INTEL CORP. 3065 Bowers Avenue, Santa Clara, California 95051 • (408) 246-7501

MCS™-8 Assembly Language Programming Manual

PRELIMINARY EDITION

November 1973

-- TABLE OF CONTENTS --

8008 PROGRAMMING MANUAL

Page No.

Page No.

Page No.

-- TERMS --

TERMS	DESCRIPTION
Address	A 14 bit number assigned to a memory location corresponding to its sequential position.
Bit	The smallest unit of information which can be represented. (A bit may be in one of two states, 0 or 1).
Byte	A group of 8 contiguous bits occupying a single memory location.
Console	The INTELLEC 8 front panel, containing switches and indicators that allow a user to operate the computer and monitor program execution.
Instruction	The smallest single operation that the computer can be directed to execute.
Object Program	A program which can be loaded directly into the computer's memory and which requires no alteration before execution. An object program is usually on paper tape, and is produced by assembling (or compiling) a source program. Instructions are represented by binary machine code in an object program.
Program	A sequence of instructions which, taken as a group, allow the computer to accomplish a desired task.
Source Program	A program which is readable by a programmer but which must be transformed into object program format before it can be loaded into the computer and executed. Instructions in an assembly language source program are represented by their assembly language mnemonic.
System Program	A program written to help in the process of creating user programs.
User Program	A program written by the user to make the computer perform any desired task.

TERMS

Word	A group of 16 contiguous bits occupying two successive memory locations. (2 bytes).
nnnnB	nnnn represents a number in binary format.
nnnnD	nnnn represents a number in decimal format.
nnnnO	nnnn represents a number in octal format.
nnnnQ	nnnn represents a number in octal format.
ijklH	ijkl represents a number in hexadecimal format.

1.0 INTRODUCTION

This manual has been written to help a design engineer program the INTEL 8008 microcomputer in assembly language, and to show why it is both economical and practical to do so. Accordingly this manual assumes that the reader has a good understanding of logic, but is completely unfamiliar with programming concepts.

For those readers who do understand programming concepts, several features of the INTEL 8008 microcomputer are described below. They include:

- 8-Bit parallel CPU on a single chip.
- 48 instructions, including extensive memory reference and jump on condition capability.
- Direct addressing for 16,384 bytes of memory.
- Seven 8-bit registers, and a seven 14-bit register stack.

INTEL 8008 microcomputer users will have widely differing programming needs. Some users may wish to write a few short programs, while other users may have extensive programming requirements.

For the user with limited programming needs, three system programs resident on the INTELLEC 8 are provided; they are an Editor, an Assembler, and a System Monitor. Use of the INTELLEC 8 and its three system programs is described in the INTELLEC 8 Operator's Manual.

For the user with extensive programming needs, cross assemblers are available which allow programs to be generated on any computer having a FORTRAN compiler whose word size is 32 bits or greater, limiting INTELLEC 8 use to final checkout of programs only.

Whether a user's programming needs are limited or extensive, this manual describes how to write efficient assembly language programs that may be assembled either on the INTELLEC 8, or using a cross assembler.

The experienced programmer should note that the assembly language has a macro capability which allows users to tailor the assembly language to individual needs, yet still generate object programs which are compatible with any 8008 system. The value of this feature will quickly become apparent to the inexperienced programmer.

2.0 COMPUTER ORGANIZATION

This section provides the programmer with a functional overview of the INTELLEC 8 computer. Information is presented in this section at a level that provides a programmer with necessary background in order to write efficient programs.

Functionally the computer can be divided into the following three blocks, as illustrated in Figure 2-1:

- Central Processing Unit
- Memory
- Input/Output and Output Modules

To the programmer, the computer is better represented as consisting of the following parts:

(1) Seven working registers in which all data operations actually occur, and which may therefore be visualized as a conduit through which all data must flow. The seven working registers form part of the Central Processing Unit.

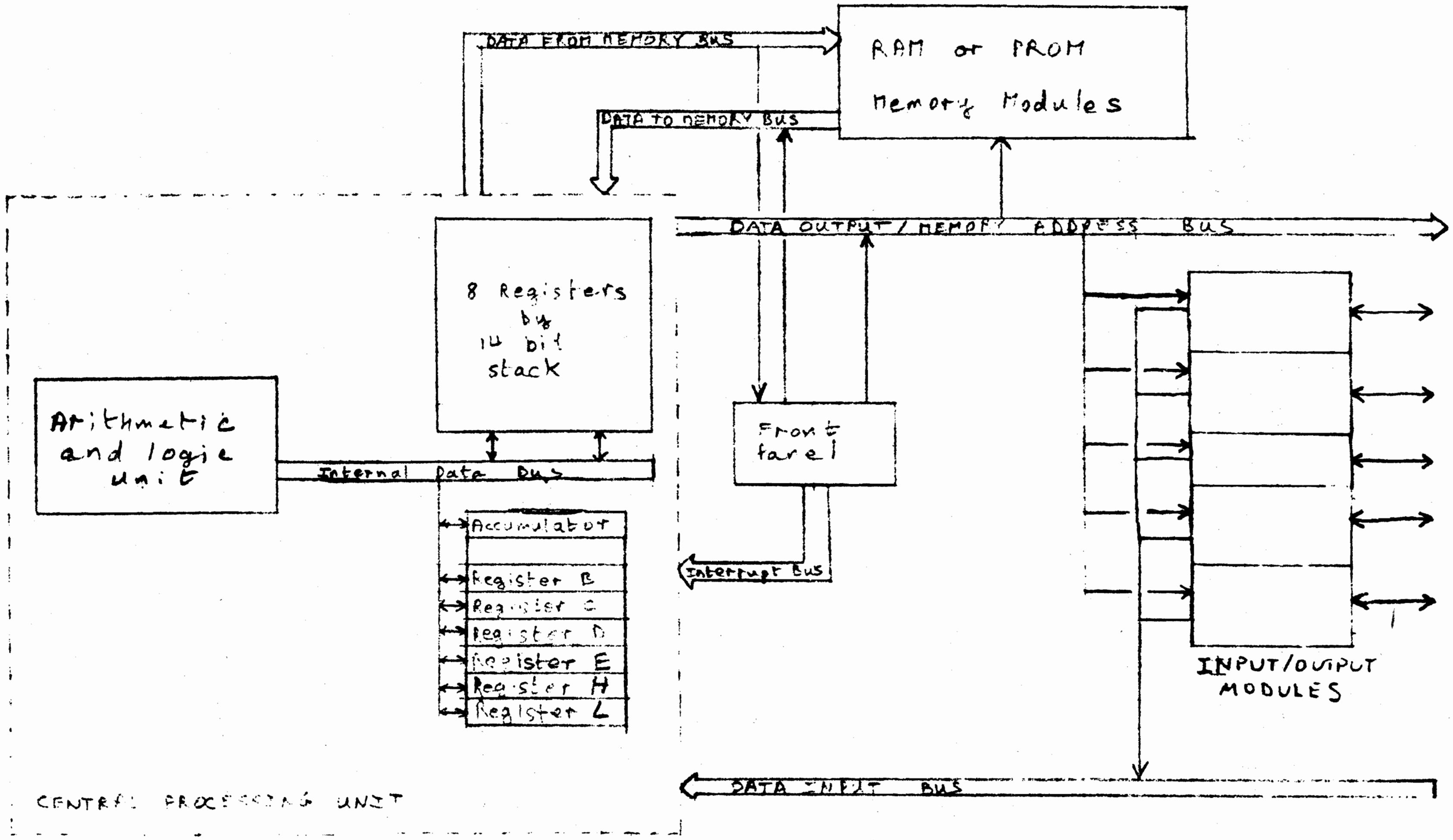
(2) A stack, which is a device used to facilitate execution of subroutines, as described later in this section. The stack forms part of the Central Processing Unit.

(3) An arithmetic and logic unit which executes instructions and forms part of the Central Processing Unit.

(4) Memory, which is a passive depository of data and instructions, and must be addressed, byte by byte, in order to access stored information.

(5) Input/Output, which is the interface between a program and the real, outside world.

The rest of Section 2 explains how the functional blocks of the 8008 computer as illustrated in Figure 2-1 support its programming capabilities.



Thin lines represent Bus extensions. No control connection lines are shown.
A control Bus (not shown) links all modules and functional blocks.

Figure 2-1 Functional Representation of the Intellec 8 Computer

2.1 THE CENTRAL PROCESSING UNIT

The entire 8008 Central Processing Unit is constructed on one LSI chip. It consists of seven working registers, a seven register stack, and logic to enable the INTELLEC 8 instruction set.

Conceptually, the 8008 is a multi-bus machine. Data transfers within the CPU take place via an internal data bus. Data transfers between CPU and memory occur via separate data-from-memory and data-to-memory buses. Separate data input and output buses provide for communication between CPU and peripheral devices.

2.1.1 WORKING REGISTERS

The 8008 provides the programmer with an accumulator and six additional "scratchpad" registers. The special significance of the accumulator is that it is accessed by nearly all arithmetic and logical instructions, whereas only a limited number of data operations can involve the scratchpad registers.

The seven working registers are numbered and referenced via the integers 0, 1, 2, 3, 4, 5, 6; by convention working registers may also be accessed via the letters A (for the accumulator), B, C, D, E, H and L. The H and L registers have special significance in that they are used to store memory addresses, as described in Section 2.4.1.

While in theory working registers could be used by the programmer in any way, certain types of use either lend themselves to efficient programming, or are forced on the programmer by the design of the computer. The following working register assignments are recommended:

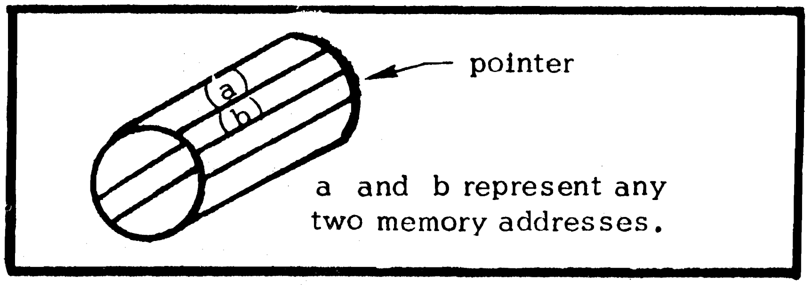
Register A	:	Most mathematical or logical operations act on, and change the contents of this register. Use register A as the destination for data operations, never to save or store data.
Registers B,C,D, and E	:	Use these registers to transfer data between program modules, and to store intermediate answers during extended computations.
Registers H and L	:	Use only for addressing.

For the novice programmer, the above working register assignments will not yet be meaningful, but the rationale for having such assignments will become clear after examining the programming examples of Sections 3 and 4.

2.1.2 THE STACK

The stack consists of seven 14-bit registers used to hold memory addresses. The concept of memory addresses is described in Section 2.2, but briefly stated, memory can be visualized as a sequence of bytes (8 bit data units) numbered sequentially from 0 to the highest memory byte present. The address of a memory byte is the same as its sequential number in memory. Having 14 bits, a stack register can address up to 16,384 bytes of memory. (Addresses run from 0 to 1 1 1 1 1 1 1 1 1 1 1 1 1 1 B = 3FFFH = 16,383D, providing 16,384 memory addresses).

Stack operations consist of writing an address to the stack, and reading an address from the stack. In order to understand these operations, it may be helpful to visualalize the stack as seven registers on the surface of a cylinder, as shown below:



There is no top or bottom to the stack. Every stack register is adjacent to two other stack registers. The 8008 keeps a pointer to the next stack register available.

Writing an Address to the Stack:

To perform a stack write operation;

(1) The address is written into the register indicated by the pointer.

(2) The pointer is advanced to the next sequential register.

Any register may be used to hold the first address written to the stack. More than seven addresses may be written to the stack; however, this will cause a corresponding number of previously stored addresses to be overwritten and lost. This is illustrated in Figure 2-2.

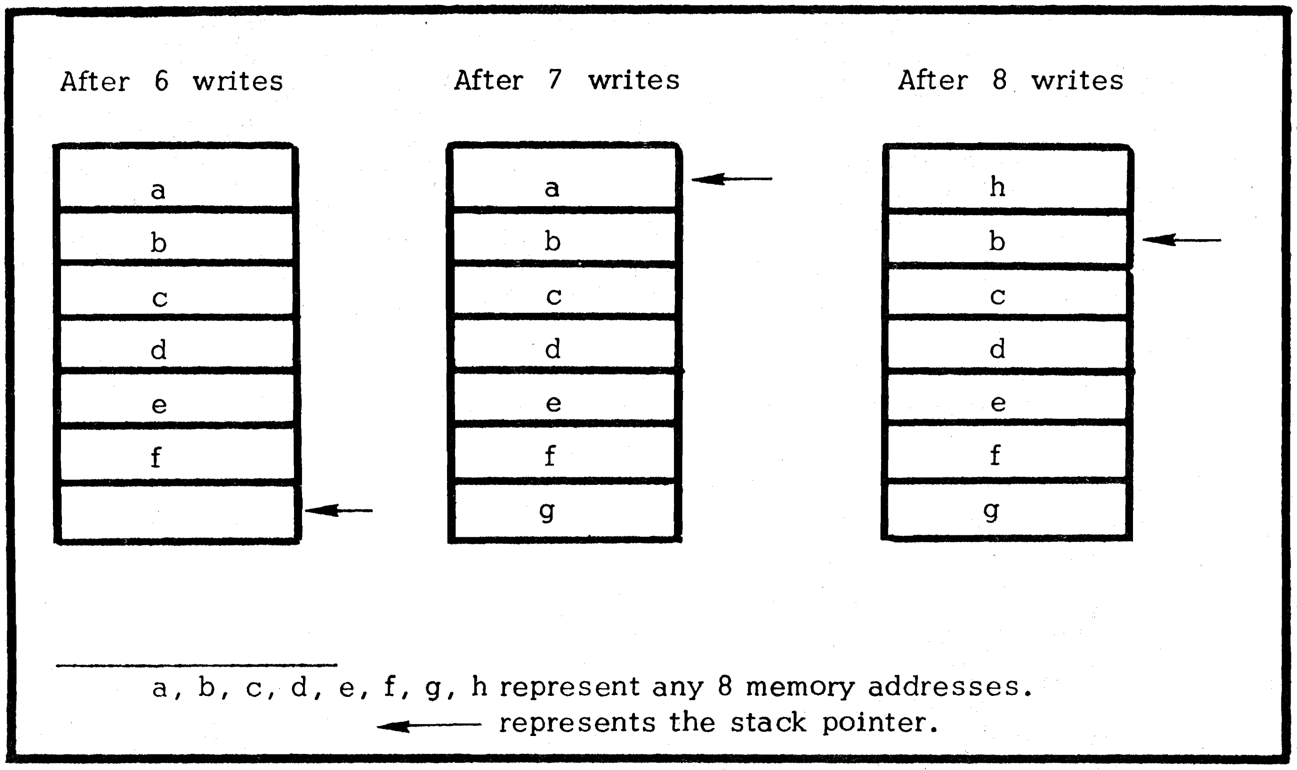


FIGURE 2-2.

Stack Write Operations.

Storing the 8th address (h) overwrites the first address stored (a).

<u>Reading an address from the stack</u>:

To perform a stack read operation;

(1) The pointer is backed up one register.

(2) The memory address indicated by the pointer is read.

The address read remains in the stack undisturbed. Thus, if 8 addresses are written to the stack and then three reads are performed, the stack will appear as in Figure 2-3.

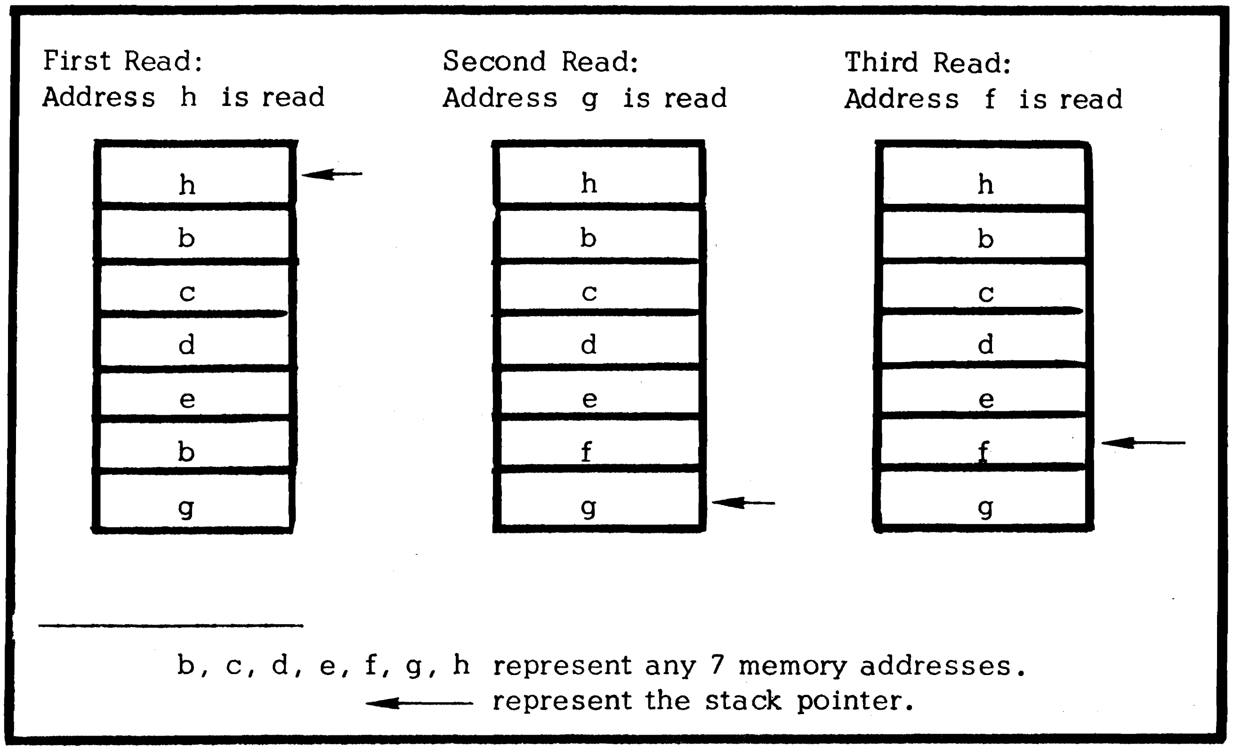


FIGURE 2-3.

Stack Read Operations.

The stack is zeroed when power is first applied to the 8008 or after a RESET operation has occurred; thus if a stack read is performed from a stack register which has not been written, a memory address of 0 will be read.

Section 2.4.5 describes how the stack is used by programs.

2.1.3 ARITHMETIC AND LOGIC UNIT

The arithmetic and logic unit ALU of the 8008 computer provides the logic for executing instructions. The representation of the ALU in Figure 2-1 is sufficient for the programmer, who need know nothing about the ALU in order to program the 8008.

2.2 MEMORY

The 8008 can be used with read only memory, programmable read only memory and read/write memory. A program can cuase data to be read from any type of memory,but can only cause data to be written into read/write memory.

The programmer visualizes memory as a sequence of bytes, each of which may store 8 bits (two hexadecimal digits). Up to 16,384 bytes of memory may be present, and an individual memory byte is addressed by its sequential number, between 0 and 16,383. The hexadecimal degits stored in a memory byte may represent the encoded form of an instruction, or it may be data, as described in Section 3.2.

2.3 COMPUTER PROGRAM REPRESENTATION IN MEMORY

A computer program consists of a sequence of instructions. Each instruction enables an elementary operation such as the movement of a data byte, an arithmetic or logical operation on a data byte, or a change in instruction execution sequence. Instructions are described individually in Section 3.

A program will be stored in memory as a sequence of hexadecimal digits which represent the instructions of the program. The memory address of the next instruction to be executed is recorded in a 14-bit register called the Program Counter and thus it is possible to track a program as it is being executed. Just before each instruction is executed, the program counter is advanced to the address of the next sequential instruction. Program execution proceeds sequentially unless a transfer-of-control instruction (jump or call) is executed, which causes the program counter to be set to a specified address. Execution then continues sequentially from this new address in memory.

Upon examining the contents of a memory byte, there is no way of telling whether the byte contains an encoded instruction or data. For example, the hexadecimal code 1AH has been arbitrarily selected to encode the instruction RAR (rotate the contents of the accumulator right through carry); thus, the value 1AH stored in a memory byte could either represent the instruction RAR, or it could represent the data value 1AH. It is up to the logic of a program to insure that data is not misinterpreted as a instruction code, but this is simply done as follows:

Every program has a starting memory address, which is the memory address of the byte holding the first instruction to be executed. Before the first instruction is executed, the program counter will automatically be advanced to address the next instruction to be executed, and this procedure will be repeated for every instruction in the program. 8008 instructions may require 1, 2, or 3 bytes to encode an instruction; in each case the program counter is automatically advanced to the start of the next instruction, as illustrated in Figure 2-4.

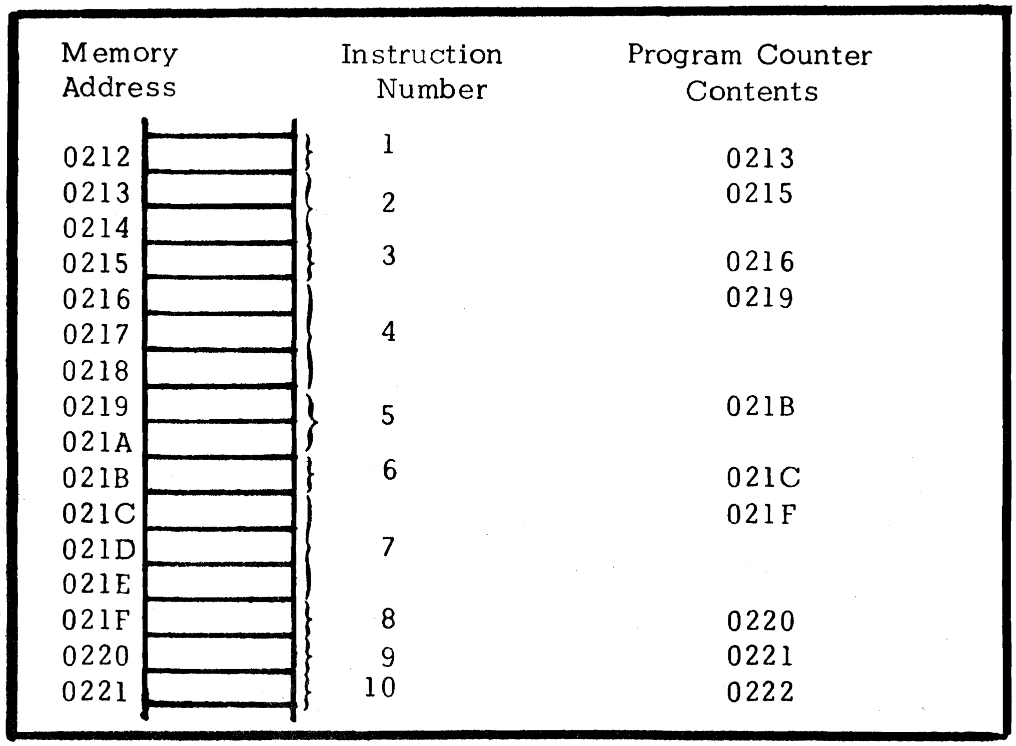


Figure 2-4 Automatic Advance of the Program Counter as Instructions are Executed

In order to avoid errors, the programmer must be sure that a data byte does not follow an instruction when another instruction is expected. Referring to Figure 2-4, an instruction is expected in byte 021FH, since instruction 8 is to be executed after instruction 7. If byte 021FH held data, the program would not execute correctly. Therefore, when writing a program, do not store data in between adjacent instructions that are to be executed consecutively.

NOTE: If a program stores data into a location, that location should not normally appear among any program instructions. This is because user programs are normally executed from read-only memory, into which data cannot be stored.

A class of instructions (referred to as transfer of control instructions) cause program execution to branch to an instruction that may be anywhere in memory. The memory address specified by the transfer of control instruction must be the address of another instruction; if it is the address of a memory byte holding data, the program will not execute correctly. For example, referring to Figure 2-4, say instruction 4 specifies a jump to memory byte 021FH, and say instructions 5, 6 and 7 are replaced by data; then following execution of instruction 4, the program would execute correctly. Buf if, in error, instruction 4 specifies a jump to memory byte 021EH, an error would result, since this byte now holds data. Even if instructions 5, 6 and 7 were not replaced by data, a jump to memory byte 021EH would cause an error, since this is not the first byte of the instruction.

Upon reading Section 3, you will see that it is easy to avoid writing an assembly language program with jump instructions that have erroneous memory addresses. Information on this subject is given rather to help the programmer who is debugging programs by entering hexadecimal codes directly into memory.

2.4 MEMORY ADDRESSING

Each byte in memory has an address which is a 14 bit number corresponding to its sequential location. Thus the range of addresses is 0 to 16,383 (the highest number which can be represented in 14 bits). The address of a memory location can be held in memory in two consecutive 8 bit bytes; normally the least significant 8 bits of the address are held in one byte, and the most significant 6 bits of the address are held in the next byte.

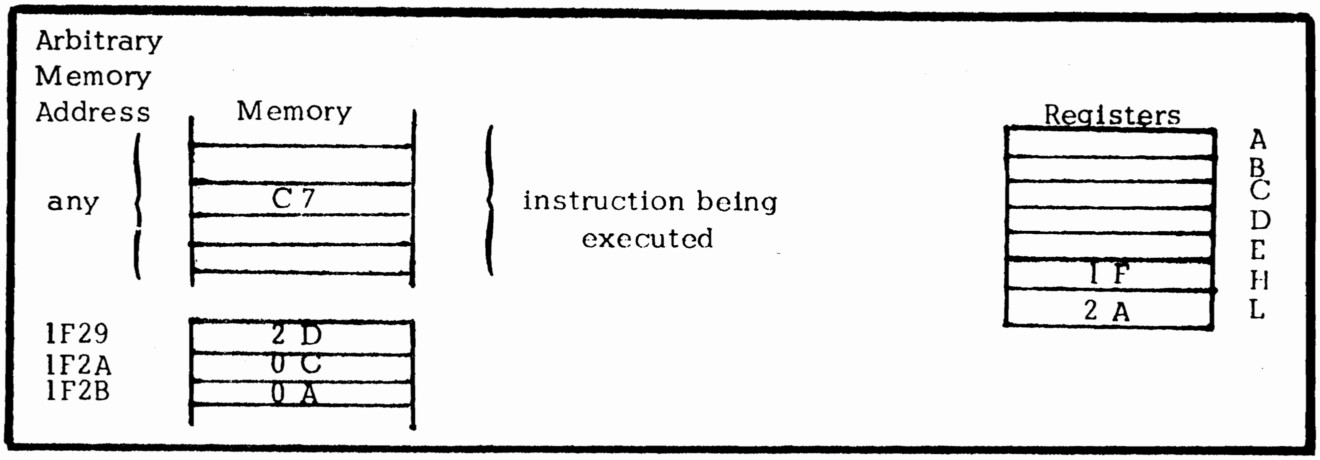
By now it will have become apparent that addressing specific memory bytes constitutes an important part of any computer program; there are a number of ways in which this can be done, as described in the following subsections.

2.4.1 DIRECT ADDRESSING

With direct addressing, as the name implies, an instruction provides an exact memory address. The following instruction provides an example of direct addressing:

"Load the contents of memory byte 1F2AH into the accumulator (Register A)"

1F2AH is a direct address. Direct addressing is the principal means used by the INTELLEC 8 to address memory, and the H and L registers are used to hold the direct memory address. For example, the direct addressing instruction described above might be illustrated as follows:



The instruction encoded by the digits C7H is being executed, and is, in fact, interpreted to mean:

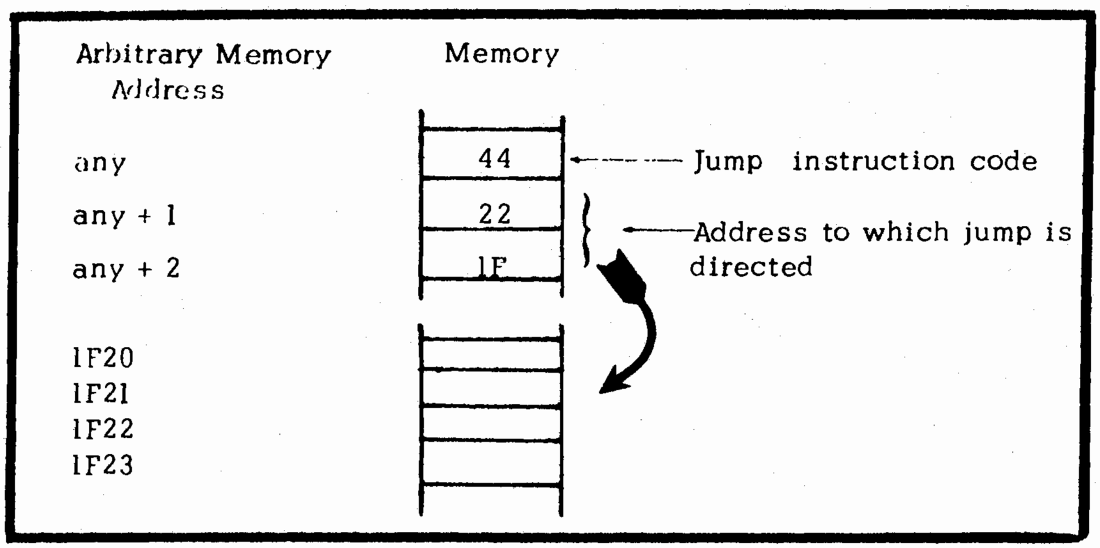
Load register A (the accumulator) with the contents of the memory byte whose address is provided by the H and L registers.

Jump and call instructions on the 8008 provide a special case of direct addressing, where the direct address is stored in the two consecutive memory bytes following the instruction code byte. The low order eight bits of the address are stored in the first (lower addressed) byte, while the high order six bits of the address are stored in the second (higher addressed) byte.

Thus the instruction:

"Jump to memory location 1F22"

would appear in memory as follows:


Arbitrary Memory Address
Memory
any
any + 1
any + 2
44
22
1F
Jump instruction code
Address to which jump is directed
1F20
1F21
1F22
1F23

2.4.2 INDEXED ADDRESSING

An indexed address is computed as the sum of two numbers, a base address and an index. For example, a table may be one hundred bytes long, in which case the address of any byte is computed as the address of the table origin (base address), plus the displacement of the byte from the table origin (index). On the 8008 the H and L registers will commonly be used to hold the base address, while either one or two of the registers B,C,D and E (but not A) will hold the index. If one index register is used, tables cannot be longer than 256 bytes. If two registers are used to store the index portion of the address, tables can be as large as memory. Figure 2-5 illustrates the concept of indexed addressing.

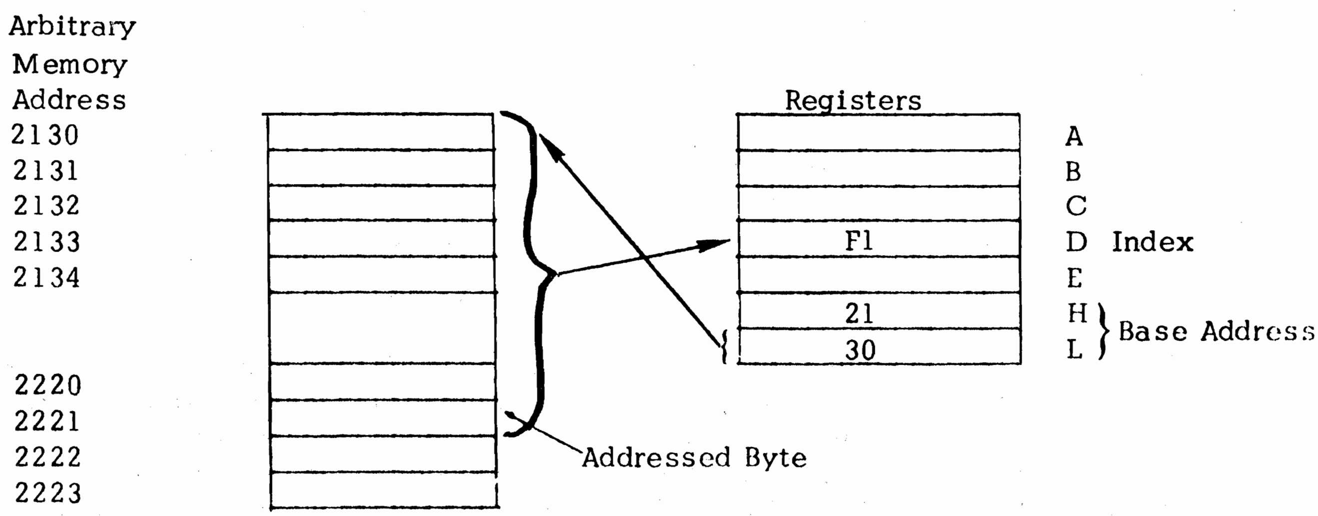


Figure 2-5 Indexed Address, Formed by H and L Register (Base) Plus D Register (Arbitrarily Selected as Index Register)

Indexed addressing can easily be accomplished on the 8008 by writing a sequence of instructions referred to as a Macro. (See Section 4.4).

