MCS-51™ MACRO ASSEMBLER
USER’S GUIDE

Manual Order Number 9800937-01
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.


Before you program one of the MCS-51 microcomputers, you should read the MCS-51 User's Manual, Order Number 121517.
CHAPTER 6
ASSEMBLER OPERATION
AND CONTROLS
How to Invoke the MCS-51 Macro Assembler ....... 6-1
Assembler Controls. 6-2

CHAPTER 7
ASSEMBLER OUTPUT: ERROR
MESSAGES AND LISTING FILE FORMAT
Error Messages and Recovery ..................... 7-1
Console Error Messages 7-1
I/O Errors .................................. 7-1
ASM51 Internal Errors. 7-2
ASM51 Fatal Errors 7-2
Listing File Error Messages 7-3
Source File Error Messages 7-4
Macro Error Messages 7-8
Control Error Messages 7-11
Special Assembler Error Messages 7-13
Fatal Error Messages 7-13
Assembler Listing File Format 7-14
List File Heading 7-17
Source Listing 7-17
Format for Macros and INCLUDE Files 7-18
Symbol Table 7-19

APPENDIX A
ASSEMBLY LANGUAGE
BNF GRAMMAR

APPENDIX B
INSTRUCTION SET SUMMARY

APPENDIX C
ASSEMBLER DIRECTIVE SUMMARY

APPENDIX D
ASSEMBLER CONTROL SUMMARY

APPENDIX E
MACRO PROCESSOR LANGUAGE

APPENDIX F
RESERVED SYMBOLS

APPENDIX G
SAMPLE PROGRAM

APPENDIX H
REFERENCE TABLES

APPENDIX J
ERROR MESSAGES
## Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Register Bank Selection</td>
<td>1-12</td>
</tr>
<tr>
<td>1-2</td>
<td>Symbolically Addressable Hardware Registers</td>
<td>1-14</td>
</tr>
<tr>
<td>1-3</td>
<td>State of the 8051 after Power-up</td>
<td>1-19</td>
</tr>
<tr>
<td>2-1</td>
<td>Special Assembler Symbols</td>
<td>2-2</td>
</tr>
<tr>
<td>2-2</td>
<td>Predefined Bit Addresses</td>
<td>2-7</td>
</tr>
<tr>
<td>2-3</td>
<td>Assembly Language Number Representation</td>
<td>2-9</td>
</tr>
<tr>
<td>2-4</td>
<td>Examples of Number Representation</td>
<td>2-9</td>
</tr>
<tr>
<td>2-5</td>
<td>Interpretations of Number Representation</td>
<td>2-10</td>
</tr>
<tr>
<td>2-6</td>
<td>Predefined Data Addresses</td>
<td>2-12</td>
</tr>
</tbody>
</table>

## Illustrations

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Assembler Outputs</td>
<td>1-2</td>
</tr>
<tr>
<td>1-2</td>
<td>MCS-51 Software Development Flow Chart</td>
<td>1-3</td>
</tr>
<tr>
<td>1-3</td>
<td>MCS-51 Example Program Listing</td>
<td>1-4</td>
</tr>
<tr>
<td>1-4</td>
<td>MCS-51 Block Diagram</td>
<td>1-9</td>
</tr>
<tr>
<td>1-5</td>
<td>MCS-51 Code Address Space and External Data Address Space</td>
<td>1-10</td>
</tr>
<tr>
<td>1-6</td>
<td>MCS-51 Data Address Space and Bit Address Space</td>
<td>1-11</td>
</tr>
<tr>
<td>1-7</td>
<td>MCS-51 Data Units</td>
<td>1-11</td>
</tr>
<tr>
<td>1-8</td>
<td>Bit Descriptions of Program Status Word</td>
<td>1-15</td>
</tr>
<tr>
<td>1-9</td>
<td>Bit Descriptions of TCON</td>
<td>1-15</td>
</tr>
<tr>
<td>1-10</td>
<td>Bit Descriptions for Port 3</td>
<td>1-16</td>
</tr>
<tr>
<td>1-11</td>
<td>Bit Descriptions for Serial Port Control</td>
<td>1-17</td>
</tr>
<tr>
<td>1-12</td>
<td>Bit Descriptions for Interrupt Enable and Interrupt Priority</td>
<td>1-18</td>
</tr>
<tr>
<td>2-1</td>
<td>Hardware Register Address Area</td>
<td>2-4</td>
</tr>
<tr>
<td>2-2a</td>
<td>Bit Addressable Bytes in RAM</td>
<td>2-6</td>
</tr>
<tr>
<td>2-2b</td>
<td>Bit Addressable Bytes in Hardware Register Address Area</td>
<td>2-6</td>
</tr>
<tr>
<td>3-1</td>
<td>Format For Instruction Definitions</td>
<td>3-2</td>
</tr>
<tr>
<td>7-1</td>
<td>Example Listing File Format</td>
<td>7-14</td>
</tr>
<tr>
<td>7-2</td>
<td>Example Heading</td>
<td>7-17</td>
</tr>
<tr>
<td>7-3</td>
<td>Example Source Listing</td>
<td>7-17</td>
</tr>
<tr>
<td>7-4</td>
<td>Examples of Macro Listing Modes</td>
<td>7-18</td>
</tr>
<tr>
<td>7-5</td>
<td>Example Symbol Table Listing</td>
<td>7-19</td>
</tr>
</tbody>
</table>
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.

![Diagram of Assembler Outputs](image)

Figure 1-1. Assembler Outputs

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*.
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

SOURCE

Figure 1-3. MCS-51™Example Program Listing
Figure 1-3. MCS-51™ Example Program Listing (Cont’d.)
This routine converts a binary 2's complement number to a 4 character ASCII string.

**INPUT:**
The binary value must be located in memory at the address contained in register 1.

**OUTPUT:**
The 4 character result is placed in memory with the first character at the address contained in register 1.

**NOTES:**
The contents of register A and B will be destroyed.
The contents of the memory location initially addressed by register 1 will be replaced with the first character in the resulting character string.

---

**Figure 1-3. MCS-51™ Example Program Listing (Cont'd.)**
This routine takes a 4 character string located in memory and converts it to a binary 2's complement number.

The number must begin with a sign character ('+' or '-'), and be between -128 and +127.

INPUT:
Four ASCII characters a sign character followed a '0' or a ',', and the last 2 characters can be any digit.

The contents of register 1 must point to the sign character.

OUTPUT:
A binary 2's complement representation of the value of the character string.
Register 1 contains the address of the binary value.

NOTES:
The contents of the memory location initially addressed by register 1 is destroyed.
The contents of registers 7 and B and the accumulator are destroyed.

TEMP EQU R7
ASCRIN: ; Go right to number compute sign at end
INC NUMB_PTR
MOV A, NUMB_PTR
SUBB A, ZERO
MOV B, #100
MUL AB
Store first digit's value and go to next digit
MOV TEMP, A
INC NUMB_PTR
MOV A, NUMB_PTR
SUBB A, ZERO
MOV B, #10
MUL AB
Add first digit value to second store and go to third digit
ADD A, TEMP
MOV TEMP, A
INC RUMB_PTR
MOV A, @NUMB_PTR
Add third digit value to total. Store and go back for sign
ADD A, TEMP
MOV TEMP, A
DEC NUMB_PTR
DEC NUMB_PTR
DEC NUMB_PTR
DEC NUMB_PTR
DEC NUMB_PTR
ADD A, TEMP
MOV @NUMB_PTR, A
CJNE A, #0, POS
MOV A, TEMP
CPL A
MOV TEMP, A
POS: MOV A, TEMP
Store result and return

Introduction
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**, **exclusive 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
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.
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.
Arithmetic and Logic Functions

The arithmetic functions include:

- ADD—signed 2's complement addition
- ADDC—signed 2's complement addition
- SUBBB—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>
<thead>
<tr>
<th>RS1</th>
<th>RS0</th>
<th>Bank</th>
<th>Memory Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00H—07H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>08H—0FH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>10H—17H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>18H—1FH</td>
</tr>
</tbody>
</table>
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,

\texttt{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):

\texttt{MOV SBUF,\#'?'}

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

**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

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
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:

<table>
<thead>
<tr>
<th>Port</th>
<th>Predefined Symbol</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P0</td>
<td>80H</td>
</tr>
<tr>
<td>1</td>
<td>P1</td>
<td>90H</td>
</tr>
<tr>
<td>2</td>
<td>P2</td>
<td>A0H</td>
</tr>
<tr>
<td>3</td>
<td>P3</td>
<td>B0H</td>
</tr>
</tbody>
</table>

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

---
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](image)

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.

<table>
<thead>
<tr>
<th>Predefined Label</th>
<th>Location</th>
<th>Interrupting Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>00H</td>
<td>Power on Reset (First instruction executed on power up.)</td>
</tr>
<tr>
<td>EXTI0</td>
<td>03H</td>
<td>External interrupt 0</td>
</tr>
<tr>
<td>TIMER0</td>
<td>0BH</td>
<td>Timer 0</td>
</tr>
<tr>
<td>EXTI1</td>
<td>13H</td>
<td>External interrupt 1</td>
</tr>
<tr>
<td>TIMER1</td>
<td>1BH</td>
<td>Timer 1</td>
</tr>
<tr>
<td>SINT</td>
<td>23H</td>
<td>Serial I/O port</td>
</tr>
</tbody>
</table>
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**

---

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>00H</td>
</tr>
<tr>
<td>Multiplication Register</td>
<td>00H</td>
</tr>
<tr>
<td>Data Pointer</td>
<td>0000H</td>
</tr>
<tr>
<td>Interrupt Enable</td>
<td>00H</td>
</tr>
<tr>
<td>Interrupt Priority</td>
<td>00H</td>
</tr>
<tr>
<td>Port 0</td>
<td>0FFH</td>
</tr>
<tr>
<td>Port 1</td>
<td>0FFH</td>
</tr>
<tr>
<td>Port 2</td>
<td>0FFH</td>
</tr>
<tr>
<td>Port 3</td>
<td>0FFH</td>
</tr>
<tr>
<td>Program Counter</td>
<td>0000H</td>
</tr>
<tr>
<td>Program Status Word</td>
<td>00H</td>
</tr>
<tr>
<td>Serial Port Control</td>
<td>00H</td>
</tr>
<tr>
<td>Serial I/O Buffer</td>
<td>undefined</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>07H</td>
</tr>
<tr>
<td>Timer Control</td>
<td>00H</td>
</tr>
<tr>
<td>Timer Mode</td>
<td>00H</td>
</tr>
<tr>
<td>Timer 0 Counter</td>
<td>0000H</td>
</tr>
<tr>
<td>Timer 1 Counter</td>
<td>0000H</td>
</tr>
</tbody>
</table>

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.
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:

\[ \text{[Label:]} \ Mnemonic \ [Operand] \ [,Operand] \ [,Operand] \ ;\text{Comment} \]

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

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

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.

\[
\begin{align*}
\text{ADD A, @R1} & \quad ; \text{Add the contents of the on-chip RAM location addressed by register 1 to the accumulator} \\
\text{INC @R0} & \quad ; \text{Increment the contents of the on-chip RAM location addressed by register 0} \\
\text{MOVX @DPTR, A} & \quad ; \text{Move the contents of the accumulator to the off-chip memory location addressed by the data pointer}
\end{align*}
\]

**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.

\[
\begin{align*}
\text{MOV A, #0E0H} & \quad ; \text{Place the hex constant E0 in the accumulator} \\
\text{MOV DPTR, #0A14FH} & \quad ; \text{This is the only instruction that uses a 16-bit immediate data operand} \\
\text{ANL A, #128} & \quad ; \text{Mask all but the high order bit of the accumulator} \\
& \quad ; 128(\text{base 10}) = 1000 0000(\text{base 2})
\end{align*}
\]

**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 incompatibility 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 yourself, 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

Figure 2-2b. Bit Addressable Bytes in Hardware Register Address Area
### Table 2-2. Predefined Bit Addresses

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bit Position</th>
<th>Bit Address</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CY</td>
<td>PSW.7</td>
<td>D7H</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>AC</td>
<td>PSW.6</td>
<td>D6H</td>
<td>Auxiliary Carry Flag</td>
</tr>
<tr>
<td>F0</td>
<td>PSW.5</td>
<td>D5H</td>
<td>Flag 0</td>
</tr>
<tr>
<td>RS1</td>
<td>PSW.4</td>
<td>D4H</td>
<td>Register Bank Select Bit 1</td>
</tr>
<tr>
<td>RS0</td>
<td>PSW.3</td>
<td>D3H</td>
<td>Register Bank Select Bit 0</td>
</tr>
<tr>
<td>OV</td>
<td>PSW.2</td>
<td>D2H</td>
<td>Overflow Flag</td>
</tr>
<tr>
<td>P</td>
<td>PSW.0</td>
<td>D0H</td>
<td>Parity Flag</td>
</tr>
<tr>
<td>TF1</td>
<td>TCON.7</td>
<td>8FH</td>
<td>Timer 1 Overflow Flag</td>
</tr>
<tr>
<td>TR1</td>
<td>TCON.6</td>
<td>8EH</td>
<td>Timer 1 Run Control Bit</td>
</tr>
<tr>
<td>TF0</td>
<td>TCON.5</td>
<td>8DH</td>
<td>Timer 0 Overflow Flag</td>
</tr>
<tr>
<td>TR0</td>
<td>TCON.4</td>
<td>8CH</td>
<td>Timer 0 Run Control Bit</td>
</tr>
<tr>
<td>IE1</td>
<td>TCON.3</td>
<td>8BH</td>
<td>Interrupt 1 Edge Flag</td>
</tr>
<tr>
<td>IT1</td>
<td>TCON.2</td>
<td>8AH</td>
<td>Interrupt 1 Type Control Bit</td>
</tr>
<tr>
<td>IE0</td>
<td>TCON.1</td>
<td>89H</td>
<td>Interrupt 0 Edge Flag</td>
</tr>
<tr>
<td>IT0</td>
<td>TCON.0</td>
<td>88H</td>
<td>Interrupt 0 Type Control Bit</td>
</tr>
<tr>
<td>SM0</td>
<td>SCON.7</td>
<td>9FH</td>
<td>Serial Mode Control Bit 0</td>
</tr>
<tr>
<td>SM1</td>
<td>SCON.6</td>
<td>9EH</td>
<td>Serial Mode Control Bit 1</td>
</tr>
<tr>
<td>SM2</td>
<td>SCON.5</td>
<td>9DH</td>
<td>Serial Mode Control Bit 2</td>
</tr>
<tr>
<td>REN</td>
<td>SCON.4</td>
<td>9CH</td>
<td>Receiver Enable</td>
</tr>
<tr>
<td>TB8</td>
<td>SCON.3</td>
<td>9BH</td>
<td>Transmit Bit 8</td>
</tr>
<tr>
<td>RB8</td>
<td>SCON.2</td>
<td>9AH</td>
<td>Receive Bit 8</td>
</tr>
<tr>
<td>T1</td>
<td>SCON.1</td>
<td>99H</td>
<td>Transmit Interrupt Flag</td>
</tr>
<tr>
<td>RI</td>
<td>SCON.0</td>
<td>98H</td>
<td>Receive Interrupt Flag</td>
</tr>
<tr>
<td>EA</td>
<td>IE.7</td>
<td>AFH</td>
<td>Enable All Interrupts</td>
</tr>
<tr>
<td>ES</td>
<td>IE.4</td>
<td>ACH</td>
<td>Enable Serial Port Interrupt</td>
</tr>
<tr>
<td>ET1</td>
<td>IE.3</td>
<td>ABH</td>
<td>Enable Timer 1 Interrupt</td>
</tr>
<tr>
<td>EX1</td>
<td>IE.2</td>
<td>AAH</td>
<td>Enable External Port Interrupt</td>
</tr>
<tr>
<td>ET0</td>
<td>IE.1</td>
<td>A9H</td>
<td>Enable Timer 0 Interrupt</td>
</tr>
<tr>
<td>EX0</td>
<td>IE.0</td>
<td>A8H</td>
<td>Enable External Port Interrupt</td>
</tr>
<tr>
<td>RD</td>
<td>P3.7</td>
<td>B7H</td>
<td>Read Data for External Memory</td>
</tr>
<tr>
<td>WR</td>
<td>P3.6</td>
<td>B6H</td>
<td>Write Data for External Memory</td>
</tr>
<tr>
<td>T1</td>
<td>P2.5</td>
<td>B5H</td>
<td>Timer/Counter 1 External Flag</td>
</tr>
<tr>
<td>T0</td>
<td>P2.4</td>
<td>B4H</td>
<td>Timer/Counter 0 External Flag</td>
</tr>
<tr>
<td>INT1</td>
<td>P3.3</td>
<td>B3H</td>
<td>Interrupt 1 Input Pin</td>
</tr>
<tr>
<td>INT0</td>
<td>P3.2</td>
<td>B2H</td>
<td>Interrupt 0 Input Pin</td>
</tr>
<tr>
<td>TXD</td>
<td>P3.1</td>
<td>B1H</td>
<td>Serial Port Transmit Pin</td>
</tr>
<tr>
<td>RXD</td>
<td>P3.0</td>
<td>B0H</td>
<td>Serial Port Receive Pin</td>
</tr>
<tr>
<td>PS</td>
<td>IP.4</td>
<td>BCH</td>
<td>Priority of Serial Port Interrupt</td>
</tr>
<tr>
<td>PT1</td>
<td>IP.3</td>
<td>BBH</td>
<td>Priority of Timer 1 Interrupt</td>
</tr>
<tr>
<td>PX1</td>
<td>IP.2</td>
<td>BAH</td>
<td>Priority of External Port Interrupt</td>
</tr>
<tr>
<td>PT0</td>
<td>IP.1</td>
<td>B9H</td>
<td>Priority of Timer 0</td>
</tr>
<tr>
<td>PX0</td>
<td>IP.0</td>
<td>B8H</td>
<td>Priority of External Port Interrupt</td>
</tr>
</tbody>
</table>
Operands and Assembly-Time Expressions

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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SJMP LABEL</td>
<td>Jump to LABEL (relative offset LABEL must be within −128 and +127 of instruction that follows SJMP)</td>
</tr>
<tr>
<td>ACALL SORT</td>
<td>Call subroutine labeled SORT (SORT must be an address to within the current 2K page)</td>
</tr>
<tr>
<td>LJMP EXIT</td>
<td>Long jump; the label or symbol EXIT must be defined somewhere in the program</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Number System</th>
<th>Base Designator</th>
<th>Digits in Order of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>B</td>
<td>0, 1</td>
</tr>
<tr>
<td>Octal</td>
<td>O or Q</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>Decimal</td>
<td>D or (nothing)</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>H</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>base 16</th>
<th>base 10</th>
<th>base 8</th>
<th>base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>50H</td>
<td>80</td>
<td>120</td>
<td>01010000B</td>
</tr>
<tr>
<td>0ACH*</td>
<td>172D</td>
<td>254Q</td>
<td>10101100B</td>
</tr>
<tr>
<td>01H</td>
<td>1</td>
<td>1Q</td>
<td>1B</td>
</tr>
<tr>
<td>10H</td>
<td>16D</td>
<td>20Q</td>
<td>10000B</td>
</tr>
</tbody>
</table>

* 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.

1. Complement each bit in the number.
2. Add 1 to the complement.

\[
\begin{align*}
0000 \ 0000 \ 0010 \ 0000B &= 20H \\
1111 \ 1111 \ 1101 \ 1111 &= \text{Not } 20H \\
1111 \ 1111 \ 1110 \ 0000 &= (\text{Not } 20H) + 1 \\
1111 \ 1111 \ 1110 \ 0000B &= -20H
\end{align*}
\]

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

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

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

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.

2-12
Arithmetic Operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Unary plus or add</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus or subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Integer division (discard remainder)</td>
</tr>
<tr>
<td>MOD</td>
<td>Modular division (discard quotient)</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>Full 16-bit OR</td>
</tr>
<tr>
<td>AND</td>
<td>Full 16-bit AND</td>
</tr>
<tr>
<td>XOR</td>
<td>Full 16-bit exclusive OR</td>
</tr>
<tr>
<td>NOT</td>
<td>Full 16-bit complement</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHR</td>
<td>16-bit shift right</td>
</tr>
<tr>
<td>SHL</td>
<td>16-bit shift left</td>
</tr>
<tr>
<td>HIGH</td>
<td>Select the high order byte of operand</td>
</tr>
<tr>
<td>LOW</td>
<td>Select the low order byte of operand</td>
</tr>
<tr>
<td>( )</td>
<td>Evaluate the contents of the parenthesis first</td>
</tr>
</tbody>
</table>

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 shifted out 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>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>
The following examples all will return TRUE (0FFFFH):

\[ 27H \text{ EQ } 39D \]
\[ 27H \text{	extless} \textgreater 27D \]
\[ 33 \text{ LT } 34 \]
\[ 7 \text{> } 5 \]
\[ 16 \text{ GE } 10H \]

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, =, \textless, \textless=, \textgreater, \textgreater=
- 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.

<table>
<thead>
<tr>
<th>HIGH</th>
<th>LOW</th>
<th>NOT</th>
<th>OR</th>
<th>XOR</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>NE</td>
<td>GT</td>
<td>GE</td>
<td>LE</td>
<td>LT</td>
</tr>
<tr>
<td>*</td>
<td>/</td>
<td>MOD</td>
<td>SHR</td>
<td>SHL</td>
<td></td>
</tr>
</tbody>
</table>
• 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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operator</th>
<th>Operand</th>
<th>Segment Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>( )</td>
<td>Value (S)</td>
<td>Segment type maintained</td>
</tr>
<tr>
<td>−</td>
<td>+</td>
<td>Value (S)</td>
<td>Segment type maintained</td>
</tr>
<tr>
<td>−</td>
<td>−</td>
<td>Value (S)</td>
<td>Segment type maintained</td>
</tr>
<tr>
<td>Value (N)</td>
<td>+</td>
<td>Value (S)</td>
<td>Segment type maintained</td>
</tr>
<tr>
<td>Value (S)</td>
<td>+</td>
<td>Value (N)</td>
<td>Segment type maintained</td>
</tr>
<tr>
<td>Value (S)</td>
<td>−</td>
<td>Value (S)</td>
<td>Segment type lost</td>
</tr>
<tr>
<td>Value (N)</td>
<td>−</td>
<td>Value (S)</td>
<td>Segment type maintained</td>
</tr>
<tr>
<td>Value (S)</td>
<td>−</td>
<td>Value (N)</td>
<td>Segment type maintained</td>
</tr>
</tbody>
</table>

(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.
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.

---

**ADD**

**Add Immediate Data**

**Mnemonic:** ADD

**Operands:**
- A
- data

**Format:** ADD A, #data

**Bit Pattern:**
```
00100100 [Immediate Data]
```

**Operation:** (A) - (A) + data

**Bytes:** 2

**Cycles:** 1

**Flags:**
```
C AC PS RS1 RS0 OV
```

**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:**
```
00100100 00110011
```

**Before**

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100100</td>
</tr>
</tbody>
</table>

**After**

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011000</td>
</tr>
</tbody>
</table>

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

---

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

**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

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

Absolute Call Within 2K Byte Page

**Mnemonic:** ACALL

**Operands:** code address

**Format:** ACALL code address

**Bit Pattern:**

```
<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Operation:**

- \( (PC) \leftarrow (PC) + 2 \)
- \( (SP) \leftarrow (SP) + 1 \)
- \( ((SP)) \leftarrow (PC \text{ low}) \)
- \( (SP) \leftarrow (SP) + 1 \)
- \( ((SP)) \leftarrow (PC \text{ high}) \)
- \( (PC) \text{ 0-10} \leftarrow \text{ page address} \)

**Bytes:** 2

**Cycles:** 2

**Flags:**

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

**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:

<table>
<thead>
<tr>
<th>01010001</th>
<th>00110011</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>Program Counter</td>
</tr>
<tr>
<td>00000000</td>
<td>00110101</td>
</tr>
<tr>
<td>15 8 7 0</td>
<td>15 8 7 0</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>00100110</td>
<td>00101000</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
<tr>
<td>(27H)</td>
<td>(27H)</td>
</tr>
<tr>
<td>00000000</td>
<td>00110111</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
<tr>
<td>(28H)</td>
<td>(28H)</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

**Notes:** 2, 3
Add Immediate Data

Mnemonic: ADD

Operands: A Accumulator
data \(-256 \leq data \leq +255\)

Format: ADD A,#data

Bit Pattern:

<table>
<thead>
<tr>
<th>0 0 1 0 0 1 0 0</th>
<th>Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: \((A) \leftarrow (A) + data\)

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

<table>
<thead>
<tr>
<th>●</th>
<th>●</th>
<th></th>
<th>●</th>
<th>●</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>0 0 1 0 0 1 0 0</th>
<th>0 0 1 1 0 0 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>0 0 1 0 0 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>0 1 0 1 1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: 4, 5, 6, 7
Add Indirect Address

Mnemonic: ADD

Operands: A Accumulator
          Rr Register 0 <= r <= 1

Format: ADD A,Rr

Bit Pattern:

\[
\begin{array}{c}
0010011r \\
7 \\
0
\end{array}
\]

Operation: \( (A) \leftarrow (A) + ((Rr)) \)

Bytes: 1
Cycles: 1

Flags:

\[
\begin{array}{cccccccc}
C & AC & F0 & RS1 & RS0 & OV & P \\
\bullet & \bullet & \square & \square & \square & \bullet & \bullet
\end{array}
\]

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:

\[
00100111
7 \\
0
\]

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 1</th>
<th>(1CH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000110</td>
<td>00011100</td>
<td>01100010</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 1</th>
<th>(1CH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101000</td>
<td>00011100</td>
<td>01100010</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Notes: 5, 6, 7, 15
Add Register

Mnemonic: ADD
Operands: A Accumulator 
Rr Register 0 <= r <= 7
Format: ADD A, Rr
Bit Pattern: 00101 rrr
Operation: (A) ← (A) + (Rr)
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P
PSW
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
Accumulator
01110110
7 0
Register 6
10000101
7 0

After
Accumulator
11111011
7 0
Register 6
10000101
7 0

Notes: 5, 6, 7
Add Memory

Mnemonic: ADD
Operands: A Accumulator
          data address 0 <= data address <= 255
Format: ADD A, data address
Bit Pattern:
          0 0 1 0 0 1 0 1 0 0 1 0 0 0 1 0
          7 0 7 0
Operation: (A) ← (A) + (data address)
Bytes: 2
Cycles: 1
Flags:

C    AC    F0    RS1    RS0    OV    P
  ●    ●    ■    ■    ■    ■    ●
PSW

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
          0 0 1 0 0 1 1 0
          7 0
(data address) 32H
          0 1 0 1 0 0 1 1
          7 0

After

Accumulator
          0 1 1 1 1 0 0 1
          7 0

Notes: 5, 6, 7, 8
Add Carry Plus Immediate Data to Accumulator

Mnemonic: ADDC
Operands: A, #data
-256 <= data <= +255
Format: ADDC A, #data
Bit Pattern:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Immediate Data
Operation: (A) ← (A) + (C) + data
Bytes: 2
Cycles: 1
Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Accumulator

After

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Accumulator

Carry

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: 4, 5, 6, 7
Add Carry Plus Indirect Address to Accumulator

**Mnemonic:** ADDC

**Operands:**
- A: Accumulator
- Register: \(0 \leq r \leq 1\)

**Format:**
ADDC A, @Rr

**Bit Pattern:**
```
0011011r
```

**Operation:**
\((A) \leftarrow (A) + (C) + ((Rr))\)

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

**Flags:**
```
C  AC  F0  RS1  RS0  OV  P
```

**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:

\[
\begin{array}{c}
\text{Before} \\
\text{Accumulator} & 11101000 \\
\text{Register 1} & 01101001 \\
\text{(69H)} & 00011000 \\
\text{Carry} & 0
\end{array}
\]

\[
\begin{array}{c}
\text{After} \\
\text{Accumulator} & 00000000 \\
\text{Register 1} & 01101001 \\
\text{(69H)} & 00011000 \\
\text{Carry} & 1
\end{array}
\]

Notes: 5, 6, 7, 15
Add Carry Plus Register to Accumulator

Mnemonic: ADDC

Operands: A, Rr

Format: ADDC A, Rr

Bit Pattern: 00111rrr

Operation: (A) ← (A) + (C) + (Rr)

Bytes: 1
Cycles: 1

Flags: C, AC, F0, RS1, RS0, OV, P

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

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 7</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110000</td>
<td>00001010</td>
<td>1</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 7</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111011</td>
<td>00001010</td>
<td>0</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

Notes: 5, 6, 7
Add Carry Plus Memory to Accumulator

Mnemonic: ADDC
Operands: A, data address
   0 <= data address <= 255
Format: ADDC A, data address
Bit Pattern:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: (A) ← (A) + (C) + (data address)
Bytes: 2
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>(25H)</td>
</tr>
<tr>
<td>0</td>
<td>C</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>(25H)</td>
</tr>
<tr>
<td>0</td>
<td>C</td>
</tr>
</tbody>
</table>

Notes: 5, 6, 7, 8
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
    7 0 7 0
```
Operation: 
\[(PC) \leftarrow (PC) + 2\]
\[(PC) 0-10 \leftarrow \text{page address}\]
Bytes: 2
Cycles: 2
Flags: 
```
    C AC F0 RS1 RS0 OV P
```
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
TOMP: MOV A,R1

