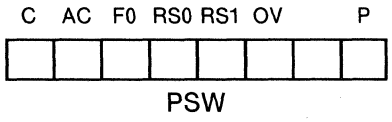



**Operation:**




**Bytes:** 1
**Cycles:** 1

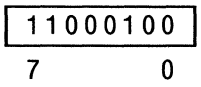
**Flags:**




**Description:** This instruction exchanges the contents of the low order nibble (0-3) with the contents of the high order nibble (4-7).

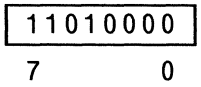
**Example:** *SWAP A* ; Swap high and low nibbles in the
; accumulator.

Encoded Instruction:



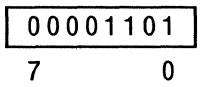

Before

Accumulator




After

Accumulator




**Notes:** None

# XCH

## Exchange Indirect Address with Accumulator

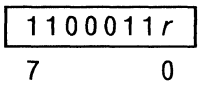
**Mnemonic:** XCH

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** XCH A,@R*r*

**Bit Pattern:**

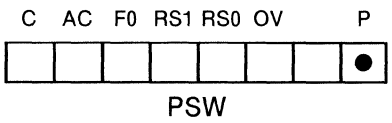



**Operation:**

temp ← ((R*r*))
((R*r*)) ← (A)
(A) ← temp

**Bytes:** 1
**Cycles:** 1

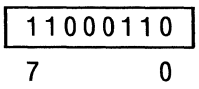
**Flags:**




**Description:** This instruction exchanges the contents of the memory location addressed by the contents of register *r* with the contents of the accumulator.

**Example:** *XCH A,@R0* ; Exchange the accumulator with ; memory

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 00111111 | 00011101 |
| Register 0 | Register 0 |
| 01010010 | 01010010 |
| (52H) | (52H) |
| 00011101 | 00111111 |

**Notes:** 5, 15

# XCH

## Exchange Register with Accumulator

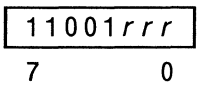
**Mnemonic:** XCH

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 7 |

**Format:** XCH A,R*r*

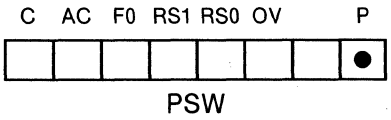
**Bit Pattern:**




**Operation:**
temp ← (R*r*)
(R*r*) ← (A)
(A) ← temp

**Bytes:** 1
**Cycles:** 1

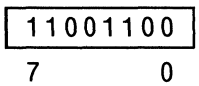
**Flags:**




**Description:** This instruction exchanges the contents of register *r* with the contents of the accumulator.

**Example:** *XCH A,R6* ; Exchange register 6 with the ; accumulator

Encoded Instruction:




| Before | After |
|---|---|
| Accumulator | Accumulator |
| 10011001 | 10000000 |
| 7 0 | 7 0 |
| Register 6 | Register 6 |
| 10000000 | 10011001 |
| 7 0 | 7 0 |

**Note:** 5

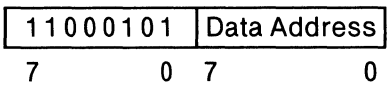
## Exchange Memory with Accumulator

**Mnemonic:** XCH

**Operands:** A Accumulator
*data address* 0 <= *data address* <= 255

**Format:** XCH A,*data address*

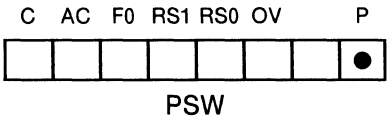
**Bit Pattern:**




**Operation:** temp ← (*data address*)
(*data address*) ← (A)
(A) ← temp

**Bytes:** 2
**Cycles:** 1

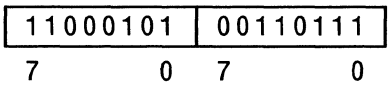
**Flags:**




**Description:** This instruction exchanges the contents of the specified data address with the contents of the accumulator.

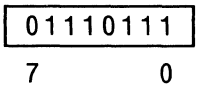
**Example:** *XCH A,37H* ; Exchange accumulator with the
; contents of location 37H

Encoded Instruction:

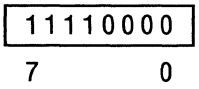



Before

Accumulator

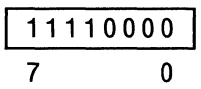



(37H)

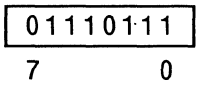



After

Accumulator




(37H)




**Notes:** 5, 9

# XCHD

## Exchange Low Nibbles (Digits) of Indirect Address with Accumulator

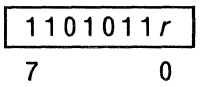
**Mnemonic:** XCHD

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 1 |

**Format:** XCHD A,@R*r*

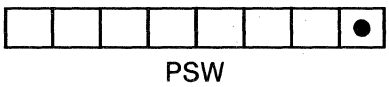
**Bit Pattern:**




**Operation:**
temp ← ((R*r*)) 0-3
((R*r*)) 0-3 ← (A) 0-3
(A) 0-3 ← temp

**Bytes:** 1
**Cycles:** 1

**Flags:**

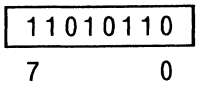



PSW

**Description:** This instruction exchanges the contents of the low order nibble (bits 0-3) of the memory location addressed by the contents of register *r* with the contents of the low order nibble (bits 0-3) of the accumulator.

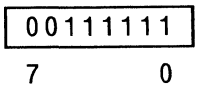
**Example:** *XCHD A,@R0* ; Exchange the accumulator with
; memory

Encoded Instruction:

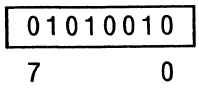



Before

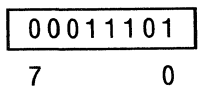
Accumulator




Register 0

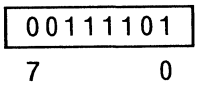



(52H)

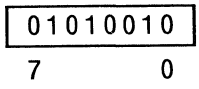



After

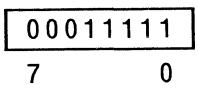
Accumulator




Register 0




(52H)




**Notes:** 5, 15

# XRL

## Logical Exclusive OR Immediate Data to Accumulator

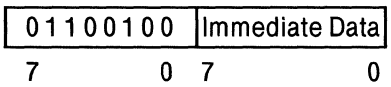
**Mnemonic:** XRL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data* | −256 <= *data* <= +255 |

**Format:** XRL A,#*data*

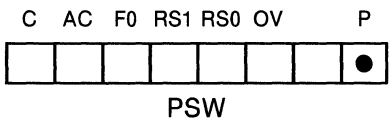
**Bit Pattern:**




**Operation:** (A) ← (A) XOR *data*

**Bytes:** 2
**Cycles:** 1

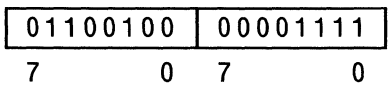
**Flags:**




**Description:** This instruction exclusive ORs the immediate data value to the contents of the accumulator. Bit *n* of the result is 0 if bit *n* of the accumulator equals bit *n* of the data value; otherwise bit *n* is 1. It places the result in the accumulator.

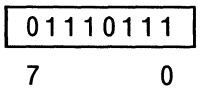
**Example:** *XRL A,#0FH* ; Complement the low order nibble

Encoded Instruction:



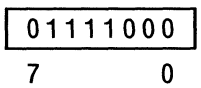

Before

Accumulator




After

Accumulator




**Notes:** 4, 5

## Logical Exclusive OR Indirect Address to Accumulator

**Mnemonic:** XRL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | 0 <= R*r* <= 1 |

**Format:** XRL A,@R*r*

**Bit Pattern:**

| 0 | 1 | 1 | 0 | 0 | 1 | 1 | *r* |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

**Operation:** (A) ← (A) XOR ((R*r*))

**Bytes:** 1
**Cycles:** 1

**Flags:**

| C | AC | F0 | RS1 | RS0 | OV | | P |
|---|---|---|---|---|---|---|---|
| | | | | | | | ● |

PSW

**Description:** This instruction exclusive ORs the contents of the memory location addressed by the contents of register *r* to the contents of the accumulator. Bit *n* of the result is 0 if bit *n* of the accumulator equals bit *n* of the addressed location; otherwise bit *n* is 1. It places the result in the accumulator.

**Example:**

```
XRL A,@R0          ; XOR indirect address with
                   ; accumulator
```

Encoded Instruction:

| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 7 | | | | | | | 0 |

| Before | | After | |
|---|---|---|---|
| Accumulator | | Accumulator | |
| 0 | 0 1 0 1 0 0 0 | 0 | 0 1 0 1 0 0 1 |
| 7 | 0 | 7 | 0 |
| Register 0 | | Register 0 | |
| 0 | 1 0 1 0 0 1 0 | 0 | 1 0 1 0 0 1 0 |
| 7 | 0 | 7 | 0 |
| (52H) | | (52H) | |
| 0 | 0 0 0 0 0 0 1 | 0 | 0 0 0 0 0 0 1 |
| 7 | 0 | 7 | 0 |

**Notes:** 5, 15

## Logical Exclusive OR Register to Accumulator

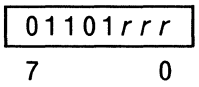
**Mnemonic:** XRL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| R*r* | Register 0 <= *r* <= 7 |

**Format:** XRL A,R*r*

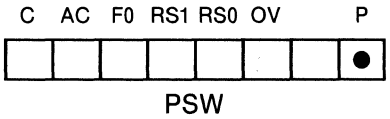
**Bit Pattern:**




**Operation:** (A) ← (A) XOR (R*r*)

**Bytes:** 1
**Cycles:** 1

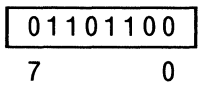
**Flags:**




**Description:** This instruction exclusive ORs the contents of register *r* to the contents of the accumulator. Bit *n* of the result is 0 if bit *n* of the accumulator equals bit *n* of the specified register; otherwise bit *n* is 1. It places the result in the accumulator.

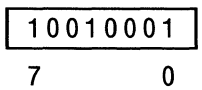
**Example:** *XRL A,R4* ; XOR R4 with accumulator

Encoded Instruction:

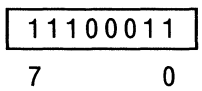



Before

Accumulator

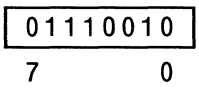



Register 4

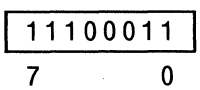



After

Accumulator




Register 4




**Note:** 5

## Logical Exclusive OR Memory to Accumulator

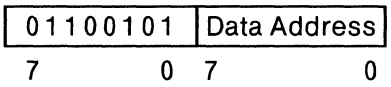
**Mnemonic:** XRL

**Operands:**

| | |
|---|---|
| A | Accumulator |
| *data address* | 0 <= *data address* <= 255 |

**Format:** XRL A,*data address*

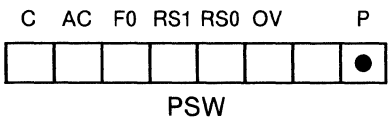
**Bit Pattern:**




**Operation:** (A) ← (A) XOR (*data address*)

**Bytes:** 2
**Cycles:** 1

**Flags:**

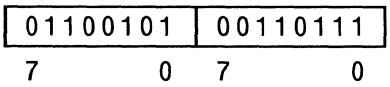



**Description:** This instruction exclusive ORs the contents of the specified data address to the contents of the accumulator. Bit *n* of the result is 0 if bit *n* of the accumulator equals bit *n* of the addressed location; otherwise bit *n* is 1. It places the result in the accumulator.

**Example:**

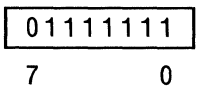
```
XRL A,37H          ; XOR the contents of location 37H
                   ; with accumulator
```

Encoded Instruction:

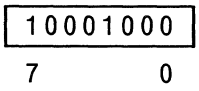



Before

Accumulator

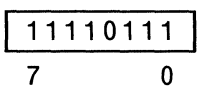



(37H)

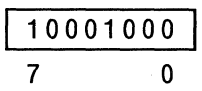



After

Accumulator




(37H)




**Notes:** 4, 8

# XRL

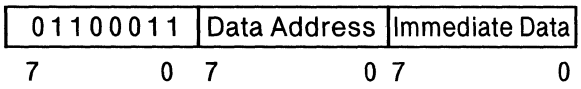
## Logical Exclusive OR Immediate Data to Memory

**Mnemonic:** XRL

**Operands:** *data address* 0 <= *data address* <= 255
*data* −256 <= *data* <= +255

**Format:** XRL *data address*,#*data*

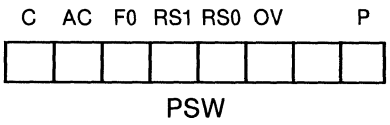
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) XOR *data*

**Bytes:** 3
**Cycles:** 2

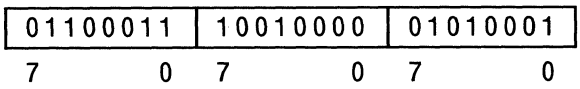
**Flags:**




**Description:** This instruction exclusive ORs the immediate data value to the contents of the specified data address. Bit *n* of the result is 0 if bit *n* of the specified address equals bit *n* of the data value; otherwise, bit *n* is 1. It places the result in data memory at the specified address.

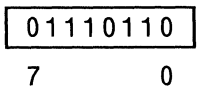
**Example:** *XRL P1,#51H* ; XOR 51H with the contents of Port 1

Encoded Instruction:



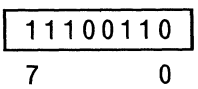

Before

Port 1 (90H)




After

Port 1 (90H)




**Notes:** 4, 9

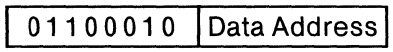
## Logical Exclusive OR Accumulator to Memory

**Mnemonic:** XRL

**Operands:** *data address* 0 <= *data address* <= 255
A Accumulator

**Format:** XRL *data address*,A

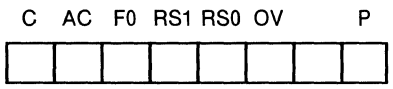
**Bit Pattern:**




**Operation:** (*data address*) ← (*data address*) XOR A

**Bytes:** 2
**Cycles:** 1

**Flags:**

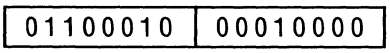



PSW

**Description:** This instruction exclusive ORs the contents of the accumulator to the contents of the specified data address. Bit *n* of the result is 0 if bit *n* of the accumulator equals bit *n* of the specified address; otherwise bit *n* is 1. It places the result in data memory at the specified address.

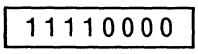
**Example:** *XRL 10H,A* ; XOR the contents of 10H with the
; accumulator

Encoded Instruction:

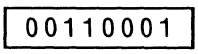



Before

Accumulator

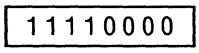



(10H)

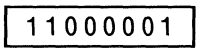



After

Accumulator




(10H)




**Note:** 9

## Notes

1. The low-order byte of the Program Counter is always placed on the stack first, followed by the high order byte.
2. The Stack Pointer always points to the byte most recently placed on the stack.
3. On the 8051 the contents of the Stack Pointer should never exceed 127. If the stack pointer exceeds 127, data pushed on the stack will be lost, and undefined data will be returned. The Stack Pointer will be incremented normally even though data is not recoverable.
4. The expression used as the data operand must evaluate to an eight-bit number. This limits the range of possible values in assembly time-expressions to between −256 and +255 inclusive.
5. The Parity Flag, PSW.0, always shows the parity of the accumulator. If the number of 1's in the accumulator is odd, the parity flag is 1; otherwise, the parity flag will be 0.
6. All addition operations affect the Carry Flag, PSW.7, and the Auxiliary Carry flag, PSW.6. The Carry flag receives the carry out from the bit 7 position (Most Significant Bit) in the accumulator. The Auxiliary Carry flag receives the carry out from the bit 3 position. Each is either set or cleared with each ADD operation.
7. The overflow flag (OV) is set when an operation produces an erroneous result (i.e. the sum of two negative numbers is positive, or the sum of two positive numbers is negative). OV is updated with each operation.
8. If one of the I/O ports is specified by the data address, then data will be taken from the port input pins.
9. If one of the I/O ports is specified by the data address, then data will be taken from, and returned to, the port latch.
10. The *code address* operand must be within the range of −128 and +127 inclusive of the incremented program counter's value.
11. The last byte of the encoded instruction is treated as a two's complement number, when it is added to the program counter.
12. The Program Counter is always incremented before the add.
13. The auxiliary carry flag is set if there is a borrow from bit 3 of the accumulator; otherwise, it is cleared.
14. The overflow flag (OV) is set when an operation produces an erroneous result (i.e. a positive number is subtracted from a negative number to produce a positive result, or a negative number is subtracted from a positive number to produce a negative result). OV is cleared with each correct operation.
15. On the 8051 the contents of the register used in the indirect address should not exceed 127. When the contents of the register is 128 or greater, source operands will return undefined data, and destination operands will cause data to be lost. In either case, the program will continue with no change in execution time or control flow.
16. If an I/O port is specified as the source operand, then the the port pins will be read. If an I/O port is the destination operand, then the port latch will receive the data.
17. If the stack pointer is 128 or greater, then invlalid data will be returned on a POP or return.

# CHAPTER 4
# ASSEMBLER DIRECTIVES

This chapter describes all of the assembler directives. It shows how to define symbols and how to control the placement of code and data in program memory.

## Introduction

The MCS-51 assembler has several directives that permit you to set symbol values, reserve and initialize storage space, and control the placement of your code.

The directives should not be confused with instructions. They do not produce executable code, and with the exception of the DB and DW directives, they have no effect on the contents of code memory. Their sole purpose is to change the state of the assembler.

The directives can be broken down into the following categories:

Symbol Definition
- EQU
- SET
- DATA
- XDATA
- BIT

Segment Controls
- BSEG
- CSEG
- DSEG
- XSEG

Location Counter Controls
- ORG
- DS
- DBIT

Memory Initialization
- DB
- DW

End of Program
- END

## The Location Counter

The location counter in ASM51 is an index to the address space of the active segment. When a segment is first activated the location counter is 0. You can change its value with the location counter control directives (ORG, DS, or DBIT). In the code address segment the memory initialization directives (DB and DW) and each instruction assembled change the location counter's value. If you change segment modes and later return to the segment, the location counter is restored to its previous value. Whenever the assembler encounters a label, it assigns the current value of the location counter to that symbol, and the segment type of the current segment mode.

You can use the value of the active segment's location counter with the symbol dollar sign ($). This can be especially useful when expressing code address operands for jump instructions. When you use the location counter symbol, keep in mind that its value changes with each instruction, but only after that instruction has been completely evaluated. If you use $ in an operand to an instruction or a directive, it represents the code address of the first byte of that instruction.

```
MOV R6,#10    ; Load register 6 with the value 10
DJNZ R6,$     ; Loop at current location
              ; until R6 is 0 (20 instruction cycles)

DIV AB        ; Divide accumulator by multiplication register
JZ $+3        ; Jump over next instruction if accumulator is 0
RET           ; Return if accumulator is not 0

CLR C         ; Set carry to 0 for loop termination
RLC A         ; Find left most 0 bit in accumulator
JC $-1        ; Jump to previous instruction if Carry is high
```

## Symbol Names

A symbol name may be composed of any of the following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
? _
```

A symbol cannot begin with a digit (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9). The remainder of the name can contain any of the legal characters.

You can use up to 255 characters in a symbol name, but only the first 31 characters are significant. A symbol name may contain upper case or lower case characters, but the assembler converts to upper case characters for internal representation. So to ASM51, 'buffer' is the same as 'BUFFER' and

'_A_THIRTY_ONE_CHARACTER_STRING_'

is the same as

'_A_THIRTY_ONE_CHARACTER_STRING_PLUS_THIS'.

The following list of instruction mnemonics, assembly-time operators, reserved words, predefined bit and data addresses, and assembler directives may not be used as user defined symbol names.

**Instruction Mnemonics**

| | | | | | |
|---|---|---|---|---|---|
| ACALL | DA | JMP | MOV | PUSH | SETB |
| ADD | DEC | JNB | MOVC | RET | SJMP |
| ADDC | DIV | JNC | MOVX | RETI | SUBB |
| AJMP | DJNZ | JNZ | MUL | RL | SWAP |
| ANL | INC | JZ | NOP | RLC | XCH |
| CJNE | JB | LCALL | ORL | RR | XCHD |
| CLR | JBC | LJMP | POP | RRC | XRL |
| CPL | JC | | | | |

**Operators**

| | | | | | |
|---|---|---|---|---|---|
| AND | GT | LOW | NE | OR | SHR |
| EQ | HIGH | LT | NOT | SHL | XOR |
| GE | LE | MOD | | | |

**Reserved Words**

| | | | | | |
|---|---|---|---|---|---|
| A | DPTR | R0 | R2 | R4 | R6 |
| AB | PC | R1 | R3 | R5 | R7 |
| C | | | | | |

**Predefined Data (byte) Addresses**

| | | | | | |
|---|---|---|---|---|---|
| ACC | IE | P2 | SBUF | TCON | TL0 |
| B | IP | P3 | SCON | TH0 | TL1 |
| DPH | P0 | SP | TH1 | TMOD | DPL |
| P1 | PSW | | | | |

**Predefined Bit Addresses**

| | | | | | |
|---|---|---|---|---|---|
| AC | EX1 | IT1 | PX1 | RXD | TF0 |
| CY | F0 | OV | RB8 | SM0 | TF1 |
| EA | IE0 | P | RD | SM1 | TI |
| ES | IE1 | PS | REN | SM2 | TR0 |
| ET0 | INT0 | PT0 | RI | T0 | TR1 |
| ET1 | INT1 | PT1 | RS0 | T1 | TXD |
| EX0 | IT0 | PX0 | RS1 | TB8 | WR |

**Predefined Code Addresses**

| | | | | | |
|---|---|---|---|---|---|
| RESET | TIMER0 | TIMER1 | EXTI0 | EXTI1 | SINT |

**Directives**

| | | | | | |
|---|---|---|---|---|---|
| BIT | DATA | DB | EQU | ORG | XDATA |
| BSEG | DSEG | DW | END | SET | XSEG |
| CSEG | DBIT | DS | | | |

## Statement Labels

A label is a symbol. All of the rules for forming symbol names apply to labels. A statement label is the first field in a line, but it may be preceded by any number of tabs or spaces. You must place a colon (:) after the label to identify it as a label. Only one label is permitted per line. If you use the label later in an expression, do not include the colon, since the colon is used only to define the label.

Any line in your program can have a label, except control lines and some directives. The following directives may not have a label.

```
BIT
DATA
END
EQU
ORG
SET
XDATA
```

When a label is defined, it receives a numeric value and a segment type. The numeric value will always be the current value of the location counter. If a label appears on an instruction line, its numeric value will be the code address of the first byte of the instruction. The segment type will depend on which segment is active when the label is defined. Segment typing of labels and symbols helps to prevent confusing the different address spaces on the 8051. Several examples of lines containing labels are shown below.

```
LABEL1:
LABEL2:  ; This line contains no instruction
LABEL3: DB 27,33,'FIVE'
LABEL4: MOV DPTR,#LABEL3
```

You can use labels like any other symbol, as a code address or a numeric value in an assembly-time expression (described in Chapter 2 Operands). A label, once defined, may not be redefined.

## Symbol Definition Directives

The symbol definition directives allow you to create symbols that can be used to enhance the readability of your code. With these directives you can define symbols to represent data addresses, bit addresses, external data addresses, numeric values to be used in assembly-time expressions, and even special assembler symbols.

### EQU Directive

The format for the EQU directive is shown below. Note that a label is not permitted.

*symbol name* EQU *expression or special assembler symbol*

The EQU directive assigns a numeric value or special assembler symbol to a specified symbol name. The symbol name must be a valid ASM51 symbol as described above. If you assign a constant or address expression to the symbol, the expression must be a valid assembly-time expression with no forward references. The special assembler symbols A, R0, R1, R2, R3, R4, R5, R6, and R7 can be represented by user symbols defined with the EQU directive.

If you use EQU to define a symbol, that symbol will have no segment type associated with it. You can use it as a data address, code address, bit address, or external data address, without error. If you define a symbol to a register value, it will have a type, 'REG'. It can only be used in the place of that register in instruction operands. A symbol defined by the EQU directive cannot be defined anywhere else.

The following examples show several uses of EQU:

```
ACCUM   EQU   A       ;  Define ACCUM to stand for A (accumulator)
N27     EQU   27      ;  Set N27 to equal 27
PI      EQU   3       ;  This program was written in Louisiana
                      ;  where PI equals 3 by law
HERE    EQU   $       ;  Set HERE to current location counter value
```

## SET Directive

The format for the SET directive is shown below. Note that a label is not permitted.

*symbol name* SET *expression or special assembler symbol*

The SET directive assigns a numeric value or special assembler symbol to a specified symbol name. The symbol name must be a valid ASM51 symbol as described above. If you assign a constant or address expression to the symbol, the expression must be a valid assembly-time expression with no forward references. The special assembler symbols A, R0, R1, R2, R3, R4, R5, R6, and R7 can be represented by user symbols defined with the SET directive.

If you use SET to define a symbol, that symbol will have no segment type associated with it. You can use it as a data address, code address, bit address, or external data address, without error. If you define a symbol to a register value, it will have a type, 'REG'. It can only be used in the place of that register in instruction operands. A symbol defined by the SET directive may be redefined by another SET directive.

The following examples show several uses of SET:

```
COUNT    SET 0          ;  Initialize Assembly-time counter
COUNT    SET COUNT+1    ;  Increment assembly time counter
HALF     SET WHOLE/2    ;  Give half of WHOLE to HALF (the
                        ;  remainder is discarded)
H20      SET 32         ;  Set H20 to 32
INDIRECT SET R1         ;  Set INDIRECT to R1
```

## DATA Directive

The format for the DATA directive is shown below. Note that a label is not permitted.

*symbol name* DATA *expression*

The DATA directive assigns an on-chip data address to the specified symbol name. The symbol name must be a valid ASM51 symbol as described above. The expression must be a valid assembly-time expression with no forward references, and it must compute to a data address (0-255).

When you define a symbol with DATA, it will be of segment type DSEG. A symbol defined by the DATA directive may not be redefined anywhere else in the program.

The following examples show several uses of DATA:

```
CONIN       DATA  SBUF    ;  Define CONIN to address the serial port buffer
TABLE_BASE  DATA  70H     ;  Define TABLE_BASE to be at location 70H
TABLE_END   DATA  7FH     ;  Define TABLE_END to be at top of RAM (7FH)
```

## XDATA Directive

The format for the XDATA directive is shown below. Note that a label is not permitted.

*symbol name* XDATA *expression*

The XDATA directive assigns an off-chip data address to the specified symbol name. The symbol name must be a valid ASM51 symbol as described above. The expression must be a valid assembly-time expression with no forward references.

When you define a symbol with XDATA, it will be of segment type XSEG. A symbol defined by the XDATA directive may not be redefined anywhere else in the program.

The following examples show several uses of XDATA:

```
DATE    XDATA  999H      ; Define DATE to be 999H
TIME    XDATA  DATE+5    ; Define TIME to be 5 bytes after DATE
PLACE   XDATA  TIME+3    ; Define PLACE to be 3 bytes after TIME
```

## BIT Directive

The format for the BIT directive is shown below. Note that a label is not permitted.

*symbol name* BIT *bit address*

The BIT directive assigns a bit address to the specified symbol name. The symbol name must be a valid ASM51 symbol as described above. The bit address must be a valid bit address (described in Chapter 2) with no forward references.

When you define a symbol with BIT, it will be of segment type BSEG. A symbol defined by the BIT directive may not be defined anywhere else in the program.

The following examples show several uses of BIT:

```
ERROR_FLAG  BIT 25H.3   ; Define ERROR_FLAG in RAM
                        ; bit address space
                        ; (bit 3 of byte 25H)
ARITH_ERR   BIT OV      ; Define ARITH_ERR to the same
                        ; address as the predefined bit
                        ; address OV (0D2H)
```

# Memory Segment Controls

There are four address spaces or segments in the 8051's architecture: code address space, data address space, external data address space, and bit address space (overlayed on the data address space). Each has its own characteristics and limits. When a segment is first activated its location counter is 0. The code address space is the ROM memory both on- and off-chip. The other areas cannot be initialized, but your program's use of these areas can be controlled with the location counter control directives.

Each address space has its own location counter; it is active only when the corresponding segment is also active. If you change segments and return to a previously used segment, the value of the location counter will be restored to the value it had when you left it.

### BSEG Directive

The format for the BSEG directive is as follows:

[*label*:] BSEG

The BSEG directive selects the 8051's bit address segment. Each BSEG will restore the location counter to the value it had when the bit segment was last active. The bit address segment's location counter can be altered with the ORG and DBIT directives. Each unit of the location counter stands for a bit in the bit address space. The location counter may assume values in the range of 0 to 255.

### CSEG Directive

The format for the CSEG directive is as follows:

[*label*:] CSEG

The CSEG directive selects the 8051's code address segment. This is the default segment when the assembler is invoked. Each CSEG will restore the location counter to the value it had when the code segment was last active. The code address segment's location counter can be altered with the ORG, DS, DB, and DW directives, and with each instruction encoded. Each unit in the location counter stands for one byte in the code address space. The location counter may assume values in the range of 0 to 65,535.

### DSEG Directive

The format for the DSEG directive is as follows:

[*label*:] DSEG

The DSEG directive selects the 8051's on-chip data address segment. Each DSEG will restore the location counter to the value it had when the data segment was last active. The data address segment's location counter can be altered with the ORG and DS directives. Each unit in the location counter stands for one byte in the on-chip data address space. The location counter may assume values in the range of 0 to 255.

### XSEG Directive

The format for the XSEG directive is as follows:

[*label*:] XSEG

The XSEG directive selects the 8051's external data address segment. Each XSEG will restore the location counter to the value it had when the external data segment was last acitve. The external data address segment's location counter can be altered with the ORG and DS directives. Each unit in the location counter stands for one byte in the off-chip data address space. The location counter may assume values in the range of 0 to 65,535.

## Location Counter Controls

There are three directives that alter the location counter of the current address space segment: ORG, DS, and DBIT. The DBIT directive can be used only when the bit address segment is active. The DS directive can be used in any address segment except the bit address segment. The ORG directive can be used in any segment.

## ORG Directive

The format of the ORG directive is as follows:

ORG *expression*

The ORG directive may be used in any segment, but the value and segment type of its expression must conform to the limitations of that segment. The expression must be a valid assembly-time expression with no forward references.

When the ORG expression is encountered in a program, the value in the expression is computed and assigned to the location counter of the current segment. There can be several ORG directives in each segment, and they do not have to be in ascending order. But, if you use non-sequential ORG statements, overlap problems may result. It is the programmer's responsibility to guard against this occurrence. The ORG statement may not have a label.

## DS Directive

The format of the DS directive is as follows:

[*label*:] DS *expression*

The DS directive reserves space in byte units. It can be used in any segment except the bit address segment. The expression must be a valid assembly-time expression with no forward references. When a DS statement is encountered in a program, the location counter of the current segment is incremented by the value of the expression. The sum of the location counter and the specified expression should not exceed the limitations of the current address space.

## DBIT Directive

The format of the DBIT directive is as follows:

[*label*:] DBIT *expression*

The DBIT directive reserves bit address space. It can be used only in the bit address segment. The expression must be a valid assembly-time expression with no forward references. When a DBIT statement is encountered in a program, the location counter of the bit address segment is incremented by the value of the expression.

# Memory Initialization

Beside normal instruction encoding there are two directives that initialize the contents of code memory. The instruction encoding proceeds as described in Chapter 3, with each instruction and its operands being evaluated and encoded sequentially. The directives DB and DW allow a programmer to specify a set of data values to be encoded.

## DB Directive

The format for a DB directive is shown below:

[*label*:] DB *expression list*

The DB directive initializes code memory with byte values (−256 to +255). Therefore, the code segment must be active. The expression list is a series of one or more byte values or strings separated by commas (,). A byte value can be represented as an assembly-time expression or as a character string. Each item in the list (expression or character in a string) is placed in memory in the same order as it appeared in the list.

The DB directive permits character strings longer than 2 characters, but they must not be part of an expression (i.e., you cannot use long character strings with an operator, including parentheses). If you specify the null character string as an item in the list (not as part of an expression) it evaluates to null and does not initialize memory. If you use the location counter ($) in the list, it evaluates to the code address of the byte being initialized. If the directive has a label, the value of the label will be the address of the first byte in the list.

The following example shows several ways you can specify the byte value list in a DB directive.

```
AGE:    DB   'MARY',27,'BILL',25,'JOE',21,'SUE',18
                          ; This DB statement lists
                          ; the names (character strings)
                          ; and ages (numbers) have been
                          ; placed in a list
                          ; (the label AGE will
                          ; address the 'M' in 'MARY')

PRIMES: DB   1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53
                          ; This DB lists the first 17 prime
                          ; numbers. (PRIMES is the address of 1)

QUOTE:  DB 'THIS IS A QUOTE'''    ; This is an example of how to
                                  ; put the quote character in a
                                  ; character string.

DB   $, $-1, $-2, $-3     ; This DB statement initializes
                          ; four bytes of memory with the
                          ; same value (the location counter
                          ; is incremented for each item in
                          ; the list)

DB   'MAY'                ; This is a valid DB statement

DB   ('MAY')              ; This is an invalid DB statement since ('MAY') is an expression, not
                          ; a string, and it will generate an error

DB   ''                   ; This expression list contains only the
                          ; null character string and will not produce
                          ; a value to be initialized (No space allocated)

DB   ('')                 ; This is an expression that will produce a
                          ; byte containing 0 (See Chapter 2
                          ; for expressions)
```

## DW Directive

The format for a DW statement is shown below:

```
[label:] DW  expression list
```

The DW directive initializes code memory with a list of word (16-bit) values. Therefore, the code segment must be active. The expression list can be a series of one or more word values separated by commas (,). A word value is an assembly-time expression. If you use the location counter ($) in the list, it evaluates to the code address of the high order byte in the word being initialized. As in all assembly-time expressions (but unlike the DB directive), no more than two characters are permitted in a character string, and the null character string evaluates to 0.

Each item in the list is placed in memory in the same order as it appears in the list, with the high order byte first followed by the low order byte. If the statement has a label, the value of the label will address the high order byte of the first value in the list.

The following examples shows several ways you can specify the word value list in a DW directive.

```
ARRIVALS:  DW    710, 'AM', 943,'AM', 1153,'AM', 315,'PM', 941,'PM'
                 ; This DW lists several flight arrivals
                 ; the numbers and characters are encoded
                 ; consecutively

INVENTORY: DW    'F',27869,'G',34524,'X',27834
                 ; This list of character and numeric values
                 ; will be encoded with the high order byte of each
                 ; character string filled with zeros
                 ; INVENTORY will address a byte containing all zeros

    DW $, $-2, $-4, $-6
                 ; This list of expressions evaluates to the same
                 ; numeric value (the location counter is incremented
                 ; for each item in the list)
```

# The END Directive

Every program must have an END statement. Its format is shown below.

