

K. Lienm



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

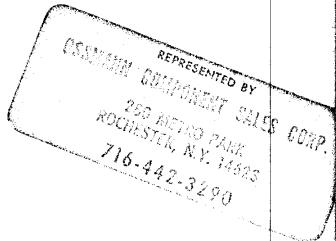
Programming Manual for the 8080 Microcomputer System

PRELIMINARY EDITION

VOLUME 1

May 1974

©Intel Corporation 1974



This manual describes the assembly language format, and how to write assembly language programs. For detailed information on the operation of specific assemblers, the corresponding Operator's manuals should be consulted.

-- TABLE OF CONTENTS --

<u>SECTION</u>		<u>Page No.</u>
1.0	INTRODUCTION	1-1
2.0	COMPUTER ORGANIZATION	2-1
2.1	WORKING REGISTERS	2-1
2.2	MEMORY	2-2
2.3	PROGRAM COUNTER	2-2
2.4	STACK POINTER	2-2
2.5	INPUT/OUTPUT	2-3
2.6	COMPUTER PROGRAM REPRESENTATION IN MEMORY	2-3
2.7	MEMORY ADDRESSING	2-5
2.7.1	DIRECT ADDRESSING	2-5
2.7.2	REGISTER PAIR ADDRESSING	2-6
2.7.3	STACK POINTER ADDRESSING	2-6
2.7.4	IMMEDIATE ADDRESSING	2-8
2.7.5	SUBROUTINES AND USE OF THE STACK FOR ADDRESSING	2-9
2.8	CONDITION BITS	2-11
2.8.1	CARRY BIT	2-12
2.8.2	AUXILIARY CARRY BIT	2-12
2.8.3	SIGN BIT	2-13
2.8.4	ZERO BIT	2-13
2.8.5	PARITY BIT	2-13
3.0	THE 8080 INSTRUCTION SET	3-1
3.1	ASSEMBLY LANGUAGE	3-1
3.1.1	HOW ASSEMBLY LANGUAGE IS USED	3-1
3.1.2	STATEMENT MNEMONICS	3-4
3.1.3	LABEL FIELD	3-5
3.1.4	CODE FIELD	3-6
3.1.5	OPERAND FIELD	3-7
3.1.6	COMMENT FIELD	3-14
3.2	DATA STATEMENTS	3-15
3.2.1	TWO'S COMPLEMENT REPRESENTATION	3-15
3.2.2	DB DEFINE BYTE(S) OF DATA	3-18
3.2.3	DW DEFINE WORD (TWO BYTES) OF DATA	3-19

-- TABLE OF CONTENTS -- (Cont.)

<u>SECTION</u>		<u>Page No.</u>
	3.2.4 DS DEFINE STORAGE (BYTES)	3-20
3.3	CARRY BIT INSTRUCTIONS	3-21
	3.3.1 STC SET CARRY	3-21
	3.3.2 CMC COMPLEMENT CARRY	3-22
3.4	SINGLE REGISTER INSTRUCTIONS	3-22
	3.4.1 INR INCREMENT REGISTER OR MEMORY	3-23
	3.4.2 DCR DECREMENT REGISTER OR MEMORY	3-24
	3.4.3 CMA COMPLEMENT ACCUMULATOR	3-25
	3.4.4 DAA DECIMAL ADJUST ACCUMULATOR	3-26
3.5	NOP INSTRUCTION	3-28
3.6	DATA TRANSFER INSTRUCTIONS	3-29
	3.6.1 MOV INSTRUCTION	3-30
	3.6.2 STAX STORE ACCUMULATOR	3-31
	3.6.3 LDAX LOAD ACCUMULATOR	3-32
3.7	REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS	3-33
	3.7.1 ADD ADD REGISTER OR MEMORY TO ACCUMULATOR	3-34
	3.7.2 ADC ADD REGISTER OR MEMORY TO ACCUMULATOR WITH CARRY	3-35
	3.7.3 SUB SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR	3-36
	3.7.4 SBB SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW	3-38
	3.7.5 ANA LOGICAL AND REGISTER OR MEMORY WITH ACCUMULATOR	3-39
	3.7.6 XRA LOGICAL EXCLUSIVE-OR REGISTER OR MEMORY WITH ACCUMULATOR (ZERO ACCUMULATOR)	3-40
	3.7.7 ORA LOGICAL OR REGISTER OR MEMORY WITH ACCUMULATOR	3-42
	3.7.8 CMP COMPARE REGISTER OR MEMORY WITH ACCUMULATOR	3-44

- TABLE OF CONTENTS -- (Cont.)

<u>SECTION</u>		<u>Page No.</u>
3.8	ROTATE ACCUMULATOR INSTRUCTIONS	3-46
3.8.1	RLC ROTATE ACCUMULATOR LEFT	3-46
3.8.2	RRC ROTATE ACCUMULATOR RIGHT	3-47
3.8.3	RAL ROTATE ACCUMULATOR LEFT THROUGH CARRY	3-48
3.8.4	RAR ROTATE ACCUMULATOR RIGHT THROUGH CARRY	3-49
3.9	REGISTER PAIR INSTRUCTIONS	3-50
3.9.1	PUSH PUSH DATA ONTO STACK	3-50
3.9.2	POP POP DATA OFF STACK	3-52
3.9.3	DAD DOUBLE ADD	3-53
3.9.4	INX INCREMENT REGISTER PAIR	3-54
3.9.5	DCX DECREMENT REGISTER PAIR	3-55
3.9.6	XCHG EXCHANGE REGISTERS	3-56
3.9.7	XTHL EXCHANGE STACK	3-57
3.9.8	SPHL LOAD SP FROM H AND L	3-58
3.10	IMMEDIATE INSTRUCTIONS	3-59
3.10.1	LXI LOAD REGISTER PAIR IMMEDIATE	3-61
3.10.2	MVI MOVE IMMEDIATE DATA	3-62
3.10.3	ADI ADD IMMEDIATE TO ACCUMULATOR	3-63
3.10.4	ACI ADD IMMEDIATE TO ACCUMULATOR WITH CARRY	3-64
3.10.5	SUI SUBTRACT IMMEDIATE FROM ACCUMULATOR	3-65
3.10.6	SBI SUBTRACT IMMEDIATE FROM ACCUMULATOR WITH BORROW	3-67
3.10.7	ANI AND IMMEDIATE WITH ACCUMULATOR	3-69
3.10.8	XRI EXCLUSIVE-OR IMMEDIATE WITH ACCUMULATOR	3-70
3.10.9	ORI OR IMMEDIATE WITH ACCUMULATOR	3-71
3.10.10	CPI COMPARE IMMEDIATE WITH ACCUMULATOR	3-72
3.11	DIRECT ADDRESSING INSTRUCTIONS	3-73
3.11.1	STA STORE ACCUMULATOR DIRECT	3-74
3.11.2	LDA LOAD ACCUMULATOR DIRECT	3-74
3.11.3	SHLD STORE H AND L DIRECT	3-75
3.11.4	LHLD LOAD H AND L DIRECT	3-76

-- TABLE OF CONTENTS -- (Cont.)

<u>SECTION</u>		<u>Page No.</u>
3.12	JUMP INSTRUCTIONS	3-77
3.12.1	PCHL LOAD PROGRAM COUNTER	3-78
3.12.2	JMP JUMP	3-80
3.12.3	JC JUMP IF CARRY	3-81
3.12.4	JNC JUMP IF NO CARRY	3-81
3.12.5	JZ JUMP IF ZERO	3-82
3.12.6	JNZ JUMP IF NOT ZERO	3-82
3.12.7	JM JUMP IF MINUS	3-83
3.12.8	JP JUMP IF POSITIVE	3-83
3.12.9	JPE JUMP IF PARITY EVEN	3-84
3.12.10	JPO JUMP IF PARITY ODD	3-84
3.13	CALL SUBROUTINE INSTRUCTIONS	3-86
3.13.1	CALL CALL	3-87
3.13.2	CC CALL IF CARRY	3-88
3.13.3	CNC CALL IF NO CARRY	3-88
3.13.4	CZ CALL IF ZERO	3-89
3.13.5	CNZ CALL IF NOT ZERO	3-89
3.13.6	CM CALL IF MINUS	3-90
3.13.7	CP CALL IF PLUS	3-90
3.13.8	CPE CALL IF PARITY EVEN	3-91
3.13.9	CPO CALL IF PARITY ODD	3-91
3.14	RETURN FROM SUBROUTINE INSTRUCTIONS	3-92
3.14.1	RET RETURN	3-93
3.14.2	RN RETURN IF CARRY	3-93
3.14.3	RNC RETURN IF NO CARRY	3-94
3.14.4	RZ RETURN IF ZERO	3-94
3.14.5	RNZ RETURN IF NOT ZERO	3-95
3.14.6	RM RETURN IF MINUS	3-95
3.14.7	RP RETURN IF PLUS	3-96
3.14.8	RPE RETURN IF PARITY EVEN	3-96
3.14.9	RPO RETURN IF PARITY ODD	3-97
3.15	RST INSTRUCTION	3-98
3.16	INTERRUPT FLIP-FLOP INSTRUCTIONS	3-99
3.16.1	EI ENABLE INTERRUPTS	3-100
3.16.2	DI DISABLE INTERRUPTS	3-100
3.17	INPUT/OUTPUT INSTRUCTIONS	3-101
3.17.1	IN INPUT	3-102
3.17.2	OUT OUTPUT	3-103

-- TABLE OF CONTENTS -- (Cont.)

<u>SECTION</u>		<u>Page No.</u>
3.18	HLT HALT INSTRUCTION	3-104
3.19	PSEUDO-INSTRUCTIONS	3-105
3.19.1	ORG ORIGIN	3-106
3.19.2	EQU EQUATE	3-107
3.19.3	SET	3-108
3.19.4	END END OF ASSEMBLY	3-109
3.19.5	IF AND ENDIF CONDITIONAL ASSEMBLY	3-110
3.19.6	MACRO AND ENDM MACRO DEFINITION	3-111
4.0	PROGRAMMING WITH MACROS	4-1
4.1	WHAT ARE MACROS?	4-1
4.2	MACRO TERMS AND USE	4-5
4.2.1	MACRO DEFINITION	4-5
4.2.2	MACRO REFERENCE OR CALL	4-6
4.2.3	MACRO EXPANSION	4-8
4.2.4	SCOPE OF LABELS AND NAMES WITHIN MACROS	4-8
4.2.5	MACRO PARAMETER SUBSTITUTION	4-12
4.3	REASONS FOR USING MACROS	4-13
4.4	USEFUL MACROS	4-13
4.4.1	LOAD INDIRECT MACRO	4-13
4.4.2	OTHER INDIRECT ADDRESSING MACROS	4-15
4.4.3	CREATE INDEXED ADDRESS MACRO	4-15
5.0	PROGRAMMING TECHNIQUES	5-1
5.1	BRANCH TABLES PSEUDO-SUBROUTINE	5-1
5.2	SUBROUTINES	5-3
5.2.1	TRANSFERRING DATA TO SUBROUTINES	5-5
5.3	SOFTWARE MULTIPLY AND DIVIDE	5-9
5.4	MULTIBYTE ADDITION AND SUBTRACTION	5-13
5.5	DECIMAL ADDITION	5-16
5.6	DECIMAL SUBTRACTION	5-18
5.7	ALTERING MACRO EXPANSIONS	5-21

-- TABLE OF CONTENTS -- (Cont.)

<u>SECTION</u>	<u>Page No.</u>
6.0 INTERRUPTS	6-1
6.1 WRITING INTERRUPT SUBROUTINES	6-4
APPENDIX A INSTRUCTION SUMMARY	A-1
APPENDIX B INSTRUCTION EXECUTION TIMES AND BIT PATTERNS	B-1
APPENDIX C ASCII TABLE	C-1
APPENDIX D BINARY-DECIMAL-HEXADECIMAL CONVERSION TABLES	D-1

LIST OF FIGURES

2-1 AUTOMATIC ADVANCE OF THE PROGRAM COUNTER AS INSTRUCTIONS ARE EXECUTED	2-4
3-1 ASSEMBLER PROGRAM CONVERTS ASSEMBLY LANGUAGE SOURCE PROGRAM TO HEXADECIMAL OBJECT PROGRAM	3-3

-- TERMS --

TERMS

DESCRIPTION

Address	A 16-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, represented by the binary digits 0 or 1).
Byte	A group of 8 contiguous bits occupying a single memory location.
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.
Word	A group of 16 contiguous bits occupying two successive memory locations. (2 bytes).
nnnnB	nnnn represents a number in binary format.

TERMS

nnnnD

nnnn represents a number in decimal format.

nnnnO

nnnn represents a number in octal format.

nnnnQ

nnnn represents a number in octal format.

nnnnH

nnnn represents a number in hexadecimal format.

0	0	1	1	R P	0
---	---	---	---	-----	---

A representation of a byte in memory. Bits which are fixed as 0 or 1 are indicated by 0 or 1; bits which may be either 0 or 1 in different circumstances are represented by letters; thus RP represents a three-bit field which contains one of the eight possible combinations of zeroes and ones.

1.0 INTRODUCTION

This manual has been written to help the reader program the INTEL 8080 microcomputer in assembly language. Accordingly, this manual assumes that the reader has a good understanding of logic, but may be completely unfamiliar with programming concepts.

For those readers who do understand programming concepts, several features of the INTEL 8080 microcomputer are described below. They include:

- 8-bit parallel CPU on a single chip
- 78 instructions, including extensive memory referencing, flexible jump-on-condition capability, and binary and decimal arithmetic modes
- Direct addressing for 65,536 bytes of memory
- Fully programmable stacks, allowing unlimited subroutine nesting and full interrupt handling capability
- Seven 8-bit registers

There are two ways in which programs for the 8080 may be assembled; either via the resident assembler or the cross assembler. The resident assembler is one of three system programs available to the user which run on the 8080, the others being an Editor and a System Monitor. The cross assembler runs on any computer having a FORTRAN compiler whose word size is 32 bits or greater, and generates programs which run on the 8080.

The experienced programmer should note that the assembly language has a macro capability which allows users to tailor the assembly language to individual needs.

2.0 COMPUTER ORGANIZATION

This section provides the programmer with a functional overview of the 8080. Information is presented in this section at a level that provides a programmer with necessary background in order to write efficient programs.

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

- (1) Seven working registers in which all data operations occur, and which provide one means for addressing memory.
- (2) Memory, which may hold program instructions or data and which must be addressed location by location in order to access stored information.
- (3) The program counter, whose contents indicate the next program instruction to be executed.
- (4) The stack pointer, a register which enables various portions of memory to be used as stacks. These in turn facilitate execution of subroutines and handling of interrupts as described later.
- (5) Input/Output, which is the interface between a program and the outside world.

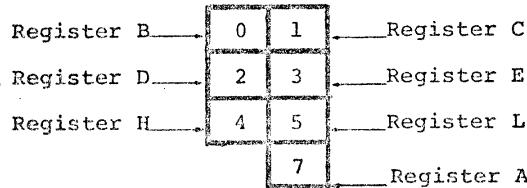
2.1 WORKING REGISTERS

The 8080 provides the programmer with an 8-bit accumulator and six additional 8-bit "scratchpad" registers.

These seven working registers are numbered and referenced via the integers 0,1,2,3,4,5, and 7; by convention, these registers may also be accessed via the letters B,C,D,E,H,L, and A (for the accumulator), respectively.

Some 8080 operations reference the working registers in pairs referenced by the letters B,D,H and PSW. These correspondences are shown as follows:

INDIVIDUAL REGISTER REFERENCE



REGISTER PAIR REFERENCE

Register Pair B	0	1
Register Pair D	2	3
Register Pair H	4	5
Register Pair PSW	6	7

special data byte

NOTE: When register pair PSW is specified, the first (most significant) 8 bits referenced are a special byte reflecting the current status of the machine, as described in Sections 3.9.1 and 3.9.2.

2.2 MEMORY

The 8080 can be used with read only memory, programmable read only memory and read/write memory. A program can cause 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 (represented by two hexadecimal digits). Up to 65,536 bytes of memory may be present, and an individual memory byte is addressed by its sequential number from 0 to 65,535D=FFFFH, the largest number which can be represented by 16 bits.

The bits stored in a memory byte may represent the encoded form of an instruction or may be data, as described in Section 3.2.

2.3 PROGRAM COUNTER

The program counter is a 16 bit register which is accessible to the programmer and whose contents indicate the address of the next instruction to be executed as described in Section 2.6.

2.4 STACK POINTER

A stack is an area of memory set aside by the programmer in which data or addresses are stored and retrieved by stack operations. Stack operations are performed by several of the 8080 instructions, and facilitate execution of subroutines and handling of program interrupts. The programmer specifies which addresses the stack operations will operate upon via a special accessible 16-bit register called the stack pointer.

2.5 INPUT/OUTPUT

To the 8080, the outside world consists of up to 256 input devices and 256 output devices. Each device communicates with the 8080 via data bytes sent to or received from the accumulator, and each device is assigned a number from 0 to 255 which is not under control of the programmer. The instructions which perform these data transmissions are described in Section 3.17.

2.6 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 bits which represent the instructions of the program, and which we will represent via hexadecimal digits. The memory address of the next instruction to be executed is held in the program counter. 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, call, or return) 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 1FH has been selected to represent the instruction RAR (rotate the contents of the accumulator right through carry); thus, the value 1FH stored in a memory byte could either represent the instruction RAR, or it could represent the data value 1FH. It is up to the logic of a program to insure that data is not misinterpreted as an 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. 8080 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.

Memory Address	Instruction Number	Program Counter Contents
0212	1	0213
0213	2	0215
0214		
0215	3	0216
0216		0219
0217	4	
0218		
0219	5	021B
021A		
021B	6	021C
021C		021F
021D	7	
021E		
021F	8	0220
0220	9	0221
0221	10	0222

FIGURE 2-1.
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. But 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.

2.7 MEMORY ADDRESSING

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.

2.7.1 DIRECT ADDRESSING

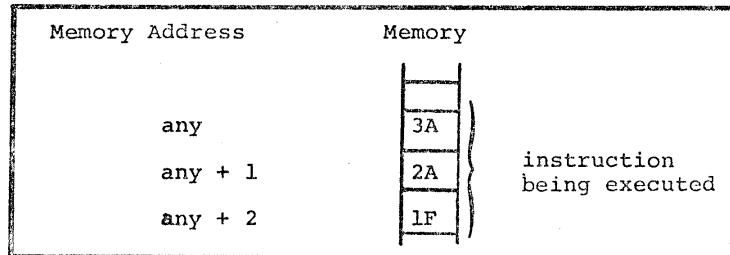
With direct addressing, an instruction supplies an exact memory address.

The instruction:

"Load the contents of memory address 1F2A into the accumulator"

is an example of an instruction using direct addressing, 1F2A being the direct address.

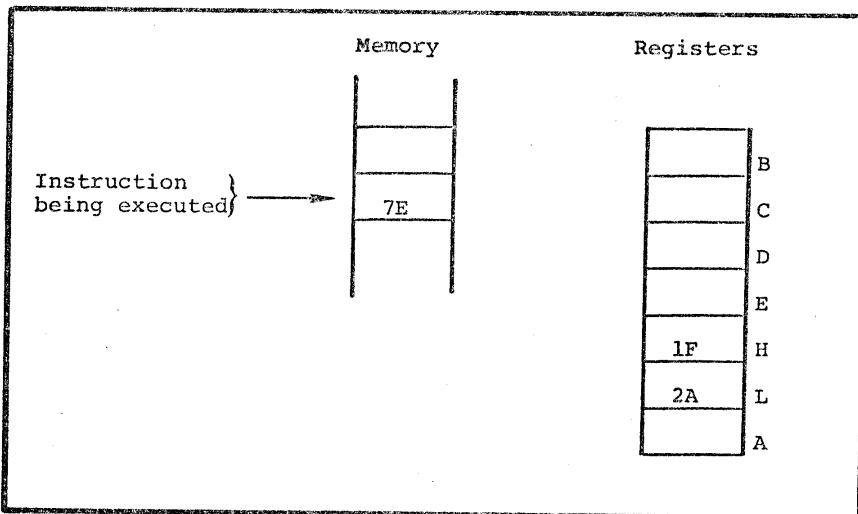
This would appear in memory as follows:



The instruction occupies three memory bytes, the second and third of which hold the direct address.

2.7.2 REGISTER PAIR ADDRESSING

A memory address may be specified by the contents of a register pair. For almost all 8080 instructions, the H and L registers must be used. The H register contains the most significant 8 bits of the referenced address, and the L register contains the least significant 8 bits. A one byte instruction which will load the accumulator with the contents of memory byte 1F2A would appear as follows:



In addition, there are two 8080 instructions which use either the B and C registers or the D and E registers to address memory. As above, the first register of the pair holds the most significant 8 bits of the address, while the second register holds the least significant 8 bits. These instructions, STAX and LDAX, are described in Section 3.6.

2.7.3 STACK POINTER ADDRESSING

Memory locations may be addressed via the 16-bit stack pointer register, as described below.

There are only two stack operations which may be performed; putting data into a stack is called a push, while retrieving data from a stack is called a pop.

NOTE: In order for stack push operations to operate, stacks must be located in read/write memory.

STACK PUSH OPERATION:

16 bits of data are transferred to a memory area (called a stack) from a register pair or the 16 bit program counter during any stack push operation. The addresses of the memory area which is to be accessed during a stack push operation are determined by using the stack pointer as follows:

- (1) The most significant 8 bits of data are stored at the memory address one less than the contents of the stack pointer.
- (2) The least significant 8 bits of data are stored at the memory address two less than the contents of the stack pointer.
- (3) The stack pointer is automatically decremented by two.

For example, suppose that the stack pointer contains the address 13A6H, register B contains 6AH, and register C contains 30H. Then a stack push of register pair B would operate as follows:

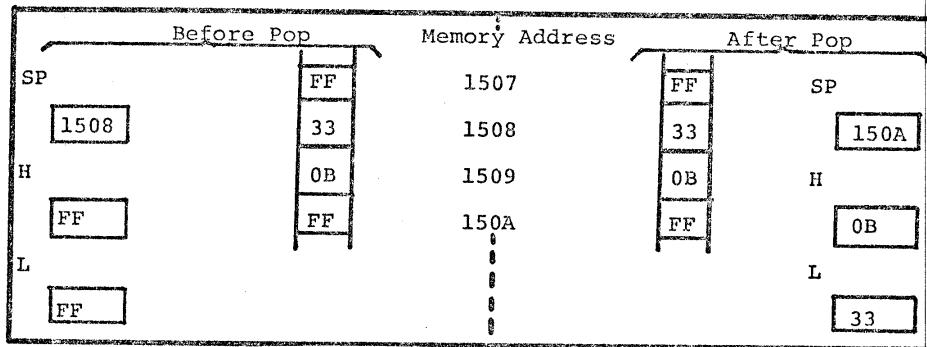
Before Push		Memory Address	After Push	
SP	13A6	13A3	FF	SP
B	6A	13A4	30	13A4
C	30	13A5	6A	B
		13A6	FF	6A
				30

STACK POP OPERATION

16 bits of data are transferred from a memory area (called a stack) to a register pair or the 16-bit program counter during any stack pop operation. The addresses of the memory area which is to be accessed during a stack pop operation are determined by using the stack pointer as follows:

- (1) The second register of the pair, or the least significant 8 bits of the program counter, are loaded from the memory address held in the stack pointer.
- (2) The first register of the pair, or the most significant 8 bits of the program counter, are loaded from the memory address one greater than the address held in the stack pointer.
- (3) The stack pointer is automatically incremented by two.