2.4.3 INDIRECT ADDRESSING

An indirect address specifies where in memory a direct address is to be found. The concept of indirect addressing, as applied to the 8008 is illustrated in Figure 2-6.

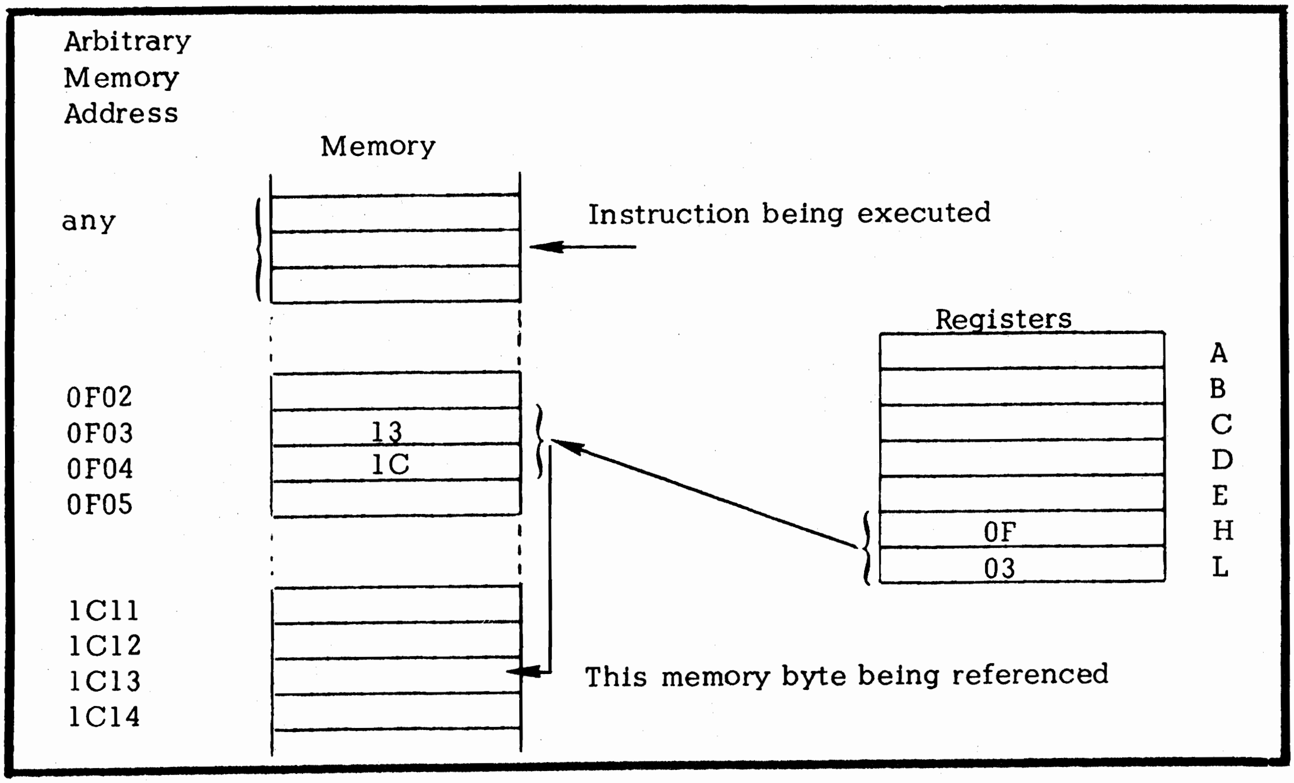


Figure 2-6 Indirect Addressing

In Figure 2-6, the instruction being executed specifies that the address of the memory byte to be referenced is stored in two memory bytes pointed to by the H and L registers. The H and L registers contain the memory address 0F03H; therefore the address of the memory byte to be referenced is to be found in memory bytes 0F03H and 0F04H. These two memory bytes hold the address 1C13H, which becomes the referenced memory location. Note that the address is stored with the least significant 8 bits in the lower addressed memory location (0F03H), while the most significant 6 bits are stored in the higher addressed location (0F04H). This is the usual method for storing addresses in the 8008.

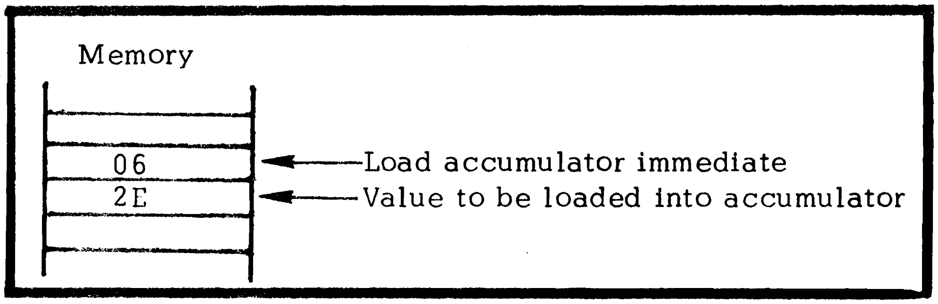
Indirect addressing on the 8008 can also be accomplished by writing a macro as described in Section 4.4.

2.4.4 IMMEDIATE ADDRESSING

An immediate instruction is one that provides its own data. The following is an example of immediate addressing:

Load register A (the accumulator) with the value 2EH.

The above instruction would be coded in memory as follows:

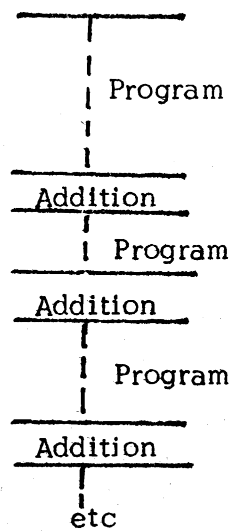


Immediate instructions do not reference memory; rather they store data in the memory byte directly following the instruction code byte.

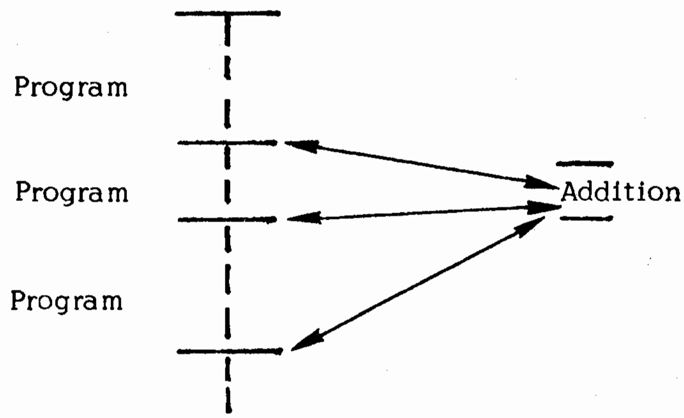
2.4.5 SUBROUTINES AND USE OF THE STACK FOR ADDRESSING

Before understanding the purpose or effectiveness of the stack, it is necessary to understand the concept of a subroutine.

Consider a frequently used operation such as addition. The INTELLEC 8 provides instructions to add one byte of data to another byte of data, but what if you wish to add numbers outside the range of 0 to 255 (the range of one data byte)? Such addition will require a number of instructions to be executed in sequence. It is quite possible that this addition routine may be required many times within one program; to repeat the identical code every time it is needed is possible, but very wasteful of memory:

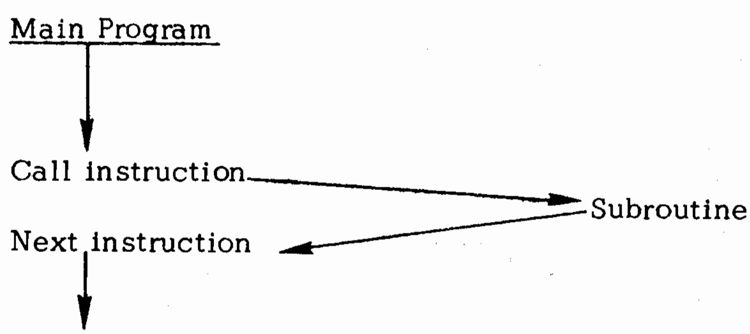


A more efficient means of accessing the addition routine would be to store it once, and find a way of accessing it when needed:



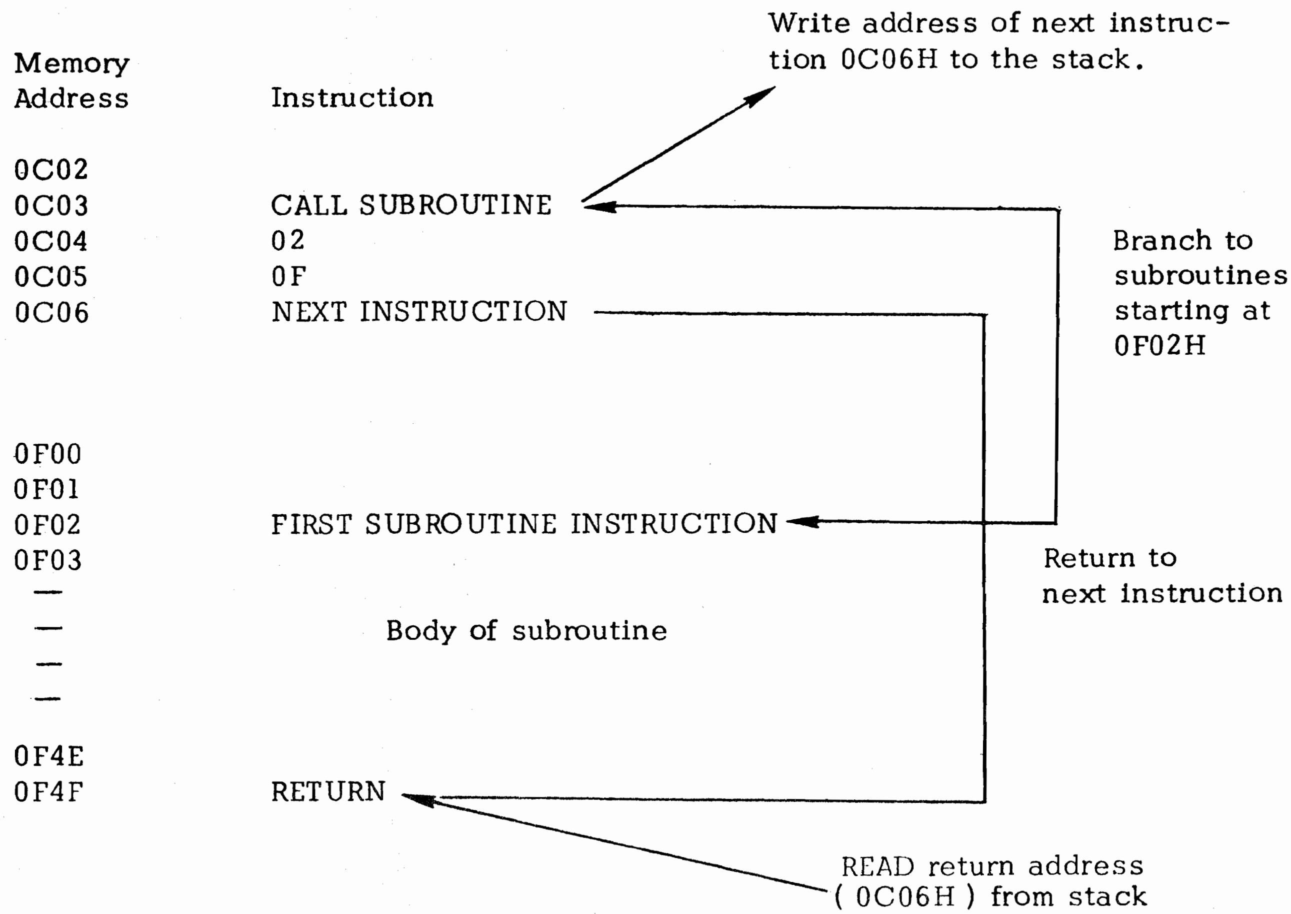
A frequently accessed routine such as the addition above is called a subroutine, and the 8008 provides instructions that call subroutines and return from subroutines.

When a subroutine is executed, the sequence of events may be depicted as follows:



The arrows indicate the execution sequence.

When the "Call" instruction is executed, the address of the "next" instruction is written to the stack. (See Section 2.1.2), and the subroutine is executed. The last executed instruction of a subroutine will always be a special "Return Instruction", which reads an address from the stack into the program counter, and thus causes program execution to continue at the "Next" instruction as illustrated on the next page.


Memory Address
Instruction
0C02
0C03 CALL SUBROUTINE
0C04 02
0C05 0F
0C06 NEXT INSTRUCTION
0F00
0F01
0F02 FIRST SUBROUTINE INSTRUCTION
0F03
Body of subroutine
0F4E
0F4F RETURN
Write address of next instruction 0C06H to the stack.
Branch to subroutines starting at 0F02H
Return to next instruction
READ return address (0C06H) from stack

Since the stack provides seven registers, subroutines may be nested up to seven deep; for example, the addition subroutine could itself call some other subroutine, and so one. An examination of the sequence of write and read stack operations will show that the return path will always be identical to the call path, even if the same subroutine is called at more than one level; however, an attempt to nest subroutines to a depth of more than 7 will cause the program to fail, since some addresses will have been overwritten.

2.5 CONDITION BITS

To make programming easier, four condition (or status) bits are provided by the 8008 to reflect the results of data operations. The descriptions of individual instructions in Section 3 specify which condition bits are affected by the execution of the instruction, and whether the execution of the instruction is dependent in any way on prior status of condition bits.

In the following discussion of condition bits, a bit is "set" to 1, and "reset" to 0.

2.5.1 CARRY BIT

Certain data operations can cause an overflow out of the high order 7 - bit. For example, addition of two numbers, each of which occupies one byte, can give rise to an answer that does not fit in one byte:

	7 6 5 4 3 2 1 0	Bit No.
+ AE	1 0 1 0 1 1 1 0	
74	0 1 1 1 0 1 0 0	
122	0 0 1 0 0 0 1 0	

Carry=1

An operation that results in a carry out of bit 7 will set the carry bit.

An operation that could have resulted in a carry out of bit 7 but did not will reset the carry bit.

NOTE: The 8008 subtract and compare operations (SUB, SBB, SUI, SBI, CMP, CMI) are exceptions to the above rules. See the appropriate sections for details.

2.5.2 SIGN BIT

As described in Section 3.2.1, it is possible to treat a byte of data as having the numerical range -128_{10} to $+127_{10}$. In this case by convention the 7 bit will always represent the sign of the number; that is, if the 7 bit is 1, the number is in the range -128_{10} to -1. If bit 7 is 0, the number is in the range 0 to $+127_{10}$.

At the conclusion of certain instructions (as specified in the instruction description sections of Section 3), the sign bit will be set to the condition of the answer 7 bit in order to reflect the sign of the answer.

2.5.3 ZERO BIT

This condition bit is set if the answer generated by the execution of certain instructions leaves a zero result in a register. The zero bit is reset if the result is not zero.

A result that has an overflow but a zero answer byte, as illustrated below, will alos set the zero bit:

```
                    76543210     Bit Number
                    10100111 +
                    01011001
                  1 00000000
Overflow out ------^
of bit 7.             Zero Answer

                    Zero bit set to 1.
```

2.5.4 PARITY BIT

In order to check that a data transfer operation occurred accurately, byte "parity" is checked. The number of 1 bits in a byte are counted, and if the total is odd, "odd" parity is flagged; if the total is even, "even" parity is flagged.

The parity bit is set to 1 for even parity, and is set to 0 for odd parity.

3.0 THE 8008 INSTRUCTION SET

This section describes the 8008 assembly language instruction set.

For the reader who understands assembly language programming, Appendix "A" provides a complete summary of the 8008 instructions.

For the reader who is not completely familiar with assembly language, Section 3 describes individual instructions with examples and machine code equivalents.

3.1 ASSEMBLY LANGUAGE

3.1.1 HOW ASSEMBLY LANGUAGE IS USED

Upon examining the contents of computer memory, a program would appear as a sequence of hexadecimal digits, which are interpreted by the CPU as instruction codes, addresses, or data. It is possible to write a program as a sequence of digits (just as they appear in memory), but that is slow and expensive. For example, many instructions reference memory to address either a data byte or another instruction:

Memory Address	
1432	C7
1433	44
1434	C4
1435	14
1436	
:	:
14C3	E2
14C4	36
14C5	36
14C6	F8

Assuming the registers H and L contain 14H and C3H respectively, the program operates as follows:

Byte 1432 specifies that the accumulator is to be loaded with the contents of byte 14C3.

Bytes 1433 through 1435 specify that execution is to continue with the instruction starting at byte 14C4.

Bytes 14C4 and 14C5 specify that the L register is to be loaded with the number "36" H.

Byte 14C6 specifies that the contents of the accumulator are to be stored in byte 1436.

Now suppose that an error discovered in the program logic necessitates placing an extra instruction after byte 1432. Program code would have to change as follows:

Memory Address	Old Code	New Code
1432	C7	C7
1433	44	New Instruction
1434	C4	44
1435	14	C5
1436		14
1437	:	
14C3	E2	:
14C4	36	E2
14C5	36	36
14C6	F8	37
14C7		F8

Most instructions have been moved and as a result many must be changed to reflect the new memory addresses of instructions or data. The potential for making mistakes is very high and is aggravated by the complete unreadability of the program.

Writing programs in assembly language is the first and most significant step towards economical programming; it provides a readable notation for instructions, and separates the programmer from a need to know or specify absolute memory addresses.

Assembly language programs are written as a sequence of instructions which are converted to executable hexadecimal code by a special program called an ASSEMBLER. Use of the INTELLEC 8 assembler is described in the INTELLEC 8 Operator's Manual.

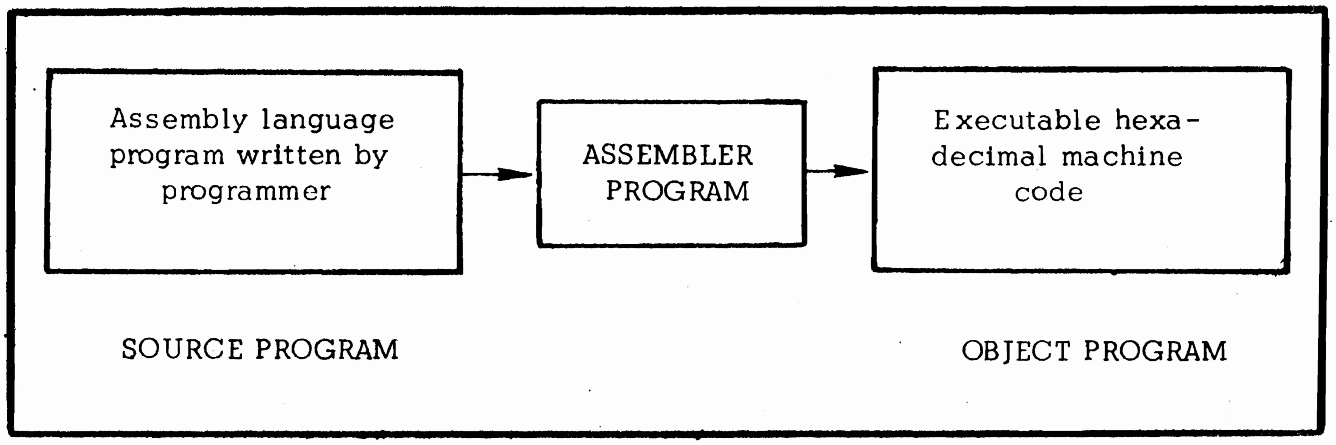


Figure 3-1

Assembler Program Converts Assembly Language Source Program to Hexadecimal Object Program

As illustrated in Figure 3-1, the assembly language program generated by a programmer is called a SOURCE PROGRAM. The assembler converts the SOURCE PROGRAM into an equivalent OBJECT PROGRAM, which consists of a sequence of hexadecimal codes that can be loaded into memory and executed.

For example:

```
         Source Program     is converted by the      One Possible Version of
                            Assembler to             the Object Program

NOW:     MOV    A,B                                      C1
         CPI    'C'                                      3C43
         JZ     LER                                      687C3D
                :                                        :
LER:     MOV    M,A                                      F8
```

Now if a new instruction must be added, only one change is required. Even the reader who is not yet familiar with assembly language will see how simple the addition is:

```
NOW:            MOV          A,B
           (New instruction inserted here)
                CPI          'C'
                JZ           LER
                      :
LER             MOV          M,A
```

The assembler takes care of the fact that a new instruction will shift the rest of the program in memory.

3.1.2 STATEMENT MNEMONICS

Assembly language instructions must adhere to a fixed set of rules as described in this section. An instruction has four separate and distinct parts or FIELDS.

Field 1 is the LABEL field. It is the instruction's label or name, and it is used to reference the instruction.

Field 2 is the CODE field. It defines the operation that is to be performed by the instruction.

Field 3 is the OPERAND field. It provides either any address or data information needed by the CODE field.

Field 4 is the COMMENT field. It is present for the programmer's convenience and is ignored by the assembler. The programmer uses comment fields to describe the operation and thus make the program more readable.

The assembler uses free fields; that is, any number of blanks may separate fields.

Before describing each field in detail, here are some general examples:

Label	Code	Operand	Comment
HERE:	MVI	C,O	; Load the C register with zero
THERE:	DB	3AH	; Create a one-byte data constant
LOOP:	ADD	E	; Add contents of E register
			; to the accumulator
	RLC		; Rotate the accumulator left

3.1.3 LABEL FIELD

This is an optional field, which, if present, may be from 1 to 5 characters long. The first character of the label must be a letter of the alphabet or one of the special characters @ (at sign) or ? (question mark). A colon (:) must follow the last character. (The operation codes, pseudo - instruction names, and register names are specially defined within the assembler and may not be used as labels. Operation codes are given in sections 3.2 - 3.13 and Appendix A; pseudo - instructions are described in section 3.14.)

Here are some examples of valid label fields:

LABEL:

F14F:

@HERE:

?ZERO:

Here are some invalid labels:

123:	begins with a decimal digit
LABEL	does not end with a colon
ADD:	is an operation code
END:	is a pseudo - instruction

The following label has more than five characters; only the first five will be recognized:

INSTRUCTION: will be read as INSTR:

Since labels serve as instruction addresses, they cannot be duplicated. For example, the sequence:

```
HERE:    JMP    THERE
         ---
         ---
THERE:   MOV    C,D
         ---
         ---
THERE:   CALL   SUB
```

is ambiguous; the assembler cannot determine which THERE: address is to be referenced by the JMP instruction.

One instruction may have more than one label, however. The following sequence is valid:

```
LOOP1:                                  ; First label
LOOP2:      MOV         C,D             ; Second label
            ---
            JMP         LOOP1
            ---
            JMP         LOOP2
```

Each JMP instruction will cause program control to be transferred to the same MOV instruction.

3.1.4 CODE FIELD

This field contains a code which identifies the machine operation (add, subtract, jump, etc.) to be performed: hence the term operation code or op-code. The instructions described in sections 3.2 - 3.13 are each identified by a mnemonic label which must appear in the code field. For example, since the "jump" instruction is identified by the letters "JMP", these letters must appear in the code field to identify the instruction as "jump".

There must be at least one space following the code field. Thus:

```
HERE:       JMP         THERE
```

is legal, but:

```
HERE:       JMPTHERE
```

is illegal.

3.1.5 OPERAND FIELD

This field contains information used in conjunction with the code field to define precisely the operation to be performed by the instruction. Depending upon the code field, the operand field may be absent or may consist of one item or two items separated by a comma.

There are four types of information [(a) through (d) below] that may be requested as items of an operand field, and the information may be specified in nine ways [(1) through (9) below].

The nine ways of specifying information are as follows:

(1) Hexadecimal data. Each hexadecimal number must be followed by the letter 'H' and must begin with a numeric digit (0 - 9).

Example:

Label	Code	Operand	Comment
HERE:	MVI	C, 0BAH	; Load register C with the
			; hexadecimal number BA

(2) Decimal data. Each decimal number may optionally be followed by the letter 'D', or may stand alone.

Example:

Label	Code	Operand	Comment
ABC:	MVI	E, 105	; Load register E with 105

(3) Octal data. Each octal number must be followed by one of the letters 'O' or 'Q'.

Example:

Label	Code	Operand	Comment
LABEL:	MVI	A, 72O	; Load the accumulator with
			; the octal number 72

(4) Binary data. Each binary number must be followed by the letter 'B'.

Example:

Label	Code	Operand	Comment
NOW:	MVI	10B, 11110110B	; Load register two (the C
			; register) with 0F6H
JUMP:	JMP	0010111011111010B	; Jump to memory address
			; 2EFA

(5) The current program counter. This is specified as the character '$' and is equal to the address of the current instruction.

Example:

Label	Code	Operand
GO:	JMP	$ + 6

The instruction above causes program control to be transferred to the address 6 bytes beyond where the JMP instruction is loaded.

(6) An ASCII constant. This is one or more ASCII characters enclosed in single quotes. Two successive single quotes must be used to represent one single quote within an ASCII constant. Appendix D contains a list of legal ASCII characters and their hexadecimal representations.

Example:

Label	Code	Operand	Comment
CHAR:	MVI	E, '*'	; Load the E register with the
			; eight bit ASCII representation
			; of an asterisk

(7) Labels that have been assigned a numeric value by the assembler. (See section 3.16.2 for the equate procedure). The following equates are built into the assembler and are therefore always active:

A	equated	to	0	representing	the accumulator	
B	"	"	1	"	register	B
C	"	"	2	"	"	C
D	"	"	3	"	"	D
E	"	"	4	"	"	E
H	"	"	5	"	"	H
L	"	"	6	"	"	L
M	"	"	7	"	a memory reference	

Example:

Suppose VALUE has been equated to the hexadecimal number 9FH. Then the following instructions all load the D register with 9FH:

Label	Code	Operand
Al:	MVI	D, VALUE
A2:	MVI	3, 9FH
A3:	MVI	3, VALUE

(8) Labels that appear in the label field of another instruction.

Example:

Label	Code	Operand	Comment
HERE:	JMP	THERE	; Jump to instruction at THERE:
	- - -		
	- - -		
THERE:	MVI	D,9FH	

(9) Arithmetic and logical expressions involving data types (1) to (8) above connected by the arithmetic operators + (addition), - (unary minus and subtraction), * (multiplication), / (division), MOD (modulo), the logical operators NOT, AND, OR, XOR, SHR, (shift right), SHL (shift left), and left and right parentheses.

All operators treat their arguments as 16 bit quantities, and generate 16 bit quantities as their result. The programmer must insure that the result generated fits the requirements of the operation being coded. For example, the second operand of an MVI instruction must be an 8 bit value.

Therefore the instruction:

```
MVI     H,NOT   0
```

is invalid, since NOT 0 produces the 16 bit hexadecimal number FFFF. However, the instruction:

```
MVI     H,NOT   0   AND   0FFH
```

is valid, since the most significant 8 bits of the result are insured to be 0, and the result can therefore be represented in 8 bits.

The SHR and SHL operators are linear shifts which cause zeroes to be shifted into the high order and low order bits, respectively, of their arguments.

NOTE: An instruction in parenthesis is a legal expression in an optional field. Its value is the encoding of the instruction.

Examples:

Label	Code	Operand	Arbitrary Memory Address
HERE:	MVI	H, HERE SHR 8	2E1A

The above instruction loads the hexadecimal number 2EH (14-bit address of HERE shifted right 8 bits) into the H register.

Label	Code	Operand
NEXT:	MVI	D, 34 + 40H/2

The above instruction will load the value 34 + (64/2) = 34 + 32 = 66 into the D register.

Label	Code	Operand
INS:	DB	(ADD C)

The above instruction defines a byte of value 82H (the encoding of ADD C instruction) at location INS:.

Label	Code	Operand
NEXT:	MVI	D, 34 + 40H/2

The above instruction will load the value 34 + (64/2) = 34 + 32 = 66 into the D register.

Operators within an expression are evaluated in the following order:

1. Left and right parentheses
2. *, /, MOD, SHL, SHR
3. +, - (unary and binary)
4. NOT
5. AND
6. OR XOR

Thus the instruction:

```
        MVI     D, (34 + 40H ) /2
```

will load the value

(34 + 64) /2 = 49 into the D register.

The operators MOD, SHL, SHR, NOT, AND, OR, and XOR must be separated from their operands by at least one blank. Thus the instruction:

```
        MVI     C, VALUE AND0FH
```

is invalid.

Using some or all of the above nine data specifications, the following four types of information may be requested:

(a) A register (or code indicating memory reference) to serve as the source or destination in a data operation - Methods 1, 2, 3, 4, 7, or 9 may be used to specify the register or memory reference, but the specification must finally evaluate to one of the numbers 0 - 7 as follows:

Value	Register
0	A (accumulator)
1	B
2	C
3	D
4	E
5	H
6	L
7	Memory Reference

Example:

Label	Code	Operand
INS1:	MVI	REG4, 2EH
INS2:	MVI	4H, 2EH
INS3:	MVI	8/2, 2EH

Assuming REG4 has been equated to 4, all the above instructions will load the value 2EH into register 4.

(b) Immediate data, to be used directly as a data item.

Example:

Label	Code	Operand	Comment
HERE:	MVI	H, DATA	; Load the H register with ; the value of DATA

Here are some examples of the form DATA could take:

ADDR AND 0FFH (where ADDR is a 14-bit address)
127
'*'
VALUE (where VALUE has been equated to a number)
3EH + 10 / (2 AND 2)

(c) A 14-bit address, or the label of another instruction in memory.

Example:

Label	Code	Operand	Comment
HERE:	JMP	THERE	; Jump to the instruction at THERE
	JMP	2EADH	; Jump to address 2EAD

(d) A number in a specific range, required by certain instructions.

The RST and IN instructions require a number in the range 0 - 7.

Example:

Label	Code	Operand	Comment
GOOD:	RST	111B	; Value of 7, valid
OK:	IN	15 - 0AH	; Value of 5, valid
BAD:	RST	10	; Value of 10, invalid instruction

The OUT instruction requires a number in the range 8 - 31 decimal.

Example:

Label	Code	Operand	Comment
GOOD:	OUT	20 + 11	; Value 31 decimal, valid
BAD:	OUT	20 + 11 H	; Value 37 decimal, invalid

The INR and DCR instructions require a number in the range 1 - 6, specifying one of registers B, C, D, E, H, or L.

Example:

Label	Code	Operand	Comment
OK:	INR	4	; Increment register E
	INR	110B	; Decrement register L

3.1.6 COMMENT FIELD

The only rule governing this field is that it must being with a semi colon (;).

```
HERE:   MVI   C, 0ADH   ; This is a comment
```

A comment field may appear alone on a line:

```
;
; Begin loop here
;
```

3.2 DATA STATEMENTS

This section describes ways in which data can be specified in and interpreted by a program. Any 8 bit byte contains one of the 256 possible combinations of zeros and ones. Any particular combination may be interpreted in various ways. As previously mentioned, the code 1AH may be interpreted as a machine instruction (Rotate Accumulator Right through Carry), as a hexidecimal value 1AH = 26D, or merely as the bit pattern 00011010

Arithmetic instructions assume that the data bytes upon which they operate are in a special format called "two's complement", and the operations performed on these bytes are called "two's complement arithmetic."

3.2.1 TWO'S COMPLEMENT

When a byte is interpreted as a signed two's complement number, the low order 7 bits supply the magnitude of the number, while the high order bit is interpreted as the sign of the number (0 for positive numbers, 1 for negative).

The range of positive numbers that can be represented in signed two's complement notation is, therefore, from 0 to 127:

0	=	0 0 0 0 0 0 0 0 B	=	0 H
1	=	0 0 0 0 0 0 0 1 B	=	1 H
	⋮			
126D	=	0 1 1 1 1 1 1 0 B	=	7 E H
127D	=	0 1 1 1 1 1 1 1 B	=	7 F H

To change the sign of a number represented in two's complement, the following rules are applied:

(a) Invert each bit of the number (producing the so-called one's complement).

(b) Add one to the result, ignoring any carry out of the high order bit position.