ORG 0EADCH
AJMP TOMP ; Jump backwards to TOMP ; at location 0E80FH
```

Encoded Instruction:
```
0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1
7 0 7 0
```

Before
\[
\begin{array}{cccc}
    1 & 1 & 1 & 0 \\
    0 & 1 & 0 & 1 \\
    1 & 0 & 1 & 1 \\
    1 & 0 & 0 & 0 \\
\end{array}
\]

After
\[
\begin{array}{cccc}
    1 & 1 & 1 & 0 \\
    1 & 1 & 0 & 0 \\
    0 & 0 & 0 & 1 \\
    1 & 1 & 0 & 0 \\
\end{array}
\]

Notes: None
Logical AND Immediate Data to Accumulator

Mnemonic: ANL

Operands: A  Accumulator
data  \(-256 \leq data \leq +255\)

Format:  ANL  A,#data

Bit Pattern:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01010100</strong></td>
<td>Immediate Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: \((A) \leftarrow (A) \ AND \ data\)

Bytes: 2
Cycles: 1

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>P</td>
</tr>
</tbody>
</table>

PSW

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:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01010100</strong></td>
<td><strong>00010000</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before  After

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>01110111</strong></td>
<td><strong>00000000</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes: 4, 5
Logical AND Indirect Address to Accumulator

Mnemonic: ANL
Operands: A Accumulator
          Rr Register 0 <= r <= 1
Format: ANL A,@Rr
Bit Pattern: 0101011r
            7 0
Operation: (A) ← (A) AND ((Rr))
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:

Before
Accumulator: 00111111
            7 0
Register 0: 01010010
            7 0
            (52H)
After
Accumulator: 00001111
            7 0
Register 0: 01010010
            7 0
            (52H)
Notes: 5, 15
Logical AND Register to Accumulator

Mnemonic: ANL
Operands: A Accumulator 	Rr 0 <= Rr <= 7
Format: ANL A,Rr
Bit Pattern: 01011rrr
Operation: (A) ← (A) AND (Rr)
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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
Accumulator 10011001
7 0

After
Accumulator 10000000
7 0

Register 4 1000000
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:

```
01010101 | Data Address
7 0 7 0
```

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

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

PSW

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:

```
01010101 00110111
7 0 7 0
```

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110111</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

(37H)

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

(37H)

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: 5, 8
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:

```
1 0 0 0 0 1 0
7 0 7 0
```

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

Bytes: 2

Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

```
   ●   ●   ●   ●   ●   ●
PSW
```

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:

```
1 0 0 0 0 1 0 0 0 1 0 1 0 1 1
7 0 7 0
```

Before

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>(37)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00101110</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>(37)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00101110</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Bit Address</th>
<th>Bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110000</td>
<td>0 7 0</td>
</tr>
</tbody>
</table>

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

Bytes: 2
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

PSW

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:

<table>
<thead>
<tr>
<th>10110000</th>
<th>01000101</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

(40)

<table>
<thead>
<tr>
<th>01011000</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 5 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

(40)

<table>
<thead>
<tr>
<th>01011000</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 5 0</td>
</tr>
</tbody>
</table>

Notes: None
Logical AND Immediate Data to Memory

Mnemonic:  ANL

Operands:  
\[
\begin{align*}
\text{data address} & \quad 0 \leq \text{data address} \leq 255 \\
data & \quad -256 \leq \text{data} \leq +255
\end{align*}
\]

Format:  ANL  data address,#data

Bit Pattern:

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
\text{Data Address} & \text{Immediate Data} \\
7 & 0 & 7 & 0 & 7 & 0 & 0 & 0
\end{array}
\]

Operation:  \((\text{data address}) \leftarrow (\text{data address}) \text{ AND } \text{data}\)

Bytes:  3

Cycles:  2

Flags:  

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW

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:  

\[
\begin{align*}
\text{MOV} & \ 57H,\text{PSW} & \quad ; \text{Move PSW to 57H} \\
\text{ANL} & \ 57H,#01H & \quad ; \text{Mask out all but parity bit} \\
& & \quad ; \text{to check accumulator parity}
\end{align*}
\]

Encoded Instruction:

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
\text{Data Address} & \text{Immediate Data} \\
7 & 0 & 7 & 0 & 7 & 0 & 0 & 0
\end{array}
\]

Before  

(57H)  

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
\text{Data Address} \\
7 & 0
\end{array}
\]

After  

(57H)  

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\text{Data Address} \\
7 & 0
\end{array}
\]

Notes:  4, 9
Logical AND Accumulator to Memory

Mnemonic: ANL

Operands: 
- data address \(0 \leq \text{data address} \leq 255\)
- A Accumulator

Format: ANL data address, A

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 0 1 0</td>
<td>7 0 7 0</td>
</tr>
</tbody>
</table>

Operation: 
\[(\text{data address}) \leftarrow (\text{data address}) \text{ AND } A\]

Bytes: 2
Cycles: 1

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

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

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 0 1</td>
<td>1 0 0 0 0 0 1</td>
</tr>
<tr>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

(10H)

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

Note: 9
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: C AC F0 RS1 RS0 OV P

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
10000000 11011100
Stack Pointer
01100100

After

Program Counter
10000010 00110011
Stack Pointer
01100110

Notes: 1, 2, 3
Compare Indirect Address to Immediate Data, Jump if Not Equal

Mnemonic: CJNE

Operands:
- Register: 0 <= r <= 1
- Immediate Data: -256 <= data <= +255
- Code Address

Format: CJNE @Rr,#data,code address

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Immediate Data</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>data</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>255</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation:
- \((PC) \leftarrow (PC) + 3\)
- IF \(((Rr)) <> data\) THEN \((PC) \leftarrow (PC) + \text{relative offset}\)
- IF \(((Rr)) < data\) THEN \((C) \leftarrow 1\)
- ELSE \((C) \leftarrow 0\)

Bytes: 3
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

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: \texttt{CJNE @R1,#01,SCAB} ; Jump if indirect address
\texttt{does not equal 1}

\texttt{SCAB: MOV C,F0} ; 54H bytes from CJNE

Encoded Instruction:

\begin{verbatim}
   1 0 1 1 0 1 1 1 0 0 0 0 0 0 0 1 0 1 0 1 1 1
\end{verbatim}

Before After

\begin{tabular}{lc}
Register 1 & Register 1 \\
0 1 0 1 0 0 1 1 & 0 1 0 1 0 0 1 1 \\
0 7 & 0 7
\end{tabular}

(53H) (53H)

\begin{tabular}{lc}
Carry Flag & Carry Flag \\
1 & 0
\end{tabular}

\begin{tabular}{lc}
Program Counter & Program Counter \\
0 0 0 0 0 0 1 1 0 1 1 0 0 & 0 0 0 0 0 0 1 0 1 1 1 0
\end{tabular}

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:

<table>
<thead>
<tr>
<th>bytes</th>
<th>immediate data</th>
<th>rel. offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011 0100</td>
<td>0 7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```
10110100 00010000 01010111
7 0 7 0 7 0
```

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accumulator</strong></td>
<td><strong>Accumulator</strong></td>
</tr>
<tr>
<td>01010000</td>
<td>01010000</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
<tr>
<td><strong>Carry Flag</strong></td>
<td><strong>Carry Flag</strong></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Program Counter</strong></td>
<td><strong>Program Counter</strong></td>
</tr>
<tr>
<td>00010000 11011100</td>
<td>00010001 00110110</td>
</tr>
<tr>
<td>15 8 7 0</td>
<td>15 8 7 0</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th></th>
<th>Data Address</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

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: C AC F0 RS1 RS0 OV P

PSW

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: \textit{CJNE A, 37H, TEST; }Jump if 37H and accumulator are not equal.

\textit{TEST: INC A; }4FH bytes from CJNE

Encoded Instruction:

<table>
<thead>
<tr>
<th>10110101</th>
<th>00110111</th>
<th>01001100</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Before After

(37H) (37H)

<table>
<thead>
<tr>
<th>01111110</th>
<th>01111110</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Accumulator Accumulator

<table>
<thead>
<tr>
<th>00100110</th>
<th>00100110</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Carry Flag Carry Flag

| 0 | 1 |

Program Counter Program Counter

<table>
<thead>
<tr>
<th>00000000</th>
<th>11011100</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00000001</th>
<th>01101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Notes: 8, 10, 11, 12
Compare Immediate Data to Register, Jump if Not Equal

Mnemonic: CJNE

Operands:
- Rr: Register $0 \leq r \leq 7$
- data: $-256 \leq data \leq +255$
- code address

Format: CJNE Rr,#data,code address

Bit Pattern:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>Immediate Data</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:

- \((PC) \leftarrow (PC) + 3\)
- IF \((Rr) \neq data\) THEN \((PC) \leftarrow (PC) + \text{relative offset}\)
- IF \((Rr) < data\) THEN \((C) \leftarrow 1\)
- ELSE \((C) \leftarrow 0\)

Bytes: 3
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

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:  

\[ \text{CJNE R5,\#32H,SKIP10} \]; Jump if register 5 does not equal 32H

\[ \text{SKIP10: MOV R5,P0} \]; 13 bytes from CJNE

Encoded Instruction:

\[ \begin{array}{cccc}
1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 \\
\end{array} \]

Before After

Register 5
\[ \begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array} \]

Carry Flag
\[ \begin{array}{cccc}
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array} \]

Program Counter
\[ \begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
\end{array} \]

Notes: 4, 10, 11, 12
Clear Accumulator

Mnemonic: CLR
Operands: A Accumulator
Format: CLR A
Bit Pattern: 
\[
\begin{array}{c}
11100100 \\
7 \\
0
\end{array}
\]
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:
\[
\begin{array}{c}
11100100 \\
7 \\
0
\end{array}
\]

Before
After

Accumulator
Accumulator
\[
\begin{array}{c}
00111111 \\
7 \\
0
\end{array}
\]
\[
\begin{array}{c}
00000000 \\
7 \\
0
\end{array}
\]

Note: 5
CLR

Clear Carry Flag

Mnemonic: CLR
Operands: C Carry Flag
Format: CLR C
Bit Pattern:

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000011</td>
<td></td>
</tr>
</tbody>
</table>

Operation: (C) ← 0
Bytes: 1
Cycles: 1
Flags: C AC FO RS1 RS0 OV P

Description: This instruction resets the carry flag to 0.

Example: CLR C ; Set carry flag to 0

Encoded Instruction:

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000011</td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Notes: None
Clear Bit

Mnemonic: CLR
Operands: \textit{bit address} 0 <= \textit{bit address} <= 255

Format: CLR \textit{bit address}

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: \((\text{bit address}) \leftarrow 0\)

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

PSW

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

Example: \textit{CLR 40.5} ; Set bit 5 of byte 40 to 0

Encoded Instruction:

Before | After
---|---
(40) | (40)
0 0 1 0 0 1 1 0 | 0 0 0 0 0 1 1 0

Notes: None
Complement Accumulator

Mnemonic: CPL
Operands: A Accumulator
Format: CPL A
Bit Pattern: 
\[
\begin{array}{c}
111110100 \\
7 \\
0
\end{array}
\]
Operation: \((A) \leftarrow \text{NOT} (A)\)
Bytes: 1
Cycles: 1
Flags: \[ C \ AC \ F0 \ RS1 \ RS0 \ OV \ P \]
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:
\[
\begin{array}{c}
11110011 \\
7 \\
0
\end{array}
\]
Before
Accumulator
\[
\begin{array}{c}
00110101 \\
7 \\
0
\end{array}
\]
After
Accumulator
\[
\begin{array}{c}
11001010 \\
7 \\
0
\end{array}
\]
Notes: None
Complement Carry Flag

Mnemonic: CPL
Operands: C Carry flag
Format: CPL C
Bit Pattern: 10110011
Operation: (C) ← NOT (C)
Bytes: 1
Cycles: 1
Flags: C AC FO RS1 RS0 OV p

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:

Before After

Carry Flag Carry Flag
1 0

Notes: None
CPL

Complement Bit

Mnemonic: CPL
Operands: bit address $0 \leq \text{bit address} \leq 255$
Format: CPL bit address
Bit Pattern: \begin{array}{c|c}
7 & 0 \\
0 & 7 \\
\end{array} Bit Address
Operation: \((\text{bit address}) \leftarrow \text{NOT} (\text{bit address})\)
Bytes: 2
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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:
\begin{array}{c}
10110010 \\
0001111 \\
\end{array}

Before After
\begin{array}{c|c}
7 & 0 \\
0 & 7 \\
\end{array}
\begin{array}{c|c}
10100110 \\
00100110 \\
\end{array}

Notes: None
Decimal Adjust Accumulator

Mnemonic: DA
Operands: A  Accumulator
Format: DA A
Bit Pattern: 11010100
Operation: (See description below.)
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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
DS DA A
; Adjust the Accumulator after add

Encoded Instruction:

```
1 1 1 1 0 1 0 1 0 0
```

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Carry Flag</th>
<th>Auxiliary Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 1 0 1 1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Carry Flag</th>
<th>Auxiliary Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Notes: 5, 6
**DEC**

**Decrement Indirect Address**

**Mnemonic:** DEC

**Operands:** Rr

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

**Format:** DEC @Rr

**Bit Pattern:**

```
 0 0 0 1 0 1 1 r
 7   0
```

**Operation:** ((Rr)) ← ((Rr)) − 1

**Bytes:** 1

**Cycles:** 1

**Flags:**

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 1 1 1</td>
</tr>
<tr>
<td>7   0</td>
</tr>
<tr>
<td>(37H)</td>
</tr>
</tbody>
</table>

**After**

<table>
<thead>
<tr>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 1 1 1</td>
</tr>
<tr>
<td>7   0</td>
</tr>
<tr>
<td>(37H)</td>
</tr>
</tbody>
</table>

**Note:** 15
Decrement Accumulator

Mnemonic: DEC
Operands: A
Format: DEC A
Bit Pattern: 00010100 7 0
Operation: (A) ← (A) - 1
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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
Decrement Register

Mnemonic: DEC
Operands: Rr  Register 0 <= r <= 7
Format: DEC Rr
Bit Pattern: 00011rrr
Operation: (Rr) ← (Rr) - 1
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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:

00011111
7  0

Before
Register 7
10101011
7  0

After
Register 7
10101010
7  0

Notes: None
Decrement Memory

Mnemonic: DEC
Operands: data address 0 <= data address <= 255
Format: DEC data address

Bit Pattern:

<table>
<thead>
<tr>
<th></th>
<th>0 0 0 1 0 1 0 1</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0 7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: \( (data\ address) \leftarrow (data\ address) - 1 \)

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW

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:

<table>
<thead>
<tr>
<th>0 0 0 1 0 1 0 1</th>
<th>0 0 1 1 0 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Before

(37H)

<table>
<thead>
<tr>
<th>1 1 0 1 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

(37H)

<table>
<thead>
<tr>
<th>1 1 0 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Note: 9
DIV

Divide Accumulator by B

Mnemonic: DIV
Operands: AB Register Pair
Format: DIV AB
Bit Pattern: 10000100
Operation: (AB) ← (A) / (B)
Bytes: 1
Cycles: 4
Flags: C AC F0 RS1 RS0 OV P

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:

10000100
7 0

Before
Accumulator
01110110
7 0

Multiplication Register (B)
0000101
7 0

After
Accumulator
00010111
7 0

Multiplication Register (B)
0000011
7 0

Note: 5
Decrement Register and Jump if Not Zero

Mnemonic: DJNZ
Operands: Rr, code address
Register 0 <= r <= 7
Format: DJNZ Rr, code address
Bit Pattern:

<table>
<thead>
<tr>
<th>110111 r r</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation:

\[
\begin{align*}
(\text{PC}) & \leftarrow (\text{PC}) + 2 \\
(\text{R}r) & \leftarrow (\text{R}r) - 1 \\
\text{IF } (\text{R}r) < > 0 & \text{ THEN} \\
(\text{PC}) & \leftarrow (\text{PC}) + \text{relative offset}
\end{align*}
\]

Bytes: 2
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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 jump to LOOP1 (15 bytes backward from INC; instruction)
        INC A
```

Encoded Instruction:

Before

<table>
<thead>
<tr>
<th>11011111</th>
<th>11100001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Register 7

Before

<table>
<thead>
<tr>
<th>00000010</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Program Counter

Before

<table>
<thead>
<tr>
<th>00000100</th>
<th>11011100</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 8 7 0</td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>00000100</th>
<th>11001111</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 8 7 0</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>11010101</th>
<th>Data Address</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 7 0 7 0 7 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation:

(PC) ← (PC) + 3
(data address) ← (data address) -1
IF (data address) <> 0
THEN

(PC) ← (PC) + relative offset

Bytes: 3
Cycles: 2
Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
</table>

PSW

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
backward from the INC A
instruction)

Encoded Instruction:

<table>
<thead>
<tr>
<th>11010101</th>
<th>01010111</th>
<th>11001010</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Before
(57H)

<table>
<thead>
<tr>
<th>01110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

After
(57H)

<table>
<thead>
<tr>
<th>01110110</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Program Counter

<table>
<thead>
<tr>
<th>00000000</th>
<th>11011100</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Program Counter

<table>
<thead>
<tr>
<th>00000000</th>
<th>10101001</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Notes: 9, 10, 11, 12
Increment Indirect Address

Mnemonic: INC

Operands: Rr, Register 0 ≤ r ≤ 1

Format: INC @Rr

Bit Pattern: 0000011r

Operation: ((Rr)) ← ((Rr)) + 1

Bytes: 1

Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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

00110010
7 0

(32H)

11011101
7 0

After

Register 0

00110010
7 0

(32H)

11011110
7 0

Note: 15
Increment Accumulator

Mnemonic: INC
Operands: A Accumulator
Format: INC A
Bit Pattern: \[
\begin{array}{c}
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 \\
\end{array}
\]
Operation: \((A) \leftarrow (A) + 1\)
Bytes: 1
Cycles: 1
Flags: 
\[
\begin{array}{cccccccc}
C & AC & F0 & RS1 & RS0 & OV & P \\
\end{array}
\]
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:
\[
\begin{array}{c}
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 \\
\end{array}
\]
Before
Accumulator
\[
\begin{array}{c}
1 & 1 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 \\
\end{array}
\]
After
Accumulator
\[
\begin{array}{c}
1 & 1 & 0 & 1 & 0 & 0 & 1 \\
7 & 0 \\
\end{array}
\]
Note: 5
INC

Increment Data Pointer

Mnemonic: INC
Operands: DPTR Data Pointer
Format: INC DPTR
Bit Pattern: 10100011
Operation: (DPTR) ← (DPTR) + 1
Bytes: 1
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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

0 0 0 0 1 0 0 1 1 1 1 1 1 1
15 8 7 0
0 0 0 0 1 0 1 0 0 0 0 0 0 0
15 8 7 0

Notes: None
Increment Register

Mnemonic: INC
Operands: Rr Register 0 <= r <= 7
Format: INC Rr
Bit Pattern: 00001rrr
Operation: (Rr) ← (Rr) + 1
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P
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:

<table>
<thead>
<tr>
<th>Rrr</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Register 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101011</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Register 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101100</td>
</tr>
</tbody>
</table>

Notes: None
Increment Memory

Mnemonic: INC

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

Format: INC data address

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101</td>
<td>0 7</td>
</tr>
</tbody>
</table>

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

Bytes: 2

Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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:

<table>
<thead>
<tr>
<th>Encoded Instruction</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000101 00110111</td>
<td>11011110</td>
<td>11011111</td>
</tr>
<tr>
<td>7 0 7 0</td>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

Note: 9
Jump if Bit Is Set

Mnemonic: JB
Operands: bit address \(0 \leq \text{bit address} \leq 255\)
code address
Format: JB bit address,code address
Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Bit Address</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 0 7 0</td>
<td></td>
<td>0 7 0</td>
</tr>
</tbody>
</table>
Operation: \((\text{PC}) \leftarrow (\text{PC}) + 3\)
If \((\text{bit address}) = 1\)
THEN
\((\text{PC}) \leftarrow (\text{PC}) + \text{relative offset}\)
Bytes: 3
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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: \texttt{JB 39.6,EXIT} \quad ; \text{Jump if bit 6 of byte 39 is 1}

\texttt{SJMP TOP}
\texttt{EXIT: MOV A,39} \quad ; \text{Move 39 to accumulator (EXIT label is 5 bytes from jump statement)}

Encoded Instruction:

\begin{tabular}{c c c c c}
0 & 0 & 1 & 0 & 0 \\
7 & 0 & 7 & 0 & 7 & 0 & 0
\end{tabular}

Before \hspace{1cm} After

\begin{tabular}{c c}
(39) & (39) \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1
7 & 6 & 0 & 7 & 6 & 0
\end{tabular}

Program Counter

\begin{tabular}{c c c c c c c c}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
1 & 1 & 0 & 1 & 1 & 0 & 0
1 & 5 & 8 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0
\end{tabular}

Notes: \text{10, 11, 12}
Jump and Clear if Bit Is Set

Mnemonic: JBC

Operands: 

- bit address \( 0 \leq \text{bit address} \leq 255 \)
- code address

Format:

\[
\text{JBC} \ bit \ address,code \ address
\]

Bit Pattern:

<table>
<thead>
<tr>
<th>0 0 0 1 0 0 0 0</th>
<th>Bit Address</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation:

\[
(\text{PC}) \leftarrow (\text{PC}) + 3
\]

IF \( \text{bit address} = 1 \) THEN

\[
(\text{bit address}) \leftarrow 0
\]

\[
(\text{PC}) \leftarrow (\text{PC}) + \text{relative offset}
\]

Bytes: 3
Cycles: 2

Flags: 

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PSW</td>
</tr>
</tbody>
</table>

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

ORG 136H
OUT3: INC R7
```

Encoded Instruction:

```
00010000 01110001 01010111
```

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>101110111</td>
<td>101110101</td>
</tr>
</tbody>
</table>

Program Counter

```
00000000 11011100
```

```
00000000 00110110
```

Notes: 10, 11, 12
Jump if Carry Is Set

Mnemonic: JC
Operands: code address
Format: JC code address
Bit Pattern: 01000000 Rel. Offset
              7 0 7 0
Operation: (PC) ← (PC) + 2
         IF (C) = 1
         THEN
                 (PC) ← (PC) + relative code
Bytes: 2
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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:

```
01000000 11001101
```

Before

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000101</td>
<td>000111100</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000101</td>
<td>01010111</td>
</tr>
</tbody>
</table>

Notes: 10, 11, 12
Generic Jump

Mnemonic: JMP
Operands: code address \(0 \leq \text{code address} \leq 65,535\)
Format: \(\text{JMP code address}\)
Bit Pattern: Translated to AJMP, LJMP, or SJMP, as needed
Operation: Either AJMP, SJMP or LJMP
Bytes: 
Cycles: 
Flags: \(C\ AC\ F0\ RS1\ RS0\ OV\ P\)
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: 
\[
\text{JMP SKIP} \quad ; \text{Jump to SKIP} \\
\text{FF: INC A} \quad ; \text{Increment A} \\
\text{SKIP: INC R5} \quad ; \text{Increment register 5}
\]
Encoded Instruction:

<table>
<thead>
<tr>
<th>00000100</th>
<th>00000100</th>
<th>10101011</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Before 
Program Counter 

<table>
<thead>
<tr>
<th>00000100</th>
<th>10100111</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

After 
Program Counter 

<table>
<thead>
<tr>
<th>00000100</th>
<th>10101011</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Notes: None
Jump to Sum of Accumulator and Data Pointer

Mnemonic: JMP
Operands: A Accumulator
          DPTR Data Pointer
Format: JMP @A + DPTR
Bit Pattern:
          01110011
          7 0
Operation: (PC) ← (A) + (DPTR)
Bytes: 1
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P
PSW
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:
          01110011
          7 0
Before                  After
Accumulator        Accumulator
          01110110          01110110
          7 0
Data Pointer         Data Pointer
          00000010 10101000          00000010 10101000
          15 8 7 0
Program Counter      Program Counter
          11001101 00001101          00000011 00011110
          15 8 7 0
Notes: None
Jump if Bit Is Not Set

Mnemonic: JNB

Operands: bit address
code address

Format: JNB bit address,code address

Bit Pattern:

<table>
<thead>
<tr>
<th>0 0 1 1 0 0 0 0</th>
<th>Bit Address</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

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

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:**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110000</td>
<td>01001110</td>
</tr>
<tr>
<td>01010111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(41)</th>
<th>(41)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110111</td>
<td>00110111</td>
</tr>
<tr>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>76</td>
<td>0</td>
</tr>
</tbody>
</table>

**Program Counter**

<table>
<thead>
<tr>
<th>00000000</th>
<th>11011100</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00000000</th>
<th>00110110</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

**Notes:** 10, 11, 12
**Jump if Carry Is Not Set**

**Mnemonic:** JNC

**Operands:** code address

**Format:** JNC code address

**Bit Pattern:**

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 0 0</td>
<td>7 0 7 0</td>
</tr>
</tbody>
</table>

**Operation:**

\[
\text{(PC) } \leftarrow \text{(PC) } + 2 \\
\text{IF (C) } = 0 \\
\text{THEN} \\
\hspace{1cm} \text{(PC) } \leftarrow \text{(PC) } + \text{relative offset}
\]

**Bytes:** 2

**Cycles:** 2

**Flags:** C AC F0 RS1 RS0 OV P

**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:

| 01010000 | 11001101 |

Before

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00011100</td>
</tr>
<tr>
<td></td>
<td>11011100</td>
</tr>
<tr>
<td></td>
<td>15 8 7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Carry Flag</th>
<th>Program Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00011100</td>
</tr>
<tr>
<td></td>
<td>10101011</td>
</tr>
<tr>
<td></td>
<td>15 8 7 0</td>
</tr>
</tbody>
</table>

Notes: 10, 11, 12
Jump if Accumulator Is Not Zero

Mnemonic: JNZ
Operands: code address
Format: JNZ code address
Bit Pattern:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: \((\text{PC}) \leftarrow (\text{PC}) + 2\)
IF \((A) \neq 0\) THEN
\((\text{PC}) \leftarrow (\text{PC}) + \text{relative offset}\)

Bytes: 2
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

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: \textit{JNZ TEST} \textit{; Jump if accumulator is not 0 ; 77 bytes forward}

\textbf{TEST: MOV R3,A}

\textbf{Encoded Instruction:}

\begin{center}
\begin{tabular}{c|c|c|c}
 & 01010000 & 01001101 & \\
7 & 0 & 7 & 0
\end{tabular}
\end{center}

\textbf{Before} \hspace{2cm} \textbf{After}

\begin{center}
\begin{tabular}{c|c|c|c}
\textbf{Accumulator} & 01110111 & 01110111 & \\
7 & 8 & 7 & 0
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c|c|c|c}
\textbf{Program Counter} & 00000000 & 11011100 & 00000001 & 00101011 & \\
15 & 8 & 7 & 0 & 15 & 8 & 7 & 0
\end{tabular}
\end{center}

\textbf{Notes:} 10, 11, 12
Jump if Accumulator Is Zero

Mnemonic: JZ
Operands: code address
Format: JZ code address
Bit Pattern:

<table>
<thead>
<tr>
<th>01100000</th>
<th>Rel. Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: (PC) ← (PC) + 2
IF (A) = 0
THEN
    (PC) ← (PC) + relative offset
Bytes: 2
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

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: \( JZ \) \textit{EMPTY} ; Jump to EMPTY if accumulator is 0

\textit{EMPTY}: \texttt{INC A} ; 25 bytes from JZ instruction

Encoded Instruction:

| 01100000 | 00010111 |
| 7 | 0 | 7 | 0 |

Before

| Accumulator | 01110110 |
| 7 | 0 |

Program Counter

| 00001111 | 11011100 |
| 15 | 8 | 7 | 0 |

After

| Accumulator | 01110110 |
| 7 | 0 |

Program Counter

| 00001111 | 11011110 |
| 15 | 8 | 7 | 0 |

Notes: 10, 11, 12
Long Call

Mnemonic: LCALL
Operands: code address  \(0 \leq \text{code address} \leq 65,535\)
Format: LCALL code address

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Code Addr. high</th>
<th>Code Addr. low</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 0 1 0</td>
<td>7 0 7 0 7 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation:

- \((\text{PC}) \leftarrow (\text{PC}) + 3\)
- \((\text{SP}) \leftarrow (\text{SP}) + 1\)
- \(((\text{SP})) \leftarrow (\text{PC low})\)
- \((\text{SP}) \leftarrow (\text{SP}) + 1\)
- \(((\text{SP})) \leftarrow (\text{PC high})\)
- \((\text{PC}) \leftarrow \text{code address}\)

Bytes: 3
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

PSW

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:

```
00010010 00000010 00110011
7 0 7 0 7 0
```

Before       After

Program Counter  Program Counter

```
10000000 11011100
15 8 7 0
```

```
00000010 00110011
15 8 7 0
```

Stack Pointer  Stack Pointer

```
00101000
7 0
```

```
00101010
7 0
```

(29H)         (29H)

```
01110111
7 0
```

```
11011111
7 0
```

(2AH)         (2AH)

```
00000000
7 0
```

```
10000000
7 0
```

Notes: 1, 2, 3
Long Jump

Mnemonic: LJMP
Operands: \textit{code address} \quad 0 \leq \textit{code address} \leq 65,535
Format: LJMP \textit{code address}
Bit Pattern: 00000010 Code Addr. high | Code Addr. low
\begin{tabular}{|c|c|c|c|c|}
\hline
7 & 0 & 7 & 0 & 7 & 0 \\
\hline
\end{tabular}
Operation: (PC) $\leftarrow$ \textit{code address}
Bytes: 3
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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

Example: \begin{verbatim}
ORG 800H
LJMP FAR ; Jump to FAR
.
.
.
FAR: INC A ; Current code location (8233H)
\end{verbatim}

Encoded Instruction:
\begin{tabular}{cccccc}
00000010 & 10000010 & 00110011 \\
7 & 0 & 7 & 0 & 7 & 0 \\
\end{tabular}