For example, suppose that the stack pointer contains the address 1508H, memory location 1508H contains 33H, and memory location 1509H contains 0BH. Then a stack pop into register pair H would operate as follows:



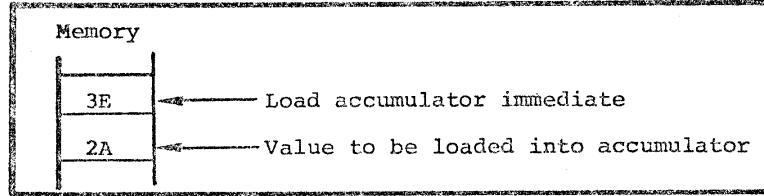
The programmer loads the stack pointer with any desired value by using the LXI instruction described in Section 3.10.1. The programmer must initialize the stack pointer before performing a stack operation, or erroneous results will occur.

2.7.4 IMMEDIATE ADDRESSING

An immediate instruction is one that contains data. The following is an example of immediate addressing:

"Load the accumulator with the value 2AH".

The above instruction would be coded in memory as follows:

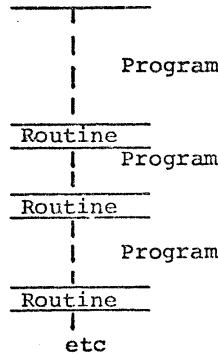


Immediate instructions do not reference memory; rather they contain data in the memory byte following the instruction code byte.

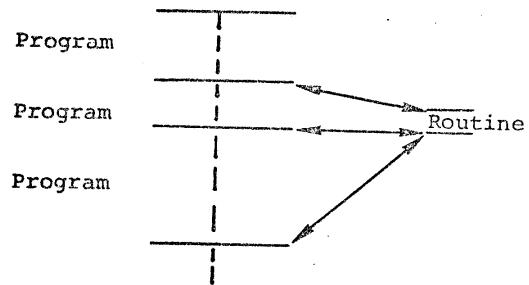
2.7.5 SUBROUTINES AND USE OF THE STACK FOR ADDRESSING

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 multiplication. The 8080 provides instructions to add one byte of data to another byte of data, but what if you wish to multiply these numbers? This will require a number of instructions to be executed in sequence. It is quite possible that this 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:

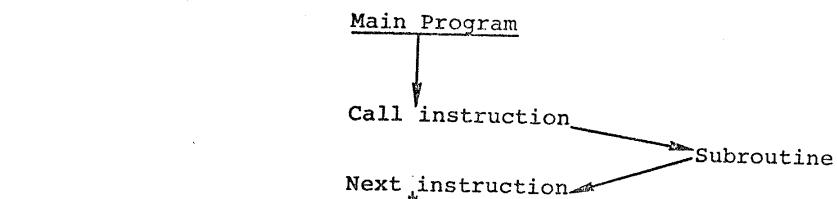


A more efficient means of accessing the routine would be to store it once, and find a way of accessing it when needed:



A frequently accessed routine such as the above is called a subroutine, and the 8080 provides instructions that call 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 (that is, the address held in the program counter), is pushed onto the stack (see Section 2.4), and the subroutine is executed. The last executed instruction of a subroutine will usually be a "Return Instruction", which pops an address off the stack into the program counter, and thus causes program execution to continue at the "Next" instruction as illustrated below:

Memory Address	Instruction	
0C02		
0C03	CALL SUBROUTINE	
0C04	02	
0C05	0F	
0C06	NEXT INSTRUCTION	
OF00		
OF01		
OF02	FIRST SUBROUTINE INSTRUCTION	
OF03		
—	Body of subroutine	
—		
—		
OF4E		
OF4F	RETURN	

Push address
of next insti-
tion (0C06H)
onto the stack
and branch
to subroutine
starting at
0F02H

Pop return
address (0C06H)
off stack and
return to next
instruction

Subroutines may be nested up to any depth limited only by the amount of memory available for the stack. For example, the first subroutine could itself call some other subroutine and so on. An examination of the sequence of stack pushes and pops 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.

2.8 CONDITION BITS

Five condition (or status) bits are provided by the 8080 to reflect the results of data operations. All but one of these bits (the auxiliary carry bit) may be tested by program instructions which affect subsequent program execution. 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, "setting" a bit causes its value to be 1, while "resetting" a bit causes its value to be 0.

2.8.1 CARRY BIT

The carry bit is set and reset by certain data operations, and its status can be directly tested by a program. The operations which affect the carry bit are addition, subtraction, rotate, and logical operations. For example, addition of two one-byte numbers can produce an answer that does not fit in one byte:

Bit No. 7 6 5 4 3 2 1 0

$$\begin{array}{r} AE = 1 0 1 0 1 1 1 0 \\ + 74 = 0 1 1 1 0 1 0 0 \\ \hline 122 \end{array}$$

overflow=1, sets carry=1

An addition operation that results in an overflow out of the high-order bit will set the carry bit; an addition operation that could have resulted in an overflow but did not will reset the carry bit.

NOTE: Addition, subtraction, rotate, and logical operations follow different rules for setting and resetting the carry bit. See Section 3.2.1 and the individual instruction descriptions in Chapter 3 for details. The 8080 instructions which use the addition operation are ADD, ADC, ADI, ACI, and DAD. The instructions which use the subtraction operation are SUB, SBB, SUI, SBI, CMP, and CPI. Rotate operations are RAL, RAR, RLC, and RRC. Logical operations are ANA, ORA, XRA, ANI, ORI, and XRI.

2.8.2 AUXILIARY CARRY BIT

The auxiliary carry bit indicates overflow out of bit 3 in the same way that the carry bit indicates overflow out of bit 7. The state of the auxiliary carry bit cannot be directly tested by a program instruction and is present only to enable one instruction (DAA, described in Section 3.4.4) to perform its function. The following addition will reset the carry bit and set the auxiliary carry bit:

Bit No. 7 6 5 4 3 2 1 0

$$\begin{array}{r} 2E = 0 0 1 0 1 1 1 0 \\ + 74 = 0 1 1 1 0 1 0 0 \\ \hline A2 \end{array}$$

Carry=0 Auxiliary Carry = 1

The auxiliary carry bit will be affected by all addition, subtraction, increment, decrement, and compare instructions.

2.8.3 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 most significant bit of the answer (bit 7).

2.8.4 ZERO BIT

This condition bit is set if the result generated by the execution of certain instructions is zero. The zero bit is reset if the result is not zero.

A result that has a carry but a zero answer byte, as illustrated below, will also set the zero bit:

Bit No.	7	6	5	4	3	2	1	0
	1	0	1	0	0	1	1	1
	$+ 0 1 0 1 1 0 0 1$							
1]	<hr/> $0 0 0 0 0 0 0 0$							
Carry out of bit 7.								
Zero answer								
Zero bit set to 1.								

2.8.5 PARITY BIT

Byte "parity" is checked after certain operations. 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 reset to 0 for odd parity.

3.0 THE 8080 INSTRUCTION SET

This section describes the 8080 assembly language instruction set.

For the reader who understands assembly language programming, Appendix A provides a complete summary of the 8080 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:

Hexadecimal Memory Address	
1432	7E
1433	C3
1434	C4
1435	14
1436	
.	:
.	:
14C3	FF
14C4	2E
14C5	36
14C6	77

Assuming that 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 36H.

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:

Hexadecimal Memory Address	Old Code	New Code
1432	7E	7E
1433	C3	C3
1434	C4	C5
1435	14	14
1436	.	.
1437	.	.
14C3	FF	FF
14C4	2E	2E
14C5	36	37
14C6	77	77
14C7		

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 8080 assembler is described in the 8080 Operator's Manual.

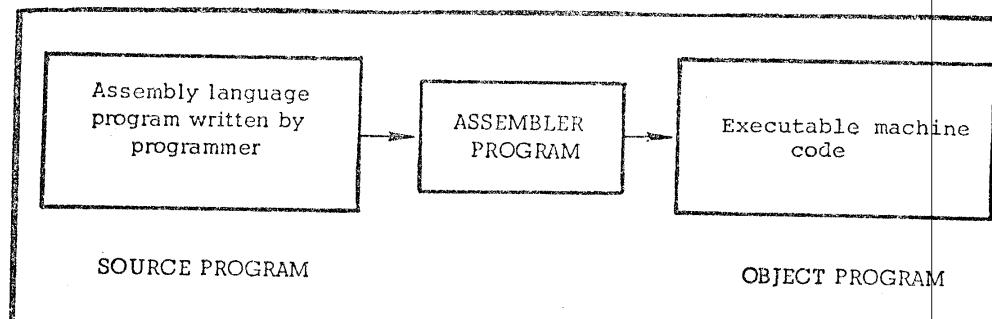


FIGURE 3-1

ASSEMBLER PROGRAM CONVERTS ASSEMBLY LANGUAGE SOURCE PROGRAM TO 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 binary codes that can be loaded into memory and executed.

For example:

<u>Source Program</u>	<u>One Possible Version of the Object Program</u>
NOW: MOV A,B	78
CPI 'C'	FE43
JZ LER	CA7C3D
.	.
LER: MOV M,A	77

NOTE: In this and subsequent examples, it is not necessary to understand the operations of the individual instructions. They are presented only to illustrate typical assembly language statements. Individual instructions are described in Sections 3.3-3.18.

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 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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	
HERE:	MVI	C,0	; 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

NOTE: These examples and the ones which follow are intended to illustrate how the various fields appear in complete assembly language statements. It is not necessary at this point to understand the operations which the statements perform.

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 - 3.18 and Appendix A; pseudo-instructions are described in Section 3.19).

Here are some examples of valid label fields:

LABEL:
F14F:
@HERE:
?ZERO:

Here are some invalid label fields:

123:	begins with a decimal digit
LABEL	is not followed by 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 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 - 3.18 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 JMP THERE

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], as summarized in the following table, and described in detail in the subsequent examples.

OPERAND FIELD INFORMATION	
<u>Information required</u>	<u>Ways of specifying</u>
(a) Register (b) Register Pair (c) Immediate Data (d) 16 bit Memory Address	(1) Hexadecimal Data (2) Decimal Data (3) Octal Data (4) Binary Data (5) Program Counter (\$) (6) ASCII Constant (7) Labels assigned values (8) Labels of instructions (9) Expressions

The nine ways of specifying information are as follows:

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

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
LABEL:	MVI	A,720	; Load the accumulator with ; the octal number 72

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

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
NOW:	MVI	10B, 11110110B	; Load register two (the ; D 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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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:

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

- (7) Labels that have been assigned a numeric value by the assembler. The following assignments are built into the assembler and are therefore always active:

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

Example:

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

<u>Label</u>	<u>Code</u>	<u>Operand</u>
A1:	MVI	D, VALUE
A2:	MVI	2, 9FH
A3:	MVI	2, VALUE

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

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
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 operator + produces the arithmetic sum of its operands.

The operator - produces the arithmetic difference of its operands when used as subtraction, or the arithmetic negative of its operand when used as unary minus.

The operator * produces the arithmetic product of its operands.

The operator / produces the arithmetic integer quotient of its operands, discarding any remainder.

The operator MOD produces the integer remainder obtained by dividing the first operand by the second.

The operator NOT complements each bit of its operand.

The operator AND produces the bit-by-bit logical AND of its operands.

The operator OR produces the bit-by-bit logical OR of its operands.

The operator XOR produces the bit-by-bit logical EXCLUSIVE-OR of its operands.

The SHR and SHL operators are linear shifts which shift their first operands right or left, respectively, by the number of bit positions specified by their second operands. Zeroes are shifted into the high-order or low-order bits, respectively, of their first operands.

The programmer must insure that the result generated by any operation 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 OFFH

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.

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

Examples:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Arbitrary Memory Address</u>
HERE:	MVI	C, HERE SHR 8	2E1A

The above instruction loads the hexadecimal number 2EH (16-bit address of HERE shifted right 8 bits) into the C register.

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

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

<u>Label</u>	<u>Code</u>	<u>Operand</u>
INS:	DB	(ADD C)

The above instruction defines a byte of value 81H (the encoding of an ADD C instruction) at location INS.

Operators cause expressions to be evaluated in the following order:

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

In the case of parenthesized expressions, the most deeply parenthesized expressions are evaluated first:

Example:

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 specifications must finally evaluate to one of the numbers 0 - 7 as follows:

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

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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 (the H register).

- (b) A register pair to serve as the source or destination in a data operation. Register pairs are specified as follows:

<u>Specification</u>	<u>Register Pair</u>
B	Registers B and C
D	Registers D and E
H	Registers H and L
PSW	One byte indicating the state of the condition bits, and Register A (see Sections 4.9.1 and 4.9.2)
SP	The 16-bit stack pointer register

NOTE: The binary value representing each register pair varies from instruction to instruction. Therefore, the programmer should always specify a register pair by its alphabetic designation.

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	PUSH	D	; Push registers D and E ; onto stack
	INX	SP	; Increment 16-bit number ; in the stack pointer

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

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
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 OFFH (where ADDR is a 16-bit address)
127

VALUE (where VALUE has been equated to a number)
3EH=10/(2 AND 2)

- (d) A 16-bit address, or the label of another instruction in memory.

Example:

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

3.1.6 COMMENT FIELD

The only rule governing this field is that it must begin with a semicolon (;).

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. For instance, the code 1FH may be interpreted as a machine instruction (Rotate Accumulator Right Through Carry), as a hexadecimal value 1FH=31D, or merely as the bit pattern 00001111.

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".

WHY TWO'S COMPLEMENT?

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

3.2.1 TWO'S COMPLEMENT REPRESENTATION

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:

$$\begin{aligned} 0 &= 00000000B=0H \\ 1 &= 00000001B=1H \\ &\vdots \\ 126D &= 01111110B=7EH \\ 127D &= 01111111B=7FH \end{aligned}$$

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

- (a) Complement 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:

$$+10D = 00001010B$$

Complement each bit : 11110101B
Add one : 11110110B

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 complement each bit and add one.

$$86H = 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ B$$

Complement 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 86H is -7AH = -122D

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

$$\begin{aligned} -1 &= 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ B = FFH \\ -2 &= 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ B = FEH \\ \cdot & \\ \cdot & \\ -127D &= 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ B = 81H \\ -128D &= 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ B = 80H \end{aligned}$$

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

Take the two's complement of 0CH=F4H

Add the result to the minuend:

$$\begin{array}{r} \text{LAH} = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0 \\ + (-0CH) = \underline{\text{F4H}} = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0 \\ \hline 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0 \end{array} = 0EH \text{ the correct answer}$$

When a byte is interpreted as an unsigned two's complement number, its value is considered positive and in the range 0 to 255₁₀:

$0 = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ B = 0H$
 $1 = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ B = 1H$
 .
 .
 $127D = 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ B = 7FH$
 $128D = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ B = 80H$
 .
 .
 $255D = 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ B = FFH$

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.

$$\begin{array}{r}
 197D = 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1 = C5H \\
 -98D = 1\ 0\ 0\ 1\ 1\ 1\ 1\ 0 = 9EH \\
 \hline
 \text{Overflow} \rightarrow 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1 = 63H = 99D
 \end{array}$$

Since the overflow out of bit 7 = 1, indicating that the answer is correct and positive, the subtract operation will reset the carry bit to 0.

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

$$\begin{array}{r}
 12D = 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0 = 0CH \\
 -15D = 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1 = 0F1H \\
 \hline
 \text{Overflow} \rightarrow 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1 = -3D
 \end{array}$$

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 8080 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

$$\begin{array}{r} +12D = 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0 \\ +(-15D) = 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1 \\ \hline 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \end{array} = -3D$$

causes carry to be reset

"SUB" +15D from +12D

$$\begin{array}{r} +12D = 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0 \\ -(+15D) = 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1 \\ \hline 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \end{array} = -3D$$

causes carry to be set

3.2.2 DB DEFINE BYTE(S) OF DATA

Format

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	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 "oplab". (The most significant bit of each ASCII character is always =0).

Example:

<u>Instruction</u>	<u>Assembled Data (hex)</u>		
HERE: DB 0A3H	A3		
WORD1: DB 5*2, 2FH-0AH	0A25		
WORD2: 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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	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 (oplab), and the most significant 8 bits are stored in the next higher addressed byte (oplab +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 address memory location 3B1CH and FILL addresses memory location 3EB4H.

<u>Instruction</u>	<u>Assembled Data (hex)</u>		
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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	DS	exp

"exp" is a single arithmetic or logical expression.

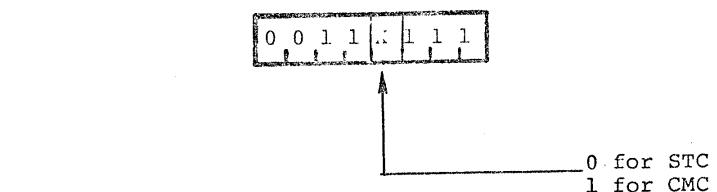
Description: The value of EXP specifies the number of memory bytes to be reserved for data storage. No data values are assembled into these bytes: in particular the programmer should not assume that they will be zero, or any other value. The next instruction will be assembled at memory location oplab+EXP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

Examples:

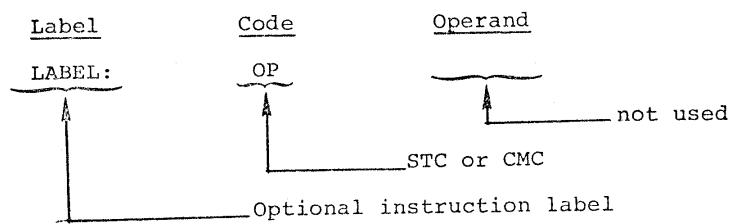
HERE:	DS	10	; Reserve the next 10 bytes
	DS	10H	; Reserve the next 16 bytes

3.3 CARRY BIT INSTRUCTIONS

This section describes the instructions which operate directly upon the carry bit. Instructions in this class occupy one byte as follows:



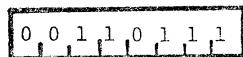
The general assembly language format is:



3.3.1 STC SET CARRY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	STC	---



Description: The carry bit is set to one.

Condition bits affected: Carry

3.3.2 CMC COMPLEMENT CARRY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	CMC	---

A rectangular box containing the binary sequence 00111111, representing the byte code for the CMC instruction.

Description: If the carry bit = 0, it is set to 1. If the carry bit = 1, it is reset to 0.

Condition bits affected: Carry

3.4 SINGLE REGISTER INSTRUCTIONS

This section describes instructions which operate on a single register or memory location. If a memory reference is specified, the memory byte addressed by the H and L registers is operated upon. The H register holds the most significant 8 bits of the address while the L register holds the least significant 8 bits of the address.

3.4.1 INR INCREMENT REGISTER OR MEMORY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	INR	reg B,C,D,E,H,L,M,or A



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

Description: The specified register or memory byte is incremented by one.

Condition bits affected: Zero, sign, parity, auxiliary carry

Example:

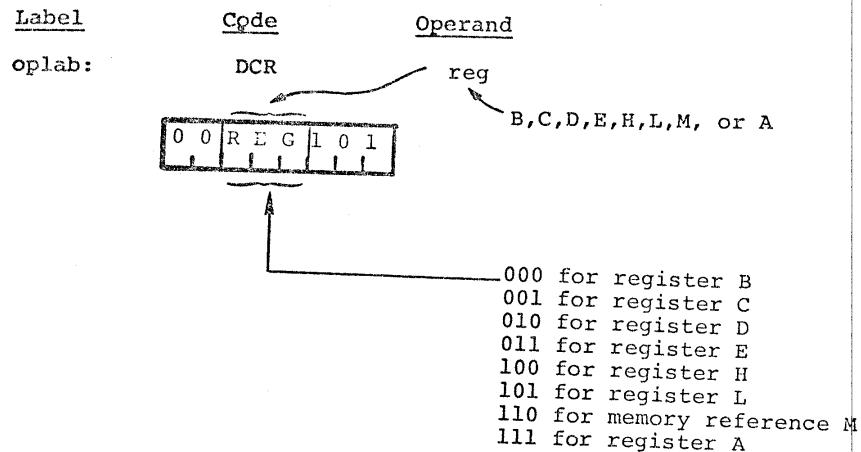
If register C contains 99H, the instruction:

INR C

will cause register C to contain 9AH.

3.4.2 DCR DECREMENT REGISTER OR MEMORY

Format:



Description: The specified register or memory byte is decremented by one.

Condition bits affected: Zero, sign, parity, auxiliary carry

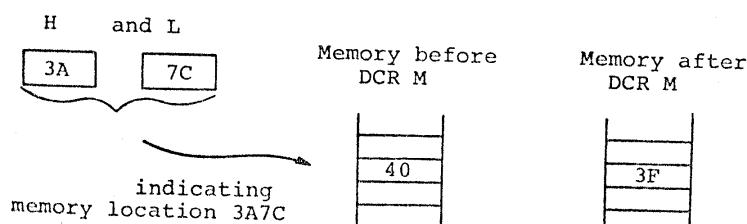
Example:

If the H register contains 3AH, the L register contains 7CH, and memory location 3A7CH contains 40H, the instruction:

DCR M

will cause memory location 3A7CH to contain 3FH. To illustrate:

DCR M references
registers



3.4.3 CMA COMPLEMENT ACCUMULATOR

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	CMA	---

0	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

Description: Each bit of the contents of the accumulator is complemented (producing the one's complement).

Condition bits affected: None

Example:

If the accumulator contains 51H, the instruction CMA will cause the accumulator to contain 0AEH.

Accumulator	= 0 1 0 1 0 0 0 1 = 51H
Accumulator	= 1 0 1 0 1 1 1 0 = AEH

3.4.4 DAA DECIMAL ADJUST ACCUMULATOR

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	DAA	---

0	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

Description: The eight-bit hexadecimal number in the accumulator is adjusted to form two four-bit binary-coded-decimal digits by the following two step process:

- (1) If the least significant four bits of the accumulator represent a number greater than 9, or if the auxiliary carry bit is equal to one, the accumulator is incremented by six. Otherwise, no incrementing occurs.
- (2) If the most significant four bits of the accumulator now represent a number greater than 9, or if the normal carry bit is equal to one, the most significant four bits of the accumulator are incremented by six. Otherwise, no incrementing occurs.

If a carry out of the least significant four bits occurs during Step (1), the auxiliary carry bit is set; otherwise it is unaffected. Likewise, if a carry out of the most significant four bits occurs during Step (2), the normal carry bit is set; otherwise, it is unaffected.

NOTE: This instruction is used when adding decimal numbers. It is the only instruction whose operation is affected by the auxiliary carry bit.

Condition bits affected: Zero, sign, parity, carry, auxiliary carry

Example:

Suppose the accumulator contains 9BH, and both carry bits = 0. The DAA instruction will operate as follows:

- (1) Since bits 0-3 are greater than 9, add 6 to the accumulator. This addition will generate a carry out of the lower four bits, setting the auxiliary carry bit.

Bit No: 7 6 5 4 3 2 1 0

$$\begin{array}{r} \text{Accumulator=} 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1 = 9\text{BH} \\ +6 = \\ \hline 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1 = A1\text{H} \end{array}$$

Auxiliary Carry = 1

- (2) Since bits 4-7 now are greater than 9, add 6 to these bits. This addition will generate a carry out of the upper four bits, setting the carry bit.