```
END
```

The END statement may not have a label, and only a comment may appear on the line with it. The END statement should be the last line in the program; otherwise, it will produce an error.

# CHAPTER 5
# MACRO PROCESSING LANGUAGE (MPL)

This chapter describes MPL, the MCS-51 Macro Processing Language. Appendix E presents a more rigorous treatment of MPL.

MPL extends the MCS-51 Assembly Language to include these capabilities:

- Macro definition and invocation
- Macro-time string manipulation
- Macro-time expression evaluation
- Conditional assembly
- Macro-time console I/O

## Conceptual Overview of Macro Processing

Understanding macro processing requires a different perspective from the way assembly languages and high-level procedural languages are understood as treating source files. When you invoke ASM51 to assemble your source file, all MPL statements in your source file are evaluated before the actual assembly process starts. Your MPL statements are either function definitions or function calls. The functions can be MPL's built-in functions or your own user-defined functions. You use the MPL built-in function DEFINE to create your own functions.

MPL deals in strings. If you think of your source file as one long string, then its MPL statements (function definitions and function calls) are substrings of that one long string. MPL replaces function definitions with a null string (nothing), and each function call with its value, which is always a string and may be a null string. Similarly, any arguments present in function calls are given as strings, and may be interpreted by the function (depending on its definition) as integer values. Thus, depending on its context, the expression "86H" could represent the 3-character string '86H' or the 16-bit value 0000 0000 1000 0110B.

The following scheme illustrates these concepts:

1. Your source file as seen by the Macro Processor:

   (----------plaintext----------(macro-def)----------(macro-call)----------plaintext----------)

2. An internal, intermediate form after the macro-definition is stored:

   (----------plaintext----------o----------(macro-call)----------plaintext----------)

   Where 'o' represents a null string and 'macro call' contains '86H'.

3. The macro called may then consider '86H' as a string or an integer value:

   (----------plaintext----------(86H)----------plaintext----------)

   (----------plaintext----------(0000 0000 1000 0110B)----------plaintext----------)

4. The resulting macro expansion then becomes input to the assembler-proper:

   (---------- plaintext ----------)

The value (a string) of a function may be the null (empty) string, which has length zero. Every string (even the null string) has as a substring the null string. The null string should not be confused with zero, which has as one of its representations 00H. The string 00H is not zero, it is a 3-byte character string which, in assembly language, we use to represent zero.

The MPL built-in functions DEFINE and MATCH both always evaluate to (are replaced by) the null string. This is where side-effects come into play; you call DEFINE and MATCH (and others) not to obtain a value (string), but to perform side effects inherent to the functions. Thus, DEFINE enters a definition into the macro symbol table so that a later call of the defined function can be evaluated, and/or side-effects can be performed. Similarly, MATCH leaves nothing in its wake (i.e., has the null string as its value), but is used to split a string argument in two, and assign the two substrings as values of identifiers.

## MPL Identifiers

MPL identifiers, used for function and parameter names, must follow the same rules as assembly-language symbol names. An MPL identifier has the following characteristics:

1. The first character must be an alphabetic character A through Z, a question mark (?), or an underscore (_) sometimes called the break character. Upper- and lower-case alphabetic characters are not distinguished.
2. Successive characters may be alphabetic, numeric (0 through 9), the underscore character, or the question mark.
3. As with the assembly-language proper, identifiers may be any length but are considered unique only up to 31 characters.

## What Is Macro Processing?

The macro processor, which is part of the MCS-51 Macro Assembler, copies your source file to an intermediate file to be assembled. During the copying process, the macro processor examines each character of your source file for a distinguished character called the *metacharacter,* which can be any ASCII character, but by default is the percent-sign (%). When the metacharacter is detected, the macro processor knows that what follows is intended for macro processing.

The metacharacter signals the macro processor that what follows is:

- A user macro definition, such as:

```
%*DEFINE (AR(NAME, TYPE, VALUE)) (%NAME: D%TYPE  %VALUE
)
```

  This defines a macro AR with three parameters (NAME, TYPE, VALUE), which, when called with actual arguments (strings or function calls which evaluate to strings), expands to an assembly-language DBIT, or DS directive defining an area of address space of %VALUE units in length (bytes, or bits) the beginning of the area is addressed by %NAME. Notice that parameters are listed in the macro-name part of the definition without metacharacters, but in the replacement-pattern part of the definition each parameter is prefixed by the metacharacter %. Notice also that the carriage-return (following %VALUE is meant to be part of the macro definition, since we want the data definition directive to be on line by itself.

- A user macro invocation (call), such as:

```
%AR(LASZLO,S, 500)
```

  This call is replaced by its value, which according to the preceding definition is the following string, including the terminating carriage-return (and line-feed):

```
LASZLO: DS 500
```

  Similarly, the call:

```
%AR(GONZO,BIT, 100)
```

  expands to:

```
GONZO: DBIT 100
```

  including the final carriage-return-line-feed.

- A user call to an MPL built-in function, such as:

  ```
  %IF (%EQS(%ANSWER,YES)) THEN (%AR(LASZLO, W, 500))
  ```

  This call to the MPL built-in function IF evaluates to the first array definition above if the value of ANSWER (a user-defined function, presumably incorporating the MPL built-in functions IN and OUT) is exactly equal to the string 'YES', and evaluates to the null (empty) string otherwise.

If a macro definition follows the metacharacter, the macro processor saves the definition.

If a macro call follows the metacharacter, the macro processor retrieves the definition of the called macro, computes the value (an ASCII string) of the macro based on the call and its arguments, and places it in the intermediate file at the point of call. This is called expanding the macro.

If a call to an MPL built-in function follows the metacharacter, the macro processor replaces the call with the value of the built-in function, much the same as in the previous case. Calls to MPL built-in functions will be discussed later; however, this section describes one such MPL built-in function—DEFINE, which you call to define your macros. Strictly speaking, then, the first item on the above list is really a special case of the third.

Aside from macro definitions and calls, the text of your source file has no meaning to the macro processor. The macro processor forms the "front-end" of the assembler, and as such, it cannot detect errors in your MCS-51 assembly language directives or instructions.

### What Is a Macro?

A macro is a shorthand notation for a source text string. The shorthand notation is the macro name; the string it represents is the macro value. You define your own macros using the MPL function DEFINE, which has the format:

```
%*DEFINE (macro-name [(parameter-list)]) (replacement-pattern)
```

## Macro Expansions and Side Effects

A careful distinction must be made between the value of a macro or built-in function and its side-effects. At call-time, when the macro or built-in function is called, the macro processor replaces the call with the value (an ASCII string) of the macro or built-in function, as well as performing the operations inherent in the macro or built-in function.

The value of the DEFINE built-in function is the null (empty) string; therefore, when the call to DEFINE is made to define your user macro, the call is replaced by the null string. That is, the call is not copied from your source file to the intermediate file. The significance of the call to DEFINE is not its value, but its side-effect; that is, defining your user macro (entering it in the macro symbol table).

If, for example, you are coding a program which contains several calls to a subroutine that requires you to save this accumulator (ACC), the multiplication register (B), the program status word (PSW), and the data pointer (DPH, DPL) before the call, and restore them after the call. You could first define the macro CALLSUBROUTINE as follows:

```
%*DEFINE (CALLSUBROUTINE) (
            PUSH ACC
            PUSH B
            PUSH PSW
            PUSH DPH
            PUSH DPL
            CALL ROUTINE
            POP  DPL
            POP  DPH
            POP  PSW
            POP  B
            POP  ACC
)
```

Now wherever the macro call %CALLSUBROUTINE appears in your source file, the macro processor replaces it with the defined character string, including all carriage-returns, line-feeds, tabs, and blanks.

Two remarks are in order:

- The definition of the macro begins with "%*DEFINE". (The asterisk (*) is termed the *call-literally* character, and means that no macro expansion is requested at this time.)
- The macro definition has the form:

  %*DEFINE (*macro-name*) (*replacement-pattern*)

## What Is Macro-Time?

Macro-time is the term given to the time-frame within which the macro processor acts on your source file, copying it to an intermediate form for assembly, and processing your macro definitions and macro calls. No object code is created during macro-time. Macro-time is followed by Assembly-time, when absolute hex format code is created.

Since MPL allows you to generate virtually any character string, which will then be assembled, it influences the entire development cycle of your program. However, since the macro processor itself produces no object code, it cannot interrogate the assembly-time status of your program (such as referencing the assembler-proper's symbol table).

## Why Use Macros?

Since a macro defines a string of text (called the macro expansion value) that will replace a macro call, the usefulness of a macro depends on two characteristics:

- Its ability to represent a string of text using a shorter string
- Its ability to be used in different contexts; in a word, its flexibility

The example CALLSUBROUTINE above has the first characteristic, but not the second; CALLSUBROUTINE is a *constant* macro—its value never changes, unless you redefine it. You can redefine your macros (but not MPL's built-in functions) any time you want (with the exception that a macro definition may not modify itself) at call-time, the macro processor refers to the most recent definition of each user macro. In order to introduce more flexibility into MPL, we need to discuss parameters and arguments.

## Parameters and Arguments

A macro can also be defined so that part of it varies, depending on the information supplied to the macro in the form of arguments.

Returning to the previous example of the procedure call to SUBROUTINE, preceded by multiple PUSHes and followed by multiple POPs, we see that the macro CALLSUBROUTINE as defined has limited usefulness—we cannot use it for calls to other procedures besides ROUTINE.

We can code a macro to specify the same sequence of PUSHes, a call to any procedure (not just ROUTINE), and the same sequence of POPs, as follows:

```
%*DEFINE ( CALLSUB(ROUTINE) ) (
            PUSH ACC
            PUSH B
            PUSH PSW
            PUSH DPH
            PUSH DPL
            CALL %ROUTINE
            POP DPL
            POP DPH
            POP PSW
            POP B
            POP ACC
)
```

Now to generate a call to procedure AXOLOTL, for example (together with the preceding PUSHes and following POPs, as well as carriage-returns, line-feeds, tabs, and blanks), all you need to code is:

```
%CALLSUB(AXOLOTL)
```

In this example, ROUTINE is called a formal parameter, or simply a parameter. (It is also known as a *dummy* parameter, since its name in the definition of CALLSUB is irrelevant.)

When CALLSUB is called with a value for the formal parameter (ROUTINE), the actual value (AXOLOTL) is referred to as an argument.

In short, the parameter ROUTINE acts as a place-holder for the argument AXOLOTL.

In using macro definitions that have parameter lists, and corresponding macro calls that have argument lists:

- The parameter list of a macro definition is enclosed in parentheses following the macro name; parameters are separated by commas, as in:

  ```
  %*DEFINE (HAMBURGER(P1,P2,P3,P4,P5)) (text-string using
  %P1, %P2, %P3, %P4, %P5)
  ```

  When a parameter (to be replaced by an argument at call-time) appears in the replacement-string of the definition, be sure to prefix the metacharacter (%) to it.

- The argument list of a macro call is enclosed in parentheses following the macro name; arguments are separated by commas, as in:

  ```
  %HAMBURGER(CATSUP,MUSTARD,ONION,PICKLE,LETTUCE)
  ```

The only occurrence of the metacharacter in the macro call is that prefixed to the macro-name, unless one or more arguments are macros. If you use a macro as an argument, then you prefix the metacharacter to the argument as well. For instance, if the macro YELLOWSTUFF is defined:

```
%*DEFINE (YELLOWSTUFF) (MUSTARD)
```

Then you could call HAMBURGER as follows:

```
%HAMBURGER(CATSUP,%YELLOWSTUFF,ONION,PICKLE,LETTUCE)
```

and obtain the same macro expansion.

- You can use any number of parameters/arguments.

- This chapter describes a subset of MPL in which commas delimit parameters/arguments. More general constructs, including the use of LOCAL macros and symbols, are possible, as described in Appendix E, Macro Processor Language: Full Capabilities.

## Evaluation of the Macro Call

The macro processor evaluates the call %CALLSUB(AXOLOTL) as follows:

1. The macro processor recognizes the metacharacter (%), and momentarily suspends copying your source file while it looks up the definition of CALLSUB in its macro symbol table.

2. Finding CALLSUB in the symbol table, the macro processor sees that CALLSUB is defined using one parameter, and hence needs one user-supplied argument in order to be expanded.

3. Upon finding the string 'AXOLOTL' in parentheses immediately following the %CALLSUB macro call, the macro processor picks up 'AXOLOTL' as the argument to the macro call.

4. Then, using the definition of CALLSUB as the string of PUSHes, POPs, the CALL, and all carriage-returns, line-feeds, tabs, and blanks in the definition, the macro processor computes the value of the call %CALLSUB(AXOLOTL) to be the ASCII string:

```
PUSH ACC
PUSH B
PUSH PSW
PUSH DPH
PUSH DPL
CALL AXOLOTL
POP DPL
POP DPH
POP PSW
POP B
POP ACC
```

5. The macro processor replaces the macro call with the value of the macro, exactly at the point of call.

## A Comment-Generation Macro

Macro definitions and calls can be placed anywhere in your source file:

- as constant character strings (the first example)
- as operands to instructions (the second example)
- as in-line routines (the example following the next)
- as arguments to function calls
- as character strings that are more easily defined as macro functions and called as needed than rekeyed each time.

Consider this comment-generating macro, HEADER, which accepts 5 arguments. and is defined as follows:

```
%*DEFINE (HEADER(ROUTINE,DATE,NAME,PURPOSE,REGCLOB)) (
;*******************************************************************************
; ROUTINE NAME: %ROUTINE
; DATE: %DATE
; PROGRAMMER'S NAME: %NAME
; PURPOSE OF ROUTINE: %PURPOSE
; REGISTERS CLOBBERED: %REGCLOB
;*******************************************************************************
)
```

Note that in the macro definition of HEADER above:

- The definition begins with %*DEFINE. This informs the macro processor that no expansion is to take place. (That is, this is a definition.)
- In the DEFINE function's pattern for parameterized macro definitions:

  %*DEFINE (*macro-name*(*parameter-list*)) (*replacement-pattern*)

- The metacharacter (%) does not appear in the *macro-name* or *parameter-list* fields.
- The metacharacter (%) does appear as a prefix to parameter names in the *replacement-pattern*, since the macro processor needs to know that the first 'ROUTINE' in 'ROUTINE NAME: %ROUTINE' is not expanded when the macro is called, but the second is.
- The "hanging" left parenthesis at the right in the first line denotes that the macro body begins with a carriage-return. (Otherwise, the expanded macro might start in the middle of a line.) Similarly, the lone right-parenthesis which terminates the *replacement-pattern* denotes that the macro body ends with a carriage-return.

The macro call:

```
%HEADER(LASZLO,5/15/79,G. BOOLE,UPDATE NETWORK STRUCTURES,A/B/R0/DPTR)
```

results in the expansion:

```
;*******************************************************************************
; ROUTINE NAME: LASZLO
; DATE: 5/15/79
; PROGRAMMER'S NAME: G. BOOLE
; PURPOSE OF ROUTINE: UPDATE NETWORK STRUCTURES
; REGISTERS CLOBBERED: A/B/R0/DPTR
;*******************************************************************************
```

## A Macro to Add 16-Bit Values at Run-Time

You can use macros for routines. For instance, your source file might require frequent use of 16-bit addition:

1. Add 16-bit immediate data to a two byte numeric value in memory

```
MOV A,#(LOW 60134)      ; Move low order byte of data into accumulator
ADD A,55H               ; Add low order bytes
MOV 40H,A               ; Store low order sum
MOV A,#(HIGH 60134)     ; Move high order byte of data into accumulator
ADDC A,56H              ; Add high order byte with carry
MOV 41H,A               ; Store high order byte of sum
```

2. Add 2 byte value in register 6 and 7 to data pointer

```
MOV A,DPL               ; Move low order byte of data pointer
ADD A,R6                ; Add low order bytes
MOV R4,A                ; Store low order sum
MOV A,DPH               ; Move high order byte of data into accumulator
ADDC A,R7               ; Add high order byte with carry
MOV R5,A                ; Store high order byte of sum
```

3. Add 2 byte value in memory to 2 byte value in memory. Store in memory

```
MOV A,TL0               ; Move low order byte of data into accumulator
ADD A,TL1               ; Add low order bytes
MOV TL0,A               ; Store low order sum
MOV A,TH0               ; Move high order byte of data into accumulator
ADDC A,TH1              ; Add high order byte with carry
MOV TH0,A               ; Store high order byte of sum
```

By parameterizing the operand fields that differ in these text-strings, we obtain the body of the macro we need to generate all 3 instances:

```
MOV A,XLOW
ADD A,YLOW
MOV SUMLOW,A
MOV A,XHIGH
ADDC A,YHIGH
MOV SUMHIGH,A
```

Using the MPL built-in function DEFINE, we can name a macro representing the common form of the 3 separate instances:

```
%*DEFINE (ADD16(XHIGH, XLOW, YHIGH, YLOW, SUMHIGH, SUMLOW)) (
MOV A,%XLOW
ADD A,%YLOW
MOV %SUMLOW,A
MOV A,%XHIGH
ADDC A,%YHIGH
MOV %SUMHIGH,A
)
```

Note that in this macro definition:

- The metacharacter (%) and the call-literally character (*) are prefixed to DEFINE.

  Note that in the pattern:

  %*DEFINE (*macro-name*) (*replacement-pattern*)

  neither the metacharacter (%) nor the call-literally character (*) occurs in the *macro-name* field, but that the metacharacter (%) is prefixed to each *parameter-name* in the *replacement-pattern*. The call-literally character does not appear in the *replacement-pattern*.

- The *replacement-pattern* is defined by its appearance between the second pair of parentheses in the pattern:

  %*DEFINE (*macro-name*) (*replacement-pattern*)

  This means that ADD16 consists of the opening and closing carriage-returns given in its body, as well as the text between them. Without these opening and closing carriage-returns, the first and last lines of the expanded macro would be run together with the last line before, and the first line after, the macro call.

## Calling ADD16 with Actual Arguments

Now with the ADD16 macro defined for this assembly, it is unnecessary to code the sequence of instructions over again every time we wish to perform a 16-bit add. Our user macro ADD16 can be invoked (called) using actual arguments in place of the formal parameters XHIGH, XLOW, YHIGH, YLOW, SUMHIGH, SUMLOW. The formal parameters are simply place-holders until you supply actual values as arguments in macro calls.

For example, the macro calls:

```
%ADD16(#(HIGH 60134),#(LOW 60134),56H,55H,41H,40)
%ADD16(DPH,DPL,R7,R6,R5,R4)
%ADD16(TH0,TL0,TH1,TL1,TH0,TL0)
```

expand to (1), (2), and (3) above, respectively.

## The LEN Built-in Function

The MPL built-in function LEN accepts a string argument (or a macro whose value is a string), and returns a valid hexadecimal number.

Thus, the value of %LEN(ABC) is the ASCII string 03H. Similarly, the value of %LEN(ABCDEFGHIJ) is the ASCII string 0AH.

Furthermore, like other MPL built-in functions and user macros, LEN can accept a macro as an argument. In this case, the value of LEN is an ASCII string representing the length of the macro value string.

If, for example, ALPHA and DECIMAL are defined as follows:

```
%*DEFINE (ALPHA) (ABCDEFGHIJKLMNOPQRSTUVWXYZ)
%*DEFINE (DECIMAL) (0123456789)
```

then it follows that %LEN(%ALPHA) has the value 1AH, and %LEN(%DECIMAL) has the value 0AH. Note that %LEN(ALPHA) and %LEN(DECIMAL) are still meaningful, and have the values 05H and 07H, respectively.

## The EVAL Built-in Function

Since MPL deals in strings, the macro processor does not normally attempt to evaluate strings expressing numeric quantities. (Exceptions to this general rule are the built-in functions REPEAT, IF, WHILE, and SUBSTR, described below).

Thus, if you code:

```
%LEN(%ALPHA) + %LEN(%DIGIT)
```

the macro processor will treat the expression as a string, and will replace it with:

```
1AH + 0AH
```

without processing it any further.

If you want an expression to be evaluated, you can use the MPL built-in function EVAL function, which takes the form:

```
%EVAL(expression)
```

In this case, the desired evaluation is performed, and an ASCII string of hexadecimal digits is returned as the value of EVAL. For the example, we have:

```
%EVAL(%LEN(%ALPHA) + %LEN(%DIGIT))
```

which first reduces to:

```
%EVAL(1AH + 0AH)
```

and is then evaluated as an arithmetic expression to obtain the string:

```
24H
```

as the value of the call.

## Arithmetic Expressions

Arithmetic operations are 16-bit, as used by the assembler proper. Note that dyadic (two-argument) operators are infix (as assembler-proper operators), unlike MPL's outfix operators, and that infix operators do not require the metacharacter preceding a call, when used in an IF, WHILE, or REPEAT.

```
Infix:   %VALUE1 EQ 3        (compare numbers)
Outfix:  %EQS(%VALUE1, 3)  (compare strings)
```

Arithmetic expressions allow the following operators, in high-to-low order of precedence:

```
Parenthesized Expressions
HIGH  LOW
Multiplication and Division: *, /, MOD, SHL, SHR
Addition and Subtraction: +, – (both unary and binary)
Relational: EQ  LT  LE  GT  GE  NE
Logical NOT
Logical AND
Logical OR  XOR
```

Expressions are evaluated left-to-right, with operations of higher precedence performed first, unless precedence is overridden using parentheses.

It is essential to remember that these arithmetic, relational, and Boolean operators are identical to the assembly-language operators of the same names. The difference between using these operators in the MPL context as opposed to the usual assembly-language context is that:

1. For the operations to be performed, MPL expressions must be enclosed within one of the built-in functions.

2. Although the value returned by EVAL is always an ASCII string of hexadecimal digits, and not a "pure number", the hexadecimal string itself can be used as a number with arithmetic operators.

## String Comparator (Lexical-Relational) Functions

The string comparator functions are:

| MPL Function | Answers the Question | With One Of |
|---|---|---|
| EQS | Are the strings lexically equal? | 0FFFFH (Yes), 00H (No) |
| NES | Are the strings lexically unequal? | 0FFFFH (Yes), 00H (No) |
| LTS | Does the first precede the second in their dictionary ordering? | 0FFFFH (Yes), 00H (No) |
| LES | Does the first precede or equal the second in their dictionary ordering? | 0FFFFH (Yes), 00H (No) |
| GES | Does the first follow or equal the second in their dictionary ordering? | 0FFFFH (Yes), 00H (No) |
| GTS | Does the first follow the second in their dictionary ordering? | 0FFFFH (Yes), 00H (No) |

The value returned (0FFFFH or 00H) is a character string, and not a "pure number".

Thus, the function call:

```
%LTS(101,101B)
```

returns the string '0FFFFH', or "True", because the string '101' precedes the string '101B' in the lexical sense.

And the function call:

```
%EQS(0AH, 10)
```

returns the string '00H', or "False", because the two strings are not equal in the lexical sense (even though, if interpreted, they represent the same number).

## Control Functions (IF, REPEAT, WHILE)

The functions IF, REPEAT, and WHILE are useful for controlling the expansion of macros depending on whether an expression evaluates to True (0FFFFH, or any odd number) or False (00H, or any even number).

Unlike most instances of expressions in MPL (except for SUBSTR, described below), expressions in the first clause of IF, REPEAT, and WHILE are automatically interpreted as numbers, not strings. As a result, you do not need to code %EVAL(*expr*) as the first clause to the functions; the expression itself suffices.

The syntax for these expressions is as follows:

```
%IF ( expr ) THEN (replacement-value ) [ ELSE ( replacement-value ) ] FI
%REPEAT ( expr ) ( replacement-value )
%WHILE ( expr ) ( replacement-value )
```

where:

- *expr* must evaluate to an integer. (Note that it is not necessary to code %EVAL( *expr* ) for these three functions; the expression is automatically evaluated without your specifying EVAL.)
- *replacement-value* is an arbitrary string with balanced parentheses.

### The IF Function

If *expr* evaluates to an ODD integer, it is considered "True" and the value of the THEN-clause replaces the IF call. If macro calls appear in the THEN clause, the calls are made and replaced by their (string) values. Any side-effects inherent in the definition of the macro(s) called are performed.

If *expr* evaluates to an EVEN integer, it is considered "False" and the THEN-clause is ignored. The ELSE clause, if present, is then treated as if it were the THEN-clause in the "True" case.

For example, the call:

```
%IF (%LEN(ABC) EQ 3) THEN (%PROCESS) FI
```

Says, in effect:

1. Treat the expression %LEN(ABC) EQ 3 as a number, and evaluate it. (The IF built-in function, like several others, accepts an expression and treats it as a number, so you do not have to use EVAL here.)
2. If %LEN(ABC) EQ 3 evaluates False (00H), end processing of this call. (There is no ELSE clause in this particular instance.)
3. If %LEN(ABC)EQ 3 evaluates True (0FFFFH), evaluate the call %PROCESS (a user-defined function). This means:
   - Replace the entire %IF call with the value of %PROCESS (possibly null).
   - Perform any side-effects indicated in the definition of %PROCESS.

Since the value of %LEN(ABC)EQ 3 is True (0FFFFH), the call to PROCESS is made, %PROCESS is evaluated, and its value (a string) replaces the %IF call. Any side-effect processing inherent in the definition of process is also performed. (For instance, PROCESS may define a new user macro.)

If, on the other hand, the following IF call is made:

```
%IF (%EQS( %LEN(ABC), 3)) THEN (%PROCESS) FI
```

The IF-clause first reduces to:

```
%EQS( 03H, 3)
```

And since the string comparator function EQS does not regard '03H' as equal to '3', the expression evaluates to False, or 00H. Hence, PROCESS it not called.

As another example, the call:

```
%IF (%LEN(%STRING) GT 255) THEN (%TRUNC) ELSE (%CONCAT) FI
```

results in the following:

1. The user macro-call %STRING is evaluated and replaced by its expanded value (possibly null).
2. The length of the string is computed by LEN.
3. The relational expression:

   ```
   xH GT 255
   ```

   is evaluated, where "xH" represents the value of %LEN(%STRING).
4. If the hexadecimal value xH returned by LEN is greater than 255, the user-macro TRUNC is evaluated, and any side-effects inherent in its definition are performed. The value of TRUNC replaces the IF call (in this case the line). The ELSE-clause is ignored.
5. If the hexadecimal value returned by LEN is less than or equal to 255, the expression %TRUNC is ignored, but the user macro CONCAT is called, expanded, and any side-effects are performed.

Since the value of %LEN(ABC) EQ 3 is True (0FFFFH), the call to PROCESS is made, %PROCESS is evaluated, and its value (a string) replaces the %IF call. Any side-effect processing inherent in the definition of process is also performed. (For instance, PROCESS may define a new user macro.)

If, on the other hand, the following IF call is made:

```
%IF (%EQS(%LEN(ABC),3) THEN (%PROCESS) FI
```

then PROCESS is not called. Since the string comparator function EQS does not regard '03H' as equal to '3', the expression evaluates to False, or 00H.

## The REPEAT Function

The expression *expr* is evaluated only once; the *replacement-value* is then evaluated *expr* times, and becomes the value of the REPEAT function.

The format of the REPEAT function call is:

```
%REPEAT (expr) (string)
```

where *expr* is evaluated exactly once, and *string* is expanded *expr* times.

For example,

```
%REPEAT (10) (%REPEAT (4)(.)+)
```

generates the string:

```
....+....+....+....+....+....+....+....+....+....+....+
```

## The WHILE Function

The WHILE function call has the format:

```
%WHILE (expr) (replacement-value)
```

where *expr* is evaluated until it is False (Even) as follows:

1. The expression *expr* is first evaluated to determine whether the second (*replacement-value*) need be evaluated:
   - If *expr* evaluates to an odd (''True'') number, then *replacement-value* is evaluated, including all macro calls and side effects.
   - If *expr* evaluates to an even number (''False''), then no further processing is performed for the macro call.
   - If the side-effects of the *replacement-value* do not modify the conditions tested in the expression, then the loop will not terminate.
2. At this point, if *expr* evaluated True, *expr* is reevaluated (*replacement-value* may have called a macro to change a value in the expression), and the two listed conditions again apply. This ''looping'' is continued until *expr* evaluates ''False''.

For example, the macro call:

```
%WHILE (%EQS(%ANSWER,YES)) (%CONTINUE)
```

Evaluates as follows:

1. %ANSWER (a user function) is evaluated, and lexically compared to the string 'YES'.
2. If the strings compare equal, %CONTINUE (a user function) is evaluated, including side-effects. The value (a string) of %CONTINUE replaces the %WHILE call. Note that side-effects should include redefining ANSWER. Step 1 above is then repeated.
3. If the strings compare unequal, processing of this WHILE call stops. Any %CONTINUE values placed in the intermediate file remain.

## MATCH Function

The MATCH function allows you to manipulate lists. The syntax is:

```
%MATCH ( name1 , name2 ) ( list )
```

where *list* is a list of strings (none of which contains a comma) separated by commas. The value of the MATCH function is always null. MATCH is used for its side-effects, which are as follows:

- *name1* is assigned the substring of *list* preceding the first occurrence of a comma
- *name2* is assigned the substring of *list* following the first occurrence of a comma

Its primary use is to isolate and name substrings of a given string, as shown in the following example, and also in the example under "Console I/O"

For example, the following call to WHILE:

```
%WHILE (%LEN(%LIST) NE 0) (%MATCH (ITEM, LIST) (%LIST) %PROCESS(%ITEM))
```

results in the following macro processing:

1. First the length of the list defined by the user-macro LIST is evaluated. If it is nonzero, the second clause of WHILE is evaluated.
2. MATCH in the second clause of WHILE looks for a comma in the string defined by LIST. If a comma is found, the substring of LIST preceding the comma is assigned as the value of ITEM, and LIST takes on as a new value its substring following the occurrence of the comma.
3. Processing at this point is still in the second clause of WHILE. Next, ITEM is evaluated (the substring just found preceding the comma) and is fed to PROCESS (a user-defined macro) as an argument. If PROCESS has a value, it is inserted in the intermediate file, replacing the WHILE call.
4. Now the second clause of WHILE has been processed, so the macro processor returns to the first clause to evaluate the condition. Here, this is the same as saying, "GO to Step 1 above."

As you can see, this represents a different perspective on algorithms from that usually encountered in assembly-languages and garden-variety procedural languages. The net effect of the preceding example is to filter through the list, stopping at each comma, and assigning each substring between commas (and the substring preceding the first comma, and the substring following the last comma) to ITEM, and then processing item with the macro call to PROCESS. Finally, when you consider that MPL permits virtually any character combination to be used as a delimiter-specifier (not just commas), you can appreciate the assembly-time processing power here.

**NOTE**

This is actually a simplified form of MATCH, using a comma as a delimiter to match against in a list. The MPL language and implementation permit delimiters of very nearly any character combination. An example below (under "Console I/O") shows a different use of MATCH, matching against the carriage-return and line-feed characters considered jointly as a single delimiter. Refer to Appendix E for the full definition of MATCH.

## Console I/O Functions

The MPL built-in functions IN and OUT perform macro-time console I/O.

IN reads one line (including line-feed and carriage-return) from the console input device. The value of IN is the string typed, including the terminating carriage-return and line-feed bytes (0D0AH). The syntax is:

```
NI%
```

OUT writes a string to the console output device. (The value of OUT is NULL.) OUT has one parameter, the string to be written. The syntax of OUT is:

```
%OUT(string)
```

where *string* must have the same number of left- and right-parentheses.

The following example, when included in your source file and submitted for assembly, will prompt you for information to define a record array in which each record contains three fields. The prompt character is >.

```
%*DEFINE (REC(F)) LOCAL RECORDNAME (
%RECORDNAME RECORD %ITEM %REPEAT (%F-1) (, %ITEM)
%ARRAYNAME %RECORDNAME %EVAL(%NUMREC) DUP (< >)
)
%*DEFINE (ITEM) (%FLDNAME : %FLDWIDTH = %FLDVAL)
%*DEFINE (FLDNAME) (%OUT(NAME OF FIELD? ) %GET)
%*DEFINE (FLDWIDTH) (%OUT(WIDTH OF FIELD? ) %GET)
%*DEFINE (FLDVAL) (%OUT(INITIAL VALUE OF FIELD? ) %GET)
%*DEFINE (ARRAYNAME) (%OUT(NAME OF RECORD ARRAY?) %GET)
%*DEFINE (NUMREC) (%OUT(NUMBER OF RECORDS IN ARRAY?) %GET)
%*DEFINE (GET) (%MATCH (LINE %(
) NULL) (%IN) %LINE)
%REC(3)
```

If you want five fields instead, for example, change the call from %REC(3) to %REC(5). Or, you can define a function prompting you (or a user) for the number of record fields. Once you have some facility with MPL, you'll see vast possibilities. For instance, by inserting calls to EVAL in the definitions, you can increase the capability of the program to include expression (rather than constant) input.

## The SET Function

The SET function allows you to assign a macro-time value to a macro-time variable. The format is:

```
%SET(name,value)
```

where:

*name* is an MPL identifier
*value* is an expression acceptable to EVAL

For instance

```
%SET(LINES,10)
%SET(MAX,80-%LEN(%STRING))
%SET(CHARS,%MAX*%LINES)
```

You can use SET to redefine the same macro-time variable. For example,

```
%SET(LINES,10)
    o
    o
    o
%SET(LINES,15)
    o
    o
    o
%SET(LINES,%LINES+1)
```

the last statement increments the macro-time variable LINES by 1.

Unlike the other MPL built-in functions, the SET function can be redefined (but this is not recommended).

For example:

```
%*DEFINE(SET(X))(%DEFINE(%X)(-H))
```

## The SUBSTR Function

You can isolate a substring of a string or string expression using the SUBSTR built-in function. The format is:

```
%SUBSTR(string-expr,expr1,expr2)
```

where:

*string-expr* is a string or an MPL expression which evaluates to a string.

*expr1* evaluates to a string constant representing a number. This number is taken to be the character number of the beginning of the selected substring of the value of string-expr. The first character of the argument string is character 1.

*expr2* evaluates to a string representing a number. This number is taken to be the length of the selected substring.

SUBSTR evaluates to a null string if:

- *expr1* = 0 or *expr1* > %LEN(*string-expr*)
- *string-expr* evaluates to a null string
- *expr2* = 0

If *expr2* > %LEN(*string-expr*) − *expr1* + 1, then the selected substring begins at character number *expr1* and ends at the character number %LEN(*string-expr*).

SUBSTR examples:

```
%SUBSTR(ABC,1,2) = AB
%SUBSTR(A B C,1,3) = A B
%SUBSTR(ABC,0,1) = (null)
%SUBSTR(ABC,4,1) = (null)
%SUBSTR(ABC,2,2) = BC
%SUBSTR(ABC,2,3) = BC
%SUBSTR(ABC,3,1) = C
%SUBSTR(%(A,B,C),1,2) = A,
```

# CHAPTER 6
# ASSEMBLER OPERATION AND CONTROLS

This chapter describes how to invoke the MCS-51 Macro Assembler from your Intellec System running under the ISIS operating system. The assembler controls are also fully described.

## How to Invoke the MCS-51 Macro Assembler

The command to invoke the assembler is shown below:

[:F*n*:]ASM51 [:F*n*:]*sourcefile*[.*extension*] [*controls*]

You must specify the filename of the assembler ([:F*n*:]ASM51) and the filename of your source code ([:F*n*:]sourcefile[.*extension*]. The controls are optional.

ASM51 normally produces two output files. One contains a formatted listing of your source code. Unless you specify a particular filename with the PRINT control, it will have the same name as your source file, but with the extension 'LST'. The format for the listing file and how to change that format will be described later in this chapter. The other file produced by the assembler is the object file. The format for the object file is described in *Absolute Object File Formats* (Order Number 9800183). Unless you specify a particular filename with the OBJECT control, it will also have the same name as your source file, but its extension will be 'HEX'.

For example note the assembler invocation below.