Example: Produce the two's complement representation of - 10D. Following the rules above,

+ 10D = 0 0 0 0 1 0 1 0

Invert each bit : 1 1 1 1 0 1 0 1
Add one : 1 1 1 1 0 1 1 0

Therefore, the two's complement representation of - 10D is F6H. (Note that the sign bit is set, indicating a negative number.)

Example: What is the value of 86H interpreted as a signed two's complement number? The high order bit is set, indicating that this is a negative number. To obtain its value, again invert each bit and add one.

86H = 1 0 0 0 0 1 1 0 B

Invert each bit	:	0 1 1 1 1 0 0 1	B
Add one	:	0 1 1 1 1 0 1 0	B

Thus, the value of 86 H is - 7A H = - 122 D

The range of negative numbers that can be represented in signed two's complement notation is from -1 to -128.

-1	=	1 1 1 1 1 1 1 1 B	=	FFH
-2	=	1 1 1 1 1 1 1 0 B	=	FEH
⋮				
-127D	=	1 0 0 0 0 0 0 1 B	=	81H
-128D	=	1 0 0 0 0 0 0 0 B	=	80H

To perform the subtraction 1AH - 0CH, the following operations are performed:

Take the two's complement of 0CH = F4H

Add the result to the minuend:

```
              1AH = 0 0 0 1 1 0 1 0
+ (-0CH) =    F4H = 1 1 1 1 0 1 0 0
              ---------------------
                    0 0 0 0 1 1 1 0  = 0EH  the correct answer
```

When a byte is interpreted as an unsigned two's complement number, its value is considered positive and in the range 0 to 255_{10} :

0	=	0 0 0 0 0 0 0 0	B	=	0H
1	=	0 0 0 0 0 0 0 1	B	=	1 H
⋮					
127D	=	0 1 1 1 1 1 1 1	B	=	7FH
128D	=	1 0 0 0 0 0 0 0	B	=	80H
⋮					
255D	=	1 1 1 1 1 1 1 1	B	=	FFH

Two's complement arithmetic is still valid. When performing an addition operation, the carry bit is set when the result is greater than 255D. When performing subtraction, the carry bit is reset when the result is positive. If the carry bit is set, the result is negative and present in its two's complement form.

Example: Subtract 98D from 197D using unsigned two's complement arithmetic.

```
             197D =  1 1 0 0 0 1 0 1  =  C5H
             -98D =  1 0 0 1 1 1 1 0  =  9EH
Overflow--------->1  0 1 1 0 0 0 1 1  =  63H  =  99D
```

Since the overflow out of bit 7=1, indicating that the answer is correct and positive, the subtract operation will reset the carry bit.

Example: Subtract 15D from 12 D using unsigned two's complement arithmetic.

```
             12D =  0 0 0 0 1 1 0 0  =  0CH
            -15D =  1 1 1 1 0 0 0 1  =  0F1H
Overflow-------->0  1 1 1 1 1 1 0 1  =  -3D
```

Since the overflow out of bit 7=0, indicating that the answer is negative and in its two's complement form, the subtract operation will set the carry bit. (This also indicates that a "borrow" occurred while subtracting multibyte numbers. See Section 5.3).

NOTE: The 8008 instructions which perform the subtraction operation are SUB, SUI, SBB, SBI, CMP, and CMI. Although the same result will be obtained by addition of a complemented number or subtraction of an uncomplemented number, the resulting carry bit will be different.

Example: If the result -3 is produced by performing an "ADD" operation on the numbers +12D and -15D, the carry bit will be reset; if the same result is produced by performing a "SUB" operation on the numbers +12D and +15D, the carry bit will be set. Both operations indicate that the result is negative; the programmer must be aware which operations set or reset the carry bit.

```
"ADD"  +12D and -15D                          "SUB"  +15D from +12D

   +12D  =  0 0 0 0 1 1 0 0                      +12D  =  0 0 0 0 1 1 0 0
+(-15D)  =  1 1 1 1 0 0 0 1                   -(+15D)  =  1 1 1 1 0 0 0 1
      0]    1 1 1 1 1 1 0 1  =  -3D                 0]    1 1 1 1 1 1 0 1  =  -3D
      └──►causes carry to be reset                  └──►causes carry to be set
```

<u>WHY TWO'S COMPLEMENT ?</u>

Using two's complement notation for negative numbers, any subtraction problem becomes a sequence of bit inversions and additions. Therefore, fewer circuits need to be built to perform subtraction.

3.2.2 DB DEFINE BYTE (S) OF DATA

Format:

Label	Code	Operand
HERE:	DB	**LIST**

LIST is a list of either:

1) Arithmetic and logical expressions involving any of the arithmetic and logical operators, which evaluate to eight-bit data quantities

2) Strings of ASCII characters enclosed in quotes

Description: The eight bit value of each expression, or the eight bit ASCII representation of each character is stored in the next available byte of memory starting with the byte addressed by HERE:

Examples:

Instruction			Assembled Data (hex)
HERE:	DB	0A3H	A3
WORD 1 :	DB	5 * 2, 2FH - OAH	0A25
WORD 2 :	DB	5ABCH SHR 8	5A
STR:	DB	'STRINGSp1'	535452494E472031
MINUS:	DB	- 03H	FD

Note: In the first example above, the hexadecimal value A3 must be written as 0A3 since hexadecimal numbers must start with a decimal digit. (See Section 3.1.5).

3.2.3 DW DEFINE WORD (TWO BYTES) OF DATA

Format:

Label	Code	Operand
HERE:	DW	LIST

LIST is a list of expressions which evaluate to 16 bit data quantities.

Description: The least significant 8 bits of the expression are stored in the lower address memory byte (HERE:), and the most significant 8 bits are stored in the next higher addressed byte (HERE:+1). This reverse order of the high and low address bytes is normally the case when storing addresses in memory. This statement is usually used to create address constants for the transfer-of-control instructions; thus LIST is usually a list of one or more statement labels appearing elsewhere in the program.

Examples:

Assume COMP addresses memory location 3B1CH and FILL addresses memory location 3EB4.

Instruction			Assembled Data (hex)
ADD1:	DW	COMP	1C3B
ADD2:	DW	FILL	B43E
ADD3:	DW	3C01H, 3CAEH	013CAE3C

Note that in each case, the data are stored with the least significant 8 bits first.

3.2.4 DS DEFINE STORAGE (BYTES)

Format:

Label	Code	Operand
HERE:	DS	EXP

EXP is a single arithmetic or logical expression.

Description: The value of EXP specifies the number of memory bytes to be reserved for data storage. No data values are assembled into these bytes: in particular the programmer should not assume that they will be zero, or any other value. The next instruction will be assembled at memory location HERE: + EXP (HERE: + 10 or HERE: + 16 in the example below).

Examples:

```
HERE:   DS    10     ;   Reserve the next 10 bytes
        DS    10H    ;   Reserve the next 16 bytes
```

3.3 SINGLE REGISTER INSTRUCTIONS

This section describes the two instructions which involve a single register. Instructions in this class occupy one byte as follows:

0	0	R	E	G	0	0	X

X: 0 for INR, 1 for DCR

REG:

001	for register	B
010	for register	C
011	for register	D
100	for register	E
101	for register	H
110	for register	L

Note: REG $\neq$ 000 or 111

The general assembly language instruction format is:

Label	Code	Operand
LABEL:	OP	REG

REG: B,C,D,E,H,L

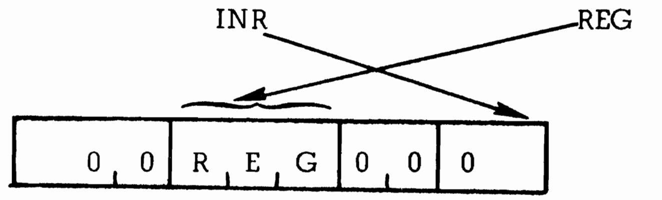
OP: INR or DCR

LABEL: Optional instruction label

3.3.1 INR INCREMENT REGISTER

Format:

Label	Code	Operand



Description: The register specified by REG is incremented by one. REG cannot evaluate to 000 or 111, implying that neither the accumulator nor any memory location can be incremented by this instruction.

Condition bits affected: Zero, sign, parity

Example: If register C contains 99H, the instruction:

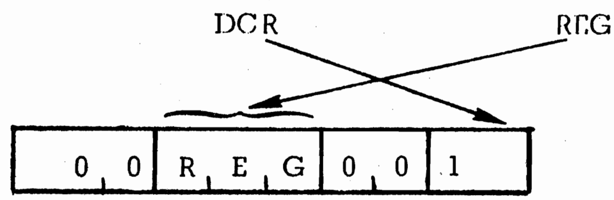
INR C

will cause register C to contain 9AH.

3.3.2 DCR DECREMENT REGISTER

Format:

Label	Code	Operand



Description: The register specified by REGis decremented by one. REG cannot evaluate to 000 or 111, implying that neither the accumulator nor any memory location can be decremented by this instruction.

Condition bits affected: Zero, sign, parity

Example: If register L contains zero, the instruction:

DCR L

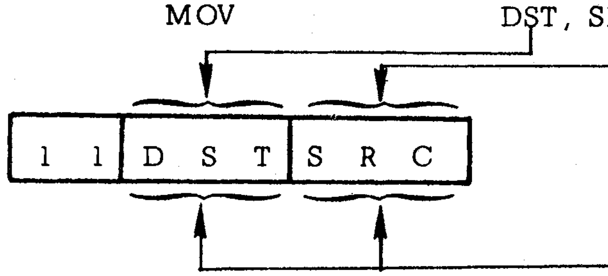
will cause register L to contain 0FFH
(minus one in two's complement form)

3.4 MOV INSTRUCTION

This section describes the MOV instruction, which transfers data between registers or between memory and registers. This instruction occupies one byte.

Format:

Label	Code	Operand
	MOV	DST, SRC



000 for register A
001 for register B
010 for register C
011 for register D
100 for register E
101 for register H
110 for register L
111 for memory reference

Note: DDD and SSS cannot both = 111 B.

Description: One byte of data is moved from the register specified by SRC (the source register) to the register specified by DST (the destination register). The data replaces the contents of the destination register; the source register remains unchanged. If a memory reference is specified (SRC or DST = 111B), the data is fetched from or stored into the memory address contained in the H and L registers. Register L contains the low-order eight bits of the address and register H contains the high-order six bits of the address.

Condition bits affected: none

Example 1:

Label	Code	Operand	Comment
	MOV	A,E	; Move contents of the E register
			; to the A register
	MOV	D, D	; Move contents of the D register
			; to the D register, i.e., this is
			; a null operation

Note: Any of the null operation instructions MOV X, X can also be specified as N (no-operation).

Example 2:

The following set of instruction will store the contents of the accumulator at memory location 2BE9H.

Label	Code	Operand	Comment
START:	MVI	H, 2BH	; H = high order address byte
	MVI	L, 0E9H	; L = low order address byte
	MOV	M,A	; Move accumulator to memory

Example 3:

The following set of instructions will store the D register at memory location FINAL:, wherever that location happens to be in memory.

Label	Code	Operand	Comment
START:	MVI	H, FINAL SHR 8	; H = high order byte
	MVI	L, FINAL AND 0FFH	; L = low order byte
	MOV	M,D	; Move D to memory
			; addressed by H and L

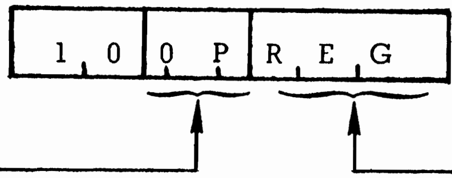
The first two instructions in the example above are so commonly used that they may be specified by the single macro instruction:

```
START:     LXI     H,FINAL
```

as described in Section 4.2.1.

3.5 REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS

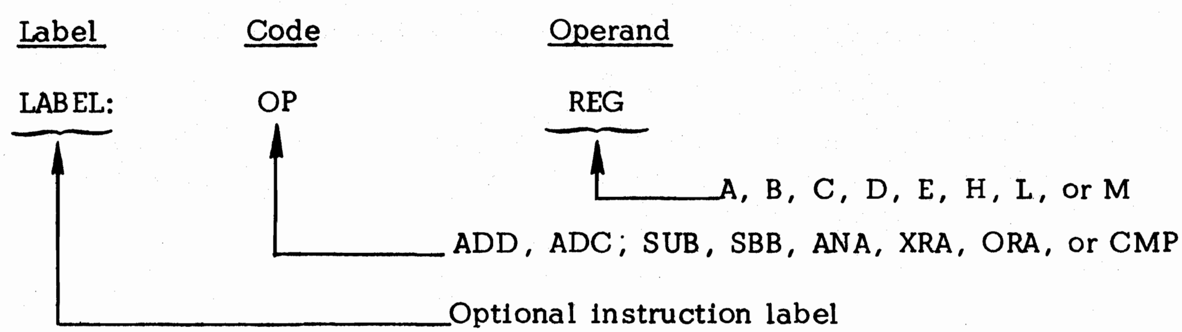
This section describes the instructions which operate on the accumulator using a byte fetched from another register or memory. Instructions in this class occupy one byte as follows:



000 for ADD	000 for register A
001 for ADC	001 for register B
010 for SUB	010 for register C
011 for SBB	011 for register D
100 for ANA	100 for register E
101 for XRA	101 for register H
110 for ORA	110 for register L
111 for CMP	111 for memory reference

When a memory reference is specified, the byte of data is fetched from the memory location addressed by registers H and L.

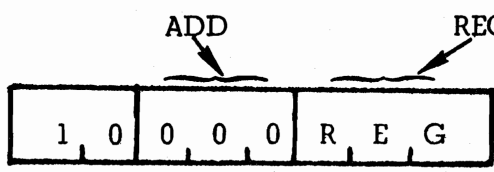
The general assembly language instruction format is:



3.5.1 ADD ADD REGISTER OR MEMORY TO ACCUMULATOR

Format:

Label	Code	Operand



Description: The byte in the register specified by REG, or the memory location addressed by H and L (if REG=111B), is added to the contents of the accumulator using two's complement arithmetic. The result is kept in the accumulator; the byte in REG is unchanged.

If there is a carry out of the high-order bit position, the carry bit is set.

The zero bit is set if the result is zero.

The parity bit is set if the result contains an even number of ones (even parity).

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example 1:

Label	Code	Operand	Assembled Data
AD1:	MVI	D, 2FH	1E2F
AD2:	MVI	A, 6CH	066C
	ADD	D	83

The instructions at AD1: and AD2: load the D register with 2FH and the accumulator with 6CH, respectively. The ADD instruction performs the addition as follows:

```
2EH  =  0 0 1 0 1 1 1 0
6CH  =  0 1 1 0 1 1 0 0
------------------------
9AH  =  1 0 0 1 1 0 1 0
```

The zero and carry bits are reset; the parity and sign bits are set. The accumulator now contains 9AH.

Example 2:

The instruction:

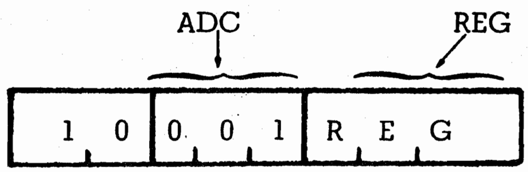
ADD A

will double the accumulator.

3.5.2 ADC ADD REGISTER OR MEMORY TO ACCUMULATOR WITH CARRY

Format:

Label	Code	Operand
	ADC	REG



Description: The byte in the register specified by REG, or the memory location, addressed by H and L (if REG = 111B) plus the contents of the carry bit is added to the contents of the accumulator. The result is kept in the accumulator; the byte in REG is unchanged.

The carry bit is set if there is a carry out of the high-order bit position.

The zero bit is set if the result is zero.

The parity bit is set if the result has even parity.

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example:

Label	Code	Operand	Assembled Data
AD1:	MVI	C, 3DH	163D
AD2:	MVI	A, 42H	0642
	ADC	C	8A

Assume that the carry bit = 0. The instructions at AD1 : and AD2 : load the C register and the accumulator with 3D and 42 respectively, but do not affect the condition bits. The ADC instruction performs the addition as follows:

```
    3DH  =  0 0 1 1 1 1 0 1
    42 H =  0 1 0 0 0 0 1 0
  CARRY  =                0
            ---------------
  RESULT =  0 1 1 1 1 1 1 1  =  7FH
```

The results can be summarized as follows:

Accumulator	=	7FH
Carry	=	0
Sign	=	0
Zero	=	0
Parity	=	0

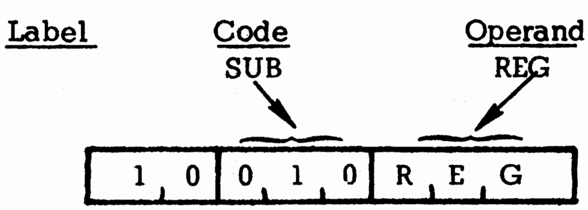
If the carry bit had been one at the beginning of the example, the following would have occurred:

```
    3DH  =  0 0 1 1 1 1 0 1
    42H  =  0 1 0 0 0 0 1 0
  CARRY  =                1
            ---------------
 RESULT  =  1 0 0 0 0 0 0 0  = 80H
```

Accumulator	=	80H
Carry	=	0
Sign	=	1
Zero	=	0
Parity	=	0

3.5.3 SUB SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR

Format:



Description: The byte in the register specified by REG, or the memory location addressed by H and L (if REG=111B), is subtracted from the accumulator using two's complement arithmetic. The result is kept in the accumulator; the byte in REG is unchanged.

If there is no overflow out of the high-order bit position, indicating that a borrow occurred, the carry bit is set. (Note that this differs from an add operation, which sets the carry if an overflow occurs.)

The zero bit is set if the result is zero.

The parity bit is set if the result has even parity.

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example:

Assume that the accumulator contains 3EH. Then the instruction:

SUB A

will subtract the accumulator from itself producing a result of zero as follows:

```
        3EH  =  0 0 1 1 1 1 1 0
    + (-3EH) =  1 1 0 0 0 0 0 1  Negate and add one
    +_______________________1  To produce two's complement
Overflow---->1] 0 0 0 0 0 0 0 0  Result = 0
```

Since there was an overflow out of the high-order bit position, and this is a subtraction operation, the carry bit will be reset.

The parity and zero bits will also be set, and the sign bit will be reset.

Thus the SUB A instruction can be used to reset the carry bit (and zero the accumulator).

3.5.4 SBB SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW

Format:

Label	Code	Operand
	SBB	REG

Description: The carry bit is internally added to the contents of the register specified by REG, or the memory location addressed by H and L (if REG=111B). This value is then subtracted from the accumulator using two's complement arithmetic. The result is stored in the accumulator; the byte in REG remains unchanged.

This instruction is most useful when performing multibyte subtractions. It adjusts the result of subtracting two bytes when a previous subtraction has produced a negative result (a borrow). For an example of this, see Section 5.3.

Condition bits affected: Carry, sign, zero, parity (See Section 3.5.3 for details)

Example:

Label	Code	Operand	Assembled Data
SB1	MVI	L, 2 H	3602
SB2	MVI	A, 4H	0604
	SBB	L	

Assume that the carry bit = 1. Then the SBB instruction will act as follows:

02H + Carry = 03H
Two's Complement of 03H = 11111101

Adding this to the accumulator produces:

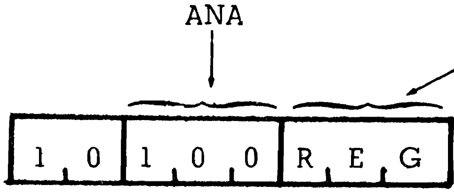
```
Accumulator = 04H  =  0 0 0 0 0 1 0 0
                      1 1 1 1 1 1 0 1
                    1] 0 0 0 0 0 0 0 1  =  01H  = Result
                    |
                    └─> overflow  =  1  causing carry to be reset.
```

The final result stored in the accumulator is one, causing the zero bit to be reset. The carry bit is reset since this is a subtract operation and there was an overflow out of the high-order bit position. The parity and the sign bits are reset.

3.5.5 ANA LOGICAL "AND" REGISTER OR MEMORY WITH ACCUMULATOR

Format:

Label	Code	Operand
	ANA	REG



Description: The byte in the register specified by REG, or the memory location addressed by H and L (if REG = 111B), is logically ANDed bit by bit with the contents of the accumulator.

The result is stored in the accumulator; the byte in REG remains unchanged. The carry bit is set to zero, while the zero, sign and parity bits are set according to the result.

The logical AND function is given by the following truth table:

	0	1
0	0	0
1	0	1

Logical AND

Condition bits affected: Carry, zero, sign, parity

Example:

Since any bit ANDed with a zero produces a zero and any bit ANDed with a one remains unchanged, the AND function is often used to zero groups of bits.

Label	Code	Operand	Assembled Data
AN1 :	MVI	A, 0FCH	06FC
AN2 :	MVI	C, 0FH	160F
	ANA	C	A2

The ANA instruction acts as follows:

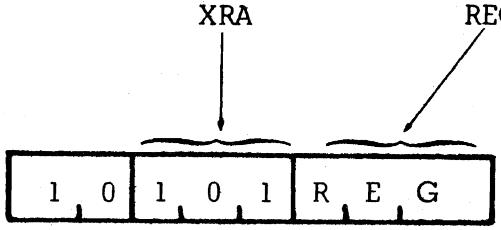
Accumulator	=	1 1 1 1 1 1 0 0	=	0FCH
C Register	=	0 0 0 0 1 1 1 1	=	0FH
Result in Accumulator	=	0 0 0 0 1 1 0 0	=	0CH

This particular example guarantees that the high-order four bits of the accumulator are zero, and the low-order four bits are unchanged.

3.5.6 XRA EXCLUSIVE - OR REGISTER OR MEMORY WITH ACCUMULATOR (ZERO ACCUMULATOR)

Format:

Label	Code	Operand
	XRA	REG



Description: The byte in the register specified by REG, or the memory location addressed by H and L (if REG = 111B), is exclusive - ORed bit by bit with the contents of the accumulator. The result is stored in the accumulator; the byte in REG remains unchanged. The carry bit is set to zero, sign and parity bits are set according to the result.

The Exclusive - OR function is given by the following truth table:

	0	1
0	0	1
1	1	0

Condition bits affected: Carry, zero, sign, parity

Example 1:

Since any bit exclusive - ORed with itself produces zero, the exclusive - OR can be used to quickly zero the accumulator. (The instruction SUB A could also be used.)

Label	Code	Operand
	XRA	A
	MOV	B,A
	MOV	C,A

These instructions quickly zero the A, B, and C register.

Example 2:

The exclusive - OR can be used to test two data bytes for equality.

Label	Code	Operand
	XRA	C

If the contents of the C register and the accumulator are equal, the result will be zero and the zero bit will be set. If the two quantities differ in any bit position a one bit will be produced in the result, and the zero bit will not be set.

Example 3:

Label	Code	Operand
	MVI	A, 0FFH
	XRA	C

Any bit Exclusive - ORed with a one is complemented (0 XOR1 = 1, 1 XOR1 = 0). The XRA instruction above will therefore store the one's complement of the C register into the accumulator.

Example 4:

Testing for change of status.

Many times a byte is used to hold the status of several (up to eight) conditions within a program; each bit signifying whether a condition is true or false, enabled or disabled, etc.

The exclusive - OR function provides a quick means of determing which bits of a word have changed from one time to another.

Label	Code	Operand	
	MVI	H, STAT@ SHR 8	; Load address of status
	MVI	L, STAT@ AND 0FFH	; into H and L registers
LA:	MOV	A,M	; STAT2 to accumulator
	INR	L	; Address next location
LB:	MOV	B,M	; STAT1 to B register
CHNG:	XRA	B	; Exclusive-OR STAT1 and STAT2
STAT:	ANA	B	; AND result with STAT1
STAT2:	DS	1	
STAT1:	DS	1	

Assume that logic elsewhere in the program has read the status of eight conditions and stored the corresponding string of eight zeros and ones at STAT1 and at some later time has read the same conditions and stored the new status at STAT2. The Exclusive - OR at CHNG: produces a one bit in the accumulator wherever a condition has changed between STAT1 and STAT2.

For example:

Bit Number		7 6 5 4 3 2 1 0
STAT1 = 5CH	=	0 1 0 1 1 1 0 0
STAT2 = 78H	=	0 1 1 1 1 0 0 0
Exclusive-OR	=	0 0 1 0 0 1 0 0

This shows that the conditions associated with bits 2 and 5 have changed between STAT1 and STAT2. Knowing this, the program can tell whether these bits were set or reset by ANDing the result with STAT1.

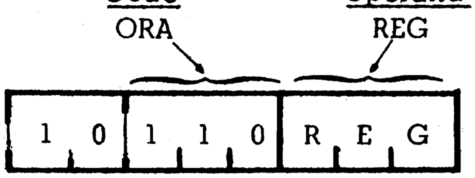
Result	=	0 0 1 0 0 1 0 0
STAT 1	=	0 1 0 1 1 1 0 0
AND	=	0 0 0 0 0 1 0 0

Since bit 2 is now one, it was set between STAT1 and STAT2 ; since bit 5 is zero it was reset.

3.5.7 ORA LOGICAL "OR" REGISTER OR MEMORY WITH ACCUMULATOR

Format:

Label	Code	Operand
	ORA	REG



Description: The byte in the register specified by REG, or the memory location addressed by H and L (if REG = 111B), is logically ORed bit by bit with the contents of the accumulator.

The result is stored in the accumulator; the byte in REG remains unchanged. The carry bit is set to zero, while the zero, sign, and parity bits are set according to the result.

The logical OR function is given by the following truth table:

	0	1
0	0	1
1	1	1

Condition bits affected: Carry, zero, sign, parity

Example:

Since any bit ORed with a one produces a one, and any bit ORed with a zero remains unchanged, the OR function is often used to set groups of bits to one.

Label	Code	Operand	Assembled Data
OR 1 :	MVI	A, 33H	0633
OR 2 :	MVI	C, 0FH	160F
	ORA	C	B2

The ORA instruction acts as follows:

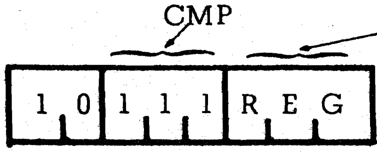
	Accumulator	=	0 0 1 1 0 0 1 1	=	33H
	C register	=	0 0 0 0 1 1 1 1	=	0FH
Result=	Accumulator	=	0 0 1 1 1 1 1 1	=	3FH

This particular example quarantees that the low-order four bits of the accumulator are one, and the high-order four bits are unchanged.

3.5.8 CMP COMPARE REGISTER OR MEMORY WITH ACCUMULATOR

Format:

Label	Code	Operand
	CMP	REG



Description: The byte in the register specified by REG, or the memory location addressed by H and L (if REG = 111B), is compared to the contents of the accumulator. The comparison is performed by internally subtracting the contents of REG from the accumulator (leaving both unchanged) and setting the condition bits according to the result. In particular, the zero bit is set if the quantities are equal, and reset if they are unequal. Since a subtract operation is performed, the carry bit will be set if there is no overflow out of bit 7, indicating that the contents of REG are greater than the contents of the accumulator, and reset otherwise.
Note: If the two quantities to be compared differ in sign, the sense of the carry bit is reversed.

Condition bits affected: Carry, zero, sign, parity

Example 1:

Label	Code	Operand	Assembled Data
	MVI	E 5	2605
	CMP	E	BC

Assume that the accumulator contains the number 0AH. The compare instruction performs the following internal subtractions:

```
      Accumulator     =  0AH  =  0 0 0 0 1 0 1 0
+     ( - E register ) = -5H  =  1 1 1 1 1 0 1 1
                                 ---------------
                              1  0 0 0 0 0 1 0 1  = result
                              └─►overflow =1, causing carry to be reset
```

The accumulator still contains 0AH and the E register still contains 05H; however the carry bit is reset and the zero bit reset, indicating E less than A.

Example 2:

If the accumulator had contained the number 2H, the internal subtraction would have produced the following:

```
        Accumulator      =    02H  = 0 0 0 0 0 0 1 0
  +     ( - E register ) =    -5H  = 1 1 1 1 1 0 1 1
                                     ---------------
                                     1 1 1 1 1 1 0 1  = result
                                   1 overflow=0 causing carry to be set
```

The zero bit would be reset and the carry bit set, indicating E greater than A.

Example 3:

Assume that the accumulator contains -1BH. The internal subtraction now produces the following:

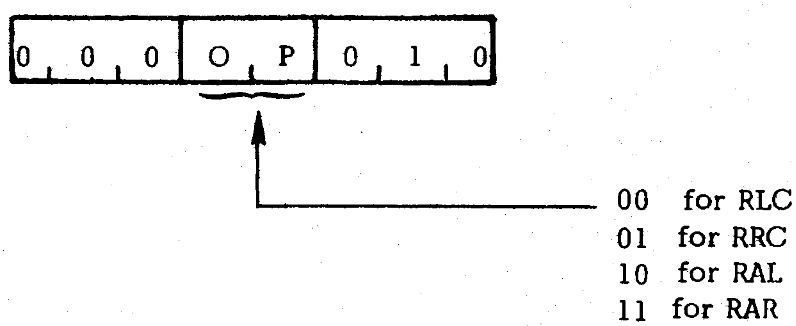
```
        Accumulator     =  -1BH = 1 1 1 0 0 1 0 1
  +     ( - E register) =  -5H  = 1 1 1 1 1 0 1 1
                                  ---------------
                                  1 1 1 0 0 0 0 0
                                1 overflow=1 causing carry to be reset
```

Since the two numbers to be compared differed in sign, the resetting of the carry bit now indicates E greater than A.

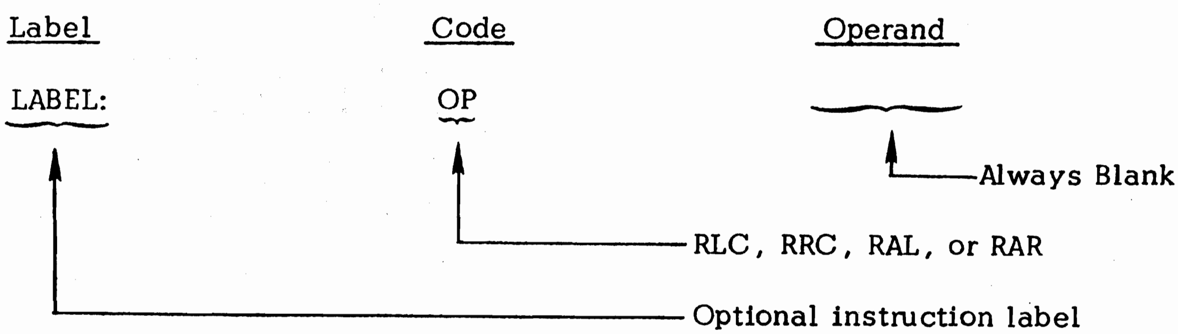
3.6 ROTATE ACCUMULATOR INSTRUCTIONS

This section describes the instructions which rotate the contents of the accumulator. No memory locations or other registers are referenced.

Instructions in this class occupy one byte as follows:



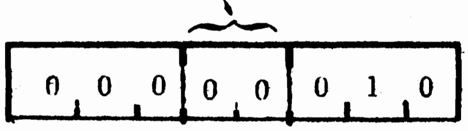
The general assembly language instruction format is:



3.6.1 RLC ROTATE ACCUMULATOR LEFT

Format:

Label	Code	Operand
	RLC	



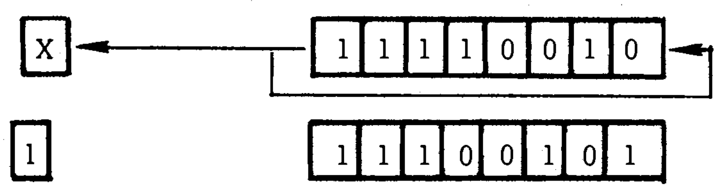
Description: The carry bit is set equal to the high order bit of the accumulator. The contents of the accumulator are rotated one bit position to the left, with the high-order bit being transferred to the low-order bit position of the accumulator.

Condition bits affected: Carry

Example:

Label	Code	Operand	Assembled Data
	MVI	A, 0F2H	06F2
	RLC		02

Before RLC is executed: Carry Accumulator



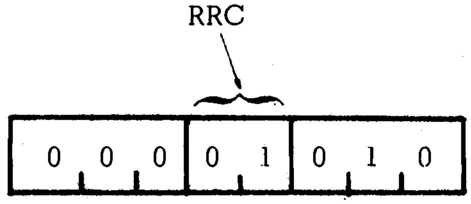
After RLC is executed:

Carry = 1 A = 0E5H

3.6.2 RRC ROTATE ACCUMULATOR RIGHT

Format:

Label	Code	Operand
	RRC	



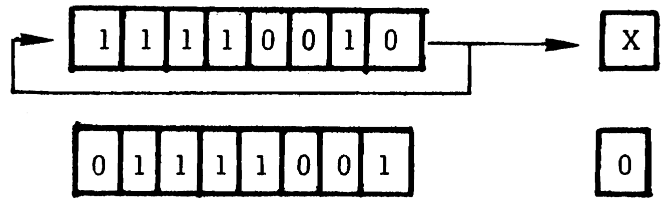
Description: The carry bit is set equal to the low-order bit of the accumulator. The contents of the accumulator are rotated one bit position to the right, with the low-order bit being transferred to the high-order bit position of the accumulator.