Bit No: 7 6 5 4 3 2 1 0

$$\begin{array}{r} \text{Accumulator=} 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1 = A1\text{H} \\ +6 = 0\ 1\ 1\ 0 \\ \hline 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \end{array}$$

Carry = 1

Thus, the accumulator will now contain 1, and both carry bits will be = 1.

For an example of decimal addition using the DAA instruction, see Section 5.5.

3.5 NOP INSTRUCTIONS

The NOP instruction occupies one byte.

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab	NOP	---

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Description: No operation occurs. Execution proceeds with the next sequential instruction.

Condition bits affected: None

3.6 DATA TRANSFER INSTRUCTIONS

This section describes instructions which transfer data between registers or between memory and registers.

Instructions in this class occupy one byte as follows:

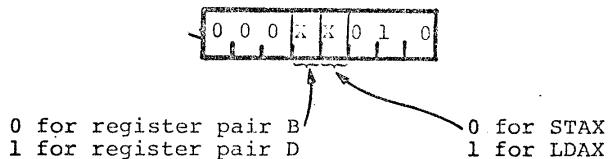
(a) For the MOV instruction:



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

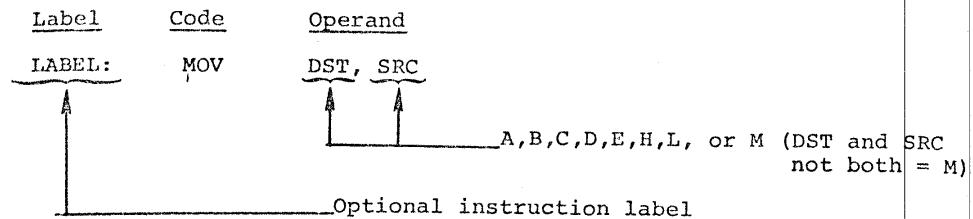
NOTE: DST and SRC cannot both = 110B

(b) For the remaining instructions:

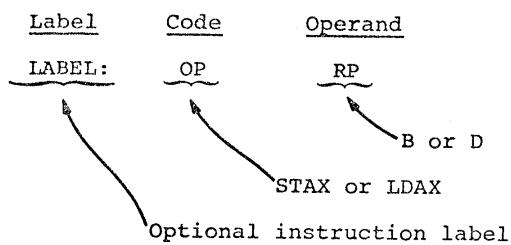


When a memory reference is specified in the MOV instruction, the addressed location is specified by the H and L registers. The L register holds the least significant 8 bits of the address; the H register holds the most significant 8 bits.

The general assembly language format is:

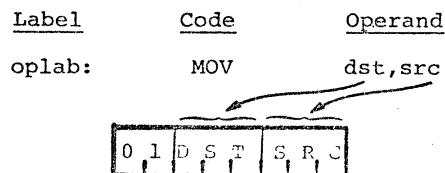


-or-



3.6.1 MOV INSTRUCTION

Format:



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.

Condition bits affected: None

Example 1:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	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 NOP (no-operation).

Example 2:

Assuming that the H register contains 2BH and the L register contains E9H, the instruction:

MOV M,A

will store the contents of the accumulator at memory location 2BE9H.

3.6.2 STAX STORE ACCUMULATOR

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	STAX	rp

Diagram of the operand field showing a 6-bit register number:

The diagram shows a horizontal row of six boxes, each containing a digit from 0 to 9. An arrow points from the 'rp' label above to the fourth box from the left, which contains an 'X'.

Description: The contents of the accumulator are stored in the memory location addressed by registers B and C, or by registers D and E.

Condition bits affected: None

Example:

If register B contains 3FH and register C contains 16H, the instruction:

STAX B

will store the contents of the accumulator at memory location 3F16H.

3.6.3 LDAX LOAD ACCUMULATOR

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LDAX	rp

↓


Description: The contents of the memory location addressed by registers B and C, or by registers D and E, replace the contents of the accumulator.

Condition bits affected: None

Example:

If register D contains 93H and register E contains 8BH, the instruction:

LDAX D

will load the accumulator from memory location 938BH.

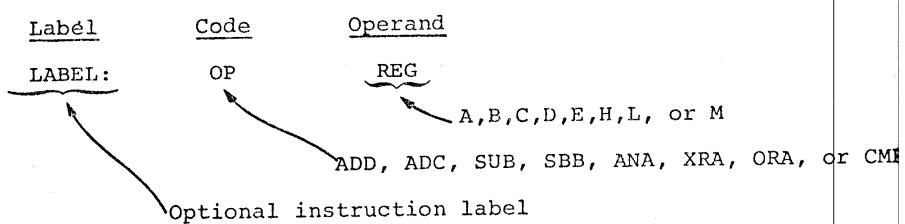
3.7 REGISTER OR MEMORY TO ACCUMULATOR INSTRUCTIONS

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

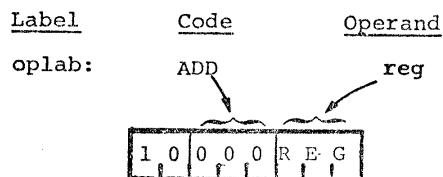
Instructions in this class operate on the accumulator using the byte in the register specified by REG. If a memory reference is specified, the instructions use the byte in the memory location addressed by registers H and L. The H register holds the most significant 8 bits of the address, while the L register holds the least significant 8 bits of the address. The specified byte will remain unchanged by any of the instructions in this class; the result will replace the contents of the accumulator.

The general assembly language instruction format is:



3.7.1 ADD ADD REGISTER OR MEMORY TO ACCUMULATOR

Format:



Description: The specified byte is added to the contents of the accumulator using two's complement arithmetic.

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

Example 1:

Assume that the D register contains 2EH and the accumulator contains 6CH. Then the instruction:

ADD D

will perform the addition as follows:

2EH = 00101110
6CH = 01101100
9AH = 10011010

The zero and carry bits are reset; the parity and sign bits are set. Since there is a carry out of bit A₃, the auxiliary carry bit is set. The accumulator now contains 9AH.

Example 2:

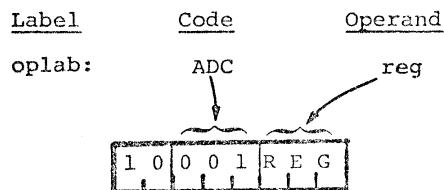
The instruction:

ADD A

will double the accumulator.

3.7.2 ADC ADD REGISTER OR MEMORY TO ACCUMULATOR WITH CARRY

Format:



Description: The specified byte plus the content of the carry bit is added to the contents of the accumulator.

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

Example:

Assume that register C contains 3DH, the accumulator contains 42H, and the carry bit = 0. The instruction:

ADC C

will perform the addition as follows:

$$\begin{array}{r} 3DH = 0\ 0\ 1\ 1\ 1\ 1\ 0\ 1 \\ 42H = 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0 \\ \text{CARRY} = \quad \quad \quad 0 \\ \text{RESULT} = \underline{0\ 1\ 1\ 1\ 1\ 1\ 1\ 1} = 7FH \end{array}$$

The results can be summarized as follows:

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

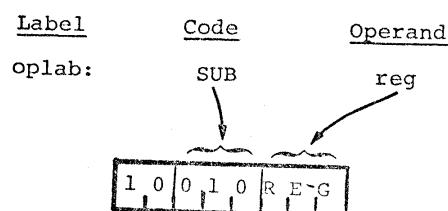
If the carry bit had been one at the beginning of the example, the following would have occurred:

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
Aux. Carry = 1

3.7.3 SUB SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR

Format:



Description: The specified byte is subtracted from the accumulator using two's complement arithmetic.

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

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

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:

$$\begin{array}{r} 3EH = 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0 \\ +(-3EH) = 1\ 1\ 0\ 0\ 0\ 0\ 1 \text{ negate and add one} \\ + \qquad \qquad \qquad 1 \text{ to produce two's complement} \\ \hline \text{Overflow} \longrightarrow 1] 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \text{ Result} = 0 \end{array}$$

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

Since there was an overflow out of bit A_3 , the auxiliary carry bit will be set.

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).

3.7.4 SBB SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	SBB	reg



Description: The carry bit is internally added to the contents of the specified byte. This value is then subtracted from the accumulator using two's complement arithmetic.

This instruction is most useful when performing 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 5.4.

Condition bits affected: Carry, sign, zero, parity, auxiliary carry (see Section 3.7.3 for details).

Example:

Assume that register L contains 2, the accumulator contains 4, and the carry bit = 1. Then the instruction SBB L will act as follows:

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

Adding this to the accumulator produces:

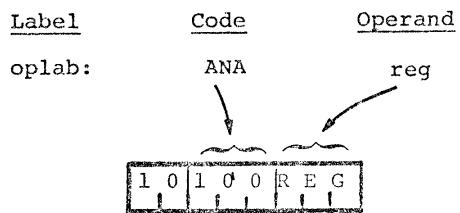
Accumulator = 04H = 0 0 0 0 0 1 0 0
1 1 1 1 1 1 0 1
1] 0 0 0 0 0 0 0 1 = 01H = Result

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 auxiliary carry bit is set since there was a overflow out of bit A₃. The parity and the sign bits are reset.

3.7.5 ANA LOGICAL AND REGISTER OR MEMORY WITH ACCUMULATOR

Format:



Description: The specified byte is logically ANDed bit by bit with the contents of the accumulator. The carry bit is reset to zero.

The logical AND function of two bits is 1 if and only if both the bits equal 1.

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.

Assuming that the accumulator contains 0FCH and the C register contains 0FH, the instruction:

ANA C

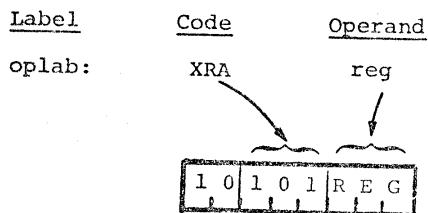
will act as follows:

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.

3.7.6 XRA LOGICAL EXCLUSIVE-OR REGISTER OR MEMORY WITH ACCUMULATOR (ZERO ACCUMULATOR)

Format:



Description: The specified byte is EXCLUSIVE-OREd bit by bit with the contents of the accumulator. The carry bit is reset to zero.

The EXCLUSIVE-OR function of two bits equals 1 if and only if the values of the bits are different.

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 zero the accumulator.

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	XRA	A
	MOV	B,A
	MOV	C,A

These instructions zero the A, B, and C registers.

Example 2:

Any bit EXCLUSIVE-ORED with a one is complemented (0 XOR 1 = 1, 1 XOR 1 = 0).

Therefore if the accumulator contains all ones (0FFH), the instruction:

XRA B

wil produce the one's complement of the B register in the accumulator.

Example 3:

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 determining which bits of a word have changed from one time to another.

<u>Label</u>	<u>Code</u>	<u>Operand</u>	
LA:	MOV	A,M	; STAT2 to accumulator
	INX	H	; 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. Also assume that the H and L registers have been initialized to address location 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	1	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 0 1 0 0
STAT1 = 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 is reset.

3.7.7 ORA LOGICAL OR REGISTER OR MEMORY WITH ACCUMULATOR

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	ORA	reg

Description: The specified byte is logically ORed bit by bit with the contents of the accumulator. The carry bit is reset to zero.

The logical OR function of two bits equals zero if and only if both the bits equal zero.

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.

Assuming that register C contains 0FH and the accumulator contains 33H, the instruction:

ORA C

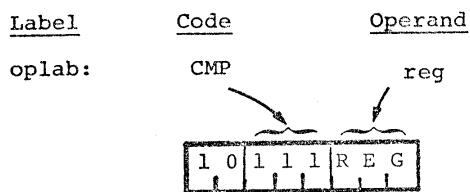
acts as follows:

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 guarantees that the low-order four bits of the accumulator are one, and the high-order four bits are unchanged.

3.7.8 CMP COMPARE REGISTER OR MEMORY WITH ACCUMULATOR

Format:



Description: The specified byte 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, auxiliary carry

Example 1:

Assume that the accumulator contains the number 0AH and the E register contains the number 05H. Then the instruction CMP E performs the following internal subtractions:

$$\begin{array}{r} \text{Accumulator} = 0AH = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0 \\ + (-E \text{ Register}) = -5H = 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \end{array} = \text{result}$$

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:

$$\begin{array}{r} \text{Accumulator} = 02H = 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\ + (-E \text{ Register}) = -5H = 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \\ \hline 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1 \end{array} = \text{result}$$

↓
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:

$$\begin{array}{r} \text{Accumulator} = -1BH = 1\ 1\ 1\ 0\ 0\ 1\ 0\ 1 \\ + (-E \text{ Register}) = -5H = 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \\ \hline 1\ 1\ 1\ 0\ 0\ 0\ 0 \end{array} = \text{result}$$

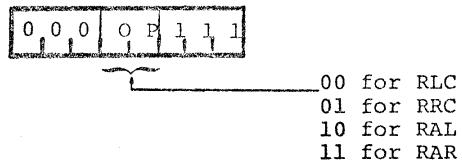
↓
overflow=1, causing carry to be reset

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

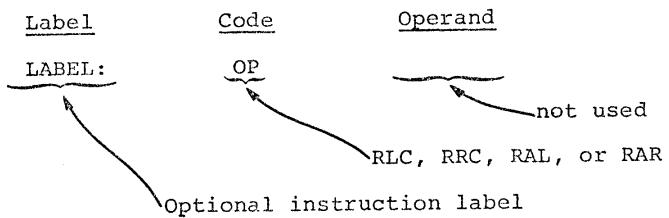
3.8 ROTATE ACCUMULATOR INSTRUCTIONS

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:

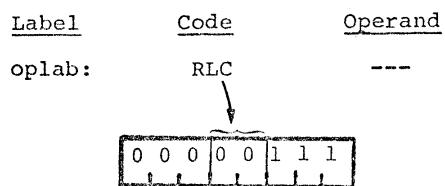


The general assembly language instruction format is:



3.8.1 RLC ROTATE ACCUMULATOR LEFT

Format:



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:

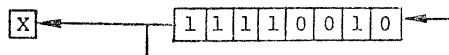
Assume that the accumulator contains 0F2H. Then the instruction:

RLC

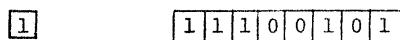
acts as follows:

Before RLC is executed: Carry

Accumulator



After RLC is executed:



Carry = 1 A = 0E5H

3.8.2 RRC ROTATE ACCUMULATOR RIGHT

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
--------------	-------------	----------------

oplab:	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

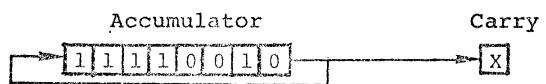
Example:

Assume that the accumulator contains 0F2H. Then the instruction:

RRC

acts as follows:

Before RRC is executed:



After RRC is executed:



3.8.3 RAL ROTATE ACCUMULATOR LEFT THROUGH CARRY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	RAL	---

↓

The diagram shows the state of the Accumulator after the RAL instruction. The Accumulator is represented by a horizontal box divided into 8 segments, each containing a binary digit. The digits from left to right are 0, 0, 0, 1, 0, 1, 1, 1. An arrow points to the fourth segment (the most significant bit).

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:

Assume that the accumulator contains 0B5H. Then the instruction:

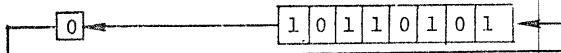
RAL

acts as follows:

Before RAL is executed:

Carry

Accumulator



After RAL is executed:

1

01101010

Carry = 1

A = 6AH

3.8.4 RAR ROTATE ACCUMULATOR RIGHT THROUGH CARRY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	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.

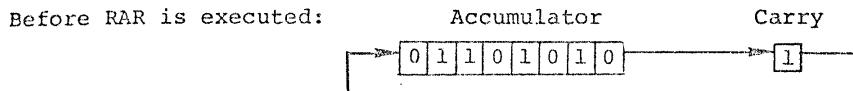
Condition bits affected: Carry

Example:

Assume that the accumulator contains 6AH. Then the instruction:

RAR

acts as follows:



After RAR is executed:

1 0 1 1 0 1 0 1	0
-----------------	---

A=0B5H

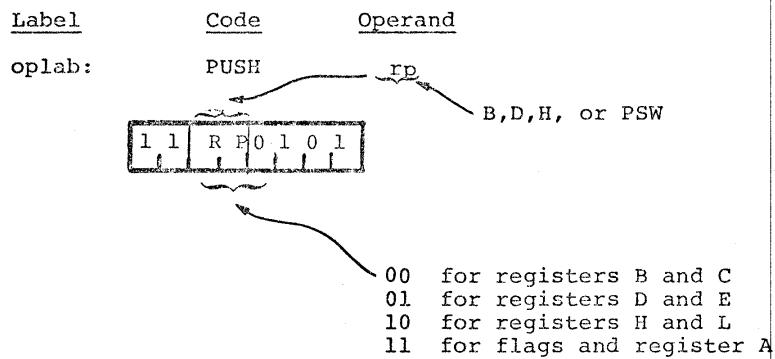
Carry=0

3.9 REGISTER PAIR INSTRUCTIONS

This section describes instructions which operate on pairs of registers.

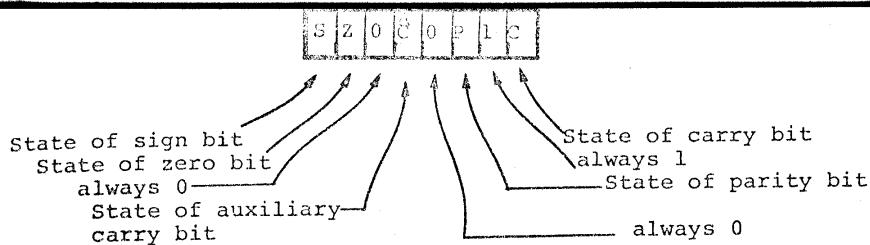
3.9.1 PUSH PUSH DATA ONTO STACK

Format:



Description: The contents of the specified register pair are saved in two bytes of memory indicated by the stack pointer SP.

The contents of the first register are saved at the memory address one less than the address indicated by the stack pointer; the contents of the second register are saved at the address two less than the address indicated by the stack pointer. If register pair PSW is specified, the first byte of information saved holds the settings of the five condition bits, i.e., carry, zero, sign, parity, and auxiliary carry. The format of this byte is:



In any case, after the data has been saved, the stack pointer is decremented by two.

Condition bits affected: None

Example 1:

Assume that register D contains 8FH, register E contains 9DH, and the stack pointer contains 3A2CH. Then the instruction:

PUSH D

stores the D register at memory address 3A2BH, stores the E register at memory address 3A2AH, and then decrements the stack pointer by two, leaving the stack pointer equal to 3A2AH.

Before PUSH		After PUSH	
		HEX	
SP	3A2C	MEMORY	3A29
			FF
			FF
D	8F	ADDRESS	3A2A
			3A2B
			3A2C
E	9D	MEMORY	FF
			9D
			8F
			9D

Example 2:

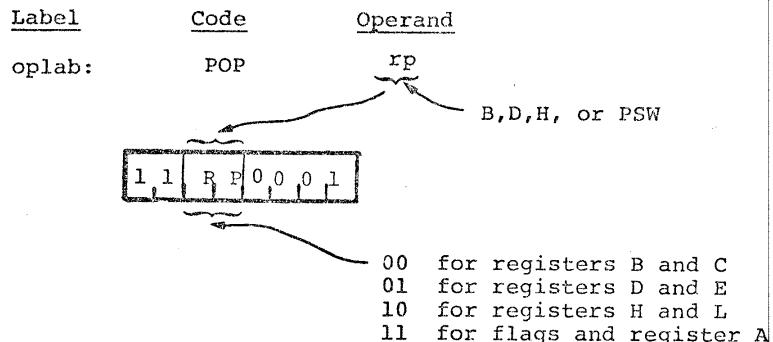
Assume that the accumulator contains 1FH, the stack pointer contains 502AH, the carry, zero and parity bits all equal 1, and the sign and auxiliary carry bits all equal 0. Then the instruction:

PUSH PSW

stores the accumulator (1FH) at location 5028H, stores the value 47H, corresponding to the flag settings at location 5029H, and decrements the stack pointer to the value 5028H.

3.9.2 POP POP DATA OFF STACK

Format:



Description: The contents of the specified register pair are restored from two bytes of memory indicated by the stack pointer SP. The byte of data at the memory address indicated by the stack pointer is loaded into the second register of the register pair; the byte of data at the address one greater than the address indicated by the stack pointer is loaded into the first register of the pair. If register pair PSW is specified, the byte of data indicated by the contents of the stack pointer plus one is used to restore the values of the five condition bits (carry, zero, sign, parity, and auxiliary carry) using the format described in Section 3.9.1.

In any case, after the data has been restored, the stack pointer is incremented by two.

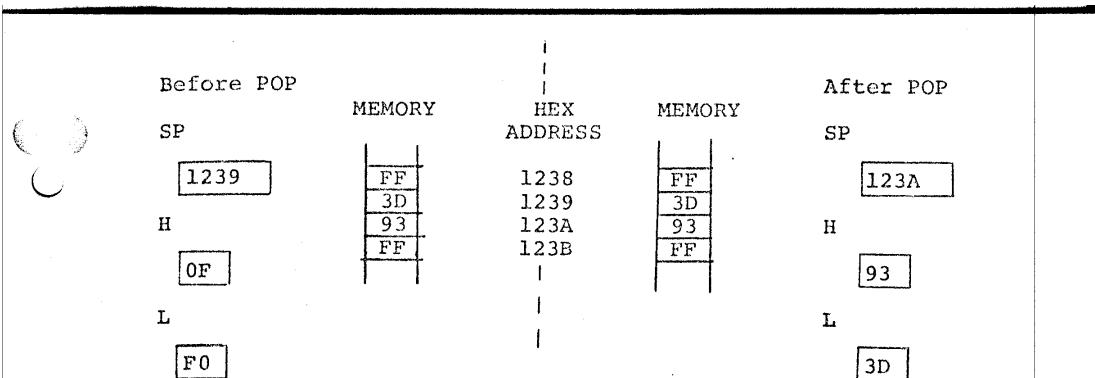
Condition bits affected: If register pair PSW is specified, carry, sign, zero, parity, and auxiliary carry may be changed. Otherwise, none are affected.

Example 1:

Assume that memory locations 1239H and 123AH contain 3DH and 93H, respectively, and that the stack pointer contains 1239H. Then the instruction:

POP H

loads register L with the value 3DH from location 1239H, loads register H with the value 93H from location 123AH, and increments the stack pointer by two, leaving it equal to 123AH.

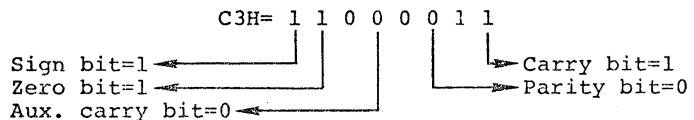


Example 2:

Assume that memory locations 2C00H and 2C01H contain FFH and C3H respectively, and that the stack pointer contains 2C00H. Then the instruction:

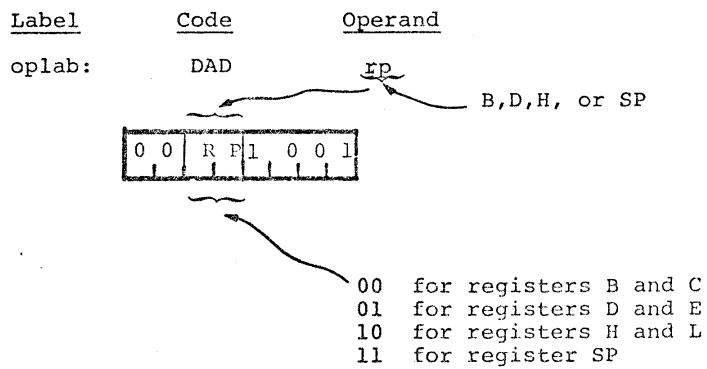
POP PSW

will load the accumulator with FFH and set the condition bits as follows:



3.9.3 DAD DOUBLE ADD

Format:



Description: The 16-bit number in the specified register pair is added to the 16-bit number held in the H and L registers using two's complement arithmetic. The result replaces the contents of the H and L registers.