```
-ASM51 PROG.SRC
```

If there were no controls in PROG.SRC that changed the default output files, ASM51 would produce two files. The listing file will be :F0:PROG.LST, and the object file will be :F0:PROG.HEX.

In addition to the output files, ASM51 uses six intermediate files (ASM51S.TMP, ASM51X.TMP, ASM51M.TMP, ASM51T.TMP, ASM51N.TMP, and ASM51I.TMP). They will be deleted before the assembler completes execution. Normally these files will be created on the same drive as your source program; however, you can specify the drives to be used with the WORKFILES control.

Any control (except INCLUDE) can be used in the invocation line.

You can continue the invocation line on one or more additional lines by typing an ampersand (&) before you type a carriage return. ASM51 prompts for the remainder of the invocation line by issuing a double asterisk followed by a blank (** ). Since everything following an ampersand on a line is echoed, but ignored, you can comment you invocation line; these comments are echoed in the listing salutation. (See Chapter 7 for an example.) Note the example below:

```
-ASM51 PROG.SRC        DATE(9-12-79) & Comment
** TITLE(COMPLETE PROJECT REV. 3.0) & Comment
** GEN
```

# Assembler Controls

Assemble controls may be entered in the invocation line as described above or on a control line in your source code. The general format for control lines is shown below:

$ *Control List* [; *Comment*]

The dollar sign ($) must be the first character on the line. The control list is zero or more controls separated by one or more spaces or tabs. The comment is optional.

ASM51 has two classes of controls: primary and general. The primary controls are set in the invocation line or the primary control lines and remain in effect throughout the assembly. For this reason, primary controls may only be used in the invocation line or in a control line at the beginning of the program. *Only other control lines (that do not contain the INCLUDE control) may precede a line containing a primary control.* The INCLUDE control terminates processing of primary controls.

The general controls are used to control the immediate action of the assembler. Typically their status is set and modified during an assembly. Control lines containing only general controls may be placed anywhere in your source code.

Table 6-1 lists all of the controls, their abbreviations, their default values, and a brief description of each.

Table 6-1. Assembler Controls

| Name | Primary/ General | Default | Abbrev. | Meaning |
|---|---|---|---|---|
| DATE(date) | P | DATE( ) | DA | Places string in header (max 9 characters) |
| DEBUG | P | NODEBUG | DB | Outputs debug symbol information to object file |
| NODEBUG | P | | NODB | Symbol information not placed in object file |
| EJECT | G | *Not Applicable* | EJ | Continue listing on next page |
| ERRORPRINT[(FILE)] | P | NOERRORPRINT | EP | Designates a file to receive error messages in addition to the listing file |
| NOERRORPRINT | P | | NOEP | Designates that error messages will be printed in listing file only |
| GEN | G | GENONLY | GE | Generates a full listing of the macro expansion process including macro calls in the listing file |
| GENONLY | G | | GO | List only the fully expanded source as if all lines generated by a macro call were already in source file |
| NOGEN | G | | NOGE | List only the original source text in listing file |

Table 6-1. Assembler Controls (Cont'd.)

| Name | Primary/ General | Default | Abbrev. | Meaning |
|---|---|---|---|---|
| INCLUDE(FILE) | G | *Not Applicable* | IC | Designates a file to be included as part of the program |
| LIST | G | LIST | LI | Print subsequent lines of source in listing file |
| NOLIST | G | | NOLI | Do not print subsequent lines of source in listing file |
| MACRO | P | MACRO | MR | Evaluate and expand all macro calls |
| NOMACRO | P | | NOMR | Do not evaluate macro calls |
| OBJECT[(FILE)] | P | OBJECT(*source*.HEX) | OJ | Designate file to receive object code |
| NOOBJECT | P | | NOOJ | Designates that no object file will be created |
| PAGING | P | PAGING | PI | Designates that listing will be broken into pages and each will have a header |
| NOPAGING | P | | NOPI | Designates that listing will contain no page breaks |
| PAGELENGTH(n) | P | PAGELENGTH(60) | PL | Sets maximum number of lines in each page of listing file (maximum = 65,535) (minimum = 10) |
| PAGEWIDTH(n) | P | PAGEWIDTH(120) | PW | Sets maximum number of characters in each line of listing file (maximum = 132; minimum = 72) |
| PRINT[(FILE)] | P | PRINT(*source*. LST) | PR | Designates file to receive source listing |
| NOPRINT | P | | NOPR | Designates that no listing file will be created |
| SAVE | G | Not Applicable | SA | Stores current control setting for LIST and GEN |
| RESTORE | G | | RS | Restores control setting from SAVE stack |
| SYMBOLS | P | NOSYMBOLS | SB | Creates a formatted table of all symbols used in program |
| NOSYMBOLS | P | | NOSB | No symbol table created |
| TITLE(string) | G | TITLE( ) | TT | Places a string in all subsequent page headers (maximum 60 characters) |
| WORKFILES(:Fn:[,:F m:]) | P | *same drive as source file* | WF | Designates alternate drives for temporary workfiles |
| XREF | P | NOXREF | XR | Creates a cross reference listing of all symbols used in program |
| NOXREF | P | | NOXR | No cross reference list created |

## Control Definitions

**Control Switch Name:** DATE

**Abbreviation:** DA

**Arguments:** (string) (*Nine characters maximum*)

**Control Class:** Primary

**Default:** (*Spaces inserted*)

**Definition:** The assembler takes the character string specified as the argument and inserts it in the header. If you specify less than 9 characters, then it will be padded with blanks. If more than 9 characters are specified, then the character string will be truncated to the first nine characters. DATE is overridden by NOPRINT.

**NOTE**

Any parentheses in the DATE string must be balanced.

**Example:**

```
$TITLE(PROJECT F.A.N. REV. 27)        DATE(1-1-80)

                (Header will look like this)
MCS-51 MACRO ASSEMBLER  PROJECT F.A.N. REV. 27      1-1-80  PAGE 1
```

**Control Switch Name:** DEBUG/NODEBUG

**Abbreviation:** DB/NODB

**Arguments:** None

**Control Class:** Primary

**Default:** NODEBUG

**Definition:** Indicates whether debug symbol information shall be output to object file. If DEBUG is in effect the debug information will be output. This control must be used if you wish to run the program with an ICE-51.

DEBUG is overridden by NOOBJECT.

**Example:** $DEBUG

**Control Switch Name:** EJECT

**Abbreviation:** EJ

**Arguments:** None

**Control Class:** General

**Default:** (*New page started when PAGELENGTH reached*)

**Definition:** Inserts formfeed into listing file, after the control line containing the EJECT, and generates a header at top of the next page. The control is ignored if NOPAGING, NOPRINT, or NOLIST is in effect.

**Example:** $EJECT

**Control Switch Name:** ERRORPRINT/NOERRORPRINT

**Abbreviation:** EP/NOEP

**Arguments:** (Filename) (*Indicates file to receive error messages—argument optional.*)

**Control Class:** Primary

**Default:** NOERRORPRINT

**Definition:** When ERRORPRINT is in effect, indicates that all erroneous lines of source and the corresponding error message shall be output to the specified file. This will not inhibit errors from being placed in listing file. If no argument is specified to ERRORPRINT, then erroneous lines and error messages will be displayed at the console.

**Example:** $ERRORPRINT

**Control Switch Name:** GEN/GENONLY/NOGEN

**Abbreviation:** GE/GO/NOGE

**Arguments:** None

**Control Class:** General

**Default:** GENONLY

**Definition:** NOGEN indicates that only the contents of the source file shall be output to the listing file with macro call expansion not shown. Expansion will take place, but source lines generated will not be displayed in listing file, only the macro call.

GENONLY indicates that only the fully expanded macro calls will appear in the listing. The listing file appears as if the expanded text was originally in the source file with no macro calls. The macro calls will not be displayed, but the source lines generated by the calls will be in the listing file.

GEN indicates that each macro call shall be expanded showing nesting of macro calls. The macro call and the source lines generated by the macro call will be displayed in the listing file.

These controls are overridden by NOPRINT and NOLIST. (See Chapter 7 for examples of a macro calls listed with GEN, GENONLY and NOGEN in effect.)

**Example:** $NOGEN

**Control Switch Name:** INCLUDE

**Abbreviation:** IC

**Arguments:** (Filename) (*Identifies file to be included into program*)

**Control Class:** General

**Default:** Not applicable.

**Definition:** Inserts the contents of the file specified in the argument into the program immediately following the control line. INCLUDE files may be nested.

The INCLUDE control may not appear in the invocation line, and it terminates processing of primary controls in the source.

**Example:** $INCLUDE(:F1:IOPACK.SRC)

**Control Switch Name:** LIST/NOLIST

**Abbreviation:** LI/NOLI

**Arguments:** None

**Control Class:** General

**Default:** LIST

**Definition:** Indicates whether subsequent lines of source text shall be displayed in listing file. A LIST control following a NOLIST will not be displayed, but listing will continue with the next sequential line. NOPRINT overrides LIST.

**NOTE**

Lines causing errors will be listed when NOLIST is in effect.

**Example:** $NOLIST

**Control Switch Name:** MACRO/NOMACRO

**Abbreviation:** MR/NOMR

**Arguments:** None

**Control Class:** Primary

**Default:** MACRO

**Definition:** Indicates whether macro calls shall be expanded. If NOMACRO is specified all macro calls will not be processed as macros. The NOMACRO control will free additional symbol table space for user-defined symbols (labels and symbols defined by SET, EQU, DATA, XDATA, and BIT).

**Example:** $NOMACRO

**Control Switch Name:** OBJECT/NOOBJECT

**Abbreviation:** OJ/NOOJ

**Arguments:** (Filename) (*Indicates file to receive hex code—argument optional.*)

**Control Class:** Primary

**Default:** OBJECT(*sourcefile*.HEX)

**Definition:** Indicates whether absolute hex code shall be generated, and if so, the file that will receive it. If you do not specify the argument, the object file will be *sourcefile*.HEX. The format of the file is described in *Absolute Object File Formats* (Order number 9800183).

**Example:** $OBJECT(:F1:FINAL.REV)

**Control Switch Name:** PAGING/NOPAGING

**Abbreviation:** PI/NOPI

**Arguments:** None

**Control Class:** Primary

**Default:** PAGING

**Definition:** Indicates whether page breaks shall be included in listing file. If NOPAGING, then there will be no page breaks in the file, and lines will appear listed consecutively. A single header will be included at the top of the file. EJECT and PAGELENGTH controls will be ignored.

If PAGING, a formfeed and a page header will be inserted into the listing file whenever the number of lines since the last page break equals the PAGELENGTH value, or an EJECT control is encountered. The header includes the assembler designation, the name of the source file, the TITLE and DATE strings (if specified), and the page number.

**Example:** $ NOPAGING

**Control Switch Name:** PAGELENGTH

**Abbreviation:** PL

**Arguments:** (n) *(Decimal number greater than 9.)*

**Control Class:** Primary

**Default:** PAGELENGTH(60)

**Definition:** Indicates the maximum number of printed lines on each page of the listing file. This number includes the page heading. The minimum value for PAGELENGTH is 10. Values less than 10 will be treated as 10. The maximum value permitted in the argument is 65,535.

**Example:**
```
$ PAGELENGTH(132)
```

**Control Switch Name:** PAGEWIDTH

**Abbreviation:** PW

**Arguments:** (n) *(Number indicates maximum characters per line.)*

**Control Class:** Primary

**Default:** PAGEWIDTH(120)

**Definition:** Indicates the maximum number of characters printed on a line in the listing file. The range of values permitted is from 72 to 132; argument values that are outside of this range will be rounded up or down accordingly.

Listing lines that exceed the PAGEWIDTH value will be wrapped around on the next lines in the listing, starting at column 30.

**Example:**
```
$ PAGEWIDTH(72)
```

**Control Switch Name:** PRINT/NOPRINT

**Abbreviation:** PR/NOPR

**Arguments:** (Filename) (*Indicates file to receive assembler listing—argument optional.*)

**Control Class:** Primary

**Default:** PRINT(*sourcefile*.LST)

**Definition:** Indicates whether formatted source listing shall be generated, and, if so, what file will receive it. If you do not specify the argument, the listing file will be *sourcefile*.LST. NOPRINT indicates no listing file will be made.

**Example:**
```
-ASM51 PROG.SRC PRINT(:LP:) & print listing at line printer
**
```

**Control Switch Name:** SAVE/RESTORE

**Abbreviation:** SA/RS

**Arguments:** None

**Control Class:** General

**Default:** Not applicable

**Definition:** Permits you to save and restore the state of the LIST and GEN controls. SAVE stores the setting of these controls on the SAVE stack, which is internal to the assembler. RESTORE restores the setting of the controls to the values most recently saved, but not yet restored. SAVEs can be nested to a depth of 8.

**NOTE**

SAVE uses the values that were in effect on the line prior to the SAVE control line. Therefore, if the LIST control is in effect and the assembler encounters a control line containing NOLIST and SAVE (in any order on the line), the status LIST is saved on the stack. (The lines following the control line are not listed until a LIST or RESTORE is encountered.)

**Example:**
```
$save
```

**Control Switch Name:** SYMBOLS/NOSYMBOLS

**Abbreviation:** SB/NOSB

**Argument:** None

**Control Class:** Primary

**Default:** NOSYMBOLS

**Definition:** Indicates whether a symbol table shall be listed. NOSYMBOLS indicates no symbol table. SYMBOLS causes the table to be listed. NOSYMBOLS is overridden by XREF. SYMBOLS is overridden by NOPRINT. (See Chapter 7 for an example symbol table listing.)

**Example:** $NOSYMBOLS

**Control Switch Name:** TITLE

**Abbreviation:** TT

**Arguments:** (string) (*Up to 60 characters.*)

**Control Class:** General

**Default:** (Spaces Inserted)

**Definition:** Permits you to include a title for the program. It will be printed in the header of every subsequent page. Titles longer than 60 characters will be truncated to the first 60 characters. (See Chapter 7 for an example of the title in the header.)

**NOTE**

Any parentheses in the TITLE string must be balanced.

**Example:** $TITLE(Final Production Run)

**Control Switch Name:** WORKFILES

**Abbreviation:** WF

**Arguments:** (:Fm:[,:Fn:]) *(Drives to use for temporary work files—second argument optional.)*

**Control Class:** Primary

**Default:** Drive that contains source file.

**Definition:** Indicates drives to be used to contain temporary workfiles. First drive listed will be used for files ASM51S.TMP, ASMF1X.TMP, and ASM51M.TMP. Second drive listed will be used for file ASM51T.TMP, ASM51N.TMP, and ASM51I.TMP. If only one drive is specified, then all workfiles will be placed on that drive. All workfiles are deleted before normal termination.

**Example:** –ASM51 :F1:BIGPR.SRC WORKFILES(:F4:,:F5:)

**Control Switch Name:** XREF/NOXREF

**Abbreviation:** XR/NOXR

**Arguments:** None

**Control Class:** Primary

**Default:** NOXREF

**Definition:** Indicates that a cross reference table of the use of symbols shall be added to the symbol table. Each cross reference table will list the line numbers of the lines that define the value of a symbol, and all of the lines that reference the symbol. A hash mark (#) follows the numbers of the lines that define the symbols value. XREF is overridden by NOPRINT. (See Chapter 7 for an example of a symbol table listing with XREF.)

**Example:** $XREF

# CHAPTER 7
# ASSEMBLER OUTPUT: ERROR MESSAGES AND LISTING FILE FORMAT

This chapter discusses the meaning of error messages issued by ASM51. The format of the listing file is also described.

## Error Messages and Recovery

All error messages issued by ASM51 are either displayed on the console or listed in the source file. Error messages listed at the console are fatal, causing ASM51 to abnormally terminate. Other than the error message printed at the console, ASM51 produces no other useful output. Error messages listed in the source file are non-fatal and usually allow at least the listing to continue.

### Console Error Messages

Upon detecting certain catastrophic conditions with the system hardware, or in the invocation line or one of the primary control lines, ASM51 will print an informative message at the console and abort processing.

These errors fall ASM51 into three broad classes: I/O errors, internal errors and ASM51 fatal errors.

A list of these fatal control error messages and a description of the cause of each is shown below.

#### I/O Errors

I/O error messages print with the following format:

```
ASM51 I/O ERROR–
 FILE: file type
 NAME: file name
 ERROR: ISIS error number and brief description
ASM51 TERMINATED
```

The list of possible file types is:

```
SOURCE
PRINT
OBJECT
INCLUDE
ERRORPRINT
ASM51 WORKFILE
ASM51 OVERLAY number
```

The list of possible error numbers is:

```
 4—ILLEGAL PATH NAME
 5—ILLEGAL OR UNRECOGNIZED DEVICE IN PATH
 9—DIRECTORY FULL
12—ATTEMPT TO OPEN ALREADY OPEN FILE
13—NO SUCH FILE
14—WRITE PROTECTED FILE
22—OUTPUT MODE IMPOSSIBLE FOR SPECIFIED FILE
23—NO FILENAME SPECIFIED FOR A DISK FILE
28—NULL FILE EXTENSION
```

## ASM51 Internal Errors

The ASM51 internal errors indicate that an internal consistency check failed. A likely cause is that one of the files containing the assembler's overlays was corrupted or that a hardware failure occurred. If the problem persists, contact Intel Corporation via the Software Problem report.

These messages print in the following format:

**** ASM51 INTERNAL ERROR: *message*

Be sure to include the exact text of the *message* on the problem report.

## ASM51 Fatal Errors

The fatal error messages print in the following format:

ASM51 FATAL ERROR–
*error message*

The possible error messages are:

NO SOURCE FILE FOUND IN INVOCATION

If ASM51 scans the invocation line and cannot find the source file name, then this error will be issued and assembly aborted.

UNRECOGNIZABLE SOURCE FILE NAME

If the first character after "ASM51" on the invocation line is not an "&" or a file character (i.e., ":", letter, digit, "."), then ASM51 issues this error and aborts.

ILLEGAL SOURCE FILE SPECIFICATION

If the source file is not a legal file name (does not conform to the ISIS-II rules for a path name), then this error is issued.

SOURCE TEXT MUST COME FROM A FILE

The source text must always come from a file, not devices like :TI: or :LP:.

NOT ENOUGH MEMORY

If there is not enough memory in your SERIES-II or MDS 800, then this error message will print out and ASM51 will abort.

If identical files are specified:

__ AND __ FILES ARE THE SAME

where the "__" can be any of SOURCE, PRINT, OBJECT, and ERRORPRINT. It doesn't make sense for any of these files to be the same.

BAD WORKFILES COMMAND

If a WORKFILES control has no parameters (i.e., devices) or a device specification is incorrect, this error message is issued.

BAD WORKFILES SYNTAX

If ASM51 encounters anything other than a "," or a ")" when it is looking for the next workfile, then this error is issued.

BAD PAGELENGTH
BAD PAGEWIDTH

The parameter to pagelength and pagewidth must be a decimal number. The number may have leading and trailing blanks, but if there are any other extra characters in the parameter, then this error will be issued.

PAGELENGTH MISSING A PARAMETER
PAGEWIDTH MISSING A PARAMETER
DATE MISSING A PARAMETER

These commands require parameters. If there is no parameter, then assembly is aborted.

CANNOT HAVE INCLUDE IN INVOCATION

The INCLUDE command may appear only in the source text. Don't forget that command lines in the source file can contain primary commands, but only if they are the very first lines in the file. Also, if one of these lines has an INCLUDE on it, then that ends the primary command lines.

EOL ENCOUNTERED IN PARAMETER

A parameter in the invocation line is missing a right parenthesis.

COMMAND TOO LONG

A command word longer than 128 characters—very unlikely.

ILLEGAL CHARACTER IN INVOCATION

There was an illegal character in the invocation line—usually a typing error. (See error 403.)

UNRECOGNIZED CONTROL: *<control-name>*

This message is issued if a problem occurs in the invocation or in one of the primary control lines. (See error 407.)

## Listing File Error Messages

ASM51 features an advanced error-reporting mechanism. Some messages pinpoint the symbol, or character at which the error was detected. Error messages printed in the source file are inserted into the listing after the lines on which the errors were detected.

They are of the following format:

*** ERROR #*eee*, LINE #*///* (*ppp*), *message*

where:

| | |
|---|---|
| *eee* | is the error number |
| *///* | is the number of the line on which the error occurred |
| *ppp* | is the line containing the last previous error |
| *message* | is the English message corresponding to the error number |

If the error is detected in pass 2, the clause "(PASS 2)" precedes the message. "(MACRO)" precedes the message for macro errors; "(CONTROL)" precedes the message for control errors.

Errors which refer to character or symbol in a particular line of the source file do so by printing a pointer to the first item in the line that is not valid; e.g.:

```
***_____^
```

The up arrow or vertical bar points to the first incorrect character in the line.

Error messages that appear in the listing file are given numbers. The numbers correspond to classes of errors. The classes of errors and the numbers reserved for these classes is shown in the list below:

```
  0 -  99  Source File Errors
300 - 399  Macro Errors
400 - 499  Control Errors
800 - 899  Special Assembler Errors
900 - 999  Fatal Errors
```

Errors numbered less than 800 are ordinary, non-fatal errors. Assembly of the error line can usually be regarded as suspect, but lines subsequent lines will be assembled. If an error occurs within a macro definition, the definition does not take place.

## Source File Error Messages

There follows a list of the error messages generated by ASM51, ordered by error number.

*** ERROR #1 SYNTAX ERROR

This message is preceded by a pointer to the character at which the syntax error was detected.

ASM51 contains an internally-encoded grammar of the MCS-51 assembly language and requires your program to conform to that grammar. The syntax error is recognized at the item indicated in the error message; e.g.,

```
... TEMP SER 10
...  _____^
```

gives a syntax error at the S. "SER" is unrecognized. However, sometimes the error is not detected until one or more characters later; e.g.,

```
... SETB EQU 1
...  _____^
```

gives a syntax error at "EQU". The error is that SETB is already defined as an instruction. The assembler interprets the line as a SETB instruction with "EQU 1" as the operand field. Since the keyword "EQU" is not a legal operand the "EQU" is flagged, even though the "SETB" is the user's mistake.

ASM51 discards the rest of the line when it finds a syntax error. If the error occurs within a macro definition, the assembler exits the definition mode.

*** ERROR #2 SOURCE LINE LISTING TERMINATED AT 255 CHARACTERS

Listing of the source line was stopped at 255 characters. The entire line was interpreted, only the listing is incomplete.

*** ERROR #3 ARITHMETIC OVERFLOW IN NUMERIC CONSTANT

This error is reported whenever the value expressed by a constant exceeds the internal representation of the assembler (65,535).

*** ERROR #4 ATTEMPT TO DIVIDE BY ZERO

This error occurs when the right hand side of a division or MOD operator evaluates to zero.

*** ERROR #5 EXPRESSION WITH FORWARD REFERENCE NOT ALLOWED

Forward references are not permitted in the expression argument to ORG, DS, EQU, SET, BIT, DATA, XDATA, and DBIT directives. Change the expression to remove the forward reference, or define the symbols earlier in the program.

*** ERROR #6 TYPE OF SET SYMBOL DOES NOT ALLOW REDEFINITION

This error occurs when the symbol being defined in a SET directive is a predefined assembler symbol or has been previously defined as a label or with EQU, DATA, BIT, or XDATA. For example, the following lines would cause this error on the second line.

```
SKIP_1: ADD A,R1
SKIP_1 SET 22D
```

*** ERROR #7 EQU SYMBOL ALREADY DEFINED

This message is given when the symbol has already been defined as a label or with the SET, DATA, BIT, or XDATA directive. To correct this error, use a different symbol name.

*** ERROR #8 ATTEMPT TO ADDRESS NON-BIT-ADDRESSABLE BIT

This error is caused when the left hand side of the bit selector (.) is not one of the bit addressable bytes. (See errors 40 and 9.) Figure 2-2 shows all bit-addressable bytes. Several examples of lines that would cause this type of error are shown below.

```
JB 10H.5,LOOP
CLR 7FH.0
MOV C,0AFH.3
```

*** ERROR #9 BAD BIT OFFSET IN BIT ADDRESS EXPRESSION

This error is caused when the right hand side of the bit selector (.) is out of range (0-7). The assembler uses 0 in its place. The byte address, if correct, remains the same. (See errors 8, and 40.) Several examples of lines that would generate this error are shown below.

```
CLR 25H.10
SETB 26H.5+4
CPL PSW.-1
```

*** ERROR #10 TEXT FOUND BEYOND END STATEMENT - IGNORED

This is a warning—there are no ill effects. The extra text appears in the listing file, but it is not assembled.

*** ERROR #11 PREMATURE END OF FILE (NO END STATEMENT)

There are no ill effects from omitting the END statement, other than this message.

*** ERROR #12 ILLEGAL CHARACTER IN NUMERIC CONSTANT

Numeric constants begin with decimal digits, and are delimited by the first non-numeric character. The set of legal characters for a constant is determined by the base:

1. Base 2: 0,1, and the concluding B.
2. Base 8: 0-7, and the concluding Q or O.
3. Base 10: 0-9, and the concluding D or null.
4. Base 16: 0-9, A-F, and the concluding H.

*** ERROR #13 ILLEGAL USE OF REGISTER NAME IN EXPRESSION

This error is caused by placing a register name (R0-R7 or A) or a symbol defined as a register in a numeric expression. It can be generated by any line that calls for a numeric expression. Several examples of this type of error are shown below:

```
POP A
DB R0
JZ A
```

*** ERROR #14 SYMBOL IN LABEL FIELD ALREADY DEFINED

You can define a label only once in your program. If the symbol name has been defined anywhere else in the program this error will be generated.

*** ERROR #15 ILLEGAL CHARACTER

This message is preceded by a pointer to the illegal character.

A character that is not accepted by ASM51 was found in the input file. Either it is an unprintable ASCII character, in which case it is printed as an up arrow (^), or it is printable but has no function in the assembly language. Edit the file to remove the illegal character.

*** ERROR #16 MORE ERRORS DETECTED, NOT REPORTED

After the ninth source file Error on a given source line, this message is given and no more errors are reported for that line. Normal reporting resumes on the next source line. (See errors 300 and 400.)

*** ERROR #17 ARITHMETIC OVERFLOW IN LOCATION COUNTER

This error is reported whenever the DS, DBIT, or ORG directive attempts to increase the location counter beyond the limits of the current segment. This may also occur in CSEG when instructions cause the location counter to increment above 65,535.

*** ERROR #18 UNDEFINED SYMBOL

This error is reported when an undefined symbol occurs in an expression. Zero is used in its place—this may cause subsequent errors.

*** ERROR #19 VALUE WILL NOT FIT INTO A BYTE

This error is issued whenever the expression used for a numeric operand that is encoded as a single byte is not in the range −256 to +255.

*** ERROR #20 OPERATION INVALID IN THIS SEGMENT

This error will occur if you use the DBIT or DS directive in the incorrect segment mode, or if you attempt to initialize memory (use DB, DW, or assemble) an instruction in any mode but CSEG.

*** ERROR #21 STRING TERMINATED BY END-OF-LINE

All strings must be completely contained on one line.

*** ERROR #22 STRING LONGER THAN 2 CHARACTERS NOT ALLOWED IN THIS CONTEXT

Outside of the DB directive all strings are treated as absolute numbers; hence, strings of 3 or more characters are overflow quantities. If this error occurs in a DW directive, you probably should be using DB.

*** ERROR #23 STRING, NUMBER, OR IDENTIFIER CANNOT EXCEED 255 CHARACTERS

The maximum length of a character string (including surrounding quotes), a number, or an identifier is 255 characters.

*** ERROR #24 DESTINATION ADDRESS OUT OF RANGE FOR AJMP
*** ERROR #26 DESTINATION ADDRESS OUT OF RANGE FOR ACALL

These errors are caused by specifying an address that is outside the 2K byte page boundary of the instruction. When the ACALL or AJMP is located near a 2K page boundary, only a few bytes may separate it from its destination address. An LJMP or LCALL will always correct the problem. It is often easier to use the generic forms of the jump and call instructions (CALL or JMP) and let the assembler select the correct translation.

*** ERROR #25 DESTINATION ADDRESS OUT OF RANGE FOR SJMP
*** ERROR #27 DESTINATION ADDRESS OUT OF RANGE FOR JC
*** ERROR #28 DESTINATION ADDRESS OUT OF RANGE FOR JNC
*** ERROR #29 DESTINATION ADDRESS OUT OF RANGE FOR JZ
*** ERROR #30 DESTINATION ADDRESS OUT OF RANGE FOR JNZ
*** ERROR #31 DESTINATION ADDRESS OUT OF RANGE FOR DJNZ
*** ERROR #32 DESTINATION ADDRESS OUT OF RANGE FOR CJNE
*** ERROR #33 DESTINATION ADDRESS OUT OF RANGE FOR JB
*** ERROR #34 DESTINATION ADDRESS OUT OF RANGE FOR JBC
*** ERROR #35 DESTINATION ADDRESS OUT OF RANGE FOR JNB

A relative jump has a 255 byte range (−128 to +127) from the instruction that follows the jump instruction. Any address outside of this range will generate one of these errors. You can correct this error in one of two ways. If the jump has a logical complement (e.g., JC and JNC) the following change could be made:

```
JC TOP          to      JNC SKIP
                        JMP TOP
                SKIP:
```

If the instruction has no logical complement, then the following change could be made:

```
DJNZ R0,TOP          to          DJNZ R0,SKIP_1
                                 JMP SKIP_2
                        SKIP_1:  JMP TOP
                        SKIP_2:
```

*** ERROR #36 CODE SEGMENT ADDRESS EXPECTED
*** ERROR #37 DATA SEGMENT ADDRESS EXPECTED
*** ERROR #38 XDATA SEGMENT ADDRESS EXPECTED
*** ERROR #39 BIT SEGMENT ADDRESS EXPECTED

These errors are caused by specifying a symbol with the wrong segment type in an operand to an instruction. The numeric value of that symbol is used, but it may cause subsequent errors (e.g., error 17).

*** ERROR #40 BYTE OF BIT ADDRESS NOT IN DATA SEGMENT

The symbol specified on the left hand side of the bit selector (.) is not segment type DSEG. The numeric value is used if possible, but may cause other errors. (See errors 37 and 8.)

## Macro Error Messages

Error messages with numbers in the 300's indicate macro call/expansion errors. Macro errors are followed by a trace of the macro call/expansion stack—a series of lines which print out the nesting of macro calls, expansions, INCLUDE files, etc.

Processing resumes in the original source file, with all INCLUDE files closed and macro calls terminated.

*** ERROR #300 MORE ERRORS DETECTED, NOT REPORTED

After 100 Macro or Control Errors on a given source line, this message is given and no more errors are reported for that line. Normal reporting resumes on the next source line. If the last error reported is a Macro Error, then this message will be issued. (See errors 16 and 400.)

*** ERROR #301 UNDEFINED MACRO NAME

The text following a metacharacter (%) is not a recognized user function name or built-in function. The reference is ignored and processing continues with the character following the name.

*** ERROR #302 ILLEGAL EXIT MACRO

The built-in macro "EXIT" is not valid in this context. The call is ignored. A call to "EXIT" must allow an exit through a user function, or the WHILE or REPEAT built-in functions.

*** ERROR #303 FATAL SYSTEM ERROR

Loss of hardware and/or software integrity was discovered by the macro processor. Contact Intel Corporation.

*** ERROR #304 ILLEGAL EXPRESSION

A numeric expression was required as a parameter to one of the built-in macros EVAL, IF, WHILE, REPEAT, and SUBSTR. The built-in function call is aborted, and processing continues with the character following the illegal expression.

*** ERROR #305 MISSING "FI" IN "IF"

The IF built-in function did not have a FI terminator. The macro is processed, but may not be interpreted as you intended.

*** ERROR #306 MISSING "THEN" IN "IF"

The IF built-in macro did not have a THEN clause following the conditional expression clause. The call to IF is aborted an processing continues at the point in the string at which the error was discovered.

*** ERROR #307 ILLEGAL ATTEMPT TO REDEFINE MACRO

It is illegal for a built-in function name or a parameter name to be redefined (with the DEFINE or MATCH built-ins). Also, a user function cannot be redefined inside an expansion of itself.

*** ERROR #308 MISSING IDENTIFIER IN DEFINE PATTERN

In DEFINE, the occurrence of "@" indicated that an identifier type delimiter followed. It did not. The DEFINE is aborted and scanning continues from the point at which the error was detected.

*** ERROR #309 MISSING BALANCED STRING

A balanced string "(...)" in a call to a built-in function is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

*** ERROR #310 MISSING LIST ITEM

In a built-in function, an item in its argument list is missing. The macro function call is aborted and scanning continues from the point at which the error was detected.

*** ERROR #311 MISSING DELIMITER

A delimiter required by the scanning of a user-defined function is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

This error can occur only if a user function is defined with a call pattern containing two adjacent delimiters. If the first delimiter is scanned, but is not immediately followed by the second, this error is reported.

*** ERROR #312 PREMATURE EOF

The end of the input file occurred while the call to the macro was being scanned. This usually occurs when a delimiter to a macro call is omitted, causing the macro processor to scan to the end of the file searching for the missing delimiter.

Note that even if the closing delimiter of a macro call is given, if any preceding delimiters are not given, this error may occur, since the macro processor searches for delimiters one at a time.

*** ERROR #313 DYNAMIC STORAGE (MACROS OR ARGUMENTS) OVERFLOW

Either a macro argument is too long (possibly because of a missing delimiter), or not enough space is available because of the number and size of macro definitions. All pending and active macros and INCLUDE's are popped and scanning continues in the primary source file.

*** ERROR #314 MACRO STACK OVERFLOW

The macro context stack has overflowed. This stack is 64 deep and contains an entry for each of the following:

1. Every currently active input file (primary source plus currently nested INCLUDE's).
2. Every pending macro call, that is, all calls to macros whose arguments are still being scanned.
3. Every active macro call, that is, all macros whose values or bodies are currently being read. Included in this category are various temporary strings used during the expansion of some built-in macro functions.

The cause of this error is excessive recursion in macro calls, expansions, or INCLUDE's. All pending and active macros and INCLUDE's are popped and scanning continues in the primary source file.

*** ERROR #315 INPUT STACK OVERFLOW

The input stack is used in conjunction with the macro stack to save pointers to strings under analysis. The cause and recovery is the same as for the macro stack overflow.

*** ERROR #317 PATTERN TOO LONG

An element of a pattern, an identifier or delimiter, is longer than 31 characters, or the total pattern is longer than 255 characters. The DEFINE is aborted and scanning continues from the point at which the error was detected.

*** ERROR #318 ILLEGAL METACHARACTER: "*char*"

The METACHAR built-in function has specified a character that cannot legally be used as a metacharacter: a blank, letter, digit, left or right parenthesis, or asterisk. The current metacharacter remains unchanged.

*** ERROR #319 UNBALANCED ")" IN ARGUMENT TO USER DEFINED MACRO

During the scan of a user-defined macro, the parenthesis count went negative, indicating an unmatched right parenthesis. The macro function call is aborted and scanning continues from the point at which the error was detected.

*** ERROR #320 ILLEGAL ASCENDING CALL

Ascending calls are not permitted in the macro language. If a call is not complete when the end of a macro expansion is encountered, this message is issued and the call is aborted. A macro call beginning inside the body of a user-defined or built-in macro was incompletely contained inside that body, possibly because of a missing delimiter for the macro call.

## Control Error Messages

Control error messages are issued when something is wrong with a control line in the source file. Command language errors, when they occur in the invocation line or in a primary control line, are fatal. However, the errors listed below are not considered fatal. (See ASM51 FATAL ERRORS, described above.)

*** ERROR #400 MORE ERRORS DETECTED, NOT REPORTED

After 100 Macro or Control Errors on a given source line, this message is given and no more errors are reported for that line. Normal reporting resumes on the next source line. If the last error reported is a Control Error, then this message will be issued. (See errors 16 and 300.)

*** ERROR #401 BAD PARAMETER TO CONTROL

What appears to be the parameter to a control is not correctly formed. This may be caused by the parameter missing a right parenthesis or if the parentheses are not correctly nested.

*** ERROR #402 MORE THAN ONE INCLUDE CONTROL ON A SINGLE LINE

ASM51 allows a maximum of one INCLUDE control on a single line. If more than one appears on a line, only the first (leftmost) is included, the rest are ignored.

*** ERROR #403 ILLEGAL CHARACTER IN COMMAND

When scanning a command line, ASM51 encountered an invalid character.

This error can be caused for a variety of reasons. The obvious one is that a command line was simply mistyped. The following example is somewhat less obvious:

```
$TITLE('1)-GO')
```

The title parameter ends with the first right parenthesis, the one after the digit 1. The title string is "'1". The next character "-" is illegal and will get error 403. The next two characters, "GO", form a valid command (the abbreviation for GENONLY) which will cause the listing mode to be set. The final two characters "')" will each receive error 403.

*** ERROR #406 TOO MANY WORKFILES - ONLY FIRST TWO USED

This error occurs when you specify more than two devices in the parameters to the WORKFILES control. Only the first two are used and the remaining list of devices is ignored until the next right parenthesis.

*** ERROR #407 UNRECOGNIZED CONTROL OR MISPLACED PRIMARY CONTROL: *<control-name>*

The indicated control is not recognized as an ASM51 control in this context. It may be misspelled, mistyped, or incorrectly abbreviated.

A misplaced primary control is a likely cause of this error. Primary control lines must be at the start of the source file, preceding all non-control lines (even comments and blank lines).

*** ERROR #408 NO TITLE FOR TITLE CONTROL

This error is issued if the title control has no parameter. The new title will be a string of blanks.

*** ERROR #409 NO PARAMETER ALLOWED WITH ABOVE CONTROL

The following controls do not have parameters:

| | | |
|---|---|---|
| EJECT | NOOBJECT | MACRO |
| SAVE | NOPRINT | NOMACRO |
| RESTORE | NOPAGING | PAGING |
| LIST | DEBUG | SYMBOLS |
| NOLIST | NODEBUG | NOSYMBOLS |
| GENONLY | NOERRORPRINT | XREF |
| GEN | NOGEN | NOXREF |

If one is included, then this error will be issued, and the parameter will be ignored.

*** ERROR #410 SAVE STACK OVERFLOW

The SAVE stack has a depth of eight. If the program tries to save more than eight levels, then this message will be printed.

*** ERROR #411 SAVE STACK UNDERFLOW

If a RESTORE command is executed and there has been no corresponding SAVE command, then this error will be printed.

*** ERROR #413 PAGEWIDTH BELOW MINIMUM, SET TO 72

The minimum pagewidth value is 72. If a pagewidth value less than 72 is given, 72 becomes the new pagewidth.

*** ERROR #414 PAGELENGTH BELOW MINIMUM, SET TO 10

The minimum number of printed lines per page is 10. If a value less than 10 is requested, 10 becomes the new pagelength.

*** ERROR #415 PAGEWIDTH ABOVE MAXIMUM, SET TO 132

The maximum pagewidth value is 132. If a value greater than 132 is requested then, 132 becomes the new pagewidth.

## Special Assembler Error Messages

Error messages in the 800's should never occur. If you get one of these error messages, please notify Intel Corporation via the Software Problem Report included with this manual. All of these errors are listed below:

*** ERROR #800 UNRECOGNIZED ERROR MESSAGE NUMBER
*** ERROR #801 SOURCE FILE READING UNSYNCHRONIZED
*** ERROR #802 INTERMEDIATE FILE READING UNSYNCHRONIZED
*** ERROR #803 BAD OPERAND STACK POP REQUEST
*** ERROR #804 PARSE STACK UNDERFLOW
*** ERROR #805 INVALID EXPRESSION STACK CONFIGURATION

## Fatal Error Messages

Errors numbered in the 900's are fatal errors. They are marked by the line

"*** FATAL ERROR ***"

preceding the message line. Assembly of the source code is halted. The remainder of the program is scanned and listed, but not assembled.

*** ERROR #900 USER SYMBOL TABLE SPACE EXHAUSTED

You must either eliminate some symbols from your program, or if you don't use macros, the NOMACRO control will free additional symbol table space.

*** ERROR #901 PARSE STACK OVERFLOW
*** ERROR #902 EXPRESSION STACK OVERFLOW

This error will be given only for grammatical entities far beyond the complication seen in normal programs.

*** ERROR #903 INTERMEDIATE FILE BUFFER OVERFLOW

This error indicates that a single source line has generated an excessive amount of information for pass 2 processing. In practical programs, the limit should be reached only for lines with a gigantic number of errors — correcting other errors should make this one go away.

*** ERROR #904 USER NAME TABLE SPACE EXHAUSTED

This error indicates that the sum of the number of characters used to define the symbols contained in a source file exceeds the macro processor's capacity. Use shorter symbol names, or reduce the number of symbols in the program.

# Assembler Listing File Format

The MCS-51 assembler, unless overridden by controls, outputs two files: an object file and a listing file. The object file contains the machine code in Absolute Hex Format. It is suitable for programming either an 8751 (EPROM version of 8051) or an Intel EPROM memory component.