Before \hspace{1cm} After

Program Counter \hspace{1cm} Program Counter
\begin{tabular}{cccc}
00010000 & 00000000 \\
15 & 8 & 7 & 0 \\
\end{tabular} \hspace{1cm} \begin{tabular}{cccc}
10000010 & 00110011 \\
15 & 8 & 7 & 0 \\
\end{tabular}

Notes: None
Move Immediate Data to Indirect Address

Mnemonic: MOV

Operands:

- Rr: Register 0 <= r <= 1
- data: -256 <= data <= +255

Format: MOV @Rr,#data

Bit Pattern:

```
0 1 1 1 0 1 1 r  | Immediate Data
7          0   7  0
```

Operation: ((Rr)) \leftarrow data

Bytes: 2
Cycles: 1

Flags: C, AC, F0, RS1, RS0, OV, P

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:

```
0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1
7   0   7   0
```

Before

<table>
<thead>
<tr>
<th>Register 1</th>
<th>Register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 0 1 1</td>
<td>0 0 0 1 0 0 1 1</td>
</tr>
<tr>
<td>7    0</td>
<td>7    0</td>
</tr>
<tr>
<td>(13H)</td>
<td>(13H)</td>
</tr>
<tr>
<td>0 1 1 1 0 1 1 1</td>
<td>0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>7    0</td>
<td>7    0</td>
</tr>
</tbody>
</table>

Notes: 4, 15
Move Accumulator to Indirect Address

Mnemonic: MOV

Operands: Rr, A

Format: MOV@Rr,A

Bit Pattern: 1111011r

Operation: ((Rr)) ← (A)

Bytes: 1
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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:

Before | After
---|---
Register 0 | Register 0
00111000 | 00111000
((38H)) | ((38H))
10011001 | 01001100
Accumulator | Accumulator
01001100 | 01001100

Note: 15
Move Memory to Indirect Address

Mnemonic: MOV

Operands: 
- \( R_r \) Register \( 0 <= r <= 1 \)
- \( data\ address \) \( 0 <= data\ address <= 255 \)

Format: \( \text{MOV} @R_r, \text{data address} \)

Bit Pattern:

<table>
<thead>
<tr>
<th>( 10100111 )</th>
<th>( r )</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: \( ((R_r)) \leftarrow (\text{data address}) \)

Bytes: 2
Cycles: 2

Flags:

<table>
<thead>
<tr>
<th>( C )</th>
<th>( AC )</th>
<th>( F_0 )</th>
<th>( R_1 )</th>
<th>( R_0 )</th>
<th>( OV )</th>
<th>( P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

PSW

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

Example: \( \text{MOV} @R1,77H \); Move the contents of 77H to indirect address

Encoded Instruction:

<table>
<thead>
<tr>
<th>( 10100111 )</th>
<th>( 01110111 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 00001000 )</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( 08H )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 00110011 )</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( 77H )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 11111110 )</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 00001000 )</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( 08H )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 11111110 )</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( 77H )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 11111110 )</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Notes: 8, 15
Move Immediate Data to Accumulator

Mnemonic: MOV
Operands: A Accumulator
          data $-256 \leq data \leq +255$
Format: MOV A,#data
Bit Pattern:

```
01110100| Immediate Data
  7  0  7  0
```
Operation: (A) ← data
Bytes: 2
Cycles: 1
Flags:

```
C  AC  F0  RS1 RS0  OV  P
```

Description: This instruction moves the 8-bit immediate data value to the accumulator.
Example: MOV A,#01H ; Initialize the accumulator to 1
Encoded Instruction:

```
01110100 00000001
  7  0  7  0
```

Before

```
Accumulator
00100110
  7  0
```

After

```
Accumulator
00000001
  7  0
```

Notes: 4, 5
## MOV

**Move Indirect Address to Accumulator**

- **Mnemonic:** MOV
- **Operands:** A (Accumulator), Rr (Register 0 \(\leq r \leq 1\))
- **Format:** MOV A, @Rr
- **Bit Pattern:**
  
  | 11100111 |
  |  7 0   |

- **Operation:** (A) \(\leftarrow\) ((Rr))

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

- **Flags:**

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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:**
  
  | 11100111 |
  |  7 0   |

### Before and After

<table>
<thead>
<tr>
<th></th>
<th>Accumulator</th>
<th>Register 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulator</td>
<td>10000110</td>
<td>00011100</td>
</tr>
<tr>
<td>Register 1</td>
<td>00011100</td>
<td>00011100</td>
</tr>
</tbody>
</table>

**Notes:** 5, 15
Move Register to Accumulator

Mnemonic: MOV

Operands:

- A: Accumulator
- Rr: Register 0 <= r <= 7

Format: MOV A,Rr

Bit Pattern: 11101rrr

Operation: (A) ← (Rr)

Bytes: 1
Cycles: 1

Flags: C, AC, F0, RS1, RS0, OV, P

PSW

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

Example: MOV A,R6 ; Move R6 to accumulator

Encoded Instruction:

11101110
7 0

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101110</td>
<td>10000101</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000101</td>
<td>10000101</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

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:

```
111 001 01  Data Address
```

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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:

```
111 001 01  100 10000
```

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>001 001 10</td>
<td>011 110 01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Port I (90H)</th>
<th>Port I (90H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>011 110 01</td>
<td>011 110 01</td>
</tr>
</tbody>
</table>

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: 10100010 Bit Address

Operation: (C) ← (bit address)

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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:

<table>
<thead>
<tr>
<th>10100010</th>
<th>10110110</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Before

Port 3 (B0H)

<table>
<thead>
<tr>
<th>00100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
</tr>
</tbody>
</table>

After

Port 3 (B0H)

<table>
<thead>
<tr>
<th>00100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
</tr>
</tbody>
</table>

Notes: None
MOV

Move Immediate Data to Data Pointer

Mnemonic: MOV
Operands: Data Pointer
\( data \) \( 0 \leq data \leq 65,535 \)
Format: MOV DPTR,#data
Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Imm. Data high</th>
<th>Imm. Data low</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0 7</td>
<td>0 7</td>
</tr>
</tbody>
</table>

Operation: \((DPTR) \leftarrow data\)

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: \( MOV \text{ DPTR},#0F4FH \); Initialize the data pointer to 0F4FH

Encoded Instruction:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Imm. Data high</th>
<th>Imm. Data low</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0 7 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Data Pointer</th>
<th>Data Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>11011100</td>
</tr>
<tr>
<td>15 8 7 0</td>
<td>15 8 7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Data Pointer</th>
<th>Data Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011111</td>
<td>01001111</td>
</tr>
<tr>
<td>15 8 7 0</td>
<td>15 8 7 0</td>
</tr>
</tbody>
</table>

Notes: None
Move Immediate Data to Register

Mnemonic: MOV
Operands: Rr
Data
Format: MOV Rr,#data
Bit Pattern:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: (Rr) ← data
Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

PSW

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

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

Encoded Instruction:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before

Register 5

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After

Register 5

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: 4
Move Accumulator to Register

Mnemonic: MOV

Operands: Rr Register 0 <= r <= 7
A Accumulator

Format: MOV Rr,A

Bit Pattern:

\[
\begin{array}{c|c|c|c}
1 & 1 & 1 & 1 \\
1 & & 1 & 1 \\
\end{array}
\]

Operation: (Rr) ← (A)

Bytes: 1
Cycles: 1

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
</table>

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

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

Encoded Instruction:

\[
\begin{array}{c|c|c}
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

Before

<table>
<thead>
<tr>
<th>Register 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>11011100</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Accumulator

| 00111000 |
| 7 0       |

After

<table>
<thead>
<tr>
<th>Register 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Accumulator

| 00111000 |
| 7 0       |

Notes: None
Move Memory to Register

Mnemonic: MOV

Operands: Rr

Format: MOV Rr, data address

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101rrr</td>
<td>Data Address</td>
</tr>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: (Rr) ← (data address)

Bytes: 2

Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

Before

<table>
<thead>
<tr>
<th>Register 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001010</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

(69H)

<table>
<thead>
<tr>
<th>Register 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11011000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Note: 8

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

Encoded Instruction:

10101100 01101001

7 0 7 0

After

<table>
<thead>
<tr>
<th>Register 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11011000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

(69H)

<table>
<thead>
<tr>
<th>Register 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>11011000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Note: 8
MOV

Move Carry Flag to Bit

Mnemonic: MOV
Operands: bit address 0 <= bit address <= 255
C Carry Flag
Format: MOV bit address,C
Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010010</td>
</tr>
</tbody>
</table>

Operation: (bit address) ← (C)
Bytes: 2
Cycles: 2
Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW

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:

```
10010010 01111111
```

Before After

<table>
<thead>
<tr>
<th>2FH</th>
<th>2FH</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>00100110</th>
<th>10100110</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>0111 0101</th>
<th>Data Address</th>
<th>Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0 7 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: (data address) ← data

Bytes: 3
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P  

PSW

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:

<table>
<thead>
<tr>
<th>01110101</th>
<th>10001001</th>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0 7 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before  

<table>
<thead>
<tr>
<th>01110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After  

<table>
<thead>
<tr>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: 4, 9
**MOV**

**Move Indirect Address to Memory**

**Mnemonic:** MOV

**Operands:**
- `data address` \(0 \leq \text{data address} \leq 255\)
- \(Rr\) \(0 \leq r \leq 1\)

**Format:**
\[
\text{MOV data address, } @Rr
\]

**Bit Pattern:**
\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
& 1 & 0 & 0 & 0 & 0 & 1 & 1 & r & & & & & & \\
\hline
7 & 0 & 7 & 0 & & & & & & & & & & & &
\end{array}
\]

**Operation:** \((\text{data address}) \leftarrow (\text{Rr})\)

**Bytes:** 2

**Cycles:** 2

**Flags:**
- \(C\)
- \(AC\)
- \(F0\)
- \(RS1\)
- \(RS0\)
- \(OV\)
- \(P\)

**PSW**

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

**Example:**
\[
\text{MOV 11H, } @R1
\]

; Move indirect address to 11H

**Encoded Instruction:**
\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c}
\hline
& 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
7 & 0 & 7 & 0 & & & & & & & & & & & &
\end{array}
\]

**Before**
- \((11H)\)
- \((58H)\)

**After**
- \((11H)\)
- \((58H)\)

**Register 1**
- \(01011000\)
- \(10010110\)

**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:

```
  1 1 1 1 0 1 0 1
  7 0 7 0 0
Data Address
```

Operation: (data address) ← (A)

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

```

```
PSW

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

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

Encoded Instruction:

```
  1 1 1 1 0 1 0 1 0 1 0 0 1 0 1
  7 0 7 0 0
```

Before

<table>
<thead>
<tr>
<th>(45H)</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 1 0 1</td>
<td>1 0 0 1 1 0 0 1</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>(45H)</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 1 0 0 1</td>
<td>1 0 0 1 1 0 0 1</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

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:

\[
\begin{array}{c|c}
10001 & r r \mid \text{Data Address} \\
7 & 0 & 7 & 0 \\
\end{array}
\]
Operation: (data address) → (Rr)
Bytes: 2
Cycles: 2
Flags:

\[
\begin{array}{cccccccc}
C & AC & F0 & RS1 & RS0 & OV & P \\
\hline
\end{array}
\]
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:

\[
10001011 \ 01111110 \\
7 \ 0 \ 7 \ 0
\]
Before After
(7EH) (7EH)

\[
11110111 \ 10010110 \\
7 \ 0 \ 7 \ 0
\]
Register 3 Register 3

\[
10010110 \ 10010110 \\
7 \ 0 \ 7 \ 0
\]
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:

\[
\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

Operation: (data address1) ← (data address2)
Bytes: 3
Cycles: 2
Flags: C, AC, F0, RS1, RS0, OV, P

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:

\[
\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
\end{array}
\]

Before After
(12H) (12H)

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 \\
\end{array}
\]

Before After
(12H) (F0H)

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

Before After
(F0H) (F0H)

Note: 16
**Mnemonic:** MOVC

**Operands:**
- A: Accumulator
- DPTR: Data Pointer

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

**Bit Pattern:**
10010011

**Operation:**
(A) ← ((A) + (DPTR))

**Bytes:** 1

**Cycles:** 2

**Flags:**
- C
- AC
- F0
- RS1
- RS0
- OV
- P

**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:**

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{C} & \text{AC} & \text{F0} & \text{RS1} & \text{RS0} & \text{OV} & \text{P} \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

**Before**

- **Accumulator:** 00010001
- **Data Pointer:** 00000010 11110001

**After**

- **Accumulator:** 00011110
- **Data Pointer:** 00000010 11110001

**Notes:** 5
Move Code Memory Offset from Program Counter to Accumulator

Mnemonic: MOVC

Operands: A Accumulator
          PC Program Counter

Format: MOVC A, @A + PC

Bit Pattern: 10000011

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

Bytes: 1
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

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: \textit{MOVCA, @A + PC} \quad ; \text{Look up value in table}

Encoded Instruction:

\begin{verbatim}
10000011
\end{verbatim}

Before 
\begin{verbatim}
01110110
\end{verbatim}

After 
\begin{verbatim}
01011000
\end{verbatim}

Before 
\begin{verbatim}
00000010 00110001
\end{verbatim}

After 
\begin{verbatim}
00000010 00110010
\end{verbatim}

\begin{verbatim}
(02A8H) 01011000
\end{verbatim}

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: 11110000
Operation: ((DPTR)) ← (A)
Bytes: 1
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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:

| 11110000 |
| 7 0 |

Before After
Data Pointer Data Pointer
| 00110000 | 00110011 |
| 00110000 | 00110011 |
| 15 8 7 0 | 15 8 7 0 |

(3033H) (3033H)
| 11111001 |
| 01001100 |

Accumulator Accumulator
| 01001100 |
| 01001100 |
| 7 0 | 7 0 |

Notes: None
Move Accumulator to External Memory
Addressed by Register

Mnemonic: MOVX

Operands: Rr Register 0 <= r <= 1
          A Accumulator

Format: MOVX @Rr,A

Bit Pattern: 1111001r

7 0

Operation: ((Rr)) ← (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 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:

```
1 1 1 0 0 0 1 0
```

Before

<table>
<thead>
<tr>
<th>Register 0</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 0</td>
<td>0 1 0 0 1 1 0</td>
</tr>
<tr>
<td>(00B8H)</td>
<td>(00B8H)</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Register 0</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1 0 0</td>
<td>0 1 0 0 1 1 0</td>
</tr>
<tr>
<td>(00B8H)</td>
<td>(00B8H)</td>
</tr>
</tbody>
</table>

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: 11100000
Operation: (A) ← ((DPTR))
Bytes: 1 
Cycles: 2
Flags: 

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

PSW

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:

```
11100000
7 0
```

Before 

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Data Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000110</td>
<td>01110011</td>
</tr>
<tr>
<td>7 0</td>
<td>15 8 7</td>
</tr>
</tbody>
</table>

(73DCH) 

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Data Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101000</td>
<td>01110011</td>
</tr>
<tr>
<td>7 0</td>
<td>15 8 7</td>
</tr>
</tbody>
</table>

Notes: 5
Move External Memory Addressed by Register to Accumulator

Mnemonic: MOVX

Operands: A Accumulator
          Rr Register 0 <= r <= 1

Format: MOVX A, @Rr

Bit Pattern: 1110001r

Operation: (A) ← ((Rr))

Bytes: 1

Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

PSW

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:

11100011
7 0

Before
Accumulator
01010100
7 0
Register 1
00011100
7 0
(551CH)
00001000
7 0

After
Accumulator
00001000
7 0
Register 1
00011100
7 0
(551CH)
00001000
7 0

Notes: 5
Multiply Accumulator by B

Mnemonic: MUL

Operands: AB Multiply/Divide operand

Format: MUL AB

Bit Pattern: 10100100

Operation: (AB) ← (A) * (B)

Bytes: 1
Cycles: 4

Flags:

C  AC  F0  RS1  RS0  OV  P

PSW

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  
      MUL AB  ; Move 10 to multiplication register  
      ; Multiply accumulator by 10

Encoded Instruction:

```
10100100
7 0
```

Before
Accumulator

```
00011111
7 0
```

Multiplication Register (B)

```
00001010
7 0
```

Overflow Flag

```
0
```

Notes: 5

After
Accumulator

```
00110110
7 0
```

Multiplication Register (B)

```
00000001
7 0
```

Overflow Flag

```
1
```
No Operation

Mnemonic: NOP
Operands: None
Format: NOP
Bit Pattern: 00000000
Operation: No operation
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P
Description: This instruction does absolutely nothing for one cycle. Control passes to the next sequential instruction.
Example: NOP ; Pause one cycle
Encoded Instruction:

00000000
7 0
Notes: None
Logical OR Immediate Data to Accumulator

Mnemonic: ORL

Operands: A, data

Format: ORL A,#data

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>01000100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: (A) ← (A) OR data

Bytes: 2
Cycles: 1

Flags:

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

PSW

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:

$\text{ORL A,#00001000B }$; Set bit 3 to 1

Encoded Instruction:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>01000100</td>
</tr>
<tr>
<td>0</td>
<td>00010000</td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110111</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: 4, 5
Logical OR Indirect Address to Accumulator

Mnemonic: ORL
Operands: A to Accumulator
          Rr to Register 0
Format:   ORL A, @Rr
Bit Pattern:
          0100011r
          7 0
Operation: (A) ← (A) OR ((Rr))
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:

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101000</td>
<td>01010010</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

(52H)          (52H)

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101001</td>
<td>01010010</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

(52H)          (52H)

Notes: 5, 15
Logical OR Register to Accumulator

Mnemonic: ORL
Operands: A Accumulator
 Rr Register 0 <= r <= 7
Format: ORL A, Rr
Bit Pattern: 01001 rrr
Operation: (A) ← (A) OR (Rr)
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
Register 4 10001000

After
Accumulator 10011001
Register 4 10001000

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:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000101</td>
<td>0 0 7 0</td>
</tr>
</tbody>
</table>

Operation: (A) ← (A) OR (data address)

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

PSW

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:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>01000101</th>
<th>00110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110111</td>
<td>11110111</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

(37H)

<table>
<thead>
<tr>
<th>(37H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: 5, 8
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:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 0 1 0</td>
<td>7 0 7 0</td>
</tr>
</tbody>
</table>

Operation: (C) ← (C) OR (bit address)
Bytes: 2
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P

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:

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

Before

Carry Flag

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
</table>

(bit address)

| 0 0 1 0 0 1 1 0 |
| 7 2 0 |

After

Carry Flag

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
</table>

(bit address)

| 0 0 1 0 0 1 1 0 |
| 7 2 0 |

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:

<table>
<thead>
<tr>
<th>Bit Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>

Operation: (C) ← (C) OR NOT bit address

Bytes: 2
Cycles: 2

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW

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:

```
1 0 1 0 0 0 0 0 0 1 0 1 1 0 1
7 0 7 0
```

Before

<table>
<thead>
<tr>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

(25H)

```
0 0 0 0 0 1 1 0
7 5 0
```

After

<table>
<thead>
<tr>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

(25H)

```
0 0 0 0 0 1 1 0
7 5 0
```

Notes: None
Logical OR Immediate Data to Memory

Mnemonic: ORL

Operands: 
- data address: \(0 \leq \text{data address} \leq 255\)
- data: \(-256 \leq \text{data} \leq +255\)

Format: ORL data address, #data

Bit Pattern:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01000011</td>
<td>Data Address</td>
<td>Immediate Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Operation: \((\text{data address}) \leftarrow (\text{data address}) \text{ OR data}\)

Bytes: 3

Cycles: 2

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW

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:

<table>
<thead>
<tr>
<th>01000011</th>
<th>01010111</th>
<th>00000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

Before | After
---|---
(57H) | (57H)

<table>
<thead>
<tr>
<th>01110110</th>
<th>01110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

Notes: 4, 9
Logical OR Accumulator to Memory

Mnemonic: ORL

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

Format: ORL data address, A

Bit Pattern:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Operation: (data address) ← (data address) OR A

Bytes: 2
Cycles: 1

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
</tr>
</tbody>
</table>

PSW

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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

(10H)

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>00110001</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

(10H)

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110001</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Note: 9
Pop Stack to Memory

Mnemonic: POP
Operands: data address 0 <= data address <= 255
Format: POP data address
Bit Pattern:  

<table>
<thead>
<tr>
<th>11010000</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Operation: (data address) ← ((SP))
(SP) ← (SP) − 1
Bytes: 2
Cycles: 2
Flags: C  AC  F0  RS1  RS0  OV  P

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>11010000</th>
<th>11010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
Before
Accumulator

<table>
<thead>
<tr>
<th>11010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
PSW (0D0H)

<table>
<thead>
<tr>
<th>10101011</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Stack Pointer

<table>
<thead>
<tr>
<th>00010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
(10H)

<table>
<thead>
<tr>
<th>11110010</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
After
Accumulator

<table>
<thead>
<tr>
<th>11010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
PSW (0D0H)

<table>
<thead>
<tr>
<th>11110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Stack Pointer

<table>
<thead>
<tr>
<th>00001111</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
(10H)

Notes: 2, 8, 17
Push Memory onto Stack

Mnemonic: PUSH
Operands: data address 0 <= data address <= 255
Format: PUSH data address
Bit Pattern:

<table>
<thead>
<tr>
<th>11000000</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

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

Bytes: 2
Cycles: 2

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>11000000</th>
<th>01001101</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Before

(4DH)

<table>
<thead>
<tr>
<th>10101010</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Stack Pointer

<table>
<thead>
<tr>
<th>00010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

(4DH)

<table>
<thead>
<tr>
<th>10101010</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Stack Pointer

<table>
<thead>
<tr>
<th>00010001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: 2, 3, 8
Return from Subroutine (Non-interrupt)

Mnemonic: RET
Operands: None
Format: RET
Bit Pattern: 00100010
Operation: 
(PC high) ← ((SP))
(SP) ← (SP) − 1
(PC low) ← ((SP))
(SP) ← (SP) − 1
Bytes: 1
Cycles: 2
Flags: C AC F0 RS1 RS0 OV P
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: \textit{RET}; Return from subroutine

Encoded Instruction:

\begin{verbatim}
00100010
7 0
\end{verbatim}

Before

\begin{verbatim}
Program Counter
00000100 01010101
15 8 7 0
\end{verbatim}

After

\begin{verbatim}
Program Counter
00000000 01110011
15 8 7 0
\end{verbatim}

Stack Pointer

\begin{verbatim}
Stack Pointer
00001010
7 0
\end{verbatim}

(0AH)

\begin{verbatim}
00000000
7 0
\end{verbatim}

(09H)

\begin{verbatim}
01110011
7 0
\end{verbatim}

Notes: 2, 17
Return from Interrupt Routine

Mnemonic: RETI
Operands: None
Format: RETI
Bit Pattern: 00110010
Operation: 
(\text{PC}_{\text{high}}) \leftarrow ((\text{SP}))
(\text{SP}) \leftarrow (\text{SP}) - 1
(\text{PC}_{\text{low}}) \leftarrow ((\text{SP}))
(\text{SP}) \leftarrow (\text{SP}) - 1
Bytes: 1
Cycles: 2
Flags:
\begin{tabular}{cccccccc}
C & AC & F0 & RS1 & RS0 & OV & P & \text{PSW} \\
\hline
& & & & & & & \\
\end{tabular}

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:**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>Program Counter</td>
</tr>
<tr>
<td>0001010</td>
<td>10101010</td>
</tr>
<tr>
<td>15 8 7 0</td>
<td>15 8 7 0</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>0001010</td>
<td>0001000</td>
</tr>
<tr>
<td>(0AH)</td>
<td>(0AH)</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

**Notes:** 2, 17
Rotate Accumulator Left

Mnemonic: RL
Operands: A Accumulator
Format: RL A
Bit Pattern: 00100011
Operation: C
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P
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 position.
Encoded Instruction:

00100011
7 0
Before
Accumulator

11010000
7 0
After
Accumulator

10100001
7 0
Notes: None
Rotate Accumulator and Carry Flag Left

Mnemonic: RLC
Operands: A
Format: RLC A
Bit Pattern: 00110011
Operation:
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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 position.

Encoded Instruction:

Before
Accumulator 00011001
Carry Flag 1

After
Accumulator 00110011
Carry Flag 0

Note: 5
Rotate Accumulator Right

Mnemonic: RR
Operands: A Accumulator
Format: RR A
Bit Pattern: 00000011
Operation: C ACCUMULATOR
    [Diagram of bit rotation]
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P
        [Flags diagram]
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 position.
Encoded Instruction:
        00000011
Before
Accumulator 11010001
7 0
After
Accumulator 11101000
7 0
Notes: None
Rotate Accumulator and Carry Flag Right

Mnemonic: RRC
Operands: A Accumulator
Format: RRC A
Bit Pattern: 00010011
Operation: C ACCUMULATOR
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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 position.

Encoded Instruction:

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>10011000</td>
<td>1</td>
</tr>
<tr>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>11001100</td>
<td>0</td>
</tr>
<tr>
<td>7 0</td>
<td></td>
</tr>
</tbody>
</table>

Note: 5
**SETB**

Set Carry Flag

Mnemonic: SETB

Operands: C Carry Flag

Format: SETB C

Bit Pattern: 11010011

Operation: (C) ← 1

Bytes: 1

Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

Description: This instruction sets the carry flag to 1.

Example: SETB C ; Set Carry to 1

Encoded Instruction:

11010011

Before After

Carry Flag Carry Flag
0 1

Notes: None
Set Bit

Mnemonic: SETB
Operands: \( bit \text{ address} \ 0 \leq bit \text{ address} \leq 255 \)
Format: SETB \( bit \text{ address} \)
Bit Pattern:

\[
\begin{array}{c|c}
\text{Bit Address} & \text{Bit Pattern} \\
11010010 & 7 \quad 0 \quad 7 \quad 0 \\
\end{array}
\]
Operation: \( (bit \text{ address}) \leftarrow 1 \)
Bytes: 2
Cycles: 1
Flags: 

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
</table>

PSW

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:

\[
\begin{array}{c|c}
11010010 & 01001101 \\
7 \quad 0 \quad 7 \quad 0 \\
\end{array}
\]

Before

\[
\begin{array}{c|c}
01000110 & \text{(41)} \\
7 \quad 5 \quad 0 \\
\end{array}
\]

After

\[
\begin{array}{c|c}
01100110 & \text{(41)} \\
7 \quad 5 \quad 0 \\
\end{array}
\]

Notes: None
Short Jump

Mnemonic: SJMP
Operands: code address
Format: SJMP code address
Bit Pattern:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>0</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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: RRA ; (15 bytes ahead from the INC instruction)

Encoded Instruction:

<table>
<thead>
<tr>
<th>Bit</th>
<th>7</th>
<th>0</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>00001111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before After

Program Counter Program Counter

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101000</td>
<td>11011100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101000</td>
<td>11101101</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Immediate Data

Operation: (A) ← (A) − (C) − data

Bytes: 2
Cycles: 1

Flags:

C AC F0 RS1 RS0 OV P

Before

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Accumulator

Carry Flag

1

Auxiliary Carry Flag

0

Overflow Flag

1

After

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Accumulator

Carry Flag

1

Auxiliary Carry Flag

1

Overflow Flag

0

Notes: 4, 5, 6, 13, 14
SUBB

Subtract Indirect Address from Accumulator with Borrow

Mnemonic: SUBB
Operands: A Accumulator
Rr Register 0 <= r <= 1
Format: SUBB A,@Rr
Bit Pattern: 1001011r
Operation: (A) ← (A) − (C) − ((Rr))
Bytes: 1
Cycles: 1
Flags: C AC F0 R51 R50 OV P

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: \textit{SUBBA, @R1}; Subtract the indirect address from accumulator

Encoded Instruction:

<table>
<thead>
<tr>
<th>encoded instruction</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010111</td>
<td>110000110</td>
<td>10010010</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

Before

Accumulator

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000110</td>
<td>00100100</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

Register 1

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011100</td>
<td>00011100</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

(1CH)

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100010</td>
<td>01100010</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

Carry Flag

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Auxiliary Carry Flag

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Overflow Flag

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Notes: 5, 6, 13, 14, 15
Subtract Register from Accumulator with Borrow

Mnemonic: SUBB
Operands: A Accumulator
          Rr Register 0 <= r <= 7
Format: SUBB A,Rr
Bit Pattern: 10011 rrr
            7 0
Operation: (A) ← (A) - (C) - (Rr)
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P
       ● ● ● ● ● ●
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:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10011110</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Before**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01110110</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>10000101</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Carry Flag</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auxiliary Carry Flag</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overflow Flag</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**After**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11100000</td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>10000101</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Carry Flag</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auxiliary Carry Flag</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overflow Flag</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:** 5, 6, 13, 14
Subtract Memory from Accumulator with Borrow

Mnemonic: SUBB
Operands: A Accumulator
data address 0 <= data address <= 255
Format: SUBB A, data address
Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern:</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: (A) ← (A) − (C) − (data address)
Bytes: 2
Cycles: 1
Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:  \textit{SUBB A,32H} \quad ; \text{Subtract 32H in memory from accumulator}

Encoded Instruction:

\[
\begin{array}{cccc}
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

Before

\begin{align*}
\text{Accumulator} & : 00100110 \\
(32H) & : 01010011 \\
\text{Carry Flag} & : 1 \\
\text{Auxiliary Carry Flag} & : 0 \\
\text{Overflow Flag} & : 0 \\
\end{align*}

After

\begin{align*}
\text{Accumulator} & : 11010010 \\
(32H) & : 01010011 \\
\text{Carry Flag} & : 1 \\
\text{Auxiliary Carry Flag} & : 1 \\
\text{Overflow Flag} & : 0 \\
\end{align*}

Notes: 5, 6, 8, 13, 14
SWAP

Exchange Nibbles in Accumulator

Mnemonic: SWAP
Operands: A Accumulator
Format: SWAP A
Bit Pattern: 11000100
Operation: \[ \begin{array}{c}
\text{h h h h}
\end{array} \]
Bytes: 1
Cycles: 1
Flags: C AC F0 RS0 RS1 OV P

Description: This instruction exchanges the contents of the low order nibble (0-3) with the contents of the high order nibble (4-7).