Condition bits affected: Carry

Example 1:

Assume that register B contains 33H, register C contains 9FH, register H contains A1H, and register L contains 7BH. Then the instruction:

DAD B

performs the following addition:

$$\begin{array}{r} \text{Registers B and C} = 339F \\ + \text{Registers H and L} = A17B \\ \hline \text{New contents of H and L} = D51A \end{array}$$

Register H now contains D5H and register L now contains 1AH. Since no carry was produced, the carry bit is reset = 0.

Example 2:

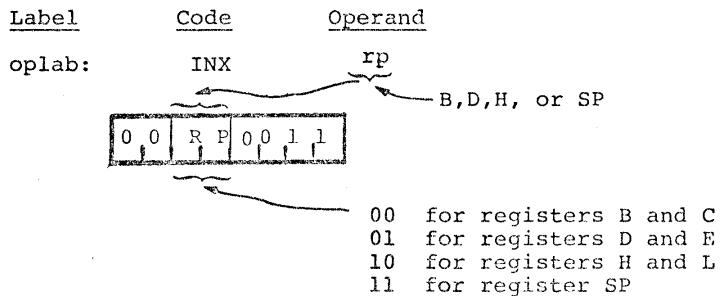
The instruction:

DAD H

will double the 16-bit number in the H and L registers (which is equivalent to shifting the 16 bits one position to the left).

3.9.4 INX INCREMENT REGISTER PAIR

Format:



Description: The 16-bit number held in the specified register pair is incremented by one.

Condition bits affected: None

Example:

If registers D and E contain 38H and FFH respectively, the instruction:

INX D

will cause register D to contain 39H and register E to contain 00H.

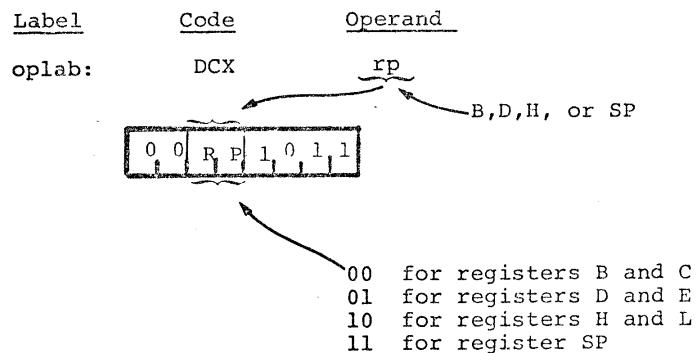
If the stack pointer SP contains FFFFH, the instruction:

INX SP

will cause register SP to contain 0000H.

3.9.5 DCX DECREMENT REGISTER PAIR

Format:



Description: The 16-bit number held in the specified register pair is decremented by one.

Condition bits affected: None

Example:

If register H contains 98H and register L contains 00H, the instruction:

DCX H

will cause register H to contain 97H and register L to contain FFH.

3.9.6 XCHG EXCHANGE REGISTERS

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	XCHG	---

1	1	1	0	1	0	1	1
---	---	---	---	---	---	---	---

Description: The 16 bits of data held in the H and L registers are exchanged with the 16 bits of data held in the D and E registers.

Condition bits affected: None

Example:

If register H contains 00H, register L contains FFH, register D contains 33H and register E contains 55H, the instruction XCHG will perform the following operation:

Before XCHG			After XCHG	
D	E		D	E
33	55		00	FF
H			L	
00	FF		33	55

3.9.7 XTHL EXCHANGE STACK

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	XTHL	---

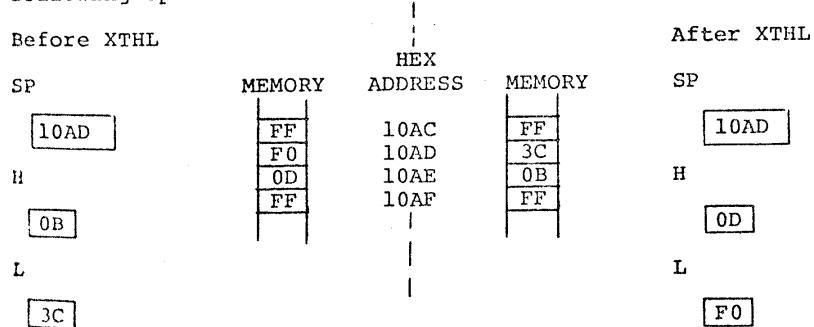
1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

Description: The contents of the L register are exchanged with the contents of the memory byte whose address is held in the stack pointer SP. The contents of the H register are exchanged with the contents of the memory byte whose address is one greater than that held in the stack pointer.

Condition bits affected: None

Example:

If register SP contains 10ADH, registers H and L contain 0BH and 3CH respectively, and memory locations 10ADH and 10AEH contain F0H and 0DH respectively, the instruction XTHL will perform the following operation:



3.9.8 SPHL LOAD SP FROM H AND L

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	SPHL	---

A rectangular box containing the binary code 11111001, representing the instruction code for SPHL.

Description: The 16 bits of data held in the H and L registers replace the contents of the stack pointer SP. The contents of the H and L registers are unchanged.

Condition bits affected: None

Example:

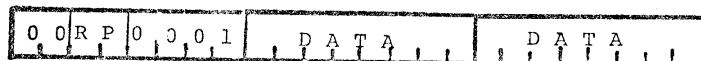
If registers H and L contain 50H and 6CH respectively, the instruction SPHL will load the stack pointer with the value 506CH.

3.10 IMMEDIATE INSTRUCTIONS

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

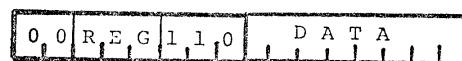
Instructions in this class occupy two or three bytes as follows:

- (a) For the LXI data instruction (3 bytes):



- 00 for registers B and C
- 01 for registers D and E
- 10 for registers H and L
- 11 for register SP

- (b) For the MVI data instruction (2 bytes):



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

- (c) For the remaining instructions (2 bytes):



- 000 for ADI
- 001 for ACI
- 010 for SUI
- 011 for SBI
- 100 for ANI
- 101 for XRI
- 110 for ORI
- 111 for CPI

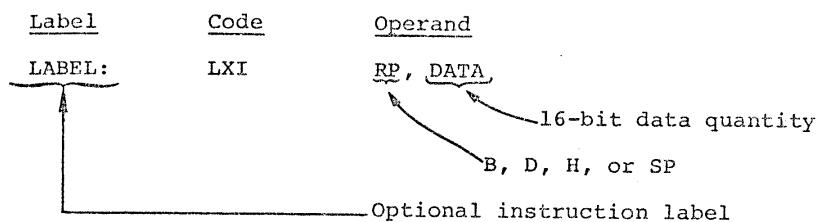
The LXI instruction operates on the register pair specified by RP using two bytes of immediate data.

The MVI instruction operates on the register specified by REG using one byte of immediate data. If a memory reference is specified, the instruction operates on the memory location

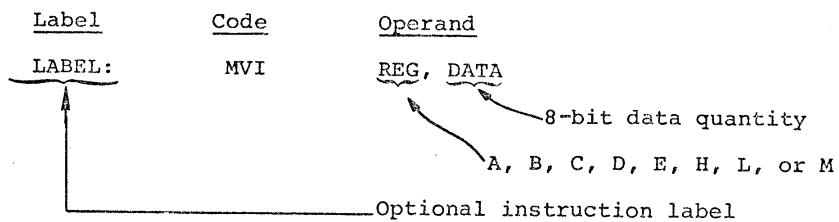
addressed by registers H and L. The H register holds the most significant 8 bits of the address, while the L register holds the least significant 8 bits of the address.

The remaining instructions in this class operate on the accumulator using one byte of immediate data. The result replaces the contents of the accumulator.

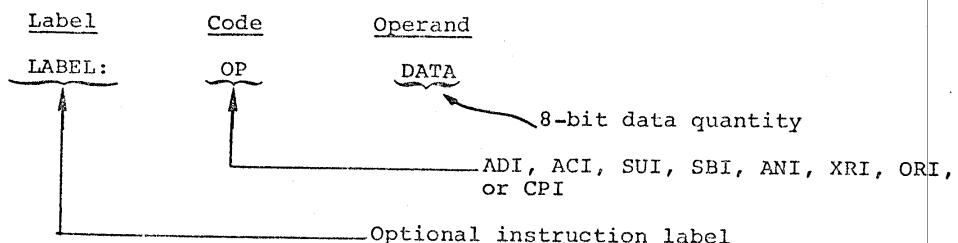
The general assembly language instruction format is:



-or-

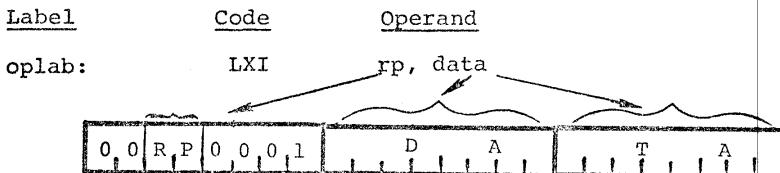


-or-



3.10.1 LXI LOAD REGISTER PAIR IMMEDIATE

Format:



Description: The third byte of the instruction (the most significant 8 bits of the 16-bit immediate data) is loaded into the first register of the specified pair, while the second byte of the instruction (the least significant 8 bits of the 16-bit immediate data) is loaded into the second register of the specified pair. If SP is specified as the register pair, the second byte of the instruction replaces the least significant 8 bits of the stack pointer, while the third byte of the instruction replaces the most significant 8 bits of the stack pointer.

Condition bits affected: None

NOTE: The immediate data for this instruction is a 16-bit quantity
All other immediate instructions require an 8-bit data value

Example 1:

Assume that instruction label STRT refers to memory location 103H (=259). Then the following instructions will each load the H register with 01H and the L register with 03H:

```
LXI H,103H  
LXI H,259  
LXI H,STRT
```

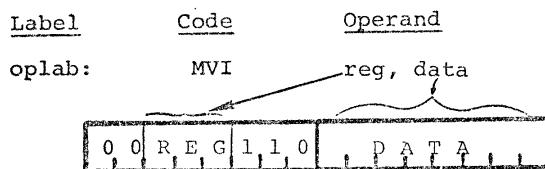
Example 2:

The following instruction loads the stack pointer with the value 3ABCH:

```
LXI SP,3ABCH
```

3.10.2 MVI MOVE IMMEDIATE DATA

Format:



Description: The byte of immediate data is stored in the specified register or memory byte.

Condition bits affected: None

EXAMPLE

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
M1:	MVI	H, 3CH	26EC
M2:	MVI	L, 0F4H	2EF4
M3:	MVI	M, OFFH	36FF

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

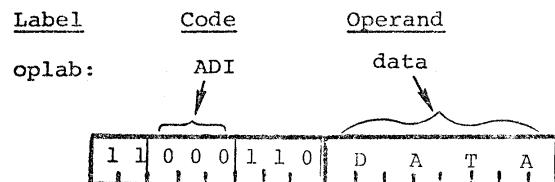
Likewise, the instruction at M2 loads the L register with 0F4H. The instruction at M3 causes the data at M3 + 1 (OFFH) to be stored at memory location 3CF4H. The memory location is obtained by concatenating the contents of the H and L registers into a 16-bit address.

NOTE: The instructions at M1 and M2 above could be replaced by the single instruction:

LXI H, 3CF4H

3.10.3 ADI ADD IMMEDIATE TO ACCUMULATOR

Format:



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

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
AD1:	MVI	A, 20	3E14
AD2:	ADI	66	C642
AD3:	ADI	-66	C6BE

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.

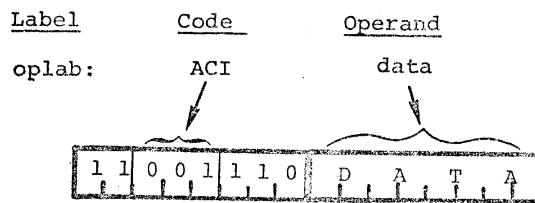
The instruction at AD3 restores the original contents of the accumulator by performing the following addition:

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

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

3.10.4 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.

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
C1:	MVI	A, 56H	3E56
C2:	ACI	-66	CEBE
C3:	ACI	66	CE42

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 3.10.3.

That is:

Accumulator = 14H
Carry = 1

The instruction at C3 then performs the following addition:

Accumulator = 14H = 00010100
C3 Immediate Data = 42H = 01000010
Carry bit = 1 = 1
Result = 01010111 = 57H

3.10.5 SUI SUBTRACT IMMEDIATE FROM ACCUMULATOR

Format:

Label Code Operand

oplab:

SUI

data



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

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

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

Example:

This instruction can be used as the equivalent of the DCR instruction.

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
S1:	MVI SUI	A, 0 1	3E00 D501

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

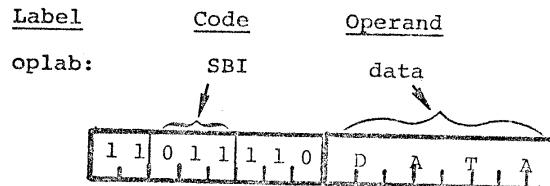
Accumulator = 0H = 00000000
-S1 Immediate Data = -1H = 11111111 two's complement
Result = 11111111 = -1H

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

The zero and auxiliary carry bits are also reset, while the sign and parity bits are set.

3.10.6 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.

This instruction and the SBB instruction are most useful when performing multibyte subtractions. For an example of this, see Section 5.4.

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.

Condition bits affected: Carry, sign, zero, parity, auxiliary carry

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
XRA	A		AF
SBI	1		DE01

The XRA instruction will zero the accumulator (see example in Section 3.7.6). If the carry bit is zero, the SBI instruction will then perform the following operation:

Immediate Data + Carry = 01H
Two's Complement of 01H = 11111111

Adding this to the accumulator produces:

Accumulator = 0H = 00000000
11111111
11111111 = -1H = Result
overflow = 0 causing carry to be set

The carry bit is set, indicating a borrow. The zero and auxiliary carry bits are reset, while the sign and parity bits are set.

If, however, 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:

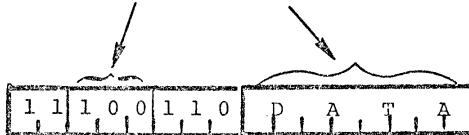
Accumulator = 0H = 00000000
11111110
11111110 = -2H = Result
overflow = 0 causing carry to be set

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

3.10.7 ANI AND IMMEDIATE WITH ACCUMULATOR

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	ANI	data



Description: The byte of immediate data is logically ANDed with the contents of the accumulator. The carry bit is reset to zero.

Condition bits affected: Carry, zero, sign, parity

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
A1:	MOV ANI	A, C OFH	79 E60F

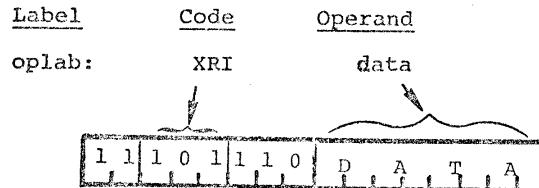
The contents of the C register are moved to the accumulator. The ANI instruction then zeroes the high-order four bits, leaving the low-order four bits unchanged. The zero bit will be set if and only if the low-order four bits were originally zero.

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

Accumulator = 3AH = 00111010
AND(A1 Immediate Data) = OFH = 00001111
Result = 00001010 = 0AH

3.10.8 XRI EXCLUSIVE-OR IMMEDIATE WITH ACCUMULATOR

Format:



Description: The byte of immediate data is EXCLUSIVE-ORed with the contents of the accumulator. The carry bit is set to zero.

Condition bits affected: Carry, zero, sign, parity

Example:

Since any bit EXCLUSIVE-ORed with a one is complemented, and any bit EXCLUSIVE-ORed with a zero is unchanged, this instruction can be used to complement specific bits of the accumulator. For instance, the instruction:

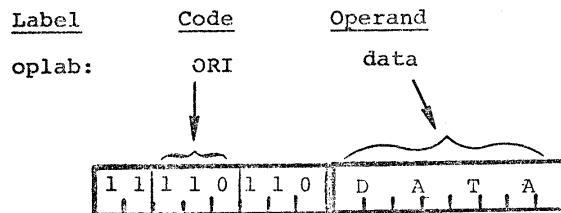
XRI 81H

will complement the least and most significant bits of the accumulator, leaving the rest unchanged. If the accumulator contained 3BH, the process would work as follows:

Accumulator = 3BH = 00111011	
XRI Immediate data = 81H = 10000001	
Result = 10111010	

3.10.9 ORI OR IMMEDIATE WITH ACCUMULATOR

Format:



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 reset to zero, while the zero, sign, and parity bits are set according to the result.

Condition bits affected: Carry, zero, sign, parity

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembly Data</u>
OR1:	MOV ORI	A,C 0FH	79 F60F

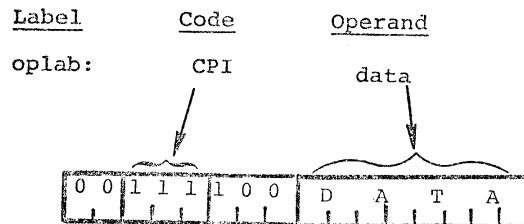
The contents of the C register are moved to the accumulator. The ORI instruction then sets the low-order four bits to one, leaving the high-order four bits unchanged.

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

Accumulator = 0B5H = 10110101
OR(ORI Immediate data) = 0FH = 00001111
Result = 10111111 = 0BFH

3.10.10 CPI COMPARE IMMEDIATE WITH ACCUMULATOR

Format:



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 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, auxiliary carry

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
MVI		A, 4AH	3E4A
CPI		40H	FE40

The CPI instruction performs the following operation:

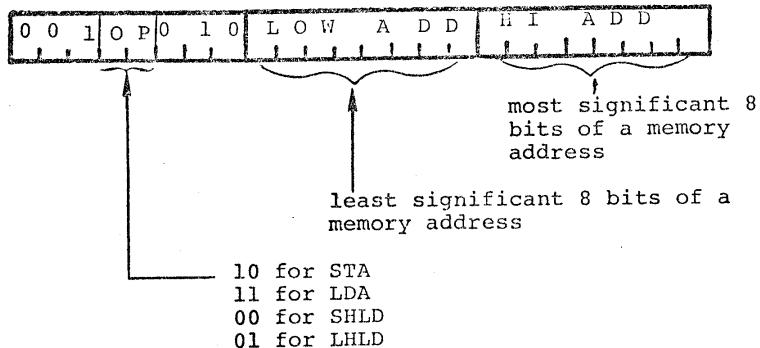
$$\begin{array}{r} \text{Accumulator} = 4AH = 01001010 \\ + (-\text{Immediate data}) = -40H = 11000000 \\ \hline 1] 00001010 = \text{Result} \end{array}$$

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 is less than the accumulator.

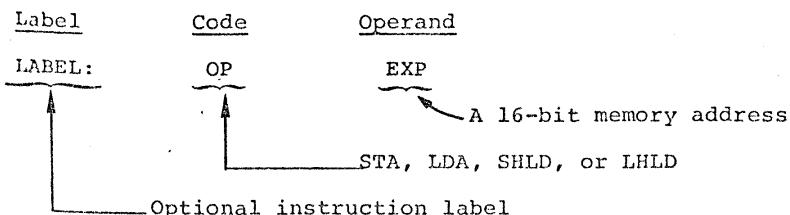
3.11 DIRECT ADDRESSING INSTRUCTIONS

This section describes instructions which reference memory by a two-byte address which is part of the instruction itself. Instructions in this class occupy three bytes as follows:



Note that the address is held least significant byte first.

The general assembly language format is:

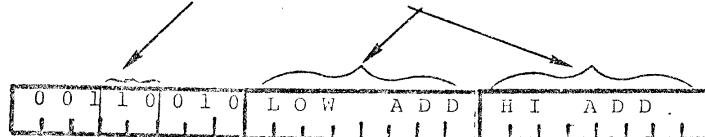


3.11.1 STA STORE ACCUMULATOR DIRECT

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
--------------	-------------	----------------

oplab: STA adr



Description: The contents of the accumulator replace the byte at the memory address formed by concatenating HI ADD with LOW ADD.

Condition bits affected: None

Example:

The following instructions will each store the contents of the accumulator at memory address 5B3H:

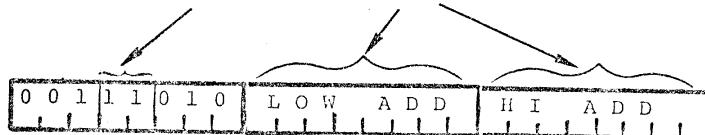
SAC: STA 5B3H
STA 1459
LAB: STA 010110110011B

3.11.2 LDA LOAD ACCUMULATOR DIRECT

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
--------------	-------------	----------------

oplab: LDA adr



Description: The byte at the memory address formed by concatenating HI ADD with LOW ADD replaces the contents of the accumulator.

Condition bits affected: None

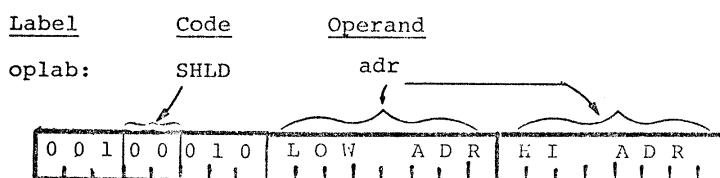
Example:

The following instructions will each replace the accumulator contents with the data held at location 300H:

LOAD: LDA 300H
 LDA 3*(16*16)
GET: LDA 200H+256

3.11.3 SHLD STORE H AND L DIRECT

Format:



Description: The contents of the L register are stored at the memory address formed by concatenating HI ADD with LOW ADD. The contents of the H register are stored at the next higher memory address.

Condition bits affected: None

Example:

If the H and L registers contain AEH and 29H respectively, the instruction:

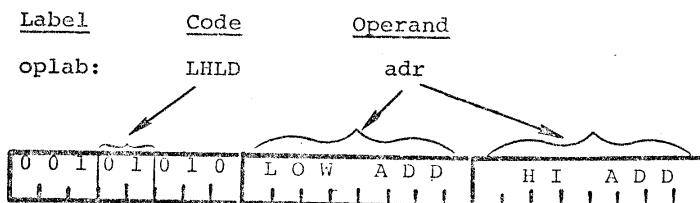
SHLD 10AH

will perform the following operation:

Memory Before SHLD		HEX ADDRESS	Memory After SHLD	
00			00	
00		10A	29	
00		10B	AE	
00		10C	00	

3.11.4 LHLD LOAD H AND L DIRECT

Format:



Description: The byte at the memory address formed by concatenating HI ADD with LOW ADD replaces the contents of the L register. The byte at the next higher memory address replaces the contents of the H register.