```
MCS-51 MACRO ASSEMBLER    THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT                    DEC.(18)   PAGE    1

I IS-II MCS-51 MACRO ASSEMBLER V1.0
NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM51 :F1:EXAMP7.SRC XREF SYMBOLS             & this is the example for chapter 7
                                            TITLE(THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT) &
                                            DATE(DEC.(18)) NOOBJECT   & I don't need the object file
                                            ERRORPRINT                & show the errors at the console

LOC  OBJ            LINE     SOURCE

                       1
                       2              ORG 1000
03E8                   3 +1  $GEN
                       4     %add16(DPH,DPL,#(HIGH $),#(LOW $),DPH,DPL)
                       5 +1
                       6 +1  MOV A,%XLOW
03E8 E582              7 +2        DPL
                       8 +1  ADD A,%YLOW
03EA 24EA              9 +2         #(LOW $)
                      10 +1  MOV %SUMLOW
03EC F582             11 +2       DPL,A
                      12 +1  MOV A,%XHIGH
03EE E583             13 +2         DPH
                      14 +1  ADDC A,%YHIGH
03F0 3403             15 +2          #(HIGH $)
                      16 +1  MOV %SUMHIGH
03F2 F583             17 +2       DPH,A
                      18 +1
                      19
                      20
                      21
                      22 +1  $GENONLY
                      23 +1
03F4 E582             24 +2  MOV A,DPL
03F6 24F6             25 +2  ADD A,#(LOW $)
03F8 F582             26 +2  MOV DPL,A
03FA E583             27 +2  MOV A,DPH
03FC 3403             28 +2  ADDC A,#(HIGH $)
03FE F583             29 +2  MOV DPH,A
                      30 +1
                      31
                      32
                      33
                      34 +1  $NOGEN
                      35     %add16(DPH,DPL,#(HIGH $),#(LOW $),DPH,DPL)
                      43
                      44
                      45
                      46     $ EJECT
```

Figure 7-1. Example Listing File Format

937-19

```
MCS-51 MACRO ASSEMBLER    THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT                    DEC,(18)   PAGE    2

LOC  OBJ            LINE     SOURCE

                      47
                      48     ;   The next two lines will generate errors
                      49       ERRONEOUS EQUALS     55
***                            ^
*** ERROR #1, LINE #49 (0), SYNTAX ERROR
                      50       SECOND_ERROR SET   ERRONEOUS
*** ERROR #18, LINE #50 (49), (PASS 2) UNDEFINED SYMBOL
  0802                51             STORE SET 6*(5+$) -100 AND 0F0FH
  0803                52             JOHN            EQU STORE + 1
----                  53             BSEG
                      54     JOHN_JOHN: ; USE LABEL IN BSEG TO DEFINE BIT ADDRESS
----                  55             DSEG
007F                  56             ORG 127
                      57     MEMTOP:   ; USE LABEL IN DSEG TO DEFINE DATA ADDRESS
----                  58             CSEG
040C C200             59             CLR JOHN_JOHN      ;  USE OF LABEL IN BSEG AS BIT ADDRESS
040E F57F             60             MOV MEMTOP,A       ;  USE OF LABEL IN DSEG AS DATA ADDRESS
0410 54484953         61     TYPE_STRING:    DB      'THIS STRING WILL LIE IN CODE MEMORY',0DH,0AH,00H
0414 20535452
0418 494E4720
041C 57494C4C
0420 204C4945
0424 20494E20
0428 434F4445
042C 204D454D
0430 4F5259
0433 0D
0434 0A
0435 00
                      62     ; ********************************************************************
                      63     ; This routine converts BCD to binary and binary to BCD.
                      64     ; It uses rotate to simulate multiplication.
                      65     ; ASSUME:
                      66     ;       Register 1 contains address of value to be converted
                      67     ;       If BCD to binary conversion,
                      68     ;                                       Carry = 0
                      69     ;                                       high order nibble (bits 4-7) of
                      70     ;                                       memory contains high order digit
                      71     ;                                     - and, low order nibble (bits 0-3)
                      72     ;                                       of memory contains low order
                      73     ;                                       digit.
                      74     ;       If binary to BCD conversion,
                      75     ;                                       Carry = 1
                      76     ;                                       value in memory is between 0 and 99
                      77     ; OUTPUT:
                      78     ;       Register 1 will be unchanged and address the converted value
                      79     ;       Carry = 0
                      80     ; SIDE EFFECTS:
                      81     ;             The contents of Accumulator and Register 2 and 3 will be
                      82     ;             changed
                      83     ;
                      84     ;*******************************************************************************
0100                  85             ORG     100H
0100 E7               86     CONVRT: MOV     A,@R1
0101 400F             87             JC      BINBCD
```

Figure 7-1. Example Listing File Format (Cont'd.)

937-20

```
MCS-51 MACRO ASSEMBLER    THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT                    DEC.(18)   PAGE    3

LOC  OBJ             LINE     SOURCE

0103 54F0              88     BCDBIN: ANL     A,#0F0H                 ; Mask out low order digit
0105 C4                89             SWAP    A                       ; Move high digit into low order nibble of accumula
                                tor
0106 FA                90             MOV     R2,A                    ; Store X in R2
                       91     ;  Multiply digit by 10
0107 23                92             RL      A                       ; X * 2
0108 23                93             RL      A                       ; 2X * 2
0109 23                94             RL      A                       ; 4X * 2
010A 2A                95             ADD     A,R2                    ; 8X + X
010B 2A                96             ADD     A,R2                    ; 9X + X
010C C7                97             XCH     A,@R1                   ; Store converted high order digit and get BCD valu
                                e so that low order digit can be converted
010D 540F              98             ANL     A,#0FH                  ; Mask out high order digit
010F 27                99             ADD     A,@R1                   ; Add high digit and low digit
0110 F7               100             MOV     @R1,A                   ; Place result at R1
0111 22               101             RET
                      102     ;  Begin binary to BCD conversion accumulator contains binary value
0112 7B00             103     BINBCD: MOV     R3,#0
0114 7AF6             104             MOV     R2,#(-10)               ; Load -10 to simulate division by -10
0116 F7               105     DIV_:   MOV     @R1,A                   ; Store intermediate remainder
0117 0B               106             INC     R3                      ; Count each subtraction
0118 2A               107             ADD     A,R2                    ; Subtract 10
0119 40FB             108             JC      DIV_                    ; If no carry A was less than 10
011B 1B               109             DEC     R3                      ; Last subtraction doesn't count
011C EB               110             MOV     A,R3                    ; Get quotient
011D C4               111             SWAP    A                       ; Place in high order nibble
011E 47               112             ORL     A,@R1                   ; Place remainder in low order nibble
011F F7               113             MOV     @R1,A                   ; Place result at R1
0120 22               114             RET
0000                  115             ORG     0
0000 2100             116             AJMP    CONVRT
                      117             END
```

```
MCS-51 MACRO ASSEMBLER    THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT                    DEC.(18)   PAGE    4

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

NAME             TYPE     VALUE AND REFERENCES

BCDBIN. . . . L CSEG   0103H  88#
BINBCD. . . . L CSEG   0112H  87 103#
CONVRT. . . . L CSEG   0100H  86# 116
DIV_. . . . . L CSEG   0116H  105# 108
DPH . . . . . N DSEG   0083H  13 17 27 29 39 41
DPL . . . . . N DSEG   0082H  7 11 24 26 36 38
EQUALS. . . . ------   --UNDEFINED--  49
ERRONEOUS . . ------   --UNDEFINED--  49 50
JOHN. . . . . N        0803H  52#
JOHN_JOHN . . L BSEG   0000H  54# 59
MEMTOP. . . . L DSEG   007FH  57# 60
SECOND_ERROR. ------   --UNDEFINED--  50
STORE . . . . N        0802H  51# 52
TYPE_STRING . L CSEG   0410H  61#


ASSEMBLY COMPLETE, 2 ERRORS FOUND (50)
```

Figure 7-1. Example Listing File Format (Cont'd.)

937-21

The list file contains a formatted copy of your source code with page headers, and, if requested through controls (SYMBOLS or XREF), a symbol table.

## List File Heading

Every page has a header on the first line. It contains the words "MCS-51 MACRO ASSEMBLER" followed by the title, if specified. On the extreme right hand side of the header, the date (if specified) and the page number is printed.

In addition to the normal header, the first page of listing includes a salutation shown in figure 7-2. In it the assembler's version number is shown, the file name of the object file, if any, and the invocation line. The entire invocation line is displayed even if it extends over several lines.

```
MCS-51 MACRO ASSEMBLER    THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT                    DEC.(18)   PAGE    1

I IS-II MCS-51 MACRO ASSEMBLER V1.0
NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM51 :F1:EXAMP7.SRC XREF SYMBOLS             & this is the example for chapter 7
                                            TITLE(THIS IS AN EXAMPLE OF THE LISTING FILE FORMAT) &
                                            DATE(DEC.(18)) NOOBJECT   & I don't need the object file
                                            ERRORPRINT                & show the errors at the console
```

**Figure 7-2. Example Heading** 937-22

## Source Listing

The main body of the listing file is the formatted source listing. A section of formatted source is shown in figure 7-3.

```
LOC  OBJ            LINE     SOURCE

                      47
                      48     ;   The next two lines will generate errors
                      49       ERRONEOUS EQUALS     55
***                            ^
*** ERROR #1, LINE #49 (0), SYNTAX ERROR
                      50       SECOND_ERROR SET   ERRONEOUS
*** ERROR #18, LINE #50 (49), (PASS 2) UNDEFINED SYMBOL
  0802                51             STORE SET 6*(5+$) -100 AND 0F0FH
  0803                52             JOHN            EQU STORE + 1
----                  53             BSEG
                      54     JOHN_JOHN: ; USE LABEL IN BSEG TO DEFINE BIT ADDRESS
----                  55             DSEG
007F                  56             ORG 127
                      57     MEMTOP:   ; USE LABEL IN DSEG TO DEFINE DATA ADDRESS
----                  58             CSEG
040C C200             59             CLR JOHN_JOHN      ;  USE OF LABEL IN BSEG AS BIT ADDRESS
040E F57F             60             MOV MEMTOP,A       ;  USE OF LABEL IN DSEG AS DATA ADDRESS
0410 54484953         61     TYPE_STRING:    DB      'THIS STRING WILL LIE IN CODE MEMORY',0DH,0AH,00H
0414 20535452
0418 494E4720
041C 57494C4C
0420 204C4945
0424 20494E20
0428 434F4445
042C 204D454D
0430 4F5259
0433 0D
0434 0A
0435 00
```

**Figure 7-3. Example Source Listing** 937-23

The format for each line in the listing file depends on the source line that appears on it. Instruction lines contain 4 fields. The name of each field and its meaning is shown in the list below:

- LOC shows the location (code address) of the first byte of the instruction. The value is displayed in hexadecimal.
- OBJ shows the actual machine code produced by the instruction, also displayed in hexadecimal.
- LINE shows the INCLUDE nesting level, if any, the number of source lines from the top of the program, and the macro nesting level, if any. All values in this field are displayed in decimal numbers.
- SOURCE shows the source line as it appears in the file. This line may be extended onto the subsequent lines in the listing file.

DB or DW directives are formatted similarly to instruction lines, except the OBJ field shows the data values placed in memory. All data values are shown. If the expression list is long, then it may take several lines in the listing file to display all of the values placed in memory. The extra lines will only contain the LOC and OBJ fields.

The directives that affect the location counter without initializing memory (e.g., ORG, DBIT, or DS) do not use the OBJ field, but the new value of the location counter is shown in the LOC field.

The SET and EQU directives do not have a LOC or OBJ field. In their place the assembler lists the value that the symbol is set to. If the symbol is defined to equal one of the registers, then 'REG' is placed in this field. The remainder of the directive line is formatted in the same way as the other directives.

## Format for Macros and INCLUDE Files

The format for lines generated by a macro call varies with the macro listing mode (GEN, GENONLY, or NOGEN). Figure 7-4 shows the format of the call macro calls listed with each of these modes in effect. In all three calls the same instructions are encoded, the only difference is in the listing of the macro call. Note the macro nesting level is shown immediately to the right of the line number.

```
                          3 +1  $GEN
                          4     %add16(DPH,DPL,#(HIGH $),#(LOW $),DPH,DPL)
                          5 +1
                          6 +1  MOV A,$XLOW
03E8 E582                 7 +2        DPL
                          8 +1  ADD A,$YLOW
03EA 24EA                 9 +2        #(LOW $)
                         10 +1  MOV $SUMLOW
03EC F582                11 +2      DPL,A
                         12 +1  MOV A,$XHIGH
03EE E583                13 +2        DPH
                         14 +1  ADDC A,$YHIGH
03F0 3403                15 +2         #(HIGH $)
                         16 +1  MOV $SUMHIGH
03F2 F583                17 +2      DPH,A
                         18 +1
                         19
                         20
                         21
                         22 +1  $GENONLY
                         23 +1
03F4 E582                24 +2  MOV A,DPL
03F6 24F6                25 +2  ADD A,#(LOW $)
03F8 F582                26 +2  MOV DPL,A
03FA E583                27 +2  MOV A,DPH
03FC 3403                28 +2  ADDC A,#(HIGH $)
03FE F583                29 +2  MOV DPH,A
                         30 +1
                         31
                         32
                         33
                         34 +1  $NOGEN
                         35     %add16(DPH,DPL,#(HIGH $),#(LOW $),DPH,DPL)
                         43
```

**Figure 7-4. Examples of Macro Listing Modes** 937-24

General control lines that appear in the source are interpreted by ASM51's macro processor and, as such, they are given a macro nesting level value. It is displayed immediately to the right of the line number. Lines added to the program as a result of the INCLUDE control are formatted just as if they appeared in the original source file, except the INCLUDE nesting level is displayed immediately to the left of the line number.

The control line shown below has both an INCLUDE nesting level and a macro nesting level. The INCLUDE nesting level is preceded by a equal sign '=', and the macro nesting level is preceded by a plus sign '+'.

```
LOC    OBJ              LINE        SOURCE

                =1       101  +1  $   SAVE    NOLIST
```

## Symbol Table

The symbol table is a list of all symbols defined in the program along with status information about the symbol. Any predefined symbols used will also be listed in the symbol table. If the XREF control is used the symbol table will contain information about where the symbol was used in the program.

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


NAME             TYPE     VALUE AND REFERENCES

BCDBIN. . . . L CSEG   0103H  88#
BINBCD. . . . L CSEG   0112H  87 103#
CONVRT. . . . L CSEG   0100H  86# 116
DIV_. . . . . L CSEG   0116H  105# 108
DPH . . . . . N DSEG   0083H  13 17 27 29 39 41
DPL . . . . . N DSEG   0082H  7 11 24 26 36 38
EQUALS. . . . ------   --UNDEFINED--  49
ERRONEOUS . . ------   --UNDEFINED--  49 50
JOHN. . . . . N        0803H  52#
JOHN_JOHN . . L BSEG   0000H  54# 59
MEMTOP. . . . L DSEG   007FH  57# 60
SECOND_ERROR. ------   --UNDEFINED--  50
STORE . . . . N        0802H  51# 52
TYPE_STRING . L CSEG   0410H  61#

ASSEMBLY COMPLETE, 2 ERRORS FOUND (50)
```

**Figure 7-5. Example Symbol Table Listing** 937-25

The status information includes a NAME field, a TYPE field (Label 'L' or Name 'N' defined by SET, EQU, BIT, etc.) and a VALUE field. If it is a label, then the segment mode will also be shown. If it is a name, it will show if the symbol was set to a register value at the end of the program. The VALUE field will show the value of the symbol when assembly was completed.

If the XREF control is used, then the symbol table listing will also contain all of the line numbers of each line of code that the symbol was used. If the value of the symbol was changed or defined on a line, then that line will have a hash mark (#) following it. The line numbers are displayed in decimal.

If an inordinate number of symbol references are generated by your program, it may be impossible for the assembler to produce a complete XREF table for your entire program. In that event, the following warning message is issued at the head of the symbol table:

*** WARNING, XREFS ABANDONED AT LINE #*line*

The XREF listing will be valid up to the specified line.

# APPENDIX A
# ASSEMBLY LANGUAGE BNF GRAMMAR

This appendix contains a Backus-Naur Form (BNF) grammar for all of the MCS-51 Assembly Language Constructions. It does not include the grammar for the macro facility. (See Chapter 5 and Appendix F.) Although BNF grammar is designed to define only syntax, the metasymbols and language breakdown have been selected to show the semantics of the language.

To simplify the grammar presented here, we have not defined all of the nuances of the language as rigorously as a complete BNF grammar would require. These exceptions are listed below.

- There are two types of controls, primary and general. A control line containing a primary control must be the first line in a program, or only preceded by other control lines.
- Some assembler directives may be used only while certain segment modes are in effect (e.g., the bit segment must be active when a DBIT directive is used).
- Operator precedence in expressions has not been defined.
- Symbol typing conventions are not identified.
- In some of the definitions we have used a few words of description, contained in double quotes.
- The ASCII string argument to the TITLE and DATE controls must either contain balanced parentheses or no parentheses at all.
- There has been no attempt to show the logical blanks (spaces or tabs) that separate the fields on a line.
- The symbol NULL is used to show that a meta-symbol may evaluate to nothing.
- Except within character strings, ASM51 makes no distinction between upper and lower case characters. All terminal symbols have been shown in upper case, but you can use upper or lower case in your source code (including within hex constants).

```
<Assembly Language Program>  ::= <Statement List> <End Statement>
<Statement List>             ::= <Statement> <Statement List> | NULL
<End Statement>              ::= END <Comment> <CRLF>
<Statement>                  ::= <Control Line> | <Instruction Line> |
                                 <Directive Line>
<Control Line>               ::= $ <Control List> <CRLF>
<Control List>               ::= <Control> <Control List> | NULL
<Control>                    ::= DATE(<ASCII String>) | DA(<ASCII String>) |
                                 DEBUG | DE |
                                 NODEBUG | NODE |
                                 EJECT | EJ |
                                 ERRORPRINT(<Filename>) | EP(<Filename>) | ERRORPRINT | EP |
                                 NOERRORPRINT | NOEP |
                                 GENONLY | GO |
                                 NOGEN | NOGE |
                                 GEN | GE |
                                 INCLUDE(<Filename>) | IC(<Filename>) |
                                 LIST | LI |
                                 NOLIST | NOLI |
                                 MACRO | MR |
                                 NOMACRO | NOMR |
                                 OBJECT(<Filename>) | OJ(<Filename>) | OBJECT | OJ |
                                 NOOBJECT | NOOJ |
                                 PAGING | PI |
                                 NOPAGING | NOPI |
                                 PAGELENGTH(<Constant>) | PL(<Constant>) |
                                 PAGEWIDTH(<Constant>) | PW(<Constant>) |
                                 PRINT(<Filename>) | PR(<Filename>) | PRINT | PR |
                                 NOPRINT | NOPR |
                                 SAVE | SA |
                                 RESTORE | RS |
                                 SYMBOLS | SB |
                                 NOSYMBOLS | NOSB |
                                 TITLE(<ASCII String>) | TT(<ASCII String>) |
                                 WORKFILES(<Drive name>,<Drive name>) | WORKFILES(<Drive name>) |
                                 WF(<Drive name>,<Drive name>) | WF(<Drive name>) |
                                 XREF | XR |
                                 NOXREF | NOXR
<Instruction Line>           ::= <Label> <Instruction> <Comment> <CRLF>
<Label>                      ::= <Symbol Name>: |
                                 NULL
<Comment>                    ::= ;<ASCII String> | NULL
<Instruction>                ::= <Arithmetic Instruction> |
                                 <Multiplication Instruction> |
                                 <Logic Instruction> |
                                 <Data Move Instruction> |
                                 <Jump Instruction> |
                                 <Subroutine Instruction> |
                                 <Special Instruction> |
                                 NULL
<Arithmetic Instruction>     ::= <Arithmetic Mnemonic> <Accumulator>,<Byte Source>
<Arithmetic Mnemonic>        ::= ADD |
                                 ADDC |
                                 SUBB
```

```
<Multiplication Instruction>    ::= DIV AB |
                                    MUL AB

<Logic Instruction>             ::= <Accumulator Logic Instruction> |
                                    <Data Address Logic Instruction> |
                                    <Bit Logic Instruction>

<Accumulator Logic Instruction> ::= <Logic Mnemonic> <Accumulator>,<Byte Source>

<Data Address Logic Instruction> ::= <Logic Mnemonic> <Data Address>,<Accumulator> |
                                     <Logic Mnemonic> <Data Address>,<Immediate Data>

<Logic Mnemonic>                ::= ANL |
                                    ORL |
                                    XRL

<Bit Logic Instruction>         ::= ANL C,<Bit Address> |
                                    ANL C,/<Bit Address> |
                                    ORL C,<Bit Address> |
                                    ORL C,/<Bit Address>

<Data Move Instruction>         ::= <Bit Move Instruction> |
                                    <Byte Move Instruction> |
                                    <External Move Instruction> |
                                    <Code Move Instruction> |
                                    <Exchange Instruction> |
                                    <Data Pointer Load>

<Bit Move Instruction>          ::= MOV C,<Bit Address> |
                                    MOV <Bit Address>,C

<Byte Move Instruction>         ::= MOV <Accumulator>,<Byte Source> |
                                    <Indirect Address Move> |
                                    <Data Address Move> |
                                    <Register Move>

<Indirect Address Move>         ::= MOV <Indirect Address>,<Accumulator> |
                                    MOV <Indirect Address>,<Immediate Data> |
                                    MOV <Indirect Address>,<Data Address>

<Data Address Move>             ::= MOV <Data Address>,<Accumulator> |
                                    MOV <Data Address>,<Byte Source>

<Register Move>                 ::= MOV <Register>,<Accumulator> |
                                    MOV <Register>,<Immediate Data> |
                                    MOV <Register>,<Data Address> |

<External Move Instruction>     ::= MOVX <Accumulator>,<Indirect Address> |
                                    MOVX <Indirect Address>,<Accumulator> |
                                    MOVX <Accumulator>,@DPTR |
                                    MOVX @DPTR,<Accumulator>

<Code Move Instruction>         ::= MOVC <Accumulator>,@A+PC |
                                    MOVC <Accumulator>,@A+DPTR

<Exchange Instruction>          ::= XCHD <Accumulator>,<Indirect Address> |
                                    XCH <Accumulator>,<Byte Destination>

<Data Pointer Load>             ::= MOV DPTR,<Immediate Data>

<Jump Instruction>              ::= <Decrement Jump> |
                                    <Compare Jump> |
                                    <Test Jump> |
                                    <Always Jump>

<Decrement Jump>                ::= DJNZ <Register>,<Code Address> |
                                    DJNZ <Data Address>,<Code Address>
```

<Compare Jump> ::= CJNE <Accumulator>,<Immediate Data,<Code Address> |
CJNE <Accumulator>,<Data Address>,<Code Address> |
CJNE <Indirect Address>,<Immediate Data>,<Code Address> |
CJNE <Register>,<Immediate Data>,<Code Address>

<Test Jump> ::= JC <Code Address> |
JNC <Code Address> |
JZ <Code Address> |
JNZ <Code Address> |
JB <Bit Address>,<Code Address> |
JBC <Bit Address>,<Code Address> |
JNB <Bit Address>,<Code Address>

<Always Jump> ::= SJMP <Code Address> |
AJMP <Code Address> |
LJMP <Code Address> |
JMP <Code Address> |
JMP @A + DPTR

<Subroutine Instruction> ::= <Call Instruction> |
<Return Instruction>

<Call Instruction> ::= ACALL <Code Address> |
LCALL <Code Address> |
CALL <Code Address>

<Return Instruction> ::= RET |
RETI

<Special Instruction> ::= <Increment Instruction> |
<Decrement Instruction> |
<Accumulator Modify Instruction> |
<Bit Modify Instruction> |
<Stack Instruction> |
NOP

<Increment Instruction> ::= INC <Accumulator> |
INC DPTR |
INC <Byte Destination>

<Decrement Instruction> ::= DEC <Accumulator> |
DEC <Byte Destination>

<Accumulator Modify Instruction> ::= <Accumulator Modify Mnemonic> <Accumulator>

<Accumulator Modify Mnemonic> ::= CLR |
CPL |
DA |
SWAP |
RL |
RR |
RLC |
RRC

<Bit Modify Instruction> ::= <Bit Modify Mnemonic> <Bit Destination>

<Bit Modify Mnemonic> ::= SETB |
CLR |
CPL

<Stack Instruction> ::= POP <Data Address> |
PUSH <Data Address>

<Directive Line> ::= <Directive Statement><Comment><CRLF>

```
<Directive Statement>          ::= <Org Statement> |
                                   <Symbol Definition Statement> |
                                   <Segment Select Statement> |
                                   <Label><Space Allocation Statement> |
                                   <Label><Memory Initialization Statement>

<Org Statement>                ::= ORG <Expression>

<Symbol Definition Statement>  ::= <Symbol> EQU <Expression> |
                                   <Symbol> EQU <Symbol Register> |
                                   <Symbol> SET <Expression> |
                                   <Symbol> SET <Symbol Register> |
                                   <Symbol> DATA <Expression> |
                                   <Symbol> XDATA <Expression> |
                                   <Symbol> BIT <Bit Address> |

<Segment Select Statement>     ::= BSEG |
                                   CSEG |
                                   DSEG |
                                   XSEG

<Space Allocation Statement>   ::= DS <Expression> |
                                   DBIT <Expression>

<Memory Initialization Statement> ::= DB <Expression List>
                                        "ASCII character strings, as items in a DB expression list,
                                        may be arbitrarily long."
                                   DW <Expression List>
                                        "ASCII character strings, as items in a DW expression list,
                                        must be no more than two characters long."

<Filename>                     ::= "ISIS-II Filename"

<Drive name>                   ::= "ISIS-II Drive Identifier"

<ASCII String>                 ::= "Any Printable ASCII Character"

<Constant>                     ::= <Decimal Digit> |
                                   <Decimal Digit><Constant>

<Decimal Digit>                ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<CRLF>                         ::= "ASCII Carriage Return Line Feed Pair"

<Byte Source>                  ::= <Indirect Address> |
                                   <Data Address> |
                                   <Immediate Data> |
                                   <Register>

<Indirect Address>             ::= @R0 | @R1 |
                                   @<Symbol>

<Data Address>                 ::= <Expression>

<Immediate Data>               ::= #<Expression>

<Register>                     ::= R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
                                   <Symbol>

<Byte Destination>             ::= <Indirect Address> |
                                   <Data Address> |
                                   <Register>

<Accumulator>                  ::= A | <Symbol>

<Symbol Register>              ::= <Accumulator> | <Register>

<Symbol>                       ::= <Alphabet><Alphanumeric List> | <Special Char><Alphanumeric List>

<Alphabet>                     ::= A | B | C | D | E | F | G | H | I |
                                   J | K | L | M | N | O | P | Q | R |
                                   S | T | U | V | W | X | Y | Z
```

```
<Special Char>          ::= _ "Underscore" |
                            ?

<Alphanumeric List>     ::= <Alphanumeric><Alphanumeric List> |
                            NULL

<Alphanumeric>          ::= <Alphabet> |
                            <Decimal Digit> |
                            <Special Char>

<Bit Destination>       ::= C |
                            <Bit Address>

<Bit Address>           ::= <Expression> |
                            <Expression>.<Expression>

<Code Address>          ::= <Expression>

<Expression List>       ::= <Expression> |
                            <Expression>,<Expression List>

<Expression>            ::= <Symbol> |
                            <Number> |
                            <Expression><Operator><Expression> |
                            (<Expression>) |
                            +<Expression> |
                            -<Expression> |
                            HIGH <Expression> |
                            LOW <Expression>

<Operator>              ::= + | - | / | MOD | SHL | SHR |
                            EQ | = | NE | <> | LT | < | LE |
                            <= | GT | > | GE | >= | AND | OR | XOR

<Number>                ::= <Hex Number> |
                            <Decimal Number> |
                            <Octal Number> |
                            <Binary Number>

<Hex Number>            ::= <Decimal Digit><Hex Digit String> H

<Hex Digit String>      ::= <Hex Digit><Hex Digit String> |
                            NULL

<Hex Digit>             ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
                            8 | 9 | A | B | C | D | E | F

<Decimal Number>        ::= <Decimal Digit String> D |
                            <Decimal Digit String>

<Decimal Digit String>  ::= <Decimal Digit > |
                            <Decimal Digit><Decimal Digit String>

<Octal Number>          ::= <Octal Digit String> O |
                            <Octal Digit String> O

<Octal Digit String>    ::= <Octal Digit > |
                            <Octal Digit><Octal Digit String>

<Octal Digit>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<Binary Number>         ::= <Binary Digit String> B

<Binary Digit String>   ::= <Binary Digit> |
                            <Binary Digit><Binary Digit String>

<Binary Digit>          ::= 0 | 1
```

# APPENDIX B
# INSTRUCTION SET SUMMARY

This appendix contains two tables: the first identifies all of the 8051's instructions in alphabetical order; the second table lists the instructions according to their hexadecimal opcodes and lists the assembly language instructions that produced that opcode.

The alphabetical listing also includes documentation of the bit pattern, flags affected, number of machine cycles per execution and a description of the instructions operation and function. The list below defines the conventions used to identify operation and bit patterns.

**Abbreviations and Notations Used**

| | |
|---|---|
| A | Accumulator |
| AB | Register Pair |
| B | Multiplication Register |
| *bit address* | 8051 bit address |
| *page address* | 11-bit code address within 2K page |
| *relative offset* | 8-bit 2's complement offset |
| C | Carry Flag |
| *code address* | Absolute code address |
| *data* | Immediate data |
| *data address* | On-chip 8-bit RAM address |
| DPTR | Data pointer |
| PC | Program Counter |
| R*r* | Register (*r*=0-7) |
| SP | Stack pointer |
| *high* | High order byte |
| *low* | Low order byte |
| i-j | Bits i through j |
| .n | Bit n |
| *aaa aaaaaaaa* | Absolute page address encoded in instruction and operand byte |
| *bbbbbbbb* | Bit address encoded in operand byte |
| *dddddddd* | Immediate data encoded in operand byte |
| *iiiiiiii* | One byte of a 16-bit address encoded in operand byte |
| *mmmmmmmm* | Data address encoded in operand byte |
| *oooooooo* | Relative offset encoded in operand byte |
| *r* or *rrr* | Register identifier encoded in operand byte |
| AND | Logical AND |
| NOT | Logical complement |
| OR | Logical OR |
| XOR | Logical exclusive OR |
| + | Plus |
| – | Minus |
| / | Divide |
| * | Multiply |
| (X) | The contents of X |
| ((X)) | The memory location addressed by (X) (The contents of X) |
| = | Is equal to |
| < > | Is not equal to |
| < | Is less than |
| > | Is greater than |
| ← | Is replaced by |

## Table B-1. Instruction Set Summary

| Mnemonic Operation | Cycles | Binary Code | Flags P OV AC C | Function |
|---|---|---|---|---|
| ACALL *code addr*<br>(PC) ← (PC) + 2<br>(SP) ← (SP) + 1<br>((SP)) ← (PC) *low*<br>(SP) ← (SP) + 1<br>((SP)) ← (PC) *high*<br>(PC) 0-10 ← *page address* | 2 | a a a 1 0 0 0 1<br>a a a a a a a a | | Push PC on stack, and replace low order 11 bits with low order 11 bits of code address. |
| ADD A,*#data*<br>(A) ← (A) + *data* | 1 | 0 0 1 0 0 1 0 0<br>d d d d d d d d | P OV AC C | Add immediate data to A |
| ADD A,@R*r*<br>(A) ← (A) + ((R*r*)) | 1 | 0 0 1 0 0 1 1 r | P OV AC C | Add contents of indirect address to A |
| ADD A,R*r*<br>(A) ← (A) + (R*r*) | 1 | 0 0 1 0 1 r r r | P OV AC C | Add register to A |
| ADD A,*data addr*<br>(A) ← (A) + (*data address*) | 1 | 0 0 1 0 0 1 0 1<br>m m m m m m m m | P OV AC C | Add contents of data address to A |
| ADDC A,*#data*<br>(A) ← (A) + (C) + *data* | 1 | 0 0 1 1 0 1 0 0<br>d d d d d d d d | P OV AC C | Add C and immediate data to A |
| ADDC A,@R*r*<br>(A) ← (A) + (C) + ((R*r*)) | 1 | 0 0 1 1 0 1 1 r | P OV AC C | Add C and contents of indirect address to A |
| ADDC A,R*r*<br>(A) ← (A) + (C) + (R*r*) | 1 | 0 0 1 1 1 r r r | P OV AC C | Add C and register to A |
| ADDC A,*data addr*<br>(A) ← (A) + (C) + (*data address*) | 1 | 0 0 1 1 0 1 0 1<br>m m m m m m m m | P OV AC C | Add C and contents of data address to A |
| AJMP *code addr*<br>(PC) 0-10 ← *code address* | 2 | a a a 0 0 0 0 1<br>a a a a a a a a | | Replace low order 11 bits of PC with low order 11 bits code address |
| ANL A,*#data*<br>(A) ← (A) AND *data* | 1 | 0 1 0 1 0 1 0 0<br>d d d d d d d d | P | Logical AND immediate data to A |
| ANL A,@R*r*<br>(A) ← (A) AND ((R*r*)) | 1 | 0 1 0 1 0 1 1 r | P | Logical AND contents of indirect address to A |
| ANL A,R*r*<br>(A) ← (A) AND (R*r*) | 1 | 0 1 0 1 1 r r r | P | Logical AND register to A |
| ANL A,*data addr*<br>(A) ← (A) AND (*data address*) | 1 | 0 1 0 1 0 1 0 1<br>m m m m m m m m | P | Logical AND contents of data address to A |
| ANL C,*bit addr*<br>(C) ← (C) AND (*bit address*) | 2 | 1 0 0 0 0 0 1 0<br>b b b b b b b b | C | Logical AND bit to C |
| ANL C,/*bit addr*<br>(C) ← (C) AND NOT (*bit address*) | 2 | 1 0 1 1 0 0 0 0<br>b b b b b b b b | C | Logical AND complement of bit to C |
| ANL *data addr*,*#data*<br>(*data address*) ←<br>(*data address*) AND *data* | 2 | 0 1 0 1 0 0 1 1<br>m m m m m m m m<br>d d d d d d d d | | Logical AND immediate data to contents of data address |
| ANL *data addr*,A<br>(*data address*) ←<br>(*data address*) AND A | 1 | 0 1 0 1 0 0 1 0<br>m m m m m m m m | | Logical AND A to contents of data address |

## Table B-1. Instruction Set Summary (Cont'd.)

| Mnemonic<br>Operation | Cycles | Binary<br>Code | Flags<br>P OV AC C | Function |
|---|---|---|---|---|
| CJNE *@Rr,#data,code addr*<br>(PC) ← (PC) + 3<br>IF ((*Rr*)) < > *data*<br>THEN<br>(PC) ← (PC) + *relative offset*<br>IF ((*Rr*)) < *data*<br>THEN (C) ← 1<br>ELSE (C) ← 0 | 2 | 1 0 1 1 0 1 1 r<br>d d d d d d d d<br>o o o o o o o o | C | If immediate data and contents of indirect address are not equal, jump to code address |
| CJNE A,*#data,code addr*<br>(PC) ← (PC) + 3<br>IF (A) < > *data*<br>THEN<br>(PC) ← (PC) + *relative offset*<br>IF (A) < *data*<br>THEN (C) ← 1<br>ELSE (C) ← 0 | 2 | 1 0 1 1 0 1 0 0<br>d d d d d d d d<br>o o o o o o o o | C | If immediate data and A are not equal, jump to code address |
| CJNE A,*data addr,code addr*<br>(PC) ← (PC) + 3<br>IF (A) < > (*data address*)<br>THEN<br>(PC) ← (PC) + *relative offset*<br>IF (A) < (*data address*)<br>THEN (C) ← 1<br>ELSE (C) ← 0 | 2 | 1 0 1 1 0 1 0 1<br>m m m m m m m m<br>o o o o o o o o | C | If contents of data address and A are not equal, jump to code address |
| CJNE *Rr,#data,code addr*<br>(PC) ← (PC) + 3<br>IF (*Rr*) < > *data*<br>THEN<br>(PC) ← (PC) + *relative offset*<br>IF (*Rr*) < *data*<br>THEN (C) ← 1<br>ELSE (C) ← 0 | 2 | 1 0 1 1 1 r r r<br>d d d d d d d d<br>o o o o o o o o | C | If immediate data and register are not equal, jump to code address |
| CLR A<br>(A) ← 0 | 1 | 1 1 1 0 0 1 0 0 | P | Set A to zero (0) |
| CLR C<br>(C) ← 0 | 1 | 1 1 0 0 0 0 1 1 | C | Set C to zero (0) |
| CLR *bit addr*<br>(*bit address*) ← 0 | 1 | 1 1 0 0 0 0 1 0<br>b b b b b b b b | | Set bit to zero (0) |
| CPL A<br>(A) ← NOT (A) | 1 | 1 1 1 1 0 1 0 0 | P | Complements each bit in A |
| CPL C<br>(C) ← NOT (C) | 1 | 1 0 1 1 0 0 1 1 | C | Complement C |
| CPL *bit addr*<br>(*bit address*) ←<br>NOT (*bit address*) | 1 | 1 0 1 1 0 0 1 1<br>b b b b b b b b | | Complement bit |
| DA A<br>(See description in Chapter 3) | 1 | 1 1 0 1 0 1 0 0 | P C | Adjust A after a BCD add |
| DEC *@Rr*<br>((*Rr*)) ← ((*Rr*)) − 1 | 1 | 0 0 0 1 0 1 1 r | | Decrement contents of indirect address |
| DEC A<br>(A) ← (A) − 1 | 1 | 0 0 0 1 0 1 0 0 | P | Decrement A |
| DEC *Rr*<br>(*Rr*) ← (*Rr*) − 1 | 1 | 0 0 0 1 1 r r r | | Decrement register |
| DEC *data addr*<br>(*data address*) ←<br>(*data address*) − 1 | 1 | 0 0 0 1 0 1 0 1<br>m m m m m m m m | | Decrement contents of data address |
| DIV AB<br>(AB) ← (A) / (B) | 4 | 1 0 0 0 0 1 0 0 | P OV C | Divide A by B (multiplication register) |