Condition bits affected: Carry

Example:

Label	Code	Operand	Assembled Data
	MVI	A, 0F2H	06F2
	RRC		0A

Before RRC is executed: Accumulator Carry



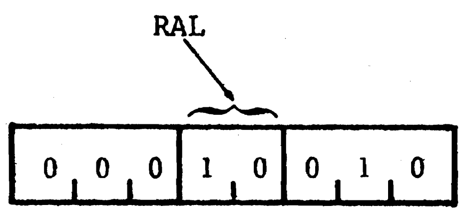
After RRC is executed:

A = 79H Carry = 0

3.6.3 RAL ROTATE ACCUMULATOR LEFT THROUGH CARRY

Format:

Label	Code	Operand



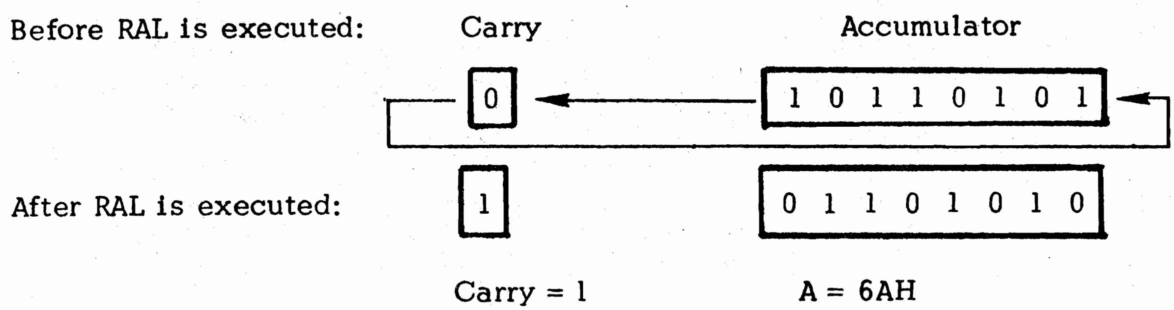
Description: The contents of the accumulator are rotated one bit position to the left.

The high-order bit of the accumulator replaces the carry bit, while the carry bit replaces the low-order bit of the accumulator.

Condition bits affected: Carry

Example:

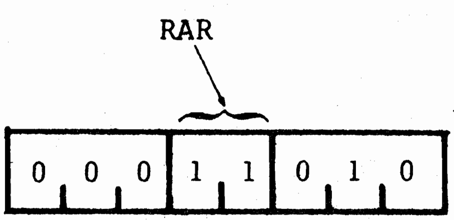
Label	Code	Operand	Assembled Data
	MVI	A, 0B5H	06B5
	RAL		12



3.6.4 RAR ROTATE ACCUMULATOR RIGHT THROUGH CARRY

Format:

Label	Code	Operand
	RAR	



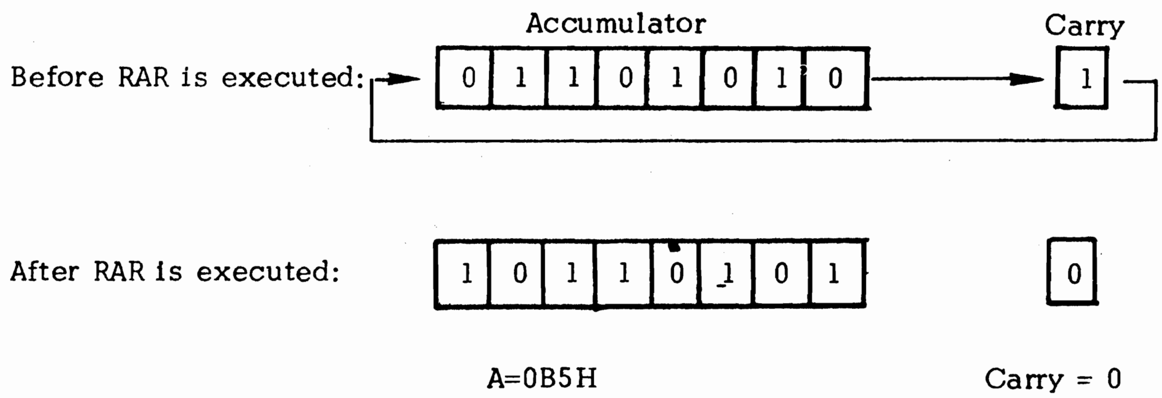
Description: The contents of the accumulator are rotated one bit position to the right.

The low-order bit of the accumulator replaces the carry bit, while the carry bit replaces the high-order bit of the accumulator.

Condition bits affected: Carry

Example:

Label	Code	Operand	Assembled Data
	MVI	A, 6AH	066A
	RAR		1A

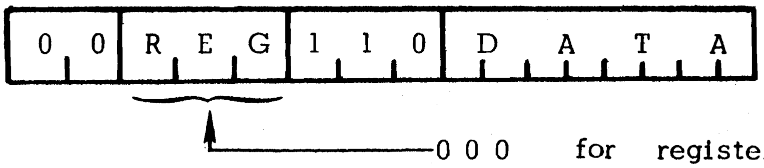

Accumulator
Carry
Before RAR is executed:
0 1 1 0 1 0 1 0
1
After RAR is executed:
1 0 1 1 0 1 0 1
0
A=0B5H
Carry = 0

3.7 IMMEDIATE INSTRUCTIONS

This section describes instructions which perform operations using a byte of data which is part of the instruction itself.

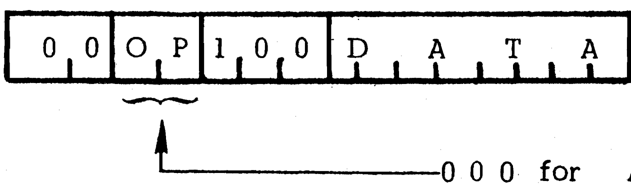
Instructions in this class occupy two bytes as follows:

(a) For the MVI instruction:



0 0 0	for	register A
0 0 1	for	register B
0 0 0	for	register C
0 1 1	for	register D
1 0 0	for	register E
1 0 1	for	register H
1 1 0	for	register L
1 1 1	for	memory reference

(b) For the remaining instructions:



0 0 0	for	ADI
0 0 1	for	ACI
0 1 0	for	SUI
0 1 1	for	SBI
1 0 0	for	ANI
1 0 1	for	XRI
1 1 0	for	ORI
1 1 1	for	CPI

When a memory reference is specified in the MVI instruction, the addressed location is specified by the H and L registers. The L register holds the low-order

8 bits of the address; the H register holds the high-order 6 bits of the address.

The general assembly language instruction format is:

Label	Code	Operand
LABEL:	MVI	REG, DATA

DATA — 8 - bit data quantity

REG — A, B, C, D, E, H, L, or M

LABEL: — Optional instruction label

- or -

Label	Code	Operand
LABEL:	OP	DATA

DATA — 8 - bit data quantity

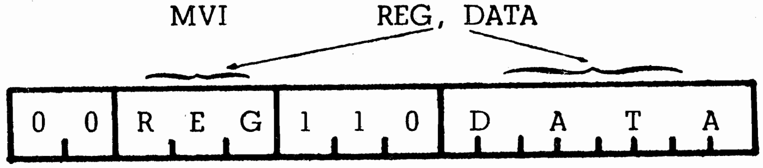
OP — ADI, ACI, SUI, SBI, ANI, XRI, ORI, or CPI

LABEL: — Optional instruction label

3.7.1 MVI MOVE IMMEDIATE DATA

Format:

Label	Code	Operand
	MVI	REG, DATA



Description: The byte of immediate data is stored in the register specified by REG, or in the memory location addressed by registers H and L (if REG = 111B).

Condition bits affected: None

Example:

Label	Code	Operand	Assembled Data
M1:	MVI	H, 3CH	2E3C
M2:	MVI	L, 0F4H	36F4
M3:	MVI	M, 0FFH	3EFF

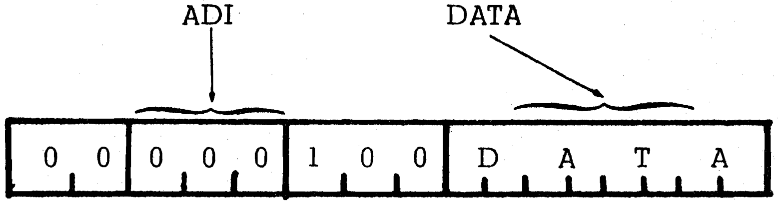
The instruction at M1: loads the H register with the byte of data at M1: + 1, i.e., 3CH.

Likewise, the instruction at M2: loads the L register with 0F4H. The instruction at M3: causes the data at M3: + 1 (0FFH) to be stored at memory location 3CF4H. The memory location is obtained by concatenating the contents of the H and L registers into a 14 bit address.

3.7.2 ADI ADD IMMEDIATE TO ACCUMULATOR

Format:

Label	Code	Operand
	ADI	DATA



Description: The byte of immediate data is added to the contents of the accumulator using two's complement arithmetic. The result is kept in the accumulator.

If there is an overflow out of the high-order bit position, the carry bit is set.

The zero bit is set if the result is zero.

The parity bit is set if the result contains an even number of ones (even parity).

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example:

Label	Code	Operand	Assembled Data
AD1:	MVI	A, 20	0614
AD2:	ADI	66	0442
AD3:	ADI	-66	04BE

The instruction at AD1: loads the accumulator with 14H. The instruction at AD2: performs the following addition:

```
      Accumulator    =  14H   = 00010100
AD2 Immediate Data   =  42H   = 01000010
                        Result = 01010110  = 56H = New accumulator
```

The parity bit is set. Other status bits are reset.

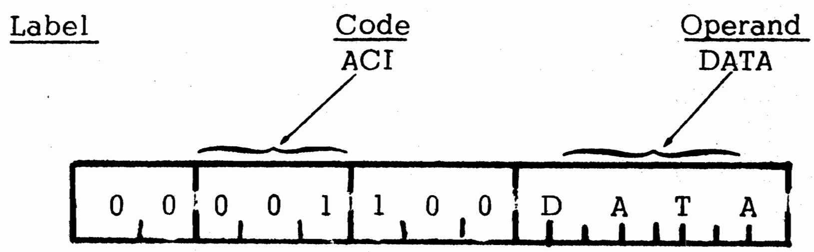
The instruction at AD3: restores the original contents of the accumulator by performing the following addition:

```
      Accumulator   =  56H  = 01010110
AD3 Immediate Data  =  0BEH = 10111110
                    Result  = 00010100 = 14H
```

The carry and parity bits are set. The zero and sign bits are reset.

3.7.3 ACI ADD IMMEDIATE TO ACCUMULATOR WITH CARRY

Format:



Description: The byte of immediate data is added to the contents of the accumulator plus the contents of the carry bit. The result is kept in the accumulator.

The carry bit is set if there is an overflow out of the high-order bit position.

The zero bit is set if the result is zero.

The parity bit is set if the result has even parity.

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example:

Label	Code	Operand	Assembled Data
C1:	MVI	A, 56H	0656
C2:	ACI	-66	0CBE
C3:	ACI	66	0C42

Assuming that the carry bit = 0 just before the instruction at C2: is executed, this instruction will produce the same result as instruction AD3: in the example of Section 3.7.2.

That is: Accumulator = 14H
Carry = 1

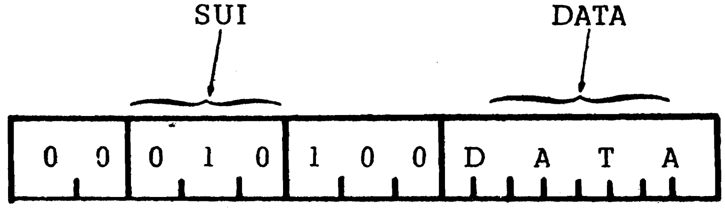
The instruction at C3: then performs the following addition:

	Accumulator	= 14H	= 0 0 0 1 0 1 0 0	
C3	Immediate Data	= 42H	= 0 1 0 0 0 0 1 0	
	Carry bit	= 1	= 1	
	Result		= 0 1 0 1 0 1 1 1	= 57H

3.7.4 SUI SUBTRACT IMMEDIATE FROM ACCUMULATOR

Format:

Label	Code	Operand
	SUI	DATA



Description: The byte of immediate data is subtracted from the contents of the accumulator using two's complement arithmetic. The result is stored in the accumulator.

Since this is a subtraction operation, the carry bit is set if there is no overflow out of the hig' -order bit position, and reset if there is an overflow.

The zero bit is set if the result is zero.

The parity bit is set if the result has even parity.

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example:

This instruction can be used as the equivalent of the DCR instruction applied to the accumulator. This is handy, since the instruction DCR A is illegal.

Label	Code	Operand	Assembled Data
	MVI	A, 0	0600
S1:	SUI	1	1401

The MVI instruction loads the accumulator with zero. The SUI instruction performs the following subtraction:

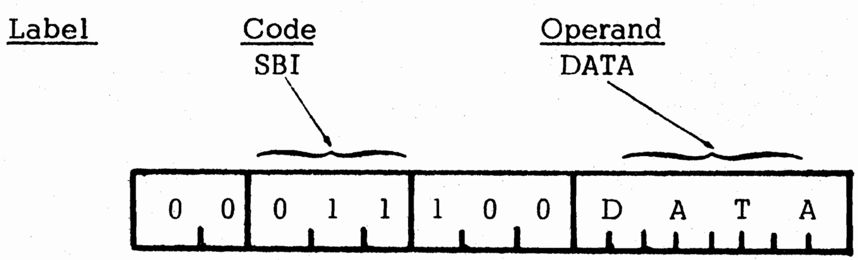
Accumulator	= 0H	=	0 0 0 0 0 0 0 0	
-S1 Immediate data	= -1H	=	1 1 1 1 1 1 1 1	two's complement
Result		=	1 1 1 1 1 1 1 1	= -1H

Since there was no overflow, and this is a subtract operation, the carry bit is set.

The zero bit is also reset, while the sign and parity bits are set.

3.7.5 SBI SUBTRACT IMMEDIATE FROM ACCUMULATOR WITH BORROW

Format:



Description: The carry bit is internally added to the byte of immediate data. This value is then subtracted from the accumulator using two's complement arithmetic. The result is stored in the accumulator; the byte of immediate data is unchanged.

This instruction and the SBB instruction are most useful when performing multibyte subtractions. For an example of this, see Section 5.3.

Since this is a subtraction operation, the carry bit is set if there is no overflow out of the high-order position, and reset if there is an overflow.

The zero bit is set if the result is zero.

The parity bit is set if the result has even parity.

The sign bit is set to the most significant bit of the result.

Condition bits affected: Carry, sign, zero, parity

Example:

Label	Code	Operand	Assembled Data
	XRA	A	A8
	SBI	1	1C01

The XRA instruction will zero the accumulator (see example in Section 3.5.6). If the carry bit is zero, the SBI instruction will produce exactly the same results as the example of Section 3.7.4.

If the carry bit is one, the SBI instruction will perform the following operation:

Immediate Data + Carry = 02H
Two's Complement of 02H = 11111110

Adding this to the accumulator produces:

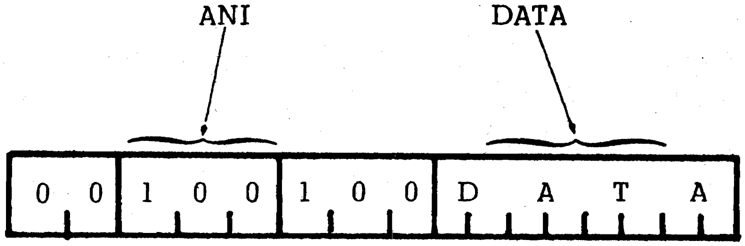
```
Accumulator = 0H= 0 0 0 0 0 0 0 0
                  1 1 1 1 1 1 1 0
                  1 1 1 1 1 1 1 0 = -2H = Result
                1 overflow = 0 causing carry to be set
```

This time the carry and sign bits are set, while the zero and parity bits are reset.

3.7.6 ANI AND IMMEDIATE WITH ACCUMULATOR

Format:

Label	Code	Operand
	ANI	DATA



Description: The byte of immediate data is logically ANDed with the contents of the accumulator.

The result is stored in the accumulator. The carry bit is set to zero, while the zero, sign, and parity bits are set according to the result.

Condition bits affected: Carry, zero, sign, parity

Example:

Label	Code	Operand	Assembled Data
	MOV	A,C	C2
A1:	ANI	0FH	240F

The contents of the C register are moved to the accumulator. The ANI instruction then zeroes the high-order four bits, leaving the low-order four bits unchanged. The zero bit will be set if and only if the low-order four bits were originally zero.

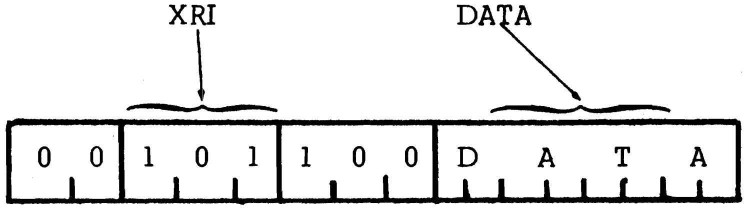
If the C register contained 3AH, the ANI would perform the following:

Accumulator =	3AH	= 0 0 1 1 1 0 1 0	
AND(A) Immediate data) =	0FH	= 0 0 0 0 1 1 1 1	
	Result =	0 0 0 0 1 0 1 0	= 0AH

3.7.7 XRI EXCLUSIVE - OR IMMEDIATE WITH ACCUMULATOR

Format:

Label	Code	Operand
	XRI	DATA



Description: The byte of immediate data is exclusive - ORed with the contents of the accumulator. The result is stored in the accumulator. The carry bit is set to zero, while the zero, sign and parity bits are set according to the result.

Condition bits affected: Carry, zero, sign, parity

Example:

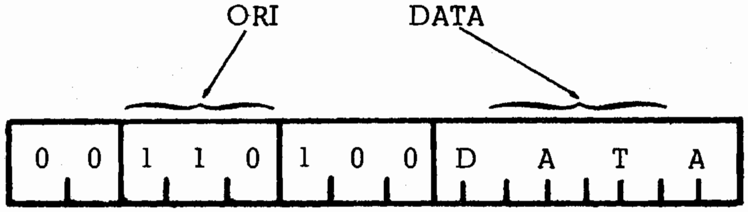
The following instructions cause the two's complement of the C register to be produced in the accumulator. (See Section 3.5.6).

Label	Code	Operand	Comment
	MOV	A,C	; C register to accumulator
	XRI	0FFH	; Produce one's complement
	ADI	1	; +1 = two's complement

3.7.8 ORI OR IMMEDIATE WITH ACCUMULATOR

Format:

Label	Code	Operand
	ORI	DATA



Description: The byte of immediate data is logically ORed with the contents of the accumulator.

The result is stored in the accumulator. The carry bit is set to zero, while the zero, sign, and parity bits are set according to the result.

Condition bits affected: Carry, zero, sign, parity

Example:

Label	Code	Operand	Assembled Data
	MOV	A,C	C2
ORI:	ORI	0FH	340F

The contents of the C register are moved to the accumulator. The ORI instruction then sets the low-order four bits to one, leaving the high-order four bits unchanged.

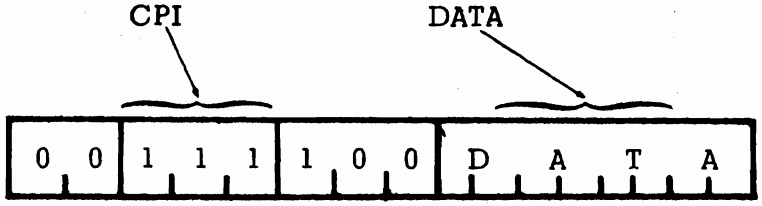
If the C register contained 0B5H, the ORI would perform the following:

Accumulator	= 0B5H	=	1 0 1 1 0 1 0 1		
OR(ORI Immediate data)	= 0FH	=	0 0 0 0 1 1 1 1		
Result		=	1 0 1 1 1 1 1 1	=	0BFH

3.7.9 CPI COMPARE IMMEDIATE WITH ACCUMULATOR

Format:

Label	Code	Operand
	CPI	DATA



Description: The byte of immediate data is compared to the contents of the accumulator.

The comparison is performed by internally subtracting the data from the accumulator using two's complement arithmetic, leaving the accumulator unchanged but setting the condition bits by the result.

In particular, the zero bit is set if the quantities are equal, and reset if they are unequal.

Since a subtract operation is performed, the carry bit will be set if there is no overflow out of bit 7, indicating that the immediate data is greater than the contents of the accumulator, and reset otherwise.

Note: If the two quantities to be compared differ in sign, the sense of the carry bit is reversed.

Condition bits affected: Carry, zero, sign, parity

Example:

Label	Code	Operand	Assembled Data
	MVI	A, 4AH	064A
	CPI	40H	3C40

The CPI instruction performs the following operation:

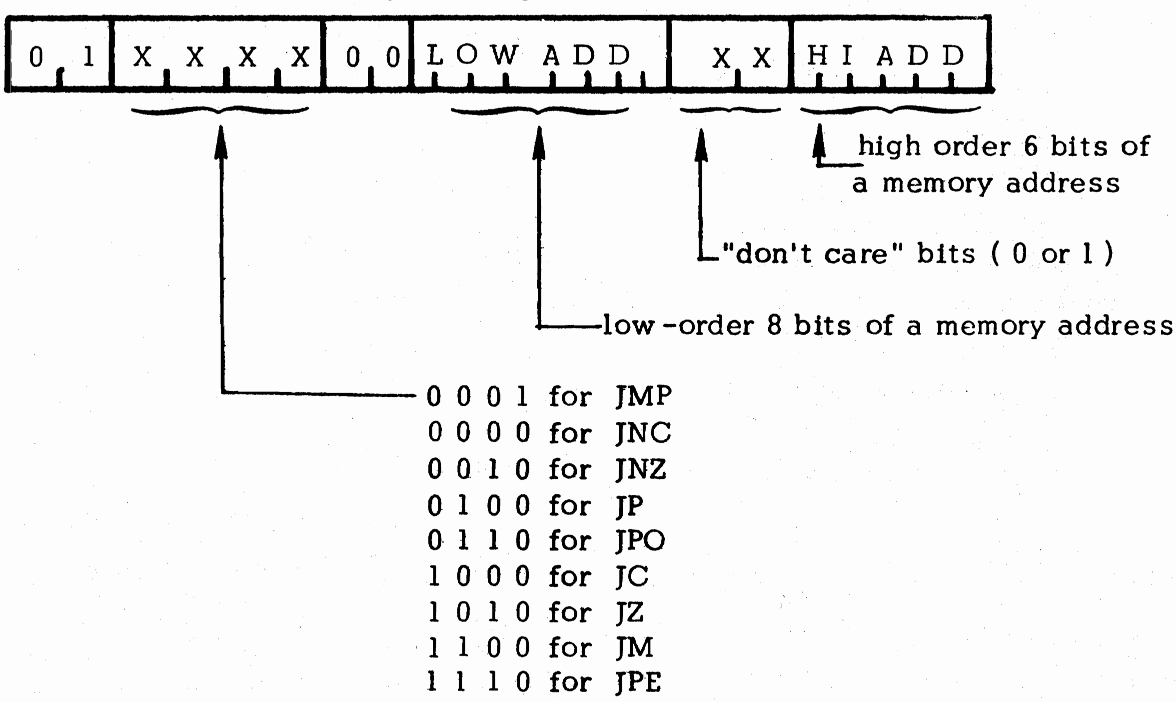
```
         Accumulator =  4AH =  0 1 0 0 1 0 1 0
+ (-Immediate data ) = -40H =  1 1 0 0 0 0 0 0
                              -----------------
                            1  0 0 0 0 1 0 1 0    Result
              Overflow = 1 causing carry to be reset
```

The accumulator still contains 4AH, but the zero bit is reset indicating that the quantities were unequal, and the carry bit is reset indicating DATA $<$ Accumulator.

3.8 JUMP INSTRUCTIONS

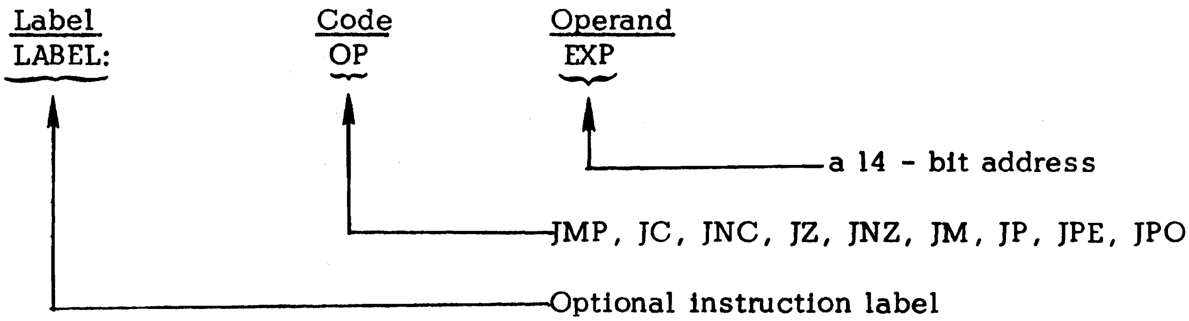
This section describes instructions which alter the normal execution sequence of instructions.

Instructions in this class occupy three bytes as follows:



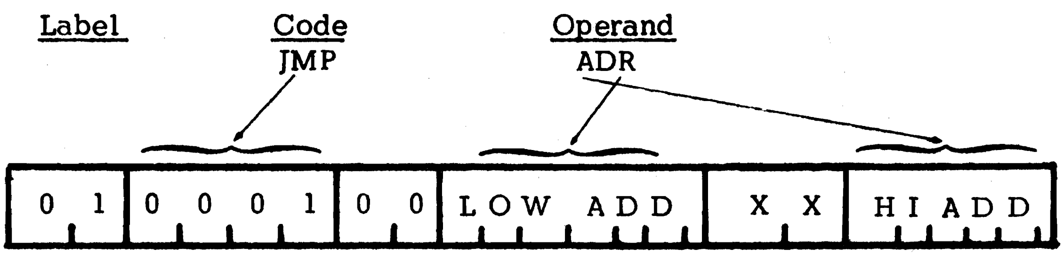
Note that, just as addresses are normally stored in memory with the low order byte first, so are the addresses represented in the Jump Instructions.

The general assembly language instruction format is:



3.8.1 JMP JUMP

Format:



Description: Program execution continues at the memory address ADR, formed by concatenating the 6 bits of HI ADD with the 8 bits of LOW ADD.

Condition bits affected: None

Example:

Arbitrary Memory Address	Label	Code	Operand	Assembled Data
3C00		JMP	CLR	44003E
3C03	AD:	ADI	2	0402
3D00	LOAD:	MVI	A, 3	0603
3D02		JMP	3C03H	44033C
3E00	CLR:	XRA	A	A8
3E01		JMP	$-101H	44003D

Normally, program instructions are executed sequentially . A 14 bit register called the program counter holds the address of the next instruction to be executed. When an instruction is fetched from memory (but before it is executed), the program counter is incremented by the length of the instruction. Thus, if a two byte instruction at address 3C00H is fetched, the next instruction will be fetched from address 3C02H. The JMP instruction replaces the program counter contents with a new address, causing program execution to continue at that address.

Thus the execution sequence of this example is as follows:

The JMP instruction at 3C00 replaces the contents of the program counter with 3E00. The next instruction executed is the XRA at CLR: , clearing the accumulator. The JMP at 3E01 is then executed.

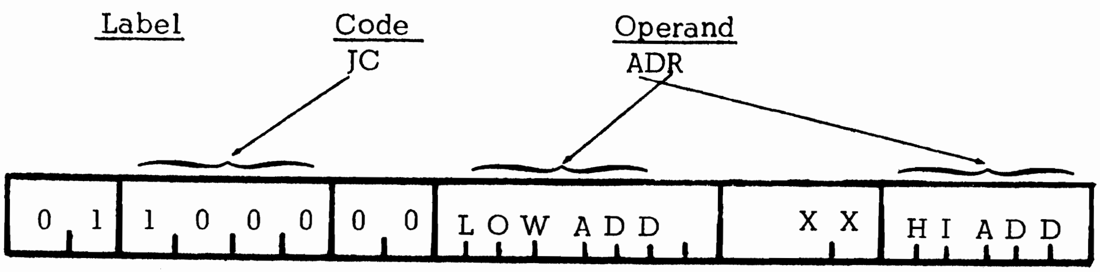
The "$" is a special character which the assembler interprets as the address of the instruction being assembled.

The program counter is set to 3D00, and the MVI at this address loads the accumulator with 3. The JMP at 3D02 sets the program counter to 3C03, causing the ADI instruction to be executed.

From here, normal program execution continues with the instruction 3C05.

3.8.2 JC JUMP IF CARRY

Format:



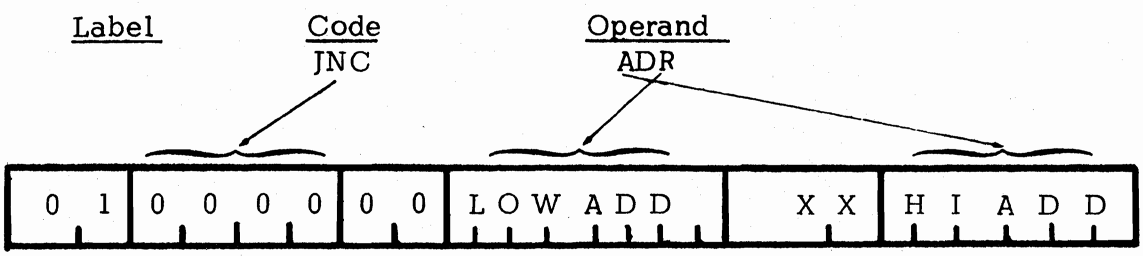
Description: If the carry bit is one, program execution continues at the memory address ADR. If the carry bit is zero, execution continues with the next sequential instruction.

Condition bits affected: None

For a programming example, see Section 3.8.9.

3.8.3 JNC JUMP IF NO CARRY

Format:



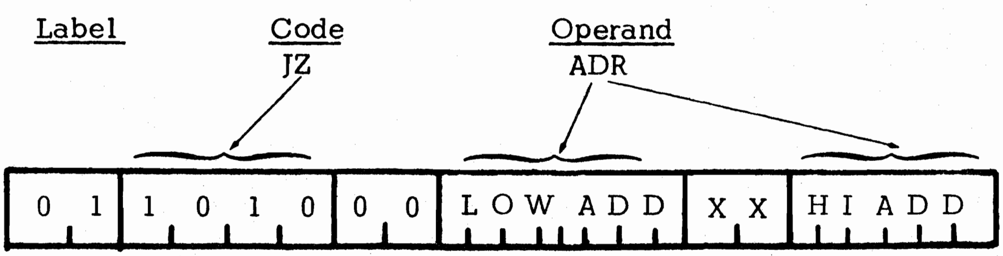
Description: If the carry bit is zero, program execution continues at the memory address ADR. If the carry bit is one, execution continues with the next sequential instruction.

Condition bits affected: None

For a programming example, see Section 3.8.9.

3.8.4 JZ JUMP IF ZERO

Format:



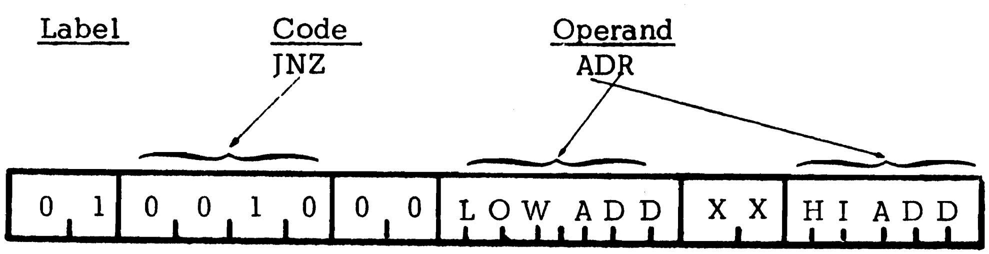
Description: If the zero bit is one, program execution continues at the memory address ADR. If the zero bit is zero, execution continues with the next sequential instruction.