Condition bits affected: None

Example:

If memory locations 25BH and 25CH contain FFH and 03H respectively, the instruction:

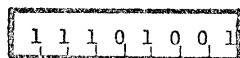
LHLD 25BH

will load the L register with FFH, and will load the H register with 03H.

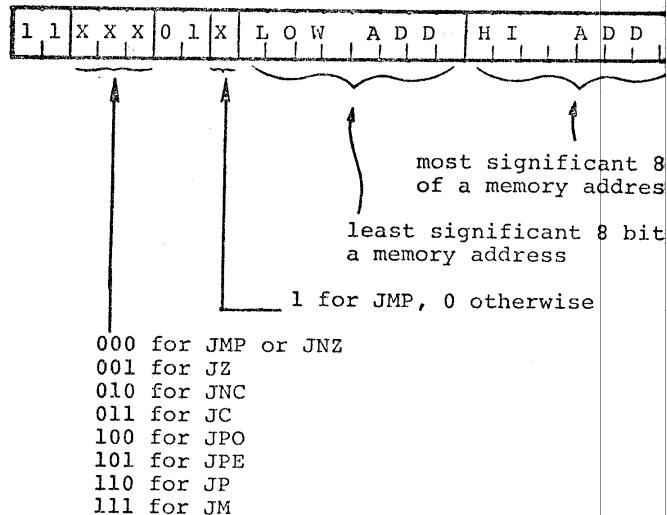
3.12 JUMP INSTRUCTIONS

This section describes instructions which alter the normal execution sequence of instructions. Instructions in this class occupy one or three bytes as follows:

- (a) For the PCHL instruction (one byte):



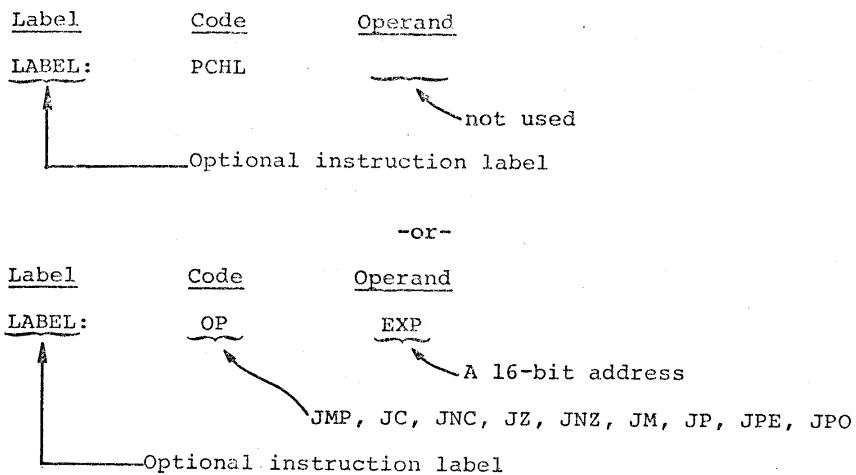
- (b) For the remaining instructions (three bytes):



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 three-byte instructions in this class cause a transfer of program control depending upon certain specified conditions. If the specified condition is true, program execution will continue at the memory address formed by concatenating the 8 bits of HI ADD (the third byte of the instruction) with the 8 bits of LOW ADD (the second byte of the instruction). If the specified condition is false, program execution will continue with the next sequential instruction.

The general assembly language format is:



3.12.1 PCHL LOAD PROGRAM COUNTER

Format:

Label Code Operand

oplab: PCHL ---

1	1	1	0	1	0	0	1
---	---	---	---	---	---	---	---

Description: The contents of the H register replace the most significant 8 bits of the program counter, and the contents of the L register replace the least significant 8 bits of the program counter. This causes program execution to continue at the address contained in the H and L registers.

Condition bits affected: None

Example 1:

If the H register contains 41H and the L register contains 3EH,
the instruction:

PCHL

will cause program execution to continue with the instruction
at memory address 413EH.

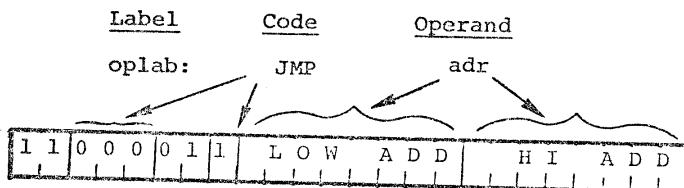
Example 2:

Arbitrary Memory Address	Label	Code	Operand	Assembled Data
40C0	ADR:	DW	LOC	0042
		.		.
		.		.
4100	STRT:	LHLD PCHL	ADR	2AC040 E9
		.		.
		.		.
4200	LOC:	NOP		00
		.		.
		.		.

Program execution begins at STRT. The LHLD instruction loads registers H and L from locations 40C1H and 40COH; that is, with 42H and 00H, respectively. The PCHL instruction then loads the program counter with 4200H, causing program execution to continue at location LOC.

3.12.2 JMP JUMP

Format:



Description: Program execution continues unconditionally at memory address adr.

Condition bits affected: None

Example:

Arbitrary Memory Address	Label	Code	Operand	Assembled Data
3C00		JMP	CLR	C3003E
3C03	AD:	ADI	2	C602
3D00		MVI	A, 3	3E03
3D02	LOAD:	JMP	3C03H	C3033C
3E00		XRA	A	AF
3E01	CLR:	JMP	\$-101H	C3003D

The execution sequence of this example is as follows:

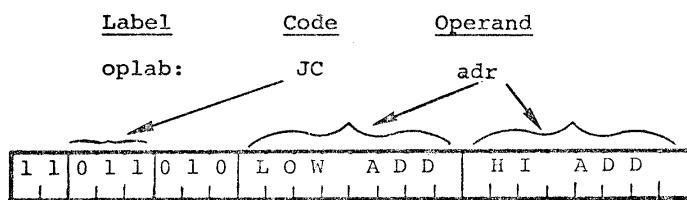
The JMP instruction at 3C00H replaces the contents of the program counter with 3E00H. The next instruction executed is the XRA at CLR, clearing the accumulator. The JMP at 3E01H is then executed.

The program counter is set to 3D00H, and the MVI at this address loads the accumulator with 3. The JMP at 3D02H sets the program counter to 3C03H, causing the ADI instruction to be executed.

- C From here, normal program execution continues with the instruction at 3C05H.

3.12.3 JC JUMP IF CARRY

Format:



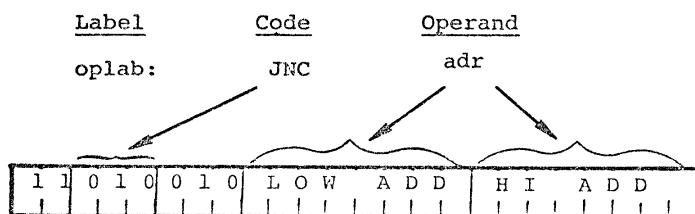
Description: If the carry bit is one, program execution continues at the memory address adr.

) Condition bits affected: None

- C For a programming example, see Section 3.12.10.

3.12.4 JNC JUMP IF NO CARRY

Format:



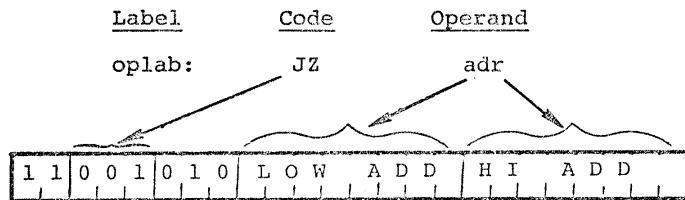
Description: If the carry bit is zero, program execution continues at the memory address adr.

Condition bits affected: None

For a programming example see Section 3.12.10.

3.12.5 JZ JUMP IF ZERO

Format:



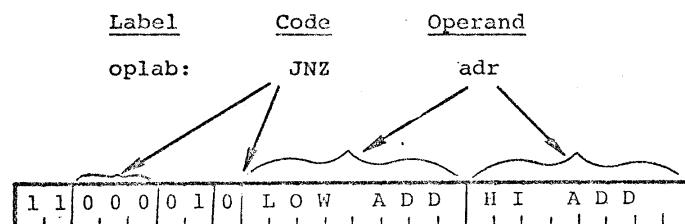
Description: If the zero bit is one, program execution continues at the memory address adr.

Condition bits affected: None

For a programming example, see Section 3.12.10.

3.12.6 JNZ JUMP IF NOT ZERO

Format:



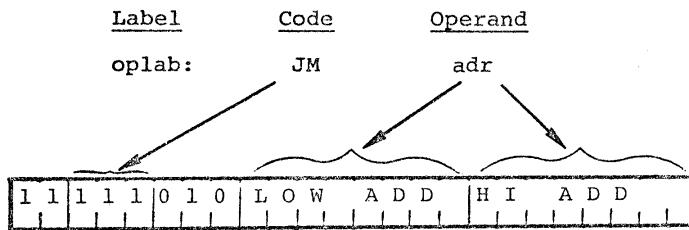
Description: If the zero bit is zero, program execution continues at the memory address adr.

Condition bits affected: None

For a programming example, see Section 3.12.10.

3.12.7 JM JUMP IF MINUS

Format:



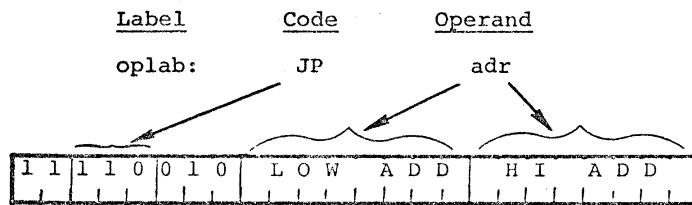
Description: If the sign bit is one (indicating a negative result), program execution continues at the memory address adr.

Condition bits affected: None

For a programming example, see Section 3.12.10.

3.12.8 JP JUMP IF POSITIVE

Format:



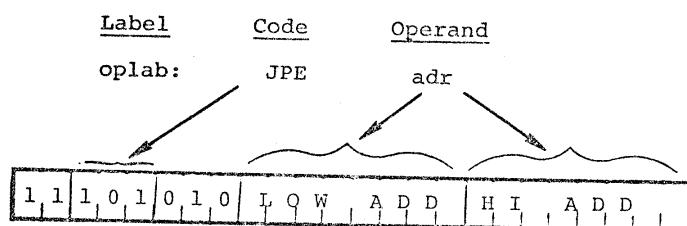
Description: If the sign bit is zero, (indicating a positive result), program execution continues at the memory address adr.

Condition bits affected: None

For a programming example, see Section 3.12.10.

3.12.9 JPE JUMP IF PARITY EVEN

Format:



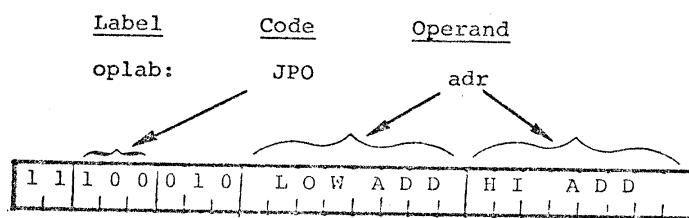
Description: If the parity bit is one (indicating a result with even parity), program execution continues at the memory address adr.

Condition bits affected: None

For a programming example, see Section 3.12.10.

3.12.10 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.

Condition bits affected: None

Examples of jump instructions:

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	79
	ANI	80H	E680
	JZ	PLUS	CAXXXX
	JNZ	MINUS	C2XXXX
TWO:	MOV	A,C	79
	RLC		07
	JNC	PLUS	D2XXXX
	JMP	MINUS	C3XXXX
THREE:	MOV	A,C ,	79
	ADI	0	C600
	JM	MINUS	FAXXXX
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 all jump instructions).

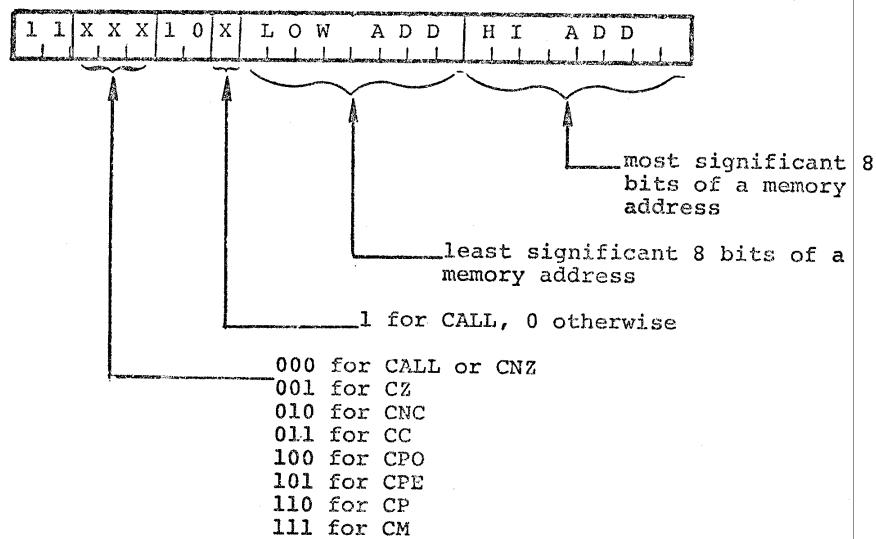
The RLC instruction in block TWO causes the carry bit to be set equal to the sign bit of the data byte. If the sign bit was reset, the JNC instruction causes a jump to PLUS. Otherwise the JMP instruction is executed, unconditionally transferring control to MINUS. (Note that, in this instance, a JC instruction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

3.13 CALL SUBROUTINE INSTRUCTIONS

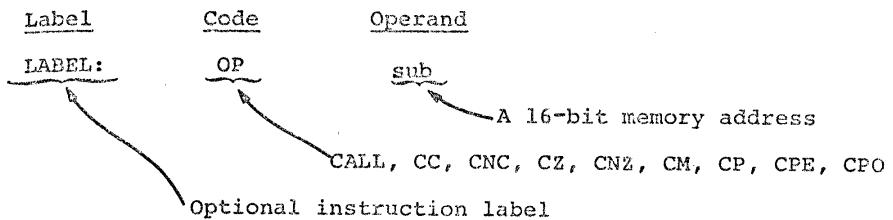
This section describes the instructions which call subroutines. These instructions operate like the jump instructions, causing a transfer of program control. In addition, a return address is pushed onto the stack for use by the RETURN instructions (Section 3.14).

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 call instructions.

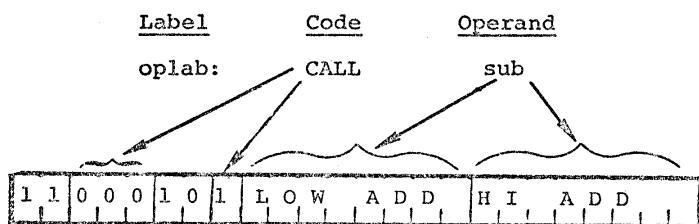
The general assembly language instruction format is:



Instructions in this class call subroutines upon certain specified conditions. If the specified condition is true, a return address is pushed onto the stack and program execution continues at memory address SUB, formed by concatenating the 8 bits of HI ADD with the 8 bits of LOW ADD. If the specified condition is false, program execution continues with the next sequential instruction.

3.13.1 CALL CALL

Format:



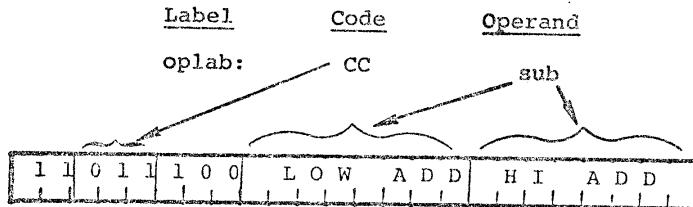
Description: A call operation is unconditionally performed to subroutine sub.

Condition bits affected: None

For programming examples see Section 5.

3.13.2 CC CALL IF CARRY

Format:



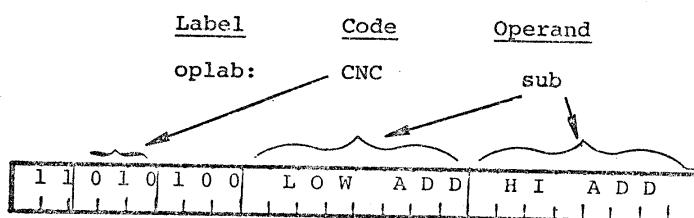
Description: If the carry bit is one, a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.3 CNC CALL IF NO CARRY

Format:



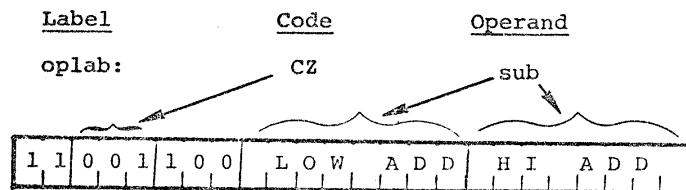
Description: If the carry bit is zero, a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.4 CZ CALL IF ZERO

Format:



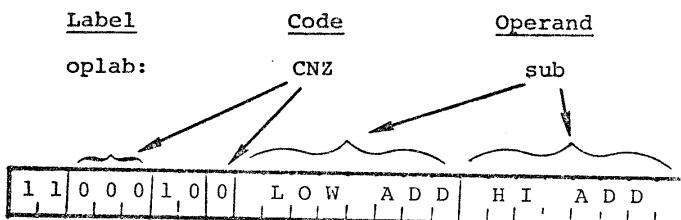
Description: If the zero bit is one, a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.5 CNZ CALL IF NOT ZERO

Format:



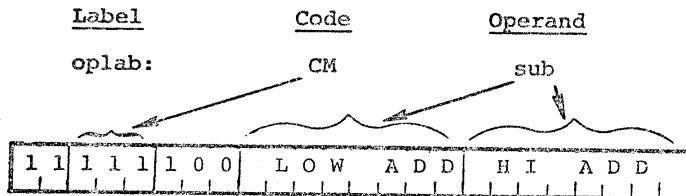
Description: If the zero bit is zero, a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.6 CM CALL IF MINUS

Format:



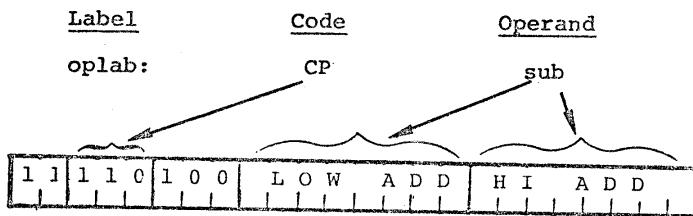
Description: If the sign bit is one (indicating a minus result), a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.7 CP CALL IF PLUS

Format:



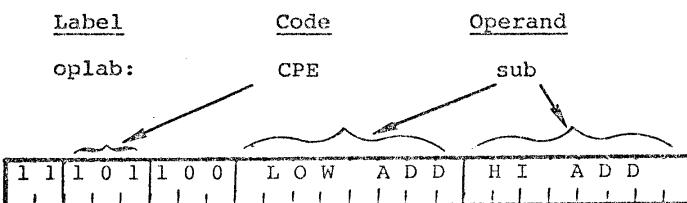
Description: If the sign bit is zero (indicating a positive result), a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.8 CPE CALL IF PARITY EVEN

Format:



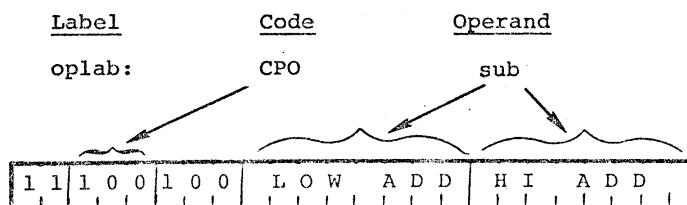
Description: If the parity bit is one (indicating even parity) a call operation is performed to subroutine sub.

Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.13.9 CPO CALL IF PARITY ODD

Format:



Description: If the parity bit is zero, (indicating odd parity), a call operation is performed to subroutine sub.

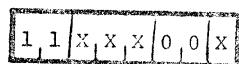
Condition bits affected: None

For programming examples using subroutines, see Section 5.

3.14 RETURN FROM SUBROUTINE INSTRUCTIONS

This section describes the instructions used to return from subroutines. These instructions pop the last address saved on the stack into the program counter, causing a transfer of program control to that address.

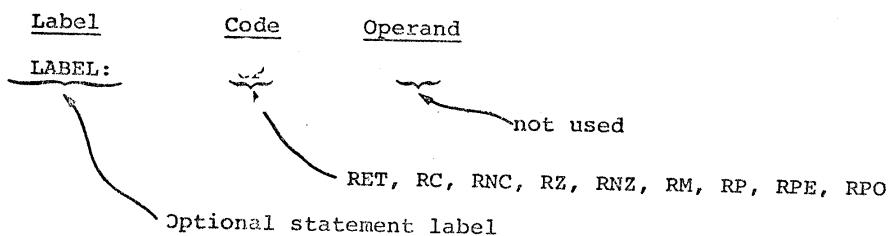
Instructions in this class occupy one byte as follows:



1 for RET, 0 otherwise

000 for RET or RNZ
001 for RZ
010 for RNC
011 for RC
100 for RPO
101 for RPE
110 for RP
111 for RM

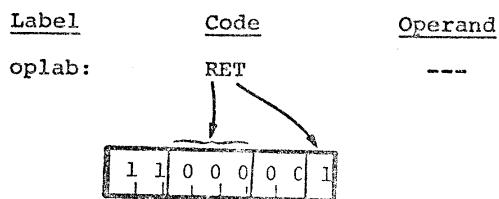
The general assembly language instruction format is:



Instructions in this class perform RETURN operations upon certain specified conditions. If the specified condition is true, a return operation is performed. Otherwise, program execution continues with the next sequential instruction.

3.14.1 RET RETURN

Format:



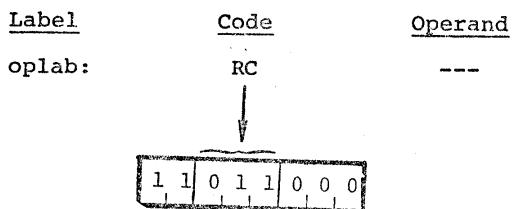
Description: A return operation is unconditionally performed.

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

Condition bits affected: None

3.14.2 RC RETURN IF CARRY

Format:



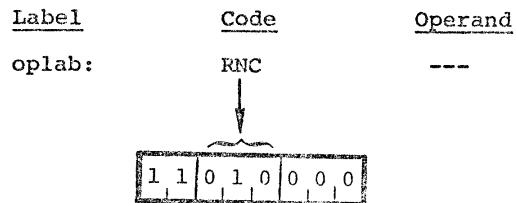
Description: If the carry bit is one, a return operation is performed.

Condition bits affected: None

For programming examples, see Section 5.

3.14.3 RNC RETURN IF NO CARRY

Format:



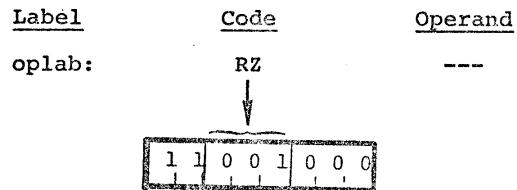
Description: If the carry bit is zero, a return operation is performed.

Condition bits affected: None

For programming examples, see Section 5.