## Table B-1. Instruction Set Summary (Cont'd.)

| Mnemonic<br>Operation | Cycles | Binary<br>Code | Flags<br>P OV AC C | Function |
|---|---|---|---|---|
| DJNZ Rr,*code addr*<br>(PC) ← (PC) + 2<br>(Rr) ← (Rr) - 1<br>IF (Rr) < > 0<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 1 1 0 1 1 r r r<br>o o o o o o o o | | Decrement register, if not zero (0), then jump to code address |
| DJNZ *data addr,code addr*<br>(PC) ← (PC) + 3<br>(*data address*) ←<br>(*data address*) – 1<br>IF (*data address*) = 0<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 1 1 0 1 0 1 0 1<br>mmmmmmmm<br>o o o o o o o o | | Decrement data address, if zero (0), then jump to code address |
| INC @Rr<br>((Rr)) ← ((Rr)) + 1 | 1 | 0 0 0 0 0 1 1 r | | Increment contents of indirect address |
| INC A<br>(A) ← (A) + 1 | 1 | 0 0 0 0 0 1 0 0 | P | Increment A |
| INC DPTR<br>(DPTR) ← (DPTR) + 1 | 1 | 1 0 1 0 0 0 1 1 | | Increment 16-bit data pointer |
| INC Rr<br>((R) ← (Rr) + 1 | 1 | 0 0 0 0 1 r r r | | Increment register |
| INC *data addr*<br>(*data address*) ←<br>(*data address*) + 1 | 2 | 0 0 0 0 0 1 0 1<br>mmmmmmmm | | Increment contents of data address |
| JB *bit addr,code addr*<br>(PC) ← (PC) + 3<br>IF (*bit address*) = 1<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 0 0 1 0 0 0 0 0<br>b b b b b b b b<br>o o o o o o o o | | If bit is one, n jump to code address |
| JBC *bit addr,code addr*<br>(PC) ← (PC) + 3<br>IF (*bit address*) = 1<br>THEN<br>(PC) ← (PC) + *relative offset*<br>(*bit address*) ← 0 | 2 | 0 0 0 1 0 0 0 0<br>b b b b b b b b<br>o o o o o o o o | | If bit is one, n clear bit and jump to code address |
| JC *code addr*<br>(PC) ← (PC) + 2<br>IF (C) = 1<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 0 1 0 0 0 0 0 0<br>o o o o o o o o | | If C is one, then jump to code address |
| JMP @A + DPTR<br>(PC) ← (A) + (DPTR) | 2 | 0 1 1 1 0 0 1 1 | | Add A to data pointer and jump to that code address |
| JNB *bit addr,code addr*<br>(PC) ← (PC) + 3<br>IF (*bit address*) = 0<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 0 0 1 1 0 0 0 0<br>b b b b b b b b<br>o o o o o o o o | | If bit is zero, n jump to code address |
| JNC *code addr*<br>(PC) + (PC) + 2<br>IF (C) = 0<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 0 1 0 1 0 0 0 0<br>o o o o o o o o | | If C is zero (0), n jump to code address |
| JNZ *code addr*<br>(PC) ← (PC) + 2<br>IF (A) < > 0<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 0 1 1 1 0 0 0 0<br>o o o o o o o o | | If A is not zero (0), n jump to code address |

## Table B-1. Instruction Set Summary (Cont'd.)

| Mnemonic<br>Operation | Cycles | Binary<br>Code | Flags<br>P OV AC C | Function |
|---|---|---|---|---|
| JZ *code addr*<br>(PC) ← (PC) + 2<br>IF (A) = 0<br>THEN<br>(PC) ← (PC) + *relative offset* | 2 | 0 1 1 0 0 0 0 0<br>o o o o o o o o | | If A is zero (0), then jump to code address |
| LCALL *code addr*<br>(PC) ← (PC) + 3<br>(SP) ← (SP) + 1<br>((SP)) ← ((PC)) *low*<br>(SP) ← (SP) + 1<br>((SP)) ← (PC) *high*<br>(PC) ← *code address* | 2 | 0 0 0 1 0 0 1 0<br>I I I I I I I I†<br>I I I I I I I I† | | Push PC on stack and replace entire PC value with code address |
| LJMP *code addr*<br>(PC) ← *code address* | 2 | 0 0 0 0 0 0 1 0<br>I I I I I I I I†<br>I I I I I I I I† | | Jump to code address |
| MOV @R*r*,#*data*<br>((R*r*)) ← *data* | 1 | 0 1 1 1 0 1 1 r<br>d d d d d d d d | | Move immediate data to indirect address |
| MOV @R*r*,A<br>((R*r*)) ← (A) | 1 | 1 1 1 1 0 1 1 r | | Move A to indirect address |
| MOV @R*r*,*data addr*<br>((R*r*)) ← (*data address*) | 2 | 1 0 1 0 0 1 1 r<br>mmmmmmmm | | Move contents of data address to indirect address |
| MOV A,#*data*<br>(A) ← *data* | 1 | 0 1 1 1 0 1 0 0<br>d d d d d d d d | P | Move immediate data to A |
| MOV A,@R*r*<br>(A) ← ((R*r*)) | 1 | 1 1 1 0 0 1 1 r | P | Move contents of indirect address to A |
| MOV A,R*r*<br>(A) ← (R*r*) | 1 | 1 1 1 0 1 r r r | P | Move register to A |
| MOV A,*data addr*<br>(A) ← (*data address*) | 1 | 1 1 1 0 0 1 0 1<br>mmmmmmmm | P | Move contents of data address to A |
| MOV C,*bit addr*<br>(C) ← (*bit address*) | 1 | 1 0 1 0 0 0 1 0<br>b b b b b b b b | C | Move bit to C |
| MOV DPTR,#*data*<br>(DPTR) ← *data* | 2 | 1 0 0 1 0 0 0 0<br>d d d d d d d d†<br>d d d d d d d d† | | Move two bytes of immediate data pointer |
| MOV R*r*,#*data*<br>(R*r*) ← *data* | 1 | 0 1 1 1 1 r r r<br>d d d d d d d d | | Move immediate data to register |
| MOV R*r*,A<br>(R*r*) ← (A) | 1 | 1 1 1 1 1 r r r | | Move A to register |
| MOV R*r*,*data addr*<br>(R*r*) ← (*data address*) | 2 | 1 0 1 0 1 r r r<br>mmmmmmmm | | Move contents of data address to register |
| MOV *bit addr*,C<br>(*bit address*) ← (C) | 2 | 1 0 0 1 0 0 1 0<br>b b b b b b b b | | Move C to bit |
| MOV *data addr*,#*data*<br>(*data address*) ← *data* | 2 | 0 1 1 1 0 1 0 1<br>mmmmmmmm<br>d d d d d d d d | | Move immediate data to data address |
| MOV *data addr*,@R*r*<br>(*data address*) ← ((R*r*)) | 2 | 1 0 0 0 0 1 1 r<br>mmmmmmmm | | Move contents of indirect address to data address |
| MOV *data addr*,A<br>(*data address*) ← (A) | 1 | 1 1 1 1 0 1 0 1<br>mmmmmmmm | | Move A to data address |

† The high order byte of the 16-bit operand is in the first byte following the opcode. The low order byte is in the second byte following the opcode.

## Table B-1. Instruction Set Summary (Cont'd.)

| Mnemonic Operation | Cycles | Binary Code | Flags P OV AC C | Function |
|---|---|---|---|---|
| MOV *data addr*,R*r*<br>(*data address*) ← (R*r*) | 2 | 1 0 0 0 1 r r r<br>mmmmmmmm | | Move register to data address |
| MOV *data addr1*,*data addr2*<br>(*data address1*) ←<br>(*data address2*) | 2 | 1 0 0 0 0 1 0 1<br>mmmmmmmm*<br>mmmmmmmm* | | Move contents of second data address to first data address |
| MOVC A,@A+DPTR<br>(A) ← ((A) + (DPTR)) | 2 | 1 0 0 1 0 0 1 1 | P | Add A to DPTR and move contents of that code address with A |
| MOVC A,@A+PC<br>(A) ← ((A) + (PC)) | 2 | 1 0 0 0 0 0 1 1 | P | Add A to PC and move contents of that code address with A |
| MOVX @DPTR,A<br>((DPTR)) ← (A) | 2 | 1 1 1 1 0 0 0 0 | | Move A to external data location addressed by DPTR |
| MOVX @R*r*,A<br>((R*r*)) ← (A) | 2 | 1 1 1 1 0 0 1 r | | Move A to external data location addressed by register |
| MOVX A,@DPTR<br>(A) ← ((DPTR)) | 2 | 1 1 1 0 0 0 0 0 | P | Move contents of external data location addressed by DPTR to A |
| MOVX A,@R*r*<br>(A) ← ((R*r*)) | 2 | 1 1 1 0 0 0 1 r | P | Move contents of external data location addressed by register to A |
| MUL AB<br>(AB) ← (A) * (B) | 4 | 1 0 1 0 0 1 0 0 | P OV C | Multiply A by B (multiplication register) |
| NOP | 1 | 0 0 0 0 0 0 0 0 | | Do nothing |
| ORL A,#*data*<br>(A) ← (A) OR *data* | 1 | 0 1 0 0 0 1 0 0<br>d d d d d d d d | P | Logical OR immediate data to A |
| ORL A,@R*r*<br>(A) ← (A) OR ((R*r*)) | 1 | 0 1 0 0 0 1 1 r | P | Logical OR contents of indirect address to A |
| ORL A,R*r*<br>(A) ← (A) OR (R*r*) | 1 | 0 1 0 0 1 r r r | P | Logical OR register to A |
| ORL A,*data addr*<br>(A) ← (A) OR (*data address*) | 1 | 0 1 0 0 0 1 0 1<br>mmmmmmmm | P | Logical OR contents of data address to A |
| ORL C,*bit addr*<br>(C) ← (C) OR (*bit address*) | 2 | 0 1 1 1 0 0 1 0<br>b b b b b b b b | C | Logical OR bit to C |
| ORL C,/*bit addr*<br>(C) ← (C) OR NOT (*bit address*) | 2 | 1 0 1 0 0 0 0 0<br>b b b b b b b b | C | Logical OR complement of bit to C |
| ORL *data addr*,#*data*<br>(*data address*) ←<br>(*data address*) OR *data* | 2 | 0 1 0 0 0 0 1 1<br>mmmmmmmm<br>d d d d d d d d | | Logical OR immediate data to data address |
| ORL *data addr*,A<br>(*data address*) ←<br>(*data address*) OR A | 1 | 0 1 0 0 0 0 1 0<br>mmmmmmmm | | Logical OR A to data address |

* The source data address (second data address) is encoded in the first byte following the opcode. The destination data address is encoded in the second byte following the opcode.

## Table B-1. Instruction Set Summary (Cont'd.)

| Mnemonic<br>Operation | Cycles | Binary<br>Code | Flags<br>P OV AC C | Function |
|---|---|---|---|---|
| POP *data addr*<br>(*data address*) ← ((SP))<br>(SP) ← (SP) – 1 | 2 | 1 1 0 1 0 0 0 0<br>mmmmmmmm | | Place top of stack at data address and decrement SP |
| PUSH *data addr*<br>(SP) ← (SP) + 1<br>((SP)) ← (*data address*) | 2 | 1 1 0 0 0 0 0 0<br>mmmmmmmm | | Increment SP and place contents of data address at top of stack |
| RET<br>(PC) *high* ← ((SP))<br>(SP) ← (SP) – 1<br>(PC) *low* ← ((SP))<br>(SP) ← (SP) – 1 | 2 | 0 0 1 0 0 0 1 0 | | Return from subroutine call |
| RETI<br>(PC) *high* ← ((SP))<br>(SP) ← (SP) – 1<br>(PC) *low* ← ((SP))<br>(SP) ← (SP) – 1 | 2 | 0 0 1 1 0 0 1 0 | | Return from interrupt routine |
| RL A<br>(See description in Chapter 3) | 1 | 0 0 1 0 0 0 1 1 | | Rotate A left one position |
| RLC A<br>(See description in Chapter 3) | 1 | 0 0 1 1 0 0 1 1 | P C | Rotate A through C left one position |
| RR A<br>(See description in Chapter 3) | 1 | 0 0 0 0 0 0 1 1 | | Rotate A right one position |
| RRC A<br>(See description in Chapter 3) | 1 | 0 0 0 1 0 0 1 1 | P C | Rotate A through C right one position |
| SETB C<br>(C) ← 1 | 1 | 1 1 0 1 0 0 1 1 | C | Set C to one (1) |
| SETB *bit addr*<br>(*bit address*) ← 1 | 1 | 1 1 0 1 0 0 1 0<br>b b b b b b b b | | Set bit to one (1) |
| SJMP *code addr*<br>(PC) ← (PC) + *relative offset* | 2 | 1 0 0 0 0 0 0 0<br>o o o o o o o o | | Jump to code address |
| SUBB A,#*data*<br>(A) ← (A) – (C) – *data* | 1 | 1 0 0 1 0 1 0 0<br>d d d d d d d d | P OV AC C | Subtract immediate data from A |
| SUBB A,@R*r*<br>(A) ← (A) – (C) – ((R*r*)) | 1 | 1 0 0 1 0 1 1 r | P OV AC C | Subtract contents of indirect address from A |
| SUBB A,R*r*<br>(A) ← (A) – (C) – (R*r*) | 1 | 1 0 0 1 1 r r r | P OV AC C | Subtract register from A |
| SUBB A,*data addr*<br>(A) ← (A) – (C) – (*data address*) | 1 | 1 0 0 1 0 1 0 1<br>mmmmmmmm | P OV AC C | Subtract contents of data address from A |
| SWAP A<br>(See description in Chapter 3) | 1 | 1 1 0 0 0 1 0 0 | | Exchange low order nibble with high order nibble in A |
| XCH A,@R*r*<br>*temp* ← ((R*r*))<br>((R*r*)) ← (A)<br>(A) ← *temp* | 1 | 1 1 0 0 0 1 1 r | P | Move A to indirect address and vice versa |
| XCH A,R*r*<br>*temp* ← (R*r*)<br>(R*r*) ← (A)<br>(A) ← *temp* | 1 | 1 1 0 0 1 r r r | P | Move A to register and vice versa |
| XCH A,*data addr*<br>*temp* ← (*data address*)<br>(*data address*) ← (A)<br>(A) ← *temp* | 1 | 1 1 0 0 0 1 0 1<br>mmmmmmmm | P | Move A to data address and vice versa |

## Table B-1. Instruction Set Summary (Cont'd.)

| Mnemonic Operation | Cycles | Binary Code | Flags P OV AC C | Function |
|---|---|---|---|---|
| XCHD A,@R*r*<br>*temp* ← ((R*r*)) 0-3<br>((R*r*)) 0-3 ← (A) 0-3<br>(A) 0-3 ← *temp* | 1 | 1 1 0 1 0 1 1 r | P | Move low order of A to low order nibble of indirect address and vice versa |
| XRL A,*#data*<br>(A) ← (A) XOR *data* | 1 | 0 1 1 0 0 1 0 0<br>d d d d d d d d | P | Logical exclusive OR immediate data to A |
| XRL A,@R*r*<br>(A) ← (A) XOR ((R*r*)) | 1 | 0 1 1 0 0 1 1 r | P | Logical exclusive OR contents of indirect address to A |
| XRL A,R*r*<br>(A) ← (A) XOR (R*r*) | 1 | 0 1 1 0 1 r r r | P | Logical exclusive OR register to A |
| XRL A,*data addr*<br>(A) ← (A) XOR (*data address*) | 1 | 0 1 1 0 0 1 0 1<br>mmmmmmmm | P | Logical exclusive OR contents of data address to A |
| XRL *data addr*,*#data*<br>(*data address*) ←<br>(*data address*) XOR *data* | 2 | 0 1 1 0 0 0 1 1<br>mmmmmmmm<br>d d d d d d d d | | Logical exclusive OR immediate data to data address |
| XRL *data addr*,A<br>(*data address*) ←<br>(*data address*) XOR A | 1 | 0 1 1 0 0 0 1 0<br>mmmmmmmm | | Logical exclusive OR A to data address |

Table B-2. Instruction Opcodes in Hexadecimal

| Hex Code | Number of Bytes | Mnemonic | Operands |
|---|---|---|---|
| 00 | 1 | NOP | |
| 01 | 2 | AJMP | *code addr* |
| 02 | 3 | LJMP | *code addr* |
| 03 | 1 | RR | A |
| 04 | 1 | INC | A |
| 05 | 2 | INC | *data addr* |
| 06 | 1 | INC | @R0 |
| 07 | 1 | INC | @R1 |
| 08 | 1 | INC | R0 |
| 09 | 1 | INC | R1 |
| 0A | 1 | INC | R2 |
| 0B | 1 | INC | R3 |
| 0C | 1 | INC | R4 |
| 0D | 1 | INC | R5 |
| 0E | 1 | INC | R6 |
| 0F | 1 | INC | R7 |
| 10 | 3 | JBC | *bit addr,code addr* |
| 11 | 2 | ACALL | *code addr* |
| 12 | 3 | LCALL | *code addr* |
| 13 | 1 | RRC | A |
| 14 | 1 | DEC | A |
| 15 | 2 | DEC | *data addr* |
| 16 | 1 | DEC | @R0 |
| 17 | 1 | DEC | @R1 |
| 18 | 1 | DEC | R0 |
| 19 | 1 | DEC | R1 |
| 1A | 1 | DEC | R2 |
| 1B | 1 | DEC | R3 |
| 1C | 1 | DEC | R4 |
| 1D | 1 | DEC | R5 |
| 1E | 1 | DEC | R6 |
| 1F | 1 | DEC | R7 |
| 20 | 3 | JB | *bit addr,code addr* |
| 21 | 2 | AJMP | *code addr* |
| 22 | 1 | RET | |
| 23 | 1 | RL | A |
| 24 | 2 | ADD | A,*#data* |
| 25 | 2 | ADD | A,*data addr* |
| 26 | 1 | ADD | A,@R0 |
| 27 | 1 | ADD | A,@R1 |
| 28 | 1 | ADD | A,R0 |
| 29 | 1 | ADD | A,R1 |
| 2A | 1 | ADD | A,R2 |
| 2B | 1 | ADD | A,R3 |
| 2C | 1 | ADD | A,R4 |
| 2D | 1 | ADD | A,R5 |
| 2E | 1 | ADD | A,R6 |
| 2F | 1 | ADD | A,R7 |
| 30 | 3 | JNB | *bit addr,code addr* |
| 31 | 2 | ACALL | *code addr* |
| 32 | 1 | RETI | |
| 33 | 1 | RLC | A |
| 34 | 2 | ADDC | A,*#data* |
| 35 | 2 | ADDC | A,*data addr* |
| 36 | 1 | ADDC | A,@R0 |
| 37 | 1 | ADDC | A,@R1 |
| 38 | 1 | ADDC | A,R0 |
| 39 | 1 | ADDC | A,R1 |
| 3A | 1 | ADDC | A,R2 |
| 3B | 1 | ADDC | A,R3 |

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

| Hex Code | Number of Bytes | Mnemonic | Operands |
|---|---|---|---|
| 3C | 1 | ADDC | A,R4 |
| 3D | 1 | ADDC | A,R5 |
| 3E | 1 | ADDC | A,R7 |
| 3F | 1 | ADDC | A,R7 |
| 40 | 2 | JC | *code addr* |
| 41 | 2 | AJMP | *code addr* |
| 42 | 2 | ORL | *data addr*,A |
| 43 | 3 | ORL | *data addr*,#*data* |
| 44 | 2 | ORL | A,#*data* |
| 45 | 2 | ORL | A,*data addr* |
| 46 | 1 | ORL | A,@R0 |
| 47 | 1 | ORL | A,@R1 |
| 48 | 1 | ORL | A,R0 |
| 49 | 1 | ORL | A,R1 |
| 4A | 1 | ORL | A,R2 |
| 4B | 1 | ORL | A,R3 |
| 4C | 1 | ORL | A,R4 |
| 4D | 1 | ORL | A,R5 |
| 4E | 1 | ORL | A,R6 |
| 4F | 1 | ORL | A,R7 |
| 50 | 2 | JNC | *code addr* |
| 51 | 2 | ACALL | *code addr* |
| 52 | 2 | ANL | *data addr*,A |
| 53 | 3 | ANL | *data addr*,#*data* |
| 54 | 2 | ANL | A,#*data* |
| 55 | 2 | ANL | A,*data addr* |
| 56 | 1 | ANL | A,@R0 |
| 57 | 1 | ANL | A,@R1 |
| 58 | 1 | ANL | A,R0 |
| 59 | 1 | ANL | A,R1 |
| 5A | 1 | ANL | A,R2 |
| 5B | 1 | ANL | A,R3 |
| 5C | 1 | ANL | A,R4 |
| 5D | 1 | ANL | A,R5 |
| 5E | 1 | ANL | A,R6 |
| 5F | 1 | ANL | A,R7 |
| 60 | 2 | JZ | *code addr* |
| 61 | 2 | AJMP | *code addr* |
| 62 | 2 | XRL | *data addr*,A |
| 63 | 3 | XRL | *data addr*,#*data* |
| 64 | 2 | XRL | A,#*data* |
| 65 | 2 | XRL | A,*data addr* |
| 66 | 1 | XRL | A,@R0 |
| 67 | 1 | XRL | A,@R1 |
| 68 | 1 | XRL | A,R0 |
| 69 | 1 | XRL | A,R1 |
| 6A | 1 | XRL | A,R2 |
| 6B | 1 | XRL | A,R3 |
| 6C | 1 | XRL | A,R4 |
| 6D | 1 | XRL | A,R5 |
| 6E | 1 | XRL | A,R6 |
| 6F | 1 | XRL | A,R7 |
| 70 | 2 | JNZ | *code addr* |
| 71 | 2 | ACALL | *code addr* |
| 72 | 2 | ORL | C,*bit addr* |
| 73 | 1 | JMP | @A+DPTR |
| 74 | 2 | MOV | A,#*data* |
| 75 | 3 | MOV | *data addr*,#*data* |
| 76 | 2 | MOV | @R0,#*data* |
| 77 | 2 | MOV | @R1,#*data* |

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

| Hex Code | Number of Bytes | Mnemonic | Operands |
|---|---|---|---|
| 78 | 2 | MOV | R0,*#data* |
| 79 | 2 | MOV | R1,*#data* |
| 7A | 2 | MOV | R2,*#data* |
| 7B | 2 | MOV | R3,*#data* |
| 7C | 2 | MOV | R4,*#data* |
| 7D | 2 | MOV | R5,*#data* |
| 7E | 2 | MOV | R6,*#data* |
| 7F | 2 | MOV | R7,*#data* |
| 80 | 2 | SJMP | *code addr* |
| 81 | 2 | AJMP | *code addr* |
| 82 | 2 | ANL | C,*bit addr* |
| 83 | 1 | MOVC | A,@A + PC |
| 84 | 1 | DIV | AB |
| 85 | 3 | MOV | *data addr*,*data addr* |
| 86 | 2 | MOV | *data addr*,@R0 |
| 87 | 2 | MOV | *data addr*,@R1 |
| 88 | 2 | MOV | *data addr*,R0 |
| 89 | 2 | MOV | *data addr*,R1 |
| 8A | 2 | MOV | *data addr*,R2 |
| 8B | 2 | MOV | *data addr*,R3 |
| 8C | 2 | MOV | *data addr*,R4 |
| 8D | 2 | MOV | *data addr*,R5 |
| 8E | 2 | MOV | *data addr*,R6 |
| 8F | 2 | MOV | *data addr*,R7 |
| 90 | 3 | MOV | DPTR,*#data* |
| 91 | 2 | ACALL | *code addr* |
| 92 | 2 | MOV | *bit addr*,C |
| 93 | 1 | MOVC | A,@A + DPTR |
| 94 | 2 | SUBB | A,*#data* |
| 95 | 2 | SUBB | A,*data addr* |
| 96 | 1 | SUBB | A,@R0 |
| 97 | 1 | SUBB | A,@R1 |
| 98 | 1 | SUBB | A,R0 |
| 99 | 1 | SUBB | A,R1 |
| 9A | 1 | SUBB | A,R2 |
| 9B | 1 | SUBB | A,R3 |
| 9C | 1 | SUBB | A,R4 |
| 9D | 1 | SUBB | A,R5 |
| 9E | 1 | SUBB | A,R6 |
| 9F | 1 | SUBB | A,R7 |
| A0 | 2 | ORL | C,/*bit addr* |
| A1 | 2 | AJMP | *code addr* |
| A2 | 2 | MOV | C,*bit addr* |
| A3 | 1 | INC | DPTR |
| A4 | 1 | MUL | AB |
| A5 | | reserved | |
| A6 | 2 | MOV | @R0,*data addr* |
| A7 | 2 | MOV | @R1,*data addr* |
| A8 | 2 | MOV | R0,*data addr* |
| A9 | 2 | MOV | R1,*data addr* |
| AA | 2 | MOV | R2,*data addr* |
| AB | 2 | MOV | R3,*data addr* |
| AC | 2 | MOV | R4,*data addr* |
| AD | 2 | MOV | R5,*data addr* |
| AE | 2 | MOV | R6,*data addr* |
| AF | 2 | MOV | R7,*data addr* |
| B0 | 2 | ANL | C,/*bit addr* |
| B1 | 2 | ACALL | *code addr* |
| B2 | 2 | CPL | *bit addr* |
| B3 | 1 | CPL | C |

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

| Hex Code | Number of Bytes | Mnemonic | Operands |
|---|---|---|---|
| B4 | 3 | CJNE | A,#*data*,*code addr* |
| B5 | 3 | CJNE | A,*data addr*,*code addr* |
| B6 | 3 | CJNE | @R0,#*data*,*code addr* |
| B7 | 3 | CJNE | @R1,#*data*,*code addr* |
| B8 | 3 | CJNE | R0,#*data*,*code addr* |
| B9 | 3 | CJNE | R1,#*data*,*code addr* |
| BA | 3 | CJNE | R2,#*data*,*code addr* |
| BB | 3 | CJNE | R3,#*data*,*code addr* |
| BC | 3 | CJNE | R4,#*data*,*code addr* |
| BD | 3 | CJNE | R5,#*data*,*code addr* |
| BE | 3 | CJNE | R6,#*data*,*code addr* |
| BF | 3 | CJNE | R7,#*data*,*code addr* |
| C0 | 2 | PUSH | *data addr* |
| C1 | 2 | AJMP | *code addr* |
| C2 | 2 | CLR | *bit addr* |
| C3 | 1 | CLR | C |
| C4 | 1 | SWAP | A |
| C5 | 2 | XCH | A,*data addr* |
| C6 | 1 | XCH | A,@R0 |
| C7 | 1 | XCH | A,@R1 |
| C8 | 1 | XCH | A,R0 |
| C9 | 1 | XCH | A,R1 |
| CA | 1 | XCH | A,R2 |
| CB | 1 | XCH | A,R3 |
| CC | 1 | XCH | A,R4 |
| CD | 1 | XCH | A,R5 |
| CE | 1 | XCH | A,R6 |
| CF | 1 | XCH | A,R7 |
| D0 | 2 | POP | *data addr* |
| D1 | 2 | ACALL | *code addr* |
| D2 | 2 | SETB | *bit addr* |
| D3 | 1 | SETB | C |
| D4 | 1 | DA | A |
| D5 | 3 | DJNZ | *data addr*,*code addr* |
| D6 | 1 | XCHD | A,@R0 |
| D7 | 1 | XCHD | A,@R1 |
| D8 | 2 | DJNZ | R0,*code addr* |
| D9 | 2 | DJNZ | R1,*code addr* |
| DA | 2 | DJNZ | R2,*code addr* |
| DB | 2 | DJNZ | R3,*code addr* |
| DC | 2 | DJNZ | R4,*code addr* |
| DD | 2 | DJNZ | R5,*code addr* |
| DE | 2 | DJNZ | R6,*code addr* |
| DF | 2 | DJNZ | R7,*code addr* |
| E0 | 1 | MOVX | A,@DPTR |
| E1 | 2 | AJMP | *code addr* |
| E2 | 1 | MOVX | A,@R0 |
| E3 | 1 | MOVX | A,@R1 |
| E4 | 1 | CLR | A |
| E5 | 2 | MOV | A,*data addr* |
| E6 | 1 | MOV | A,@R0 |
| E7 | 1 | MOV | A,@R1 |
| E8 | 1 | MOV | A,R0 |
| E9 | 1 | MOV | A,R1 |
| EA | 1 | MOV | A,R2 |
| EB | 1 | MOV | A,R3 |
| EC | 1 | MOV | A,R4 |
| ED | 1 | MOV | A,R5 |
| EE | 1 | MOV | A,R6 |
| EF | 1 | MOV | A,R7 |

Table B-2. Instruction Opcodes in Hexadecimal (Cont'd.)

| Hex Code | Number of Bytes | Mnemonic | Operands |
|---|---|---|---|
| F0 | 1 | MOVX | @DPTR,A |
| F1 | 2 | ACALL | *code addr* |
| F2 | 1 | MOVX | @R0,A |
| F3 | 1 | MOVX | @R1,A |
| F4 | 1 | CPL | A |
| F5 | 2 | MOV | *data addr*,A |
| F6 | 1 | MOV | @R0,A |
| F7 | 1 | MOV | @R1,A |
| F8 | 1 | MOV | R0,A |
| F9 | 1 | MOV | R1,A |
| FA | 1 | MOV | R2,A |
| FB | 1 | MOV | R3,A |
| FC | 1 | MOV | R4,A |
| FD | 1 | MOV | R5,A |
| FE | 1 | MOV | R6,A |
| FF | 1 | MOV | R7,A |

# APPENDIX C
# ASSEMBLER DIRECTIVE SUMMARY

The following is a list of al MCS-51 Macro Assembly Language directives. The format for each directive is shown along with a brief description of its operation. Chapter 4 contains a complete description of all directives.

**Assembler Directives**

| | |
|---|---|
| *symbol* BIT *bit address* | Define a bit address symbol |
| BSEG | Select bit address segment |
| CSEG | Select Code segment |
| *symbol* DATA *expression* | Define a data address symbol |
| [*label:*] DB *expression list* | Insert a list of byte values |
| [*label:*] DBIT *expression* | Advance bit location counter |
| [*label:*] DS *expression* | Advance active location counter |
| DSEG | Select internal Data Segment |
| [*label:*] DW *expression list* | Insert a list of word values |
| END | End of program |
| *symbol* EQU *expression or register* | Set symbol value permanently |
| ORG *expression* | Set location counter value |
| *symbol* SET *expression or register* | Set symbol value temporarily |
| *symbol* XDATA *expression* | Define an off chip data address symbol |
| XSEG | Select external Data Segment |

# APPENDIX D
# ASSEMBLER CONTROL SUMMARY

The table below contains all of the MCS-51 Macro assembler controls, their meaning, their defaults and their abbreviations. The table also defines whether the control is primary or general. (Primary controls must only appear at the head of the program or in the invocation lines; general controls may appear anywhere in the program.)

Table D-1. Assembler Controls

| Name | Primary/ General | Default | Abbrev. | Meaning |
|---|---|---|---|---|
| DATE(date) | P | DATE( ) | DA | Places string in header (max 9 characters) |
| DEBUG | P | NODEBUG | DB | Outputs debug symbol information to object file |
| NODEBUG | P | | NODB | Symbol information not placed in object file |
| EJECT | G | *Not Applicable* | EJ | Continue listing on next page |
| ERRORPRINT[(FILE)] | P | NOERRORPRINT | EP | Designates a file to receive error messages in addition to the listing file |
| NOERRORPRINT | P | | NOEP | Designates that error messages will be printed in listing file |
| GEN | G | GENONLY | GE | Generates a full listing of the macro expansion process including macro calls in the listing file |
| GENONLY | G | | GO | List only the fully expanded source as if all lines generated by a macro call were already in source file |
| NOGEN | G | | NOGE | List only the original source text in listing file |
| INCLUDE(FILE) | G | *Not Applicable* | IC | Designates a file to be included as part of the program |
| LIST | G | LIST | LI | Print subsequent lines of source in listing file |
| NOLIST | G | | NOLI | Do not print subsequent lines of source in listing file |
| MACRO | P | MACRO | MR | Evaluate and expand all macro calls |
| NOMACRO | P | | NOMR | Do not evaluate macro calls |
| OBJECT[(FILE)] | P | OBJECT(*source*.HEX) | OJ | Designate file to receive object code |
| NOOBJECT | P | | NOOJ | Designates that no object file will be created |
| PAGING | P | PAGING | PI | Designates that listing will be broken into pages and each will have a header |
| NOPAGING | P | | NOPI | Designates that listing will contain no page breaks |
| PAGELENGTH(n) | P | PAGELENGTH(60) | PL | Sets maximum number of lines in each page of listing file (maximum = 65,535) (minimum = 10) |
| PAGEWIDTH(n) | P | PAGEWIDTH(120) | PW | Sets maximum number of characters in each line of listing file (maximum = 132; minimum = 72) |
| PRINT[(FILE)] | P | PRINT(*source*. LST) | PR | Designates file to receive source listing |
| NOPRINT | P | | NOPR | Designates that no listing file will be created |

Table D-1. Assembler Controls (Cont'd.)

| Name | Primary/ General | Default | Abbrev. | Meaning |
|---|---|---|---|---|
| SAVE | G | Not Applicable | SA | Stores current control setting for LIST and GEN |
| RESTORE | G | | RS | Restores control setting from SAVE stack |
| SYMBOLS | P | NOSYMBOLS | SB | Creates a formatted table of all symbols used in program |
| NOSYMBOLS | P | | NOSB | No symbol table created |
| TITLE(string) | G | TITLE( ) | TT | Places a string in all subsequent page headers (maximum 60 characters) |
| WORKFILES(:Fn:[,:F m:]) | P | *same drive as source file* | WF | Designates alternate drives for temporary workfiles |
| XREF | P | NOXREF | XR | Creates a cross reference listing of all symbols used in program |
| NOXREF | P | | NOXR | No cross reference list created |

# APPENDIX E
# MACRO PROCESSOR LANGUAGE

## Introduction

This appendix is intended as a reference document for the macro language and as a guide to more advanced use of the macro processor. It is assumed that the reader is already familiar with the material on macros presented in Chapter 5, and he can use the macro processing language.

### Terminology and Conventions

A percent sign will be used as the Metacharacter throughout this appendix although the user may temporarily change the metacharacter by using the METACHAR function.

The term "logical blank" refers to a blank, horizontal tab, carriage return, or linefeed character.

Throughout the appendix the term "parameter" refers to what are sometimes known as "dummy parameters" or "formal parameters" while the term "argument" is reserved for what are sometimes known as "actual parameters". The terms "Normal" and "Literal", names for the two fundamental modes used by the macro processor in reading characters, will be capitalized in order to distinguish these words from their ordinary usage.

In the syntax diagrams, non-terminal syntactic types are represented by lower case words, sometimes containing the break character, "_". If a single production contains more than one instance of a syntactic type each instance may be followed by a unique integer so that the prose description may unambiguously refer to each occurrence.

## Basic Elements of the Macro Language

### Identifiers

With the exception of some built-in functions, all macro processor functions begin with an identifier, which names the function. Parameters also are represented by identifiers. A macro processor identifier has the following syntax.

*id = alphabetic | id id_continuation*

The *alphabetic* characters include upper and lower case letters, the break character ("_"), and the question mark ("?"). An *id_continuation* character is an *alphabetic* character or a decimal digit.

Examples:

```
WHILE  Add_2  MORE_TO_DO  Much_More
```

An identifier must not be split across the boundary of a macro and may not contain Literal characters.

For example,

```
%%(FOO)
```

is illegal, the first metacharacter is followed by the letters "FOO", but they do not constitute an identifier since they are Literal characters.

```
%ADD%SUFFIX
```

where SUFFIX is defined as "UP" is a call to ADD followed by a call to SUFFIX, rather than a call to ADDUP, because identifiers may not cross macro boundaries.

A null-string bracket or escape function ("%( )" or "%0") will also end an identifier, and since these functions have no textual value themselves, may be used as separators.

Example:

```
%TOM%0SMITH
```

concatenates the value of the macro, TOM, to the string, "SMITH".

This could also be done by writing, "%TOM%(SMITH)". Upper and lower case letters are equivalent in their use in identifiers. ("CAT", "cat", and "cAt" are equivalent.)

## Text and Delimiters

"Text" is an undistinguished string of characters. It may or may not contain items of significance to the macro processor. In general the MPL processor simply copies characters from its input to its output stream. This copying process continues until an instance of the metacharacter is encountered, whereupon the macro processor begins analyzing the text that follows.

Each macro function has a calling pattern that must match the text in an actual macro function call. The pattern consists of text strings, which are the arguments to the function, and a number of delimiter strings.

For example,

```
JOIN (FIRST, SECOND)
```

might be a pattern for a macro, JOIN, which takes two arguments. The first argument will correspond to the parameter, FIRST, and the second to the parameter, SECOND. The delimiters of this pattern are "(", ",", and ")".

A text string corresponding to a parameter in the pattern must be balanced with respect to parentheses (see below). A delimiter which follows a parameter in the pattern will be used to mark the end of the argument in an actual call to the macro.

An argument text string is recognized by finding the specific delimiter that the pattern indicates will end the string. A text string for a given argument consists of the characters between the delimiter (or macro identifier) that precedes the text and the delimiter which follows the text.

In the case of built-in functions, there are sometimes additional requirements on the syntax of an argument. For example, the text of an argument might be required to conform to the syntax for a numeric expression.

## Balanced Text

Arguments must be balanced with respect to left and right parentheses in the usual manner of requiring that the string contain the same number of left and right parentheses and that at no point during a left to right scan may there have been more right parentheses than left parentheses. (An "unbalanced" parenthesis may be quoted with the escape function to make the text meet this requirement.)

## Expressions

Balanced text strings appearing in certain places in built-in macro processor functions are interpreted as numeric expressions:

1. As arguments that control the execution of "IF", "WHILE", "REPEAT", and "SUBSTR" functions.
2. As the argument to the evaluate function, "EVAL".

Operators (in order of precedence from high to low):

**Parenthesized Expressions**

```
HIGH    LOW
*  /  MOD  SHL  SHR
+  -
EQ  LT  LE  GT  GE  NE
NOT
AND
OR  XOR
```

All arithmetic is performed in an internal format of 16-bit two's complement integers.

# The Macro Processor Scanning Algorithm

## Literal or Normal Mode of Expansion

At any given time the macro processor is reading text in one of two fundamental modes. When processing of the primary input file begins, the mode is Normal. Normal mode means that macro calls will be expanded, i.e., the metacharacter in the input will cause the following macro function to be executed.

In the simplest possible terms, Literal mode means that characters are read literally, i.e., the text is not examined for function calls. The text read in this mode is similar to the text inside a quoted character string familiar to most users of high level languages; that is, the text is considered to be merely a sequence of characters having no semantic weight. There are important exceptions to this very simple view of the Literal mode. If the characters are being read from a user defined macro with parameters, the parameter references will be replaced with the corresponding argument values regardless of the mode. The Escape function and the Comment function will also be recognized in either mode.