Condition bits affected: None

For a programming example, see Section 3.8.9.

3.8.5 JNZ JUMP IF NOT ZERO

Format:



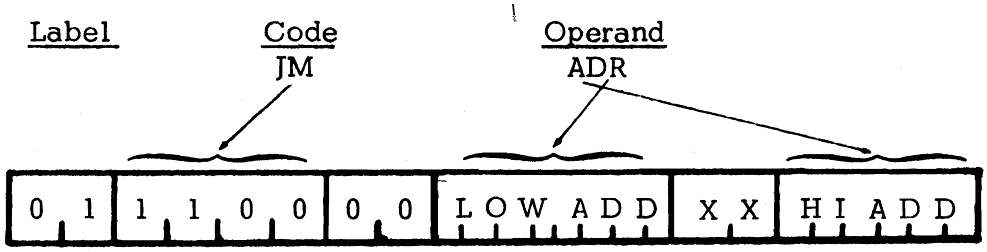
Description: If the zero bit is zero, program execution continues at the memory address ADR. If the zero bit is one, execution continues with the next sequential instruction.

Condition bits affected: None

For a programming example, see Section 3.8.9.

3.8.6 JM JUMP IF MINUS

Format:



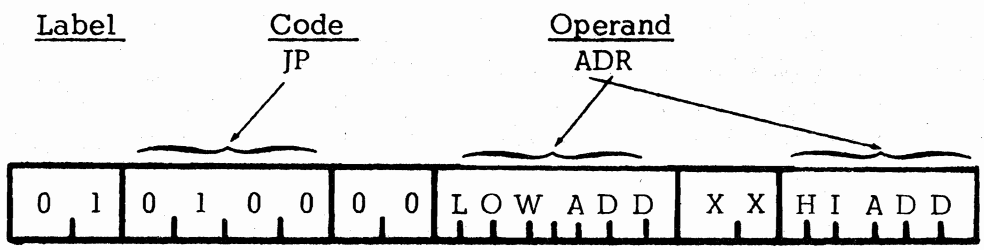
Description: If the sign bit is one (indicating a minus result), program execution continues at the memory address ADR. If the sign bit is zero, execution continues with the next sequential instruction.

Condition bit affected: None

For a programming example, see Section 3.8.9.

3.8.7 JP JUMP IF POSITIVE

Format:



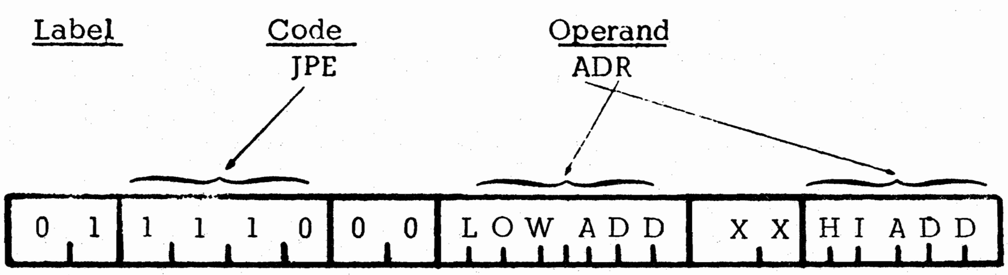
Description: If the sign bit is zero (indicating a positive result), program execution continues at the memory address ADR. If the sign bit is one, execution continues with the next sequential instruction.

Condition bits affected: None

For a programming example, see Section 3.8.9.

3.8.8 JPE JUMP IF PARITY EVEN

Format:

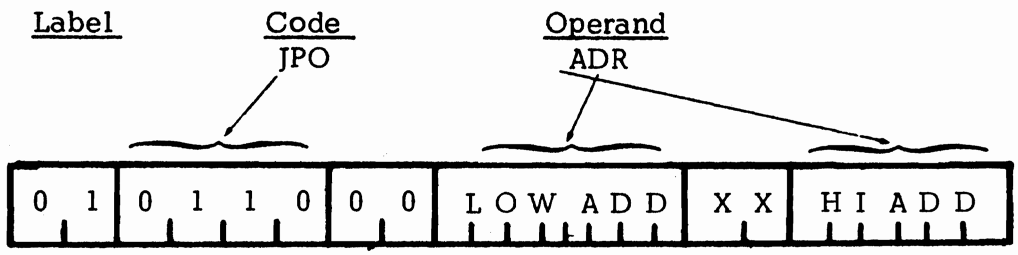


Description: If the parity bit is one (indicating a result with even parity), program execution continues at the memory address ADR. If the parity bit is zero, execution continues with the next sequential instruction.

Condition bits affected: None

3.8.9 JPO JUMP IF PARITY ODD

Format:



Description: If the parity bit is zero (indicating a result with odd parity), program execution continues at the memory address ADR. If the parity bit is one, execution continues with the next sequential instruction.

Condition bits affected: None

Examples of jump instruction:

Example:

This example shows three different but equivalent methods for jumping to one of two points in a program based upon whether or not the sign bit of a number is set. Assume that the byte to be tested is in the C register.

Label	Code	Operand	Assembled Data
ONE:	MOV	A,C	C2
	ANI	80H	2480
	JZ	PLUS	68XXXX
	JNZ	MINUS	48XXXX
TWO:	MOV	A,C	C2
	RLC		02
	JNC	PLUS	40XXXX
	JMP	MINUS	44XXXX
THREE:	MOV	A,C	C2
	ADI	0	0400
	JM	MINUS	70XXXX
PLUS:		; SIGN BIT RESET	
MINUS:		; SIGN BIT SET	

The AND - Immediate instruction in block ONE: zeroes all bits of the data byte except the sign bit, which remains unchanged. If the sign bit was zero, the zero condition bit will be set, and the JZ instruction will cause program control to be transferred to the instruction at PLUS: . Otherwise, the JZ instruction will merely update the program counter by three, and the JNZ instruction will be executed, causing control to be transferred to the instruction at MINUS: . (The zero bit is unaffected by any jump instructions).

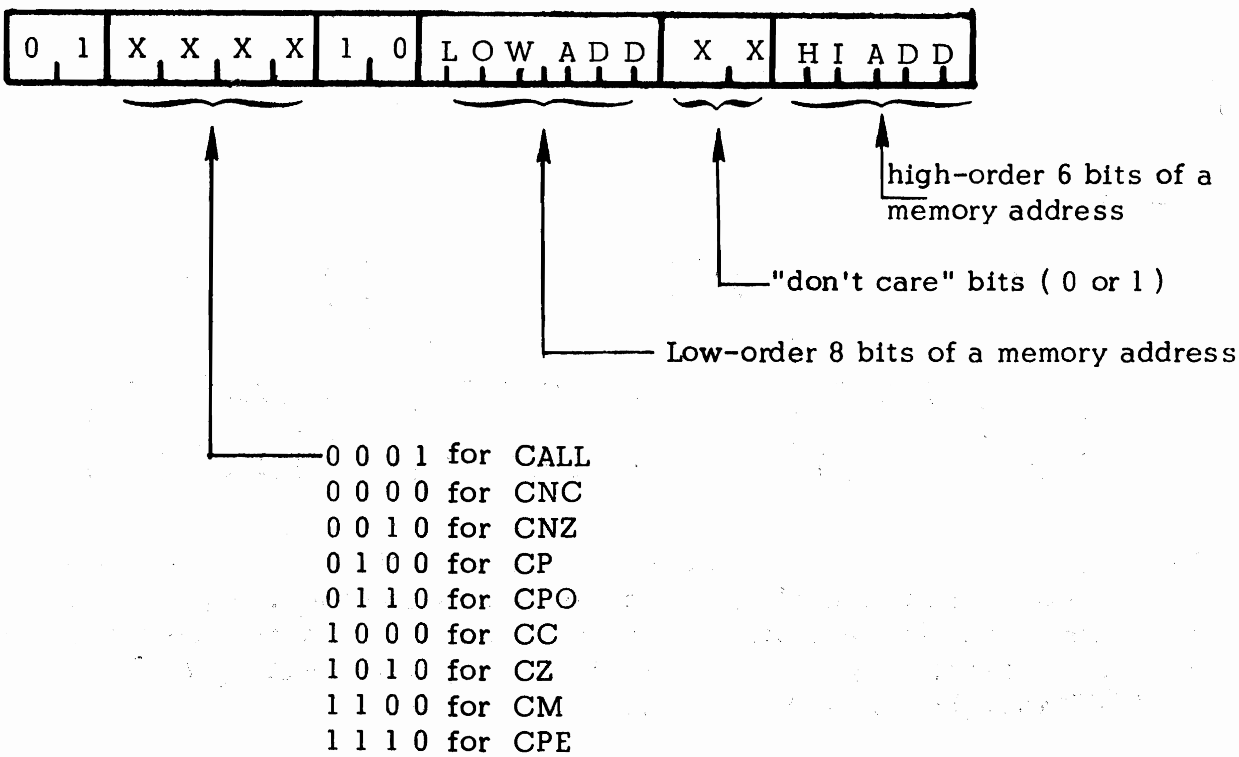
The RLC instruction in block TWO: causes the carry bit to be set equal to the sign bit of the data byte. If the sign bit was reset, the JNC instruction causes a jump to PLUS: . Otherwise the JMP instruction is executed, unconditionally transferring control to MINUS: . (Note that, in this instance, a JC instruction could be substituted for the unconditional jump with identical results).

The add-immediate instruction in block THREE: causes the condition bits to be set. If the sign bit was set, the JM instruction causes program control to be transferred to MINUS: . Otherwise, program control flows automatically into the PLUS: routine.

3.9 CALL SUBROUTINE INSTRUCTIONS

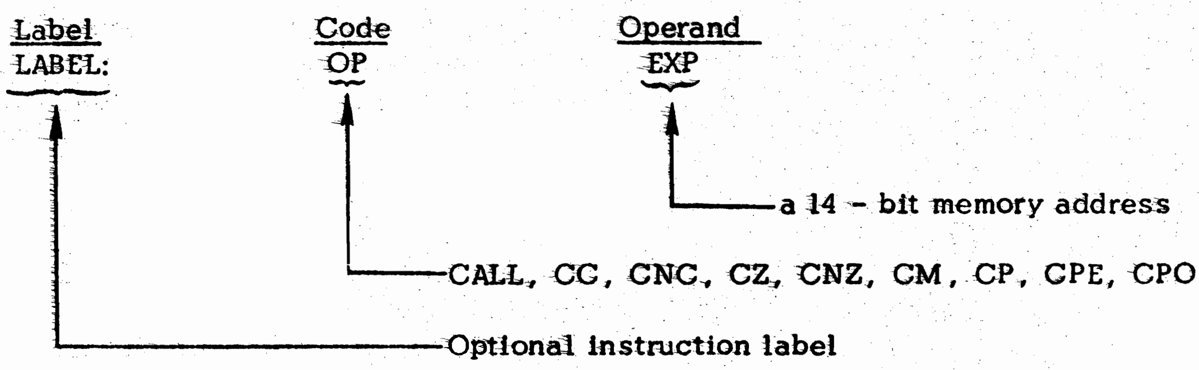
This section describes the instructions which call subroutines. These instructions operate like the jump instructions, causing a transfer of program control. In addition, a return address is saved on the address stack (see Section 2.4) for use by the RETURN instructions (Section 3.10). A discussion of the techniques and reasons for writing and using subroutines appears in Section 5.3

Instructions in this class occupy three bytes as follows:



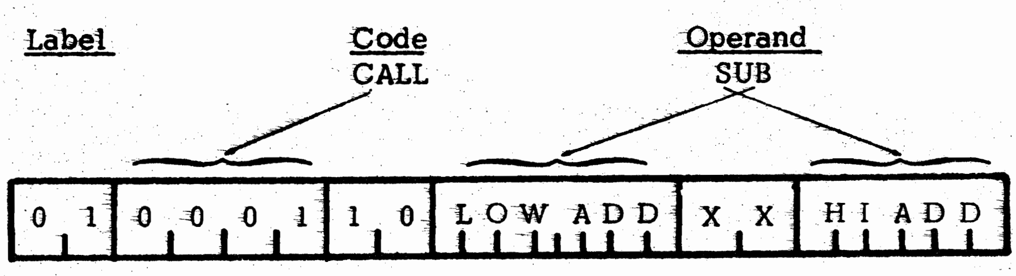
Note that, just as addresses are normally stored in memory with the low order byte first, so are the addresses represented the call instructions.

The general assembly language instruction format is:



3.9.1 CALL

Format:



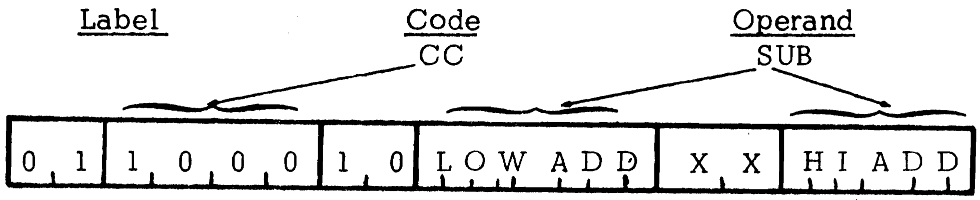
Description: The contents of the program counter, which equals the address of the instruction immediately following the CALL instruction, is placed on the address stack for later use by a Return instruction. Program execution continues at the memory address SUB, obtained by concatenating the 6 bits of HI ADD with the 8 bits of LOW ADD.

Condition bits affected: None

For programming examples see Section 5.3.

3.9.2 CC CALL IF CARRY

Format:



Description: If the carry bit is one, a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

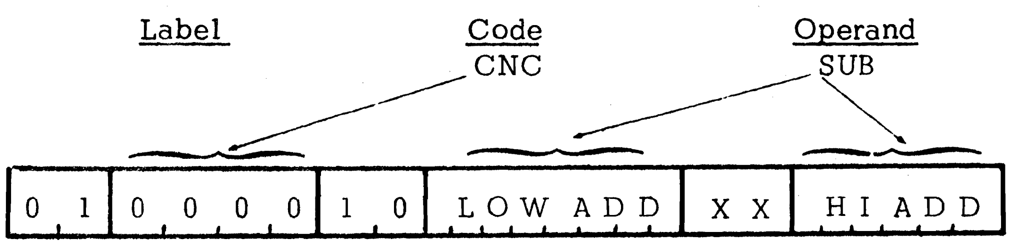
If the carry bit is zero, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.3 CNC CALL IF NO CARRY

Format:



Description: If the carry bit is zero, a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

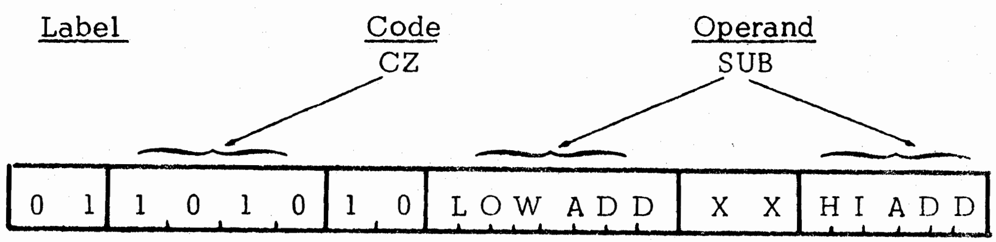
If the carry bit is one, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.4 CZ CALL IF ZERO

Format:



Description: If the zero bit is one, a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

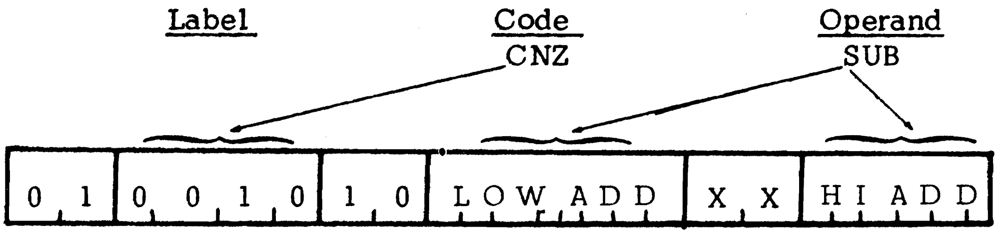
If the zero bit is zero, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.5 CNZ CALL IF NOT ZERO

Format:



Description: If the zero bit is zero, a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

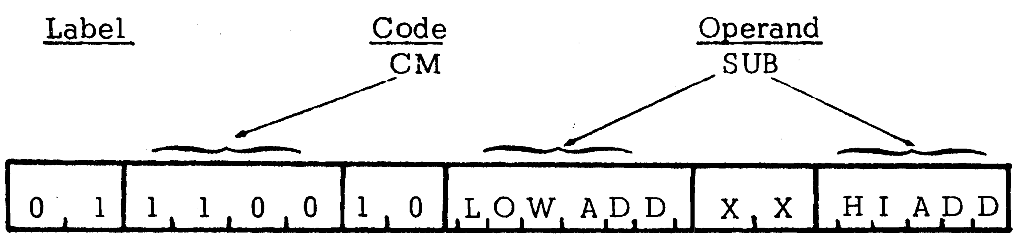
If the zero bit is one, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.6 CM CALL IF MINUS

Format:



Description: If the sign bit is one (indicating a minus result), a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

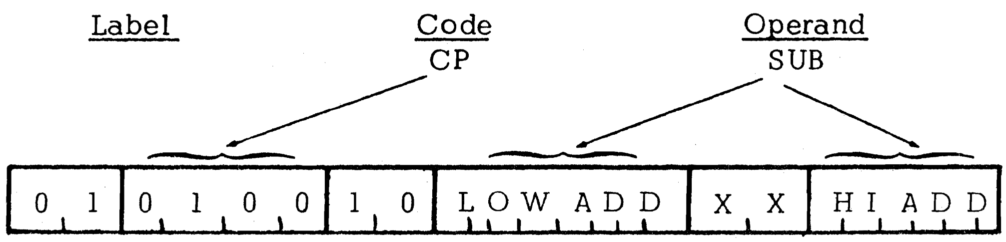
If the sign bit is zero, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.7 CP CALL IF PLUS

Format:



Description: If the sign bit is zero (indicating a positive result), a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

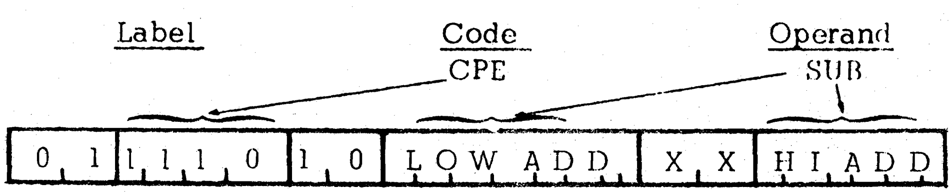
If the sign bit is one, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.8 CPE CALL IF PARITY EVEN

Format:



Description: If the parity bit is one (indicating even parity), a CALL is performed to subroutine SUB. The program counter is saved on the address stack and execution continues with the first instruction of SUB.

If the parity bit is zero, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.9.9 CPO CALL IF PARITY ODD

Format:

Label	Code	Operand
	CPO	SUB

0 1	0 1 1 0	1 0	LOW ADD	X X	HI ADD

Description: If the parity bit is zero (indicating odd parity), a CALL is performed to subroutine SUB. The program counter is saved on the address stack, and execution continues with the first instruction of SUB.

If the parity bit is one, program execution continues with the next sequential instruction.

Condition bits affected: None

For programming examples using subroutines, see Section 5.3.

3.10 RETURN FROM SUBROUTINE INSTRUCTIONS

This section describes the instructions used to return from subroutines. These instructions transfer program control to the last address saved on the address stack, and remove that address from the stack. A discussion of the techniques and reasons for writing and using subroutines appears in Section 5.3.

Instructions in this class occupy one byte as follows:

0	0	X	X	X	X	1	1

XXXX:

0 0 0 1 for RET
0 0 0 0 for RNC
0 0 1 0 for RNZ
0 1 0 0 for RP
0 1 1 0 for RPO
1 0 0 0 for RC
1 0 1 0 for RZ
1 1 0 0 for RM
1 1 1 0 for RPE

The general assembly language instruction format is:

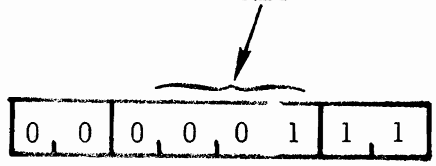
Label	Code	Operand
LABEL:	OP	

- Label: Optional statement label
- Code (OP): RET, RC, RNC, RZ, RNZ, RM, RP, RPE, RPO
- Operand: blank

3.10.1 RET RETURN

Format:

Label	Code	Operand
	RET	



Description: The last address saved on the address stack (by a call instruction) is removed from the stack and placed in the program counter.

Thus, execution proceeds with the instruction immediately following the last call instruction.

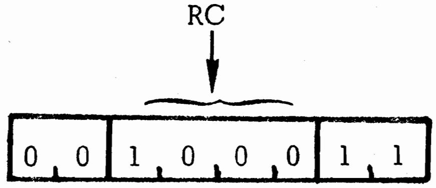
Condition bits affected: None

For programming examples see Section 5.3.

3.10.2 RC RETURN IF CARRY

Format:

Label	Code	Operand
	RC	



Description: If the carry bit is one, a return operation is performed. Otherwise, program execution continues with the next sequential instruction.

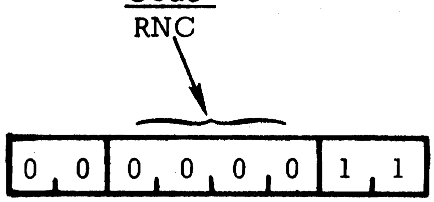
Condition bits affected: None

For programming examples, see Section 5.3.

3.10.3 RNC RETURN IF NO CARRY

Format:

Label	Code	Operand



Description: If the carry bit is zero, a return operation is performed. Otherwise, program execution continues with the next sequential instruction.

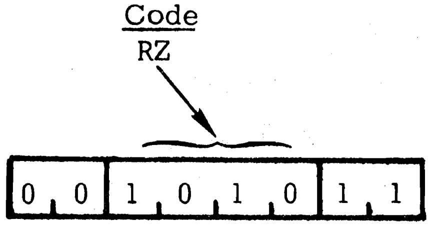
Condition bits affected: None

For programming examples, see Section 5.3.

3.10.4 RZ RETURN IF ZERO

Format:

Label	Code	Operand



Description: If the zero bit is one, a return operation is performed. Otherwise, program execution continues with the next sequential instruction.

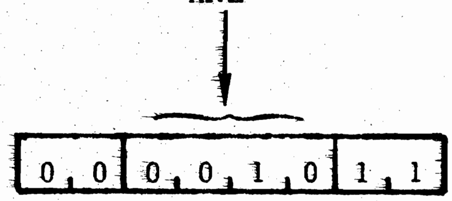
Condition bits affected: None

For programming examples, see Section 5.3.

3.10.5 RNZ RETURN IF NOT ZERO

Format:

Label	Code	Operand
	RNZ	



Description: If the zero bit is zero, a return operation is performed. Otherwise, program execution continues with the next sequential instruction.

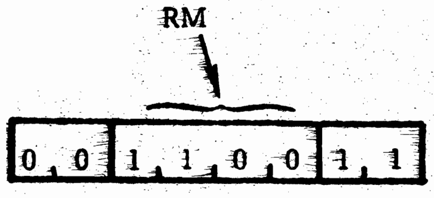
Condition bits affected: None

For programming examples, see Section 5.3.

3.10.6 RM RETURN IF MINUS

Format:

Label	Code	Operand
	RM	



Description: If the sign bit is one (indicating a minus result), a return operation is performed.

Otherwise, program execution continues with the next sequential instruction.

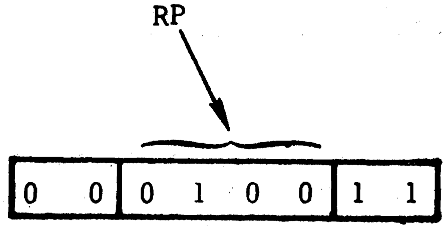
Condition bits affected: None

For programming examples see Section 5.3.

3.10.7 RP RETURN IF PLUS

Format:

Label	Code	Operand
	RP	



Description: If the sign bit is zero, (indicating a positive result), a return operation is performed.

Otherwise, program execution continues with the next sequential instruction.

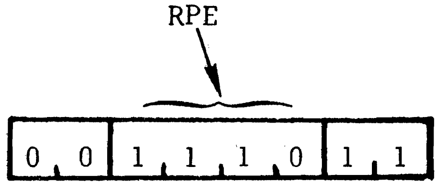
Condition bits affected: None

For programming examples see Section 5.3.

3.10.8 RPE RETURN IF PARITY EVEN

Format:

Label	Code	Operand
	RPE	



Description: If the parity bit is one (indicating even parity), a return operation is performed.

Otherwise, program execution continues with the next sequential instruction.

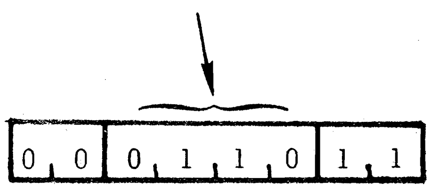
Condition bits affected: None

For programming examples see Section 5.3.

3.10.9 RPO RETURN IF PARITY ODD

Format:

Label	Code	Operand
	RPO	



Description: If the parity bit is zero,(indicating odd parity), a return operation is performed.

Otherwise, program execution continues with the next sequential instruction.

Condition bits affected: None

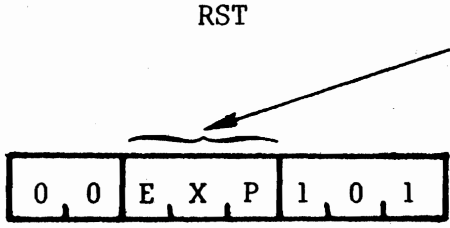
For programming examples see Section 5.3.

3.11 RST INSTRUCTION

This section describes the RST (restart) instruction, which is a special purpose subroutine jump. This instruction occupies one byte.

Format:

Label	Code	Operand
	RST	EXP



Note: EXP must evaluate to a number in the range 000B to 111B.

Description: The contents of the program counter are placed on the address stack, providing a return address for later use by a RETURN instruction.

Program execution continues at memory address:

0 0 0 0 0 0 0 0 E X P 0 0 0 B

Normally, this instruction is used in conjunction with up to eight eight-byte routines in the lower 64 words of memory in order to service interrupts to the processor. The interrupting device causes a particular RST instruction to be executed, transferring control to a subroutine which deals with the situation, as described in Section 6.

A RETURN instruction then causes the program which was originally running to resume execution at the instruction where the interrupt occurred.

Condition bits affected: None

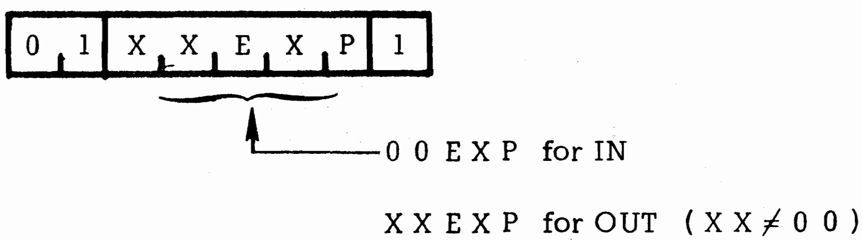
Example:

Label	Code	Operand		Comment
	RST	10 - 7	;	Call the subroutine at
			;	address 24 (011000B)
	RST	D SHL 1	;	Call the subroutine at address
			;	48 (110000B). D is equated
			;	to 11B.
	RST	8	;	Invalid instruction
	RST	3	;	Call the subroutine at
			;	address 24 (011000B)

For detailed examples of interrupt handling see Section 6.

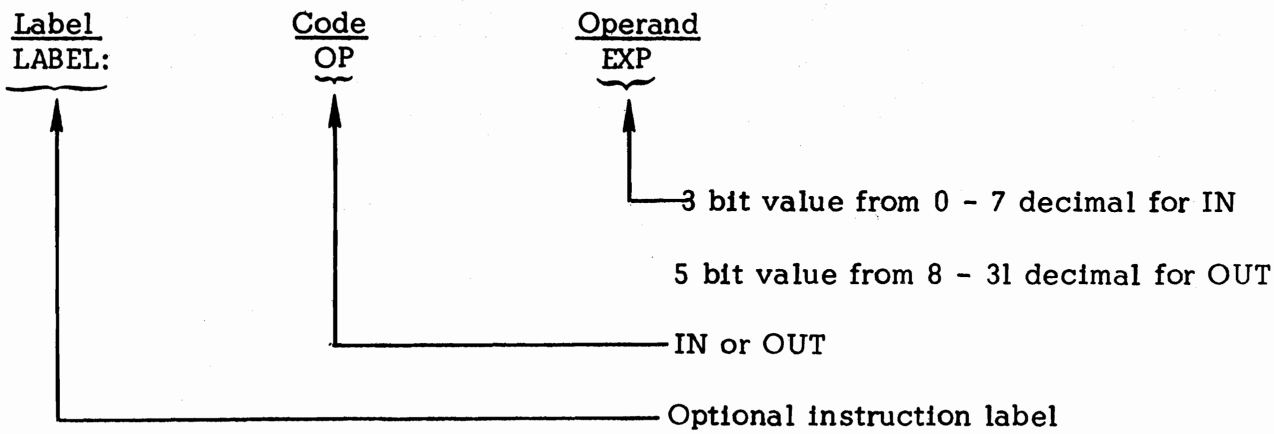
3.12 INPUT/OUTPUT INSTRUCTIONS

This section describes the instructions which cause data to be input to or output from the 8008. Instructions in this class occupy one byte as follows:



XXEXP is an input or output device number, which is a hardware characteristic of the device, not under the programmer's control.

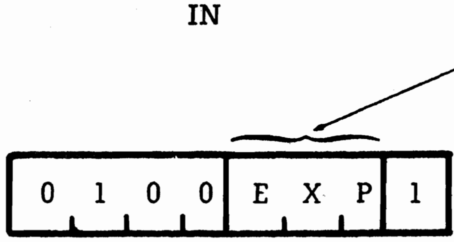
The general assembly language format is:



3.12.1 IN INPUT

Format:

Label	Code	Operand
	IN	EXP



Description: An eight bit data byte is read from input device number EXP (0 - 7) and replaces the contents of the accumulator.

Condition bits affected: None

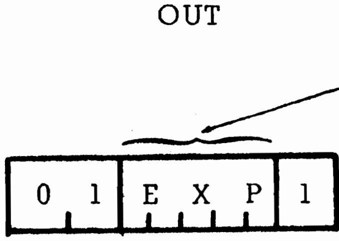
Example:

Label	Code	Operand	Comment
	IN	0	; Read one byte from input
			; device #0 into the accumulator
	IN	10/2	; Read one byte from input
			; device #5 into the accumulator
	IN	8	; Invalid instruction

3.12.2 OUT OUTPUT

Format:

Label	Code	Operand
	OUT	EXP



Description: The contents of the accumulator are written to output device number EXP (8 - 31).

Condition bits affected: None

Example:

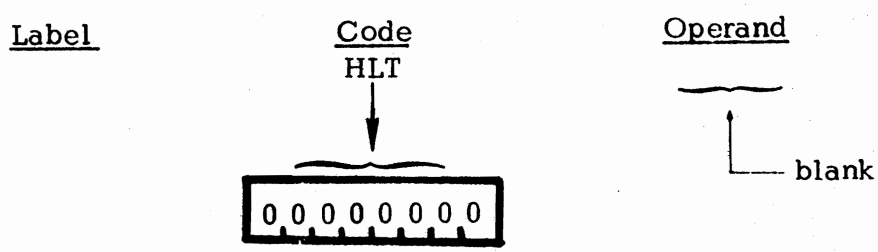
Label	Code	Operand	Comment
	OUT	10	; Write the contents of the
			; accumulator to output device #10
	OUT	1FH	; Write the contents of the
			; accumulator to output device #31
	OUT	7	; Invalid instruction

3.13 HLT HALT INSTRUCTION

This section describes the HLT instruction.

This instruction occupies one byte.

Format:



Description: The program counter is incremented to the address of the next sequential instruction. The CPU then enters the STOPPED state and no further activity takes place until an interrupt occurs.

3.14 PSEUDO - INSTRUCTIONS

This section describes pseudo instructions recognized by the assembler. A pseudo instruction is written in the same fashion as the machine instructions described in Sections 3.3 - 3.13, but does not cause any object code to be generated. It acts merely to provide the assembler with information to be used subsequently while generating object code.