3.14.4 RZ RETURN IF ZERO

Format:



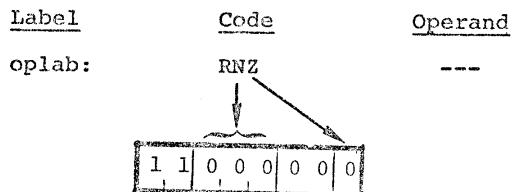
Description: If the zero bit is one, a return operation is performed.

Condition bits affected: None

For programming examples, see Section 5.

3.14.5 RNZ RETURN IF NOT ZERO

Format:



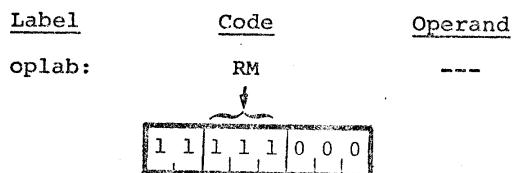
Description: If the zero bit is zero, a return operation is performed.

Condition bits affected: None

For programming examples, see Section 5.

3.14.6 RM RETURN IF MINUS

Format:



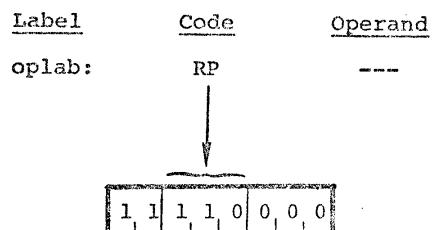
Description: If the sign bit is one (indicating a minus result), a return operation is performed.

Condition bits affected: None

For programming examples, see Section 5.

3.14.7 RP RETURN IF PLUS

Format:



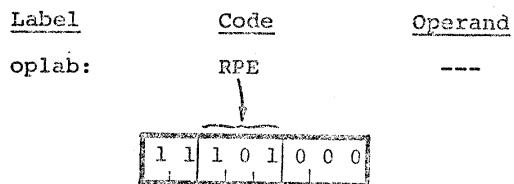
Description: If the sign bit is zero (indicating a positive result), a return operation is performed.

Condition bits affected: None

For programming examples, see Section 5.

3.14.8 RPE RETURN IF PARITY EVEN

Format:



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

Condition bits affected: None

For programming examples, see Section 5.

3.14.9 RPO RETURN IF PARITY ODD

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	RPO	---

↓

1	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---

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

Condition bits affected: None

For programming examples, see Section 5.

3.15 RST INSTRUCTION

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

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	RST	exp

↓

NOTE: "exp" must evaluate to a number in the range 000B to 111B.

Description: The contents of the program counter are pushed onto the 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 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

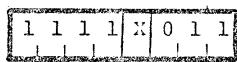
Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	RST	10 - 7	; Call the subroutine at ; address 24 (011000B)
	RST	E SHL 1	; Call the subroutine at ; address 48 (110000B). E 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.

3.16 INTERRUPT FLIP-FLOP INSTRUCTIONS

This section describes the instructions which operate directly upon the Interrupt Enable flip-flop INT_E. Instructions in this class occupy one byte as follows:



1 for EI
0 for DI

The general assembly language format is:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
LABEL:	OP	not used
		EI or DI
		Optional instruction label

3.16.1 EI ENABLE INTERRUPTS

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	EI	---

1	1	1	1	1	0	1	1

Description: This instruction sets the INTE flip-flop, enabling the CPU to recognize and respond to interrupts.

Condition bits affected: None

3.16.2 DI DISABLE INTERRUPTS

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	DI	---

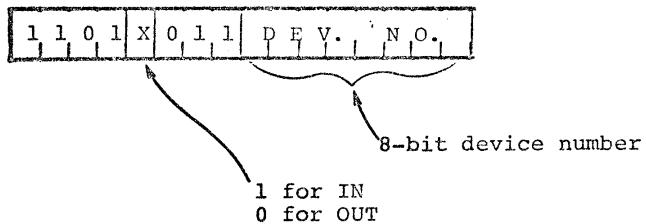
1	1	1	1	0	0	1	1

Description: This instruction resets the INTE flip-flop, causing the CPU to ignore all interrupts.

Condition bits affected: None

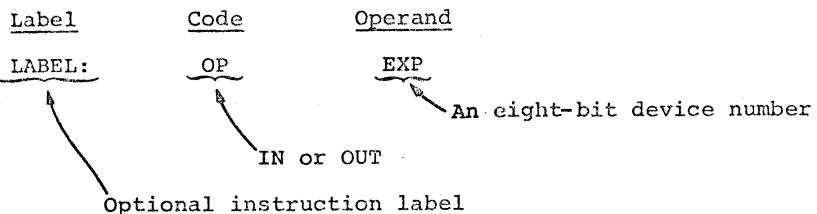
3.17 INPUT/OUTPUT INSTRUCTIONS

This section describes the instructions which cause data to be input to or output from the 8080. Instructions in this class occupy two bytes as follows:



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

The general assembly language format is:



3.17.1 IN INPUT

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	IN	exp

1	1	0	1	1	0	1	1	E	X	P
---	---	---	---	---	---	---	---	---	---	---

Description: An eight-bit data byte is read from input device number exp and replaces the contents of the accumulator.

Condition bits affected: None

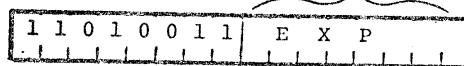
Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	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

3.17.2 OUT OUTPUT

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	OUT	exp



Description: The contents of the accumulator are sent to output device number exp.

Condition bits affected: None

Example:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	OUT	10	; Write the contents of the ; accumulator to output ; device # 10
	OUT	1FH	; Write the contents of the ; accumulator to output ; device # 31

3.18 HLT HALT INSTRUCTION

This section describes the HLT instruction, which occupies one byte.

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	HLT	 not used

0	1	1	1	0	1	1	0

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.

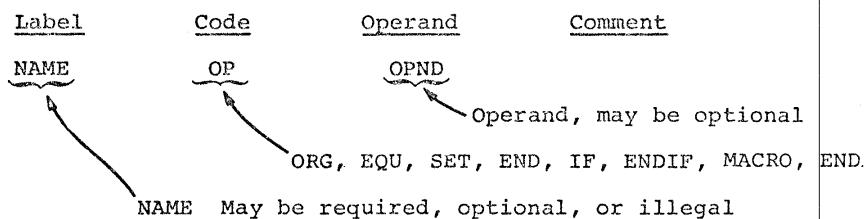
NOTE: If the interrupt system is disabled (INTE flip-flop = 0) and a HLT is executed, the 8080 must be powered down and then repowered to resume operation.

Condition bits affected: None

3.19 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 - 3.18, 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 psuedo-instruction is:



NOTE: Names on pseudo-instructions are not followed by a colon, as are labels. Names are required in the label field of MACRO, EQU, and SET pseudo-instructions. The label fields of the remaining pseudo-instructions may contain optional labels, exactly like the labels on machine instructions. In this case, the label refers to the memory location immediately following the last previously assembled machine instruction.

3.19.1 ORG ORIGIN

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	ORG	exp

A 16-bit address

Description: The assembler's location counter is set to the value of exp, which must be a valid 16-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:

<u>Hex Memory Address</u>	<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
1000		ORG	1000H	
1001		MOV	A,C	79
1003		ADI	2	C602
	HERE:	JMP	NEXT	C35010
1050	NEXT:	ORG	1050H	
		XRA	A	AF

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. The label HERE refers to memory location 1006H, since this is the address immediately following the jump instruction. Note that the range of memory from 1006H to 104FH 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.

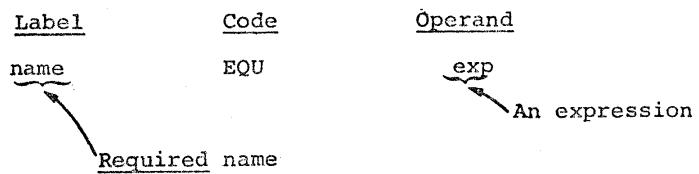
Example 2:

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	79
2C01		JMP	NEXT		JMP	NEXT	C3102C
2C04		DS	12		ORG	\$+12	
2C10	NEXT:	XRA	A	NEXT:	XRA	A	AF

3.19.2 EQU EQUATE

Format:



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 or only one EQU pseudo-instruction; i.e., an EQU symbol may not be redefined.

Example:

Label	Code	Operand	Assembled Data
PTO	EQU	8	
		•	
		•	
	OUT	PTO	D308

The OUT instruction in this example is equivalent to the statement:

OUT 8

If at some later time the programmer wanted the name PTO to refer to a different output port, it would be necessary only to change the EQU statement, not every OUT statement.

3.19.3 SET

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
name	SET	exp

An expression

Required name

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 equation, except that symbols may be defined more than once.

Example 1:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
IMMED	SET	5	
IMMED	ADI	IMMED	C605
	SET	10H-6	
	ADI	IMMED	C60A

Example 2:

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

<u>Label</u>	<u>Code</u>	<u>Operand</u>
B	SET	0
C	SET	1
D	SET	2
E	SET	3
H	SET	4
L	SET	5
M	SET	6
A	SET	7

If this were not done, a statement like:

MOV D,A

would be invalid, forcing the programmer to write:

MOV 2,7

3.19.4 END END OF ASSEMBLY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	END	---

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.

3.19.5 IF AND ENDIF CONDITIONAL ASSEMBLY

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
oplab:	IF	exp an expression
s t a t e m e n t s		
oplab:	ENDIF	---

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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
COND	SET	0FFH	
	IF	COND	
	MOV	A,C	79
	ENDIF		
COND	SET	0	
	IF	COND	
	MOV	A,C	
	ENDIF		
	XRA	C	A9

3.19.6 MACRO AND ENDM MACRO DEFINITION

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
name	MACRO	<u>list</u>
		A list of expressions, normally ASCII constants
<u>Required name</u>		
s t a t e m e n t s		
oplab:	ENDM	---

Description: For a detailed explanation of the definition and use of macros, 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 occurrences 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 to the right, while a zero is shifted into the high-order bit position. We will call this macro SHRT, and define it by writing the following instructions in the program:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
SHRT	MACRO	
	RRC	; Rotate accumulator right
	ANI	7FH ; Clear high-order bit
	ENDM	

We can now reference the macro by placing the following instructions later in the same program:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LDA	TEMP ; Load accumulator
	SHRT	

which would be equivalent to writing:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LDA	TEMP ; Load accumulator
	RRC	
	ANI	7FH

The example above illustrates the 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 name. Thus:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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 and only once in a program.

The reference is the point in a program where the macro is referenced. A macro may be referenced in any number of statements by inserting the macro name in the code field of the statements:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
LDA	TEMP	
SHRT		; Macro reference
STA	TEMP	

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

<u>Label</u>	<u>Code</u>	<u>Operand</u>
LDA	TEMP	
RRC		{ ; Macro reference
ANI	7FH	
STA	TEMP	

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.

Now consider a more complex case, a macro that shifts the accumulator right by a variable number of bit positions specified by the D register contents.

This macro is named SHV, and defined as follows:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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 then be referenced as follows:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LDA	TEMP
	MVI	D, 3 ; Specify 3 right shifts
	SHV	
	STA	TEMP

The above instruction sequence is equivalent to the expression:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LDA	TEMP
	MVI	D, 3
LOOP:	RRC	
	ANI	7FH
	DCR	D
	JNZ	LOOP
	STA	TEMP

Note that the D register contents will change whenever the SHV macro is referenced, 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 macro parameters. Such a macro is defined as follows:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LDA	TEMP
; Assume Register C is free, and a 5-place shift is needed		
	SHV	C, 5

the expansion of which is given by:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
LOOP:	MVI	C, 5
	RRC	
	ANI	7FH
	DCR	C
	JNZ	LOOP

Here is another example of an SHV reference:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
; Assume Register E is free, and a 2-place shift is needed,		
	SHV	E, 2

and the equivalent expansion:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
LOOP:	MVI	E, 2
	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 referred to 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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
name	MACRO	plist
m a c r o	b o d y	
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 3.19.6). 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. These expressions, which serve only to mark the positions where macro parameters are to be inserted into the macro body, are called dummy parameters.

Example:

The following macro takes the memory address of the label specified by the macro reference, loads the most significant 8 bits of the address into the C register and loads the least significant 8 bits of the address into the B register. (This is the opposite of what the instruction LXI B,ADDR would do).

<u>Label</u>	<u>Code</u>	<u>Operand</u>
LOAD	MACRO	ADDR
	MVI	C, ADDR SHR 8
	MVI	B, ADDR AND 0FFH
LABEL:	_____	
INST:	_____	

The reference:

<u>Code</u>	<u>Operand</u>
LOAD	LABEL

is equivalent to the expansion:

<u>Code</u>	<u>Operand</u>
MVI	C, LABEL SHR 8
MVI	B, LABEL AND OFFH

The reference:

<u>Code</u>	<u>Operand</u>
LOAD	INST

is equivalent to the expansion:

<u>Code</u>	<u>Operand</u>
MVI	C, INST SHR 8
MVI	B, INST AND OFFH

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 inserted into the statements at that point of the program and assembled.

4.2.2 MACRO REFERENCE OR CALL

Format:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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 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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
MAC1	MACRO	P1, P2, COMMENT
	XRA	P2
	DCR	P1 COMMENT
	ENDM	

The reference:

<u>Code</u>	<u>Operand</u>
MAC1	C, D, ; DECREMENT REG

is equivalent to the expansion:

<u>Code</u>	<u>Operand</u>
XRA	D
DCR	C ; DECREMENT REG

The reference:

<u>Code</u>	<u>Operand</u>
MAC1	E, B

is equivalent to the expansion:

<u>Code</u>	<u>Operand</u>
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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
MAC	MACRO	P1
	PUSH	P1
	ENDM	

the reference:

MAC B

will produce the legal expansion:

PUSH B

but the reference:

MAC C

will produce the illegal expansion:

PUSH C

which will be flagged as an error.

4.2.4 SCOPE OF LABELS AND NAMES WITHIN MACROS

In this section, the terms global and local are important. For our purposes, they will be defined as follows: A symbol is globally defined in a program if its value is known and can be referenced by any statement in the program, whether or not the statement was produced by the expansion of a macro. A symbol is locally defined if its value is known and can be referenced only within a particular macro expansion.

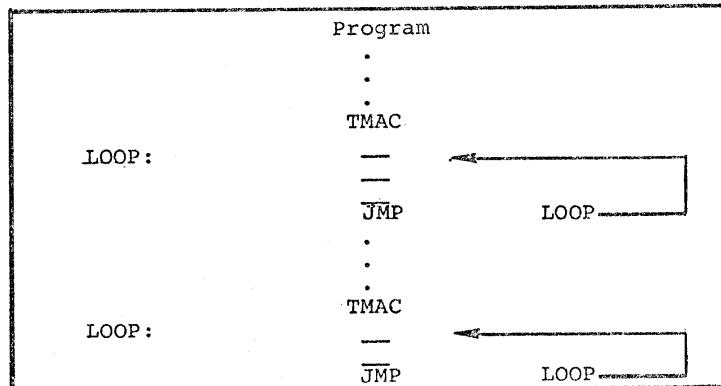
Instruction Labels: Normally a symbol may appear in the label field of only one instruction. If a label appears in the body of a macro, however, it will be generated whenever the macro is referenced. To avoid multiple-label conflicts, the assembler treats labels within macros as local labels, applying only to a particular expansion of a macro. Thus, each "jump to LOOP" instruction generated in the example of Section 4.1 refers uniquely to the label LOOP generated in the local macro expansion.

Conversely, if the programmer wishes to generate a global label from a macro expansion, he must follow the label with two colons in the macro definition, rather than one. Now, this global label must not be generated more than once, since it is global and therefore must be unique in the program.

For example, consider the macro definition:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
TMAC	MACRO	
LOOP:	—	
	—	
	—	
	JMP	LOOP
	ENDM	

If two references to TMAC appear in a program, the label LOOP will be a local label and each JMP LOOP instruction will refer to the label generated within its own expansion:



If in the macro definition, LOOP had been followed by two successi colons, LOOP would be generated as a global label by the first reference to TMAC, while the second reference would be flagged as an error.

"Equate" Names: Names on equate statements within a macro are always local, defined only within the expansion in which they are generated.

For example, consider the following macro definition:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
EQMAC	MACRO	
VAL	EQU	8
	DB	VAL
	ENDM	

The following program section is valid:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
VAL	EQU	6	
DB1:	DB	VAL	06
	EQMAC		
VAL	EQU	8	
	DB	VAL	08
DB2:	DB	VAL	06

VAL is first defined globally with a value of 6. Therefore the reference to VAL at DB1 produces a byte equal to 6. The macro reference EQMAC generates a symbol VAL defined only within the macro expansion with a value of 8; therefore the reference to VAL by the second statement of the macro produces a byte equal to 8. Since this statement ends the macro expansion, the reference to VAL at DB2 refers to the global definition of VAL. The statement at DB2 therefore produces a byte equal to 6.

"Set" Names: Suppose that a "set" statement is generated by a macro. If its name has already been defined globally by another set statement, the generated statement will change the global value of the name for all subsequent references. Otherwise, the name is defined locally, applying only within the current macro expansion. These cases are illustrated as follows:

Consider the macro definition:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
STMAC	MACRO	
SYM	SET	5
	DB	SYM
	ENDM	

The following program section is valid:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
SYM	SET	0	
DB1:	DB	SYM	00
	STMAC		
SYM	SET	5	
	DB	SYM	05
DB2:	DB	SYM	05

SYM is first defined globally with a value of zero, causing the reference at DB1 to produce a byte of 0. The macro reference STMAC resets this global value to 5, causing the second statement of the macro to produce a value of 5. Although this ends the macro expansion, the value of SYM remains equal to 5, as shown by the reference at DB2.

Using the same macro definition as above, the following program section is invalid:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Assembled Data</u>
	STMAC		
SYM	SET	5	
DB3:	DB	SYM	05
	DB	SYM	**ERROR**

Since in this case SYM is first defined in a macro expansion, its value is defined locally. Therefore the second (and final) statement of the macro expansion produces a byte equal to 5. The statement at DB3 is invalid, however, since SYM is unknown globally.

4.2.5 MACRO PARAMETER SUBSTITUTION

The value of macro parameters is determined and passed into the macro body at the time of the macro referenced, 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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
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 instead 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:

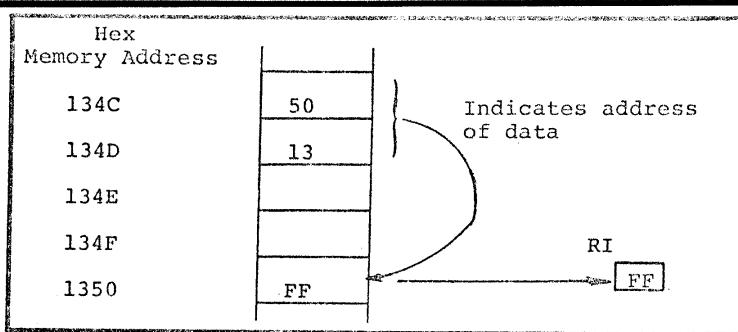
- (a) Often, a small group of instructions must be repeated many times throughout a program with only minor changes for each repetition.
Macros can reduce the tedium (and resultant increased chance for error) associated with these operations.
- (b) If an error in a macro definition is discovered, the 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 INDIRECT MACRO

The following macro, LIND, loads register RI indirect from memory location INADD.

That is, location INADD will be assumed to hold a two-byte memory address (least significant byte first) from which register RI will be loaded.



If the address of INADD is 134CH, register RI will be loaded from the address held in memory locations 134CH and 134DH, which is 1350H.

Macro definition:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
LIND	MACRO	RI, INADD	
	LHLD	INADD	; Load indirect address
			; into H and L registers
	MOV	RI, M	; Load data into RI
	ENDM		

Macro reference:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
; Load register C indirect with the contents of memory ; location LABEL.	LIND	C, LABEL

Macro expansion:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	LHLD	LABEL
	MOV	C, M

4.4.2 OTHER INDIRECT ADDRESSING MACROS

Refer to the LIND macro definition of Section 4.4.1. Only the 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 4.4.3.

4.4.3 CREATE INDEXED ADDRESS MACRO

The following macro, IXAD, loads registers H and L with the base address BSADD, plus the 16-bit index formed by register pair RP (RP=B,D,H, or SP).

Macro definition:

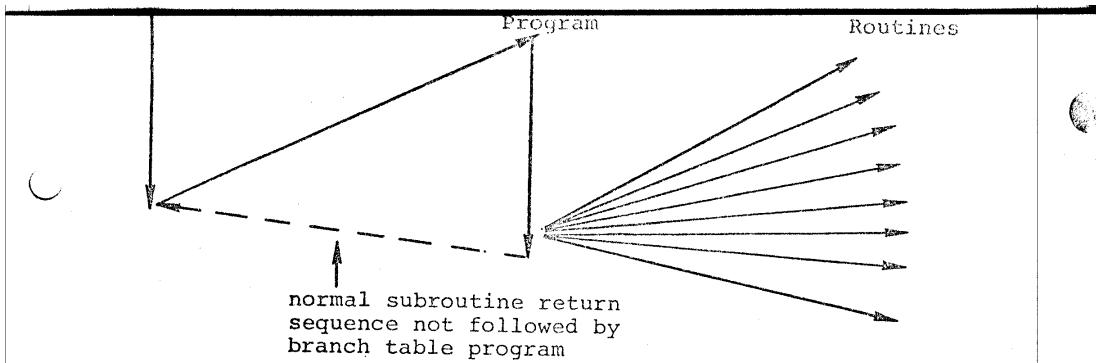
<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
IXAD	MACRO	RP, BSADD	
	LXI	H, BSADD	; Load the base address
	DAD	RP	; Add index to base address
	ENDM		

Macro reference:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
; 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

Macro expansion:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	MVI	D, 1
	MVI	E, 2EH
	LXI	H, BSADD
	DAD	D



<u>Label</u>	<u>Code</u>	<u>Operand</u>	
START:	LXI	H, BTBL	; Registers H and L will point to branch table.
GTBIT:	RAR		
	JC	GETAD	
	INX	H	; (H,L)=(H,L)+2 to point to
	INX	H	; next address in branch table.
GETAD:	JMP	GTBIT	
	MOV	E,M	; A one bit was found. Get
	INX	H	; address in D and E.
	MOV	D,M	
	XCHG		; Exchange D and E with H and L.
	PCHL		; Jump to routine address.
<hr/>			
BTBL:	DW	ROUT1	; Branch table. Each entry
	DW	ROUT2	; is a two-byte address
	DW	ROUT3	; held least significant
	DW	ROUT4	; byte first.
	DW	ROUT5	
	DW	ROUT6	
	DW	ROUT7	
	DW	ROUT8	

The control routine at START uses the H and L registers as a pointer into the branch table (BTBL) corresponding to the bit of the accumulator that is set. The routine at GETAD then transfers the address held in the corresponding branch table entry to the H and L registers via the D and E registers, and then uses a PCHL instruction, thus transferring control to the selected routine.

5.2 SUBROUTINES

Frequently, a group of instructions must be repeated many times in a program. As we have seen in Section 4, it is sometimes 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 stack, (see Section 2.4), 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 popped 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 16-bit number held least-significant byte first in two consecutive memory locations, and then returns to the instruction following the last CALL statement executed. The address of the number to be incremented is passed in the H and L registers.