The mode can change when a macro is called. For user-defined macros, the presence or absence of the call-literally character following the metacharacter sets the mode for the reading of the macro's value. The arguments to a user defined macro are evaluated in the Normal mode but when the processor begins reading the macro's value, the mode changes to that indicated by the call. When the processor finishes reading the macro's definition, the mode reverts to what it was before the macro's processing began.

To illustrate, suppose the parameterless macros, CAT and TOM are defined as follows.

CAT is:

```
abcd %TOM efgh
```

and TOM is:

```
xyz
```

Now consider the text fragment,

```
... %CAT, %*CAT ...
```

Assume the string is being read in the Normal mode. The first call to CAT is recognized and called Normally. Since CAT is called Normally, the definition of CAT is examined for macro calls as it is read. Thus the characters "%TOM" in the definition for CAT are recognized as a macro call and so TOM is expanded Normally. The definition for TOM is read, but it contains no macro calls. After the definition for TOM is processed, the mode reverts back to its value in reading CAT (Normal). After the definition of CAT is processed the mode reverts back to its original value (Normal). At this point, immediately before processing the comma following the first call to CAT, the value of the text fragment processed thus far is:

```
... abcd xyz efgh
```

Now the processor continues reading Normally, finally encountering the second call to CAT, this time a Literal call. The mode changes to Literal as the definition of CAT is read. This time the characters from the definition are read Literally. When the end of the definition of CAT is reached the mode reverts to its original value (Normal) and processing continues. The value of the entire fragment is,

```
... abcd xyz efgh, abcd %TOM efgh ...
```

The use of the call-literally character on calls to built-in macro functions is discussed in the description of each function. The important thing to keep in mind when analyzing how a piece of text is going to be expanded is the Normal or Literal Mode of the environment in which it is read.

## The Call Pattern

In general, each macro function has a distinctive name which follows the metacharacter (and possibly the call-literally character). This name is usually an identifier, although a few built-in functions have other symbols for names. For identifier named functions, the macro processor allows the identifier to be the result of another macro call.

For example, suppose the macro, NAME, has the value "HAMBURGER" and that the macro HAMBURGER has the calling pattern, "HAMBURGER X & Y;". Then the call,

```
... %%NAME catsup & mustard; ...
```

is a call to the macro HAMBURGER with the first argument having the value, "catsup" and the second argument having the value, "mustard".

Associated with this name is, possibly, a pattern of delimiters and parameters which must be matched if the macro call is to be syntactically correct. The pattern for each built-in macro function is described in the section of this appendix dealing with that function. The pattern for a user-defined macro is defined at the time the macro is defined.

At the time of a macro call, the matching of text to the pattern occurs by using the delimiters one at a time, left to right. When a delimiter is located, the next delimiter of the pattern becomes the new goal. The delimiters in the call are separated by either argument text (if there was a corresponding parameter in the macro's definition pattern), or by any number of logical blanks (in the case of adjacent delimiters in the pattern). The argument text corresponding to a parameter in the definition pattern becomes the value of the parameter for the duration of the macro's expansion. Null arguments are permitted.

See the section "Macro Definition and Invocation" for more information on delimiters and their relationship to argument strings.

### Evaluation of Arguments—Parameter Substitution

MPL uses "call-by-immediate-value" as the ordinary scheme for argument evaluation. This means that as the text is being scanned for the delimiter which marks the end of an argument, any macro calls will be evaluated as they are encountered. In order to be considered as a possible delimiter, characters must all be on the same level of macro nesting as the metacharacter which began the call. In other words, the arguments to a macro can be any mixture of plaintext and macro calls, but the delimiters of a call must be plaintext.

For example, suppose STRG is defined as "dogs,cats" and MAC1 is a macro with the calling pattern, "MAC1 (P1, P2)". Then in the call,

```
... %MAC1 (%STRG, mouse) ...
```

the first argument will be "dogs,cats" and the second argument will be "mouse". The comma in the middle of the first argument is not taken as the delimiter because it is on a different level than the metacharacter which began the call to MAC1.

When all arguments of a macro have been evaluated, the expansion of the body begins, with characters being read either Normally or Literally as discussed under "Literal or Normal Mode of Expansion". One should keep in mind that parameter substitution is a high priority function, i.e., arguments will be substituted for parameters even if the macro has been called Literally.

## The Evaluate Function

The syntax for the Evaluate function is:

*evaluate_function* = EVAL ( *expr* )

The single argument is a text string which will be passed to the engineer-supplied evaluator procedure for evaluation as an expression. The character string returned by the evaluator procedure will be the value of the "EVAL" function.

Examples:

```
%EVAL(7)
```

evaluates to "07H"

```
%EVAL( (7+3)*2 )
```

evaluates to "14H"

If NUM has the value "0101B" then

```
%EVAL( %NUM - 5)
```

evaluates to "00H".

## Numeric Functions: LEN, and String Compare Functions

These functions take text string arguments and return some numeric information in the form of hexadecimal integers.

*length_function* = *LEN* ( *balanced_text* )

*string_compare_function* = *op_code* ( *balanced_text*,*balanced_text* )

*op_code* = EQS | GTS | LTS | NES | GES | LES

The length numeric function returns an integer equal to the number of characters in the text string. The string comparison functions all return the character representation for minus one if the relation between the strings holds, or zero otherwise. These relations are for string compares. These functions should not be confused with the arithmetic compare operators that might appear in expressions. The ASCII code for each character is considered a binary number and represents the relative value of the character. "Dictionary" ordering is used: Strings differing first in their Nth character are ranked according to the Nth character. A string which is a prefix of another string is ranked lower than the longer string.

## The Bracket Function

The bracket function is used to introduce literal strings into the text and to prevent the interpretation of functions contained therein. (Except the high priority functions: comment, escape, and parameter substitution.) A call-literally character is not allowed; the function is always called Literally.

*bracket_function* = ( *balanced_text* )

The value of the function is the value of the text between the matching parentheses, evaluated Literally. The text must be balanced with respect to left and right parentheses. (An unbalanced left or right parenthesis may be quoted with the escape function.) Text inside the bracket function that would ordinarily be recognized as a function call is not recognized; thus, when an argument in a macro call is put inside a bracket function, the evaluation of the argument is delayed—it will be substituted as it appears in the call (but without the enclosing bracket function).

The null string may be represented as %( ).

Examples:

```
%(This is a string.)          %( %EVAL( %NUM ) )
evaluates to:                 evaluates to:
This is a string.             %EVAL( %NUM )
```

## The Escape Function

The escape function provides an easy way to quote a few characters to prevent them from having their ordinary interpretation. Typical uses are to insert an "unbalanced" parenthesis into a balanced text string, or to quote the metacharacter. The syntax is:

*escape_function* = /* A single digit, 0 through 9, followed by that many characters. */

The call-literally character may not be present in the call. The escape function is a high priority function, that is one of the functions (the others are the comment functions and parameter substitution) which are recognized in both Normal and Literal mode.

Examples:

```
... %2%% ... evaluates to ... %% ...
... %(ab%1)cd) ... evaluates to ... ab)cd ...
```

## Macro Definition and Invocation

The macro definition function associates an identifier with a functional string. The macro may or may not have an associated pattern consisting of parameters and/or delimiters. Also optionally present are local symbols. The syntax for a macro definition is:

*macro_def_function* = DEFINE ( *macro_id define_pattern* ) [ LOCAL {*id*} ]
( *balanced_text* )

The define_pattern is a balanced string which is further analyzed by the macro processor as follows:

*define_pattern* = { [{*parm_id*}] [{*delimiter_specifier*}] }

*delimiter_specifier* = /* String not containing non-Literal *id_continuation*, logical blank, or "@" characters. */

| *@ delimiter_id*

The syntax for a macro invocation is as follows:

*macro_call* = *macro_id* [ *call_pattern* ]

*call_pattern* = /* Pattern of text and delimiters corresponding to the definition pattern. */

As seen above, the *macro_id* optionally may be defined to have a pattern, which consists of parameters and delimiters. The presence of this define pattern specifies how the arguments in the macro call will be recognized. Three kinds of delimiters may be specified in a define pattern. Literal and Identifier delimiters appear explicitly in the define pattern, while Implied Blank delimiters are implicit where a parameter in the define pattern is not followed by an explicit delimiter. Literal delimiters are the most common and typically include commas, parentheses, other punctuation marks, etc. Id delimiters are delimiters that look like and are recognized like identifiers. The presence of an Implied Blank delimiter means that the preceding argument is terminated by the first logical blank encountered. We will examine these various forms of delimiter in greater detail later in this description.

Recognition of a macro name (which uniquely identifies a macro) is followed by the matching of the call pattern to the define pattern. The two patterns must match for the call to be well formed. It must be remembered that arguments are *balanced* strings, thus parentheses can be used to prevent an enclosed substring from being matched with a delimiter. The strings in the call pattern corresponding to the parameters in the define pattern become the values of those parameters.

Reuse of the name for another definition at a later time will replace a previous definition. Built-in macro processor functions (as opposed to user-defined macros) may not be redefined. A macro may not be redefined during the evaluation of its own body. A parameter may not be redefined within the body of its macro.

Parameters appearing in the body of a macro definition (as parameter substitution functions) are preceded by the metacharacter. When the body is being expanded after a call, the parameter substitution function calls will be replaced by the value of the corresponding arguments.

The evaluation of the *balanced_text* that defines the body of the macro being defined is evaluated in the mode specified by the presence or absence of the call literally character on the call to DEFINE. If the DEFINE function is called Normally, the balanced text is evaluated in the Normal mode before it is stored as the macro's value. If the define function is called Literally, the *balanced_text* is evaluated Literally before it is stored.

## Literal Delimiters

A Literal delimiter which contains *id_continuation* characters, "@", or logical blanks must be quoted by a bracket function, escape function, or by being produced by a Literal call. Other literal delimiters need not be quoted in the define pattern.

Example 1:

```
%*DEFINE ( SAY(ANIMAL,COLOR) ) (THE %ANIMAL IS %COLOR.)
%SAY(HORSE,TAN)
```

produces,

```
THE HORSE IS TAN.
```

Example 2:

```
%*DEFINE ( REVERSE [ P1 %(.AND.) P2 ] ) (%P2 %P1)
%REVERSE [FIRST.AND.SECOND]
```

produces,

```
SECOND FIRST
```

## Id Delimiters

Id delimiters are specified in the define pattern by using a *delimiter_specifier* having the form, "@ id". The following example should make the distinction between literal and identifier delimiters clear. Consider two *delimiter_specifiers*, "%(AND)" and "@AND" (the first a Literal delimiter and the second an Id delimiter), and the text string,

```
... GRAVEL, SAND AND CINDERS ...
```

Using the first delimiter specifier, the first "AND", following the letter "S", would be recognized as the end of the argument. However, using the second delimiter specifier, only the second "AND" would match, because the second delimiter is recognized like an identifier. Another example:

Definition:

```
%*DEFINE ( ADD P1 @TO P2 @STORE P3. )
(    MOV  A,%P1
     ADD  A,%P2
     MOV  %P3,A
)
```

Macro call:

```
%ADD TOTAL1 TO TOTAL2 STORE GRAND.
```

Generates:

```
MOV  A,TOTAL1
ADD  A,TOTAL2
MOV  GRAND,A
```

## Implied Blank Delimiters

If a parameter is not followed by an explicit Literal or Id delimiter then it is terminated by an Implied Blank delimiter. A logical blank is implied as the terminator of the argument corresponding to the preceding parameter. In this case any logical blank in the actual argument must be literalized to prevent its being recognized as the end of the argument. In scanning for an argument having this kind of delimiter, leading non-literal logical blanks will be discarded and the first following non-literal logical blank will terminate the argument.

Example:

```
%*DEFINE ( SAY ANIMAL COLOR ) (THE %ANIMAL IS %COLOR.)
```

The call,

```
%SAY HORSE TAN
```

will evaluate to,

```
THE HORSE IS TAN.
```

In designating delimiters for a macro one should keep in mind the text strings which are likely to appear as arguments. One might base the choice of delimiters for the define pattern on whether the arguments will be numeric, strings of identifiers, or may contain imbeded blanks or punctuation marks.

## LOCAL Macros and Symbols

The LOCAL option, can be used to designate identifiers that will be used within the scope of the macro for local macros. A reference to a LOCAL identifier of a macro occurring after the expansion of the text of the macro has begun and before the expansion of the macro is completed will be a reference to the definition of this local macro. Every time a macro having the LOCAL option is called a new incarnation of the listed symbols are created. The local symbols thus have dynamic, inclusive scope.

At the time of the call to a macro having locals, the local symbols are initialized to a string whose value is the symbol name concatenated with a unique 3 digit number. The number is generated by incrementing a counter each time the macro is called.

Definition:

```
%*DEFINE (MAC 1 (FIRST,SECOND,THIRD)) LOCAL LABL
(%LABL:   MOV   @R1,%FIRST
          MOV   A,@R1
          MOV   @R1,%SECOND
          MOV   R7,@R1
          MOV   @R1,%THIRD
          MOV   R6,@R1
)
```

Macro call:

```
%MAC(ITEM,NEXT,ANOTHER)
```

Generates: (Typically, depending on value for local, "LABL")

```
LABL3:    MOV   @R1,ITEM
          MOV   A,@R1
          MOV   @R1,NEXT
          MOV   R7,@R1
          MOV   @R1,ANOTHER
          MOV   R6,@R1
```

## The Control Functions: IF, REPEAT, and WHILE

These functions can be used to alter the flow of control in a sense analogous to that of their similarly named counterparts in procedural languages; however, they are different in that they may be used as value generating functions as well as control statements.

The three functions all have a "body" which is analogous to the defined value, or body, of a user-defined macro function. The syntax of these functions is:

*if_function* = IF ( *expr* ) THEN ( *body* ) [ ELSE ( *body* ) FI

*repeat_function* = REPEAT ( *expr* ) ( *body* )

*while_function* = WHILE ( *expr* ) ( *body* )

The expressions will evaluate to binary numbers. As in PL/M 80, twos complement representation is used so that negative expressions will map into a large positive number. (E.g., "−1" maps into 0FFFFH.) (See Part III, Engineer Supplied Functions.) The bodies of these functions are balanced_text strings, and although they look exactly like arguments in the syntax diagrams, they are processed very much like the bodies of user-defined macro functions; the bodies are "called" based upon some aspect of the expression in the IF, REPEAT, or WHILE function. The effects for each control function are described below.

## The IF Function

The first argument is evaluated Normally and interpreted as a numeric expression.

If the value of the expression is odd (=TRUE) then the body of the THEN phrase is evaluated and becomes the value of the function. The body of the ELSE clause is not evaluated.

If the value of the expression is even (=FALSE) and the ELSE clause is present, then the body of the ELSE phrase is evaluated and becomes the value of the function. The body of the THEN clause is not evaluated.

Otherwise, the value is the null string.

In the cases in which the body is evaluated, evaluation is Normal or Literal as determined by the presence or absence of the call-literally character on the IF.

Examples:

```
%IF (%VAL GT 0) THEN( %DEFINE(SIGN)(1) ) ELSE( %DEFINE(SIGN)(0) ) FI
```

If the value of the numeric symbol VAL is positive then the SIGN will be defined as "1"; otherwise, it will be defined as "0". In either case the value of the IF function is the null string.

```
%DEFINE(SIGN) (%IF (%VAL GT 0) THEN(1) ELSE(0) FI)
```

This example has exactly the same effect as the previous one.

## The Repeat Function

The REPEAT function causes its body to be expanded a predetermined number of times. The first argument is evaluated Normally and interpreted as a numeric expression. This expression, specifying the number of repetitions, is evaluated only once, before the expansion of the text to be repeated begins. The body is then evaluated the indicated number of times, Normally or Literally, and the resulting string becomes the value of the function. A repetition number of zero yields the null string as the value of the REPEAT function call.

Examples:

Rotate the accumulator of the 8051 right six times:

```
%REPEAT (6)
(     RR A
)
```

Generate a horizontal coordinate line to be used in plotting a curve on a line printer. The line is to be 51 characters long and is to be marked every 5 characters:

```
%REPEAT (10) (+ %REPEAT (9) (.)) +
```

evaluates to:

```
+.....+.....+.....+.....+.....+
```

## The While Function

The WHILE function tests a condition to determine whether the body is to be evaluated. The first argument is evaluated Normally and interpreted as a numeric expression. If the expression is TRUE (=odd) then the body is evaluated, and after each evaluation, the condition is again tested. Reevaluation of the functional string continues until the condition fails. The body of the WHILE function must alter the arguments to the WHILE call, or else, if the body is evaluated once, it will never stop.

The body of the WHILE function is expanded Normally or Literally depending on how the function was called.

Example:

```
%WHILE (%I LT 10)  (...
                    ...
                    %IF (%FLAG) THEN(%DEFINE(I) (20)) FI
                    ...
                    ...)
```

## The Exit Function

The syntax for exit function is:

*exit_function* = EXIT

This function causes termination of processing of the body of the most recently called REPEAT, IF, WHILE, or user-defined macro. The value of the text already evaluated becomes the value of the function. The value of the exit function, itself, is the null string.

Example:

```
%WHILE (%Cond) (...
                ...
                %IF (%FLAG) THEN (%EXIT) FI
                ...
                ...
               )
```

## Console Input and Output

The Macro Processor Language provides functions to allow macro time interaction with the user.

The IN function allows the user to enter a string of characters from the console. This string becomes the value of the function. The IN function will read one line from the console including the terminating carriage return line feed).

The OUT function allows a string to be output to the console output device. It has the null string as a value. Before it is written out, the string will be evaluated Normally or Literally as indicated by the mode of the call to OUT.

The syntax of these two functions is:

*in_function* = IN

*out_function* = OUT ( *balanced_text* )

Examples:

```
%OUT (Enter the date:)
%DEFINE(DATE)(%IN)
```

## The Substring Function

The syntax of the substring function is:

*substr_function* = SUBSTR ( *balanced_text*,*expr1*,*expr2* )

The text string is evaluated Normally or Literally as indicated by the mode of the call to SUBSTR. Assume the characters of the text string are consecutively numbered, starting with one. If expression 1 is zero, or greater than the length of the text string, then the value of this function is the null string. Otherwise, the value of this function is the substring of the text string which begins at character number expression 1 of the text string and continues for expression 2 number of characters or to the end of the string (if the remaining length is less than expression 2).

Examples:

```
%SUBSTR (ABCDEFGH,3,4)
```

has the value "CDEF"

```
%SUBSTR (%(A,B,C,D,E,F,G),2,100)
```

has the value ",B,C,D,E,F,G"

## The Match Function

The syntax of the match function is:

```
match_function =
MATCH ( id1 delimiter_specifier id2 ) ( balanced_text )
```

The match function uses a pattern that is similar to the define pattern of the DEFINE function. It contains two identifiers, both of which are given new values as a result of the MATCH function, and a *delimiter_specifier.*. The identifiers in a MATCH function call are not preceded by the metacharacter (%). The *delimiter_specifier* has the same syntax as that of the DEFINE function. The *balanced_text* is evaluated Normally or Literally, as indicated by the call of MATCH, and then scanned for an occurrence of the delimiter. The algorithm used to find a match is exactly the same as that used to find the delimiter of an argument to a user-defined macro. If a match is found, then id1 will be defined as the value of the characters of the text which precede the matched string and id2 will be defined as the value of the characters of the text which follow the matched string. If a match is not found, then id1 will be defined as the value of the text string, and the id2 will be defined as the null string. The value of the MATCH function is always the null string.

Examples:

Assume XYZ has the value "100,200,300,400,500". Then the call,

```
%MATCH(NEXT,XYZ) (%XYZ)
```

results in NEXT having the value "100" and XYZ having the value "200,300,400,500".

```
%DEFINE (LIST) (FLD1,20H,FLD3)

%WHILE (%LEN(%LIST) NE 0)
(       %MATCH(PARM,LIST) (%LIST)
        MOV   A,%PARM
        INC   A
)
```

The above will generate the following code:

```
MOV   A,FLD1
INC   A
MOV   A,20H
INC   A
MOV   A,FLD3
INC   A
```

Assume that SENTENCE has the value "The Cat is Fat." and that VERB has the value "is", then the call,

```
%MATCH(FIRST %VERB LAST) (%SENTENCE)
```

results in FIRST having the value "The Cat" and LAST having the value "Fat.".

## The Comment Function

The comment function allows the programmer to comment his macro definition and/or source text without having the comments stored into the macro definitions or passed on to the host language processor. The call-literally character may not be present in the call to the comment function. The syntax is:

*comment_function* = ' | *text* | '*linefeed* )

When a comment function is recognized, text is unconditionally skipped until either another apostrophe is recognized, or until a linefeed character is encountered. All text, including the terminating character, is discarded; i.e., the value of the function is always the null string. The comment is always recognized except inside an escape function. Notice that the comment function provides a way in which a programmer can spread out a macro definition on several lines for readability, and yet not include unwanted end of line characters in the called value of the macro.

Examples:

```
%' This comment fits within one line.'

%' This comment continues through the end of the line.
```

## The Metachar Function

The metachar function allows the programmer to change the character that will be recognized by the macro processor as the metachar. If the argument to the function contains more than one character, only the first character becomes the metacharacter. The use of this function requires extreme care. The value of the metachar function is the null string. The syntax is:

*metachar_function* = METACHAR ( *balanced_text* )

The first character of the *balanced_text* is taken to be the new value of the metachar. The following characters cannot be specified as metacharacters: a logical blank, left or right parentheses, an identifier character, an asterisk, or control characters (i.e., ASCII value < 20H).

# APPENDIX F
# RESERVED SYMBOLS

The following is a list of all of the MCS-51 Macro Assembly Language reserved symbols. They can not be used as symbol names or for any other purpose in your program.

| Operators | | | | |
|---|---|---|---|---|
| AND | GT | LOW | NE | SHL |
| EQ | HIGH | LT | NOT | SHR |
| GE | LE | MOD | OR | XOR |

| Opcodes | | | | |
|---|---|---|---|---|
| ACALL | DEC | JNC | NOP | RRC |
| ADD | DIV | JNZ | ORL | SETB |
| ADDC | DJNZ | JZ | POP | SJMP |
| AJMP | INC | LCALL | PUSH | SUBB |
| ANL | JB | LJMP | RET | SWAP |
| CJNE | JBC | MOV | RETI | XCH |
| CLR | JC | MOVC | RL | XCHD |
| CPL | JMP | MOVX | RLC | XRL |
| DA | JNB | MUL | RR | |

| Operands | | | | |
|---|---|---|---|---|
| A | EXTI1 | PC | RD | TB8 |
| AB | F0 | PS | REN | TCON |
| AC | IE | PSW | RESET | TF0 |
| ACC | IE0 | PT0 | RI | TF1 |
| B | IE1 | PT1 | RS0 | TH0 |
| C | IP | PX0 | RS1 | TH1 |
| CY | INT0 | PX1 | RXD | TI |
| DPH | INT1 | R0 | SBUF | TIMER0 |
| DPL | IT0 | R1 | SCON | TIMER1 |
| DPTR | IT1 | R2 | SINT | TL0 |
| EA | OV | R3 | SM0 | TL1 |
| ES | P | R4 | SM1 | TMOD |
| ET0 | P0 | R5 | SM2 | TR0 |
| ET1 | P1 | R6 | SP | TR1 |
| EX0 | P2 | R7 | T0 | TXD |
| EX1 | P3 | RB8 | T1 | WR |
| EXTI0 | P4 | | | |

| Directives | | | | |
|---|---|---|---|---|
| BIT | DATA | DS | END | SET |
| BSEG | DB | DSEG | EQU | XDATA |
| CSEG | DBIT | DW | ORG | XSEG |

# APPENDIX G
# SAMPLE PROGRAM

The following is a fully expanded listing file of an MCS-51 Macro Assembly Language program (an abbreviated form is shown in figure 1-3). It includes two simple ASCII-binary conversion routines, and a set of output routines. The program will run on any member of the MCS-51 family of single-chip processors. It requires a minimum of hardware support. In this assembly the program is set to control a 110 baud terminal. You can change the baud rate by changing the initial value of TH1. The equation shown in the comments shows how to compute the correct initial value.

```
MCS-51 MACRO ASSEMBLER                                                    PAGE     1


ISIS-II MCS-51 MACRO ASSEMBLER V1.0
NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM51 :F1:EXAMPG.SRC NOOBJECT ERRORPRINT


LOC  OBJ             LINE     SOURCE

  0032                  1      FIRST_NUMBER  DATA 50  ; STORAGE LOCATION FOR FIRST NUMBER
  003C                  2      SECOND_NUMBER DATA 60  ; STORAGE FOR SECOND NUMBER
0BB8                    3     ORG 3000
                        4     ; These strings will be placed in high memory
                        5     ; They will be used to output messages to the terminal
                        6     ; The 00H byte at the end of each string identifies the end character
0BB8 54595045           7     TYPO:           DB        'TYPE ^X TO RETYPE A NUMBER',00H
0BBC 205E5820
0BC0 544F2052
0BC4 45545950
0BC8 45204120
0BCC 4E554D42
0BD0 4552
0BD2 00
0BD3 54595045           8     F_NUMB:         DB        'TYPE IN FIRST NUMBER: ',00H
0BD7 20494E20
0BDB 46495253
0BDF 54204E55
0BE3 4D424552
0BE7 3A20
0BE9 00
0BEA 54595045           9     S_NUMB:         DB        'TYPE IN SECOND NUMBER: ',00H
0BEE 20494E20
0BF2 5345434F
0BF6 4E44204E
0BFA 554D4245
0BFE 523A20
0C01 00
0C02 54484520          10     SUM:                DB        'THE SUM IS ',00H
0C06 53554D20
0C0A 495320
0C0D 00
0000                   11     ORG 0
                       12     ;  The following instructions prepare the serial port to recieve and
                       13     ;   send data at 110 baud
                       14     ;   Hardware assumptions:
                       15     ;                         Proper power supply
                       16     ;                         Logic to modify TTL signal to current loop
                       17     ;                         Necessary cabeling to connect terminal
                       18     ;
0000 758920            19     MOV TMOD,#00100000B     ; SET TIMER MODE TO AUTO-RELOAD
0003 758D03            20     MOV TH1,#(-253)         ; SET TIMER FOR 110 BAUD
                       21                             ; 110 = 10.7MHz/12*16*2*253
                       22                             ; 110 = desired baud rate
                       23                             ; 10.7MHz = external clock rate
                       24                             ; -253 = timer preset value
                       25                             ; 12*16*2 = conversion constant
0006 7598DA            26     MOV SCON,#11011010B     ; PREPARE SERIAL PORT FOR OPERATION
0009 D28E              27     SETB TR1                ; START CLOCK
                       28     START:
```

Figure G-1. Sample Program

937-26

```
MCS-51 MACRO ASSEMBLER                                                  PAGE    2

LOC  OBJ            LINE     SOURCE

                      29     ; This part of program starts communication and gets first number
000B 900BB8           30               MOV DPTR,#TYPO
000E 12006C           31               CALL PUT_STRING              ;  OUTPUT HOW TO RECOVER FROM TYPO
0011 120061           32               CALL PUT_CRLF
0014 900BD3           33                    MOV DPTR,#F_NUMB        ; GET ADDRESS OF DB STRING
0017 12006C           34                    CALL PUT_STRING         ; OUTPUT STRING FOR FIRST NUMBER
001A 120061           35                    CALL PUT_CRLF           ; OUTPUT CARRIAGE RETURN LINE FEED
001D 7832             36                    MOV R0,#FIRST_NUMBER
001F 120077           37                    CALL GET_NUMB           ; GET FIRST NUMBER
0022 120061           38                    CALL PUT_CRLF
                      39     ; THIS SECTION GETS SECOND NUMBER FROM CONSOLE
0025 900BEA           40                    MOV DPTR,#S_NUMB        ; OUTPUT STRING FOR SECOND NUMBER
0028 12006C           41                    CALL PUT_STRING
002B 120061           42                    CALL PUT_CRLF
002E 783C             43                    MOV R0,#SECOND_NUMBER
0030 120077           44                    CALL GET_NUMB           ; GET SECOND NUMBER
0033 120061           45                    CALL PUT_CRLF
                      46     ; THIS SECTION OF CODE CONVERTS ASCII NUMBERS TO BINARY
0036 7932             47                    MOV R1,#FIRST_NUMBER
0038 1200BF           48                    CALL ASCBIN             ; TRANSLATE ASCII STRING TO BINARY NUMBER
003B 793C             49                    MOV R1,#SECOND_NUMBER
003D 1200BF           50                    CALL ASCBIN             ; TRANSLATE SECOND ASCII STRING
0040 E532             51                    MOV A,FIRST_NUMBER      ; GET RESULT OF FIRST TRANSLATION
                      52     ; ADD NUMBERS AND CHANGE BINARY SUM TO ASCII STRING
0042 253C             53                    ADD A,SECOND_NUMBER     ; ADD BOTH NUMBERS
0044 F532             54                    MOV FIRST_NUMBER,A
0046 7932             55                    MOV R1,#FIRST_NUMBER     ; PREPARE FOR RETRANSLATION
0048 120099           56                    CALL BINASC             ; TRANSLATE BINARY NUMBER TO ASCII
004B 900C02           57                    MOV DPTR,#SUM
                      58     ; OUTPUT SUM STRING AND CONVERTED ASCII SUM
004E 12006C           59                    CALL PUT_STRING         ; OUTPUT SUM STRING
0051 AA04             60                    MOV R2,4
0053 7932             61                    MOV R1,#FIRST_NUMBER
0055 E7               62     PUT_SUM:       MOV A,@R1
0056 120091           63                    CALL PUT_CHAR
0059 09               64                    INC R1
005A DAF9             65                    DJNZ R2,PUT_SUM
005C 120061           66                    CALL PUT_CRLF
005F 80AA             67                    JMP START
                      68 +1  $eject
```

Figure G-1. Sample Program (Cont'd.)

937-27

```
MCS-51 MACRO ASSEMBLER                                                  PAGE    3

LOC  OBJ            LINE     SOURCE

                      69     ;     BEGIN SERVICE ROUTINES
  000D                70     CR                EQU               0DH
  000A                71     LF                EQU               0AH
                      72     PUT_CRLF:
                      73     ;  THIS ROUTINE OUTPUTS A CARRIAGE RETURN AND A LINE FEED (0DH,0AH)
0061 740D             74                       MOV A,#CR
0063 120091           75                       CALL PUT_CHAR
0066 740A             76                       MOV A,#LF
0068 120091           77                       CALL PUT_CHAR
006B 22               78                       RET
                      79     PUT_STRING:
                      80     ; THIS ROUTINE OUTPUTS A CHARACTER STRING LOCATED IN CODE
                      81     ;   MEMORY.  THE ADDRESS MUST BE CONTAINED IN CODE MEMORY
006C E4               82                       CLR A
006D 93               83                       MOVC A,@A+DPTR
006E 120091           84                       CALL PUT_CHAR
0071 6003             85                       JZ EXIT
0073 A3               86                       INC DPTR
0074 80F6             87                       JMP PUT_STRING
                      88     EXIT:
0076 22               89                       RET
                      90     GET_NUMB:
                      91     ;  THIS ROUTINE TAKES A 4 CHARACTER STRING FROM THE
                      92     ;  CONSOLE AND STORES THE STRING IN MEMORY AT THE
                      93     ;  ADDRESS CONTAINED IN R0
                      94     ; IF A ^X IS RECEIVED IT STARTS OVER
0077 7A04             95                       MOV R2,#4
                      96     G_N_LOOP:
0079 120089           97                       CALL GET_CHAR
007C C2E7             98                       CLR ACC.7
007E B41904           99               CJNE A,#19H,SKIP      ; COMPARE TO SKIP
0081 1161            100               CALL PUT_CRLF
0083 80F2            101               JMP GET_NUMB
                     102     SKIP:
0085 F6              103                       MOV @R0,A
0086 DAF1            104                       DJNZ R2,G_N_LOOP
0088 22              105                       RET
                     106     GET_CHAR:
                     107               ; THIS ROUTINE GETS A SINGLE CHARACTER FROM THE CONSOLE
0089 3098FD          108                       JNB RI,$        ; LOOP HERE UNTIL CHARACTER RECIEVED
008C C298            109                       CLR RI
008E E599            110                       MOV A,SBUF
0090 22              111                       RET
                     112     PUT_CHAR:
                     113               ; THIS ROUTINE OUTPUTS A SINGLE CHARACTER TO THE CONSOLE
0091 3099FD          114                       JNB TI,$        ; LOOP HERE UNTIL CHARACTER TRANSMITTED
0094 C299            115                       CLR TI
0096 F599            116                       MOV SBUF,A
0098 22              117                       RET
  REG                118     NUMB_PTR    EQU         R1
  0030               119     ZERO        EQU         ('0')
  002D               120     MINUS       EQU         ('-')
  002B               121     PLUS        EQU         ('+')
                     122 +1  $    EJE..:
```

Figure G-1. Sample Program (Cont'd.)

937-28

```
MCS-51 MACRO ASSEMBLER                                                  PAGE    4

LOC  OBJ            LINE     SOURCE

                     123     ; ***************************************************************************
                     124     ;
                     125     ; This routine converts a binary 2's complement number to a 4 character
                     126     ; ASCII string.
                     127     ; INPUT:
                     128     ;       The binary value must be located in memory at the address contained
                     129     ;       in register 1.
                     130     ; OUTPUT:
                     131     ;        The 4 charater result is placed in memory with the first character
                     132     ;        at the address contained in register 1.
                     133     ; NOTES:
                     134     ;        The contents of register A and B will be destroyed.
                     135     ;        The contents of the memory location initially addressed by
                     136     ;        register 1 will be replaced with the first charater in the
                     137     ;        resulting character string.
                     138     ; ***************************************************************************
                     139     BINASC:
  00E7               140     SIGN    BIT         ACC.7
0099 E7              141             MOV         A,@NUMB_PTR             ; Get number
009A 772B            142             MOV         @NUMB_PTR,#PLUS
009C 30E704          143             JNB         SIGN,VAL                ; Test bit 7 for] sign
009F 772D            144                     MOV         @NUMB_PTR,#MINUS  ;  Insert negative sign
                     145     ;  Change negative number to positive.
00A1 14              146             DEC         A
00A2 F4              147             CPL         A
                     148     ;  Now work on first digit
                     149     VAL:
00A3 09              150                     INC         NUMB_PTR
                     151     ; Factor out first digit
00A4 75F064          152                     MOV         B,#100
00A7 84              153                     DIV AB
00A8 2430            154                     ADD A,#ZERO
00AA F7              155                     MOV @NUMB_PTR,A
00AB 09              156                     INC NUMB_PTR
                     157     ; Factor out second digit from remainder
00AC E5F0            158                     MOV A,B
00AE 75F00A          159                     MOV B,#10
00B1 84              160                     DIV AB
00B2 2430            161                     ADD A,#ZERO
00B4 F7              162                     MOV @NUMB_PTR,A
00B5 09              163                     INC NUMB_PTR
                     164     ;  Get third and final digit
00B6 E5F0            165                     MOV A,B
00B8 2430            166                     ADD A,#ZERO
00BA F7              167                     MOV @NUMB_PTR,A
                     168     ;  restore NUMB_PTR
00BB 19              169                     DEC NUMB_PTR
00BC 19              170                     DEC NUMB_PTR
00BD 19              171                     DEC NUMB_PTR
00BE 22              172                     RET
                     173 +1  $   EJECT
```

Figure G-1. Sample Program (Cont'd.)