The general assembly language format of a pseudo - instruction is:

Label	Code	Operand	Comment
NAME	OP	OPND	

OPND — Operand, may be optional

OP — ORG, EQU, SET, END, IF, ENDIF, MAC, ENDM

NAME — May be required, optional, or illegal

Note: Names on pseudo instructions do not end in colons, as do labels on machine operations.

3.14.1 ORG ORIGIN

Format:

Label	Code	Operand
	ORG	EXP

Label: Always blank

Operand: A 14 - bit address

Description: The assembler's location counter, identified by the special character $, is set to the value of EXP, which must be a valid 14 bit memory address. The next machine instruction or data byte (s) generated will be assembled at address EXP, EXP + 1, etc.

If no ORG appears before the first machine instruction or data byte in the program, assembly will begin at location 0.

Example 1:

Memory Address	Label	Code	Operand	Assembled Data
		ORG	1000H	
1000		MOV	A,C	C2
1001		ADI	2	0402
1003		JMP	NEXT	445010
		ORG	1050H	
1050	NEXT:	XRA	A	A8
		=		

The first ORG pseudo instruction informs the assembler that the object program will begin at memory address 1000H. The second ORG tells the assembler to set its location counter to 1050H and continue assembling machine instructions or data bytes from that point. Note that the range of memory from 1006H to 104F is still included in the object program, but does not contain assembled data. In particular, the programmer should not assume that these locations will contain zero, or any other value.

Example 2:

The ORG pseudo instruction can perform a function equivalent to the DS (define storage) instruction (see Section 3.2.4). The following two sections of code are exactly equivalent:

Memory Address	Label	Code	Operand	Label	Code	Operand	Assembled Data
2C00		MOV	A,C		MOV	A,C	C2
2C01		JMP	NEXT		JMP	NEXT	44102C
2C04		DS	12		ORG	$+12	
2C10	NEXT:	XRA	A	NEXT:	XRA	A	A8

3.14.2 EQU EQUATE

Format:

Label	Code	Operand
NAME	EQU	EXP

NAME — Required name

EXP — An expression

Description: The symbol NAME is assigned the value of EXP by the assembler. Whenever the symbol NAME is encountered subsequently in the assembly, this value will be used.

Note: A symbol may appear in the name field of only one EQU pseudo instruction: i.e., an EQU symbol may not be redefined.

Example 1:

Before every assembly, the assembler performs the following EQU statements:

Label	Code	Operand
A	EQU	0
B	EQU	1
C	EQU	2
D	EQU	3
E	EQU	4
H	EQU	5
L	EQU	6
M	EQU	7

If this were not done, a statement like:

```
MOV        C, A
```

would be invalid, forcing the programmer to write:

```
MOV        2, 0
```

Example 2:

The EQU and ORG pseudo instructions can define the DS operation of Section 3.2.4.

The following two sections of code are equivalent:

Memory Address	Label	Code	Operand	Label	Code	Operand	Assembled Data
2C00		MOV	A,C		MOV	A,C	C2
2C01		JMP	NEXT		JMP	NEXT	44072C
2C04	DATA:	DS	3	DATA	EQU	$	
					ORG	$+3	
2C07	NEXT:	XRA	0	NEXT:	XRA	A	A8

A reference to DATA will address a three byte block of memory beginning at location 2C04H.

3.14.3 SET

Format:

Label	Code	Operand
NAME	SET	EXP

NAME — Required name

EXP — An expression

Description: The symbol NAME is assigned the value of EXP by the assembler. Whenever the symbol NAME is encountered subsequently in the assembly, this value will be used unless changed by another SET instruction.

This is identical to the EQU operation, except that symbols may be defined more than once.

Example:

Label	Code	Operand	Assembled Data
IMMED	SET	5	
	ADI	IMMED	0405
IMMED	SET	10H-6	
	ADI	IMMED	040A

3.14.4 END END OF ASSEMBLY

Format:

Label	Code	Operand
blank	END	blank

Description: The END statement signifies to the assembler that the physical end of the program has been reached, and that generation of the object program and (possibly) listing of the source program should now begin.

One and only one END statement must appear in every assembly, and it must be the (physically) last statement of the assembly.

3.14.5 IF AND ENDIF CONDITIONAL ASSEMBLY

Format:

Label	Code	Operand
blank	IF	EXP — An expression
	statements	
blank	ENDIF	blank

Description: The assembler evaluates EXP.

If EXP evaluates to zero, the statements between IF and ENDIF are ignored. Otherwise the intervening statements are assembled as if the IF and ENDIF were not present.

Example:

Label	Code	Operand	Assembled Data
COND	SET	0FFH	
	IF	COND	
	MOV	A,C	C2
	ENDIF		
COND	SET	0	
	IF	COND	
	MOV	A,C	
	ENDIF		
	XRA	C	AA

3.14.6 MACRO AND ENDM MACRO DEFINITION

Format:

Label	Code	Operand
NAME	MACRO	LIST
Required name		a list of expressions, normally ASCII constants
	statements	
blank	ENDM	blank

Description: For a detailed explanation of the definition and use of marcos together with programming examples, see Section 4.

The assembler accepts the statements between MACRO and ENDM as the definition of the macro named NAME. Upon encountering NAME in the code field of an instruction, the assembler substitutes the parameters specified in the operand field of the instruction for the occurences of LIST in the macro definition, and assembles the statements.

Note: The pseudo instruction MACRO may not appear in the list of statements between MACRO and ENDM; i.e., macros may not define other macros.

4.0 PROGRAMMING WITH MACROS

Macros (or macro instructions) are an extremely important tool provided by the assembler. Properly utilized, they will increase the efficiency of programming and the readability of programs. It is strongly suggested that the user become familiar with the use of macros and utilize them to tailor programming to suit his specific needs.

4.1 WHAT ARE MACROS?

A macro is a means of specifying to the assembler that a symbol (the macro name) appearing in the code field of a statement actually stands for a group of instructions. Both the macro name and the instructions for which it stands are chosen by the programmer.

Consider a simple macro which shifts the contents of the accumulator one bit position right, while a zero is shifted into the high order bit position. We will call this macro SHRT, and define it as follows:

Label	Code	Operand	
SHRT	MACRO		
	RRC		; Rotate accumulator right
	ANI	7FH	; Clear high order bit
	ENDM		

We can now reference the macro as follows:

Label	Code	Operand
	MOV	A,M
	SHRT	

which would be equivalent to:

Label	Code	Operand
	MOV	A,M
	RRC	
	ANI	7FH

The example above illustrates three aspects of a macro; the definition, the reference and the expansion.

The definition specifies the instruction sequence that is to be represented by the macro label. Thus:

```
SHRT      MACRO
          RRC
          ANI      7FH
          ENDM
```

is the definition of SHRT, and specifies that SHRT stands for the two instructions:

```
          RRC
          ANI      7FH
```

Every macro must be defined once in a program.

The reference is the point in a program where the macro is referenced. A macro may be referenced any number of times by inserting the macro label in the code field of an instruction:

```
          MOV      A,M
          SHRT                ; Macro reference
          MOV      A,M
```

The expansion of a macro is the complete instruction sequence represented by the macro reference:

```
          MOV      A,M
          RRC               }  Macro expansion
          ANI      7FH      }
          MOV      M,A
```

The macro expansion will not be present in a source program, but its machine language equivalent will be generated by the assembler in the object program.

You may question the value of representing two instructions by a macro, but consider a more complex case, a macro that shifts the accumulator right by a variable number of bit position, as defined by the D register contents.

This macro is labeled SHV, and defined as follows:

Label	Code	Operand	
SHV	MACRO		
LOOP:	RRC		; rotate right once
	ANI	7FH	; clear the high-order bit
	DCR	D	; decrement shift counter
	JNZ	LOOP	; return for another shift
	ENDM		

The SHV macro may be referenced as follows:

Label	Code	Operand	
	MOV	A,M	
	MVI	D, 3	; specify 3 right shifts
	SHV		
	MOV	M,A	

The above instruction sequence is equivalent to the expansion:

Label	Code	Operand
	MOV	A,M
	MVI	D, 3
LOOP:	RRC	
	ANI	7FH
	DCR	D
	JNZ	LOOP
	MOV	M,A

Note that the D register is no longer available for general use across the SHV macro, since it is used to specify shift count.

A better method is to write a macro which uses an arbitrary register and loads its own shift amount using <u>macro parameters</u>. The macro is defined as follows:

```
Label       Code      Operand
SHV         MACRO     REG, AMT
            MVI       REG, AMT ; load shift count into register specified by REG
LOOP:       RRC                ; perform right rotate
            ANI       7FH      ; clear high order bit
            DCR       REG      ; decrement shift counter
            JNZ       LOOP
            ENDM
```

SHV may now be referenced as follows:

```
            MOV       A,M
; Assume  Register C is free, and a 5 place shift is needed,
            SHV       C, 5
```

The expansion of which is given by:

```
            MVI
LOOP:       RRC
            ANI       7FH
            DCR       C
            JNZ       LOOP
```

Here is another example of an SHV reference:

```
; Assume Register E is free, and a 2 place shift is needed,
            SHV       E, 2
```

and the equivalent expansion:

```
            MVI       E, 2
LOOP:       RRC
            ANI       7FH
            DCR       E
            JNZ       LOOP
```

While the preceding examples will provide a general idea of the efficiency and capabilities of macros, a rigorous description of each aspect of macro programming is given in the next section.

4.2 MACRO TERMS AND USE

Section 4.1 explains how a macro must be defined, is then referenced, and how every reference has an equivalent expansion. Each of these three aspects of a macro will be described in the following subsections.

4.2.1 MACRO DEFINITION

Format:

Label	Code	Operand
NAME	MACRO	PLIST

```
                     macro body

                       ENDM
```

Description: The macro definition produces no assembled data in the object program. It merely indicates to the assembler that the symbol NAME is to be considered equivalent to the group of statements appearing between the pseudo instructions MACRO and ENDM (Section 3.14.6). This group of statements,called the macro body, may consist of assembly language instructions, pseudo instructions (except MACRO or ENDM), comments, or references to other macros.

PLIST is a list of expressions (usually unquoted character strings) which indicate parameters specified by the macro reference that are to be substituted into the macro body. Since these expressions serve only to mark the positions where macro parameters will be inserted into the macro body, they are often called dummy parameters.

Example:

The following macro will load the H and L registers with the memory address of the label specified by the macro reference.

Label	Code	Operand
LOAD	MACRO	ADDR
	MVI	H, ADDR SHR 8
	MVI	L, ADDR AND 0FFH
	ENDM	
	—	
LABEL:	—	
	—	
INST:	—	

The reference:

```
        LOAD        LABEL
```

is equivalent to the expansion:

```
        MVI         H, LABEL   SHR 8
        MVI         L, LABEL AND 0FFH
```

The reference:

```
        LOAD        INST
```

is equivalent to the expansion:

```
        MVI         H, INST  SHR 8
        MVI         L, INST AND 0FFH
```

The MACRO and ENDM statements inform the assembler that when the symbol LOAD appears in the code field of a statement, the characters appearing in the operand field of the statement are to be substituted everywhere the symbol ADDR appears in the macro body, and the two MVI instructions are to be assembled at that point of the program.

4.2.2 MACRO REFERENCE OR CALL

Format:

Label	Code	Operand
	NAME	PLIST

NAME must be the name of a macro; that is, NAME appears in the label field of a MACRO pseudo - instruction.

PLIST is a list of expressions. Each expression is converted to a character string, and the resulting strings are substituted into the macro body as indicated by the operand field of the MACRO pseudo instruction. Substitution proceeds left to right; that is, the first string of PLIST replaces every occurrence of the first dummy parameter in the macro body, the second replaces the second, and so on.

If fewer parameters appear in the macro reference than in the definition, a null string is substituted for the remaining expressions in the definition.

If more parameters appear in the reference than the definition, the extras are ignored.

Example:

Given the macro definition:

Label	Code	Operand
MAC1	MACRO	P1, P2, COMMENT
	XRA	P2
	DCR	P1 COMMENT
	ENDM	

The reference:

```
                MAC1            C, D  ; DECREMENT REG C'
```

is equivalent to the expansion:

```
                XRA             D
                DCR             C     ; DECREMENT REG C
```

The reference:

```
            MAC1            E, B
```

is equivalent to the expansion:

```
            XRA             B
            DCR             E
```

4.2.3 MACRO EXPANSION

The result obtained by substituting the macro parameters into the macro body is called the macro expansion. The assembler assembles the statements of the expansion exactly as it assembles any other statements. In particular, every statement produced by expanding the macro must be a legal assembler statement.

Example:

Given the macro definition:

Label	Code	Operand
MAC	MACRO	P1
	INR	P1
	ENDM	

the reference:

```
            MAC             B
```

will produce the legal expansion:

```
            INR             B
```

but the reference:

```
            MAC             A
```

will produce the illegal expansion:

```
            INR             A
```

which will be flagged as an error.

There is one exception to this rule. Normally, a symbol may appear in the label field of only one instruction. If a label appears in the body of a macro, however, it will be generated whenever the macro is referenced. (See Section 4.0). To avoid multiple label conflicts, the assembler treats labels within macros as local labels, applying only to a particular expansion of a macro. Thus each "jump to LOOP" instruction generated in the Section 4.0 example refers uniquely to the label LOOP: generated in the local macros expansion.

4.2.4 PARAMETER SUBSTITUTION

The operand field of the MACRO pseudo instruction specifies which character strings in the macro body (the dummy parameters) are to be replaced by parameters listed in the operand field of the macro calls. For this substitution to occur, the strings in the macro body must be surrounded by seperators (comma, blank, colon, etc.), and must exactly match the dummy parameters. Substitution will never be made for a portion of a symbol.

For example, consider the macro definition:

```
MAC1        MACRO       REG, AMT
            ADI         AMT       }
            JNC         REG1      }  macro body
ONE:        MOV         REG,A     }
REG1:       XRA         A         }
            ENDM
```

Although the characters REG appear three times within the macro body, the only place parameter substitution for REG will occur is at line ONE:, since this is the only place REG is not part of a larger symbol.

Thus, the macro reference:

```
            MAC1        D, 6
```

will produce the expansion:

```
                ADI         6
                JNC         REG1
ONE:            MOV         D,A
REG1:           XRA         A
```

The programmer must be careful to choose dummy parameters which do not duplicate operation codes, labels, or symbol names used within the macro body, to avoid unwanted substitution.

For example, suppose a macro has one parameter which specifies an accumulator increment, and the programmer (unwisely) calls it INR. This could easily cause trouble, as follows:

Given the macro definition,

```
MAC2            MACRO       INR
                ADD         INR
                JNC         OUT
                INR         H
OUT:
                ENDM
```

the macro reference:

```
                MAC2        6
```

will cause the assembler to produce the invalid expansion:

```
                ADD         6
                JNC         OUT
                6           H   <------- Illegal instruction
OUT:
```

When a parameter specified by a macro reference is an expression, it is evaluated just before the macro expansion is produced. This allows identical macro calls to produce different results.

For example, suppose the following macro is defined at the beginning of a program:

```
MAC3        MACRO       REG, AMT
            MVI         REG, AMT
COUNT       SET         COUNT + 1
            ENDM
```

Further suppose that the statement:

```
COUNT       SET         0
```

has been written before the first reference to MAC3 setting the value of COUNT to zero.

Then the first macro reference:

```
            MAC3        D, COUNT * 2
```

will cause the assembler to evaluate COUNT * 2, and to substitute a value of zero for the dummy parameter AMT.

Expansion produced:

```
            MVI         D, 0
COUNT:      SET         COUNT + 1
```

The second statement of the expansion increases the value of COUNT to one. If the macro reference:

```
            MAC3        D, COUNT * 2
```

appears a second time in the program, COUNT * 2 will again be evaluated, producing the expansion:

```
            MVI         D, 2
COUNT:      SET         COUNT + 1
```

a third reference

```
            MAC         D, COUNT * 2
```

will produce the expansion

```
            MVI         D, 6
COUNT:      SET         COUNT + 1
```

The value of macro parameters is determined and passed into the macro body at the time of the macro reference, before the expansion is produced. This evaluation may be delayed by enclosing a parameter in quotes, causing the actual character string to be passed into the macro body. The string will then be evaluated when the macro expansion is produced.

Example:

Suppose that the following macro MAC4 is defined at the beginning of the program:

```
MAC4        MACRO       P1
ABC         SET         14
            DB          P1
            ENDM
```

Further suppose that the statement:

```
ABC         SET         3
```

has been written before the first reference to MAC4, setting the value of ABC to 3.

Then the macro reference:

```
            MAC4        ABC
```

will cause the assembler to evaluate ABC and to substitute the value 3 for parameter P1, then produce the expansion:

```
ABC         SET         14
            DB           3
```

If, however, the user had <u>instead</u> written the macro reference:

```
            MAC4        'ABC'
```

the assembler would evaluate the expression 'ABC', producing the characters ABC as the value of parameter P1. Then the expansion is produced, and, since ABC is altered by the first statement of the expansion, P1 will now produce the value 14.

Expansion produced:

```
ABC        SET     14
           DB      ABC ; Assembles as 14
```

4.3 REASONS FOR USING MACROS

The use of macros is an important programming technique that can substantially ease the user's task in the following ways:

(a) Often, a small group of instructions must be repeated many times throughout a program with only minor changes for each repetition.

For example, the load H and load L instructions must be used every time an arbitrary memory location is referenced. Macros can reduce the tedium (and resultant increased chance for error) associated with these operations.

(b) If an error in a macro definition is discovered, the program can be corrected by changing the definition and reassembling. If the same routine had been repeated many times throughout the program without using macros, each occurrence would have to be located and changed. Thus debugging time is decreased.

(c) Duplication of effort between programmers can be reduced. Once the most efficient coding of a particular function is discovered, the macro definition can be made available to all other programmers.

(d) As has been seen with the SHRT (shift right) macro, new and useful instructions can be easily simulated.

4.4 USEFUL MACROS

4.4.1 LOAD ADDRESS MACRO

The following macro, LXI, loads two adjacent registers (B and C, D and E, or H and L) with the high-order and low-order bytes, respectively, of a sixteen bit data quantity. The primary purpose of this macro is to load as memory address into the H and L registers.

<u>This operation is performed so frequently that the definition of LXI is built into the assembler. Thus, the programmer may write LXI in the code field of a statement without previously defining it.</u> This is the <u>only</u> macro which is built into the assembler.

Macro definition:

Label	Code	Operand
LXI	MACRO	REG, ADDR
	MVI	REG, ADDR SHR 8
	MVI	REG + 1, ADDR AND 0FFH
	ENDM	

Macro reference:

	LXI	H, DATA 1

Macro expansion:

	MVI	H, DATA 1 SHR 8
	MVI	H+1, DATA 1 AND 0FFH

If H is equated to 5, H + 1 will be assembled as 6, indicating the L register.

The following macros are useful examples which are not built into the assembler. Therefore, they must be defined in any program which uses them.

4.4.2 LOAD INDIRECT MACRO (WITHOUT SUBROUTINES)

The following macro, LIND, loads register RI indirect from memory location INADD. Register RJ is used to hold intermediate address information.

Macro definition:

```
Label     Code      Operand              Comment
LIND      MACRO     RI, INADD, RJ
          LXI       H, INADD         ;  Load the indirect address
          MOV       RI, M            ;  Load the high-order byte
          INR       L                ;  Point to low-order byte
          JNZ       LINN             ;  Bypass H.0. increment if non-zero
          INR       H                ;  result

LINN:     MOV       RJ, M            ; Load  L. 0. byte of memory address
          MOV       H, RI
          MOV       L, RJ
          MOV       RI, M            ;  Load RI indirect
          ENDM
```

Macro reference:

```
; Load register C indirect with the contents of memory location
; LABEL.  Use register D as a scratch register.

           LIND        C, LABEL, D
```

Macro expansion:

```
           MVI         H, LABEL SHR 8
           MVI         L, LABEL AND 0FFH
           MOV         C, M
           INR         L
           JNZ         LINN
           INR         H
LINN:      MOV         D, M
           MOV         H, C
           MOV         L, D
           MOV         C, M
```

4.4.3 MEMORY INCREMENT SUBROUTINE AND LOAD INDIRECT MACRO (WITH SUBROUTINE)

The programming concept of subroutines is described in Section 2, and a number of examples are provided in Section 5. However, the memory increment subroutine is introduced here to show how there is frequently a trade off between the use of macros and the use of subroutines.

While macros are useful programming aids, they do not necessarily economize memory use. Thus in the macro of Section 4.4.2, the five byte instruction sequence:

```
           INR         L
           JNZ         LINN
           INR         H
```

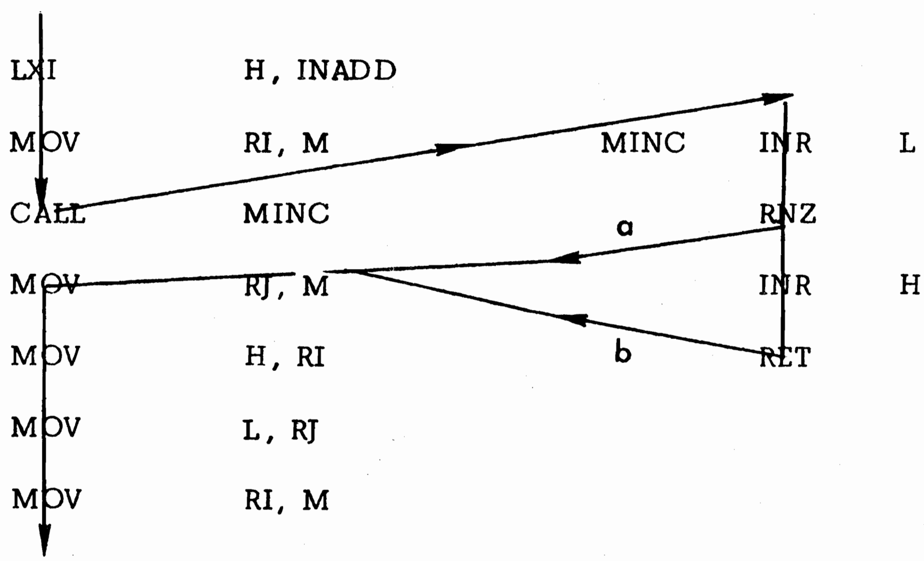
will be coded every time the Load Indirect macro is called, or any time an increment memory operation is required. Memory increment is such a common operation that it is more economically programmed as a subroutine that will occur only once in memory, and will be called when needed. The memory increment subroutine is:

```
MINC      INR     L        ; Increment low-order address byte
          RNZ              ; Return from subroutine if no carry
          INR     H        ; Increment high-order address byte
          RET              ; Return from subroutine unconditional
```

A load indirect macro using the memory increment subroutine may be defined as follows:

```
Label     Code      Operand              Comment
; Load register RI indirect from memory location
; INADD     Register  RJ is used to hold intermediate data
LINS      MACRO     RI, INADD, RJ
          LXI       H, INADD        ; Load the indirect address
          MOV       RI, M           ; Load the high-order direct address byte
          CALL      MINC            ; Increment the memory address
          MOV       RJ, M           ; Load the low-order direct address byte
          MOV       H, RI           ; Transfer direct address to
          MOV       L, RJ           ; H and L registers
          MOV       RI, M           ; Load desired value
          ENDM
```

When macro LINS is executed, the sequence is as follows:



(a) return if low order byte incremented only
(b) return if low order byte increment is from 0FFH to 00H, so high order byte is also incremented

The macro LINS and the subroutine MINC may each reside anywhere in memory. Note that the CALL MINC instruction uses three bytes and the MINC subroutine uses four bytes. If the LINS macro occurs just once, the increment portion will require 3 + 4 = 7 bytes versus the 5 bytes of macro LIND. If the LINS macro occurs twice, the increment portion will require 2 x 3 + 4 = 10 bytes, versus 2 x 5 = 10 for macro LIND. If the LINS macro occurs ten times, the increment portion will require 10 x 3 + 4 = 34 bytes versus 10 x 5 = 50 bytes for macro LIND. Clearly a considerable memory saving results when the macro is frequently used.

The single penalty incurred by using subroutines is that normal programming techniques only allow subroutines to be called to a depth of 7. Most users of the 8008 will not be hindered by this limitation, and use of macro LINS is recommended over macro LIND.

4.4.4 OTHER INDIRECT ADDRESSING MACROS

Refer to the LINS macro definition of Section 4.4.4. Only one instruction in this macro, the last MOV RI, M instruction, need be altered to create any other indirect addressing macro. For example, substituting MOV M, RI will create a "store indirect" macro. Providing RI is the accumulator, substituting ADD M will create an "add to accumulator indirect" macro.

As an alternative to having Load indirect, store indirect, and other such indirect macros, we could have a create indirect address macro, followed by selected instructions. This alternative approach is illustrated for indexed addressing in Section 4.4.5.

4.4.5 CREATE INDEXED ADDRESS MACRO

The following macro, IXAD, loads the address registers (H and L) with the base address BSADD, plus the 16 bit index formed by register RJ (high order byte) and RK (low order byte).

Macro definition:

Label	Code	Operand	Comment
IXAD	MACRO	RJ, BSADD, RK	
	LXI	H, BSADD	; Load the base address
	MOV	A, L	; Move L.O. byte to accumulator
	ADD	RK	; Add the L.O. index byte
	MOV	L, A	; Return sum to L
	MOV	A, H	; Move H.O. byte to accumulator
	ADC	RJ	; Add the H.O. byte of index with carry
	MOV	H, A	; Return H.O. address byte to H
	ENDM		

Macro reference:

```
; The address created in H and L by the following macro
; call will be Label +012EH

        MVI     D, 1
        MVI     E, 2EH
        IXAD    D, LABEL, E
```

Macro expansion:

```
        MVI     D, 1
        MVI     E, 2EH
        MVI     H, BSADD  SHR 8
        MVI     H + 1, BSADD  AND  0FFH
        MOV     A, L
        ADD     E
        MOV     L, A
        MOV     A, H
        ADC     D
        MOV     H, A
```

Consider now a program to successively load data bytes from a table origined at TBLE,incrementing a counter every time a negative value (high order bit = 1) is encountered. This program is simply implemented as illustrated below. We will assume that the table is terminated by a byte holding 0FFH, which acts as an end of table marker.

```
Label   Code    Operand         Comment
        XRA     A             ; Accumulator = 0
        MOV     B, A          ; Zero  B and C registers to
        MOV     C, A          ;       use as the index
        MOV     D, A          ; Zero D register as counter
LOOP:   IXAD    B, TBLE, C    ; Compute indexed address
        MOV     A,M           ; Load next data byte
        ADI     0             ; Add zero to set condition bits
        JP      LPI           ; Bypass increment if positive
        INR     D             ; Increment D if negative
LPI:    CPI     0FFH          ; Test for end
        JNZ     LOOP          ; Return to loop if not zero
        DCR     D             ; At end, decrement D for end byte
        HLT                   ; End
```

5.0 PROGRAMMING TECHNIQUES

This section describes some techniques other than macros which may be of help to the programmer.

5.1 BRANCH TABLES PSEUDOSUBROUTINE

Suppose a program consists of several separate routines, any of which may be executed depending upon some initial condition (such as a number passed in a register). One way to code this would be to check each condition sequentially and branch to the routines accordingly as follows:

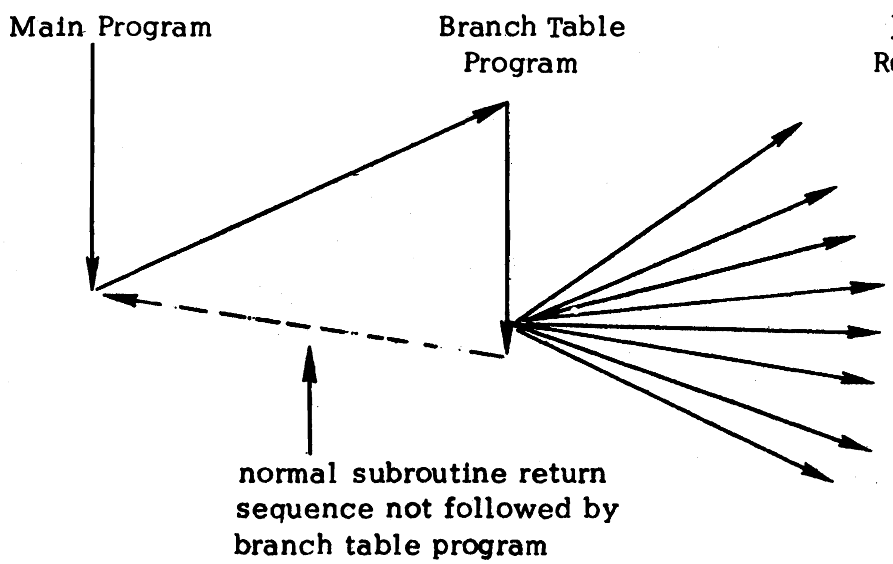
```
CONDITION = CONDITION 1?
IF YES BRANCH TO ROUTINE 1
CONDITION = CONDITION 2?
IF YES BRANCH TO ROUTINE 2
          .
          .
          .
BRANCH TO CONDITION N
```

A sequence as above is inefficient, and can be improved by using a branch table.

The logic at the beginning of the branch table program computes an index into the branch table. The branch table itself consists of a list of starting addresses for the routines to be branched to. Using the table index, the branch table program loads the selected routine's starting address into the address bytes of a jump instruction, then executes the jump. For example, consider a program that executes one of eight routines depending on which bit of the accumulator is set:

Jump	to	routine	1	if	accumulator	holds	00000001
"	"	"	2	"	"	"	00000010
"	"	"	3	"	"	"	00000100
"	"	"	4	"	"	"	00001000
"	"	"	5	"	"	"	00010000
"	"	"	6	"	"	"	00100000
"	"	"	7	"	"	"	01000000
"	"	"	8	"	"	"	10000000

A program that provides the above logic is given at the end of this section. The program is termed a "pseudosubroutine" because it is treated as a subroutine by the programmer, (i.e. it appears just once in memory), but it is entered via a regular JUMP instruction rather than via a CALL instruction. This is possible because the branch routine controls subsequent execution, and will never return to the instruction following the call:



```
Label       Code        Operand
START:      MVI         E, 0          ;  E will hold branch table index
GTBIT:      RAR
            JC          GETAD         ;  A one bit was found; form address
            INR         E             ;  E=E+2 to point to next address
            INR         E             ;        in branch table
            JMP         GTBIT
GETAD:      MVI         D, 0
            IXAD        BTBL, D, E    ;  H and L address BTBL+index
                                      ;  ( see Section 4.4 )
            MOV         A, M          ;  Get first byte of address
            LXI         H, JUMP+1
            MOV         M, A          ;  Store in jump instruction
            INR         E
            IXAD        BTBL, D,E
            MOV         A, M          ;  Get second byte of address
            LXI         H, JUMP+2
            MOV         M, A          ;  Store in jump instruction
            JMP         JUMP
                  -
                  -
JUMP:       JMP         0             ;  Dummy jump instruction
                  -
                  -
BTBL:       DW          ROUT1         ;  Branch table.  Each entry
            DW          ROUT2         ;  is a two byte address
            DW          ROUT3
            DW          ROUT4
            DW          ROUT5
            DW          ROUT6
            DW          ROUT7
            DW          ROUT8
```

The control routine at START computes an index into the branch table (BTBL:) corresponding to the bit of the accumulator that is set. It then transfers the address held in the corresponding branch table entry to the second and third bytes of the jump instruction (at JUMP:) and executes the jump instruction, thus transferring control to the selected routine.

CAUTION: The location JUMP: must appear in read/write memory in order for this routine to work correctly. If JUMP: is located in read-only memory, it is impossible to store the address bytes into the jump instruction.

5.2 SOFTWARE MULTIPLY AND DIVIDE

The multiplication of two unsigned 8 - bit data bytes may be accomplished by one of two techniques; repetitive addition, or use of a register shifting operation.

Repetitive addition provides the simplest, but slowest form of multiplication. For example, 2AH * 74H may be generated by adding 74H to the (initially zeroed) accumulator 2AH times.

Using shift operations provides faster multiplication. Shifting a byte left one bit is equivalent to multiplying by 2, and shifting a byte right one bit is equivalent to dividing by 2. The following process will produce the correct 2 byte result of multiplying a one byte multiplicand by a one byte multiplier:

(a) Test the least significant bit of the multiplier. If zero, go to step b. If one, add the multiplicand to the <u>most</u> significant byte of the result.

(b) Shift the entire two byte result right one bit position.

(c) Repeat steps a and b until all 8 bits of the multiplier have been tested.