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
MINC:	INR	M	; Increment low-order byte
	RNZ		; If non-zero, return to ; calling routine
	INX	H	; Address high-order byte
	INR	M	; Increment high-order byte
	RET		; Return unconditionally

Arbitrary
Memory Address

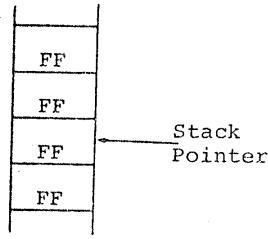
2C00 CALL MINC
2EF0 CALL MINC

Arbitrary
Memory Address

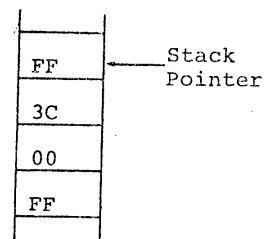
3C00 MINC

When the first call is executed, address 2C03H is pushed onto the stack indicated by the stack pointer, and control is transferred to 3C00H. Execution of either RETURN statement in MINC will cause the top entry to be popped off the stack into the program counter, causing execution to continue at 2C03H (since the CALL statement is three bytes long).

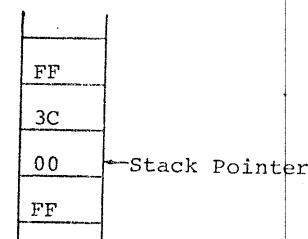
Stack Before
CALL



Stack While
MINC Executes



Stack After
RETURN is Performed



When the second call is executed, address 2EF3H is pushed onto the stack, and control is again transferred to MINC. This time, either RETURN instruction will cause execution to resume at 2EF3H.

Note that MINC could have called another subroutine during its execution, causing another address to be pushed onto the stack. This can occur as many times as necessary, limited only by the size of memory available for the stack.

Note also that any subroutine could push data onto the stack for temporary storage without affecting the call and return sequences as long as the same amount of data is popped off the stack before executing a RETURN statement.

5.2.1 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 5.2, for example, receives the memory address which it requires in the H and L registers.

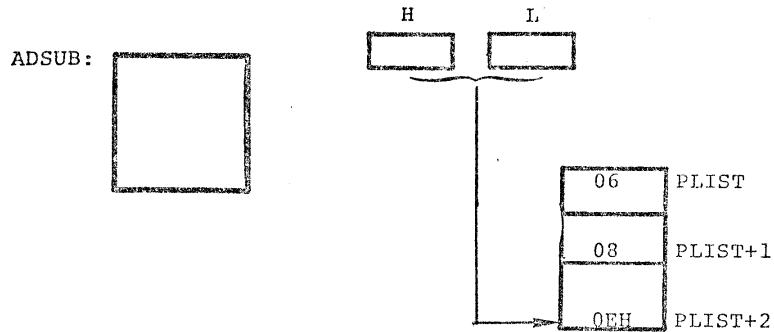
Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data (called a parameter list) in some data area of memory, and pass the address of this list to the subroutine in the H and L registers.

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:

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	LXI	H, PLIST	; Load H and L with addresses ; of the parameter list
	CALL	ADSUB	; Call the subroutine
RETL:	—		
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
	INX	H	; Increment memory address
	MOV	B, M	; Get second parameter
	ADD	B	; Add first to second
	INX	H	; Increment memory address
	MOV	M, A	; Store result at third parameter ; store
	RET		; Return unconditionally

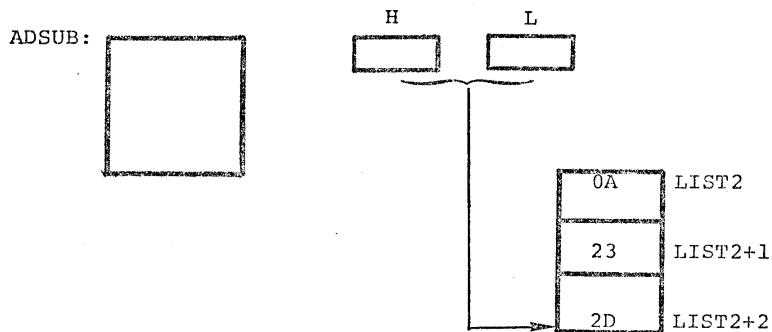
The first time ADSUB is called, it loads the A and B registers from PLIST and PLIST+1 respectively, adds them and stores the result in PLIST+2. Return is then made to the instruction at RETL.

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 RET2.

Second call to ADSUB:

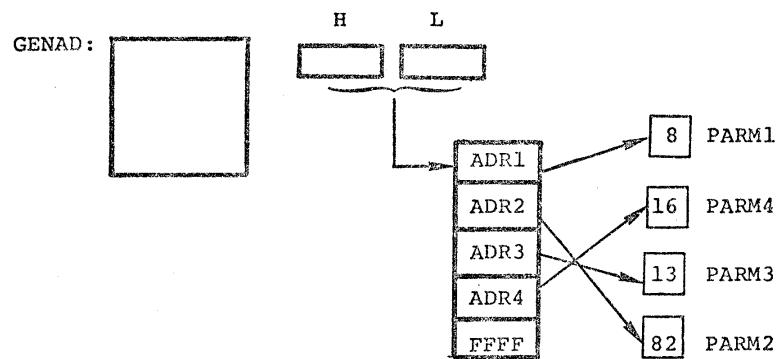


Note that the parameter lists PLIST and LIST2 could appear anywhere in memory without altering the results produced by ADSUB.

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 number whose first byte is FFH (assuming that no parameters will be stored above address FF00H).

Call to GENAD:



As implemented below, GENAD saves the current sum (beginning with zero) in the C register. It then loads the address of the first parameter into the D and E registers. If this address is greater than or equal to FF00H, it reloads the accumulator with the sum held in the C register and returns to the calling routine. Otherwise, it loads the parameter into the accumulator and adds the sum in the C register to the accumulator. The routine then loops back to pick up the remaining parameters.

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
	LXI CALL	H, PLIST GENAD	; Calling program

PLIST:	DW	PARM1	; List of parameter
	DW	PARM2	; addresses
	DW	PARM3	
	DW	PARM4	
	DW	0FFFFH	; Terminator

PARM1:	DB	6	
PARM4:	DB	16	

PARM3:	DB	13	

PARM2:	DB	82	

GENAD:	XRA	A	; Clear accumulator
LOOP:	MOV	C, A	; Save current total in C
	MOV	E, M	; Get low order address byte ; of first parameter
	INX	H	
	MOV	A, M	; Get high order address byte ; of first parameter
	CPI	0FFH	; Compare to FFH
	JZ	BACK	; If equal, routine is complete
	MOV	D, A	; D and E now address parameter
	LDAX	D	; Load accumulator with parameter
	ADD	C	; Add previous total
	INX	H	; Increment H and L to point ; to next parameter address
	JMP	LOOP	; Get next parameter
BACK:	MOV	A, C	; Routine done -- restore total
	RET		; Return to calling routine

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

	LXI	H, PLIST
	CALL	GENAD
	<hr/>	
	<hr/>	
PLIST:	DW	PARM4
	DW	PARM1
	DW	0FFFFH

would cause PARM1 and PARM4 to be added, no matter where in memory they might be located (excluding addresses above FF00H).

Many variations of parameter passing are possible. For example, if it was necessary to allow parameters to be stored at any address, a calling program could pass the total number of parameters as the first parameter; the subroutine would load this first parameter into a register and use it as a counter to determine when all parameters had been accepted.

5.3 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 most 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	LOW-ORDER BYTE
			OF RESULT	OF RESULT
Start	00111100	00101010	00000000	00000000
Step 1	a----- b		00000000	00000000
Step 2	a----- b		00000000	00000000
Step 3	a----- b		00101010	00000000
Step 4	a----- b		00010101	00000000
Step 5	a----- b		00111111	00000000
Step 6	a----- b		00011111	10000000
Step 7	a----- b		01001001	01000000
Step 8	a----- b		00100100	11000000
			01001110	11000000
			00100111	01100000
			00010011	10110000
			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:

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

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:

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

Step two produces:

$$\begin{aligned} & ((\text{BIT0} * \text{MCND} * 2^8) * 2^{-1} + (\text{BIT1} * \text{MCND} * 2^8)) * 2^{-1} \\ & = \text{BIT0} * \text{MCND} * 2^6 + \text{BIT1} * \text{MCND} * 2^7 \end{aligned}$$

And so on, until step eight produces:

$$\text{BIT0} * \text{MCND} * 2^0 + \text{BIT1} * \text{MCND} * 2^1 + \dots + \text{BIT7} * \text{MCND} * 2^7$$

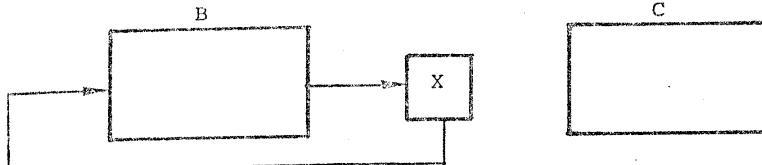
which is equivalent to Equation 1 above, and therefore is the correct result.

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

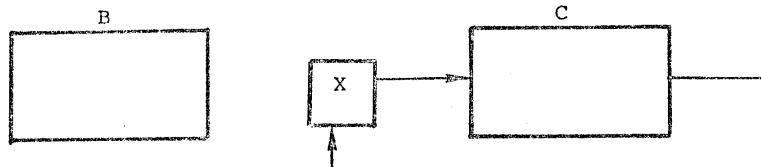
The program uses the B register to hold the most significant byte of the result, and the C register to hold the least significant byte of the result.