937-29

```
MCS-51 MACRO ASSEMBLER                                                  PAGE    5

LOC  OBJ             LINE     SOURCE

                      174     ; ***************************************************************************
                      175     ; This routine takes a 4 character string located in memory and converts
                      176     ; it to a binary 2's complement number.
                      177     ; The number must begin with a sign character ('+' or '-'), and be
                      178     ; between -128 and +127.
                      179     ; INPUT:
                      180     ;       Four ASCII characters a sign character followed a '0' or a '1'
                      181     ;       and the last 2 characters can be any digit.
                      182     ;       The contents of register 1 must point to the sign character.
                      183     ; OUTPUT:
                      184     ;        A binary 2's complement representation of the value of the
                      185     ;        character string.
                      186     ;        Register 1 contains the address of the binary value.
                      187     ; NOTES:
                      188     ;        The contents of the memory location initially
                      189     ;        addressed by register 1 is destroyed.
                      190     ;        The contents of registers 7 and B and the accumulator
                      191     ;        are destroyed.
                      192     ; ***************************************************************************
  REG                 193     TEMP            EQU  R7
                      194     ASCBIN:
                      195     ; Go right to number compute sign at end
00BF 09               196                     INC NUMB_PTR
00C0 E7               197                     MOV A,@NUMB_PTR
00C1 9430             198                     SUBB A,#ZERO
00C3 75F064           199                     MOV B,#100
00C6 A4               200                     MUL AB
                      201     ;  Store first digit's value and go to next digit
00C7 FF               202                     MOV TEMP,A
00C8 09               203                     INC NUMB_PTR
00C9 E7               204                     MOV A,@NUMB_PTR
00CA 9430             205                     SUBB A,#ZERO
00CC 75F00A           206                     MOV B,#10
00CF A4               207                     MUL AB
                      208     ;  Add first digit value to secon store and go to third digit
00D0 2F               209                     ADD A,TEMP
00D1 FF               210                     MOV TEMP,A
00D2 09               211                     INC NUMB_PTR
00D3 E7               212                     MOV A,@NUMB_PTR
00D4 9430             213                     SUBB A,#ZERO
                      214     ;  Add third digit value to total.  Store and go back for sign
00D6 2F               215                     ADD A,TEMP
00D7 FF               216                     MOV TEMP,A
00D8 19               217                     DEC NUMB_PTR
00D9 19               218                     DEC NUMB_PTR
00DA 19               219                     DEC NUMB_PTR
00DB E7               220                     MOV A,@NUMB_PTR
                      221     ; Test for sign value
00DC B42D04           222                     CJNE A,#MINUS,POS
00DF EF               223                     MOV A,TEMP
00E0 F4               224                     CPL A
00E1 04               225                     INC A
00E2 FF               226                     MOV TEMP,A
00E3 EF               227     POS:                MOV A,TEMP
                      228     ; store result and return


MCS-51 MACRO ASSEMBLER                                                  PAGE    6

LOC  OBJ             LINE     SOURCE

00E4 F7               229                     MOV @NUMB_PTR,A
00E5 22               230                     RET
                      231             END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure G-1. Sample Program (Cont'd.)

937-30

# APPENDIX H
# REFERENCE TABLES

This appendix contains the following general reference tables:

- ASCII codes
- Powers of two
- Powers of 16 (in base 10)
- Powers of 10 (in base 16)
- Hexadecimal-decimal integer conversion

## ASCII Codes

The 8051 uses the 7-bit ASCII code, with the high-order 8th bit (parity bit) always reset.

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|---|---|
| NUL | 00 |
| SOH | 01 |
| STX | 02 |
| ETX | 03 |
| EOT | 04 |
| ENQ | 05 |
| ACK | 06 |
| BEL | 07 |
| BS | 08 |
| HT | 09 |
| LF | 0A |
| VT | 0B |
| FF | 0C |
| CR | 0D |
| SO | 0E |
| SI | 0F |
| DLE | 10 |
| DC1 (X-ON) | 11 |
| DC2 (TAPE) | 12 |
| DC3 (X-OFF) | 13 |
| DC4 (~~TAPE~~) | 14 |
| NAK | 15 |
| SYN | 16 |
| ETB | 17 |
| CAN | 18 |
| EM | 19 |
| SUB | 1A |
| ESC | 1B |
| FS | 1C |
| GS | 1D |
| RS | 1E |
| US | 1F |
| SP | 20 |
| ! | 21 |
| " | 22 |
| # | 23 |
| $ | 24 |
| % | 25 |
| & | 26 |
| ' | 27 |
| ( | 28 |
| ) | 29 |
| * | 2A |

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|---|---|
| + | 2B |
| , | 2C |
| – | 2D |
| . | 2E |
| / | 2F |
| 0 | 30 |
| 1 | 31 |
| 2 | 32 |
| 3 | 33 |
| 4 | 34 |
| 5 | 35 |
| 6 | 36 |
| 7 | 37 |
| 8 | 38 |
| 9 | 39 |
| : | 3A |
| ; | 3B |
| < | 3C |
| = | 3D |
| > | 3E |
| ? | 3F |
| @ | 40 |
| A | 41 |
| B | 42 |
| C | 43 |
| D | 44 |
| E | 45 |
| F | 46 |
| G | 47 |
| H | 48 |
| I | 49 |
| J | 4A |
| K | 4B |
| L | 4C |
| M | 4D |
| N | 4E |
| O | 4F |
| P | 50 |
| Q | 51 |
| R | 52 |
| S | 53 |
| T | 54 |
| U | 55 |

| GRAPHIC OR CONTROL | ASCII (HEXADECIMAL) |
|---|---|
| V | 56 |
| W | 57 |
| X | 58 |
| Y | 59 |
| Z | 5A |
| [ | 5B |
| \ | 5C |
| ] | 5D |
| ∧ (↑) | 5E |
| – (←) | 5F |
| ` | 60 |
| a | 61 |
| b | 62 |
| c | 63 |
| d | 64 |
| e | 65 |
| f | 66 |
| g | 67 |
| h | 68 |
| i | 69 |
| j | 6A |
| k | 6B |
| l | 6C |
| m | 6D |
| n | 6E |
| o | 6F |
| p | 70 |
| q | 71 |
| r | 72 |
| s | 73 |
| t | 74 |
| u | 75 |
| v | 76 |
| w | 77 |
| x | 78 |
| y | 79 |
| z | 7A |
| { | 7B |
| \| | 7C |
| } (ALT MODE) | 7D |
| ~ | 7E |
| DEL (RUB OUT) | 7F |

## POWERS OF TWO

| $2^n$ | n | $2^{-n}$ |
|---|---|---|
| 1 | 0 | 1.0 |
| 2 | 1 | 0.5 |
| 4 | 2 | 0.25 |
| 8 | 3 | 0.125 |
| 16 | 4 | 0.062 5 |
| 32 | 5 | 0.031 25 |
| 64 | 6 | 0.015 625 |
| 128 | 7 | 0.007 812 5 |
| 256 | 8 | 0.003 906 25 |
| 512 | 9 | 0.001 953 125 |
| 1 024 | 10 | 0.000 976 562 5 |
| 2 048 | 11 | 0.000 488 281 25 |
| 4 096 | 12 | 0.000 244 140 625 |
| 8 192 | 13 | 0.000 122 070 312 5 |
| 16 384 | 14 | 0.000 061 035 156 25 |
| 32 768 | 15 | 0.000 030 517 578 125 |
| 65 536 | 16 | 0.000 015 258 789 062 5 |
| 131 072 | 17 | 0.000 007 629 394 531 25 |
| 262 144 | 18 | 0.000 003 814 697 265 625 |
| 524 288 | 19 | 0.000 001 907 348 632 812 5 |
| 1 048 576 | 20 | 0.000 000 953 674 316 406 25 |
| 2 097 152 | 21 | 0.000 000 476 837 158 203 125 |
| 4 194 304 | 22 | 0.000 000 238 418 579 101 562 5 |
| 8 388 608 | 23 | 0.000 000 119 209 289 550 781 25 |
| 16 777 216 | 24 | 0.000 000 059 604 644 775 390 625 |
| 33 554 432 | 25 | 0.000 000 029 802 322 387 695 312 5 |
| 67 108 864 | 26 | 0.000 000 014 901 161 193 847 656 25 |
| 134 217 728 | 27 | 0.000 000 007 450 580 596 923 828 125 |
| 268 435 456 | 28 | 0.000 000 003 725 290 298 461 914 062 5 |
| 536 870 912 | 29 | 0.000 000 001 862 645 149 230 957 031 25 |
| 1 073 741 824 | 30 | 0.000 000 000 931 322 574 615 478 515 625 |
| 2 147 483 648 | 31 | 0.000 000 000 465 661 287 307 739 257 812 5 |
| 4 294 967 296 | 32 | 0.000 000 000 232 830 643 653 869 628 906 25 |
| 8 589 934 592 | 33 | 0.000 000 000 116 415 321 826 934 814 453 125 |
| 17 179 869 184 | 34 | 0.000 000 000 058 207 660 913 467 407 226 562 5 |
| 34 359 738 368 | 35 | 0.000 000 000 029 103 830 456 733 703 613 281 25 |
| 68 719 476 736 | 36 | 0.000 000 000 014 551 915 228 366 851 806 640 625 |
| 137 438 953 472 | 37 | 0.000 000 000 007 275 957 614 183 425 903 320 312 5 |
| 274 877 906 944 | 38 | 0.000 000 000 003 637 978 807 091 712 951 660 156 25 |
| 549 755 813 888 | 39 | 0.000 000 000 001 818 989 403 545 856 475 830 078 125 |
| 1 099 511 627 776 | 40 | 0.000 000 000 000 909 494 701 772 928 237 915 039 062 5 |
| 2 199 023 255 552 | 41 | 0.000 000 000 000 454 747 350 886 464 118 957 519 531 25 |
| 4 398 046 511 104 | 42 | 0.000 000 000 000 227 373 675 443 232 059 478 759 765 625 |
| 8 796 093 022 208 | 43 | 0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5 |
| 17 592 186 044 416 | 44 | 0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25 |
| 35 184 372 088 832 | 45 | 0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125 |
| 70 368 744 177 664 | 46 | 0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5 |
| 140 737 488 355 328 | 47 | 0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25 |
| 281 474 976 710 656 | 48 | 0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625 |
| 562 949 953 421 312 | 49 | 0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5 |
| 1 125 899 906 842 624 | 50 | 0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25 |
| 2 251 799 813 685 248 | 51 | 0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125 |
| 4 503 599 627 370 496 | 52 | 0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5 |
| 9 007 199 254 740 992 | 53 | 0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25 |
| 18 014 398 509 481 984 | 54 | 0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625 |
| 36 028 797 018 963 968 | 55 | 0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5 |
| 72 057 594 037 927 936 | 56 | 0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25 |
| 144 115 188 075 855 872 | 57 | 0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125 |
| 288 230 376 151 711 744 | 58 | 0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5 |
| 576 460 752 303 423 488 | 59 | 0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25 |
| 1 152 921 504 606 846 976 | 60 | 0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625 |
| 2 305 843 009 213 693 952 | 61 | 0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5 |
| 4 611 686 018 427 387 904 | 62 | 0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25 |
| 9 223 372 036 854 775 808 | 63 | 0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125 |

## POWERS OF 16 (IN BASE 10)

| $16^n$ | n | $16^{-n}$ | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 0.10000 | 00000 | 00000 | 00000 | x $10$ |
| 16 | 1 | 0.62500 | 00000 | 00000 | 00000 | x $10^{-1}$ |
| 256 | 2 | 0.39062 | 50000 | 00000 | 00000 | x $10^{-2}$ |
| 4 096 | 3 | 0.24414 | 06250 | 00000 | 00000 | x $10^{-3}$ |
| 65 536 | 4 | 0.15258 | 78906 | 25000 | 00000 | x $10^{-4}$ |
| 1 048 576 | 5 | 0.95367 | 43164 | 06250 | 00000 | x $10^{-6}$ |
| 16 777 216 | 6 | 0.59604 | 64477 | 53906 | 25000 | x $10^{-7}$ |
| 268 435 456 | 7 | 0.37252 | 90298 | 46191 | 40625 | x $10^{-8}$ |
| 4 294 967 296 | 8 | 0.23283 | 06436 | 53869 | 62891 | x $10^{-9}$ |
| 68 719 476 736 | 9 | 0.14551 | 91522 | 83668 | 51807 | x $10^{-10}$ |
| 1 099 511 627 776 | 10 | 0.90949 | 47017 | 72928 | 23792 | x $10^{-12}$ |
| 17 592 186 044 416 | 11 | 0.56843 | 41886 | 08080 | 14870 | x $10^{-13}$ |
| 281 474 976 710 656 | 12 | 0.35527 | 13678 | 80050 | 09294 | x $10^{-14}$ |
| 4 503 599 627 370 496 | 13 | 0.22204 | 46049 | 25031 | 30808 | x $10^{-15}$ |
| 72 057 594 037 927 936 | 14 | 0.13877 | 78780 | 78144 | 56755 | x $10^{-16}$ |
| 1 152 921 504 606 846 976 | 15 | 0.86736 | 17379 | 88403 | 54721 | x $10^{-18}$ |

## POWERS OF 10 (IN BASE 16)

| $10^n$ | n | $10^{-n}$ | | | | |
|---|---|---|---|---|---|---|
| 1 | 0 | 1.0000 | 0000 | 0000 | 0000 | |
| A | 1 | 0.1999 | 9999 | 9999 | 999A | |
| 64 | 2 | 0.28F5 | C28F | 5C28 | F5C3 | x $16^{-1}$ |
| 3E8 | 3 | 0.4189 | 374B | C6A7 | EF9E | x $16^{-2}$ |
| 2710 | 4 | 0.68DB | 8BAC | 710C | B296 | x $16^{-3}$ |
| 1 86A0 | 5 | 0.A7C5 | AC47 | 1B47 | 8423 | x $16^{-4}$ |
| F 4240 | 6 | 0.10C6 | F7A0 | B5ED | 8D37 | x $16^{-4}$ |
| 98 9680 | 7 | 0.1AD7 | F29A | BCAF | 4858 | x $16^{-5}$ |
| 5F5 E100 | 8 | 0.2AF3 | 1DC4 | 6118 | 73BF | x $16^{-6}$ |
| 3B9A CA00 | 9 | 0.44B8 | 2FA0 | 9B5A | 52CC | x $16^{-7}$ |
| 2 540B E400 | 10 | 0.6DF3 | 7F67 | SEF6 | EADF | x $16^{-8}$ |
| 17 4876 E800 | 11 | 0.AFEB | FF0B | CB24 | AAFF | x $16^{-9}$ |
| E8 D4A5 1000 | 12 | 0.1197 | 9981 | 2DEA | 1119 | x $16^{-9}$ |
| 918 4E72 A000 | 13 | 0.1C25 | C268 | 4976 | 81C2 | x $16^{-10}$ |
| 5AF3 107A 4000 | 14 | 0.2D09 | 370D | 4257 | 3604 | x $16^{-11}$ |
| 3 8D7E A4C6 8000 | 15 | 0.480E | BE7B | 9D58 | 566D | x $16^{-12}$ |
| 23 8652 6FC1 0000 | 16 | 0.734A | CA5F | 6226 | F0AE | x $16^{-13}$ |
| 163 4578 5D8A 0000 | 17 | 0.B877 | AA32 | 36A4 | B449 | x $16^{-14}$ |
| DE0 B6B3 A764 0000 | 18 | 0.1272 | 5DD1 | D243 | ABA1 | x $16^{-14}$ |
| 8AC7 2304 89E8 0000 | 19 | 0.1D83 | C94F | B6D2 | AC35 | x $16^{-15}$ |

# HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the following figures:

| Hexadecimal | Decimal | Hexadecimal | Decimal |
|---|---|---|---|
| 01 000 | 4 096 | 20 000 | 131 072 |
| 02 000 | 8 192 | 30 000 | 196 608 |
| 03 000 | 12 288 | 40 000 | 262 144 |
| 04 000 | 16 384 | 50 000 | 327 680 |
| 05 000 | 20 480 | 60 000 | 393 216 |
| 06 000 | 24 576 | 70 000 | 458 752 |
| 07 000 | 28 672 | 80 000 | 524 288 |
| 08 000 | 32 768 | 90 000 | 589 824 |
| 09 000 | 36 864 | A0 000 | 655 360 |
| 0A 000 | 40 960 | B0 000 | 720 896 |
| 0B 000 | 45 056 | C0 000 | 786 432 |
| 0C 000 | 49 152 | D0 000 | 851 968 |
| 0D 000 | 53 248 | E0 000 | 917 504 |
| 0E 000 | 57 344 | F0 000 | 983 040 |
| 0F 000 | 61 440 | 100 000 | 1 048 576 |
| 10 000 | 65 536 | 200 000 | 2 097 152 |
| 11 000 | 69 632 | 300 000 | 3 145 728 |
| 12 000 | 73 728 | 400 000 | 4 194 304 |
| 13 000 | 77 824 | 500 000 | 5 242 880 |
| 14 000 | 81 920 | 600 000 | 6 291 456 |
| 15 000 | 86 016 | 700 000 | 7 340 032 |
| 16 000 | 90 112 | 800 000 | 8 388 608 |
| 17 000 | 94 208 | 900 000 | 9 437 184 |
| 18 000 | 98 304 | A00 000 | 10 485 760 |
| 19 000 | 102 400 | B00 000 | 11 534 336 |
| 1A 000 | 106 496 | C00 000 | 12 582 912 |
| 1B 000 | 110 592 | D00 000 | 13 631 488 |
| 1C 000 | 114 688 | E00 000 | 14 680 064 |
| 1D 000 | 118 784 | F00 000 | 15 728 640 |
| 1E 000 | 122 880 | 1 000 000 | 16 777 216 |
| 1F 000 | 126 976 | 2 000 000 | 33 554 432 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000 | 0000 | 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | 0007 | 0008 | 0009 | 0010 | 0011 | 0012 | 0013 | 0014 | 0015 |
| 010 | 0016 | 0017 | 0018 | 0019 | 0020 | 0021 | 0022 | 0023 | 0024 | 0025 | 0026 | 0027 | 0028 | 0029 | 0030 | 0031 |
| 020 | 0032 | 0033 | 0034 | 0035 | 0036 | 0037 | 0038 | 0039 | 0040 | 0041 | 0042 | 0043 | 0044 | 0045 | 0046 | 0047 |
| 030 | 0048 | 0049 | 0050 | 0051 | 0052 | 0053 | 0054 | 0055 | 0056 | 0057 | 0058 | 0059 | 0060 | 0061 | 0062 | 0063 |
| 040 | 0064 | 0065 | 0066 | 0067 | 0068 | 0069 | 0070 | 0071 | 0072 | 0073 | 0074 | 0075 | 0076 | 0077 | 0078 | 0079 |
| 050 | 0080 | 0081 | 0082 | 0083 | 0084 | 0085 | 0086 | 0087 | 0088 | 0089 | 0090 | 0091 | 0092 | 0093 | 0094 | 0095 |
| 060 | 0096 | 0097 | 0098 | 0099 | 0100 | 0101 | 0102 | 0103 | 0104 | 0105 | 0106 | 0107 | 0108 | 0109 | 0110 | 0111 |
| 070 | 0112 | 0113 | 0114 | 0115 | 0116 | 0117 | 0118 | 0119 | 0120 | 0121 | 0122 | 0123 | 0124 | 0125 | 0126 | 0127 |
| 080 | 0128 | 0129 | 0130 | 0131 | 0132 | 0133 | 0134 | 0135 | 0136 | 0137 | 0138 | 0139 | 0140 | 0141 | 0142 | 0143 |
| 090 | 0144 | 0145 | 0146 | 0147 | 0148 | 0149 | 0150 | 0151 | 0152 | 0153 | 0154 | 0155 | 0156 | 0157 | 0158 | 0159 |
| 0A0 | 0160 | 0161 | 0162 | 0163 | 0164 | 0165 | 0166 | 0167 | 0168 | 0169 | 0170 | 0171 | 0172 | 0173 | 0174 | 0175 |
| 0B0 | 0176 | 0177 | 0178 | 0179 | 0180 | 0181 | 0182 | 0183 | 0184 | 0185 | 0186 | 0187 | 0188 | 0189 | 0190 | 0191 |
| 0C0 | 0192 | 0193 | 0194 | 0195 | 0196 | 0197 | 0198 | 0199 | 0200 | 0201 | 0202 | 0203 | 0204 | 0205 | 0206 | 0207 |
| 0D0 | 0208 | 0209 | 0210 | 0211 | 0212 | 0213 | 0214 | 0215 | 0216 | 0217 | 0218 | 0219 | 0220 | 0221 | 0222 | 0223 |
| 0E0 | 0224 | 0225 | 0226 | 0227 | 0228 | 0229 | 0230 | 0231 | 0232 | 0233 | 0234 | 0235 | 0236 | 0237 | 0238 | 0239 |
| 0F0 | 0240 | 0241 | 0242 | 0243 | 0244 | 0245 | 0246 | 0247 | 0248 | 0249 | 0250 | 0251 | 0252 | 0253 | 0254 | 0255 |

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 0256 | 0257 | 0258 | 0259 | 0260 | 0261 | 0262 | 0263 | 0264 | 0265 | 0266 | 0267 | 0268 | 0269 | 0270 | 0271 |
| 110 | 0272 | 0273 | 0274 | 0275 | 0276 | 0277 | 0278 | 0279 | 0280 | 0281 | 0282 | 0283 | 0284 | 0285 | 0286 | 0287 |
| 120 | 0288 | 0289 | 0290 | 0291 | 0292 | 0293 | 0294 | 0295 | 0296 | 0297 | 0298 | 0299 | 0300 | 0301 | 0302 | 0303 |
| 130 | 0304 | 0305 | 0306 | 0307 | 0308 | 0309 | 0310 | 0311 | 0312 | 0313 | 0314 | 0315 | 0316 | 0317 | 0318 | 0319 |
| 140 | 0320 | 0321 | 0322 | 0323 | 0324 | 0325 | 0326 | 0327 | 0328 | 0329 | 0330 | 0331 | 0331 | 0333 | 0334 | 0335 |
| 150 | 0336 | 0337 | 0338 | 0339 | 0340 | 0341 | 0342 | 0343 | 0344 | 0345 | 0346 | 0347 | 0348 | 0349 | 0350 | 0351 |
| 160 | 0352 | 0353 | 0354 | 0355 | 0356 | 0357 | 0358 | 0359 | 0360 | 0361 | 0362 | 0363 | 0364 | 0365 | 0366 | 0367 |
| 170 | 0368 | 0369 | 0370 | 0371 | 0372 | 0373 | 0374 | 0375 | 0376 | 0377 | 0378 | 0379 | 0380 | 0381 | 0382 | 0383 |
| 180 | 0384 | 0385 | 0386 | 0387 | 0388 | 0389 | 0390 | 0391 | 0392 | 0393 | 0394 | 0395 | 0396 | 0397 | 0398 | 0399 |
| 190 | 0400 | 0401 | 0402 | 0403 | 0404 | 0405 | 0406 | 0407 | 0408 | 0409 | 0410 | 0411 | 0412 | 0413 | 0414 | 0415 |
| 1A0 | 0416 | 0417 | 0418 | 0419 | 0420 | 0421 | 0422 | 0423 | 0424 | 0425 | 0426 | 0427 | 0428 | 0429 | 0430 | 0431 |
| 1B0 | 0432 | 0433 | 0434 | 0435 | 0436 | 0437 | 0438 | 0439 | 0440 | 0441 | 0442 | 0443 | 0444 | 0445 | 0446 | 0447 |
| 1C0 | 0448 | 0449 | 0450 | 0451 | 0452 | 0453 | 0454 | 0455 | 0456 | 0457 | 0458 | 0459 | 0460 | 0461 | 0462 | 0463 |
| 1D0 | 0464 | 0465 | 0466 | 0467 | 0468 | 0469 | 0470 | 0471 | 0472 | 0473 | 0474 | 0475 | 0476 | 0477 | 0478 | 0479 |
| 1E0 | 0480 | 0481 | 0482 | 0483 | 0484 | 0485 | 0486 | 0487 | 0488 | 0489 | 0490 | 0491 | 0492 | 0493 | 0494 | 0495 |
| 1F0 | 0496 | 0497 | 0498 | 0499 | 0500 | 0501 | 0502 | 0503 | 0504 | 0505 | 0506 | 0507 | 0508 | 0509 | 0510 | 0511 |
| 200 | 0512 | 0513 | 0514 | 0515 | 0516 | 0517 | 0518 | 0519 | 0520 | 0521 | 0522 | 0523 | 0524 | 0525 | 0526 | 0527 |
| 210 | 0528 | 0529 | 0530 | 0531 | 0532 | 0533 | 0534 | 0535 | 0536 | 0537 | 0538 | 0539 | 0540 | 0541 | 0542 | 0543 |
| 220 | 0544 | 0545 | 0546 | 0547 | 0548 | 0549 | 0550 | 0551 | 0552 | 0553 | 0554 | 0555 | 0556 | 0557 | 0558 | 0559 |
| 230 | 0560 | 0561 | 0562 | 0563 | 0564 | 0565 | 0566 | 0567 | 0568 | 0569 | 0570 | 0571 | 0572 | 0573 | 0574 | 0575 |
| 240 | 0576 | 0577 | 0578 | 0579 | 0580 | 0581 | 0582 | 0583 | 0584 | 0585 | 0586 | 0587 | 0588 | 0589 | 0590 | 0591 |
| 250 | 0592 | 0593 | 0594 | 0595 | 0596 | 0597 | 0598 | 0599 | 0600 | 0601 | 0602 | 0603 | 0604 | 0605 | 0606 | 0607 |
| 260 | 0608 | 0609 | 0610 | 0611 | 0612 | 0613 | 0614 | 0615 | 0616 | 0617 | 0618 | 0619 | 0620 | 0621 | 0622 | 0623 |
| 270 | 0624 | 0625 | 0626 | 0627 | 0628 | 0629 | 0630 | 0631 | 0632 | 0633 | 0634 | 0635 | 0636 | 0637 | 0638 | 0639 |
| 280 | 0640 | 0641 | 0642 | 0643 | 0644 | 0645 | 0646 | 0647 | 0648 | 0649 | 0650 | 0651 | 0652 | 0653 | 0654 | 0655 |
| 290 | 0656 | 0657 | 0658 | 0659 | 0660 | 0661 | 0662 | 0663 | 0664 | 0665 | 0666 | 0667 | 0668 | 0669 | 0670 | 0671 |
| 2A0 | 0672 | 0673 | 0674 | 0675 | 0676 | 0677 | 0678 | 0679 | 0680 | 0681 | 0682 | 0683 | 0684 | 0685 | 0686 | 0687 |
| 2B0 | 0688 | 0689 | 0690 | 0691 | 0692 | 0693 | 0694 | 0695 | 0696 | 0697 | 0698 | 0699 | 0700 | 0701 | 0702 | 0703 |
| 2C0 | 0704 | 0705 | 0706 | 0707 | 0708 | 0709 | 0710 | 0711 | 0712 | 0713 | 0714 | 0715 | 0716 | 0717 | 0718 | 0719 |
| 2D0 | 0720 | 0721 | 0722 | 0723 | 0724 | 0725 | 0726 | 0727 | 0728 | 0729 | 0730 | 0731 | 0732 | 0733 | 0734 | 0735 |
| 2E0 | 0736 | 0737 | 0738 | 0739 | 0740 | 0741 | 0742 | 0743 | 0744 | 0745 | 0746 | 0747 | 0748 | 0749 | 0750 | 0751 |
| 2F0 | 0752 | 0753 | 0754 | 0755 | 0756 | 0757 | 0758 | 0759 | 0760 | 0761 | 0762 | 0763 | 0764 | 0765 | 0766 | 0767 |
| 300 | 0768 | 0769 | 0770 | 0771 | 0772 | 0773 | 0774 | 0775 | 0776 | 0777 | 0778 | 0779 | 0780 | 0781 | 0782 | 0783 |
| 310 | 0784 | 0785 | 0786 | 0787 | 0788 | 0789 | 0790 | 0791 | 0792 | 0793 | 0794 | 0795 | 0796 | 0797 | 0798 | 0799 |
| 320 | 0800 | 0301 | 0802 | 0803 | 0804 | 0805 | 0806 | 0807 | 0808 | 0809 | 0810 | 0811 | 0812 | 0813 | 0814 | 0815 |
| 330 | 0816 | 0817 | 0818 | 0819 | 0820 | 0821 | 0822 | 0823 | 0824 | 0825 | 0826 | 0827 | 0828 | 0829 | 0830 | 0831 |
| 340 | 0832 | 0833 | 0834 | 0835 | 0836 | 0837 | 0838 | 0839 | 0840 | 0841 | 0842 | 0843 | 0844 | 0845 | 0846 | 0847 |
| 350 | 0848 | 0849 | 0850 | 0851 | 0852 | 0853 | 0854 | 0855 | 0856 | 0857 | 0858 | 0859 | 0860 | 0861 | 0862 | 0863 |
| 360 | 0864 | 0865 | 0866 | 0867 | 0868 | 0869 | 0870 | 0871 | 0872 | 0873 | 0874 | 0875 | 0876 | 0877 | 0878 | 0879 |
| 370 | 0880 | 0881 | 0882 | 0883 | 0884 | 0885 | 0886 | 0887 | 0888 | 0889 | 0890 | 0891 | 0892 | 0893 | 0894 | 0895 |
| 380 | 0896 | 0897 | 0898 | 0899 | 0900 | 0901 | 0902 | 0903 | 0904 | 0905 | 0906 | 0907 | 0908 | 0909 | 0910 | 0911 |
| 390 | 0212 | 0913 | 0914 | 0915 | 0916 | 0917 | 0918 | 0919 | 0920 | 0921 | 0922 | 0923 | 0924 | 0925 | 0926 | 0927 |
| 3A0 | 0928 | 0929 | 0930 | 0931 | 0932 | 0933 | 0934 | 0935 | 0936 | 0937 | 0938 | 0939 | 0940 | 0941 | 0942 | 0943 |
| 3B0 | 0944 | 0945 | 0946 | 0947 | 0948 | 0949 | 0950 | 0951 | 0952 | 0953 | 0954 | 0955 | 0956 | 0957 | 0958 | 0959 |
| 3C0 | 0960 | 0961 | 0962 | 0963 | 0964 | 0965 | 0966 | 0967 | 0968 | 0969 | 0970 | 0971 | 0972 | 0973 | 0974 | 0975 |
| 3D0 | 0976 | 0977 | 0978 | 0979 | 0980 | 0981 | 0982 | 0983 | 0984 | 0985 | 0986 | 0987 | 0988 | 0989 | 0990 | 0991 |
| 3E0 | 0992 | 0993 | 0994 | 0995 | 0996 | 0997 | 0998 | 0999 | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 |
| 3F0 | 1008 | 1009 | 1010 | 1011 | 1012 | 1013 | 1014 | 1015 | 1016 | 1017 | 1018 | 1019 | 1020 | 1021 | 1022 | 1023 |

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 400 | 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 | 1031 | 1032 | 1033 | 1034 | 1035 | 1036 | 1037 | 1038 | 1039 |
| 410 | 1040 | 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 | 1050 | 1051 | 1052 | 1053 | 1054 | 1055 |
| 420 | 1056 | 1057 | 1058 | 1059 | 1060 | 1061 | 1062 | 1063 | 1064 | 1065 | 1066 | 1067 | 1068 | 1069 | 1070 | 1071 |
| 430 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 | 1081 | 1082 | 1083 | 1084 | 1085 | 1086 | 1087 |
| 440 | 1088 | 1089 | 1090 | 1091 | 1092 | 1093 | 1094 | 1095 | 1096 | 1097 | 1098 | 1099 | 1100 | 1101 | 1102 | 1103 |
| 450 | 1104 | 1105 | 1106 | 1107 | 1108 | 1109 | 1110 | 1111 | 1112 | 1113 | 1114 | 1115 | 1116 | 1117 | 1118 | 1119 |
| 460 | 1120 | 1121 | 1122 | 1123 | 1124 | 1125 | 1126 | 1127 | 1128 | 1129 | 1130 | 1131 | 1132 | 1133 | 1134 | 1135 |
| 470 | 1136 | 1137 | 1138 | 1139 | 1140 | 1141 | 1142 | 1143 | 1144 | 1145 | 1146 | 1147 | 1148 | 1149 | 1150 | 1151 |
| 480 | 1152 | 1153 | 1154 | 1155 | 1156 | 1157 | 1158 | 1159 | 1160 | 1161 | 1162 | 1163 | 1164 | 1165 | 1166 | 1167 |
| 490 | 1168 | 1169 | 1170 | 1171 | 1172 | 1173 | 1174 | 1175 | 1176 | 1177 | 1178 | 1179 | 1180 | 1181 | 1182 | 1183 |
| 4A0 | 1184 | 1185 | 1186 | 1187 | 1188 | 1189 | 1190 | 1191 | 1192 | 1193 | 1194 | 1195 | 1196 | 1197 | 1198 | 1199 |
| 4B0 | 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 | 1209 | 1210 | 1211 | 1212 | 1213 | 1214 | 1215 |
| 4C0 | 1216 | 1217 | 1218 | 1219 | 1220 | 1221 | 1222 | 1223 | 1224 | 1225 | 1226 | 1227 | 1228 | 1229 | 1230 | 1231 |
| 4D0 | 1232 | 1233 | 1234 | 1235 | 1236 | 1237 | 1238 | 1239 | 1240 | 1241 | 1242 | 1243 | 1244 | 1245 | 1246 | 1247 |
| 4E0 | 1248 | 1249 | 1250 | 1251 | 1252 | 1253 | 1254 | 1255 | 1256 | 1257 | 1258 | 1259 | 1260 | 1261 | 1262 | 1263 |
| 4F0 | 1264 | 1265 | 1266 | 1267 | 1268 | 1269 | 1270 | 1271 | 1272 | 1273 | 1274 | 1275 | 1276 | 1277 | 1278 | 1279 |
| 500 | 1280 | 1281 | 1282 | 1283 | 1284 | 1285 | 1286 | 1287 | 1288 | 1289 | 1290 | 1291 | 1292 | 1293 | 1294 | 1295 |
| 510 | 1296 | 1297 | 1298 | 1299 | 1300 | 1301 | 1302 | 1303 | 1304 | 1305 | 1306 | 1307 | 1308 | 1309 | 1310 | 1311 |
| 520 | 1312 | 1313 | 1314 | 1315 | 1316 | 1317 | 1318 | 1319 | 1320 | 1321 | 1322 | 1323 | 1324 | 1325 | 1326 | 1327 |
| 530 | 1328 | 1329 | 1330 | 1331 | 1332 | 1333 | 1334 | 1335 | 1336 | 1337 | 1338 | 1339 | 1340 | 1341 | 1342 | 1343 |
| 540 | 1344 | 1345 | 1346 | 1347 | 1348 | 1349 | 1350 | 1351 | 1352 | 1353 | 1354 | 1355 | 1356 | 1357 | 1358 | 1359 |
| 550 | 1360 | 1361 | 1362 | 1363 | 1364 | 1365 | 1366 | 1367 | 1368 | 1369 | 1370 | 1371 | 1372 | 1373 | 1374 | 1375 |
| 560 | 1376 | 1377 | 1378 | 1379 | 1380 | 1381 | 1382 | 1383 | 1384 | 1385 | 1386 | 1387 | 1388 | 1389 | 1390 | 1391 |
| 570 | 1392 | 1393 | 1394 | 1395 | 1396 | 1397 | 1398 | 1399 | 1400 | 1401 | 1402 | 1403 | 1404 | 1405 | 1406 | 1407 |
| 580 | 1408 | 1409 | 1410 | 1411 | 1412 | 1413 | 1414 | 1415 | 1416 | 1417 | 1418 | 1419 | 1420 | 1421 | 1422 | 1423 |
| 590 | 1424 | 1425 | 1426 | 1427 | 1428 | 1429 | 1430 | 1431 | 1432 | 1433 | 1434 | 1435 | 1436 | 1437 | 1438 | 1439 |
| 5A0 | 1440 | 1441 | 1442 | 1443 | 1444 | 1445 | 1446 | 1447 | 1448 | 1449 | 1450 | 1451 | 1452 | 1453 | 1454 | 1455 |
| 5B0 | 1456 | 1457 | 1458 | 1459 | 1460 | 1461 | 1462 | 1463 | 1464 | 1465 | 1466 | 1467 | 1468 | 1469 | 1470 | 1471 |
| 5C0 | 1472 | 1473 | 1474 | 1475 | 1476 | 1477 | 1478 | 1479 | 1480 | 1481 | 1482 | 1483 | 1484 | 1485 | 1486 | 1487 |
| 5D0 | 1488 | 1489 | 1490 | 1491 | 1492 | 1493 | 1494 | 1495 | 1496 | 1497 | 1498 | 1499 | 1500 | 1501 | 1502 | 1503 |
| 5E0 | 1504 | 1505 | 1506 | 1507 | 1508 | 1509 | 1510 | 1511 | 1512 | 1513 | 1514 | 1515 | 1516 | 1517 | 1518 | 1519 |
| 5F0 | 1520 | 1521 | 1522 | 1523 | 1524 | 1525 | 1526 | 1527 | 1528 | 1529 | 1530 | 1531 | 1532 | 1533 | 1534 | 1535 |
| 600 | 1536 | 1537 | 1538 | 1539 | 1540 | 1541 | 1542 | 1543 | 1544 | 1545 | 1546 | 1547 | 1548 | 1549 | 1550 | 1551 |