For example, consider the multiplication:

2AH * 3CH = 9D8H

	MULTIPLIER	MULTIPLICAND	HIGH ORDER BYTE OF RESULT	LOW ORDER BYTE OF RESULT
Start	00111100	00101010	00000000	00000000
Step 1 a	----------------	----------------		
b			00000000	00000000
Step 2 a	----------------	----------------		
b			00000000	00000000
Step 3 a	----------------	----------------	00101010	00000000
b			00010101	00000000
Step 4 a	----------------	----------------	00111111	00000000
b			00011111	10000000
Step 5 a	----------------	----------------	01001001	10000000
b			00100100	11000000
Step 6 a	----------------	----------------	01001110	11000000
b			00100111	01100000
Step 7 a	----------------	----------------		
b			00010011	10110000
Step 8 a	----------------	----------------		
b			00001001	11011000

Step 1 : Test multiplier 0-bit; it is 0, so shift 16 bit result right one bit

Step 2 : Test multiplier 1-bit; it is 0, so shift 16 bit result right one bit.

Step 3 : Test multiplier 2-bit; it is 1, so add 2AH to high order byte of result and shift 16 bit result right one bit.

Step 4 : Test multiplier 3-bit; it is 1, so add 2AH to high order byte of result and shift 16 bit result right one bit.

Step 5 : Test multiplier 4-bit; it is 1, so add 2AH to high order byte of result and shift 16 bit result right one bit.

Step 6 : Test multiplier 5-bit; it is 1, so add 2AH to high order byte of result and shift 16 bit result right one bit.

Step 7 : Test multiplier 6-bit; it is 0, so shift 16-bit result right one bit.

Step 8 : Test multiplier 7-bit; it is 0, so shift 16-bit result right one bit.

The result produced is 09D8.

The process works for the following reason:

The result of any multiplication may be written:

Equation 1: $BIT7*MCND*2^{7} + BIT6*MCND*2^{6} + ... + BIT0*MCND*2^{0}$

where BIT0 through BIT8 are the bits of the multiplier (each equal to zero or one), and MCND is the multiplicand.

For example:

MULTIPLICAND		MULTIPLIER	
00001010	*	00000101	=

$0*0AH*2^{7} + 0*0AH*2^{6} + 0*0AH*2^{5} + 0*0AH*2^{4} +$

$0*0AH*2^{3} + 1*0AH*2^{2} + 0*0AH*2^{1} + 1*0AH*2^{0} =$

$00101000 + 00001010 = 00110010 = 50_{10}$

Adding the multiplicand to the high order byte of the result is the same as adding $MCND*2^{8}$ to the full 16-bit result; shifting the 16-bit result one position to the right is equivalent to multiplying the result by 2^{-1} (dividing by 2).

Therefore, step one above produces:

$$(BIT0 * MCND * 2^{8}) * 2^{-1}$$

Step two produces:

$$((BIT0 * MCND * 2^{8}) * 2^{-1} + (BIT1 * MCND*2^{8}))*2^{-1}$$
$$= BIT0*MCND*2^{6} + BIT1 * MCND*2^{7}$$

And so on, until step eight produces:

$$BIT0 * MCND * 2^{0} + BIT1 *MCND*2^{1} + ... + BIT7 *MCND*2^{7}$$

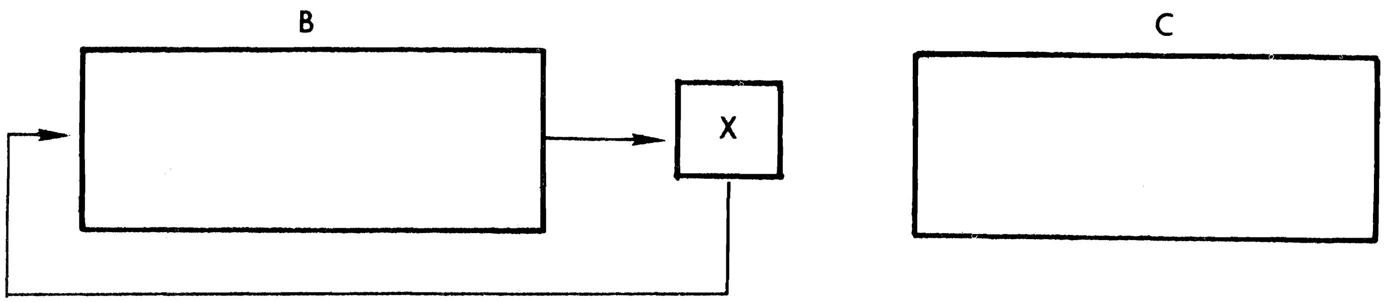
which is equivalent to Equation 1 above, and therefore is the correct result.

Since the multiplication routine described above uses a number of important programming techniques, a sample program is given with comments.

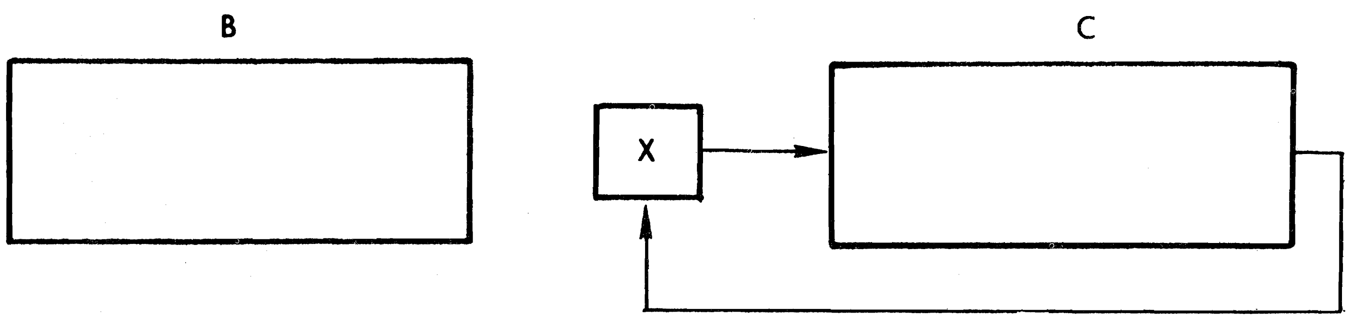
The program uses the B register to hold the most significant byte of the result, and the C register to hold the least significant byte of the result.

The 16 bit right shift of the result is performed by two rotate-right-through-carry instructions:

Zero carry and then rotate B



Then rotate C to complete the shift



Register D holds the multiplicand, and register C originally holds the multiplier.

```
MULT:     MVI    B, 0       ; Initialize most significant byte
                            ; of result
          MVI    E, 9       ; Bit counter
MULT0:    MOV    A, C       ; Rotate least significant bit of
          RAR               ; multiplier to carry and shift
          MOV    C, A       ; low order byte of result.
          DCR    E
          JZ     DONE       ; Exit if complete
          MOV    A, B
          JNC    MULT1
          ADA    D          ; Add multiplicand to high-order byte
                            ; of result if bit was a one.
MULT1:    RAR               ; Carry =0 here; shift high-order
                            ; byte of result
          MOV    B, A
          JMP    MULT
```

An analagous procedure is used to divide an unsigned 16 bit number by an unsigned 8 bit number. Here, the process involves subtraction rather than addition, and rotate-left instructions instead of rotate-right instructions.

The program uses the B and C registers to hold the most and least significant byte of the dividend respectively, and the D register to hold the divisor. The 8 bit quotient is generated in the C register, and the remainder is generated in the B register.

```
DIV:      MVI    E, 9       ; Bit counter
          MOV    A, B
DIV0:     MOV    B, A
          MOV    A, C       ; Rotate carry into C register; rotate
          RAL               ; next most significant bit to carry
          MOV    C, A
          DCR    E
          JZ     DIV1
          MOV    A, B       ; Rotate most significant bit to
          RAL               ; high-order quotient
          SUB    D          ; Subtract division. If less than
          JNC    DIV0       ; high-order quotient, go to DIV0
          ADD    D          ; Otherwise add it back
          JMP    DIV0
DIV1:     RAL
          MOV    E, A
          MVI    A, 0FFH    ; Complement the quotient
```

```
         XRA      C
         MOV      C, A
         MOV      A, E
         RAR
DONE:
```

5.3 MULTIBYTE ADDITION AND SUBTRACTION

The carry bit and the ADC (add with carry) instructions may be used to add unsigned data quantities of arbitrary length. Consider the following addition of two three-byte unsigned hexadecimal numbers:

```
   32AF8A
 + 84BA90
   B76A1A
```

This addition may be performed on the 8008 by adding the two low-order bytes of the numbers, then adding the resulting carry to the two next higher order bytes, and so on:

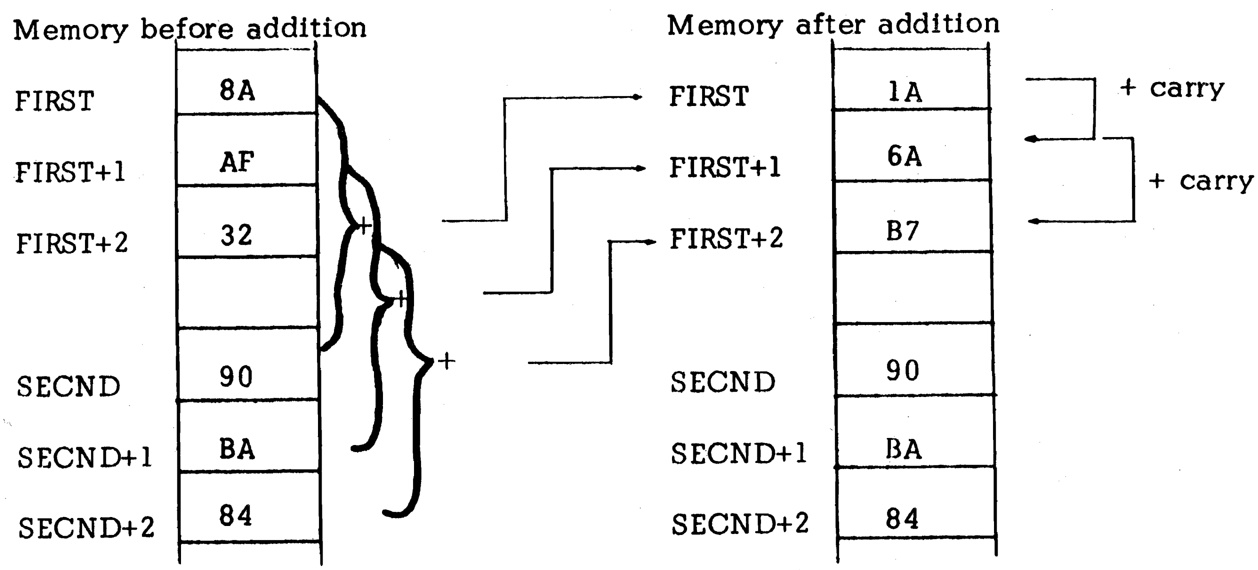
```
 32          AF          8A
 84          BA          90
 B7          6A          1A
     carry=1     carry=1
```

The following routine will perform this multibyte addition, making these assumptions:

The C register holds the length of each number to be added (in this case, 3).

The numbers to be added are stored from low-order byte to high-order byte beginning at memory locations FIRST and SECND, respectively.

The result will be stored from low-order byte to high-order byte beginning at memory location FIRST, replacing the original contents of this number.



```
Label       Code        Operand                Comment

IXAD        MACRO       INXRG, ADR
            MVI         H, ADR SHR 8
            MVI         L, ADR AND 0FFH
            MOV         A, L
            ADD         INXRG
            MOV         L, A
            JNC         OUT
            INR         H
OUT:
            ENDM

MADD:       XRA         A
            MOV         D, A          ; Index register D = 0
            MOV         E, A          ; Low order bit of E holds state of
                                      ; carry ( initially zero )
LOOP:       IXAD        D, SECND      ; H and L = address of next byte of SECND
            MOV         B, M          ; B = next byte of SECND
            IXAD        D, FIRST      ; H and L = address of next byte of FIRST
            MOV         A, E          ; Restore state of carry bit
            RAR                       ;      ( low order bit of E )
            MOV         A, M          ; A = next byte of FIRST
            ADC         B             ; Result of addition in accumulator
            MOV         M, A          ; Store result in current byte of FIRST
            RAL                       ; Save state of carry bit in low
            MOV         E, A          ;   order bit of E
            DCR         C             ; Done if C = 0
            JZ          DONE
            INR         D             ; Index = index +1; point to next bytes
            JMP         LOOP          ; Add next bytes
DONE:       —
            —
FIRST:      DB          90H
            DB          0BAH
            DB          84H
SECND:      DB          8AH
            DB          0AFH
            DB          32H
```

When location DONE is reached, bytes FIRST through FIRST + 2 will contain 1A6AB7, which is the sum shown at the beginning of this section arranged from low order to high order byte. (The reason multibyte numbers are usually stored in this fashion is that it is easier to add numbers from low to high order bytes, and it is easier to increment memory addresses than to decrement them).

The first time through the program loop, macro IXAD generates addresses SECND+0 and FIRST+0 in the H and L registers, enabling the program to access the two low-order bytes to be added. The carry produced by this addition is saved by rotating the carry bit into the low-order bit of the E register, since the carry could be altered before the next addition is performed. The result is stored at FIRST+0.

The second time through the loop, register D contains 1 (the number one), causing IXAD to generate generate addresses SECND+1 and FIRST+1. Thus, the second bytes to be added are accessed, summed together with the carry from the previous addition, and placed in FIRST+1.

This process is repeated until the C register is decremented to zero.

The carry (or borrow) bit and the SBB (subtract with borrow) instruction may be used to subtract unsigned data quantities of arbitrary length. Consider the following subtraction of two two-byte unsigned hexadecimal numbers:

```
  1301
 -0503
  ----
  0DFE
```

This subtraction may be performed on the 8008 by subtracting the two low-order bytes of the numbers, then using the resulting carry bit to adjust the difference of the two higher-order bytes if a borrow occurred (by using the SBB instruction).

Low order subtraction (carry bit=0 indicating no borrow):

```
   0 0 0 0 0 0 0 1  =  01H
   1 1 1 1 1 1 0 1  =  -(03H+carry)
   ---------------
   1 1 1 1 1 1 1 0  =  0FEH, the low order result
 0 overflow = 0, setting carry = 1 indicating a borrow
```

High order subtraction:

```
   0 0 0 1 0 0 1 1  =  13H
   1 1 1 1 1 0 1 0  =  -(05H+carry)
   ---------------
   0 0 0 0 1 1 0 1
 1 overflow = 1, resetting the carry bit indicating no borrow.
```

Whenever a borrow has occurred, the SBB instruction increments the subtrahend by one, which is equivalent to borrowing one from the minuend.

In order to create a multibyte subtraction routine, it is necessary only to duplicate the multibyte addition routine of this section, changing the ADC instruction to an SBB instruction. The program will then subtract the number beginning at SECND from the number beginning at FIRST, replacing the result at FIRST.

5.4 SUBROUTINES

Frequently, a group of instructions must be repeated many times in a program. As we have seen in Section 4, it is somtimes helpful to define a macro to produce these groups. If a macro becomes too lengthy or must be repeated many times, however, better economy can be obtained by using subroutines.

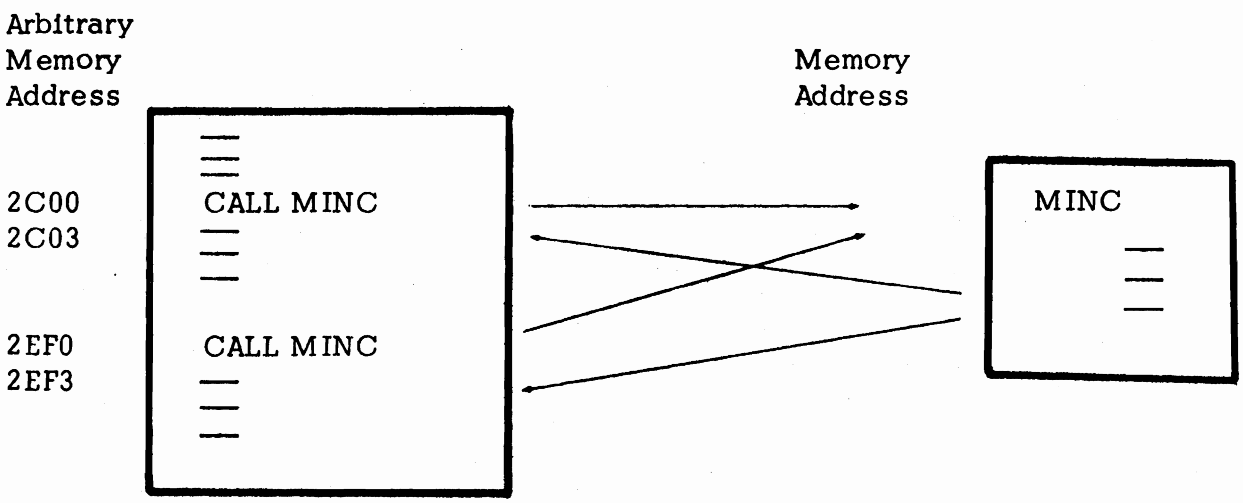
A subroutine is coded like any other group of assembly language statements, and is referred to by its name, which is the label of the first instruction. The programmer references a subroutine by writing its name in the operand field of a CALL instruction. When the CALL is executed, the address of the next sequential instruction after the CALL is "pushed" onto the address stack, (See Section 2.1.2),and program execution proceeds with the first instruction of the subroutine. When the subroutine has completed its work, a RETURN instruction is executed, which causes the top address in the stack to be "pulled" into the program counter, causing program execution to continue with the instruction following the CALL. Thus, one copy of a subroutine may be called from many different points in memory, preventing duplication of code.

Example:

Subroutine MINC increments a memory address passed in the H and L registers and then returns to the instruction following the last CALL statement executed.

```
MINC:     INR      L      ; Increment low order address byte
          RNZ             ; If not zero, return to calling routine
          INR      H      ; Increment high order address byte
          RET             ; Return unconditionally
```

Assume MINC appears in the following program:



When the first call is executed, address 2C03 is written to the address stack, and control is transferred to 3C00. Execution of either RETURN statement in MINC will cause the top entry to be read from the address stack and placed in the program counter, causing execution to continue at 2C03 (since the CALL statement is three bytes long).

Address stack before CALL	Stack while MINC is executing	Stack after RETURN is performed
ADR 1 ←	2C03H	2C03H ←
ADR 2	ADR 2 ←	ADR 2
ADR 3	ADR 3	ADR 3
ADR 4	ADR 4	ADR 4
ADR 5	ADR 5	ADR 5
ADR 6	ADR 6	ADR 6
ADR 7	ADR 7	ADR7

When the second call is executed, address 2EF3 is pushed onto the stack, and control is again transferred to MINC. This time, either RETURN instruction will cause execution to resume at 2EF3.

Note that MINC could have called another subroutine during its execution, causing another address to be pushed onto the stack. This can occur only up to seven levels, however, since the stack can only hold seven addresses. Beyond this point, the RETURN addresses will be lost and RETURN instructions will transfer program control to incorrect addresses.

5.5 TRANSFERRING DATA TO SUBROUTINES

A subroutine often requires data to perform its operations. In the simplest case, this data may be transferred in one or more registers. Subroutine MINC in Section 5.4 for example, receives the memory address upon which it operates in the H and L registers.

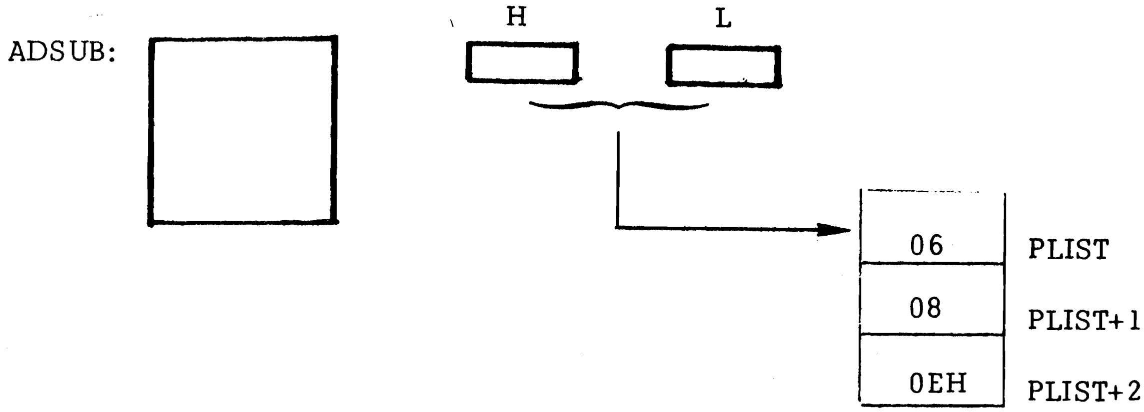
Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data, (called a parameter list), in some data area of memory, and pass the address of this list to the subroutine in the H and L registers.

For example, the subroutine ADSUB expects the address of a three byte parameter list in the H and L registers. It adds the first and second bytes of the list, and stores the result in the third byte of the list:

Label	Code	Operand	Comment
	LXI	H, PLIST	; Load H and L with addresses
			; of the parameter list
	CALL	ADSUB	; Call the subroutine
RET1:	—		
	—		
PLIST:	DB	6	; First number to be added
	DB	8	; Second number to be added
	DS	1	; Result will be stored here
	LXI	H, LIST2	; Load H and L registers for
	CALL	ADSUB	; another call to ADSUB
RET2:	—		
	—		
LIST2:	DB	10	
	DB	35	
	DS	1	
	—		
ADSUB:	MOV	A, M	; Get first parameter
	CALL	MINC	; Increment memory address
	MOV	B, M	; Get second parameter
	ADD	B	; Add first to second
	CALL	MINC	; Increment memory address
	MOV	M, A	; Store result at third parameter store
	RET		; Return unconditionally

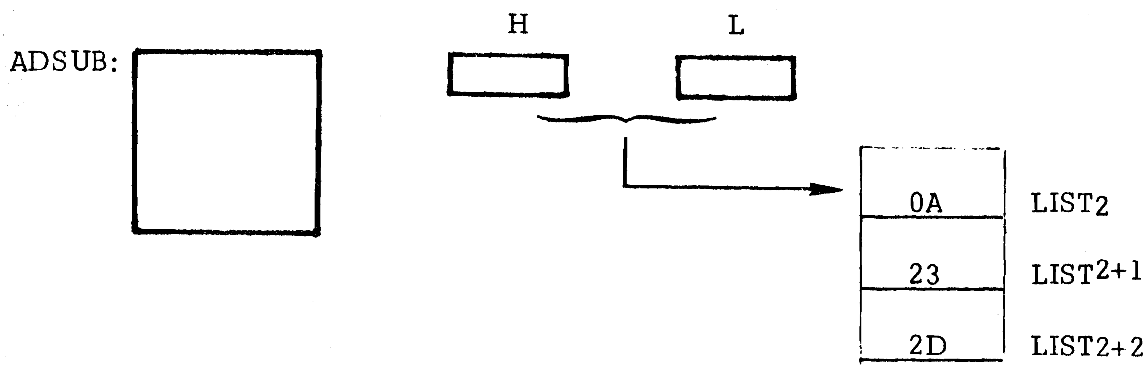
The first time ADSUB is called, it loads the A and B registers from PLIST and PLIST + 1 respectively, adds them and stores the result in PLIST + 2. Return is then made to the instruction at RET1:.

First call to ADSUB:



The second time ADSUB is called, the H and L registers point to the parameter list LIST2. The A and B registers are loaded with 10 and 35 respectively, and the sum is stored at LIST2+2. Return is then made to the instruction at RET1.

Second call to ADSUB:

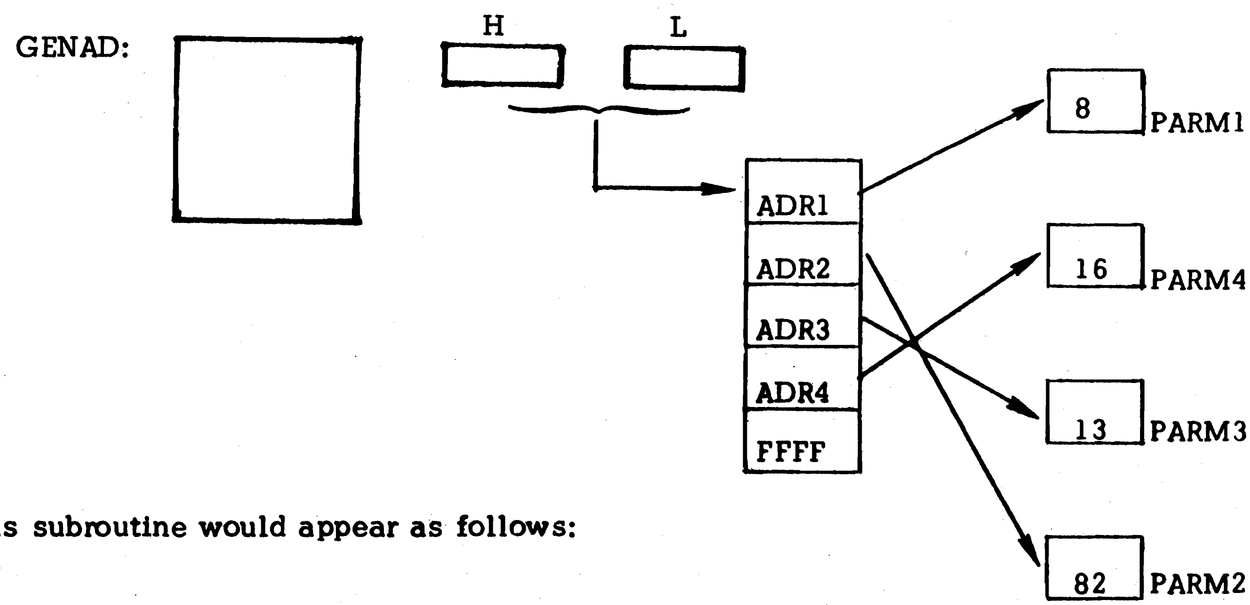


Note that the parameter lists PLIST and LIST2 could appear anywhere in memory without altering the results produced by ADSUB.

This approach does have its limitations, however. As coded, ADSUB must receive a list of two and only two numbers to be added, and they must be contiguous in memory. Suppose we wanted a subroutine (GENAD) which would add an arbitrary number of bytes, located anywhere in memory, and leave the sum in the accumulator.

This can be done by passing the subroutine a parameter list which is a list of addresses of parameters, rather than the parameters themselves, and signifying the end of the parameter list by a negative number:

Call to GENAD:



This subroutine would appear as follows:

```
Label     Code      Operand

          LXI       H, PLIST
          CALL      GENAD
          —
          —
PLIST     DW        PARM1
          DW        PARM2
          DW        PARM3
          DW        PARM4
          DW        0FFFFH

PARM1     DB        8
PARM4     DB        16
          —
PARM3     DB        13
          —
PARM2     DB        82
          —
          —
GENAD:    XRA       A         ; Clear accumulator
LOOP:     MOV       D, H      ; Save address of parameter list
          MOV       E, L      ;
          MOV       C, A      ; Save accumulator
          MOV       B, M      ; Get low order address byte of first
                              ; parameter
          CALL      MINC
          MOV       H, M      ; Get high order address byte of first
                              ; parameter
          MOV       A, H      ; Test high address byte for negative
          ORA       A         ; Set condition bits
          MOV       A, C      ; Restore accumulator - - does not affect
                              ; condition bits
          RM                  ; Return if last address was negative;
                              ;     accumulator holds sum
          MOV       L, B      ; H + L hold address of parameter
          ADD       M         ; Add parameter to accumulator
          MOV       H, D
          MOV       L, E
          CALL      MINC      ; Increment to point to second parameter
          CALI      MINC      ; Address ( PLIST + 2 )
          JMP       LOOP      ; Get next parameter
```

Note that GETAD could add any combination of the parameters with no change to the parameters themselves. The sequence:

```
        LXI     H, PLIST
        CALL    GENAD
PLIST:  ==      ==
        DW      PARM4
        DW      PARM1
        DW      0FFFFH
```

would cause PARM1 and PARM4 to be added, no matter where in memory they might be located.

6.0 INTERRUPTS

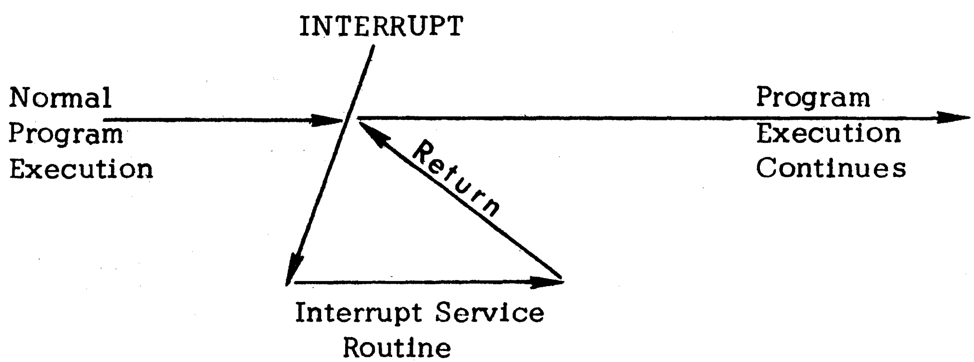
Often, events occur external to the central processing unit which require immediate action by the CPU. For example, suppose a device is sending/ receiving a string of 80 characters to/from the CPU, one at a time, at fixed intervals. There are two ways to handle such a situation:

(a) A program could be written which inputs/outputs the first character, stalls until the next character is ready (eg. executes a timeout by incrementing a sufficiently large counter), then inputs/outputs the next character, and proceeds in this fashion until the entire 80 character string has been received/transmitted.

This method is referred to as programmed Input/Output.

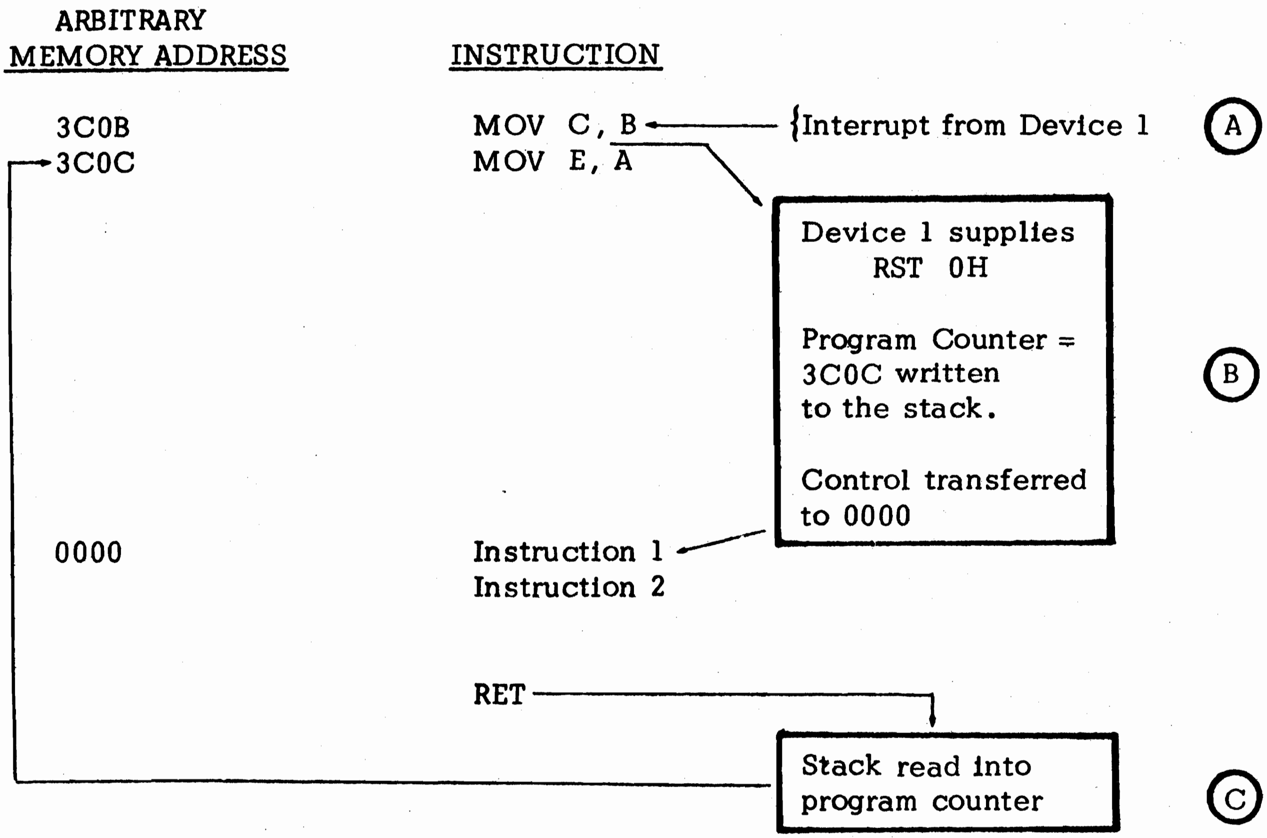
(b) The device controller could interrupt the CPU when a character is ready to be input, or the device is ready to receive a character, forcing a branch from the executing program to a special interrupt service routine.