Example: \textit{SWAP A} ; Swap high and low nibbles in the accumulator.

Encoded Instruction:

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11010000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001101</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

Notes: None
Exchange Indirect Address with Accumulator

Mnemonic: XCH

Operands:
- A                     Accumulator
- Rr                  Register 0 \( \leq r \leq 1 \)

Format: XCH A, @Rr

Bit Pattern:

\[
\begin{array}{cccc}
& & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & r \\
& 7 & 0
\end{array}
\]

Operation:
\[
temp \leftarrow ((Rr)) \\
((Rr)) \leftarrow (A) \\
(A) \leftarrow temp
\]

Bytes: 1
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

Description: This instruction exchanges the contents of the memory location addressed by the contents of register \( r \) with the contents of the accumulator.

Example: \textit{XCH A, @R0} ; Exchange the accumulator with memory

Encoded Instruction:

\[
\begin{array}{cccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
7 & 0
\end{array}
\]

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111111</td>
<td>01010010</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
<tr>
<td>(52H)</td>
<td>(52H)</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00011101</td>
<td>01010010</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
<tr>
<td>(52H)</td>
<td>(52H)</td>
</tr>
</tbody>
</table>

Notes: 5, 15
Exchange Register with Accumulator

Mnemonic: XCH
Operands: A Accumulator
          Rr Register 0 <= r <= 7
Format:  XCH A,Rr
Bit Pattern: 11001 r r r
             7 0
Operation: temp ← (Rr)
           (Rr) ← (A)
           (A) ← temp
Bytes:  1
Cycles: 1
Flags:  C AC F0 RS1 RS0 OV P
PSW
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
Accumulator  10011001  7 0
Register 6   10000000  7 0

After
Accumulator  10000000  7 0
Register 6   10011001  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:

```
1 1 0 0 0 1 0 1
7 0 7 0
```
Operation: temp ← (data address)
(data address) ← (A)
(A) ← temp
Bytes: 2
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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:

```
1 1 0 0 0 1 0 1 0 0 1 1 0 1 1 1
7 0 7 0
```

Before
Accumulator: 01110111
(data address) 11110000

After
Accumulator: 11110000
(data address) 01110111

Notes: 5, 9
Exchange Low Nibbles (Digits) of Indirect Address with Accumulator

Mnemonic: XCHD

Operands: A Accumulator
          Rr Register 0 <= r <= 1

Format: XCHD A,@Rr

Bit Pattern:

```
11 0 1 0 1 1 r
```

Operation:

- temp ← ((Rr)) 0-3
- ((Rr)) 0-3 ← (A) 0-3
- (A) 0-3 ← temp

Bytes: 1
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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:**

<table>
<thead>
<tr>
<th>Binary Code</th>
<th>Accumulator</th>
<th>Register 0</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>110101110</td>
<td>00111111</td>
<td>01010010</td>
<td>5, 15</td>
</tr>
</tbody>
</table>

**Before**

- Accumulator: \( 00\text{H} \)
- Register 0: \( 01\text{H} \)

**After**

- Accumulator: \( 01\text{H} \)
- Register 0: \( 00\text{H} \)
Logical Exclusive OR Immediate Data to Accumulator

Mnemonic: XRL
Operands: A Accumulator
data -256 <= data <= +255
Format: XRL A,#data
Bit Pattern: 011 001 00 Immediate Data
7 0 7 0
Operation: (A) ← (A) XOR data
Bytes: 2
Cycles: 1
Flags: C AC F0 RS1 RS0 OV P

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,#OFH ; Complement the low order nibble
Encoded Instruction:

Before
Accumulator 01110111 7 0

After
Accumulator 01111000 7 0

Notes: 4, 5
Logical Exclusive OR Indirect Address to Accumulator

Mnemonic: XRL
Operands: A  Accumulator
          Rr  0 <= Rr <= 1
Format:   XRL A,@Rr
Bit Pattern:

```
0110011r
  7  0
```
Operation: \((A) \leftarrow (A) \text{ XOR } ((Rr))\)
Bytes: 1
Cycles: 1
Flags: C AC F0 RS1 RS0 OV 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:

```
01100110
  7  0
```
Before

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 0</th>
<th>(52H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101000</td>
<td>01010010</td>
<td>00000001</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Register 0</th>
<th>(52H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101001</td>
<td>01010010</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Notes: 5, 15
Logical Exclusive OR Register to Accumulator

Mnemonic: XRL

Operands: A Accumulator  
Rr Register $0 \leq r \leq 7$

Format: XRL A, Rr

Bit Pattern: \[
\begin{array}{c}
0 & 1 & 1 & 0 & 1 & r & r & r \\
7 & 0
\end{array}
\]

Operation: \((A) \leftarrow (A) \text{ XOR } (Rr)\)

Bytes: 1
Cycles: 1

Flags:

<table>
<thead>
<tr>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PSW

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:

\[
\begin{array}{c}
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
7 & 0
\end{array}
\]

Before

Accumulator

\[
\begin{array}{c}
1 & 0 & 0 & 1 & 0 & 0 & 1 \\
7 & 0
\end{array}
\]

Register 4

\[
\begin{array}{c}
1 & 1 & 1 & 0 & 0 & 0 & 1 \\
7 & 0
\end{array}
\]

After

Accumulator

\[
\begin{array}{c}
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
7 & 0
\end{array}
\]

Register 4

\[
\begin{array}{c}
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
7 & 0
\end{array}
\]

Note: 5
**Logical Exclusive OR Memory to Accumulator**

**Mnemonic:** XRL

**Operands:**
- A: Accumulator
- data address: $0 \leq \text{data address} \leq 255$

**Format:**
- XRL A, data address

**Bit Pattern:**

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100101</td>
<td>0 7 0 7 0</td>
</tr>
</tbody>
</table>

**Operation:**
- $(A) \leftarrow (A) \text{ XOR } (\text{data address})$

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

**Flags:**

<table>
<thead>
<tr>
<th>Flag</th>
<th>C</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>01100101</th>
<th>00110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0 0</td>
<td></td>
</tr>
</tbody>
</table>

**Before**

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111</td>
<td>11110111</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

**After**

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110111</td>
<td>11110111</td>
</tr>
<tr>
<td>7 0</td>
<td>7 0</td>
</tr>
</tbody>
</table>

**Notes:** 4, 8
Logical Exclusive OR Immediate Data to Memory

Mnemonic: XRL

Operands: 
- \( data \text{ address} \) \( 0 \leq data \text{ address} \leq 255 \)
- \( data \) \( -256 \leq data \leq 255 \)

Format: \( \text{XRL data address,} \#\text{data} \)

Bit Pattern:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Data Address</th>
<th>Immediate Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 0 1 1</td>
<td>0 7</td>
<td>0 7</td>
</tr>
</tbody>
</table>

Operation: \((data \text{ address}) \leftarrow (data \text{ address}) \text{ XOR data}\)

Bytes: 3
Cycles: 2

Flags: C AC F0 RS1 RS0 OV P

| Flags | | | | | | |
|-------|---|---|---|---|---|

PSW

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: \( \text{XRL P1,} \#51\text{H} \); XOR 51H with the contents of Port 1

Encoded Instruction:

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

Before

Port 1 (90H)

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

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:

<table>
<thead>
<tr>
<th>01100010</th>
<th>Data Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Operation: (data address) ← (data address) XOR A

Bytes: 2
Cycles: 1

Flags: C AC F0 RS1 RS0 OV P

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:

<table>
<thead>
<tr>
<th>01100010</th>
<th>00010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 7 0</td>
<td></td>
</tr>
</tbody>
</table>

Before

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

(10H)

<table>
<thead>
<tr>
<th>00110001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110000</td>
</tr>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

(10H)

<table>
<thead>
<tr>
<th>11000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0</td>
</tr>
</tbody>
</table>

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 invalid data will be returned on a POP or return.
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:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
?
```

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**

<table>
<thead>
<tr>
<th>ACALL</th>
<th>AD</th>
<th>JMP</th>
<th>MOV</th>
<th>PUSH</th>
<th>SETB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDC</td>
<td>DIV</td>
<td>JNC</td>
<td>MOVX</td>
<td>RETI</td>
<td>SUBB</td>
</tr>
<tr>
<td>AJMP</td>
<td>DJNZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL</td>
<td>INC</td>
<td>JZ</td>
<td>NOP</td>
<td>RLC</td>
<td>XCH</td>
</tr>
<tr>
<td>CJNE</td>
<td>JB</td>
<td>LCALL</td>
<td>ORL</td>
<td>RR</td>
<td>XCHD</td>
</tr>
<tr>
<td>CLR</td>
<td>JBC</td>
<td>LJMP</td>
<td>POP</td>
<td>RRC</td>
<td>XRL</td>
</tr>
<tr>
<td>CPL</td>
<td>JC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
Assembler Directives

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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>EQU Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUM</td>
<td>A</td>
<td>Define ACCUM to stand for A (accumulator)</td>
</tr>
<tr>
<td>N27</td>
<td>27</td>
<td>Set N27 to equal 27</td>
</tr>
<tr>
<td>PI</td>
<td>3</td>
<td>This program was written in Louisiana</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where PI equals 3 by law</td>
</tr>
<tr>
<td>HERE</td>
<td>$</td>
<td>Set HERE to current location counter value</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>SET Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>SET 0</td>
<td>Initialize Assembly-time counter</td>
</tr>
<tr>
<td>COUNT</td>
<td>SET COUNT +1</td>
<td>Increment assembly time counter</td>
</tr>
<tr>
<td>HALF</td>
<td>SET WHOLE/2</td>
<td>Give half of WHOLE to HALF (the remainder is discarded)</td>
</tr>
<tr>
<td>H20</td>
<td>SET 32</td>
<td>Set H20 to 32</td>
</tr>
<tr>
<td>INDIRECT</td>
<td>SET R1</td>
<td>Set INDIRECT to R1</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>DATA Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONIN</td>
<td>DATA SBUF</td>
<td>Define CONIN to address the serial port buffer</td>
</tr>
<tr>
<td>TABLE__BASE</td>
<td>DATA 70H</td>
<td>Define TABLE__BASE to be at location 70H</td>
</tr>
<tr>
<td>TABLE__END</td>
<td>DATA 7FH</td>
<td>Define TABLE__END to be at top of RAM (7FH)</td>
</tr>
</tbody>
</table>
XDATA Directive

The format for the XDATA directive is shown below. Note that a label is not permitted.

\[ \text{symbol name} \quad \text{XDATA} \quad \text{expression} \]

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:

\[
\begin{align*}
\text{DATE} & \quad \text{XDATA} \quad 999H \quad ; \quad \text{Define DATE to be 999H} \\
\text{TIME} & \quad \text{XDATA} \quad \text{DATE}+5 \quad ; \quad \text{Define TIME to be 5 bytes after DATE} \\
\text{PLACE} & \quad \text{XDATA} \quad \text{TIME}+3 \quad ; \quad \text{Define PLACE to be 3 bytes after TIME}
\end{align*}
\]

BIT Directive

The format for the BIT directive is shown below. Note that a label is not permitted.

\[ \text{symbol name} \quad \text{BIT} \quad \text{bit address} \]

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:

\[
\begin{align*}
\text{ERROR\_FLAG} & \quad \text{BIT} \quad 25H.3 \quad ; \quad \text{Define ERROR\_FLAG in RAM} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \]}
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 active. 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:

\[
\text{ORG expression}
\]

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:

\[
\text{[label:] DS expression}
\]

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:

\[
\text{[label:] DBIT expression}
\]

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

Besides 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:**

```
ARRIVALS:  DV  710, 'AM', 943,'AM', 1153,'AM', 315,'PM', 941,'PM'
```

This OW lists several flight arrivals, the numbers and characters are encoded consecutively.

**INVENTORY:**

```
INVENTORY: DV '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
```

```
DV $, $-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.
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 0110.

The following scheme illustrates these concepts:

1. Your source file as seen by the Macro Processor:

   \[
   \begin{array}{c}
   \text{plaintext} \quad \text{macro-def} \quad \text{macro-call} \quad \text{plaintext} \\
   \end{array}
   \]

2. An internal, intermediate form after the macro-definition is stored:

   \[
   \begin{array}{c}
   \text{plaintext} \quad \text{macro-call} \quad \text{plaintext} \\
   \end{array}
   \]

   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:

   \[
   \begin{array}{c}
   \text{plaintext} \quad \text{86H} \quad \text{plaintext} \\
   \text{plaintext} \quad \text{0000 0000 1000 0110B} \quad \text{plaintext} \\
   \end{array}
   \]

4. The resulting macro expansion then becomes input to the assembler-proper:

   \[
   \begin{array}{c}
   \text{plaintext} \\
   \end{array}
   \]

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: O%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:

```plaintext
%*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:

  ```plaintext
  %*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:

```plaintext
%*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:

```plaintext
%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:
  ```plaintext
  %*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:
  ```plaintext
  %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:

\[
\text{%DEFINE (YELLOWSTUFF) (MUSTARD)}
\]

Then you could call HAMBURGER as follows:

\[
\text{%HAMBURGER(CATSUP,%YELLOWSTUFF,ONION,PICKLE,LETTUCE)}
\]

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:

```c
%'DEFINE (ADD16(XHIGH, XLOW, YHIGH, YLOW, SUMHIGH, SUMLOW)) (  
  MOV A,%XLOW
  ADD A,%YLOW
  MOV %SUMLOW,A
  MOV A, %XHIGH
  ADDCA,%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:

```c
%'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:

```c
%'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:

```c
%ADD16(#(HIGH 60134),(#LOW 60134),56H,55H,41H,40)
%ADD16(DPH,DP,7,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(ABCDEFGHIJKLMNOPQRSTUVWXYZ) is the ASCII string OAH.

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: %EQL(%VALUE1, 3) (compare strings)

Arithmetic expressions allow the following operators, in high-to-low order of precedence:

Parenthesized Expressions

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:

<table>
<thead>
<tr>
<th>MPL Function</th>
<th>Answers the Question</th>
<th>With One Of</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQS</td>
<td>Are the strings lexically equal?</td>
<td>0FFFFH (Yes), 00H (No)</td>
</tr>
<tr>
<td>NES</td>
<td>Are the strings lexically unequal?</td>
<td>0FFFFH (Yes), 00H (No)</td>
</tr>
<tr>
<td>LTS</td>
<td>Does the first precede the second in their dictionary ordering?</td>
<td>0FFFFH (Yes), 00H (No)</td>
</tr>
<tr>
<td>LES</td>
<td>Does the first precede or equal the second in their dictionary ordering?</td>
<td>0FFFFH (Yes), 00H (No)</td>
</tr>
<tr>
<td>GES</td>
<td>Does the first follow or equal the second in their dictionary ordering?</td>
<td>0FFFFH (Yes), 00H (No)</td>
</tr>
<tr>
<td>GTS</td>
<td>Does the first follow the second in their dictionary ordering?</td>
<td>0FFFFH (Yes), 00H (No)</td>
</tr>
</tbody>
</table>
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 (0OH), 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:

\%EOS( 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 is 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:

\[ \text{\texttt{NI}}\%
\]

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:

\[ \text{\texttt{%OUT(string)}}\]

where \textit{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 >.

\[
\begin{align*}
\text{%*DEFINE (REC(F)) LOCAL RECORDNAME (} \\
\text{%RECORDNAME RECORD %ITEM %REPEAT (%F-1), %ITEM) \\
\text{%ARRAYNAME %RECORDNAME %EVAL(%NUMREC) DUP (<>)} \\
\text{)} \\
\text{%*DEFINE (ITEM) (%FLDNAME : %FLDWIDTH = %FLDVAL) } \\
\text{%*DEFINE (FLDNAME) (%OUT(NAME OF FIELD?) %GET) } \\
\text{%*DEFINE (FLDWIDTH) (%OUT(WIDTH OF FIELD?) %GET) } \\
\text{%*DEFINE (FLDVAL) (%OUT(INITIAL VALUE OF FIELD?) %GET) } \\
\text{%*DEFINE (ARRAYNAME) (%OUT(NAME OF RECORD ARRAY?) %GET) } \\
\text{%*DEFINE (NUMREC) (%OUT(NUMBER OF RECORDS IN ARRAY?) %GET) } \\
\text{%*DEFINE (GET) (%MATCH (LINE %) } \\
\text{)} NULL) (%IN) %LINE) \\
\text{%REC(3)}
\end{align*}
\]

If you want five fields instead, for example, change the call from \texttt{%REC(3)} to \texttt{%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 \texttt{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:

\[ \text{\texttt{%SET(name, value)}}\]

where:

\textit{name} is an MPL identifier
\textit{value} is an expression acceptable to \texttt{EVAL}

For instance

\[
\begin{align*}
\text{%SET(LINES,10)} \\
\text{%SET(MAX,80-%LEN(%STRING))} \\
\text{%SET(CHARS,%MAX*%LINES)}
\end{align*}
\]
You can use SET to redefine the same macro-time variable. For example,

```
%SET(LINES,10)
  0
  0
  0
%SET(LINES,15)
  0
  0
  0
%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
```
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:

[:Fn:]ASM51 [:Fn:]sourcefile[.extension] [controls]

You must specify the filename of the assembler ([:Fn:]ASM51) and the filename of your source code ([:Fn:]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 your 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(COMPLET PROJECT REV. 3.0) & Comment
** GEN
Assembler Operation and 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>
<thead>
<tr>
<th>Name</th>
<th>Primary/General</th>
<th>Default</th>
<th>Abbrev.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE(date)</td>
<td>P</td>
<td>DATE( )</td>
<td>DA</td>
<td>Places string in header (max 9 characters)</td>
</tr>
<tr>
<td>DEBUG</td>
<td>P</td>
<td>NODEBUG</td>
<td>DB</td>
<td>Outputs debug symbol information to object file</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>P</td>
<td></td>
<td>NODB</td>
<td>Symbol information not placed in object file</td>
</tr>
<tr>
<td>EJECT</td>
<td>G</td>
<td>Not Applicable</td>
<td>EJ</td>
<td>Continue listing on next page</td>
</tr>
<tr>
<td>ERRORPRINT[(FILE)]</td>
<td>P</td>
<td>NOERRORPRINT</td>
<td>EP</td>
<td>Designates a file to receive error messages in addition to the listing file</td>
</tr>
<tr>
<td>NOERRORPRINT</td>
<td>P</td>
<td></td>
<td>NOEP</td>
<td>Designates that error messages will be printed in listing file only</td>
</tr>
<tr>
<td>GEN</td>
<td>G</td>
<td>GENONLY</td>
<td>GE</td>
<td>Generates a full listing of the macro expansion process including macro calls in the listing file</td>
</tr>
<tr>
<td>GENONLY</td>
<td>G</td>
<td></td>
<td>GO</td>
<td>List only the fully expanded source as if all lines generated by a macro call were already in source file</td>
</tr>
<tr>
<td>NOGEN</td>
<td>G</td>
<td></td>
<td>NOGE</td>
<td>List only the original source text in listing file</td>
</tr>
<tr>
<td>Name</td>
<td>Primary/General</td>
<td>Default</td>
<td>Abbrev.</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>---------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INCLUDE(FILE)</td>
<td>G</td>
<td>Not Applicable</td>
<td>IC</td>
<td>Designates a file to be included as part of the program</td>
</tr>
<tr>
<td>LIST</td>
<td>G</td>
<td>LIST</td>
<td>LI</td>
<td>Print subsequent lines of source in listing file</td>
</tr>
<tr>
<td>Nolist</td>
<td>G</td>
<td>NOLIST</td>
<td>NOLI</td>
<td>Do not print subsequent lines of source in listing file</td>
</tr>
<tr>
<td>Macro</td>
<td>P</td>
<td>MACRO</td>
<td>MR</td>
<td>Evaluate and expand all macro calls</td>
</tr>
<tr>
<td>NOMacro</td>
<td>P</td>
<td>NOMR</td>
<td></td>
<td>Do not evaluate macro calls</td>
</tr>
<tr>
<td>Object(FILE)</td>
<td>P</td>
<td>OBJECT(source.HEX)</td>
<td>OJ</td>
<td>Designate file to receive object code</td>
</tr>
<tr>
<td>Noobject</td>
<td>P</td>
<td>NOOJ</td>
<td></td>
<td>Designates that no object file will be created</td>
</tr>
<tr>
<td>PAGING</td>
<td>P</td>
<td>PAGING</td>
<td>PI</td>
<td>Designates that listing will be broken into pages and each will have a header</td>
</tr>
<tr>
<td>NOPAGING</td>
<td>P</td>
<td>NOPI</td>
<td></td>
<td>Designates that listing will contain no page breaks</td>
</tr>
<tr>
<td>PAGELENGTH(n)</td>
<td>P</td>
<td>PAGELENGTH(60)</td>
<td>PL</td>
<td>Sets maximum number of lines in each page of listing file (maximum = 65,535; minimum = 10)</td>
</tr>
<tr>
<td>PAGEWIDTH(n)</td>
<td>P</td>
<td>PAGEWIDTH(120)</td>
<td>PW</td>
<td>Sets maximum number of characters in each line of listing file (maximum = 132; minimum = 72)</td>
</tr>
<tr>
<td>PRINT(FILE)</td>
<td>P</td>
<td>PRINT(source.LST)</td>
<td>PR</td>
<td>Designates file to receive source listing</td>
</tr>
<tr>
<td>NOPRINT</td>
<td>P</td>
<td>NOPR</td>
<td></td>
<td>Designates that no listing file will be created</td>
</tr>
<tr>
<td>Save</td>
<td>G</td>
<td>Not Applicable</td>
<td>SA</td>
<td>Stores current control setting for LIST and GEN</td>
</tr>
<tr>
<td>Restore</td>
<td>G</td>
<td>RS</td>
<td></td>
<td>Restores control setting from SAVE stack</td>
</tr>
<tr>
<td>Symbols</td>
<td>P</td>
<td>NOSYMBOLES</td>
<td>SB</td>
<td>Creates a formatted table of all symbols used in program</td>
</tr>
<tr>
<td>NOSymbols</td>
<td>P</td>
<td>NOSB</td>
<td></td>
<td>No symbol table created</td>
</tr>
<tr>
<td>Title(string)</td>
<td>G</td>
<td>TITLE()</td>
<td>TT</td>
<td>Places a string in all subsequent page headers (maximum 80 characters)</td>
</tr>
<tr>
<td>Workfiles(Fn;[;Fm:])</td>
<td>P</td>
<td>same drive as source file</td>
<td>WF</td>
<td>Designates alternate drives for temporary workfiles</td>
</tr>
<tr>
<td>Xref</td>
<td>P</td>
<td>NOXREF</td>
<td>XR</td>
<td>Creates a cross reference listing of all symbols used in program</td>
</tr>
<tr>
<td>Noxref</td>
<td>P</td>
<td>NOXR</td>
<td></td>
<td>No cross reference list created</td>
</tr>
</tbody>
</table>
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 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 ASM511.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
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 andpagewidth 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:

```plaintext
*** ERROR #eee, LINE #III (ppp), message
```

where:

- `eee` is the error number
- `III` 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.

*** 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:

POPA
DB R0
JZA

*** 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 and 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
- SAVE
- RESTORE
- LIST
- NOLIST
- GENONLY
- GEN
- NOOBJECT
- NPRINT
- NOPAGING
- DEBUG
- NODEBUG
- NOERRORPRINT
- NOOBJECT
- NOMACRO
- NOMACRO
- NOSYMBOLS
- PAGING
- SYMBOLS
- XREF
- 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 page length.

*** 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.

![Figure 7-1. Example Listing File Format](image-url)
Figure 7-1. Example Listing File Format (Cont’d.)
Figure 7-1. Example Listing File Format (Cont’d.)
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.

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.
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.

Figure 7-4. Examples of Macro Listing Modes
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 '+'.

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>=1</td>
<td>101</td>
<td>+1</td>
<td>$ SAVE NOLIST</td>
</tr>
</tbody>
</table>

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 where the symbol was used in the program.

```
XREF SYMBOL TABLE LISTING
-- NAME -- TYPE -- VALUE AND REFERENCES
BCDBIN. .... L CSEG 0103H 88#
DBDBIN. .... L CSEG 0172H 11 10#
CONFRT. .... L CSEG 0100H 86#
CONV. ...... L CSEG 0100H 116#
DPL. ...... N DSEG 0155H 105# 128#
PFR. ...... N DSEG 0082H 13 17 27 29 39 41
SIE. ....... - ____ _
ERRORS .... --UNDEFINED--
JOHN. ....... N
JOHN_JOHN ... L BSEG 0000H 54#
MEMTOP .... L DSEG 0000H 54#
SECOND_ERROR ... --UNDEFINED--
STORE ....... N
TYPE_STRING . L CSEG 0000H 61#
```

Figure 7-5. Example Symbol Table Listing

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.
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 | ERRORMPRINT(<Filename>) | EP(<Filename>) | ERRORPRINT | EP |
             NOERRORPRINT | NOEP |
             GENONLY | GO |
             NOGEN | NOGE |
             GEN | GE |
             INCLUDE(<Filename>) | IC(<Filename>) |
             LIST | LJ |
             NOLIST | NOLJ |
             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> ::= ANLC, <Bit Address> | ANLC, I<Bit Address> | ORLC, <Bit Address> | ORLC, I<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>, <Immediate Data> | MOV <Data Address>, <Register Move>

_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>
Assembly Language BNF Grammar

```
<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>

<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
```

A-5
Assembly Language BNF Grammar

<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><Operator><Expression><Operator><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>