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

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

An analogous 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
        MOV      B, A
        MOV      A, C      ; Rotate carry into C register; rot
        RAL      C, A      ; next most significant bit to carry
        MOV      E
        DCR
        JZ       DIV1
        MOV      A, B      ; Rotate most significant bit to
        RAL      ; high-order quotient
        SUB      D          ; Subtract divisor. If less than
        JNC      DIV0      ; high-order quotient, go to DIV0
        ADD      D          ; Otherwise add it back
        JMP      DIV0
        RAL
        MOV      E, A
        MVI      A, OFFH    ; Complement the quotient
        XRA      C
        MOV      C, A
        MOV      A, E
        RAR
DONE:

```

5.4 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:

$$\begin{array}{r}
 32AF8A \\
 + 84BA90 \\
 \hline
 B76A1A
 \end{array}$$

This addition may be performed on the 8080 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:



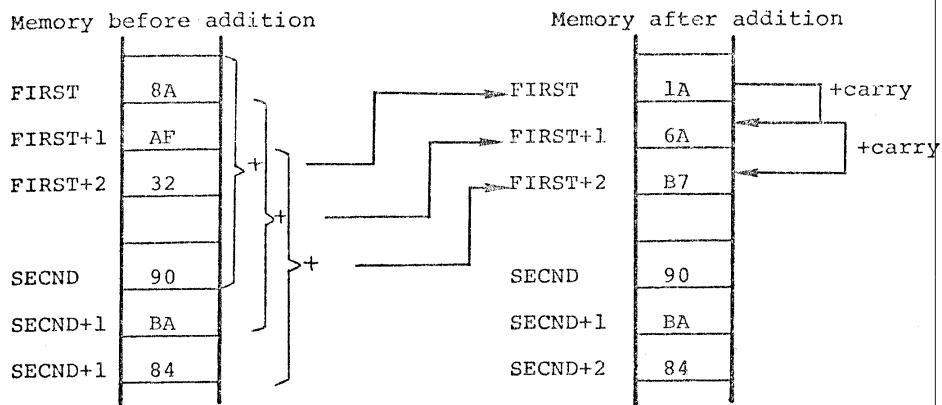
carry=1 carry=1

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).

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 these locations.



<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
MADD:	LXI	B, FIRST	; B and C address FIRST
	LXI	H, SECND	; H and L address SECND
	XRA	A	; Clear carry bit
LOOP:	LDAX	B	; Load byte of FIRST
	ADC	M	; Add byte of SECND with carry
	STAX	B	; Store result at FIRST
	DCR	C	; Done if C = 0
	JZ	DONE	
	INX	B	; Point to next byte of FIRST
	INX	H	; Point to next byte of SECND
	JMP	LOOP	; Add next two bytes
DONE:	---		
FIRST:	DB	90H	
	DB	0BAH	
	DB	84H	
SECND:	DB	8AH	
	DB	0AFH	
	DB	32H	

Since none of the instructions in the program loop affect the carry bit except ADC, the addition with carry will proceed correctly.

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 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:

$$\begin{array}{r} 1301 \\ - 0503 \\ \hline 0DFE \end{array}$$

This subtraction may be performed on the 8080 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):

$$\begin{array}{r} 00000001 = 01H \\ 11111101 = -(03H+carry) \\ \hline 11111110 = 0FEH, \text{ the low-order result} \\ \text{overflow} = 0, \text{ setting carry} = 1 \text{ indicating a borrow} \end{array}$$

High-order subtraction:

$$\begin{array}{r} 00010011 = 13H \\ 11111010 = -(05H+carry) \\ \hline 00001101 \\ \text{overflow} = 1, \text{ resetting the carry bit indicating no borrow} \end{array}$$

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, placing the result at FIRST.

5.5 DECIMAL ADDITION

Any 4-bit data quantity may be treated as a decimal number as long as it represents one of the decimal digits from 0 through 9, and does not contain any of the bit patterns representing the hexadecimal digits A through F. In order to preserve this decimal interpretation when performing addition, the value 6 must be added to the 4-bit quantity whenever the addition produces a result between 10 and 15. This is because each 4-bit data quantity can hold 6 more combinations of bits than there are decimal digits.

Decimal addition is performed on the 8080 by letting each 8-bit byte represent two 4-bit decimal digits. The bytes are summed in the accumulator in standard fashion, and the DAA (decimal adjust accumulator) instruction is then used as in Section 3. to convert the 8-bit binary result to the correct representation of 2 decimal digits. The settings of the carry and auxiliary carry bits also affect the operation of the DAA, permitting the addition of decimal numbers longer than two digits.

To perform the decimal addition:

$$\begin{array}{r} 2985 \\ + 4936 \\ \hline 7921 \end{array}$$

the process works as follows:

- (1) Clear the carry and add the two lowest-order digits of each number (remember that each 2 decimal digits are represented by one byte).

$$\begin{array}{r} 85 = 10000101B \\ 36 = 00110110B \\ \text{carry} = 0 \\ \hline 0] 10111011B \\ \text{carry} = 0 \quad \text{auxiliary carry} = 0 \end{array}$$

The accumulator now contains BBH.

- (2) Perform a DAA operation. Since the rightmost four bits are 10D, 6 will be added to the accumulator.

$$\begin{array}{r} \text{Accumulator} = 10111011B \\ 6 = 0110B \\ \hline 11000001B \end{array}$$

Since the leftmost 4 bits are now ≥ 10 , 6 will be added to these bits, setting the carry bit.

$$\begin{array}{r} \text{Accumulator} = 11000001B \\ 6 = 0110 \quad B \\ \hline 1] 00100001B \\ \text{carry bit} = 1 \end{array}$$

The accumulator now contains 21H. Store these two digits.

- (3) Add the next group of two digits:

$$\begin{array}{r} 29 = 00101001B \\ 49 = 01001001B \\ \text{carry} = 1 \\ \hline 0] 01110011B \\ \text{carry} = 0 \quad \text{auxiliary carry} = 1 \end{array}$$

The accumulator now contains 72H.

- (4) Perform a DAA operation. Since the auxiliary carry bit is set, 6 will be added to the accumulator.

$$\begin{array}{r} \text{Accumulator} = 01110011B \\ 6 = 0110B \\ \hline 0] 01111001B \\ \text{carry bit} = 0 \end{array}$$

Since the leftmost 4 bits are < 10 and the carry bit is reset, no further action occurs.

Thus, the correct decimal result 7921 is generated in two bytes.

A routine which adds decimal numbers, then, is exactly analogous to the multibyte addition routine MADD of Section 5.4, and may be produced by inserting the instruction DAA after the ADC M instruction of that example. Each iteration of the program loop will add two decimal digits (one byte) of the numbers.

5.6 DECIMAL SUBTRACTION

Each 4-bit data quantity may be treated as a decimal number as long as it represents one of the decimal digits 0 through 9. The DAA (decimal adjust accumulator) instruction may be used to permit subtraction of one byte (representing a 2-digit decimal number) from another, generating a 2-digit decimal result. In fact, the DAA permits subtraction of multidigit decimal numbers.

The process consists of generating the hundred's complement of the subtrahend digit (the difference between the subtrahend digit and 100 decimal), and adding the result to the minuend digit. For instance, to subtract 34D from 56D, the hundred's complement of 34D ($100D - 34D = 66D$) is added to 56D, producing 122D which, when truncated to 8 bits gives 22D, the correct result. If a borrow was generated by the previous subtraction, the 99's complement of the subtrahend digit is produced to compensate for the borrow.

In detail, the procedure for subtracting one multi-digit decimal from another is as follows:

- (1) Set the carry bit = 1 indicating no borrow.
- (2) Load the accumulator with 99H, representing the number 99 decimal.
- (3) Add zero to the accumulator with carry, producing either 99H or 9AH, and resetting the carry bit.
- (4) Subtract the subtrahend digits from the accumulator, producing either the 99's or 100's complement.
- (5) Add the minuend digits to the accumulator.
- (6) Use the DAA instruction to make sure the result in the accumulator is in decimal format, and to indicate a borrow in the carry bit if one occurred.

Save this result.

- (7) If there are more digits to subtract, go to step 2.

Otherwise, stop.

Example:

Perform the decimal subtraction:

$$\begin{array}{r} 4358D \\ - 1362D \\ \hline 2996D \end{array}$$

- (1) Set carry = 1
- (2) Load accumulator with 99H.
- (3) Add zero with carry to the accumulator, producing 9AH.

$$\begin{array}{r} \text{Accumulator} = 10011001B \\ 0 = 00000000B \\ \text{Carry} = \underline{1} \\ \hline 10011010B = 9AH \end{array}$$

- (4) Subtract the subtrahend digits 62H from the accumulator.

$$\begin{array}{r} \text{Accumulator} = 10011010B \\ 62H = \underline{10011110B} \\ \hline 1] 00111000B \end{array}$$

- (5) Add the minuend digits 58H to the accumulator.

$$\begin{array}{r} \text{Accumulator} = 00111000B \\ 58H = \underline{01011000B} \\ \hline 0] 10010000B = 90H \end{array}$$

carry = 0 auxiliary carry = 1

- (6) DAA converts accumulator to 96H (since auxiliary carry = 1) and leaves carry bit = 0 indicating that a borrow occurred.

- (7) Load accumulator with 99H.

- (8) Add zero with carry to accumulator, leaving accumulator = 99H.

- (9) Subtract the subtrahend digits 13H from the accumulator.

$$\begin{array}{r} \text{Accumulator} = 10011001B \\ 13H = \underline{11101101B} \\ \hline 1] 10000110B \end{array}$$

- (10) Add the minuend digits 43H to the accumulator.

$$\begin{array}{r} \text{Accumulator} = 10000110B \\ 43H = \underline{01000011B} \\ \hline 0] 11001001B = C9H \end{array}$$

carry = 0 auxiliary carry = 0

- (11) DAA converts accumulator to 29H and sets the carry bit = 1, indicating no borrow occurred.

Therefore, the result of subtracting 1362D from 4358D is 2996D

The following subroutine will subtract one 16-digit decimal number from another using the following assumptions:

The minuend is stored least significant (2) digits first beginning at location MINU.

The subtrahend is stored least significant (2) digits first beginning at location SBTRA.

The result will be stored least significant (2) digits first, replacing the minuend.

<u>Label</u>	<u>Code</u>	<u>Operand</u>	<u>Comment</u>
DSUB:	LXI	D, MINU	; D and E address minuend
	LXI	H, SBTRA	; H and L address subtrahend
	MVI	C, 8	; Each loop subtracts 2 digits ; (one byte), therefore program ; will subtract 16 digits.
LOOP:	STC		; Set carry indicating no borrow
	MVI	A, 99H	; Load accumulator with 99H.
	ACI	0	; Add zero with carry
	SUB	M	; Produce complement of subtrahend
	XCHG		; Switch D and E with H and L
	ADD	M	; Add minuend
	DAA		; Decimal adjust accumulator
	MOV	M, A	; Store result
	XCHG		; Reswitch D and E with H and L
	DCR	C	; Done if C = 0
	JZ	DONE	
DONE:	INX	D	; Address next byte of minuend
	INX	H	; Address next byte of subtrahend
	JMP	LOOP	; Get next 2 decimal digits

5.7 ALTERING MACRO EXPANSIONS

This section describes how a macro may be written such that identical references to the macro produce different expansions. As a useful example of this, consider a macro SBMAC which needs to call a subroutine SUBR to perform its function. One way to provide the macro with the necessary subroutine would be to include a separate copy of the subroutine in any program which contains the macro. A better method is to let the macro itself generate the subroutine during the first macro expansion, but skip the generation of the subroutine on any subsequent expansion. This may be accomplished as follows:

Consider the following program section which consists of one global set statement and the definition of SBMAC (dashes indicate those assembly language statements necessary to the program, but irrelevant to this discussion):

<u>Label</u>	<u>Code</u>	<u>Operand</u>
FIRST	SET	0FFH
;		
SBMAC	MACRO	
	--	
	--	
	CALL	SUBR
	--	
	--	
	IF	FIRST
FIRST	SET	0
	JMP	OUT
SUBR::	--	
	--	
	RET	
OUT:	NOP	
	ENDIF	
	ENDM	

The symbol FIRST is set to FFH, then the macro SBMAC is defined.

The first time SBMAC is referenced, the expansion produced will be the following:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	SBMAC	
	--	
	--	
	CALL	SUBR
	--	
	--	
	IF	FIRST
FIRST	SET	0
	JMP	OUT
SUBR:	--	
	--	
	RET	
OUT:	NOP	

Since FIRST is non-zero when encountered during this expansion, the statements between the IF and ENDIF are assembled into the program. The first statement thus assembled sets the value of FIRST to 0, while the remaining statements are the necessary subroutine SUBR and a jump around the subroutine. When this portion of the program is executed, the subroutine SUBR will be called, but program execution will not flow into the subroutine's definition.

On any subsequent reference to SBMAC in the program, however, the following expansion will be produced:

<u>Label</u>	<u>Code</u>	<u>Operand</u>
	SBMAC	
	--	
	--	
	CALL	SUBR
	--	
	--	
	IF	FIRST

Since FIRST is now equal to zero, the IF statement ends the macro expansion and does not cause the subroutine to be generated again. The label SUBR is known during this expansion because it was defined globally (followed by two colons in the definition).

6.0 INTERRUPTS

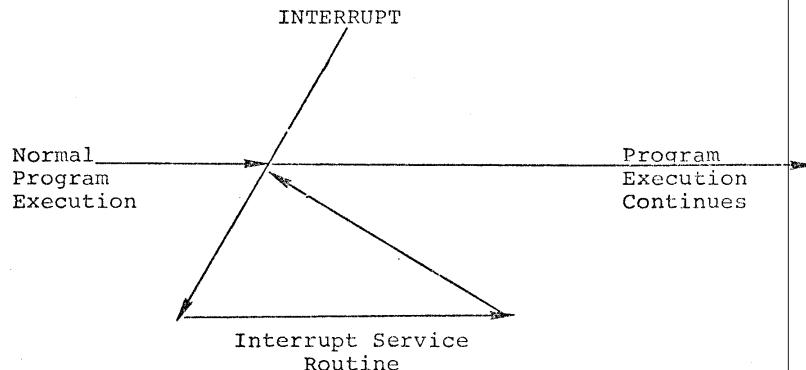
Often, events occur external to the central processing unit which require immediate action by the CPU. For example, suppose a device is receiving a string of 80 characters from the CPU, one at a time, at fixed intervals. There are two ways to handle such a situation:

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

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, forcing a branch from the executing program to a special interrupt service routine.

The interrupt sequence may be illustrated as follows:



The 8080 contains a bit named INTE which may be set or reset by the instructions EI and DI described in Section 3.16. Whenever INTE is equal to 0, the entire interrupt handling system is disabled, and no interrupts will be accepted. When INTE is equal to 1, the interrupt handling system is enabled, and any interrupt will be accepted.

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 interrupt enable bit, INTE, is reset = 0.
- 3) 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 3.15), 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:

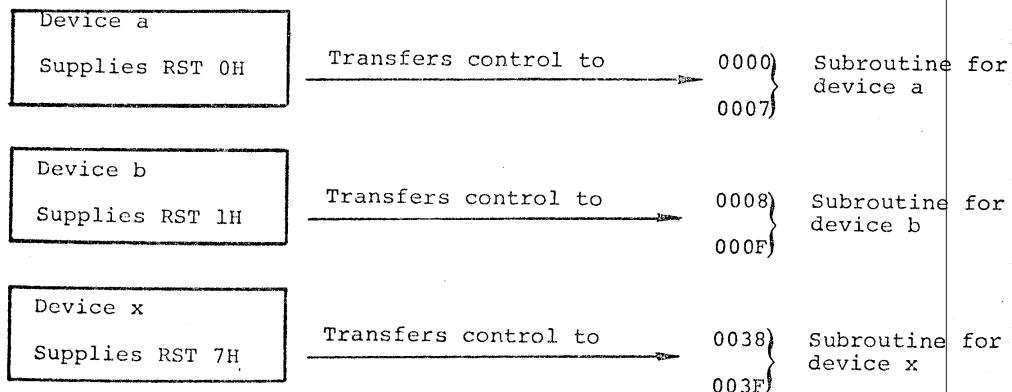
RST 0H

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.



Note that any of these 8-byte subroutines may in turn call longer subroutines to process the interrupt, if necessary.

Any device may supply an RST instruction (and indeed may supply any 8080 instruction).

The following is an example of an Interrupt sequence:

ARBITRARY
MEMORY ADDRESS

3C0B
→ 3C0C

INSTRUCTION

MOV C,B
MOV E,A

Device 1 supplies
RST 0H

Program Counter =
3C0C pushed onto
the stack.

Control transferred
to 0000

0000

Instruction 1
Instruction 2

RET

Stack popped into
program counter

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 pushed onto the stack and program control is transferred to location 0000H. (This subroutine may perform jumps, calls, or any other operation). When the RETURN is executed, address 3C0C is popped off the stack and replaces the contents of the program counter, causing execution to continue at the instruction following the point where the interrupt occurred.

6.1 WRITING INTERRUPT SUBROUTINES

In general, any registers or condition bits changed by an interrupt subroutine must be restored before returning to the interrupted program, or errors will occur.

For example, suppose a program is interrupted just prior to the instruction:

JC LOC

and the carry bit equals 1. If the interrupt subroutine happens to zero the carry bit just before returning to the interrupted program, the jump to LOC which should have occurred will not, causing the interrupted program to produce erroneous results.

Like any other subroutine then, any interrupt subroutine should save at least the condition bits and restore them before performing a RETURN operation. (The obvious and most convenient way to do this is to save the data in the stack, using PUSH and POP operations.)

Further, the interrupt enable system is automatically disabled whenever an interrupt is acknowledged. Except in special cases, therefore, an interrupt subroutine should include an EI instruction somewhere to permit detection and handling of future interrupts. Any time after an EI is executed, the interrupt subroutine may itself be interrupted. This process may continue to any level, but as long as all pertinent data are saved and restored, correct program execution will continue automatically.

A typical interrupt subroutine, then, could appear as follows:

<u>Code</u>	<u>Operand</u>	<u>Comment</u>
PUSH		; Save condition bits and accumulator
EI	PSW	; Re-enable interrupts
.		;
.		; Perform necessary actions to service
.		the interrupt
.		
POP	PSW	; Restore machine status
RET		; Return to interrupted program

APPENDIX A

-- INSTRUCTION SUMMARY --

This appendix provides a summary of 8080 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 and 0 is the least significant (rightmost) bit.
ADDR	Any memory address
Aux. carry	The auxiliary carry bit
Carry	The carry bit
CODE	An operation code
DATA	8 bits (one byte) of data
DATA16	16 bits (2 bytes) of data
DST	Destination register or memory byte
EXP	A constant or mathematical expression
INTE	The 8080 interrupt enable flip-flop
LABEL:	Any instruction label
M	A memory byte
Parity	The parity bit
PC	Program Counter
PCH	The most significant 8 bits of the program counter
PCL	The least significant 8 bits of the program counter
REGM	Any register or memory byte

RP	A register pair. Legal register pair symbols are:
	B for registers B and C
	D for registers D and E
	H for registers H and L
	SP for the 16 bit stack pointer
	PSW for condition bits and register A
RP1	The first register of register pair RP
RP2	The second register of register pair RP.
sign	The sign bit
SP	The 16-bit stack pointer register
SRC	Source register or memory byte
zero	The zero bit
XY	The value obtained by concatenating the values X and Y
[]	An optional field enclosed by brackets
()	Contents of register or memory byte enclosed by parentheses
→	Replace value on lefthand side of arrow with value on right-hand side of arrow

CARRY BIT INSTRUCTIONS

Format:

[LABEL:] CODE

CODE	DESCRIPTION
STC	(carry) ← 1 Set carry
CMC	(carry) ← (carry) Complement carry

Condition bits affected: Carry

SINGLE REGISTER INSTRUCTIONS

Format:

[LABEL:]	INR	REGM
-or-		
[LABEL:]	DCR	REGM
-or-		
[LABEL:]	CMA	
-or-		
[LABEL:]	DAA	

Code	Description	
INR	(REGM) \leftarrow	(REGM)+1 Increment register REGM
DCR	(REGM) \leftarrow	(REGM)-1 Decrement register REGM
CMA	(A) \leftarrow	Complement accumulator
DAA	If $(A_0 - A_3) > 9$ or (aux. carry)=1, $(A) \xrightarrow{(A_0 - A_3)} (A) + 6$ Then if $(A_4 - A_7) > 9$ or (carry)=1 $(A) = (A) + 6 * 2^4$	Convert accumulator contents to form two decimal digits

Condition bits affected: INR, DCR : Zero, sign, parity
CMA : None
DAA : Zero, sign, parity, carry, aux. carry

NOP INSTRUCTION

Format:

[LABEL:] NOP

Code	Description	
NOP	----- No operation	

Condition bits affected: None

Format:

[LABEL:]	MOV	DST, SRC
	-or-	
[LABEL:]	CODE	RP

NOTE: SRC and DST not both = M

NOTE: RP = B or D

Code	Description	
MOV	(DST) ——— (SRC)	Load register DST from register SRC
STAX	((RP)) ——— (A)	Store accumulator at memory location referenced by the specified register pair
LDAX	(A) ——— ((RP))	Load accumulator from memory location referenced by the specified register pair

Condition bits affected: None

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 to accumulator with carry
SUB	(A) ——— (A)-(REGM)	Subtract REGM from accumulator
SBB	(A) ——— (A)-(REGM)-(carry)	Subtract REGM from accumulator with borrow
ANA	(A) ——— (A) AND (REGM)	AND accumulator with REGM
XRA	(A) ——— (A) XOR (REGM)	EXCLUSIVE-OR accumulator with REGM

Code	Description	
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, aux. carry
 AND, XRA, ORA: Sign, zero, parity. Carry is zeroed.

CMP: Carry, sign, zero, parity, aux. carry. Zero set if (A)=(REGM)
 Carry reset if (A) < (REGM)
 Carry set if (A) ≥ (REGM)

ROTATE ACCUMULATOR INSTRUCTIONS

Format:

[LABEL:] CODE

Code	Description
RLC	(carry) —— A ₇ , A _{n+1} , —— A _n , A ₀ —— A ₇
RRC	(carry) —— A ₀ , A _n —— A _{n+1} , A ₇ —— A ₀
RAL	A _{n+1} —— A _n , (carry) —— A ₇ , A ₀ —— (carry)
RAR	A _n —— A _{n+1} , (carry) —— A ₀ , A ₇ —— (carry)

Condition bits affected: Carry

REGISTER PAIR INSTRUCTIONS

Format:

[LABEL:] CODE1 RP
 -or-
 [LABEL:] CODE2

Note: For PUSH and POP, RP=B,D,H, or PSW
 For DAD, INX, and DCX, RP=B,D,H, or SP

Code1	Description	
PUSH	$((SP)-1) \leftarrow (RP1)$, $((SP)-2) \leftarrow (RP2)$, $(SP) \leftarrow (SP)-2$	Save RP on the stack RP=A saves accumulator and condition bits.
POP	$(RP1) \leftarrow ((SP)+1)$, $(RP2) \leftarrow ((SP))$, $(SP) \leftarrow (SP)+2$	Restore RP from the stack RP=A restores accumulator and condition bits.
DAD	$(HL) \leftarrow (HL) + (RP)$	Add RP to the 16-bit number in H and L.
INX DCX	$(RP) \leftarrow (RP)+1$ $(RP) \leftarrow (RP)-1$	Increment RP by 1 Decrement RP by 1
Code2	Description	
XCHG	$(H) \leftarrow (D)$, $(L) \leftarrow (E)$	Exchange the 16 bit number in H and L with that in D and E.
XTHL	$(L) \leftarrow ((SP))$, $(H) \leftarrow ((SP)+1)$	Exchange the last values saved in the stack with H and L.
SPHL	$(SP) \leftarrow (H):(L)$	Load stack pointer from H and L.

Condition bits affected:

PUSH, INX, DCX, XCHG, XTHL, SPHL: None

POP : If RP=PSW, all condition bits are restored from the stack, otherwise none are affected.

DAD : Carry

IMMEDIATE INSTRUCTIONS

Format:

[LABEL:]	LXI	RP, DATA16
	-or-	
[LABEL:]	MVI	REGM, DATA
	-or-	
[LABEL:]	CODE	REGM

Note: RP=B, D, H, or SP

LXI	(RP) —— DATA 16	Move 16 bit immediate Data into RP
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 to accumulator with carry
SUI	(A) —— (A) - DATA	Subtract immediate data from accumulator
SBI	(A) —— (A) - DATA - (carry)	Subtract immediate data from accumulator with borrow
ANI	(A) —— (A) AND DATA	AND accumulator with immediate data
XRI	(A) —— (A) XOR DATA	EXCLUSIVE-OR ccumulator 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:

LXI, MVI: None

ADI, ACI, SUI, SBI: Carry, sign, zero, parity, aux. carry

ANI, XRI, ORI: Zero, sign, parity. Carry is zeroed.

CPI: Carry, sign, zero, parity, aux. carry. Zero set if $(A) = DATA$
 Carry reset if $(A) < DATA$
 Carry set if $(A) \geq DATA$

DIRECT ADDRESSING INSTRUCTIONS

Format:

[LABEL:]	CODE	ADDR
CODE	DESCRIPTION	
STA	(ADDR) —— (A)	Store accumulator at location ADDR
LDA	(A) —— (ADDR)	Load accumulator from location ADDR
SHLD	(ADDR) — (L), (ADDR+1) — (H)	Store L and H at ADDR and ADDR+1
LHLD	(L) — (ADDR), (H) — (ADDR+1)	Load L and H from ADDR and ADDR+1

Condition bits affected: None

JUMP INSTRUCTIONS

Format:

[LABEL:] PCHL

-or-

[LABEL:] CODE ADDR

CODE	DESCRIPTION
PCHL	(PC)---(HL)
JMP	(PC)---ADDR
JC	If (carry) = 1, (PC)---ADDR If (carry) = 0, (PC)---(PC)+3
JNC	If (carry) = 0, (PC)---ADDR If (carry) = 1, (PC)---(PC)+3
JZ	If (zero) = 1, (PC)---ADDR If (zero) = 0, (PC)---(PC)+3
JNZ	If (zero) = 0, (PC)---ADDR If (zero) = 1, (PC)---(PC)+3
JP	If (sign) = 0, (PC)---ADDR If (sign) = 1, (PC)---(PC)+3
JM	If (sign) = 1, (PC)---ADDR If (sign) = 0, (PC)---(PC)+3
JPE	If (parity)= 1, (PC)---ADDR If (parity)= 0, (PC)---(PC)+3
JPO	If (parity)= 0, (PC)---ADDR If (parity)= 1, (PC)---(PC)+3

Condition bits affected: None

CALL INSTRUCTIONS

Format:

[LABEL:]	CODE	ADDR
----------	------	------

CODE	DESCRIPTION	
CALL	((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR	Call subroutine and push return address onto stack
CC	If (carry) = 1, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (carry) = 0, (PC) ← (PC)+3	Call subroutine if carry set
CNC	If (carry) = 0, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (carry) = 1, (PC) ← (PC)+3	Call subroutine if carry reset
CZ	If (zero) = 1, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (zero) = 0, (PC) ← (PC)+3	Call subroutine if zero set
CNZ	If (zero) = 0, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (zero) = 1, (PC) ← (PC)+3	Call subroutine if zero reset
CP	If (sign) = 0, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (sign) = 1, (PC) ← (PC)+3	Call subroutine if sign plus
CM	If (sign) = 1, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (sign) = 0, (PC) ← (PC)+3	Call subroutine if sign minus
CPE	If (parity)= 1, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (parity)= 0, (PC) ← (PC)+3	Call subroutine if parity even
CPO	If (parity)= 0, ((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2, (PC) ← ADDR If (parity)= 1, (PC) ← (PC)+3	Call subroutine if parity odd

Condition bits affected: None

RETURN INSTRUCTIONS

Format:

[LABEL:] CODE

CODE	DESCRIPTION
RET	(PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 Return from subroutine
RC	If (carry) = 1, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (carry) = 0, (PC)---(PC)+3 Return if carry set
RNC	If (carry) = 0, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (carry) = 1, (PC)---(PC)+3 Return if carry reset
RZ	If (zero) = 1, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (zero) = 0, (PC)---(PC)+3 Return if zero set
RNZ	If (zero) = 0, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (zero) = 1, (PC)---(PC)+3 Return if zero reset
RM	If (sign) = 1, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (sign) = 0, (PC)---(PC)+3 Return if minus
RP	If (sign) = 0, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (sign) = 1, (PC)---(PC)+3 Return if plus
RPE	If (parity)= 1, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (parity)= 0, (PC)---(PC)+3 Return if parity even
RPO	If (parity)= 0, (PCL)---((SP)), (PCH)---((SP)+1), (SP)---(SP)+2 If (parity)= 1, (PC)---(PC)+3 Return if parity odd

Condition bits affected: None

RST INSTRUCTION

Format:

[LABEL:] RST EXP

Note: 0 EXP 7

CODE	DESCRIPTION
RST	((SP)-1)---(PCH), ((SP)-2)---(PCL), (SP)---(SP)+2 (PC)---000000000EXP000B Call subroutine at address specified by EXP

Condition bits affected: None

INTERRUPT FLIP FLOP INSTRUCTIONS

Format:

[LABEL:] CODE

CODE	DESCRIPTION	
EI	(INTE) —— 1	Enable the interrupt system
DI	(INTE) —— 0	Disable the interrupt system

Condition bits affected: None

INPUT/OUTPUT INSTRUCTIONS

Format:

[LABEL:] CODE EXP

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

HLT INSTRUCTION

Format:

[LABEL:] HLT

CODE	DESCRIPTION	
HLT	-----	Instruction execution halts until an interrupt occurs.

Condition bits affected: None

PSEUDO - INSTRUCTIONS

ORG PSEUDO - INSTRUCTION

Format:

ORG EXP

Code	Description	
ORG	LOCATION COUNTER ← EXP	Set Assembler location counter to EXP

EQU PSEUDO- INSTRUCTION

Format:

NAME EQU EXP

Code	Description	
EQU	NAME ← EXP	Assign the value EXP to the symbol NAME

SET PSEUDO - INSTRUCTION

Format:

NAME SET EXP

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

END PSEUDO - INSTRUCTION

Format:

END

Code	Description
END	End the assembly.

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.

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 EXECUTION TIMES AND BIT PATTERNS--

This appendix summarizes the bit patterns and number of time states associated with every 8080 CPU instruction.

When using this summary, note the following symbology:

- 1) DDD represents a destination register. SSS represents a source register. Both DDD and SSS are interpreted as follows:

DDD or SSS	Interpretation
000	Register B
001	Register C
010	Register D
011	Register E
100	Register H
101	Register L
110	A memory register
111	The accumulator

- 2) Instruction execution time equals number of time periods multiplied by the duration of a time period.

A time period may vary from 480 nanosecs to 2 μ sec.

Where two numbers of time periods are shown (eg. 5/11), it means that the smaller number of time periods will be required if a condition is not met, and the larger number of time periods will be required if the condition is met.

PINEMONIC	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	Number of Time Periods
CALL	1	1	0	0	1	1	0	1	17
CC	1	1	0	1	1	1	0	0	11/17
CNC	1	1	0	1	0	1	0	0	11/17
CZ	1	1	0	0	1	1	0	0	11/17
CNZ	1	1	0	0	0	1	0	0	11/17
CP	1	1	1	1	0	1	0	0	11/17
CM	1	1	1	1	1	1	0	0	11/17
CPE	1	1	1	0	0	1	0	0	11/17
CPO	1	1	1	1	0	1	0	0	11/17
RET	1	1	0	0	1	0	0	1	10
RC	1	1	0	1	1	0	0	0	5/11
RNC	1	1	0	1	0	0	0	0	5/11
RZ	1	1	0	0	1	0	0	0	5/11
RNZ	1	1	0	0	0	0	0	0	5/11
RP	1	1	1	1	1	0	0	0	5/11
RM	1	1	1	1	1	0	0	0	5/11
RPE	1	1	1	0	1	1	0	0	5/11
RPO	1	1	1	1	0	0	0	0	5/11
RST	1	1	A	A	A	1	1	1	11
IN	1	1	0	1	1	0	1	1	10
OUT	1	1	0	1	0	0	1	1	10
LXI B	1	0	0	0	0	0	0	0	10
LXI D	0	0	0	0	1	0	0	1	10
LXI H	0	0	0	1	0	0	0	1	10
LXI SP	0	0	0	1	1	0	0	1	10
PUSH B	1	1	0	0	0	1	0	0	11
PUSH D	1	1	0	1	0	1	0	1	11
PUSH H	1	1	1	0	0	1	0	1	11
PUSH A	1	1	1	1	0	1	0	1	11
POP B	1	1	1	0	0	1	0	0	10
POP D	1	1	0	1	0	0	0	1	10
POP H	1	1	1	0	0	0	0	1	10
POP A	1	1	1	1	0	0	0	1	10
STA	0	0	1	1	1	0	0	1	13
LDA	0	0	1	1	1	1	0	0	13
XCHG	1	1	1	0	0	0	1	1	4
XTHL	1	1	1	1	0	0	0	0	13
SPHL	1	1	1	1	1	1	0	0	5
PCHL	1	1	1	0	1	1	0	0	5
DAD B	0	0	0	0	1	0	0	1	10
DAD D	0	0	0	0	1	1	0	0	10
DAD H	0	0	0	1	0	1	0	0	10
DAD SP	0	0	0	1	1	0	0	0	10
STAX B	0	0	0	1	0	0	0	1	7
STAX D	0	0	0	0	1	0	0	1	7
LDAx B	0	0	0	0	0	1	0	1	7
LDAx D	0	0	0	0	0	1	1	1	5
INX B	0	0	0	0	1	0	0	0	5
INX D	0	0	0	0	1	0	0	1	5
INX H	0	0	0	0	1	1	0	1	5
INX SP	0	0	0	1	1	0	0	1	5

MINIMONIC	R7	R6	R5	R4	R3	R2	R1	R0		
MOV r1,r2	0	1	D	D	D	S	S	S		5
MOV M,r	0	1	1	1	0	S	S	S		7
MOV r,M	0	1	D	D	0	1	1	0		7
HLT	0	1	1	1	0	1	1	0		7
MVI r	0	0	D	D	D	1	1	0		7
MVI M	0	0	1	1	0	1	1	0		7
INR	0	0	D	D	D	1	1	0		5
DCR	0	0	D	D	D	1	1	0		5
INR A	0	0	1	1	1	1	1	0		5
DCR A	0	0	1	1	1	1	1	0		5
INR M	0	0	1	1	1	0	1	0		10
DCR M	0	0	1	1	1	0	1	0		10
ADD r	1	0	0	0	0	1	S	S		4
ADC r	1	0	0	0	1	0	S	S		4
SUB r	1	0	0	0	1	1	S	S		4
SBB r	1	0	0	0	1	1	S	S		4
AND r	1	0	1	0	0	1	S	S		4
XRA r	1	0	1	1	0	1	S	S		4
ORA r	1	0	1	1	1	0	S	S		4
CMP r	1	0	1	1	1	1	S	S		4
ADD M	1	0	0	0	0	1	1	1		7
ADC M	1	0	0	0	0	1	1	1		7
SUB M	1	0	0	0	1	0	1	1		7
SBB M	1	0	0	0	1	1	1	1		7
AND M	1	0	1	0	0	1	1	1		7
XRA M	1	0	1	1	0	1	1	1		7
ORA M	1	0	1	1	1	0	1	1		7
CMP M	1	0	1	1	1	1	1	0		7
ADI	1	1	0	0	0	1	1	1		7
ACI	1	1	0	0	1	0	1	1		7
SUI	1	1	0	0	1	1	1	0		7
SBI	1	1	0	0	1	0	1	1		7
ANI	1	1	1	1	0	0	1	1		7
XRI	1	1	1	1	0	1	1	1		7
ORI	1	1	1	1	1	1	1	1		7
CPI	1	1	1	1	1	1	1	1		4
RLC	0	0	0	0	0	0	1	1		4
RRD	0	0	0	0	0	0	1	1		4
RAL	0	0	0	0	0	1	0	1		4
RAR	0	0	0	0	0	1	1	1		4
JMP	1	1	0	0	0	0	0	0		10
JC	1	1	0	0	0	1	1	0		10
JNC	1	1	0	0	0	1	0	1		10
JZ	1	1	0	0	0	0	0	0		10
JP	1	1	0	1	1	1	0	1		10
JM	1	1	0	1	1	1	1	0		10
JPE	1	1	0	1	1	0	1	1		10
JPO	1	1	0	1	1	0	0	0		10

MNEMONIC	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	Number of Time Periods
DCX B	0	0	0	0	1	0	1	1	5
DXC D	0	0	0	1	1	0	1	1	5
DCX H	0	0	1	0	1	0	1	1	5
DCX SP	0	0	1	1	1	0	1	1	5
CMA	0	0	1	0	1	1	1	1	4
STC	0	0	1	1	0	1	1	1	4
CMC	0	0	1	1	1	1	1	1	4
DAA	0	0	1	0	0	1	1	1	4
SHLD	0	0	1	0	0	0	1	0	17
LHLD	0	0	1	0	1	0	1	0	17
EI	1	1	1	1	1	0	0	1	4
DI	1	1	1	1	0	0	1	1	4
NOP	0	0	0	0	0	0	0	0	4

APPENDIX "C"

--ASCII TABLE--

The 8080 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

C-1

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