The interrupt sequence may be illustrated as follows:



Any device may supply an RST instruction (and indeed may supply an INTELLEC 8 instruction).

The following is an example of an Interrupt sequence:



Device one signals an interrupt as the CPU is executing the instruction at 3C0B. This instruction is completed. The program counter remains set to 3C0C, and the instruction RST 0H supplied by device one is executed. Since this is a call to location zero, 3C0C is written to the address stack and this is a call to location zero, 0000H. (This subroutine may perform jumps, calls, or any other operation). When the RETURN is executed, address 3C0C is read from the stack and replaces the contents of the program counter, causing execution to continue at the instruction following the point where the interrupt occurred.

Note that an interrupting device may specify an instruction. For instance, if HLT is specified, the only action taken by the CPU is to complete the current instruction and then stop. The CPU will remain stopped until another interrupt

When the CPU recognizes an interrupt request from an external device, the following actions occur:

1) The instruction currently being executed is completed.

2) The interrupting device supplies, via hardware, one instruction which the CPU executes. This instruction does not appear anywhere in memory, and the programmer has no control over it, since it is a function of the interrupting device's controller design. The program counter is not incremented before this instruction.

The instruction supplied by the interrupting device is normally an RST instruction, (see Section 3.11), since this is an efficient one byte call to one of 8 eight-byte subroutines located in the first 64 words of memory. For instance, the teletype may supply the instruction

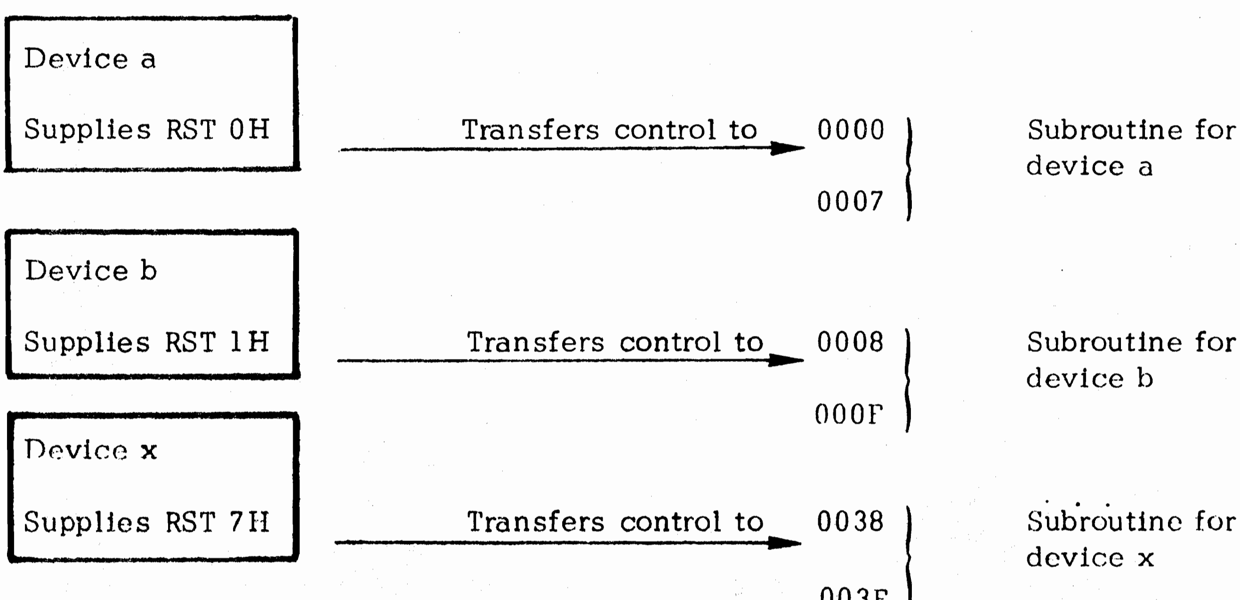
RST 0H

with each teletype input interrupt. Then the subroutine which processes data transmitted from the teletype to the CPU will be called into execution via an eight byte instruction sequence at memory locations 0000H to 0007H.

A digital input device may supply the instruction:

RST 1H

Then the subroutine that processes the digital input signals will be called via a sequence of instructions occupying memory locations 0008H to 000FH.



occurs.

Example:

Assume that there are eight recorders transmitting data to the CPU. The recorders have device numbers 6 through D, plus a common device number E, via which an identifying signal can be sent. The controller for each of the eight recorders requests a program interrupt when data is ready to be transmitted to the CPU. When the CPU acknowledges the interrupt, the controller supplies the instruction:

```
RST     3
```

and transmits to device address 0EH the data byte:

```
00000001B  for device 1
00000010B  for device 2
00000100B  for device 3
00001000B  for device 4
00010000B  for device 5
00100000B  for device 6
01000000B  for device 7
10000000B  for device 8
```

Everything described so far is a function of hardware design, and while the programmer must know about it, he cannot change it in any way.

When any one of the eight recorders causes an interrupt, a jump to memory location 0010H is forced. At this location the following five byte routine is located:

```
        IN    0EH   ; read the identifying data byte from device 0EH
        JMP   START ; jump to the branch table pseudosubroutine
BACK:   RET         ; all service routines return here
```

Pseudosubroutine START is described in Section 5.1.

Thus eight devices have been serviced via one interrupt service routine.

Note that, if the interrupted program was using the accumulator, erroneous results could occur when a RETURN was made. This problem can be avoided by requiring the interrupt routines to save the accumulator in memory, and restore it before returning to the interrupted routine.

APPENDIX "A"

- - INSTRUCTION SUMMARY- -

This appendix provides a summary of INTELLEC 8 assembly language instructions. Abbreviations used are as follows:

A	The accumulator (register A)
A_n	Bit n of the accumulator contents, where n may have any value from 0 to 7.
ADDR	Any memory address
Carry	The carry bit
CODE	An operation code
DATA	Any byte of data
DST	Destination register or memory byte
EXP	A constant or mathematical expression
LABEL:	Any instruction label
M	A memory byte
Parity	The parity bit
PC	Program Counter
REGM	Any register or memory byte
sign	The sign bit
SRC	Source register or memory byte
STK	Top stack register
zero	The zero bit

[ ] An optional field enclosed by brackets

( ) Contents of register or memory byte enclosed by brackets

$\leftarrow$ Replace left hand side with right hand side of arrow

A.1 SINGLE REGISTER INSTRUCTIONS

Format:

[LABEL:] CODE REGM

Note: REGM ≠ A or M

Code	Description	
INR	(REGM) $\leftarrow$ (REGM) +1	Increment register REGM
DCR	(REGM) $\leftarrow$ (REGM) -1	Decrement register REGM

Condition bits affected: Zero, sign, parity

A.2 MOV INSTRUCTIONS

Format:

[LABEL:] MOV DST,SRC

Note SRC and DST not both =M

Code	DESCRIPTION	
MOV	(DST) $\leftarrow$ (SRC)	Load register DST from register SRC

Condition bits affected: None

A. 3 REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS

Format:

[LABEL:] CODE REGM

Code	DESCRIPTION	
ADD	(A) ← (A) + (REGM)	Add REGM to accumulator
ADC	(A) ← (A) + (REGM) + (carry)	Add REGM plus carry bit to accumulator
SUB	(A) ← (A) - (REGM)	Subtract REGM from accumulator
SBB	(A) ← (A) - (REGM) - (carry)	Subtract REGM minus carry
ANA	(A) ← (A) AND (REGM)	AND accumulator with REGM
XRA	(A) ← (A) XOR (REGM)	Exclusive-OR accumulator with REGM
ORA	(A) ← (A) OR (REGM)	OR accumulator with REGM
CMP	Condition bits set by (A) - (REGM)	Compare REGM with accumulator

Condition bits affected:

ADD, ADC, SUB, SBB : Carry, sign, zero, parity

ANA, XRA, DRA : Sign, zero, parity. Carry is zeroed.

CMP: Carry, sign, zero, parity.

Zero set if (A) = (REGM)
Carry reset if (A) < (REGM)
Carry set if (A) ≥ (REGM)

A.4 ROTATE ACCUMULATOR INSTRUCTIONS

Format:

[LABEL.] CODE REGM

CODE	DESCRIPTION	
RLC	(carry) ← A_7, A_n+1, ← A_n, A_0 ← A_7	Set carry = A_7, rotate accumulator left
RRC	(carry) ← A_0, A_n ← A_n+1, A_7 ← A_0	Set carry = A_0, rotate accumulator right
RAL	A_n+1 ← A_n, (carry) ← A_7, A_0 ← (carry)	Rotate accumulator right through the carry
RAR	A_n ← A_n+1, (carry) ← A_0, A_7 ← (carry)	Rotate accumulator left through the carry

Condition bits affected: Carry

A.5 IMMEDIATE INSTRUCTIONS

Format:

[LABEL:] MVI REGM, DATA

- or -

[LABEL:] CODE REGM

CODE	DESCRIPTION	
MVI	(REGM) ←—— DATA	Move immediate DATA into REGM
ADI	(A) ←—— (A) + DATA	Add immediate data to accumulator
ACI	(A) ←—— (A) + DATA+(carry)	Add immediate data + carry to accumulator
SUI	(A) ←—— (A) - DATA	Subtract immediate data from accumulator
SBI	(A) ←—— (A) - DATA - (carry)	Subtract immediate data and carry from accumulator
ANI	(A) ←—— (A) AND DATA	AND accumulator with immediate data
XRI	(A) ←—— (A) XOR DATA	Exclusive-OR accumulator with immediate data
ORI	(A) ←—— (A) OR DATA	OR accumulator with immediate data
CPI	Condition bits set by (A) - DATA	Compare immediate data with accumulator

Condition bits affected:

MVI: None
ADI, ACI, SUI, SBI : Carry, sign, zero, parity
ANI, XRI, ORI : Zero, sign, parity. Carry is zeroed.
CPI: Carry, sign, zero, parity

Zero set if (A)	=	DATA
Carry reset if (A)	<	DATA
Carry set if (A)	≥	DATA

A.6 JUMP INSTRUCTIONS

Format:

[LABEL:] CODE ADDR

CODE	DESCRIPTION	
JMP	(PC) ⟵ ADDR	Jump to location ADDR
JC	If (carry) =1, (PC) ⟵ ADDR	
	If (carry) =0, (PC) ⟵ (PC)+3	Jump to ADDR if carry set
JNC	If (carry) =0, (PC) ⟵ ADDR	
	If (carry) =1, (PC) ⟵ (PC)+3	Jump to ADDR if carry reset
JZ	If (zero) =1, (PC) ⟵ ADDR	
	If (zero) =0, (PC) ⟵ (PC)+3	Jump to ADDR of zero set
JNZ	If (zero) =0, (PC) ⟵ ADDR	
	If (zero) =1, (PC) ⟵ (PC)+3	Jump to ADDR if zero reset
JP	If (sign) =0, (PC) ⟵ ADDR	
	If (sign) =1, (PC) ⟵ (PC)+3	Jump to ADDR if plus
JM	If (sign) =1, (PC) ⟵ ADDR	
	If (sign) =0, (PC) ⟵ (PC)+3	Jump to ADDR if minus
JPE	If (parity) =1, (PC) ⟵ ADDR	
	If (parity) =0, (PC) ⟵ (PC)+3	Jump to ADDR if parity even
JPO	If (parity) =0, (PC) ⟵ ADDR	
	If (parity) =1, (PC) ⟵ (PC)+3	Jump to ADDR if parity odd

Condition bits affected: None

A.7 CALL INSTRUCTIONS

Format:

[LABEL:] CODE ADDR

CODE	DESCRIPTION	
CALL	(STK)←(PC), (PC)←ADDR	Call subroutine and push return address onto stack
CC	If (carry) =1, (STK)←(PC), (PC)←(ADDR)	
	If (carry) =0, (PC)←(PC)+3	Call subroutine if carry set
CNC	If (carry) =0, (STK)←(PC), (PC)←(ADDR)	
	If (carry) =1, (PC)←(PC)+3	Call subroutine if carry reset
CZ	If (zero) = 1, (STK)←(PC), (PC)←(ADDR)	
	If (zero) = 0, (PC)←(PC)+3	Call subroutine if zero set
CNZ	If (zero) = 0, (STK)←(PC), (PC)←(ADDR)	
	If (zero) = 1, (PC)←(PC)+3	Call subroutine if zero reset
CP	If (sign) = 0, (STK)←(PC), (PC)←(ADDR)	
	If (sign) = 1 (PC)←(PC)+3	Call subroutine if sign plus
CM	If (sign) = 1 (STK)←(PC), (PC)←(ADDR)	
	If (sign) = 0 (PC)←(PC)+3	Call subroutine if sign minus
CPE	If (parity)=1 (STK)←(PC), (PC)←(ADDR)	
	If (parity)=0 (PC)←(PC)+3	Call subroutine if parity even
CPO	If (parity)=0 (STK)←(PC), (PC)←(ADDR)	
	If (parity)=1 (PC)←(PC)+3	Call subroutine if parity odd

Condition bits affected: None

A.8 RETURN INSTRUCTIONS

Format:

[LABEL:]   CODE

CODE	DESCRIPTION	
RET	(PC) ←— STK	Return from subroutine
RC	If (carry)=1, (PC) ←— STK	
	If (carry)=0, (PC) ←— (PC)+3	Return if carry set
RNC	If (carry)=0, (PC) ←— STK	
	If (carry)=1, (PC) ←— (PC)+3	Return if carry reset
RZ	If (zero) =1, (PC) ←— STK	
	If (zero) =0, (PC) ←— (PC)+3	Return if zero set
RNZ	If (zero) =0, (PC) ←— STK	
	If (zero) =1, (PC) ←— (PC)+3	Return if zero reset
RM	If (sign)= 1, (PC) ←— STK	
	If (sign) =0, (PC) ←— (PC)+3	Return if minus
RP	If (sign) =0, (PC) ←— STK	
	If (sign) =1, (PC) ←— (PC)+3	Return if plus
RPE	If (parity)=1, (PC) ←— STK	
	If (parity)=0, (PC) ←— (PC)+3	Return if parity even
RPO	If (parity)=0, (PC) ←— STK	
	If (parity)=1, (PC) ←— (PC)+3	Return if parity odd

Condition bits affected: None

A.9 RST INSTRUCTION

Format:

[LABEL:] RST EXP

Note: $0 \leq EXP \leq 7$

CODE	DESCRIPTION	
RST	(STK) ← (PC) (PC) ← 00000000EXP000B	Call subroutine at address specified by EXP

Condition bits affected: None

A.10 INPUT/OUTPUT INSTRUCTIONS

Format:

[LABEL:] CODE EXP

Note: For IN, $0 \leq EXP \leq 7$
For OUT, $8 \leq EXP \leq 31$

CODE	DESCRIPTION	
IN	(A) ← input device	Read a byte from device EXP into the accumulator
OUT	output device ← (A)	Send the accumulator contents to device EXP

Condition bits affected: None

PSEUDO - INSTRUCTIONS

A.11 ORG PSEUDO - INSTRUCTION

Format:

ORG EXP

Code	Description	
ORG	LOCATION COUNTER ◄—— EXP	Set Assembler location counter to EXP

A.12 EQU PSEUDO- INSTRUCTION

Format:

LABEL EQU EXP

Code	Description	
EQU	LABEL ◄—— EXP	Assign the value EXP to the symbol LABEL.

A.13 SET PSEUDO - INSTRUCTION

Format:

LABEL SET EXP

Code	Description	
SET	LABEL ←——— EXP	Assign the value EXP to the symbol LABEL, which may have been previously SET.

A.14 END PSEUDO - INSTRUCTION

Format:

END

Code	Description
END	End the assembly.

A.15 CONDITIONAL ASSEMBLY PSEUDO - INSTRUCTIONS

Format:

IF EXP

-and-

ENDIF

Code	Description
IF	If EXP =0, ignore assembler statements until ENDIF is reached. Otherwise, continue assembling statements.
ENDIF	End range of preceding IF.

A.16 MACRO DEFINITION PSEUDO - INSTRUCTIONS

Format:

NAME MACRO LIST

-and-

ENDM

Code	Description
MACRO	Define a macro named NAME with parameters LIST
ENDM	End macro definition

APPENDIX "B"

- - INSTRUCTION MACHINE CODES - -

In order to help the programmer examine memory when debugging programs, this appendix provides the assembly language instruction represented by each of the 256 possible instruction code bytes.

Where an instruction occupies two bytes (immediate instruction) or three bytes (jump instruction), only the first (code) byte is given.

DEC	OCTAL	HEX	MNEMONIC	COMMENT
0	000	00	HLT	
1	001	01	-	
2	002	02	RLC	
3	003	03	RNC	
4	004	04	ADI EXP	
5	005	05	RST EXP	
6	006	06	MVI A, EXP	
7	007	07	RET	
8	010	08	INR B	
9	011	09	DCR B	
10	012	0A	RRC	
11	013	0B	RNZ	
12	014	0C	ACI EXP	
13	015	0D	RST EXP	EXP =1
14	016	0E	MVI B, EXP	
15	017	0F	RET	
16	020	10	INR C	
17	021	11	DCR C	
18	022	12	RAL	
19	023	13	RP	
20	024	14	SUI EXP	
21	025	15	RST EXP	EXP =2
22	026	16	MVI C, EXP	
23	027	17	RET	
24	030	18	INR D	
25	031	19	DCR D	
26	032	1A	RAR	
27	033	1B	RPO	
28	034	1C	SBI EXP	
29	035	1D	RST EXP	EXP =3
30	036	1E	MVI D, EXP	
31	037	1F	RET	
32	040	20	INR E	
33	041	21	DCR E	
34	042	22	-	
35	043	23	RC	
36	044	24	ANI EXP	
37	045	25	RST EXP	EXP =4
38	046	26	MVI E, EXP	
39	047	27	-	
40	050	28	INR H	

DEC	OCTAL	HEX	MNEMONIC	COMMENT
41	051	29	DCR H	
42	052	2A	-	
43	053	2B	RZ	
44	054	2C	XRI EXP	
45	055	2D	RST EXP	EXP =5
46	056	2E	MVI H, EXP	
47	057	2F	-	
48	060	30	INR L	
49	061	31	DCR L	
50	062	32	-	
51	063	33	RM	
52	064	34	ORI EXP	
53	065	35	RST EXP	EXP =6
54	066	36	MVI L, EXP	
55	067	37	RET	
56	070	38	-	
57	071	39	-	
58	072	3A	-	
59	073	3B	RPE	
60	074	3C	CPI EXP	
61	075	3D	RST EXP	EXP =7
62	076	3E	MVI M, EXP	
63	077	3F	RET	
64	100	40	JNC EXP	
65	101	41	IN EXP	EXP =0
66	102	42	CNC EXP	
67	103	43	IN EXP	EXP =1
68	104	44	JMP EXP	
69	105	45	IN EXP	EXP =2
70	106	46	CALL EXP	
71	107	47	IN EXP	EXP =3
72	110	48	JNZ EXP	
73	111	49	IN EXP	EXP =4
74	112	4A	CNZ EXP	
75	113	4B	IN EXP	EXP =5
76	114	4C	-	
77	115	4D	IN EXP	EXP =6
78	116	4E	-	
79	117	4F	IN EXP	EXP =7
80	120	50	JP EXP	
81	121	51	OUT EXP	EXP =8

DEC	OCTAL	HEX	MNEMONIC	COMMENT
82	122	52	CP EXP	
83	123	53	OUT EXP	EXP =9
84	124	54	-	
85	125	55	OUT EXP	EXP =10
86	126	56	-	
87	127	57	OUT EXP	EXP =11
88	130	58	JPO EXP	
89	131	59	OUT EXP	EXP =12
90	132	5A	CPO EXP	
91	133	5B	OUT EXP	EXP =13
92	134	5C	-	
93	135	5D	OUT EXP	EXP =14
94	136	5E	-	
95	137	5F	OUT EXP	EXP =15
96	140	60	JC EXP	
97	141	61	OUT EXP	EXP =16
98	142	62	CC EXP	
99	143	63	OUT EXP	EXP =17
100	144	64	-	
101	145	65	OUT EXP	EXP =18
102	146	66	-	
103	147	67	OUT EXP	EXP =19
104	150	68	JZ EXP	
105	151	69	OUT EXP	EXP =20
106	152	6A	CZ EXP	
107	153	6B	OUT EXP	EXP =21
108	154	6C	-	
109	155	6D	OUT EXP	EXP =22
110	156	6E	-	
111	157	6F	OUT EXP	EXP =23
112	160	70	JM EXP	
113	161	71	OUT EXP	EXP =24
114	162	72	CM EXP	
115	163	73	OUT EXP	EXP =25
116	164	74	-	
117	165	75	OUT EXP	EXP =26
118	166	76	-	
119	167	77	OUT EXP	EXP =27
120	170	78	JPE EXP	
121	171	79	OUT EXP	EXP =28
122	172	7A	CPE EXP	

DEC	OCTAL	HEX	MNEMONIC	COMMENT
123	173	7B	OUT EXP	EXP =29
124	174	7C	-	
125	175	7D	OUT EXP	EXP =30
126	176	7E	-	
127	177	7F	OUT EXP	EXP =31
128	200	80	ADD A	
129	201	81	ADD B	
130	202	82	ADD C	
131	203	83	ADD D	
132	204	84	ADD E	
133	205	85	ADD H	
134	206	86	ADD L	
135	207	87	ADD M	
136	210	88	ADC A	
137	211	89	ADC B	
138	212	8A	ADC C	
139	213	8B	ADC D	
140	214	8C	ADC E	
141	215	8D	ADC H	
142	216	8E	ADC L	
143	217	8F	ADC M	
144	220	90	SUB A	
145	221	91	SUB B	
146	222	92	SUB C	
147	223	93	SUB D	
148	224	94	SUB E	
149	225	95	SUB H	
150	226	96	SUB L	
151	227	97	SUB M	
152	230	98	SBB A	
153	231	99	SBB B	
154	232	9A	SBB C	
155	233	9B	SBB D	
156	234	9C	SBB E	
157	235	9D	SBB H	
158	236	9E	SBB L	
159	237	9F	SBB M	
160	240	A0	ANA A	
161	241	A1	ANA B	
162	242	A2	ANA C	
163	243	A3	ANA D	

DEC	OCTAL	HEX	MNEMONIC	COMMENT
164	244	A4	ANA E	
165	245	A5	ANA H	
166	246	A6	ANA L	
167	247	A7	ANA M	
168	250	A8	XRA A	
169	251	A9	XRA B	
170	252	AA	XRA C	
171	253	AB	XRA D	
172	254	AC	XRA E	
173	255	AD	XRA H	
174	256	AE	XRA L	
175	257	AF	XRA M	
176	260	B0	ORA A	
177	261	B1	ORA A	
178	262	B2	ORA C	
179	263	B3	ORA D	
180	264	B4	ORA E	
181	265	B5	ORA H	
182	266	B6	ORA L	
183	267	B7	ORA M	
184	270	B8	CMP A	
185	271	B9	CMP B	
186	272	BA	CMP C	
187	273	BB	CMP D	
188	274	BC	CMP E	
189	275	BD	CMP H	
190	276	BE	CMP L	
191	277	BF	CMP M	
192	300	C0	NOP	
193	301	C1	MOV A,B	
194	302	C2	MOV A,C	
195	303	C3	MOV A,D	
196	304	C4	MOV A,E	
197	305	C5	MOV A,H	
198	306	C6	MOV A,L	
199	307	C7	MOV A,M	
200	310	C8	MOV A,B	
201	311	C9	MOV B,B	
202	312	CA	MOV B,C	
203	313	CB	MOV B,D	
204	314	CC	MOV B,E	

DEC	OCTAL	HEX	MNEMONIC	COMMENT
205	315	CD	MOV B,H	
206	316	CE	MOV B,L	
207	317	CF	MOV B,M	
208	320	D0	MOV C,A	
209	321	D1	MOV C,B	
210	322	D2	MOV C,C	
211	323	D3	MOV C,D	
212	324	D4	MOV C,E	
213	325	D5	MOV C,H	
214	326	D6	MOV C,L	
215	327	D7	MOV C,M	
216	330	D8	MOV D,A	
217	331	D9	MOV D,B	
218	332	DA	MOV D,C	
219	333	DB	MOV D,D	
220	334	DC	MOV D,E	
221	335	DD	MOV D,H	
222	336	DE	MOV D,L	
223	337	DF	MOV D,M	
224	340	E0	MOV E,A	
225	341	E1	MOV E,B	
226	342	E2	MOV E,C	
227	343	E3	MOV E,D	
228	344	E4	MOV E,E	
229	345	E5	MOV E,H	
230	346	E6	MOV E,L	
231	347	E7	MOV E,M	
232	350	E8	MOV H,A	
233	351	E9	MOV H,B	
234	352	EA	MOV H,C	
235	353	EB	MOV H,D	
236	354	EC	MOV H,E	
237	355	ED	MOV H,H	
238	356	EE	MOV H,L	
239	357	EF	MOV H,M	
240	360	F0	MOV L,A	
241	361	F1	MOV L,B	
242	362	F2	MOV L,C	
243	363	F3	MOV L,D	
244	364	F4	MOV L,E	
245	365	F5	MOV L,H	

DEC	OCTAL	HEX	MNEMONIC	COMMENT
246	366	F6	MOV L,L	
247	367	F7	MOV L,M	
248	370	F8	MOV M,A	
249	371	F9	MOV M,B	
250	372	FA	MOV M,C	
251	373	FB	MOV M,D	
252	374	FC	MOV M,E	
253	375	FD	MOV M,H	
254	376	FE	MOV M,L	
255	377	FF	—	

APPENDIX "C"

- - INSTRUCTION EXECUTION TIMES - -

The number of machine cycles needed to complete each INTELLEC 8 instruction is given in this appendix. The time required to complete on INTELLEC 8 machine cycle is 12.5 microseconds.

INSTRUCTION	CYCLES	
ACI	2	
ADD	1	; 2 cycles if memory is referenced
ADC	1	; 2 cycles if memory is referenced
ADI	2	
ANA	1	; 2 cycles if memory is referenced
ANI	2	
All CALL instructions	3	
CP	1	; 2 cycles if memory is referenced
CPI	2	
DCR	1	
HLT	1	
IN	2	
INR	1	
All JUMP instructions	3	
MOV	1	; 2 cycles if memory is referenced
MVI	2	; 3 cycles if memory is referenced
OR	1	; 2 cycles if memory is referenced
ORI	2	
OUT	2	
RAL	1	
RAR	1	
All RETURN instructions	1	
RLC	1	
RRC	1	
RST	1	
SBB	1	; 2 cycles if memory is referenced
SBI	2	
SUB	1	; 2 cycles if memory is referenced
SUI	1	
XOR	1	; 2 cycles if memory is referenced
XRI	2	

APPENDIX "D"

-- ASCII TABLE --

The 8008 uses a seven-bit ASCII code, which is the normal 8 bit ASCII code with the parity (high order) bit always reset.

Graphic or Control	ASCII (Hexadecimal)
NULL	00
SOM	01
EOA	02
EOM	03
EOT	04
WRU	05
RU	06
BELL	07
FE	08
H.Tab	09
Line Feed	0A
V. Tab	0B
Form	0C
Return	0D
SO	0E
SI	0F
DCO	10
X-On	11
Tape Aux. On	12
X-Off	13
Tape Aux. Off	14
Error	15
Sync	16
LEM	17
S0	18
S1	19
S2	1A
S3	1B
S4	1C
S5	1D
S6	1E
S7	1F

Graphic or Control	ASCII Hexadecimal
ACK	7C
Alt. Mode	7D
Rubout	7F
!	21
"	22
#	23
$	24
%	25
&	26
'	27
(	28
)	29
*	2A
+	2B
,	2C
-	2D
.	2E
/	2F
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
[	5B
/	5C
]	5D
↑	5E
←	5F
@	40
blank	20
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39

Graphic or Control	ASCII Hexadecimal
A	41
B	42
C	43
D	44
E	45
F	46
G	47
H	48
I	49
J	4A
K	4B
L	4C
M	4D
N	4E
O	4F
P	50
Q	51
R	52
S	53
T	54
U	55
V	56
W	57
X	58
Y	59
Z	5A

APPENDIX "E"

-- BINARY-DECIMAL-HEXADECIMAL CONVERSION TABLES --

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

TABLE OF POWERS OF SIXTEEN$_{10}$

16^n	n	16^{-n}					
1	0	0.10000	00000	00000	00000	x	10
16	1	0.62500	00000	00000	00000	x	10^{-1}
256	2	0.39062	50000	00000	00000	x	10^{-2}
4 096	3	0.24414	06250	00000	00000	x	10^{-3}
65 536	4	0.15258	78906	25000	00000	x	10^{-4}
1 048 576	5	0.95367	43164	06250	00000	x	10^{-6}
16 777 216	6	0.59604	64477	53906	25000	x	10^{-7}
268 435 456	7	0.37252	90298	46191	40625	x	10^{-8}
4 294 967 296	8	0.23283	06436	53869	62891	x	10^{-9}
68 719 476 736	9	0.14551	91522	83668	51807	x	10^{-10}
1 099 511 627 776	10	0.90949	47017	72928	23792	x	10^{-12}
17 592 186 044 416	11	0.56843	41886	08080	14870	x	10^{-13}
281 474 976 710 656	12	0.35527	13678	80050	09294	x	10^{-14}
4 503 599 627 370 496	13	0.22204	46049	25031	30808	x	10^{-15}
72 057 594 037 927 936	14	0.13877	78780	78144	56755	x	10^{-16}
1 152 921 504 606 846 976	15	0.86736	17379	88403	54721	x	10^{-18}

TABLE OF POWERS OF 10_{16}

10^n	n	10^{-n}					
1	0	1.0000	0000	0000	0000		
A	1	0.1999	9999	9999	999A		
64	2	0.28F5	C28F	5C28	F5C3	x	16^{-1}
3E8	3	0.4189	374B	C6A7	EF9E	x	16^{-2}
2710	4	0.68DB	8BAC	710C	B296	x	16^{-3}
1 86A0	5	0.A7C5	AC47	1B47	8423	x	16^{-4}
F 4240	6	0.10C6	F7A0	B5ED	8D37	x	16^{-4}
98 9680	7	0.1AD7	F29A	BCAF	4858	x	16^{-5}
5F5 E100	8	0.2AF3	1DC4	6118	73BF	x	16^{-6}
3B9A CA00	9	0.44B8	2FA0	9B5A	52CC	x	16^{-7}
2 540B E400	10	0.6DF3	7F67	5EF6	EADF	x	16^{-8}
17 4876 E800	11	0.AFEB	FF0B	CB24	AAFF	x	16^{-9}
E8 D4A5 1000	12	0.1197	9981	2DEA	1119	x	16^{-9}
918 4E72 A000	13	0.1C25	C268	4976	81C2	x	16^{-10}
5AF3 107A 4000	14	0.2D09	370D	4257	3604	x	16^{-11}
3 8D7E A4C6 8000	15	0.480E	BE7B	9D58	566D	x	16^{-12}
23 8652 6FC1 0000	16	0.734A	CA5F	6226	F0AE	x	16^{-13}
163 4578 5D8A 0000	17	0.B877	AA32	36A4	B449	x	16^{-14}
DE0 B6B3 A764 0000	18	0.1272	5DD1	D243	ABA1	x	16^{-14}
8AC7 2304 89E8 0000	19	0.1D83	C94F	B6D2	AC35	x	16^{-15}

HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0–FFF and decimal integers in the range 0–4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
510	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
590	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont.)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

intel®

West: 17291 Irvine Blvd., Suite 262/(714)838-1126, TWX: 910-595-1114/Tustin, California 92680
Mid-America: 800 Southgate Office Plaza/501 West 78th St./(612)835-6722, TWX: 910-576-2867/Bloomington, Minnesota 55437
Northeast: 2 Militia Drive, Suite 4/(617)861-1136, Telex: 92-3493/Lexington, Massachusetts 02173
Mid-Atlantic: 21 Bala Avenue/(215)664-6636/Bala Cynwyd, Pennsylvania 19004
Europe: Intel Office/Vester Farimagsgade 7/45-1-11 5644, Telex: 19567/DK 1606 Copenhagen V
Orient: Intel Japan Corp./Han-Ei 2nd Building/1-1, Shinjuku, Shinjuku-Ku/03-354-8251, Telex: 781-28426/Tokyo 160