<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
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**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Accumulator</td>
</tr>
<tr>
<td>AB</td>
<td>Register Pair</td>
</tr>
<tr>
<td>B</td>
<td>Multiplication Register</td>
</tr>
<tr>
<td>bit address</td>
<td>8051 bit address</td>
</tr>
<tr>
<td>page address</td>
<td>11-bit code address within 2K page</td>
</tr>
<tr>
<td>relative offset</td>
<td>8-bit 2’s complement offset</td>
</tr>
<tr>
<td>C</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>code address</td>
<td>Absolute code address</td>
</tr>
<tr>
<td>data</td>
<td>Immediate data</td>
</tr>
<tr>
<td>data address</td>
<td>On-chip 8-bit RAM address</td>
</tr>
<tr>
<td>DPTR</td>
<td>Data pointer</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Rr</td>
<td>Register (r=0-7)</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>high</td>
<td>High order byte</td>
</tr>
<tr>
<td>low</td>
<td>Low order byte</td>
</tr>
<tr>
<td>i-j</td>
<td>Bits i through j</td>
</tr>
<tr>
<td>.n</td>
<td>Bit n</td>
</tr>
<tr>
<td>aaa aaaaaaa</td>
<td>Absolute page address encoded in instruction and operand byte</td>
</tr>
<tr>
<td>bbbbbbbb</td>
<td>Bit address encoded in operand byte</td>
</tr>
<tr>
<td>dddddddd</td>
<td>Immediate data encoded in operand byte</td>
</tr>
<tr>
<td>11111111</td>
<td>One byte of a 16-bit address encoded in operand byte</td>
</tr>
<tr>
<td>mmmmmmmm</td>
<td>Data address encoded in operand byte</td>
</tr>
<tr>
<td>oooooooo</td>
<td>Relative offset encoded in operand byte</td>
</tr>
<tr>
<td>r or rrr</td>
<td>Register identifier encoded in operand byte</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical complement</td>
</tr>
<tr>
<td>OR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>−</td>
<td>Minus</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>(X)</td>
<td>The contents of X</td>
</tr>
<tr>
<td>((X))</td>
<td>The memory location addressed by (X) (The contents of X)</td>
</tr>
<tr>
<td>=</td>
<td>Is equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Is not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Is less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Is greater than</td>
</tr>
<tr>
<td>←</td>
<td>Is replaced by</td>
</tr>
<tr>
<td>Instruction</td>
<td>Operation</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>ACALL</td>
<td>code addr (PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP)) ← (PC) low (SP) ← (SP) + 1 ((SP)) ← (PC) high (PC) 0-10 ← page address</td>
</tr>
<tr>
<td>ADD A, #data</td>
<td>(A) ← (A) + data</td>
</tr>
<tr>
<td>ADD A, (Rr)</td>
<td>(A) ← (A) + ((Rr))</td>
</tr>
<tr>
<td>ADD A, #data</td>
<td>(A) ← (A) + (data address)</td>
</tr>
<tr>
<td>ADDC A, #data</td>
<td>(A) ← (A) + (C) + data</td>
</tr>
<tr>
<td>ADDC A, (Rr)</td>
<td>(A) ← (A) + (C) + ((Rr))</td>
</tr>
<tr>
<td>ADDC A, (Rr)</td>
<td>(A) ← (A) + (C) + (Rr)</td>
</tr>
<tr>
<td>ADDC A, #data</td>
<td>(A) ← (A) + (C) + (data address)</td>
</tr>
<tr>
<td>AJMP</td>
<td>code addr (PC) ← code address</td>
</tr>
<tr>
<td>ANL A, #data</td>
<td>(A) ← (A) AND data</td>
</tr>
<tr>
<td>ANL A, (Rr)</td>
<td>(A) ← (A) AND ((Rr))</td>
</tr>
<tr>
<td>ANL A, (Rr)</td>
<td>(A) ← (A) AND (Rr)</td>
</tr>
<tr>
<td>ANL A, #data</td>
<td>(A) ← (A) AND (data address)</td>
</tr>
<tr>
<td>ANL C, bit addr</td>
<td>(C) ← (C) AND (bit address)</td>
</tr>
<tr>
<td>ANL C, bit addr</td>
<td>(C) ← (C) AND NOT (bit address)</td>
</tr>
<tr>
<td>ANL data addr, #data</td>
<td>(data address) ← (data address) AND data</td>
</tr>
<tr>
<td>ANL data addr, A</td>
<td>(data address) ← (data address) AND A</td>
</tr>
<tr>
<td>Mnemonic Operation</td>
<td>Cycles</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
</tr>
<tr>
<td>CJNE @Rr,#data,code addr (PC) ← (PC) + 3 IF ((Rr) &lt; &gt; data THEN (PC) ← (PC) + relative offset IF ((Rr) &lt; &gt; data THEN (C) ← 1 ELSE (C) ← 0 2 1 0 1 1 0 1 1 r d d d d d d d d 0 0 0 0 0 0 0 0 C If immediate data and contents of indirect address are not equal, jump to code address</td>
<td></td>
</tr>
<tr>
<td>CJNE A,#data,code addr (PC) ← (PC) + 3 IF (A) &lt; &gt; data THEN (PC) ← (PC) + relative offset IF (A) &lt; &gt; data THEN (C) ← 1 ELSE (C) ← 0 2 1 0 1 1 0 1 0 0 d d d d d d d d 0 0 0 0 0 0 0 C If immediate data and A are not equal, jump to code address</td>
<td></td>
</tr>
<tr>
<td>CJNE A,#data,code addr (PC) ← (PC) + 3 IF (A) &lt; &gt; (data address) THEN (PC) ← (PC) + relative offset IF (A) &lt; &gt; (data address) THEN (C) ← 1 ELSE (C) ← 0 2 1 0 1 1 0 1 0 1 m m m m m m m m 0 0 0 0 0 0 0 C If contents of data address and A are not equal, jump to code address</td>
<td></td>
</tr>
<tr>
<td>CJNE Rr,#data,code addr (PC) ← (PC) + 3 IF (Rr) &lt; &gt; data THEN (PC) ← (PC) + relative offset IF (Rr) &lt; &gt; data THEN (C) ← 1 ELSE (C) ← 0 2 1 0 1 1 1 r r r d d d d d d d d 0 0 0 0 0 0 0 C If immediate data and register are not equal, jump to code address</td>
<td></td>
</tr>
<tr>
<td>CLR A (A) ← 0</td>
<td>1</td>
</tr>
<tr>
<td>CLR C (C) ← 0</td>
<td>1</td>
</tr>
<tr>
<td>CLR bit addr (bit address) ← 0</td>
<td>1</td>
</tr>
<tr>
<td>CPL A (A) ← NOT(A)</td>
<td>1</td>
</tr>
<tr>
<td>CPL C (C) ← NOT(C)</td>
<td>1</td>
</tr>
<tr>
<td>CPL bit addr (bit address) ← NOT(bit address)</td>
<td>1</td>
</tr>
<tr>
<td>DA A (See description in Chapter 3)</td>
<td>1</td>
</tr>
<tr>
<td>DEC @Rr ((Rr)) ← ((Rr)) - 1</td>
<td>1</td>
</tr>
<tr>
<td>DEC A (A) ← (A) - 1</td>
<td>1</td>
</tr>
<tr>
<td>DEC Rr (Rr) ← (Rr) - 1</td>
<td>1</td>
</tr>
<tr>
<td>DEC data addr (data address) ← (data address) - 1</td>
<td>1</td>
</tr>
<tr>
<td>DIV AB (AB) ← (A) / (B)</td>
<td>4</td>
</tr>
<tr>
<td>Mnemonic Operation</td>
<td>Cycles</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
</tr>
<tr>
<td>DJNZ Rr, code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 2</td>
<td></td>
</tr>
<tr>
<td>(Rr) ← (Rr) - 1</td>
<td></td>
</tr>
<tr>
<td>IF (Rr) &lt; 0 THEN</td>
<td></td>
</tr>
<tr>
<td>(PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>DJNZ data addr, code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 3</td>
<td></td>
</tr>
<tr>
<td>(data address) ← (data address) - 1</td>
<td></td>
</tr>
<tr>
<td>IF (data address) = 0</td>
<td></td>
</tr>
<tr>
<td>THEN (PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>INC @Rr</td>
<td>1</td>
</tr>
<tr>
<td>((Rr)) ← ((Rr)) + 1</td>
<td></td>
</tr>
<tr>
<td>INC A</td>
<td>1</td>
</tr>
<tr>
<td>(A) ← (A) + 1</td>
<td></td>
</tr>
<tr>
<td>INC DPTR</td>
<td>1</td>
</tr>
<tr>
<td>(DPTR) ← (DPTR) + 1</td>
<td></td>
</tr>
<tr>
<td>INC Rr</td>
<td>1</td>
</tr>
<tr>
<td>((Rr)) ← (Rr) + 1</td>
<td></td>
</tr>
<tr>
<td>INC data addr</td>
<td>2</td>
</tr>
<tr>
<td>(data address) ← (data address) + 1</td>
<td></td>
</tr>
<tr>
<td>JB bit addr, code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 3</td>
<td></td>
</tr>
<tr>
<td>IF (bit address) = 1</td>
<td></td>
</tr>
<tr>
<td>THEN (PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>JBC bit addr, code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 3</td>
<td></td>
</tr>
<tr>
<td>IF (bit address) = 1</td>
<td></td>
</tr>
<tr>
<td>THEN (PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>JC code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 2</td>
<td></td>
</tr>
<tr>
<td>IF (C) = 1 THEN</td>
<td></td>
</tr>
<tr>
<td>(PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>JMP @A + DPTR</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (A) + (DPTR)</td>
<td></td>
</tr>
<tr>
<td>JNB bit addr, code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 3</td>
<td></td>
</tr>
<tr>
<td>IF (bit address) = 0</td>
<td></td>
</tr>
<tr>
<td>THEN (PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>JNC code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 2</td>
<td></td>
</tr>
<tr>
<td>IF (C) = 0 THEN</td>
<td></td>
</tr>
<tr>
<td>(PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>JNZ code addr</td>
<td>2</td>
</tr>
<tr>
<td>(PC) ← (PC) + 2</td>
<td></td>
</tr>
<tr>
<td>IF (A) &lt; 0 THEN</td>
<td></td>
</tr>
<tr>
<td>(PC) ← (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>Mnemonic Operation</td>
<td>Cycles</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------</td>
</tr>
<tr>
<td>JZ code addr (PC) - (PC) + 2</td>
<td>2</td>
</tr>
<tr>
<td>IF (A) = 0 THEN (PC) - (PC) + relative offset</td>
<td></td>
</tr>
<tr>
<td>LCALL code addr (PC) - (PC) + 3 (SP) - (SP) + 1 ((SP)) - (IP) low (SP) - (SP) + 1 ((SP)) - (IP) high (PC) - code address</td>
<td>2</td>
</tr>
<tr>
<td>LJMP code addr (PC) - code address</td>
<td>2</td>
</tr>
<tr>
<td>MOV @Rr, #data (Rr) - data</td>
<td>1</td>
</tr>
<tr>
<td>MOV @Rr, A (Rr) - (A)</td>
<td>1</td>
</tr>
<tr>
<td>MOV @Rr, data addr (Rr) - (data address)</td>
<td>2</td>
</tr>
<tr>
<td>MOV A, @Rr (A) - (Rr)</td>
<td>1</td>
</tr>
<tr>
<td>MOV A, data addr (A) - (data address)</td>
<td>1</td>
</tr>
<tr>
<td>MOV C, bit addr (C) - (bit address)</td>
<td>1</td>
</tr>
<tr>
<td>MOV D PTR, #data (D PTR) - data</td>
<td>2</td>
</tr>
<tr>
<td>MOV Rr, #data (Rr) - data</td>
<td>1</td>
</tr>
<tr>
<td>MOV Rr, A (Rr) - (A)</td>
<td>1</td>
</tr>
<tr>
<td>MOV Rr, data addr (Rr) - (data address)</td>
<td>2</td>
</tr>
<tr>
<td>MOV bit addr C (bit address) - (C)</td>
<td>2</td>
</tr>
<tr>
<td>MOV data addr, #data (data address) - data</td>
<td>2</td>
</tr>
<tr>
<td>MOV data addr, @Rr (data address) - (Rr)</td>
<td>2</td>
</tr>
<tr>
<td>MOV data addr, A (data address) - (A)</td>
<td>1</td>
</tr>
</tbody>
</table>

1 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.)

<table>
<thead>
<tr>
<th>Mnemonic Operation</th>
<th>Cycles</th>
<th>Binary Code</th>
<th>Flags</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV data addr, P</td>
<td>2</td>
<td>1 0 0 0 1 rr mmm mmm mmmm</td>
<td>P</td>
<td>Move register to data address</td>
</tr>
<tr>
<td>MOV data addr1, data addr2 (data address1) ← (data address2)</td>
<td>2</td>
<td>1 0 0 0 1 0 1 mmm mmm mmm mmmm</td>
<td>P</td>
<td>Move contents of second data address to first data address</td>
</tr>
<tr>
<td>MOVC A, @A + DPTR (A) ← ((A) + (DPTR))</td>
<td>2</td>
<td>1 0 0 1 0 0 1 1</td>
<td>P</td>
<td>Add A to DPTR and move contents of that code address with A</td>
</tr>
<tr>
<td>MOVC A, @A + PC (A) ← ((A) + (PC))</td>
<td>2</td>
<td>1 0 0 0 0 0 1 1</td>
<td>P</td>
<td>Add A to PC and move contents of that code address with A</td>
</tr>
<tr>
<td>MOVC A, data addr (A) ← (data address)</td>
<td>2</td>
<td>1 1 1 1 0 0 0 0</td>
<td>P</td>
<td>Move A to external data location addressed by DPTR</td>
</tr>
<tr>
<td>MOVC @Rr, A (Rr) ← (A)</td>
<td>2</td>
<td>1 1 1 1 0 0 1 r</td>
<td>P</td>
<td>Move A to external data location addressed by register</td>
</tr>
<tr>
<td>MOVC A, @DPTR (A) ← ((DPTR))</td>
<td>2</td>
<td>1 1 1 0 0 0 0 0</td>
<td>P</td>
<td>Move contents of external data location addressed by DPTR to A</td>
</tr>
<tr>
<td>MOVC A, @Rr (A) ← ((Rr))</td>
<td>2</td>
<td>1 1 1 0 0 0 1 r</td>
<td>P</td>
<td>Move contents of external data location addressed by register to A</td>
</tr>
<tr>
<td>MUL AB (AB) ← (A) * (B)</td>
<td>4</td>
<td>1 0 1 0 0 1 0 0</td>
<td>P</td>
<td>Multiply A by B (multiplication register)</td>
</tr>
<tr>
<td>NOP</td>
<td>1</td>
<td>0 0 0 0 0 0 0</td>
<td></td>
<td>Do nothing</td>
</tr>
<tr>
<td>ORL A, #data (A) ← (A) OR data</td>
<td>1</td>
<td>0 1 0 0 0 1 0 0 d d d d d d d d</td>
<td>P</td>
<td>Logical OR immediate data to A</td>
</tr>
<tr>
<td>ORL A, @Rr (A) ← (A) OR (Rr)</td>
<td>1</td>
<td>0 1 0 0 0 1 1 r</td>
<td>P</td>
<td>Logical OR contents of indirect address to A</td>
</tr>
<tr>
<td>ORL A, Rr (A) ← (A) OR (Rr)</td>
<td>1</td>
<td>0 1 0 0 1 r r r</td>
<td>P</td>
<td>Logical OR register to A</td>
</tr>
<tr>
<td>ORL A, data addr (A) ← (data address) OR (data address)</td>
<td>1</td>
<td>0 1 0 0 0 1 0 1 mmm mmm mmm mmm</td>
<td>P</td>
<td>Logical OR contents of data address to A</td>
</tr>
<tr>
<td>ORL C, bit addr (C) ← (C) OR (bit address)</td>
<td>2</td>
<td>0 1 1 1 0 0 1 0 b b b b b b b b</td>
<td>C</td>
<td>Logical OR bit to C</td>
</tr>
<tr>
<td>ORL C, (bit addr) OR NOT (bit address)</td>
<td>2</td>
<td>1 0 1 0 0 0 0 0 b b b b b b b b</td>
<td>C</td>
<td>Logical OR complement of bit to C</td>
</tr>
<tr>
<td>ORL data addr, #data (data address) ← (data address) OR data</td>
<td>2</td>
<td>0 1 0 0 0 0 1 1 mmm mmm mmm mmm</td>
<td></td>
<td>Logical OR immediate data to data address</td>
</tr>
<tr>
<td>ORL data addr, A (data address) ← (data address) OR A</td>
<td>1</td>
<td>0 1 0 0 0 0 1 0 mmm mmm mmm</td>
<td></td>
<td>Logical OR A to data address</td>
</tr>
</tbody>
</table>

* 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.)

<table>
<thead>
<tr>
<th>Mnemonic Operation</th>
<th>Cycles</th>
<th>Binary Code</th>
<th>Flags</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP data addr (data address) ← ((SP)) (SP) ← (SP) - 1</td>
<td>2</td>
<td>1 1 0 1 0 0 0 0 mmmmmmmmm</td>
<td>P OV AC C</td>
<td>Place top of stack at data address and decrement SP</td>
</tr>
<tr>
<td>PUSH data addr (SP) ← (SP) + 1 ((SP)) ← (data address)</td>
<td>2</td>
<td>1 1 0 0 0 0 0 0 mmmmmmmmm</td>
<td>Increment SP and place contents of data address at top of stack</td>
<td></td>
</tr>
<tr>
<td>RET (PC) high ← ((SP)) (SP) ← (SP) - 1 (PC) low ← ((SP)) (SP) ← (SP) - 1</td>
<td>2</td>
<td>0 0 1 0 0 0 0 0</td>
<td>Return from subroutine call</td>
<td></td>
</tr>
<tr>
<td>RETI (PC) high ← ((SP)) (SP) ← (SP) - 1 (PC) low ← ((SP)) (SP) ← (SP) - 1</td>
<td>2</td>
<td>0 0 1 1 0 0 1 0</td>
<td>Return from interrupt routine</td>
<td></td>
</tr>
<tr>
<td>RL A (See description in Chapter 3)</td>
<td>1</td>
<td>0 0 1 0 0 0 1 1</td>
<td>Rotate A left one position</td>
<td></td>
</tr>
<tr>
<td>RLC A (See description in Chapter 3)</td>
<td>1</td>
<td>0 0 1 1 0 0 1 1 P C</td>
<td>Rotate A through C left one position</td>
<td></td>
</tr>
<tr>
<td>RR A (See description in Chapter 3)</td>
<td>1</td>
<td>0 0 0 0 0 0 1 1</td>
<td>Rotate A right one position</td>
<td></td>
</tr>
<tr>
<td>RRC A (See description in Chapter 3)</td>
<td>1</td>
<td>0 0 0 1 0 0 1 1 P C</td>
<td>Rotate A through C right one position</td>
<td></td>
</tr>
<tr>
<td>SETB C (C) ← 1</td>
<td>1</td>
<td>1 1 0 1 0 0 1 1</td>
<td>Set C to one (1)</td>
<td></td>
</tr>
<tr>
<td>SETB bit addr (bit address) ← 1</td>
<td>1</td>
<td>1 1 0 1 0 0 1 0 b b b b b b b b</td>
<td>Set bit to one (1)</td>
<td></td>
</tr>
<tr>
<td>SJMP code addr (PC) ← (PC) + relative offset</td>
<td>2</td>
<td>1 0 0 0 0 0 0 0 o o o o o o o o</td>
<td>Jump to code address</td>
<td></td>
</tr>
<tr>
<td>SUBB A.data addr (A) ← (A) - (C) - data</td>
<td>1</td>
<td>1 0 0 1 0 1 0 0 d d d d d d d d</td>
<td>Subtract immediate data from A</td>
<td></td>
</tr>
<tr>
<td>SUBB A.@Rr (A) ← (A) - (C) - (Rr)</td>
<td>1</td>
<td>1 0 0 1 0 1 1 r</td>
<td>Subtract contents of indirect address from A</td>
<td></td>
</tr>
<tr>
<td>SUBB A.Rr (A) ← (A) - (C) - (Rr)</td>
<td>1</td>
<td>1 0 0 1 1 r r r</td>
<td>Subtract register from A</td>
<td></td>
</tr>
<tr>
<td>SUBB A.data addr (A) ← (A) - (C) - (data address)</td>
<td>1</td>
<td>1 0 0 1 0 1 0 1 mmmmmmmmm</td>
<td>Subtract contents of data address from A</td>
<td></td>
</tr>
<tr>
<td>SWAP A (See description in Chapter 3)</td>
<td>1</td>
<td>1 1 0 0 0 1 0 0</td>
<td>Exchange low order nibble with high order nibble in A</td>
<td></td>
</tr>
<tr>
<td>XCH A.@Rr temp ← ((Rr)) ((Rr)) ← (A) (A) ← temp</td>
<td>1</td>
<td>1 1 0 0 0 1 1 r</td>
<td>Move A to indirect address and vice versa</td>
<td></td>
</tr>
<tr>
<td>XCH A.Rr temp ← (Rr) (Rr) ← (A) (A) ← temp</td>
<td>1</td>
<td>1 1 0 0 1 r r r</td>
<td>Move A to register and vice versa</td>
<td></td>
</tr>
<tr>
<td>XCH A.data addr temp ← (data address) (data address) ← (A) (A) ← temp</td>
<td>1</td>
<td>1 1 0 0 0 1 0 1 mmmmmmmmm</td>
<td>Move A to data address and vice versa</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-1. Instruction Set Summary (Cont’d.)

<table>
<thead>
<tr>
<th>Mnemonic Operation</th>
<th>Cycles</th>
<th>Binary Code</th>
<th>Flags</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCHD A, @Rr</td>
<td>1</td>
<td>11101011r</td>
<td>P</td>
<td>Move low order of A to low order nibble of indirect address and vice versa</td>
</tr>
<tr>
<td>temp ← ((Rr)) 0-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((Rr)) 0-3 ← (A) 0-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(A) 0-3 ← temp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL A, data addr</td>
<td>1</td>
<td>01100100 ddddddd d</td>
<td>P</td>
<td>Logical exclusive OR immediate data to A</td>
</tr>
<tr>
<td>(A) ← (A) XOR data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL A, @Rr</td>
<td>1</td>
<td>0110011r</td>
<td>P</td>
<td>Logical exclusive OR contents of indirect address to A</td>
</tr>
<tr>
<td>(A) ← (A) XOR ((Rr))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL A, Rr</td>
<td>1</td>
<td>01101rrr</td>
<td>P</td>
<td>Logical exclusive OR register to A</td>
</tr>
<tr>
<td>(A) ← (A) XOR (Rr)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL A, data addr</td>
<td>1</td>
<td>0110101 mmmm mmmm</td>
<td>P</td>
<td>Logical exclusive OR contents of data address to A</td>
</tr>
<tr>
<td>(A) ← (A) XOR (data address)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL data addr, data addr</td>
<td>2</td>
<td>0110011 mmmm mmmm ddddddd d</td>
<td>P</td>
<td>Logical exclusive OR immediate data to data address</td>
</tr>
<tr>
<td>(data address) ← (data address) XOR data</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XRL data addr, A</td>
<td>1</td>
<td>0110010 mmmm mmmm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(data address) ← (data address) XOR A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hex Code</td>
<td>Number of Bytes</td>
<td>Mnemonic</td>
<td>Operands</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-----------</td>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>3</td>
<td>LJMP</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>1</td>
<td>RR</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>1</td>
<td>INC</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>2</td>
<td>INC</td>
<td>data addr</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>1</td>
<td>INC</td>
<td>@R0</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>1</td>
<td>INC</td>
<td>@R1</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>1</td>
<td>INC</td>
<td>R0</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>1</td>
<td>INC</td>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>0A</td>
<td>1</td>
<td>INC</td>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>0B</td>
<td>1</td>
<td>INC</td>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>0C</td>
<td>1</td>
<td>INC</td>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>0D</td>
<td>1</td>
<td>INC</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>0E</td>
<td>1</td>
<td>INC</td>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td>1</td>
<td>INC</td>
<td>R7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>JBC</td>
<td>bit addr,code addr</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>LCALL</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>RRC</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>DEC</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>DEC</td>
<td>data addr</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>DEC</td>
<td>@R0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>DEC</td>
<td>@R1</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>DEC</td>
<td>R0</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>DEC</td>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>1A</td>
<td>1</td>
<td>DEC</td>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td>1</td>
<td>DEC</td>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>1</td>
<td>DEC</td>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>1D</td>
<td>1</td>
<td>DEC</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>1E</td>
<td>1</td>
<td>DEC</td>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>1F</td>
<td>1</td>
<td>DEC</td>
<td>R7</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>JB</td>
<td>bit addr,code addr</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>RL</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>ADD</td>
<td>A,#data</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>ADD</td>
<td>A,data addr</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>ADD</td>
<td>A,@R0</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>ADD</td>
<td>A,@R1</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>ADD</td>
<td>A,R0</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>ADD</td>
<td>A,R1</td>
<td></td>
</tr>
<tr>
<td>2A</td>
<td>1</td>
<td>ADD</td>
<td>A,R2</td>
<td></td>
</tr>
<tr>
<td>2B</td>
<td>1</td>
<td>ADD</td>
<td>A,R3</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>1</td>
<td>ADD</td>
<td>A,R4</td>
<td></td>
</tr>
<tr>
<td>2D</td>
<td>1</td>
<td>ADD</td>
<td>A,R5</td>
<td></td>
</tr>
<tr>
<td>2E</td>
<td>1</td>
<td>ADD</td>
<td>A,R6</td>
<td></td>
</tr>
<tr>
<td>2F</td>
<td>1</td>
<td>ADD</td>
<td>A,R7</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>JNB</td>
<td>bit addr,code addr</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>RETI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>RLC</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>ADDC</td>
<td>A,#data</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>ADDC</td>
<td>A,data addr</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>ADDC</td>
<td>A,@R0</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>ADDC</td>
<td>A,@R1</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>ADDC</td>
<td>A,R0</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>ADDC</td>
<td>A,R1</td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>1</td>
<td>ADDC</td>
<td>A,R2</td>
<td></td>
</tr>
<tr>
<td>3B</td>
<td>1</td>
<td>ADDC</td>
<td>A,R3</td>
<td></td>
</tr>
<tr>
<td>Hex Code</td>
<td>Number of Bytes</td>
<td>Mnemonic</td>
<td>Operands</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>----------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>1</td>
<td>ADDC</td>
<td>A,R4</td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>1</td>
<td>ADDC</td>
<td>A,R5</td>
<td></td>
</tr>
<tr>
<td>3E</td>
<td>1</td>
<td>ADDC</td>
<td>A,R7</td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td>1</td>
<td>ADDC</td>
<td>A,R7</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>JC</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>ORL</td>
<td>data addr,A</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>3</td>
<td>ORL</td>
<td>data addr,#data</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>2</td>
<td>ORL</td>
<td>A,#data</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>2</td>
<td>ORL</td>
<td>A,data addr</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>ORL</td>
<td>A,@R0</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>ORL</td>
<td>A,@R1</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1</td>
<td>ORL</td>
<td>A,R0</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>1</td>
<td>ORL</td>
<td>A,R1</td>
<td></td>
</tr>
<tr>
<td>4A</td>
<td>1</td>
<td>ORL</td>
<td>A,R2</td>
<td></td>
</tr>
<tr>
<td>4B</td>
<td>1</td>
<td>ORL</td>
<td>A,R3</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>1</td>
<td>ORL</td>
<td>A,R4</td>
<td></td>
</tr>
<tr>
<td>4D</td>
<td>1</td>
<td>ORL</td>
<td>A,R5</td>
<td></td>
</tr>
<tr>
<td>4E</td>
<td>1</td>
<td>ORL</td>
<td>A,R6</td>
<td></td>
</tr>
<tr>
<td>4F</td>
<td>1</td>
<td>ORL</td>
<td>A,R7</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>JNC</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>2</td>
<td>ANL</td>
<td>data addr,A</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>3</td>
<td>ANL</td>
<td>data addr,#data</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>2</td>
<td>ANL</td>
<td>A,#data</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>2</td>
<td>ANL</td>
<td>A,data addr</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>ANL</td>
<td>A,@R0</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>1</td>
<td>ANL</td>
<td>A,@R1</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>1</td>
<td>ANL</td>
<td>A,R0</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>ANL</td>
<td>A,R1</td>
<td></td>
</tr>
<tr>
<td>5A</td>
<td>1</td>
<td>ANL</td>
<td>A,R2</td>
<td></td>
</tr>
<tr>
<td>5B</td>
<td>1</td>
<td>ANL</td>
<td>A,R3</td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>1</td>
<td>ANL</td>
<td>A,R4</td>
<td></td>
</tr>
<tr>
<td>5D</td>
<td>1</td>
<td>ANL</td>
<td>A,R5</td>
<td></td>
</tr>
<tr>
<td>5E</td>
<td>1</td>
<td>ANL</td>
<td>A,R6</td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>1</td>
<td>ANL</td>
<td>A,R7</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>2</td>
<td>JZ</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>2</td>
<td>XRL</td>
<td>data addr,A</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>3</td>
<td>XRL</td>
<td>data addr,#data</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>XRL</td>
<td>A,#data</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>2</td>
<td>XRL</td>
<td>A,data addr</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>1</td>
<td>XRL</td>
<td>A,@R0</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>XRL</td>
<td>A,@R1</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>1</td>
<td>XRL</td>
<td>A,R0</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>1</td>
<td>XRL</td>
<td>A,R1</td>
<td></td>
</tr>
<tr>
<td>6A</td>
<td>1</td>
<td>XRL</td>
<td>A,R2</td>
<td></td>
</tr>
<tr>
<td>6B</td>
<td>1</td>
<td>XRL</td>
<td>A,R3</td>
<td></td>
</tr>
<tr>
<td>6C</td>
<td>1</td>
<td>XRL</td>
<td>A,R4</td>
<td></td>
</tr>
<tr>
<td>6D</td>
<td>1</td>
<td>XRL</td>
<td>A,R5</td>
<td></td>
</tr>
<tr>
<td>6E</td>
<td>1</td>
<td>XRL</td>
<td>A,R6</td>
<td></td>
</tr>
<tr>
<td>6F</td>
<td>1</td>
<td>XRL</td>
<td>A,R7</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>2</td>
<td>JNZ</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>2</td>
<td>ORL</td>
<td>C,bit addr</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>1</td>
<td>JMP</td>
<td>@A+DPTR</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>2</td>
<td>MOV</td>
<td>A,#data</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>3</td>
<td>MOV</td>
<td>data addr,#data</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>2</td>
<td>MOV</td>
<td>@R0,#data</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>2</td>
<td>MOV</td>
<td>@R1,#data</td>
<td></td>
</tr>
</tbody>
</table>
### Table B-2. Instruction Opcodes in Hexadecimal (Cont’d.)