| 610 | 1552 | 1553 | 1554 | 1555 | 1556 | 1557 | 1558 | 1559 | 1560 | 1561 | 1562 | 1563 | 1564 | 1565 | 1566 | 1567 |
| 620 | 1568 | 1569 | 1570 | 1571 | 1572 | 1573 | 1574 | 1575 | 1576 | 1577 | 1578 | 1579 | 1580 | 1581 | 1582 | 1583 |
| 630 | 1584 | 1585 | 1586 | 1587 | 1588 | 1589 | 1590 | 1591 | 1592 | 1593 | 1594 | 1595 | 1596 | 1597 | 1598 | 1599 |
| 640 | 1600 | 1601 | 1602 | 1603 | 1604 | 1605 | 1606 | 1607 | 1608 | 1609 | 1610 | 1611 | 1612 | 1613 | 1614 | 1615 |
| 650 | 1616 | 1617 | 1618 | 1619 | 1620 | 1621 | 1622 | 1623 | 1624 | 1625 | 1626 | 1627 | 1628 | 1629 | 1630 | 1631 |
| 660 | 1632 | 1633 | 1634 | 1635 | 1636 | 1637 | 1638 | 1639 | 1640 | 1641 | 1642 | 1643 | 1644 | 1645 | 1646 | 1647 |
| 670 | 1648 | 1649 | 1650 | 1651 | 1652 | 1653 | 1654 | 1655 | 1656 | 1657 | 1658 | 1659 | 1660 | 1661 | 1662 | 1663 |
| 680 | 1664 | 1665 | 1666 | 1667 | 1668 | 1669 | 1670 | 1671 | 1672 | 1673 | 1674 | 1675 | 1676 | 1677 | 1678 | 1679 |
| 690 | 1680 | 1681 | 1682 | 1683 | 1684 | 1685 | 1686 | 1687 | 1688 | 1689 | 1690 | 1691 | 1692 | 1693 | 1694 | 1695 |
| 6A0 | 1696 | 1697 | 1698 | 1699 | 1700 | 1701 | 1702 | 1703 | 1704 | 1705 | 1706 | 1707 | 1708 | 1709 | 1710 | 1711 |
| 6B0 | 1712 | 1713 | 1714 | 1715 | 1716 | 1717 | 1718 | 1719 | 1720 | 1721 | 1722 | 1723 | 1724 | 1725 | 1726 | 1727 |
| 6C0 | 1728 | 1729 | 1730 | 1731 | 1732 | 1733 | 1734 | 1735 | 1736 | 1737 | 1738 | 1739 | 1740 | 1741 | 1742 | 1743 |
| 6D0 | 1744 | 1745 | 1746 | 1747 | 1748 | 1749 | 1750 | 1751 | 1752 | 1753 | 1754 | 1755 | 1756 | 1757 | 1758 | 1759 |
| 6E0 | 1760 | 1761 | 1762 | 1763 | 1764 | 1765 | 1766 | 1767 | 1768 | 1769 | 1770 | 1771 | 1772 | 1773 | 1774 | 1775 |
| 6F0 | 1776 | 1777 | 1778 | 1779 | 1780 | 1781 | 1782 | 1783 | 1784 | 1785 | 1786 | 1787 | 1788 | 1789 | 1790 | 1791 |

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 700 | 1792 | 1793 | 1794 | 1795 | 1796 | 1797 | 1798 | 1799 | 1800 | 1801 | 1802 | 1803 | 1804 | 1805 | 1806 | 1807 |
| 710 | 1808 | 1809 | 1810 | 1811 | 1812 | 1813 | 1814 | 1815 | 1816 | 1817 | 1818 | 1819 | 1820 | 1821 | 1822 | 1823 |
| 720 | 1824 | 1825 | 1826 | 1827 | 1828 | 1829 | 1830 | 1831 | 1832 | 1833 | 1834 | 1835 | 1836 | 1837 | 1838 | 1839 |
| 730 | 1840 | 1841 | 1842 | 1843 | 1844 | 1845 | 1846 | 1847 | 1848 | 1849 | 1850 | 1851 | 1852 | 1853 | 1854 | 1855 |
| 740 | 1856 | 1857 | 1858 | 1859 | 1860 | 1861 | 1862 | 1863 | 1864 | 1865 | 1866 | 1867 | 1868 | 1869 | 1870 | 1871 |
| 750 | 1872 | 1873 | 1874 | 1875 | 1876 | 1877 | 1878 | 1879 | 1880 | 1881 | 1882 | 1883 | 1884 | 1885 | 1886 | 1887 |
| 760 | 1888 | 1889 | 1890 | 1891 | 1892 | 1893 | 1894 | 1895 | 1896 | 1897 | 1898 | 1899 | 1900 | 1901 | 1902 | 1903 |
| 770 | 1904 | 1905 | 1906 | 1907 | 1908 | 1909 | 1910 | 1911 | 1912 | 1913 | 1914 | 1915 | 1916 | 1917 | 1918 | 1919 |
| 780 | 1920 | 1921 | 1922 | 1923 | 1924 | 1925 | 1926 | 1927 | 1928 | 1929 | 1930 | 1931 | 1932 | 1933 | 1934 | 1935 |
| 790 | 1936 | 1937 | 1938 | 1939 | 1940 | 1941 | 1942 | 1943 | 1944 | 1945 | 1946 | 1947 | 1948 | 1949 | 1950 | 1951 |
| 7A0 | 1952 | 1953 | 1954 | 1955 | 1956 | 1957 | 1958 | 1959 | 1960 | 1961 | 1962 | 1963 | 1964 | 1965 | 1966 | 1967 |
| 7B0 | 1968 | 1969 | 1970 | 1971 | 1972 | 1973 | 1974 | 1975 | 1976 | 1977 | 1978 | 1979 | 1980 | 1981 | 1982 | 1983 |
| 7C0 | 1984 | 1985 | 1986 | 1987 | 1988 | 1989 | 1990 | 1991 | 1992 | 1993 | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 |
| 7D0 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2009 | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
| 7E0 | 2016 | 2017 | 2018 | 2019 | 2020 | 2021 | 2022 | 2023 | 2024 | 2025 | 2026 | 2027 | 2028 | 2029 | 2030 | 2031 |
| 7F0 | 2032 | 2033 | 2034 | 2035 | 2036 | 2037 | 2038 | 2039 | 2040 | 2041 | 2042 | 2043 | 2044 | 2045 | 2046 | 2047 |
| 800 | 2048 | 2049 | 2050 | 2051 | 2052 | 2053 | 2054 | 2055 | 2056 | 2057 | 2058 | 2059 | 2060 | 2061 | 2062 | 2063 |
| 810 | 2064 | 2065 | 2066 | 2067 | 2068 | 2069 | 2070 | 2071 | 2072 | 2073 | 2074 | 2075 | 2076 | 2077 | 2078 | 2079 |
| 820 | 2080 | 2081 | 2082 | 2083 | 2084 | 2085 | 2086 | 2087 | 2088 | 2089 | 2090 | 2091 | 2092 | 2093 | 2094 | 2095 |
| 830 | 2096 | 2097 | 2098 | 2099 | 2100 | 2101 | 2102 | 2103 | 2104 | 2105 | 2106 | 2107 | 2108 | 2109 | 2110 | 2111 |
| 840 | 2112 | 2113 | 2114 | 2115 | 2116 | 2117 | 2118 | 2119 | 2120 | 2121 | 2122 | 2123 | 2124 | 2125 | 2126 | 2127 |
| 850 | 2128 | 2129 | 2130 | 2131 | 2132 | 2133 | 2134 | 2135 | 2136 | 2137 | 2138 | 2139 | 2140 | 2141 | 2142 | 2143 |
| 860 | 2144 | 2145 | 2146 | 2147 | 2148 | 2149 | 2150 | 2151 | 2152 | 2153 | 2154 | 2155 | 2156 | 2157 | 2158 | 2159 |
| 870 | 2160 | 2161 | 2162 | 2163 | 2164 | 2165 | 2166 | 2167 | 2168 | 2169 | 2170 | 2171 | 2172 | 2173 | 2174 | 2175 |
| 880 | 2176 | 2177 | 2178 | 2179 | 2180 | 2181 | 2182 | 2183 | 2184 | 2185 | 2186 | 2187 | 2188 | 2189 | 2190 | 2191 |
| 890 | 2192 | 2193 | 2194 | 2195 | 2196 | 2197 | 2198 | 2199 | 2200 | 2201 | 2202 | 2203 | 2204 | 2205 | 2206 | 2207 |
| 8A0 | 2208 | 2209 | 2210 | 2211 | 2212 | 2213 | 2214 | 2215 | 2216 | 2217 | 2218 | 2219 | 2220 | 2221 | 2222 | 2223 |
| 8B0 | 2224 | 2225 | 2226 | 2227 | 2228 | 2229 | 2230 | 2231 | 2232 | 2233 | 2234 | 2235 | 2236 | 2237 | 2238 | 2239 |
| 8C0 | 2240 | 2241 | 2242 | 2243 | 2244 | 2245 | 2246 | 2247 | 2248 | 2249 | 2250 | 2251 | 2252 | 2253 | 2254 | 2255 |
| 8D0 | 2256 | 2257 | 2258 | 2259 | 2260 | 2261 | 2262 | 2263 | 2264 | 2265 | 2266 | 2267 | 2268 | 2269 | 2270 | 2271 |
| 8E0 | 2272 | 2273 | 2274 | 2275 | 2276 | 2277 | 2278 | 2279 | 2280 | 2281 | 2282 | 2283 | 2284 | 2285 | 2286 | 2287 |
| 8F0 | 2288 | 2289 | 2290 | 2291 | 2292 | 2293 | 2294 | 2295 | 2296 | 2297 | 2298 | 2299 | 2300 | 2301 | 2302 | 2303 |
| 900 | 2304 | 2305 | 2306 | 2307 | 2308 | 2309 | 2310 | 2311 | 2312 | 2313 | 2314 | 2315 | 2316 | 2317 | 2318 | 2319 |
| 910 | 2320 | 2321 | 2322 | 2323 | 2324 | 2325 | 2326 | 2327 | 2328 | 2329 | 2330 | 2331 | 2332 | 2333 | 2334 | 2335 |
| 920 | 2336 | 2337 | 2338 | 2339 | 2340 | 2341 | 2342 | 2343 | 2344 | 2345 | 2346 | 2347 | 2348 | 2349 | 2350 | 2351 |
| 930 | 2352 | 2353 | 2354 | 2355 | 2356 | 2357 | 2358 | 2359 | 2360 | 2361 | 2362 | 2363 | 2364 | 2365 | 2366 | 2367 |
| 940 | 2368 | 2369 | 2370 | 2371 | 2372 | 2373 | 2374 | 2375 | 2376 | 2377 | 2378 | 2379 | 2380 | 2381 | 2382 | 2383 |
| 950 | 2384 | 2385 | 2386 | 2387 | 2388 | 2389 | 2390 | 2391 | 2392 | 2393 | 2394 | 2395 | 2396 | 2397 | 2398 | 2399 |
| 960 | 2400 | 2401 | 2402 | 2403 | 2404 | 2405 | 2406 | 2407 | 2408 | 2409 | 2410 | 2411 | 2412 | 2413 | 2414 | 2415 |
| 970 | 2416 | 2417 | 2418 | 2419 | 2420 | 2421 | 2422 | 2423 | 2424 | 2425 | 2426 | 2427 | 2428 | 2429 | 2430 | 2431 |
| 980 | 2432 | 2433 | 2434 | 2435 | 2436 | 2437 | 2438 | 2439 | 2440 | 2441 | 2442 | 2443 | 2444 | 2445 | 2446 | 2447 |
| 990 | 2448 | 2449 | 2450 | 2451 | 2452 | 2453 | 2454 | 2455 | 2456 | 2457 | 2458 | 2459 | 2460 | 2461 | 2462 | 2463 |
| 9A0 | 2464 | 2465 | 2466 | 2467 | 2468 | 2469 | 2470 | 2471 | 2472 | 2473 | 2474 | 2475 | 2476 | 2477 | 2478 | 2479 |
| 9B0 | 2480 | 2481 | 2482 | 2483 | 2484 | 2485 | 2486 | 2487 | 2488 | 2489 | 2490 | 2491 | 2492 | 2493 | 2494 | 2495 |
| 9C0 | 2496 | 2497 | 2498 | 2499 | 2500 | 2501 | 2502 | 2503 | 2504 | 2505 | 2506 | 2507 | 2508 | 2509 | 2510 | 2511 |
| 9D0 | 2512 | 2513 | 2514 | 2515 | 2516 | 2517 | 2518 | 2519 | 2520 | 2521 | 2522 | 2523 | 2524 | 2525 | 2526 | 2527 |
| 9E0 | 2528 | 2529 | 2530 | 2531 | 2532 | 2533 | 2534 | 2535 | 2536 | 2537 | 2538 | 2539 | 2540 | 2541 | 2542 | 2543 |
| 9F0 | 2544 | 2545 | 2546 | 2547 | 2548 | 2549 | 2550 | 2551 | 2552 | 2553 | 2554 | 2555 | 2556 | 2557 | 2558 | 2559 |

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A00 | 2560 | 2561 | 2562 | 2563 | 2564 | 2565 | 2566 | 2567 | 2568 | 2569 | 2570 | 2571 | 2572 | 2573 | 2574 | 2575 |
| A10 | 2576 | 2577 | 2578 | 2579 | 2580 | 2581 | 2582 | 2583 | 2584 | 2585 | 2586 | 2587 | 2588 | 2589 | 2590 | 2591 |
| A20 | 2592 | 2593 | 2594 | 2595 | 2596 | 2597 | 2598 | 2599 | 2600 | 2601 | 2602 | 2603 | 2604 | 2605 | 2606 | 2607 |
| A30 | 2608 | 2609 | 2610 | 2611 | 2612 | 2613 | 2614 | 2615 | 2616 | 2617 | 2618 | 2619 | 2620 | 2621 | 2622 | 2623 |
| A40 | 2624 | 2625 | 2626 | 2627 | 2628 | 2629 | 2630 | 2631 | 2632 | 2633 | 2634 | 2635 | 2636 | 2637 | 2638 | 2639 |
| A50 | 2640 | 2641 | 2642 | 2643 | 2644 | 2645 | 2646 | 2647 | 2648 | 2649 | 2650 | 2651 | 2652 | 2653 | 2654 | 2655 |
| A60 | 2656 | 2657 | 2658 | 2659 | 2660 | 2661 | 2662 | 2663 | 2664 | 2665 | 2666 | 2667 | 2668 | 2669 | 2670 | 2671 |
| A70 | 2672 | 2673 | 2674 | 2675 | 2676 | 2677 | 2678 | 2679 | 2680 | 2681 | 2682 | 2683 | 2684 | 2685 | 2686 | 2687 |
| A80 | 2688 | 2689 | 2690 | 2691 | 2692 | 2693 | 2694 | 2695 | 2696 | 2697 | 2698 | 2699 | 2700 | 2701 | 2702 | 2703 |
| A90 | 2704 | 2705 | 2706 | 2707 | 2708 | 2709 | 2710 | 2711 | 2712 | 2713 | 2714 | 2715 | 2716 | 2717 | 2718 | 2719 |
| AA0 | 2720 | 2721 | 2722 | 2723 | 2724 | 2725 | 2726 | 2727 | 2728 | 2729 | 2730 | 2731 | 2732 | 2733 | 2734 | 2735 |
| AB0 | 2736 | 2737 | 2738 | 2739 | 2740 | 2741 | 2742 | 2743 | 2744 | 2745 | 2746 | 2747 | 2748 | 2749 | 2750 | 2751 |
| AC0 | 2752 | 2753 | 2754 | 2755 | 2756 | 2757 | 2758 | 2759 | 2760 | 4761 | 2762 | 2763 | 2764 | 2765 | 2766 | 2767 |
| AD0 | 2768 | 2769 | 2770 | 2771 | 2772 | 2773 | 2774 | 2775 | 2776 | 2777 | 2778 | 2779 | 2780 | 2781 | 2782 | 2783 |
| AE0 | 2784 | 2785 | 2786 | 2787 | 2788 | 2789 | 2790 | 2791 | 2792 | 2793 | 2794 | 2795 | 2796 | 2797 | 2798 | 2799 |
| AF0 | 2800 | 2801 | 2802 | 2803 | 2804 | 2805 | 2806 | 2807 | 2808 | 2809 | 2810 | 2811 | 2812 | 2813 | 2814 | 2815 |
| B00 | 2816 | 2817 | 2818 | 2819 | 2820 | 2821 | 2822 | 2823 | 2824 | 2825 | 2826 | 2827 | 2828 | 2829 | 2830 | 2831 |
| B10 | 2832 | 2833 | 2834 | 2835 | 2836 | 2837 | 2838 | 2839 | 2840 | 2841 | 2842 | 2843 | 2844 | 2845 | 2846 | 2847 |
| B20 | 2848 | 2849 | 2850 | 3851 | 2852 | 2853 | 2854 | 2855 | 2856 | 2857 | 2858 | 2859 | 2860 | 2861 | 2862 | 2863 |
| B30 | 2864 | 2865 | 2866 | 2867 | 2868 | 2869 | 2870 | 2871 | 2872 | 2873 | 2874 | 2875 | 2876 | 2877 | 2878 | 2879 |
| B40 | 2880 | 2881 | 2882 | 2883 | 2884 | 2885 | 2866 | 2887 | 2888 | 2889 | 2890 | 2891 | 2892 | 2893 | 2894 | 2895 |
| B50 | 2896 | 2897 | 2898 | 2899 | 2900 | 2901 | 2902 | 2903 | 2904 | 2905 | 2906 | 2907 | 2908 | 2909 | 2910 | 2911 |
| B60 | 2912 | 2913 | 2914 | 2915 | 2916 | 2917 | 2918 | 2919 | 2920 | 2921 | 2922 | 2923 | 2924 | 2925 | 2926 | 2927 |
| B70 | 2928 | 2929 | 2930 | 2931 | 2932 | 2933 | 2934 | 2935 | 2936 | 2937 | 2938 | 2939 | 2940 | 2941 | 2942 | 2943 |
| B80 | 2944 | 2945 | 2946 | 2947 | 2948 | 2949 | 2950 | 2951 | 2952 | 2953 | 2954 | 2955 | 2956 | 2957 | 2958 | 2959 |
| B90 | 2960 | 2961 | 2962 | 2963 | 2964 | 2965 | 2966 | 2967 | 2968 | 2969 | 2970 | 2971 | 2972 | 2973 | 2974 | 2975 |
| BA0 | 2976 | 2977 | 2978 | 2979 | 2980 | 2981 | 2982 | 2983 | 2984 | 2985 | 2986 | 2987 | 2988 | 2989 | 2990 | 2991 |
| BB0 | 2992 | 2993 | 2994 | 2995 | 2996 | 2997 | 2998 | 2999 | 3000 | 3001 | 3002 | 3003 | 3004 | 3005 | 3006 | 3007 |
| BC0 | 3008 | 3009 | 3010 | 3011 | 3012 | 3013 | 3014 | 3015 | 3016 | 3017 | 3018 | 3019 | 3020 | 3021 | 3022 | 3023 |
| BD0 | 3024 | 3025 | 3026 | 3027 | 3028 | 3029 | 3030 | 3031 | 3032 | 3033 | 3034 | 3035 | 3036 | 3037 | 3038 | 3039 |
| BE0 | 3040 | 3041 | 3042 | 3043 | 3044 | 3045 | 3046 | 3047 | 3048 | 3049 | 3050 | 3051 | 3052 | 3053 | 3054 | 3055 |
| BF0 | 3056 | 3057 | 3058 | 3059 | 3060 | 3061 | 3062 | 3063 | 3064 | 3065 | 3066 | 3067 | 3068 | 3069 | 3070 | 3071 |
| C00 | 3072 | 3073 | 3074 | 3075 | 3076 | 3077 | 3078 | 3079 | 3080 | 3081 | 3082 | 3083 | 3084 | 3085 | 3086 | 3087 |
| C10 | 3088 | 3089 | 3090 | 3091 | 3092 | 3093 | 3094 | 3095 | 3096 | 3097 | 3098 | 3099 | 3100 | 3101 | 3102 | 3103 |
| C20 | 3104 | 3105 | 3106 | 3107 | 3108 | 3109 | 3110 | 3111 | 3112 | 3113 | 3114 | 3115 | 3116 | 3117 | 3118 | 3119 |
| C30 | 3120 | 3121 | 3122 | 3123 | 3124 | 3125 | 3126 | 3127 | 3128 | 3129 | 3130 | 3131 | 3132 | 3133 | 3134 | 3135 |
| C40 | 3136 | 3137 | 3138 | 3139 | 3140 | 3141 | 3142 | 3143 | 3144 | 3145 | 3146 | 3147 | 3148 | 3149 | 3150 | 3151 |
| C50 | 3152 | 3153 | 3154 | 3155 | 3156 | 3157 | 3158 | 3159 | 3160 | 3161 | 3162 | 3163 | 3164 | 3165 | 3166 | 3167 |
| C60 | 3168 | 3169 | 3170 | 3171 | 3172 | 3173 | 3174 | 3175 | 3176 | 3177 | 3178 | 3179 | 3180 | 3181 | 3182 | 3183 |
| C70 | 3184 | 3185 | 3186 | 3187 | 3188 | 3189 | 3190 | 3191 | 3192 | 3193 | 3194 | 3195 | 3196 | 3197 | 3198 | 3199 |
| C80 | 3200 | 3201 | 3202 | 3203 | 3204 | 3205 | 3206 | 3207 | 3208 | 3209 | 3210 | 3211 | 3212 | 3213 | 3214 | 3215 |
| C90 | 3216 | 3217 | 3218 | 3219 | 3220 | 3221 | 3222 | 3223 | 3224 | 3225 | 3226 | 3227 | 3228 | 3229 | 3230 | 3231 |
| CA0 | 3232 | 3233 | 3234 | 3235 | 3236 | 3237 | 3238 | 3239 | 3240 | 3241 | 3242 | 3243 | 3244 | 3245 | 3246 | 3247 |
| CB0 | 3248 | 3249 | 3250 | 3251 | 3252 | 3253 | 3254 | 3255 | 3256 | 3257 | 3258 | 3259 | 3260 | 3261 | 3262 | 3263 |
| CC0 | 3264 | 3265 | 3266 | 3267 | 3268 | 3269 | 3270 | 3271 | 3272 | 3273 | 3274 | 3275 | 3276 | 3277 | 3278 | 3279 |
| CD0 | 3280 | 3281 | 3282 | 3283 | 3284 | 3285 | 3286 | 3287 | 3288 | 3289 | 3290 | 3291 | 3292 | 3293 | 3294 | 3295 |
| CE0 | 3296 | 3297 | 3298 | 3299 | 3300 | 3301 | 3302 | 3303 | 3304 | 3305 | 3306 | 3307 | 3308 | 3309 | 3310 | 3311 |
| CF0 | 3312 | 3313 | 3314 | 3315 | 3316 | 3317 | 3318 | 3319 | 3320 | 3321 | 3322 | 3323 | 3324 | 3325 | 3326 | 3327 |

## HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D00 | 3328 | 3329 | 3330 | 3331 | 3332 | 3333 | 3334 | 3335 | 3336 | 3337 | 3338 | 3339 | 3340 | 3341 | 3342 | 3343 |
| D10 | 3344 | 3345 | 3346 | 3347 | 3348 | 3349 | 3350 | 3351 | 3352 | 3353 | 3354 | 3355 | 3356 | 3357 | 3358 | 3359 |
| D20 | 3360 | 3361 | 3362 | 3363 | 3364 | 3365 | 3366 | 3367 | 3368 | 3369 | 3370 | 3371 | 3372 | 3373 | 3374 | 3375 |
| D30 | 3376 | 3377 | 3378 | 3379 | 3380 | 3381 | 3382 | 3383 | 3384 | 3385 | 3386 | 3387 | 3388 | 3389 | 3390 | 3391 |
| D40 | 3392 | 3393 | 3394 | 3395 | 3396 | 3397 | 3398 | 3399 | 3400 | 3401 | 3402 | 3403 | 3404 | 3405 | 3406 | 3407 |
| D50 | 3408 | 3409 | 3410 | 3411 | 3412 | 3413 | 3414 | 3415 | 3416 | 3417 | 3418 | 3419 | 3420 | 3421 | 3422 | 3423 |
| D60 | 3424 | 3425 | 3426 | 3427 | 3428 | 3429 | 3430 | 3431 | 3432 | 3433 | 3434 | 3435 | 3436 | 3437 | 3438 | 3439 |
| D70 | 3440 | 3441 | 3442 | 3443 | 3444 | 3445 | 3446 | 3447 | 3448 | 3449 | 3450 | 3451 | 3452 | 3453 | 3454 | 3455 |
| D80 | 3456 | 3457 | 3458 | 3459 | 3460 | 3461 | 3462 | 3463 | 3464 | 3465 | 3466 | 3467 | 3468 | 3469 | 3470 | 3471 |
| D90 | 3472 | 3473 | 3474 | 3475 | 3476 | 3477 | 3478 | 3479 | 3480 | 3481 | 3482 | 3483 | 3484 | 3485 | 3486 | 3487 |
| DA0 | 3488 | 3489 | 3490 | 3491 | 3492 | 3493 | 3494 | 3495 | 3496 | 3497 | 3498 | 3499 | 3500 | 3501 | 3502 | 3503 |
| DB0 | 3504 | 3505 | 3506 | 3507 | 3508 | 3509 | 3510 | 3511 | 3512 | 3513 | 3514 | 3515 | 3516 | 3517 | 3518 | 3519 |
| DC0 | 3520 | 3521 | 3522 | 3523 | 3524 | 3525 | 3526 | 3527 | 3528 | 3529 | 3530 | 3531 | 3532 | 3533 | 3534 | 3535 |
| DD0 | 3536 | 3537 | 3538 | 3539 | 3540 | 3541 | 3542 | 3543 | 3544 | 3545 | 3546 | 3547 | 3548 | 3549 | 3550 | 3551 |
| DE0 | 3552 | 3553 | 3554 | 3555 | 3556 | 3557 | 3558 | 3559 | 3560 | 3561 | 3562 | 3563 | 3564 | 3565 | 3566 | 3567 |
| DF0 | 3568 | 3569 | 3570 | 3571 | 3572 | 3573 | 3574 | 3575 | 3576 | 3577 | 3578 | 3579 | 3580 | 3581 | 3582 | 3583 |
| E00 | 3584 | 3585 | 3586 | 3587 | 3588 | 3589 | 3590 | 3591 | 3592 | 3593 | 3594 | 3595 | 3596 | 3597 | 3598 | 3599 |
| E10 | 3600 | 3601 | 3602 | 3603 | 3604 | 3605 | 3606 | 3607 | 3608 | 3609 | 3610 | 3611 | 3612 | 3613 | 3614 | 3615 |
| E20 | 3616 | 3617 | 3618 | 3619 | 3620 | 3621 | 3622 | 3623 | 3624 | 3625 | 3626 | 3627 | 3628 | 3629 | 3630 | 3631 |
| E30 | 3632 | 3633 | 3634 | 3635 | 3636 | 3637 | 3638 | 3639 | 3640 | 3641 | 3642 | 3643 | 3644 | 3645 | 3646 | 3647 |
| E40 | 3648 | 3649 | 3650 | 3651 | 3652 | 3653 | 3654 | 3655 | 3656 | 3657 | 3658 | 3659 | 3660 | 3661 | 3662 | 3663 |
| E50 | 3664 | 3665 | 3666 | 3667 | 3668 | 3669 | 3670 | 3671 | 3672 | 3673 | 3674 | 3675 | 3676 | 3677 | 3678 | 3679 |
| E60 | 3680 | 3681 | 3682 | 3683 | 3684 | 3685 | 3686 | 3687 | 3688 | 3689 | 3690 | 3691 | 3692 | 3693 | 3694 | 3695 |
| E70 | 3696 | 3697 | 3698 | 3699 | 3700 | 3701 | 3702 | 3703 | 3704 | 3705 | 3706 | 3707 | 3708 | 3709 | 3710 | 3711 |
| E80 | 3712 | 3713 | 3714 | 3715 | 3716 | 3717 | 3718 | 3719 | 3720 | 3721 | 3722 | 3723 | 3724 | 3725 | 3726 | 3727 |
| E90 | 3728 | 3729 | 3730 | 3731 | 3732 | 3733 | 3734 | 3735 | 3736 | 3737 | 3738 | 3739 | 3740 | 3741 | 3742 | 3743 |
| EA0 | 3744 | 3745 | 3746 | 3747 | 3748 | 3749 | 3750 | 3751 | 3752 | 3753 | 3754 | 3755 | 3756 | 3757 | 3758 | 3759 |
| EB0 | 3760 | 3761 | 3762 | 3763 | 3764 | 3765 | 3766 | 3767 | 3768 | 3769 | 3770 | 3771 | 3772 | 3773 | 3774 | 3775 |
| EC0 | 3776 | 3777 | 3778 | 3779 | 3780 | 3781 | 3782 | 3783 | 3784 | 3785 | 3786 | 3787 | 3788 | 3789 | 3790 | 3791 |
| ED0 | 3792 | 3793 | 3794 | 3795 | 3796 | 3797 | 3798 | 3799 | 3800 | 3801 | 3802 | 3803 | 3804 | 3805 | 3806 | 3807 |
| EE0 | 3808 | 3809 | 3810 | 3811 | 3812 | 3813 | 3814 | 3815 | 3816 | 3817 | 3818 | 3819 | 3820 | 3821 | 3822 | 3823 |
| EF0 | 3824 | 3825 | 3826 | 3827 | 3828 | 3829 | 3830 | 3831 | 3832 | 3833 | 3834 | 3835 | 3836 | 3837 | 3838 | 3839 |
| F00 | 3840 | 3841 | 3842 | 3843 | 3844 | 3845 | 3846 | 3847 | 3848 | 3849 | 3850 | 3851 | 3852 | 3853 | 3854 | 3855 |
| F10 | 3856 | 3857 | 3858 | 3859 | 3860 | 3861 | 3862 | 3863 | 3864 | 3865 | 3866 | 3867 | 3868 | 3869 | 3870 | 3871 |
| F20 | 3872 | 3873 | 3874 | 3875 | 3876 | 3877 | 3878 | 3879 | 3880 | 3881 | 3882 | 3883 | 3884 | 3885 | 3886 | 3887 |
| F30 | 3888 | 3889 | 3890 | 3891 | 3892 | 3893 | 3894 | 3895 | 3896 | 3897 | 3898 | 3899 | 3900 | 3901 | 3902 | 3903 |
| F40 | 3904 | 3905 | 3906 | 3907 | 3908 | 3909 | 3910 | 3911 | 3912 | 3913 | 3914 | 3915 | 3916 | 3917 | 3918 | 3919 |
| F50 | 3920 | 3921 | 3922 | 3923 | 3924 | 3925 | 3926 | 3927 | 3928 | 3929 | 3930 | 3931 | 3932 | 3933 | 3934 | 3935 |
| F60 | 3936 | 3937 | 3938 | 3939 | 3940 | 3941 | 3942 | 3943 | 3944 | 3945 | 3946 | 3947 | 3948 | 3949 | 3950 | 3951 |
| F70 | 3952 | 3953 | 3954 | 3955 | 3956 | 3957 | 3958 | 3959 | 3960 | 3961 | 3962 | 3963 | 3964 | 3965 | 3966 | 3967 |
| F80 | 3968 | 3969 | 3970 | 3971 | 3972 | 3973 | 3974 | 3975 | 3976 | 3977 | 3978 | 3979 | 3980 | 3981 | 3982 | 3983 |
| F90 | 3984 | 3985 | 3986 | 3987 | 3988 | 3989 | 3990 | 3991 | 3992 | 3993 | 3994 | 3995 | 3996 | 3997 | 3998 | 3999 |
| FA0 | 4000 | 4001 | 4002 | 4003 | 4004 | 4005 | 4006 | 4007 | 4008 | 4009 | 4010 | 4011 | 4012 | 4013 | 4014 | 4015 |
| FB0 | 4016 | 4017 | 4018 | 4019 | 4020 | 4021 | 4022 | 4023 | 4024 | 4025 | 4026 | 4027 | 4028 | 4029 | 4030 | 4031 |
| FC0 | 4032 | 4033 | 4034 | 4035 | 4036 | 4037 | 4038 | 4039 | 4040 | 4041 | 4042 | 4043 | 4044 | 4045 | 4046 | 4047 |
| FD0 | 4048 | 4049 | 4050 | 4051 | 4052 | 4053 | 4054 | 4055 | 4056 | 4057 | 4058 | 4059 | 4060 | 4061 | 4062 | 4063 |
| FE0 | 4064 | 4065 | 4066 | 4067 | 4068 | 4069 | 4070 | 4071 | 4072 | 4073 | 4074 | 4075 | 4076 | 4077 | 4078 | 4079 |
| FF0 | 4080 | 4081 | 4082 | 4083 | 4084 | 4085 | 4086 | 4087 | 4088 | 4089 | 4090 | 4091 | 4092 | 4093 | 4094 | 4095 |

# APPENDIX J
# ERROR MESSAGES

When the assembler is unable to correctly assemble a source file, it generates an error message describing the trouble. If possible, it will continue execution. In some cases the assembler is unable to continue (e.g., too many symbols in a program), and it must abort execution. If your program should generate an error message, make the necessary corrections and reassemble. The object file will probably not be executable, and, if the error caused an abort, the list file may also be unreadable.

The general format for all errors listed in your code is shown below:

*** ERROR #*eee*, LINE #*lll* (*ppp*), *Message*

where:

| | |
|---|---|
| eee | is the error number |
| lll | is the line causing the error |
| ppp | is the line causing the lost error |

Message is the error message

(See Chapter 6 for a complete description of all error messages generated by the assembler.)

# Source File Error Messages

This type of error is caused by syntactic errors in your source code. They appear in your listing file immediately following the source line that caused the error.

In attempting to further define the error, ASM51 may generate more than one message for a single error. Since the assembler attempts to continue processing your code a single error may have side effects that cause subsequent errors.

A list of all Assembler Error messages is shown below:

### Assembler Error Messages

1 SYNTAX ERROR
2 SOURCE LISTING TERMINATED AT 255 CHARACTERS
3 ARITHMETIC OVERFLOW IN NUMERIC CONSTANT
4 ATTEMPT TO DIVIDE BY ZERO
5 EXPRESSION WITH FORWARD REFERENCE NOT ALLOWED
6 TYPE OF SET SYMBOL DOES NOT ALLOW REDEFINITION
7 EQU SYMBOL ALREADY DEFINED
8 ATTEMPT TO ADDRESS NON-BIT-ADDRESSABLE BIT
9 BAD BIT OFFSET IN BIT ADDRESS EXPRESSION
10 TEXT FOUND BEYOND END STATEMENT—IGNORED
11 PREMATURE END OF FILE (NO END STATEMENT)
12 ILLEGAL CHARACTER IN NUMERIC CONSTANT
13 ILLEGAL USE OF REGISTER NAME IN EXPRESSION
14 SYMBOL IN LABEL FIELD ALREADY DEFINED
15 ILLEGAL CHARACTER
16 MORE ERRORS DETECTED, NOT REPORTED
17 ARITHMETIC OVERFLOW IN LOCATION COUNTER
18 UNDEFINED SYMBOL
19 VALUE WILL NOT FIT INTO A BYTE
20 OPERATION INVALID IN THIS SEGMENT
21 STRING TERMINATED BY END-OF-LINE
22 STRING LONGER THAN 2 CHARACTERS NOT ALLOWED IN THIS CONTEXT
23 STRING, NUMBER, OR IDENTIFIER CANNOT EXCEED 225 CHARACTERS
24 DESTINATION ADDRESS OUT OF RANGE FOR AJMP
25 DESTINATION ADDRESS OUT OF RANGE FOR SJMP
26 DESTINATION ADDRESS OUT OF RANGE FOR ACALL
27 DESTINATION ADDRESS OUT OF RANGE FOR JC
28 DESTINATION ADDRESS OUT OF RANGE FOR JNC
29 DESTINATION ADDRESS OUT OF RANGE FOR JZ
30 DESTINATION ADDRESS OUT OF RANGE FOR JNZ
31 DESTINATION ADDRESS OUT OF RANGE FOR DJNZ
32 DESTINATION ADDRESS OUT OF RANGE FOR CJNE
33 DESTINATION ADDRESS OUT OF RANGE FOR JB
34 DESTINATION ADDRESS OUT OF RANGE FOR JBC
35 DESTINATION ADDRESS OUT OF RANGE FOR JNB
36 CODE SEGMENT ADDRESS EXPECTED
37 DATA SEGMENT ADDRESS EXPECTED
38 XDATA SEGMENT ADDRESS EXPECTED
39 BIT SEGMENT ADDRESS EXPECTED
40 BYTE OF BIT ADDRESS NOT IN DATA SEGMENT

# Macro Error Messages

Macro errors are caused by errors using the Macro Processing Language (MPL). They are listed immediately following the line in which the error was recognized, and is followed by a trace of the macro call/expression stack. This is not necessarily the line that contains the error.

Since the Macro Processor attempts to define the error completely, several messages may be generated. A macro error may be responsible for subsequent macro errors and assembler errors.

All of the Macro Error messages are listed below:

**Macro Error Messages**

300 MORE ERRORS DETECTED, NOT REPORTED
301 UNDEFINED MACRO NAME
302 ILLEGAL EXIT MACRO
303 FATAL SYSTEM ERROR
304 ILLEGAL EXPRESSION
305 MISSING "FI" IN "IF"
306 MISSING "THEN" IN "IF"
307 ILLEGAL ATTEMPT TO REDEFINE MACRO
308 MISSING IDENTIFIER IN DEFINE PATTERN
309 MISSING BALANCED STRING
310 MISSING LIST ITEM
311 MISSING DELIMITER
312 PREMATURE EOF
313 DYNAMIC STORAGE (MACROS OR ARGUMENTS) OVERFLOW
314 MACRO STACK OVERFLOW
315 INPUT STACK OVERFLOW
317 PATTERN TOO LONG
318 ILLEGAL METACHARACTER: <char>
319 UNBALANCED ")" IN ARGUMENT TO USER DEFINED MACRO
320 ILLEGAL ASCENDING CALL

# Control Error Messages

Control errors are announced when something is wrong with the invocation line or a control line in the source file. In general, command language errors are fatal, causing ASM51 to abort assembly. However, the errors listed below are not considered fatal.

**Control Error Messages**

400 MORE ERRORS DETECTED NOT REPORTED
401 BAD PARAMETER TO CONTROL
402 MORE THAN ONE INCLUDE CONTROL ON A SINGLE LINE
403 ILLEGAL CHARACTER IN COMMAND
406 TOO MANY WORKFILES—ONLY FIRST TWO USED
407 UNRECOGNIZED CONTROL OR MISPLACED PRIMARY CONTROL: <control>
408 NO TITLE FOR TITLE CONTROL
409 NO PARAMETER ALLOWED WITH ABOVE CONTROL
410 SAVE STACK OVERFLOW
411 SAVE STACK UNDERFLOW
413 PAGEWIDTH BELOW MINIMUM, SET TO 72
414 PAGELENGTH BELOW MINIMUM, SET TO 10
415 PAGEWIDTH ABOVE MAXIMUM, SET TO 132

# Special Assembler Error Messages

These error messages are displayed on the console. They are displayed immediately before the assembler aborts operation. You should never receive one of these errors; if you should encounter this type of error notify Intel Corporation via the Software Problem Report included with this manual. The content of all output files will be undefined. A list of all of the special assembler error messages is shown below:

**Special Assembler Error Messages**

800 UNRECOGNIZED ERROR MESSAGE NUMBER
801 SOURCE FILE READING UNSYNCHRONIZED
802 INTERMEDIATE FILE READING UNSYNCHRONIZED
803 BAD OPERAND STACK POP REQUEST
804 PARSE STACK UNDERFLOW
805 INVALID EXPRESSION STACK CONFIGURATION

## Fatal Error Messages

This type of error causes the assembler to abort execution. All output files will be unusable. This type of error is usually the result of exceptionally large or complex source files.

These errors are printed at the console just before ASM51 terminates operation. To overcome this type of error, divide your source file into smaller files and assemble them separately. If necessary reduce the number of symbols used in your program, or the complexity of the expressions you use. Then rejoin files for final assembly. A list of all fatal error messages is shown below:

900 USER SYMBOL TABLE SPACE EXHAUSTED
901 PARSE STACK OVERFLOW
902 EXPRESSION STACK OVERFLOW
903 INTERMEDIATE FILE BUFFER OVERFLOW
904 USER NAME TABLE SPACE EXHAUSTED

# INDEX

# REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

**WE'D LIKE YOUR COMMENTS ...**

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

**BUSINESS REPLY MAIL**

**FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA**

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.