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Number of Bytes</th>
<th>Mnemonic</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>2</td>
<td>MOV</td>
<td>R0, #data</td>
</tr>
<tr>
<td>79</td>
<td>2</td>
<td>MOV</td>
<td>R1, #data</td>
</tr>
<tr>
<td>7A</td>
<td>2</td>
<td>MOV</td>
<td>R2, #data</td>
</tr>
<tr>
<td>7B</td>
<td>2</td>
<td>MOV</td>
<td>R3, #data</td>
</tr>
<tr>
<td>7C</td>
<td>2</td>
<td>MOV</td>
<td>R4, #data</td>
</tr>
<tr>
<td>7D</td>
<td>2</td>
<td>MOV</td>
<td>R5, #data</td>
</tr>
<tr>
<td>7E</td>
<td>2</td>
<td>MOV</td>
<td>R6, #data</td>
</tr>
<tr>
<td>7F</td>
<td>2</td>
<td>MOV</td>
<td>R7, #data</td>
</tr>
<tr>
<td>80</td>
<td>2</td>
<td>SJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>81</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>82</td>
<td>2</td>
<td>ANL</td>
<td>C, bit addr</td>
</tr>
<tr>
<td>83</td>
<td>1</td>
<td>MOVC</td>
<td>A, @A + PC</td>
</tr>
<tr>
<td>84</td>
<td>1</td>
<td>DIV</td>
<td>AB</td>
</tr>
<tr>
<td>85</td>
<td>3</td>
<td>MOV</td>
<td>data addr, data addr</td>
</tr>
<tr>
<td>86</td>
<td>2</td>
<td>MOV</td>
<td>data addr, @R0</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>MOV</td>
<td>data addr, @R1</td>
</tr>
<tr>
<td>88</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R0</td>
</tr>
<tr>
<td>89</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R1</td>
</tr>
<tr>
<td>8A</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R2</td>
</tr>
<tr>
<td>8B</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R3</td>
</tr>
<tr>
<td>8C</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R4</td>
</tr>
<tr>
<td>8D</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R5</td>
</tr>
<tr>
<td>8E</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R6</td>
</tr>
<tr>
<td>8F</td>
<td>2</td>
<td>MOV</td>
<td>data addr, R7</td>
</tr>
<tr>
<td>90</td>
<td>3</td>
<td>MOV</td>
<td>DPT, #data</td>
</tr>
<tr>
<td>91</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>92</td>
<td>2</td>
<td>MOV</td>
<td>bit addr, C</td>
</tr>
<tr>
<td>93</td>
<td>1</td>
<td>MOVC</td>
<td>A, @A + DPT</td>
</tr>
<tr>
<td>94</td>
<td>2</td>
<td>SUBB</td>
<td>A, #data</td>
</tr>
<tr>
<td>95</td>
<td>2</td>
<td>SUBB</td>
<td>A, data addr</td>
</tr>
<tr>
<td>96</td>
<td>1</td>
<td>SUBB</td>
<td>A, @R0</td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>SUBB</td>
<td>A, @R1</td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>SUBB</td>
<td>A, R0</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>SUBB</td>
<td>A, R1</td>
</tr>
<tr>
<td>9A</td>
<td>1</td>
<td>SUBB</td>
<td>A, R2</td>
</tr>
<tr>
<td>9B</td>
<td>1</td>
<td>SUBB</td>
<td>A, R3</td>
</tr>
<tr>
<td>9C</td>
<td>1</td>
<td>SUBB</td>
<td>A, R4</td>
</tr>
<tr>
<td>9D</td>
<td>1</td>
<td>SUBB</td>
<td>A, R5</td>
</tr>
<tr>
<td>9E</td>
<td>1</td>
<td>SUBB</td>
<td>A, R6</td>
</tr>
<tr>
<td>9F</td>
<td>1</td>
<td>SUBB</td>
<td>A, R7</td>
</tr>
<tr>
<td>A0</td>
<td>2</td>
<td>ORL</td>
<td>C, bit addr</td>
</tr>
<tr>
<td>A1</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>A2</td>
<td>2</td>
<td>MOV</td>
<td>C, bit addr</td>
</tr>
<tr>
<td>A3</td>
<td>1</td>
<td>INC</td>
<td>DPT</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>MUL</td>
<td>AB</td>
</tr>
<tr>
<td>A5</td>
<td>1</td>
<td>reserved</td>
<td>@R0, data addr</td>
</tr>
<tr>
<td>A6</td>
<td>2</td>
<td>MOV</td>
<td>@R0, data addr</td>
</tr>
<tr>
<td>A7</td>
<td>2</td>
<td>MOV</td>
<td>@R1, data addr</td>
</tr>
<tr>
<td>A8</td>
<td>2</td>
<td>MOV</td>
<td>R0, data addr</td>
</tr>
<tr>
<td>A9</td>
<td>2</td>
<td>MOV</td>
<td>R1, data addr</td>
</tr>
<tr>
<td>AA</td>
<td>2</td>
<td>MOV</td>
<td>R2, data addr</td>
</tr>
<tr>
<td>AB</td>
<td>2</td>
<td>MOV</td>
<td>R3, data addr</td>
</tr>
<tr>
<td>AC</td>
<td>2</td>
<td>MOV</td>
<td>R4, data addr</td>
</tr>
<tr>
<td>AD</td>
<td>2</td>
<td>MOV</td>
<td>R5, data addr</td>
</tr>
<tr>
<td>AE</td>
<td>2</td>
<td>MOV</td>
<td>R6, data addr</td>
</tr>
<tr>
<td>AF</td>
<td>2</td>
<td>MOV</td>
<td>R7, data addr</td>
</tr>
<tr>
<td>B0</td>
<td>2</td>
<td>ANL</td>
<td>C, bit addr</td>
</tr>
<tr>
<td>B1</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
<td>CPL</td>
<td>bit addr</td>
</tr>
<tr>
<td>B3</td>
<td>1</td>
<td>CPL</td>
<td>C</td>
</tr>
<tr>
<td>Hex Code</td>
<td>Number of Bytes</td>
<td>Mnemonic</td>
<td>Operands</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>----------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>B4</td>
<td>3</td>
<td>CJNE</td>
<td>A, #data, code addr</td>
</tr>
<tr>
<td>B5</td>
<td>3</td>
<td>CJNE</td>
<td>A, data addr, code addr</td>
</tr>
<tr>
<td>B6</td>
<td>3</td>
<td>CJNE</td>
<td>@R6, #data, code addr</td>
</tr>
<tr>
<td>B7</td>
<td>3</td>
<td>CJNE</td>
<td>@R1, #data, code addr</td>
</tr>
<tr>
<td>B8</td>
<td>3</td>
<td>CJNE</td>
<td>R0, #data, code addr</td>
</tr>
<tr>
<td>B9</td>
<td>3</td>
<td>CJNE</td>
<td>R1, #data, code addr</td>
</tr>
<tr>
<td>BA</td>
<td>3</td>
<td>CJNE</td>
<td>R2, #data, code addr</td>
</tr>
<tr>
<td>BB</td>
<td>3</td>
<td>CJNE</td>
<td>R3, #data, code addr</td>
</tr>
<tr>
<td>BC</td>
<td>3</td>
<td>CJNE</td>
<td>R4, #data, code addr</td>
</tr>
<tr>
<td>BD</td>
<td>3</td>
<td>CJNE</td>
<td>R5, #data, code addr</td>
</tr>
<tr>
<td>BE</td>
<td>3</td>
<td>CJNE</td>
<td>R6, #data, code addr</td>
</tr>
<tr>
<td>BF</td>
<td>3</td>
<td>CJNE</td>
<td>R7, #data, code addr</td>
</tr>
<tr>
<td>C0</td>
<td>2</td>
<td>PUSH</td>
<td>data addr</td>
</tr>
<tr>
<td>C1</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>C2</td>
<td>2</td>
<td>CLR</td>
<td>bit addr</td>
</tr>
<tr>
<td>C3</td>
<td>1</td>
<td>CLR</td>
<td>C</td>
</tr>
<tr>
<td>C4</td>
<td>1</td>
<td>SWAP</td>
<td>A</td>
</tr>
<tr>
<td>C5</td>
<td>2</td>
<td>XCH</td>
<td>A, data addr</td>
</tr>
<tr>
<td>C6</td>
<td>1</td>
<td>XCH</td>
<td>A, @R0</td>
</tr>
<tr>
<td>C7</td>
<td>1</td>
<td>XCH</td>
<td>A, @R1</td>
</tr>
<tr>
<td>C8</td>
<td>1</td>
<td>XCH</td>
<td>A, R0</td>
</tr>
<tr>
<td>C9</td>
<td>1</td>
<td>XCH</td>
<td>A, R1</td>
</tr>
<tr>
<td>CA</td>
<td>1</td>
<td>XCH</td>
<td>A, R2</td>
</tr>
<tr>
<td>CB</td>
<td>1</td>
<td>XCH</td>
<td>A, R3</td>
</tr>
<tr>
<td>CC</td>
<td>1</td>
<td>XCH</td>
<td>A, R4</td>
</tr>
<tr>
<td>CD</td>
<td>1</td>
<td>XCH</td>
<td>A, R5</td>
</tr>
<tr>
<td>CE</td>
<td>1</td>
<td>XCH</td>
<td>A, R6</td>
</tr>
<tr>
<td>CF</td>
<td>1</td>
<td>XCH</td>
<td>A, R7</td>
</tr>
<tr>
<td>D0</td>
<td>2</td>
<td>POP</td>
<td>data addr</td>
</tr>
<tr>
<td>D1</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>D2</td>
<td>2</td>
<td>SETB</td>
<td>bit addr</td>
</tr>
<tr>
<td>D3</td>
<td>1</td>
<td>SETB</td>
<td>C</td>
</tr>
<tr>
<td>D4</td>
<td>1</td>
<td>DA</td>
<td>A</td>
</tr>
<tr>
<td>D5</td>
<td>3</td>
<td>DJNZ</td>
<td>data addr, code addr</td>
</tr>
<tr>
<td>D6</td>
<td>1</td>
<td>XCHD</td>
<td>A, @R0</td>
</tr>
<tr>
<td>D7</td>
<td>1</td>
<td>XCHD</td>
<td>A, @R1</td>
</tr>
<tr>
<td>D8</td>
<td>2</td>
<td>DJNZ</td>
<td>R0, code addr</td>
</tr>
<tr>
<td>D9</td>
<td>2</td>
<td>DJNZ</td>
<td>R1, code addr</td>
</tr>
<tr>
<td>DA</td>
<td>2</td>
<td>DJNZ</td>
<td>R2, code addr</td>
</tr>
<tr>
<td>DB</td>
<td>2</td>
<td>DJNZ</td>
<td>R3, code addr</td>
</tr>
<tr>
<td>DC</td>
<td>2</td>
<td>DJNZ</td>
<td>R4, code addr</td>
</tr>
<tr>
<td>DD</td>
<td>2</td>
<td>DJNZ</td>
<td>R5, code addr</td>
</tr>
<tr>
<td>DE</td>
<td>2</td>
<td>DJNZ</td>
<td>R6, code addr</td>
</tr>
<tr>
<td>DF</td>
<td>2</td>
<td>DJNZ</td>
<td>R7, code addr</td>
</tr>
<tr>
<td>E0</td>
<td>1</td>
<td>MOVX</td>
<td>A, @DPTR</td>
</tr>
<tr>
<td>E1</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>E2</td>
<td>1</td>
<td>MOVX</td>
<td>A, @R0</td>
</tr>
<tr>
<td>E3</td>
<td>1</td>
<td>MOVX</td>
<td>A, @R1</td>
</tr>
<tr>
<td>E4</td>
<td>1</td>
<td>CLR</td>
<td>A</td>
</tr>
<tr>
<td>E5</td>
<td>2</td>
<td>MOV</td>
<td>A, data addr</td>
</tr>
<tr>
<td>E6</td>
<td>1</td>
<td>MOV</td>
<td>A, @R0</td>
</tr>
<tr>
<td>E7</td>
<td>1</td>
<td>MOV</td>
<td>A, @R1</td>
</tr>
<tr>
<td>E8</td>
<td>1</td>
<td>MOV</td>
<td>A, R0</td>
</tr>
<tr>
<td>E9</td>
<td>1</td>
<td>MOV</td>
<td>A, R1</td>
</tr>
<tr>
<td>EA</td>
<td>1</td>
<td>MOV</td>
<td>A, R2</td>
</tr>
<tr>
<td>EB</td>
<td>1</td>
<td>MOV</td>
<td>A, R3</td>
</tr>
<tr>
<td>EC</td>
<td>1</td>
<td>MOV</td>
<td>A, R4</td>
</tr>
<tr>
<td>ED</td>
<td>1</td>
<td>MOV</td>
<td>A, R5</td>
</tr>
<tr>
<td>EE</td>
<td>1</td>
<td>MOV</td>
<td>A, R6</td>
</tr>
<tr>
<td>EF</td>
<td>1</td>
<td>MOV</td>
<td>A, R7</td>
</tr>
</tbody>
</table>
Table B-2. Instruction Opcodes in Hexadecimal (Cont’d.)

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Number of Bytes</th>
<th>Mnemonic</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>1</td>
<td>MOVX</td>
<td>@DPTR, A</td>
</tr>
<tr>
<td>F1</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>F2</td>
<td>1</td>
<td>MOVX</td>
<td>@R0, A</td>
</tr>
<tr>
<td>F3</td>
<td>1</td>
<td>MOVX</td>
<td>@R1, A</td>
</tr>
<tr>
<td>F4</td>
<td>1</td>
<td>CPL</td>
<td>A</td>
</tr>
<tr>
<td>F5</td>
<td>2</td>
<td>MOV</td>
<td>data addr, A</td>
</tr>
<tr>
<td>F6</td>
<td>1</td>
<td>MOV</td>
<td>@R0, A</td>
</tr>
<tr>
<td>F7</td>
<td>1</td>
<td>MOV</td>
<td>@R1, A</td>
</tr>
<tr>
<td>F8</td>
<td>1</td>
<td>MOV</td>
<td>R0, A</td>
</tr>
<tr>
<td>F9</td>
<td>1</td>
<td>MOV</td>
<td>R1, A</td>
</tr>
<tr>
<td>FA</td>
<td>1</td>
<td>MOV</td>
<td>R2, A</td>
</tr>
<tr>
<td>FB</td>
<td>1</td>
<td>MOV</td>
<td>R3, A</td>
</tr>
<tr>
<td>FC</td>
<td>1</td>
<td>MOV</td>
<td>R4, A</td>
</tr>
<tr>
<td>FD</td>
<td>1</td>
<td>MOV</td>
<td>R5, A</td>
</tr>
<tr>
<td>FE</td>
<td>1</td>
<td>MOV</td>
<td>R6, A</td>
</tr>
<tr>
<td>FF</td>
<td>1</td>
<td>MOV</td>
<td>R7, A</td>
</tr>
</tbody>
</table>
The following is a list of all 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

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>symbol BIT bit address</code></td>
<td>Define a bit address symbol</td>
</tr>
<tr>
<td><code>BSEG</code></td>
<td>Select bit address segment</td>
</tr>
<tr>
<td><code>CSEG</code></td>
<td>Select Code segment</td>
</tr>
<tr>
<td><code>symbol DATA expression</code></td>
<td>Define a data address symbol</td>
</tr>
<tr>
<td><code>[label:] DB expression list</code></td>
<td>Insert a list of byte values</td>
</tr>
<tr>
<td><code>[label:] DBIT expression</code></td>
<td>Advance bit location counter</td>
</tr>
<tr>
<td><code>[label:] DS expression</code></td>
<td>Advance active location counter</td>
</tr>
<tr>
<td><code>DSEG</code></td>
<td>Select internal Data Segment</td>
</tr>
<tr>
<td><code>[label:] DW expression list</code></td>
<td>Insert a list of word values</td>
</tr>
<tr>
<td><code>END</code></td>
<td>End of program</td>
</tr>
<tr>
<td><code>symbol EQU expression or register</code></td>
<td>Set symbol value permanently</td>
</tr>
<tr>
<td><code>ORG expression</code></td>
<td>Set location counter value</td>
</tr>
<tr>
<td><code>symbol SET expression or register</code></td>
<td>Set symbol value temporarily</td>
</tr>
<tr>
<td><code>symbol XDATA expression</code></td>
<td>Define an off chip data address symbol</td>
</tr>
<tr>
<td><code>XSEG</code></td>
<td>Select external Data Segment</td>
</tr>
</tbody>
</table>

C-1
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Primary/General</th>
<th>Default</th>
<th>Abbrev.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE(date)</td>
<td>P</td>
<td>DATE( )</td>
<td>DA</td>
<td>Place string in header (max 9 characters)</td>
</tr>
<tr>
<td>DEBUG</td>
<td>P</td>
<td>NODEBUG</td>
<td>DB</td>
<td>Outputs debug symbol information to object file</td>
</tr>
<tr>
<td>NODEBUG</td>
<td>P</td>
<td>NODB</td>
<td></td>
<td>Symbol information not placed in object file</td>
</tr>
<tr>
<td>EJECT</td>
<td>G</td>
<td></td>
<td>EJ</td>
<td>Continue listing on next page</td>
</tr>
<tr>
<td>ERRORPRINT(FILE)</td>
<td>P</td>
<td>NOERRORPRINT</td>
<td>EP</td>
<td>Designates a file to receive error messages</td>
</tr>
<tr>
<td>NOERRORPRINT</td>
<td>P</td>
<td>NOEP</td>
<td></td>
<td>Designates that error messages will be printed</td>
</tr>
<tr>
<td>GEN</td>
<td>G</td>
<td>GENONLY</td>
<td>GE</td>
<td>Generates a full listing of the macro expansion</td>
</tr>
<tr>
<td>GENONLY</td>
<td>G</td>
<td></td>
<td>GO</td>
<td>Lists only the fully expanded source as if all</td>
</tr>
<tr>
<td>NOGEN</td>
<td>G</td>
<td></td>
<td>NOGE</td>
<td>List only the original source text in listing</td>
</tr>
<tr>
<td>INCLUDE(FILE)</td>
<td>G</td>
<td></td>
<td>IC</td>
<td>Designates a file to be included as part of the</td>
</tr>
<tr>
<td>LIST</td>
<td>G</td>
<td>LIST</td>
<td>LI</td>
<td>Print subsequent lines of source in listing file</td>
</tr>
<tr>
<td>NOLIST</td>
<td>G</td>
<td></td>
<td>NOLI</td>
<td>Do not print subsequent lines of source in listing file</td>
</tr>
<tr>
<td>MACRO</td>
<td>P</td>
<td>MACRO</td>
<td>MR</td>
<td>Evaluate and expand all macro calls</td>
</tr>
<tr>
<td>NOMACRO</td>
<td>P</td>
<td></td>
<td>NOMR</td>
<td>Do not evaluate macro calls</td>
</tr>
<tr>
<td>OBJECT(FILE)</td>
<td>P</td>
<td>OBJECT(source.HEX)</td>
<td>OJ</td>
<td>Designate file to receive object code</td>
</tr>
<tr>
<td>NOOBJECT</td>
<td>P</td>
<td></td>
<td>NOOJ</td>
<td>Designates that no object file will be created</td>
</tr>
<tr>
<td>PAGING</td>
<td>P</td>
<td>PAGING</td>
<td>PI</td>
<td>Designates that listing will be broken into</td>
</tr>
<tr>
<td>NOPAGING</td>
<td>P</td>
<td></td>
<td>NOPI</td>
<td>Designates that listing will contain no page</td>
</tr>
<tr>
<td>PAGELENGTH(n)</td>
<td>P</td>
<td>PAGELENGTH(60)</td>
<td>PL</td>
<td>Sets maximum number of lines in each page</td>
</tr>
<tr>
<td>PAGEWIDTH(n)</td>
<td>P</td>
<td>PAGEWIDTH(120)</td>
<td>PW</td>
<td>Sets maximum number of characters in each line</td>
</tr>
<tr>
<td>PRINT(FILE)</td>
<td>P</td>
<td>PRINT(source.LST)</td>
<td>PR</td>
<td>Designates file to receive source listing</td>
</tr>
<tr>
<td>NOPRINT</td>
<td>P</td>
<td></td>
<td>NOPR</td>
<td>Designates that no listing file will be created</td>
</tr>
</tbody>
</table>


### Table D-1. Assembler Controls (Cont’d.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Primary/General</th>
<th>Default</th>
<th>Abbrev.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE</td>
<td>G</td>
<td>Not Applicable</td>
<td>SA</td>
<td>Stores current control setting for LIST and GEN</td>
</tr>
<tr>
<td>RESTORE</td>
<td>G</td>
<td></td>
<td>RS</td>
<td>Restores control setting from SAVE stack</td>
</tr>
<tr>
<td>SYMBOLS</td>
<td>P</td>
<td>NOSYMBOLS</td>
<td>SB</td>
<td>Creates a formatted table of all symbols used in program</td>
</tr>
<tr>
<td>NOSYMBOLS</td>
<td>P</td>
<td></td>
<td>NOSB</td>
<td>No symbol table created</td>
</tr>
<tr>
<td>TITLE(string)</td>
<td>G</td>
<td>TITLE( )</td>
<td>TT</td>
<td>Places a string in all subsequent page headers (maximum 60 characters)</td>
</tr>
<tr>
<td>WORKFILES(:Fn;[:F m:])</td>
<td>P</td>
<td>same drive as source file</td>
<td>WF</td>
<td>Designates alternate drives for temporary workfiles</td>
</tr>
<tr>
<td>XREF</td>
<td>P</td>
<td>NOXREF</td>
<td>XR</td>
<td>Creates a cross reference listing of all symbols used in program</td>
</tr>
<tr>
<td>NOXREF</td>
<td>P</td>
<td></td>
<td>NOXR</td>
<td>No cross reference list created</td>
</tr>
</tbody>
</table>
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.

\[ id = \text{alphabetic} \ | \ id \ id\_\text{continuation} \]

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:

\[ \text{WHILE } \text{Add\_2 } \text{MORE\_TO\_DO } \text{Much\_More} \]

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 “0100”) 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):

<table>
<thead>
<tr>
<th>Parenthesized Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH</td>
</tr>
<tr>
<td>MOD</td>
</tr>
<tr>
<td>+ -</td>
</tr>
<tr>
<td>NOT</td>
</tr>
</tbody>
</table>

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:

  abed %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:

  ... abed 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,

  ... abed xyz efgh, abed %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,

```plaintext
... %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:

```plaintext
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.)  evaluates to:  This is a string.
%( %EVAL( %NUM ) )  evaluates to:  %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:

\[
\text{escape\_function} = / \text{A single digit, 0 through 9, followed by that many characters.} / \n\]

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:

\[
\text{macro\_def\_function} = \text{DEFINE} \ (\text{macro\_id} \text{ define\_pattern}) \ [\text{LOCAL} \ (\text{id})] \ \ (\text{balanced\_text}) \ 
\]

The define_pattern is a balanced string which is further analyzed by the macro processor as follows:

\[
\text{define\_pattern} = \{ \{\text{parm\_id}\}\} \{\text{delimiter\_specifier}\} \}
\]

\[
\text{delimiter\_specifier} = \text{/* String not containing non-Literal id\_continuation, logical blank, or "@" characters. */}
\]

\[
| \text{@ delimiter\_id}
\]
The syntax for a macro invocation is as follows:

```plaintext
macro_call = macro_id [ call_pattern ]
```

```plaintext
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:

```plaintext
% *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:

\[
\text{if\_function} = \text{IF} (\ expr \ ) \ \text{THEN} \ (\ body \ ) \ | \ \text{ELSE} \ (\ body \ ) \ \text{FI}
\]

\[
\text{repeat\_function} = \text{REPEAT} (\ expr \ ) \ (\ body \ )
\]

\[
\text{while\_function} = \text{WHILE} (\ expr \ ) \ (\ body \ )
\]

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.) 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:

\[
%\text{IF} (\%\text{VAL GT 0}) \ \text{THEN}(\ %\text{DEFINE}(\text{SIGN})(1)) \ \text{ELSE}(\ %\text{DEFINE}(\text{SIGN})(0)) \ \text{FI}
\]

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.

\[
%\text{DEFINE}(\text{SIGN})(%\text{IF} (\%\text{VAL GT 0}) \ \text{THEN}(1) \ \text{ELSE}(0) \ \text{FI})
\]

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:

```plaintext
%REPEAT (6)
(   RRA
)
```

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:

```plaintext
%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:

```plaintext
%WHILE (%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:

```plaintext
%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:

```plaintext
in_function = IN
```

```plaintext
out_function = OUT ( balanced_text )
```

Examples:

```plaintext
%OUT (Enter the date:)
%DEFINE(DATE)(%IN)
```

**The Substring Function**

The syntax of the substring function is:

```plaintext
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:

```plaintext
%SUBSTR (ABCDEFGH,3,4)
```

has the value “CDEF”

```plaintext
%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:

\[
\text{comment\_function} = ' | \text{text} | \text{\linefeed})
\]

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:

\[
\% ' \text{This comment fits within one line.}'
\]

\[
\% ' \text{This comment continues through the end of the line.}'
\]

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:

\[
\text{metachar\_function} = \text{METACHAR( balanced\_text )}
\]

The first character of the \text{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).
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.

<table>
<thead>
<tr>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
</tr>
<tr>
<td>EQ</td>
</tr>
<tr>
<td>GE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACALL</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>ADDC</td>
</tr>
<tr>
<td>AJMP</td>
</tr>
<tr>
<td>ANL</td>
</tr>
<tr>
<td>CJNE</td>
</tr>
<tr>
<td>CLR</td>
</tr>
<tr>
<td>CPL</td>
</tr>
<tr>
<td>DA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>AB</td>
</tr>
<tr>
<td>AC</td>
</tr>
<tr>
<td>ACC</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>CY</td>
</tr>
<tr>
<td>DPH</td>
</tr>
<tr>
<td>DPL</td>
</tr>
<tr>
<td>DPTR</td>
</tr>
<tr>
<td>EA</td>
</tr>
<tr>
<td>ES</td>
</tr>
<tr>
<td>ET0</td>
</tr>
<tr>
<td>ET1</td>
</tr>
<tr>
<td>EX0</td>
</tr>
<tr>
<td>EX1</td>
</tr>
<tr>
<td>EXTI0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
</tr>
<tr>
<td>BSEG</td>
</tr>
<tr>
<td>CSEG</td>
</tr>
</tbody>
</table>
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.

```asm
ORG 3000

These strings will be placed in high memory; They will be used to output messages to the terminal

SOURCE FIRST_NUMBER DATA 50 ; STORAGE LOCATION FOR FIRST NUMBER
SECOND_NUMBER DATA 60 ; STORAGE FOR SECOND NUMBER

SUM: DB 'THE SUM IS ',DOH

The following instructions prepare the serial port to receive and send data at 110 baud

Set timer mode to auto-reload

Proper power supply
Logic to modify TTL signal to current loop
Necessary cabling to connect terminal

110 = 10.7MHz/12*16*2*253
110 = desired baud rate

-253 ; timer preset value

Prepare serial port for operation
Start clock

Figure G-1. Sample Program
```
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>900888</td>
<td>MOV DPTR, #TIPO</td>
</tr>
<tr>
<td>0006</td>
<td>120065</td>
<td>CALL PUT_STRING</td>
</tr>
<tr>
<td>0011</td>
<td>120061</td>
<td>CALL PUT_CRLF</td>
</tr>
<tr>
<td>0014</td>
<td>900BD3</td>
<td>MOV DPTR, #NUMB</td>
</tr>
<tr>
<td>0017</td>
<td>12006C</td>
<td>CALL PUT_STRING</td>
</tr>
<tr>
<td>001A</td>
<td>120061</td>
<td>CALL PUT_CRLF</td>
</tr>
<tr>
<td>001D</td>
<td>7832</td>
<td>MOV RO, 'FIRST_NUMBER</td>
</tr>
<tr>
<td>001F</td>
<td>120077</td>
<td>CALL GET_NUMB</td>
</tr>
<tr>
<td>0022</td>
<td>120061</td>
<td>CALL PUT_CRLF</td>
</tr>
<tr>
<td>0025</td>
<td>900BB8</td>
<td>MOV DPTR, #NUMB</td>
</tr>
<tr>
<td>0028</td>
<td>12006C</td>
<td>CALL PUT_STRING</td>
</tr>
<tr>
<td>002B</td>
<td>120061</td>
<td>CALL PUT_CRLF</td>
</tr>
<tr>
<td>002E</td>
<td>783C</td>
<td>MOV RO, #SECOND_NUMBER</td>
</tr>
<tr>
<td>0030</td>
<td>120077</td>
<td>CALL GET_NUMB</td>
</tr>
<tr>
<td>0033</td>
<td>120061</td>
<td>CALL PUT_CRLF</td>
</tr>
<tr>
<td>0036</td>
<td>7932</td>
<td>MOV A, FIRST_NUMBER</td>
</tr>
<tr>
<td>0038</td>
<td>1200BF</td>
<td>CALL ASCBIN</td>
</tr>
<tr>
<td>003B</td>
<td>793C</td>
<td>MOV A, SECOND_NUMBER</td>
</tr>
<tr>
<td>003D</td>
<td>1200BF</td>
<td>CALL ASCBIN</td>
</tr>
<tr>
<td>0040</td>
<td>6532</td>
<td>MOV A, FIRST_NUMBER</td>
</tr>
<tr>
<td>0042</td>
<td>253C</td>
<td>ADD A, SECOND_NUMBER</td>
</tr>
<tr>
<td>0044</td>
<td>F532</td>
<td>MOV FIRST_NUMBERS, A</td>
</tr>
<tr>
<td>0046</td>
<td>7932</td>
<td>MOV A, FIRST_NUMBER</td>
</tr>
<tr>
<td>0048</td>
<td>120099</td>
<td>CALL BINASC</td>
</tr>
<tr>
<td>004B</td>
<td>900CC02</td>
<td>MOV DPTR, #NUM</td>
</tr>
<tr>
<td>004E</td>
<td>12006C</td>
<td>MOV A, #SUM</td>
</tr>
<tr>
<td>0051</td>
<td>6404</td>
<td>CALL PUT_STRING</td>
</tr>
<tr>
<td>0053</td>
<td>7932</td>
<td>MOV A, #FIRST_NUMBER</td>
</tr>
<tr>
<td>0055</td>
<td>ET</td>
<td>PUT_SUM:</td>
</tr>
<tr>
<td>0056</td>
<td>100091</td>
<td>CALL PUT_CHAR</td>
</tr>
<tr>
<td>0059</td>
<td>09</td>
<td>INC A</td>
</tr>
<tr>
<td>005A</td>
<td>8A99</td>
<td>DJNZ A, #PT</td>
</tr>
<tr>
<td>005C</td>
<td>120061</td>
<td>CALL PUT_CRLF</td>
</tr>
<tr>
<td>005F</td>
<td>BOAA</td>
<td>JMP START</td>
</tr>
</tbody>
</table>

This part of program starts communication and gets first number.

Figure G-1. Sample Program (Cont'd.)
BEGIN SERVICE ROUTINES

MOV A,lcR
RETURN AND A LINE FEED (DOH, OAH)

CALL PUT_CHAR

MOV A,ILF
CALL PUT_CHAR
RET

This routine outputs a character string located in code memory. The address must be contained in code memory.

EXIT:

GET_NUMB:

CALL GET_CHAR

MOV A,@A+DPTR
CALL PUT_CHAR
JZ EXIT
INC DPTR
JMP PUT_STRING
RET

This routine gets a single character from the console and stores the string in memory at the address contained in RO.

SKIP:

MOV R2,14
CALL GET_CHAR
CLR ACC.7
CJNE A,#19H,SKIP
CALL PUT_CRLF
JMP GET_NUMB

MOV @RO,A
DJNZ R2,GET_LOOP
RET

COMPARE TO SKIP

This routine gets a single character from the console.

JNB RI,$ ; LOOP HERE UNTIL CHARACTER RECEIVED
CLR RI
MOV A,SBUF
RET

This routine outputs a single character to the console.

JNB TI,$ ; LOOP HERE UNTIL CHARACTER TRANSMITTED
CLR TI
MOV SBUF,A
RET

This routine takes a character string from the console and stores the string in memory at the address contained in RO. If a 'x' is received it starts over.

Sample Program

Figure G-1. Sample Program (Cont'd.)
This routine converts a binary 2's complement number to a 4 character ASCII string.

**INPUT:**
The binary value must be located in memory at the address contained in register 1.

**OUTPUT:**
The 4 character result is placed in memory with the first character at the address contained in register 1.

**NOTES:**
The contents of register A and B will be destroyed.
The contents of the memory location initially addressed by register 1 will be replaced with the first character in the resulting character string.

**BINASC:**

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>00E7</td>
<td>0099</td>
<td>E7</td>
<td>MOV A,#UMB_PTR</td>
</tr>
<tr>
<td>00E8</td>
<td>009A</td>
<td>T72B</td>
<td>MOV #UMB_PTR,#PLUS</td>
</tr>
<tr>
<td>00E9</td>
<td>009C</td>
<td>108704</td>
<td>JNB SGN,VAL</td>
</tr>
<tr>
<td>00E7</td>
<td>009F</td>
<td>772D</td>
<td>MOV #UMB_PTR,#MINUS</td>
</tr>
<tr>
<td>00E7</td>
<td>00A0</td>
<td>14</td>
<td>MOV ,A</td>
</tr>
<tr>
<td>00A0</td>
<td>00A1</td>
<td>FA</td>
<td>CPL A</td>
</tr>
<tr>
<td>00A0</td>
<td>00A2</td>
<td>14</td>
<td>MOV B,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00A3</td>
<td>09</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00A4</td>
<td>75F06A</td>
<td>MOV B,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00A5</td>
<td>84</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00A6</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00A7</td>
<td>75F000</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00A8</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00A9</td>
<td>75F00A</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00AA</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00AB</td>
<td>75F00B</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00AC</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00AD</td>
<td>75F00C</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00AE</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00AF</td>
<td>75F00D</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00B0</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00B1</td>
<td>75F00E</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00B2</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00B3</td>
<td>75F010</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00B4</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00B5</td>
<td>75F011</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00B6</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00B7</td>
<td>75F012</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00B8</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00B9</td>
<td>75F013</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00BA</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00BB</td>
<td>75F014</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00BC</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00BD</td>
<td>75F015</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00BE</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00BF</td>
<td>75F016</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00C0</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00C1</td>
<td>75F017</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00C2</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00C3</td>
<td>75F018</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00C4</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00C5</td>
<td>75F019</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00C6</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00C7</td>
<td>75F01A</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00C8</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00C9</td>
<td>75F01B</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00CA</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00CB</td>
<td>75F01C</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00CC</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00CD</td>
<td>75F01D</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00CE</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00CF</td>
<td>75F01E</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00D0</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00D1</td>
<td>75F01F</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00D2</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00D3</td>
<td>75F020</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00D4</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00D5</td>
<td>75F021</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00D6</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00D7</td>
<td>75F022</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00D8</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00D9</td>
<td>75F023</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00DA</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00DB</td>
<td>75F024</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00DC</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00DD</td>
<td>75F025</td>
<td>MOV A,B</td>
</tr>
<tr>
<td>00A0</td>
<td>00DE</td>
<td>243D</td>
<td>MOV A,#100</td>
</tr>
<tr>
<td>00A0</td>
<td>00DF</td>
<td>75F026</td>
<td>MOV A,B</td>
</tr>
</tbody>
</table>

Figure G-1. Sample Program (Cont'd.)
This routine takes a 4 character string located in memory and converts it to a binary 2’s complement number. The number must begin with a sign character (‘+’ or ‘-’), and be between -128 and +127.

Four ASCII characters a sign character followed a ‘0’ or a ‘1’ and the last 2 characters can be any digit.

The contents of register 1 must point to the sign character.

OUTPUT: A binary 2’s complement representation of the value of the character string.

Register 1 contains the address of the binary value.

NOTES: The contents of the memory location initially addressed by register 1 is destroyed.
The contents of registers 7 and 8 and the accumulator are destroyed.

REG:

TEMP EQU R7

; Go right to number compute sign at end
INC NUMB_PTR
MOV A,fNUMB_PTR
SUBB A,#ZERO
MOV B,’100
MUL AB

; Store first digit's value and go to next digit
INC NUMB_PTR
MOV A,@NUMB_PTR
SUBB A,#ZERO
MOV B,’10
MUL AB

; Add first digit value to second store and go to third digit
ADD A,TEMP
INC NUMB_PTR
MOV A,@NUMB_PTR
SUBB A,#ZERO
ADD A,TEMP
INC NUMB_PTR
MOV A,@NUMB_PTR
TEST for sign value
DEC NUMB_PTR
DEC NUMB_PTR
DEC NUMB_PTR
MOV A,#POS;

POS: MOV A,TEMP
; store result and return
MOV @NUMB_PTR,A
RET

Figure G-1. Sample Program (Cont’d.)
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.

<table>
<thead>
<tr>
<th>GRAPHIC OR CONTROL</th>
<th>ASCII (HEXADECIMAL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>00</td>
</tr>
<tr>
<td>SOH</td>
<td>01</td>
</tr>
<tr>
<td>STX</td>
<td>02</td>
</tr>
<tr>
<td>ETX</td>
<td>03</td>
</tr>
<tr>
<td>EOT</td>
<td>04</td>
</tr>
<tr>
<td>ENQ</td>
<td>05</td>
</tr>
<tr>
<td>ACK</td>
<td>06</td>
</tr>
<tr>
<td>BEL</td>
<td>07</td>
</tr>
<tr>
<td>BS</td>
<td>08</td>
</tr>
<tr>
<td>HT</td>
<td>09</td>
</tr>
<tr>
<td>LF</td>
<td>0A</td>
</tr>
<tr>
<td>VT</td>
<td>0B</td>
</tr>
<tr>
<td>FF</td>
<td>0C</td>
</tr>
<tr>
<td>CR</td>
<td>0D</td>
</tr>
<tr>
<td>SO</td>
<td>0E</td>
</tr>
<tr>
<td>SI</td>
<td>0F</td>
</tr>
<tr>
<td>DLE</td>
<td>10</td>
</tr>
<tr>
<td>DC1 (X-OFF)</td>
<td>11</td>
</tr>
<tr>
<td>DC2 (TAPE)</td>
<td>12</td>
</tr>
<tr>
<td>DC3 (X-ON)</td>
<td>13</td>
</tr>
<tr>
<td>DC4 (X-ON)</td>
<td>14</td>
</tr>
<tr>
<td>NAK</td>
<td>15</td>
</tr>
<tr>
<td>SYN</td>
<td>16</td>
</tr>
<tr>
<td>ETB</td>
<td>17</td>
</tr>
<tr>
<td>CAN</td>
<td>18</td>
</tr>
<tr>
<td>EM</td>
<td>19</td>
</tr>
<tr>
<td>SUB</td>
<td>1A</td>
</tr>
<tr>
<td>ESC</td>
<td>1B</td>
</tr>
<tr>
<td>FS</td>
<td>1C</td>
</tr>
<tr>
<td>GS</td>
<td>1D</td>
</tr>
<tr>
<td>RS</td>
<td>1E</td>
</tr>
<tr>
<td>US</td>
<td>1F</td>
</tr>
<tr>
<td>SP</td>
<td>20</td>
</tr>
<tr>
<td>!</td>
<td>21</td>
</tr>
<tr>
<td>&quot;</td>
<td>22</td>
</tr>
<tr>
<td>#</td>
<td>23</td>
</tr>
<tr>
<td>$</td>
<td>24</td>
</tr>
<tr>
<td>%</td>
<td>25</td>
</tr>
<tr>
<td>&amp;</td>
<td>26</td>
</tr>
<tr>
<td>(</td>
<td>27</td>
</tr>
<tr>
<td>)</td>
<td>28</td>
</tr>
<tr>
<td>*</td>
<td>29</td>
</tr>
<tr>
<td>+</td>
<td>2B</td>
</tr>
<tr>
<td>,</td>
<td>2C</td>
</tr>
<tr>
<td>-</td>
<td>2D</td>
</tr>
<tr>
<td>.</td>
<td>2E</td>
</tr>
<tr>
<td>/</td>
<td>2F</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
</tr>
<tr>
<td>:</td>
<td>3A</td>
</tr>
<tr>
<td>;</td>
<td>3B</td>
</tr>
<tr>
<td>&lt;</td>
<td>3C</td>
</tr>
<tr>
<td>=</td>
<td>3D</td>
</tr>
<tr>
<td>&gt;</td>
<td>3E</td>
</tr>
<tr>
<td>?</td>
<td>3F</td>
</tr>
<tr>
<td>@</td>
<td>40</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
</tr>
<tr>
<td>E</td>
<td>45</td>
</tr>
<tr>
<td>F</td>
<td>46</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
</tr>
<tr>
<td>I</td>
<td>49</td>
</tr>
<tr>
<td>J</td>
<td>4A</td>
</tr>
<tr>
<td>K</td>
<td>4B</td>
</tr>
<tr>
<td>L</td>
<td>4C</td>
</tr>
<tr>
<td>M</td>
<td>4D</td>
</tr>
<tr>
<td>N</td>
<td>4E</td>
</tr>
<tr>
<td>O</td>
<td>4F</td>
</tr>
<tr>
<td>P</td>
<td>50</td>
</tr>
<tr>
<td>Q</td>
<td>51</td>
</tr>
<tr>
<td>R</td>
<td>52</td>
</tr>
<tr>
<td>S</td>
<td>53</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
</tr>
<tr>
<td>U</td>
<td>55</td>
</tr>
<tr>
<td>V</td>
<td>56</td>
</tr>
<tr>
<td>W</td>
<td>57</td>
</tr>
<tr>
<td>X</td>
<td>58</td>
</tr>
<tr>
<td>Y</td>
<td>59</td>
</tr>
<tr>
<td>Z</td>
<td>5A</td>
</tr>
<tr>
<td>[</td>
<td>5B</td>
</tr>
<tr>
<td>]</td>
<td>5C</td>
</tr>
<tr>
<td>)</td>
<td>5D</td>
</tr>
<tr>
<td>A (↑)</td>
<td>5E</td>
</tr>
<tr>
<td>6 (→)</td>
<td>5F</td>
</tr>
<tr>
<td>a</td>
<td>60</td>
</tr>
<tr>
<td>b</td>
<td>61</td>
</tr>
<tr>
<td>c</td>
<td>62</td>
</tr>
<tr>
<td>d</td>
<td>63</td>
</tr>
<tr>
<td>e</td>
<td>64</td>
</tr>
<tr>
<td>f</td>
<td>65</td>
</tr>
<tr>
<td>g</td>
<td>66</td>
</tr>
<tr>
<td>h</td>
<td>67</td>
</tr>
<tr>
<td>i</td>
<td>68</td>
</tr>
<tr>
<td>j</td>
<td>69</td>
</tr>
<tr>
<td>k</td>
<td>6A</td>
</tr>
<tr>
<td>l</td>
<td>6B</td>
</tr>
<tr>
<td>m</td>
<td>6C</td>
</tr>
<tr>
<td>n</td>
<td>6D</td>
</tr>
<tr>
<td>o</td>
<td>6E</td>
</tr>
<tr>
<td>p</td>
<td>6F</td>
</tr>
<tr>
<td>q</td>
<td>70</td>
</tr>
<tr>
<td>r</td>
<td>71</td>
</tr>
<tr>
<td>s</td>
<td>72</td>
</tr>
<tr>
<td>t</td>
<td>73</td>
</tr>
<tr>
<td>u</td>
<td>74</td>
</tr>
<tr>
<td>v</td>
<td>75</td>
</tr>
<tr>
<td>w</td>
<td>76</td>
</tr>
<tr>
<td>x</td>
<td>77</td>
</tr>
<tr>
<td>y</td>
<td>78</td>
</tr>
<tr>
<td>z</td>
<td>79</td>
</tr>
<tr>
<td>{</td>
<td>7A</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>7C</td>
</tr>
<tr>
<td>(ALT MODE)</td>
<td>7D</td>
</tr>
<tr>
<td>DEL (RUB OUT)</td>
<td>7F</td>
</tr>
</tbody>
</table>
### POWERS OF TWO

<table>
<thead>
<tr>
<th>(2^n)</th>
<th>(2^1)</th>
<th>(2^2)</th>
<th>(2^3)</th>
<th>(2^4)</th>
<th>(2^5)</th>
<th>(2^6)</th>
<th>(2^7)</th>
<th>(2^8)</th>
<th>(2^9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.03125</td>
<td>0.015625</td>
<td>0.0078125</td>
<td>0.00390625</td>
<td>0.001953125</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.03125</td>
<td>0.015625</td>
<td>0.0078125</td>
<td>0.00390625</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.03125</td>
<td>0.015625</td>
<td>0.0078125</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.03125</td>
<td>0.015625</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.03125</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1024</td>
<td>512</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Reference Tables

MCS-51
### POWERS OF 16 (IN BASE 10)

<table>
<thead>
<tr>
<th>$16^n$</th>
<th>$n$</th>
<th>$10^{n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.10000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.62500</td>
</tr>
<tr>
<td>256</td>
<td>2</td>
<td>0.39062</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0.24414</td>
</tr>
<tr>
<td>65</td>
<td>4</td>
<td>0.15258</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0.09537</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>0.56904</td>
</tr>
<tr>
<td>268</td>
<td>7</td>
<td>0.37252</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.23238</td>
</tr>
<tr>
<td>68</td>
<td>9</td>
<td>0.14551</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0.090949</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>0.056483</td>
</tr>
<tr>
<td>281</td>
<td>12</td>
<td>0.035527</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>0.022204</td>
</tr>
<tr>
<td>72</td>
<td>14</td>
<td>0.013877</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0.0086736</td>
</tr>
<tr>
<td>152</td>
<td></td>
<td>0.0012379</td>
</tr>
</tbody>
</table>

### POWERS OF 10 (IN BASE 16)

<table>
<thead>
<tr>
<th>$10^n$</th>
<th>$n$</th>
<th>$16^{n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.10000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.1999</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>0.28F5</td>
</tr>
<tr>
<td>3E8</td>
<td>3</td>
<td>0.4189</td>
</tr>
<tr>
<td>2710</td>
<td>4</td>
<td>0.66D8</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0.0A7C5</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>0.10C6</td>
</tr>
<tr>
<td>98</td>
<td>7</td>
<td>0.1AD7</td>
</tr>
<tr>
<td>5F5</td>
<td>8</td>
<td>0.2AF3</td>
</tr>
<tr>
<td>389A</td>
<td>9</td>
<td>0.44B8</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0.26F3</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>0.1AEF</td>
</tr>
<tr>
<td>8E</td>
<td>12</td>
<td>0.2D09</td>
</tr>
<tr>
<td>918</td>
<td>13</td>
<td>0.0C25</td>
</tr>
<tr>
<td>5AF3</td>
<td>14</td>
<td>0.2D09</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>0.480E</td>
</tr>
<tr>
<td>23</td>
<td>16</td>
<td>0.734A</td>
</tr>
<tr>
<td>163</td>
<td>17</td>
<td>0.637B</td>
</tr>
<tr>
<td>DE0</td>
<td>18</td>
<td>0.1272</td>
</tr>
<tr>
<td>8AC7</td>
<td>19</td>
<td>0.01D83</td>
</tr>
</tbody>
</table>

H-3
**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:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 000</td>
<td>4 096</td>
<td>20 000</td>
<td>131 072</td>
</tr>
<tr>
<td>02 000</td>
<td>8 192</td>
<td>30 000</td>
<td>196 608</td>
</tr>
<tr>
<td>03 000</td>
<td>12 288</td>
<td>40 000</td>
<td>262 144</td>
</tr>
<tr>
<td>04 000</td>
<td>16 384</td>
<td>50 000</td>
<td>327 680</td>
</tr>
<tr>
<td>05 000</td>
<td>20 480</td>
<td>60 000</td>
<td>393 216</td>
</tr>
<tr>
<td>06 000</td>
<td>24 576</td>
<td>70 000</td>
<td>458 752</td>
</tr>
<tr>
<td>07 000</td>
<td>28 672</td>
<td>80 000</td>
<td>524 288</td>
</tr>
<tr>
<td>08 000</td>
<td>32 768</td>
<td>90 000</td>
<td>590 824</td>
</tr>
<tr>
<td>09 000</td>
<td>36 864</td>
<td>A0 000</td>
<td>657 360</td>
</tr>
<tr>
<td>0A 000</td>
<td>40 960</td>
<td>B0 000</td>
<td>723 896</td>
</tr>
<tr>
<td>0B 000</td>
<td>45 056</td>
<td>C0 000</td>
<td>786 432</td>
</tr>
<tr>
<td>0C 000</td>
<td>49 152</td>
<td>D0 000</td>
<td>851 968</td>
</tr>
<tr>
<td>0D 000</td>
<td>53 248</td>
<td>E0 000</td>
<td>917 504</td>
</tr>
<tr>
<td>0E 000</td>
<td>57 344</td>
<td>F0 000</td>
<td>983 040</td>
</tr>
<tr>
<td>0F 000</td>
<td>61 440</td>
<td>100 000</td>
<td>1 048 576</td>
</tr>
<tr>
<td>10 000</td>
<td>65 536</td>
<td>200 000</td>
<td>2 097 152</td>
</tr>
<tr>
<td>11 000</td>
<td>69 632</td>
<td>300 000</td>
<td>3 145 728</td>
</tr>
<tr>
<td>12 000</td>
<td>73 728</td>
<td>400 000</td>
<td>4 194 304</td>
</tr>
<tr>
<td>13 000</td>
<td>77 824</td>
<td>500 000</td>
<td>5 242 880</td>
</tr>
<tr>
<td>14 000</td>
<td>81 920</td>
<td>600 000</td>
<td>6 291 456</td>
</tr>
<tr>
<td>15 000</td>
<td>86 016</td>
<td>700 000</td>
<td>7 340 032</td>
</tr>
<tr>
<td>16 000</td>
<td>90 112</td>
<td>800 000</td>
<td>8 388 608</td>
</tr>
<tr>
<td>17 000</td>
<td>94 208</td>
<td>900 000</td>
<td>9 437 184</td>
</tr>
<tr>
<td>18 000</td>
<td>98 304</td>
<td>A0 000</td>
<td>1 048 576</td>
</tr>
<tr>
<td>19 000</td>
<td>102 400</td>
<td>B0 000</td>
<td>2 097 152</td>
</tr>
<tr>
<td>1A 000</td>
<td>106 496</td>
<td>C0 000</td>
<td>3 145 728</td>
</tr>
<tr>
<td>1B 000</td>
<td>110 592</td>
<td>D0 000</td>
<td>4 194 304</td>
</tr>
<tr>
<td>1C 000</td>
<td>114 688</td>
<td>E0 000</td>
<td>5 242 880</td>
</tr>
<tr>
<td>1D 000</td>
<td>118 784</td>
<td>F0 000</td>
<td>6 291 456</td>
</tr>
<tr>
<td>1E 000</td>
<td>122 880</td>
<td>100 000</td>
<td>7 340 032</td>
</tr>
<tr>
<td>1F 000</td>
<td>126 976</td>
<td>200 000</td>
<td>8 388 608</td>
</tr>
</tbody>
</table>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 0000 | 0001 | 0002 | 0003 | 0004 | 0005 | 0006 | 0007 | 0008 | 0009 | 0010 | 0011 | 0012 | 0013 | 0014 | 0015 |
| 01 | 0016 | 0017 | 0018 | 0019 | 0020 | 0021 | 0022 | 0023 | 0024 | 0025 | 0026 | 0027 | 0028 | 0029 | 0030 | 0031 |
| 02 | 0032 | 0033 | 0034 | 0035 | 0036 | 0037 | 0038 | 0039 | 0040 | 0041 | 0042 | 0043 | 0044 | 0045 | 0046 | 0047 |
| 03 | 0048 | 0049 | 0050 | 0051 | 0052 | 0053 | 0054 | 0055 | 0056 | 0057 | 0058 | 0059 | 0060 | 0061 | 0062 | 0063 |
| 04 | 0064 | 0065 | 0066 | 0067 | 0068 | 0069 | 0070 | 0071 | 0072 | 0073 | 0074 | 0075 | 0076 | 0077 | 0078 | 0079 |
| 05 | 0080 | 0081 | 0082 | 0083 | 0084 | 0085 | 0086 | 0087 | 0088 | 0089 | 0090 | 0091 | 0092 | 0093 | 0094 | 0095 |
| 06 | 0096 | 0097 | 0098 | 0099 | 0100 | 0101 | 0102 | 0103 | 0104 | 0105 | 0106 | 0107 | 0108 | 0109 | 0110 | 0111 |
| 07 | 0112 | 0113 | 0114 | 0115 | 0116 | 0117 | 0118 | 0119 | 0120 | 0121 | 0122 | 0123 | 0124 | 0125 | 0126 | 0127 |
| 08 | 0128 | 0129 | 0130 | 0131 | 0132 | 0133 | 0134 | 0135 | 0136 | 0137 | 0138 | 0139 | 0140 | 0141 | 0142 | 0143 |
| 09 | 0144 | 0145 | 0146 | 0147 | 0148 | 0149 | 0150 | 0151 | 0152 | 0153 | 0154 | 0155 | 0156 | 0157 | 0158 | 0159 |
| 0A | 0160 | 0161 | 0162 | 0163 | 0164 | 0165 | 0166 | 0167 | 0168 | 0169 | 0170 | 0171 | 0172 | 0173 | 0174 | 0175 |
| 0B | 0176 | 0177 | 0178 | 0179 | 0180 | 0181 | 0182 | 0183 | 0184 | 0185 | 0186 | 0187 | 0188 | 0189 | 0190 | 0191 |
| 0C | 0192 | 0193 | 0194 | 0195 | 0196 | 0197 | 0198 | 0199 | 0200 | 0201 | 0202 | 0203 | 0204 | 0205 | 0206 | 0207 |
| 0D | 0208 | 0209 | 0210 | 0211 | 0212 | 0213 | 0214 | 0215 | 0216 | 0217 | 0218 | 0219 | 0220 | 0221 | 0222 | 0223 |
| 0E | 0224 | 0225 | 0226 | 0227 | 0228 | 0229 | 0230 | 0231 | 0232 | 0233 | 0234 | 0235 | 0236 | 0237 | 0238 | 0239 |
| 0F | 0240 | 0241 | 0242 | 0243 | 0244 | 0245 | 0246 | 0247 | 0248 | 0249 | 0250 | 0251 | 0252 | 0253 | 0254 | 0255 |
MCS-Sl

Reference Tables

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)
F

100
110
120
130

1
0
2
0256 0257 0258
0272 0273 0274
0288 0289 0290
0304 0305 0306

3
0259
0275
0291
0307

4
0260
0276
0292
0308

5
0261
0277
0293
0309

6
0262
0278
0294
0310

7
0263
0279
0295
0311

8
9
A
0264 0265 0266
0280 0281 0282
0296 0297 0298
0312 0313 0314

8
0267
0283
0299
0315

C
0268
0284
0300
0316

0
0269
0285
0301
0317

E
0270
0286
0302
0318

0271
0287
0303
0319

140
150
160
170

0320
0336
0352
0368

0321
0337
0353
0369

0323
0339
0355
0371

0324
0340
0356
0372

0325
0341
0357
0373

0326
0342
0358
0374

0327
0343
0359
0375

0328 0329 0330 0331
0344 0345 0346 0347
0360 0361 0362 0363
0376 0377 0378 0379

0331
0348
0364
0380

0333
0349
0365
0381

0334
0350
0366
0382

0335
0351
0367
0383

180
190
lAO
180

0384
0400
0416
0432

0385 0386 0387
0401 ~02 0403
0417 0418 0419
0433 0434 0435

0388
0404
0420
0436

0389
0405
0421
0437

0390
0406
0422
0438

0391
0407
0423
0439

0392 0393
0408 0409
0424 0425
0440 0441

0394 0395
0410 0411
0426 0427
0442 0443

0396
0412
0428
0444

0397
0413
0429
0445

0398
0414
0430
0446

0399
0415
0431
0447

lCO

100
lEO
lFO

0448
0464
0480
0496

0449
0465
0481
0497

0450
0466
0482
0498

0451
0467
0483
0499

0452
0468
0484
0500

0453 0454 0455
0469 0470 0471
0485 0486 0487
0501 0502 0503

0456
0472
0488
0504

0457
0473
0489
0505

0458 0459
0474 0475
0490 0491
0506 050"1

0460
0476
0492
0508

0461
0477
0493
0509

0462
0478
0494
0510

0463
0479
0495
0511

200
210
220
230

0512
0528
0544
0560

0513
0529
0545
0561

0514
0530
0546
0562

0515
0531
0547
0563

0516
0532
0548
0564

0517 0518 0519
0533 0534 0535
0549 0550 0551
0565 0566 0567

0520
0536
0552
0568

0521
0537
0553
0569

0522
0538
0554
0570

0523
0539
0555
0571

0524
0540
0556
0572

0525
0541
0557
0573

0526
0542
0558
0574

0527
0543
0559
0575

240
250
260
270

0576
0592
0608
0624

0577
0593
0609
0625

0578
0594
0610
0626

0579
0595
0611
0627

0580
0596
0612
0628

0581
0597
0613
0629

0582
0598
0614
0630

0583
0599
0615
0631

0584 0585 0586
0600 0601 0602
0616 0617 0618
0632 0633 0634

0587
0603
0619
0635

0588
0604
0620
0636

0589
0605
0621
0637

0590
0606
0622
0638

0591
0607
0623
0639

280
290
2AO
280

0640
0656
0672
0688

0641
0657
0673
0689

0642
0658
0674
0690

0643
0659
0675
0691

0644
0660
0676
0692

0645
0661
0677
0693

0646
0662
0678
0694

0647
0663
0679
0695

0648
0664
0680
0696

0649
0665
0681
0697

0650
0666
0682
0698

0651
0667
0683
0699

0652
0668
0684
0700

0653
0669
0685
0701

0654
0670
0686
0702

0655
0671
0687
0703

2CO
200
2EO
2FO

0704 0705 0706 0707
0720 0721 0722 0723
0736 0737 0738 0739
0752 0753 0754 0755

0708
0724
0740
0756

0709
0725
0741
0757

0710
0726
0742
0758

0711
0727
0743
0759

0712
0728
0744
0760

0713
0729
0745
0761

0714
0730
0746
0762

0715
0731
0747
0763

0716
0732
0748
0764

0717
0733
0749
0765

0718
0734
0750
0766

0719
0735
0751
0767

300
310
320
330

0768
0784
0800
0816

0769
0785
0301
0817

0770 0771
0786 0787
0802 0803
0818 0819

0772
0788
0804
0820

0773
0789
0805
0821

0774
0790
0806
0822

0775
0791
0807
0823

0776
0792
0808
0824

0777
0793
0809
0825

0778
0794
0810
0826

0779
0795
0811
0827

0780
0796
0812
0828

0781
0797
0813
0829

0782
0798
0814
0830

0783
0799
0815
0831

340
350
360
370

0832
0848
0864
0880

0833
0849
0865
0881

0834
0850
0866
0882

0835
0851
0867
0883

0836 0837
0852 0853
0868 0869
0884 0885

0838
0854
0870
0886

0839
0855
0871
0887

0840
0856
0872
0888

0841
0857
0873
0889

0842
0858
0874
0890

0843
0859
0875
0891

0844
0860
0876
0892

0845
0861
0877
0893

0846
0862
0878
0894

0847
0863
0879
0895

380
390
3AO
380

0896
0212
0928
0944

0897
0913
0929
0945

0898
0914
0930
0946

0899
0915
0931
0947

0900
0916
0932
0948

0901
0917
0933
0949

0902
0918
0934
0950

0903
0919
0935
0951

0904 0905 0906 0907
0920 0921 0922 0923
0936 0937 0938 0939
0952 0953 0954 0955

0908
0924
0940
0956

0909
0925
0941
0957

0910
0926
0942
0958

0911
0927
0943
0959

3CO
300
3EO
3FO

0960 0961 0962 0963
0976 0977 0978 0979
0992 0993 0994 0995
1008 1009 1010 1011

0969 0970 0971
0984 0985 0986 0987
1000 1001 1002 1003
1016 1017 1018 1019

0972
0988
1004
1020

0973 0974 0975
0989 0990 0991
1005 1006 1007
1021 1022 1023

0322
0338
0354
0370

0964 0965 0966 0967
0980 098·1 0982 0983
0996 0997 0998 0999
1012 1013 1014 1015

0968

H-S


## 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 | 1040 | 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 |
| 410 | 1040 | 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 | 1050 | 1051 | 1052 | 1053 | 1054 | 1055 | 1056 | 1057 | 1058 | 1059 | 1060 | 1061 | 1062 | 1063 | 1064 | 1065 |
| 420 | 1056 | 1057 | 1058 | 1059 | 1060 | 1061 | 1062 | 1063 | 1064 | 1065 | 1066 | 1067 | 1068 | 1069 | 1070 | 1071 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 | 1081 |
| 430 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 | 1081 | 1082 | 1083 | 1084 | 1085 | 1086 | 1087 | 1088 | 1089 | 1090 | 1091 | 1092 | 1093 | 1094 | 1095 | 1096 | 1097 |
| 440 | 1098 | 1099 | 1100 | 1101 | 1102 | 1103 | 1104 | 1105 | 1106 | 1107 | 1108 | 1109 | 1110 | 1111 | 1112 | 1113 | 1114 | 1115 | 1116 | 1117 | 1118 | 1119 | 1120 | 1121 | 1122 | 1123 |
| 450 | 1124 | 1125 | 1126 | 1127 | 1128 | 1129 | 1130 | 1131 | 1132 | 1133 | 1134 | 1135 | 1136 | 1137 | 1138 | 1139 | 1140 | 1141 | 1142 | 1143 | 1144 | 1145 | 1146 | 1147 | 1148 | 1149 |
| 460 | 1150 | 1151 | 1152 | 1153 | 1154 | 1155 | 1156 | 1157 | 1158 | 1159 | 1160 | 1161 | 1162 | 1163 | 1164 | 1165 | 1166 | 1167 | 1168 | 1169 | 1170 | 1171 | 1172 | 1173 | 1174 | 1175 |
| 470 | 1176 | 1177 | 1178 | 1179 | 1180 | 1181 | 1182 | 1183 | 1184 | 1185 | 1186 | 1187 | 1188 | 1189 | 1190 | 1191 | 1192 | 1193 | 1194 | 1195 | 1196 | 1197 | 1198 | 1199 | 1200 | 1201 |
| 480 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 | 1209 | 1210 | 1211 | 1212 | 1213 | 1214 | 1215 | 1216 | 1217 | 1218 | 1219 | 1220 | 1221 | 1222 | 1223 | 1224 | 1225 | 1226 | 1227 |
| 490 | 1228 | 1229 | 1230 | 1231 | 1232 | 1233 | 1234 | 1235 | 1236 | 1237 | 1238 | 1239 | 1240 | 1241 | 1242 | 1243 | 1244 | 1245 | 1246 | 1247 | 1248 | 1249 | 1250 | 1251 | 1252 | 1253 |
| 500 | 1254 | 1255 | 1256 | 1257 | 1258 | 1259 | 1260 | 1261 | 1262 | 1263 | 1264 | 1265 | 1266 | 1267 | 1268 | 1269 | 1270 | 1271 | 1272 | 1273 | 1274 | 1275 | 1276 | 1277 | 1278 | 1279 |

### Reference Tables

- LCS-1
- H-6
- MCS-51

---

**H-6**
### Hexadecimal-Decimal Integer Conversion (Cont'd)

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0100 1100 1101 1000 1000 1000 1000 1100 1100 1100 1100 1100 1100 1100 1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0001 0101 0101 0001 0001 0001 0001 0101 0101 0101 0101 0101 0101 0101 0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0010 1010 1010 0110 0110 0110 0110 1010 1010 1010 1010 1010 1010 1010 1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>0011 1111 1111 0011 0011 0011 0011 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**References:**

- MCS-51 Reference Tables
- **HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)**
|   | A00 | A10 | A20 | A30 | A40 | A50 | A60 | A70 | A80 | A90 | AAO | ABO | B00 | B10 | B20 | B30 | B40 | B50 | B60 | B70 | B80 | B90 | BAO | BAO | B90 | C00 | C10 | C20 | C30 | C40 | C50 | C60 | C70 | C80 | C90 | CAO | CBO | CFO |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 2560 | 2561 | 2562 | 2563 | 2564 | 2565 | 2566 | 2567 | 2568 | 2569 | 2570 | 2571 | 2572 | 2573 | 2574 | 2575 | 2576 | 2577 | 2578 | 2579 | 2580 | 2581 | 2582 | 2583 | 2584 | 2585 | 2586 | 2587 | 2588 | 2589 | 2590 | 2591 |
| 1 | 2608 | 2609 | 2610 | 2611 | 2612 | 2613 | 2614 | 2615 | 2616 | 2617 | 2618 | 2619 | 2620 | 2621 | 2622 | 2623 | 2624 | 2625 | 2626 | 2627 | 2628 | 2629 | 2630 | 2631 | 2632 | 2633 | 2634 | 2635 | 2636 | 2637 | 2638 | 2639 |
| 2 | 2656 | 2657 | 2658 | 2659 | 2660 | 2661 | 2662 | 2663 | 2664 | 2665 | 2666 | 2667 | 2668 | 2669 | 2670 | 2671 | 2672 | 2673 | 2674 | 2675 | 2676 | 2677 | 2678 | 2679 | 2680 | 2681 | 2682 | 2683 | 2684 | 2685 | 2686 | 2687 |
| 3 | 2704 | 2704 | 2705 | 2706 | 2707 | 2708 | 2709 | 2710 | 2711 | 2712 | 2713 | 2714 | 2715 | 2716 | 2717 | 2718 | 2719 | 2720 | 2721 | 2722 | 2723 | 2724 | 2725 | 2726 | 2727 | 2728 | 2729 | 2730 | 2731 | 2732 | 2733 | 2734 | 2735 |
| 4 | 2752 | 2753 | 2754 | 2755 | 2756 | 2757 | 2758 | 2759 | 2760 | 2761 | 2762 | 2763 | 2764 | 2765 | 2766 | 2767 | 2768 | 2769 | 2770 | 2771 | 2772 | 2773 | 2774 | 2775 | 2776 | 2777 | 2778 | 2779 | 2780 | 2781 | 2782 | 2783 | 2784 |
| 5 | 2800 | 2801 | 2802 | 2803 | 2804 | 2805 | 2806 | 2807 | 2808 | 2809 | 2810 | 2811 | 2812 | 2813 | 2814 | 2815 | 2816 | 2817 | 2818 | 2819 | 2820 | 2821 | 2822 | 2823 | 2824 | 2825 | 2826 | 2827 | 2828 | 2829 | 2830 | 2831 |
| 6 | 2880 | 2881 | 2882 | 2883 | 2884 | 2885 | 2886 | 2887 | 2888 | 2889 | 2890 | 2891 | 2892 | 2893 | 2894 | 2895 | 2896 | 2897 | 2898 | 2899 | 2900 | 2901 | 2902 | 2903 | 2904 | 2905 | 2906 | 2907 | 2908 | 2909 | 2910 | 2911 |
| 7 | 2992 | 2993 | 2994 | 2995 | 2996 | 2997 | 2998 | 2999 | 3000 | 3001 | 3002 | 3003 | 3004 | 3005 | 3006 | 3007 | 3008 | 3009 | 3010 | 3011 | 3012 | 3013 | 3014 | 3015 | 3016 | 3017 | 3018 | 3019 | 3020 | 3021 | 3022 | 3023 |
| 8 | 3056 | 3057 | 3058 | 3059 | 3060 | 3061 | 3062 | 3063 | 3064 | 3065 | 3066 | 3067 | 3068 | 3069 | 3070 | 3071 | 3072 | 3073 | 3074 | 3075 | 3076 | 3077 | 3078 | 3079 | 3080 | 3081 | 3082 | 3083 | 3084 | 3085 | 3086 | 3087 |
| 9 | 3136 | 3137 | 3138 | 3139 | 3140 | 3141 | 3142 | 3143 | 3144 | 3145 | 3146 | 3147 | 3148 | 3149 | 3150 | 3151 | 3152 | 3153 | 3154 | 3155 | 3156 | 3157 | 3158 | 3159 | 3160 | 3161 | 3162 | 3163 | 3164 | 3165 | 3166 | 3167 |
| A | 3296 | 3297 | 3298 | 3299 | 3300 | 3301 | 3302 | 3303 | 3304 | 3305 | 3306 | 3307 | 3308 | 3309 | 3310 | 3311 | 3312 | 3313 | 3314 | 3315 | 3316 | 3317 | 3318 | 3319 | 3320 | 3321 | 3322 | 3323 | 3324 | 3325 | 3326 | 3327 |
| B | 3312 | 3313 | 3314 | 3315 | 3316 | 3317 | 3318 | 3319 | 3320 | 3321 | 3322 | 3323 | 3324 | 3325 | 3326 | 3327 | 3328 | 3329 | 3330 | 3331 | 3332 | 3333 | 3334 | 3335 | 3336 | 3337 | 3338 | 3339 | 3340 | 3341 | 3342 | 3343 |

**HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)**

**Reference Tables**

**MCS-51**

**HEX-A-D-DECIMAL INTEGER CONVERSION (Cont'd)**

**Reference Tables**

**MCS-51**
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3328</td>
<td>3329</td>
<td>3330</td>
<td>3331</td>
<td>3332</td>
<td>3333</td>
</tr>
<tr>
<td>1</td>
<td>3334</td>
<td>3335</td>
<td>3336</td>
<td>3337</td>
<td>3338</td>
<td>3339</td>
</tr>
<tr>
<td>2</td>
<td>3340</td>
<td>3341</td>
<td>3342</td>
<td>3343</td>
<td>3344</td>
<td>3345</td>
</tr>
<tr>
<td>3</td>
<td>3346</td>
<td>3347</td>
<td>3348</td>
<td>3349</td>
<td>3350</td>
<td>3351</td>
</tr>
<tr>
<td>4</td>
<td>3352</td>
<td>3353</td>
<td>3354</td>
<td>3355</td>
<td>3356</td>
<td>3357</td>
</tr>
<tr>
<td>5</td>
<td>3358</td>
<td>3359</td>
<td>3360</td>
<td>3361</td>
<td>3362</td>
<td>3363</td>
</tr>
<tr>
<td>6</td>
<td>3364</td>
<td>3365</td>
<td>3366</td>
<td>3367</td>
<td>3368</td>
<td>3369</td>
</tr>
<tr>
<td>7</td>
<td>3370</td>
<td>3371</td>
<td>3372</td>
<td>3373</td>
<td>3374</td>
<td>3375</td>
</tr>
<tr>
<td>8</td>
<td>3376</td>
<td>3377</td>
<td>3378</td>
<td>3379</td>
<td>3380</td>
<td>3381</td>
</tr>
<tr>
<td>9</td>
<td>3382</td>
<td>3383</td>
<td>3384</td>
<td>3385</td>
<td>3386</td>
<td>3387</td>
</tr>
<tr>
<td>A</td>
<td>3388</td>
<td>3389</td>
<td>3390</td>
<td>3391</td>
<td>3392</td>
<td>3393</td>
</tr>
<tr>
<td>B</td>
<td>3394</td>
<td>3395</td>
<td>3396</td>
<td>3397</td>
<td>3398</td>
<td>3399</td>
</tr>
<tr>
<td>C</td>
<td>3400</td>
<td>3401</td>
<td>3402</td>
<td>3403</td>
<td>3404</td>
<td>3405</td>
</tr>
<tr>
<td>D</td>
<td>3406</td>
<td>3407</td>
<td>3408</td>
<td>3409</td>
<td>3410</td>
<td>3411</td>
</tr>
<tr>
<td>E</td>
<td>3412</td>
<td>3413</td>
<td>3414</td>
<td>3415</td>
<td>3416</td>
<td>3417</td>
</tr>
<tr>
<td>F</td>
<td>3418</td>
<td>3419</td>
<td>3420</td>
<td>3421</td>
<td>3422</td>
<td>3423</td>
</tr>
</tbody>
</table>

**HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)**

MCS-51

Reference Tables

H-9
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 #III (ppp), Message

where:

- **eee** is the error number
- **III** 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

A (accumulator), 1-13, 2-2
AC (auxiliary carry flag), 1-9, 1-15, 2-7
ACC (accumulator), 1-14
ACALL code address, 3-4—3-5, B-2, B-9—B-13
see also, CALL, 3-24
LCALL, 3-69, 3-70
ADD
Arithmetic Function, 1-12
A,#data, 3-6, B-2, B-9
A,@Rr, 3-7, B-2, B-9
A,Rr, 3-8, B-2, B-9
A,data address, 3-9, B-2, B-9
ADDC
Arithmetic function, 1-12
A,#data, 3-10, B-2, B-9
A,@Rr, 3-11, 3-12, B-2, B-9
A,Rr, 3-13, B-2, B-9
A,data address, 3-14, B-2, B-9, B-10
Address, Data unit, 1-11
AJMP code address, 3-15, B-2, B-9—B-12
see also, page jump, 2K, 2-8
JMP, 3-59
LMJP, 3-71, 3-72
SJMP, 3-122
ANL
Logical function, 1-12
A,#data, 3-16, B-2, B-10
A,@Rr, 3-17, B-2, B-10
A,Rr, 3-18, B-2, B-10
A,data address, 3-19, B-2, B-10
C, bit address, 3-20, B-2, B-11
C,/bit address, 3-21, B-2, B-11
data address,#data, 3-22, B-2, B-10
data address, A, 3-23, B-2, B-10
Arithmetic and Logic Unit, 1-8, 1-9, 1-12
ASCII Characters
in strings, 2-10, 4-9, 4-10
Codes, H-1

B (multiplication register), 1-9, 1-14
Binary numbers, expressing, 2-9
Bit addressing, 1-14, 2-5—2-7
Bit address space, 1-8, 1-11, 2-6
Bit, Data unit, 1-11
BIT directive, 4-6, C-1
Bit selector (.), 1-14, 2-5
BNF, A-1
Boolean Functions, 1-8
see also, ANL, 3-20, 3-21
CLR, 3-34, 3-35
CPL, 3-37, 3-38
JB, 3-53, 3-54
JBC, 3-55, 3-56
JC, 3-57, 3-58
JNB, 3-61, 3-62
JNC, 3-63, 3-64
MOV, 3-79, 3-84
ORL, 3-106, 3-107
SETB, 3-120, 3-121
BSEG
directive, 4-7, C-1
segment mode, 1-10
see also segment type, 2-1
Byte, Data unit, 1-11

C, 1-13, 2-2
CALL code address, 3-24
see also ACALL, 3-4, 3-5
LCALL, 3-69, 3-70
Character strings in expressions, 2-10, 2-11, 4-9, 4-10
see also ASCII, H-1
CJNE
@Rr,#data address, 3-25, 3-26, B-3, B-12
A,#data ,code address, 3-27, 3-28, B-3, B-12
A,data address ,code address, 3-29, 3-30, B-3, B-12
Rr,#data ,code address, 3-31, 3-32, B-3, B-12
CLR
A, 3-33, B-3, B-12
C, 3-34, B-3, B-12
bit address, 3-35, B-3, B-12
Code Addressing, 2-8, 2-9
Code address space, 1-8
see also CSEG, 1-10
Console I/O built-in macro, 5-16, E-13
Control line, 6-2, A-2
CPL
Logical Function, 1-12
A, 3-36, B-3, B-13
C, 3-37, B-3, B-11
bit address, 3-38, B-3, B-11
CSEG
directive, 4-7, C-1
segment mode, 1-10
see also segment type, 2-1
CY (carry flag), 1-15, 2-7
DA (control) see DATE control
DA
Arithmetic function, 1-12
A, 3-39, B-3, B-12
Data Addressing, 2-3, 2-4
Data address space, 1-8, 1-11
DATA directive, 4-5, C-1
Data Pointer (DPTR), 2-2
DATE control, 6-2, 6-4, D-1
DB (control) see DEBUG control
DB directive, 4-9, C-1
MCS-51

DBIT directive, 4-8, C-1
DEBUG control, 6-2, 6-4, D-1
DEC
Arithmetic function, 1-12
@Rr, 3-40, B-3, B-9
A, 3-41, B-3, B-9
Rr, 3-42, B-3, B-9
data address, 3-43, B-3, B-9
Decimal numbers, expressing, 2-9
Directives
Assembler, 4-1—4-10, C-1
end of program, 4-10
location counter control, 4-7, 4-8
memory initialization, 4-8—4-10
segment control, 4-6, 4-7
symbol definition, 4-4—4-6
DIV
Arithmetic function, 1-12
AB, 3-44, B-3, B-11
DJNZ
Rr, code address, 3-45, B-4, B-12
data address, code address, 3-46, 3-47, B-4, B-12
DPH, 1-9, 1-14
see also Data Pointer, 2-2
DPL, 1-9, 1-14
see also Data Pointer, 2-2
DPTR see Data Pointer, 2-2
DS directive, 4-8, C-1
DSEG
directive, 4-7, C-1
segment mode, 1-10
see also segment type, 2-1
DW directive, 4-10, C-1
EA (Enable All Interrupts), 1-18, 2-7
EJ see EJECT
EJECT control, 6-2, 6-5, D-1
END directive, 4-10, C-1
EP see ERRORPRINT control
EQS built-in macro, 5-11, E-6
EQU directive, 4-4, 4-5
Error messages
Console, printed at
Fatal, 7-2, 7-3
Internal, 7-2
I/O, 7-1
Listing file, printed in
control, 7-11, 7-12
Fatal, 7-13
macro, 7-8—7-11
source, 7-4—7-8
Special, 7-13
ERRORPRINT control, 6-2, 6-5, D-1
ES (Enable Serial port interrupt), 1-18, 2-7
ESCAPE macro function, E-7
ETO (Enable Timer 0 interrupt), 1-18, 2-7
ETI (Enable Timer 1 interrupt), 1-18, 2-7
EVAL built-in macro, 5-10, E-5, E-6
EX0 (Enable external interrupt 0), 1-18, 2-7
EXI (Enable external interrupt 1), 1-18, 2-7
EXIT built-in macro, E-12, E-13
EXT0, 1-17
EXTI, 1-17
External Data address space, 1-8
see also XSEG, 1-10
F0, 1-9, 1-15, 2-7
GE see GEN
GEN
control, 6-2, 6-6, D-1
macro listing format, 7-18
Generic call, 2-9
Generic jump, 2-9
GENONLY
control, 6-2, 6-6, D-1
macro listing format, 7-18
GES built-in macro, 5-11, E-6
grammar, language, A-1
GO see GENONLY
GTS built-in macro, 5-11, E-6
hardware requirements to run ASM51, 1-1
heading format, 7-17
see also DATE, 6-4
TITLE, 6-11
Hexadecimal, 2-9
IC see INCLUDE control
IE (Interrupt Enable), 1-9, 1-14, 1-17, 1-18
IE0 (Interrupt 0 Edge flag), 1-15, 2-7
IEI (Interrupt 1 Edge flag), 1-15, 2-7
IF (built-in macro), 5-12, 5-13, E-11
Immediate Data(#), 2-3
IN built-in macro, 5-16, E-13
INC
Arithmetic function, 1-12
@Rr, 3-48, B-4, B-9
A, 3-49, B-4, B-9
DPTR, 3-50, B-4, B-11
Rr, 3-51, B-4, B-9
data address, 3-52, B-4, B-9
INCLUDE control, 6-3, 6-6, D-1
Indirect addressing (@), 2-2, 2-3
instruction cycle, 1-8
INT0 (Interrupt 0 input pin), 1-16, 2-7
INT1 (Interrupt 1 input pin), 1-16, 2-7
Internal data address space, 1-8
see also DSEG, 1-10
interrupt
control, 1-17—1-18, 1-19
priority, 1-8
invocation line, 6-1
I/O port, 1-8
IP (Interrupt Priority), 1-9, 1-14, 1-17, 1-18
IT0 (Interrupt 0 Type control bit), 1-15, 2-4
IT1 (Interrupt 1 Type control bit), 1-15, 2-4
JB bit address, code address, 3-53, 3-54, B-4, B-9
JBC bit address, code address, 3-55, 3-56, B-4, B-9
JC code address, 3-57, 3-58, B-4, B-10
JMP code address, generic, 3-59
JMP @A+DPTR, 3-60, B-4, B-10
JNB bit address, code address, 3-61, 3-62, B-4, B-9
JNC code address, 3-63, 3-64, B-4, B-10
JNZ code address, 3-65, 3-66, B-4, B-10
JZ code address, 3-67, 3-68, B-5, B-10

Label, 4-3, 4-4
LCALL code address, 3-69, 3-70, B-5, B-9
see also ACALL, 3-4, 3-5
CALL, 3-24

LEN built-in macro, 5-10, E-6
LI see LIST
LIST control, 6-3, 6-7, D-1
listing file, 1-2
format, 7-14—7-19
LJMP code address, 3-71, 3-72, B-5, B-9
see also AJMP, 3-15
JMP, 3-59
SJMP, 3-122
location counter ($), 2-12, 4-1, 4-2
controls, 4-7, 4-8
long jump or call, 2-8
see also LCAL1, 3-69, 3-70
LJMP, 3-71, 3-72

macro
arithmetic expressions in, 5-11, E-3
Call, 5-6—5-9, E-4, E-5
call, 5-6—5-9, E-4, E-5
definition, 5-3, E-7, E-8
delimiters, E-8—E-10
expansion, 5-3, E-3—E-5
listing format, 7-17, 7-18
parameters, 5-5, E-5
—time, 5-4
MACRO control, 6-3, 6-7, D-1
MATCH built-in macro, 5-15, E-14
METACHAR built-in macro, E-15
metacharacter (%), the, 5-2

MOV
@Rr,#data, 3-72, B-5, B-10
@Rr, A, 3-73, B-5, B-13
@Rr.data address, 3-74, B-5, B-11
A,#data, 3-75, B-5, B-10
A,@Rr, 3-76, B-5, B-12
A,Rr, 3-77, B-5, B-12
A.data address, 3-78, B-5, B-12
C,bit address, 3-79, B-5, B-11
DPTR,#data, 3-80, B-5, B-11
Rr,#data, 3-81, B-5, B-11
Rr, A, 3-82, B-5, B-13
Rr.data address, 3-83, B-5, B-11
bit address, C, 3-84, B-5, B-11
data address,#data, 3-85, B-5, B-10
data address,@Rr, 3-86, B-5, B-11
data address, A, 3-87, B-5, B-13
data address,Rr, 3-88, B-6, B-11
data address,data address, 3-89, B-6, B-11
MOVC
A,@A+DPTR, 3-90, B-6, B-11
A,@A+PC, 3-91, 3-92, B-6, B-11
MOVX
@DPTR, A, 3-93, B-6, B-13
@Rr, A, 3-94—3-95, B-6, B-13
A,@DPTR, 3-96, B-6, B-12
A,@Rr, 3-97, 3-98, B-6, B-12
MR see MACRO control
MUL
Arithmetic function, 1-12
AB, 3-99, 3-100, B-6, B-11

nibble, Data unit, 1-11
NODB see NODEBUG control
NODEBUG control, 6-2, 6-4, D-1
NOEP see NOERRORPRINT control
NOERRORPRINT control, 6-2, 6-5, D-1
NOGE see NOGEN control
NOGEN control, 6-2, 6-6, D-1
listing format, 7-18
NOLI see NOLIST control
NOLIST control, 6-3, 6-7, D-1
NOMACRO control, 6-3, 6-7, D-1
NOMR see NOMACRO control
NOOBJECT control, 6-3, 6-8, D-1
NOOJ see NOOBJECT control
NOR, 3-101, B-6, B-9
NOPAGING control, 6-3, 6-8, D-1
NOPI see NOPAGING control
NOPR see NOPRINT control
NOPRINT control, 6-3, 6-10, D-1
NOSB see NOSYMBOLS control
NOSYMBOLS control, 6-3, 6-11, D-2
NOXR see NOXREF control
NOXREF control, 6-3, 6-12, D-2
null string, 2-11, 4-9
Numbers
specifying, 2-9
representation of, 2-10

OBJECT control, 6-3, 6-8, D-1
Object file, 1-2
Octal, 2-9
OJ see OBJECT control
Operators, Assembly-time
Arithmetic, 2-13
Logical, 2-13
Relational, 2-14, 2-15
Special, 2-14
Precedence, 2-15
Operators, macro, 5-11, E-3
ORG directive, 4-8, C-1
ORL
Logical function, 1-12
A,#data, 3-102, B-6, B-10
A,@Rr, 3-103, B-6, B-10
A,Rr, 3-104, B-6, B-10
A.data address, 3-105, B-6, B-10
C,bit address, 3-106, B-6, B-10
C,bit address, 3-107, B-6, B-11
data address,#data, 3-108, B-6, B-10
data address,A, 3-109, B-6, B-10
OUT built-in macro, 5-16, E-13
OV (overflow flag), 1-9, 1-15, 2-7
P (parity flag), 1-9, 1-15, 2-7
page jump or call, 2K, 2-8
see also ACALL, 3-4, 3-5
AJMP, 3-15
PAGING control, 6-3, 6-8, D-2
PAGELength control, 6-3, 6-9, D-2
PAGEWIDTH control, 6-3, 6-9, D-2
PC, 1-9, 1-13, 2-2
see also, program counter, 2-2
PI see PAGING control
PL see PAGELength control
PUSH data address, 3-111, B-7, B-12
Port 0 (P0) see I/O Port, 1-8
Port 1 (P1) see I/O Port, 1-8
Port 2 (P2) see I/O Port, 1-8
Port 3 (P3), 1-16
see also I/O Port, 1-8
PR see PRINT control
PRINT control, 6-3, 6-10, D-2
Program counter, 1-8, 2-2
Program memory, 1-8
see also CSEG, 1-10
Program Status Word (PSW), 1-15
PS (Priority of Serial Port Interrupt), 1-18, 2-7
PSW see Program Status Word, 1-15
P0 (Priority of Timer 0 Interrupt), 1-18, 2-7
P1 (Priority of Timer 1 Interrupt), 1-18, 2-7
PUSH data address, 3-111, B-7, B-12
PW see PAGELength control
PX0 (Priority of External Interrupt 0), 1-18, 2-7
PX1 (Priority of External Interrupt 1), 1-18, 2-7
R0, R1, R2, R3, R4, R5, R6, R7, 1-13, 2-2
see also, registers, General-purpose, 1-12
RAM memory, 1-8
see also DSEG, 1-10
RD (Read Data external), 1-16, 2-7
register
Banks, 1-12
General-purpose, 1-12
Program addressable, 1-13
value at reset, 1-19
Relative Jump, 2-8
Relative offset, 2-8
REN (Receive Enable), 1-17, 2-7
REPEAT built-in macro, 5-12, 5-14, E-11, E-12
RESET, 1-17
RESTORE control, 6-3, 6-10, D-2
RET, 3-112, 3-113, B-7, B-9
RETI, 3-114, 3-115, B-7, B-9
RL A, 3-116, B-7, B-9
RLCA A, 3-117, B-7, B-9
RR A, 3-118, B-7, B-9
RRC A, 3-119, B-7, B-9
RS see RESTORE control
RS0 (Register Select Bit 0), 1-9, 1-12, 1-15, 2-7
RS1 (Register Select Bit 1), 1-9, 1-12, 1-15, 2-7
RXD (Serial Port Receive pin), 1-16, 2-7
SA see SAVE control
SAVE control, 6-3, 6-10
SB see SYMBOLS control
SBUF (Serial Port Buffer), 1-9, 1-14
SCON (Serial Port Control), 1-9, 1-14, 1-17
segment type, 2-1
in expressions, 2-15, 2-16
of operands, 2-3 — 2-5, 2-8, 2-9
of symbols, 4-4 — 4-6
Serial I/O Port, 1-8, 1-9, 1-17
SETB
C, 3-120, B-7, B-12
bit address, 3-121, B-7, B-12
SET built-in macro, 5-16, 5-17
SET directive, 4-5, C-1
SINT, 1-17
SJMP code address, 3-122, B-7, B-11
SM0 (Serial Mode Control bit 0), 1-17, 2-7
SM1 (Serial Mode Control bit 1), 1-17, 2-7
SM2 (Serial Mode Control bit 2), 1-17, 2-7
SP (Stack Pointer), 1-14, 1-19
see also stack, 1-13
Special Assembler symbols, 1-13, 2-2
see also EQU directive, 4-4, 4-5
SET directive, 4-5
stack, 1-13
SUBB
Arithmetic function, 1-12
A, data, 3-123, B-7, B-11
A, @Rr, 3-124 — 3-125, B-7, B-11
A, Rr, 3-126, 127, B-7, B-11
A, data address, 3-128, 3-129, B-7, B-11
SUBSTR built-in macro, 5-17, E-13
SWAP A, 3-130, B-7, B-12
symbol
definition, 4-2, 4-3
see also BIT, 4-6
DATA, 4-5
EQU, 4-4, 4-5
SET, 4-5
XDATA, 4-6
table listing format, 7-19
use of, 2-11, 2-12
SYMBOLS control, 6-3, 6-11, D-2
TITLE control, 6-3, 6-11, D-1
T0 (Timer/counter 0 External flag), 1-16, 2-7
T1 (Timer/counter 1 External flag), 1-16, 2-7
TCON (Timer Control), 1-9, 1-14
TF0 (Timer 0 Overflow Flag), 1-15, 2-7
TF1 (Timer 1 Overflow Flag), 1-15, 2-7
TH0 (Timer 0 high byte), 1-9, 1-14
TH1 (Timer 1 high byte), 1-9, 1-14
TIMERO, 7-17
TIMER1, 7-17
TLO (Timer 0 low byte), 1-9, 1-14
TL1 (Timer 1 low byte), 1-9, 1-14
TMOD (Timer Mode), 1-9, 1-14, 1-15
TR0 (Timer 0 Run control bit), 1-15, 2-7
TR1 (Timer 1 Run control bit), 1-15, 2-7
TT see TITLE control
TXD (Serial Port Transmit bit), 1-16, 2-7

WF see WORKFILES control
WHILE built-in macro, 5-12, 5-14, E-11, E-12
WR (write Data for External Memory), 1-16, 2-7
WORKFILES control, 6-3, 6-12, D-2

XCH
A, @Rr, 3-131, B-7, B-12
A, Rr, 3-132, B-7, B-12
A, data address, 3-133, B-7, B-12
XCHD A, @Rr, 3-134, 3-135, B-8, B-12
XDATA directive, 4-6
XR see XREF control
XREF control, 6-3, 6-12, D-3
XRL
Logical function, 1-12
A, #data, 3-136, B-8, B-10
A, @Rr, 3-137, B-8, B-10
A, Rr, 3-138, B-8, B-10
A, data address, 3-139, B-8, B-10
data address, #data, 3-140, B-8, B-10
data address, A, 3-141/3-142, B-8, B-10
XSEG
directive, 4-7, C-1
segment mode, 1-10
see also, segment type, 2-1
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.

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