COMPUCOLOR II

ASSEMBLER OPERATING

MANUAL

999214 Rev. 1

COMPUCOLOR ASSEMBLER OPERATING MANUAL

Purpose: The purpose of the assembler is to convert source assembly language text into 8080 object code. It is also used to find syntactical errors.It is implemented from FCS:

```
FCS>RUN ASM
```

FCS will load the assembler and then will respond with a >, to this you will type the following:

```
>ASM <file specifier> TO <drive>:
```

If the user wishes an error only listing a /E should follow the file specifier. The file specifier is assumed to be on the default device and an SRC type file however the file specifier may be <drive:name.type;version> The output file will be type .LDA with the same version number as the source. An error will occur if an attempt is made to reassemble the same version. If the user types

```
>ASM <file specifier>
```

no output file will be generated.

Testing: Two sample programs are provided. The first being named SAMPLE. This program is used to show the user how to access a disk file. It copies to contents of a file to another file with the type of .DOC. The second is the source for the program PRINT provided with the EDITOR. It is used to print a .SRC file. The program will not work on a BAS file. Print may be temporarily stopped by pressing the break key and restarted by pressing the return key. It may be halted by pressing the escape key.

Disk access: Other than the two programs mentioned above, there are many other useful file acesss routines which are described on the pages following.

General utility routines:

Several useful ulitity routines are found in ROM and may be of use to the user.

```
BASOUT  0033H   ;Sends 1 byte to the screen
                ;from A register
OSTR    33F4H   ;Sends string ending with 239 (decimal)
                ;to the screen. HL points to string.
                ;OSTR does not do a CRLF
MOVDH   343BH   ;Moves string of length B from location
                ;HL to DE
MOVHD   3444H   ;Same as MOVDH except DE to HL
CMPHD   344DH   ;Compares registers HL with DE
CMPDH   3453H   ;Compares registers DE with HL
SUBHD   3459H   ;Subtracts DE from HL
ADHLA   3518H   ;Adds A to HL
ANHD    351DH   ;Ands A to HL
NEGH    3524H   ;Negates HL
MULHD   3562H   ;Multiplies HL by DE high word out in
                ;DE low in HL
DIVHD   3581H   ;Divides DE by HL quotient in HL remainder
                ;in DE
XORHD   3533H   ;Exclusive ors HL with DE
RXSER   0020H   ;Reads a character from the serial line
                ;READY (81FF)=50H if no character
TXSER   0028H   ;Transmits a character to the serial line
KEYTEST 0024H   ;Returns with the Z bit if a character is
                ;in KBCHA (81FE)
```

```
A000 (368B)            ORG     368BH
368B (0002)   IDEV:    DS      2         ;INITIAL DEFAULT DEVICE
368D (0001)   IUNT:    DS      1         ;INITIAL DEFAULT UNIT
368E (0000)   HDVCT:   DS      0         ;START OF HANDLER VECTORS

              ; OPEN TYPE CODE BIT DEFINITIONS :

     (0001)   FNEW     EQU     01H       ;0: OLD FILE, 1: NEW FILE

              ; DIRECTORY ENTRY TYPE CODE BIT DEFINITIONS :

     (0001)   TFREE    EQU     01H       ;"FREE SPACE" ENTRY - BYTE VALUE

     (0001)   TPROT    EQU     01H       ;PROTECTED FILE
     (0002)   TFILE    EQU     02H       ;PERMANENT FILE ENTRY

              ; RAM ALLOCATION :

     (8042)   STACK    EQU     8042H     ;INITIAL SP VALUE

368E (8082)            ORG     8082H

8082 (0001)   ZFPB:    DS      1
8083 (0001)   ZFATR:   DS      1
8084 (0006)   ZFNAM:   DS      6
808A (0003)   ZFTYP:   DS      3
808D (0001)   ZFVER:   DS      1
808E (0002)   ZFSBK:   DS      2
8090 (0002)   ZFSIZ:   DS      2
8092 (0001)   ZFLBC:   DS      1
8093 (0002)   ZFLAD:   DS      2
8095 (0002)   ZFSAD:   DS      2
8097 (0001)            DS      1
8098 (0001)   ZFDBK:   DS      1
8099 (0001)   ZFDEN:   DS      1
809A (0002)   ZFAUX:   DS      2
809C (0002)   ZFHAN:   DS      2
809E (0001)   ZFFCN:   DS      1
809F (0001)   ZFDRV:   DS      1
80A0 (0002)   ZFBLK:   DS      2
80A2 (0002)   ZFBUF:   DS      2
80A4 (0002)   ZFXBC:   DS      2
80A6 (0002)   ZFPTR:   DS      2
80A8          ZFPBE:   ; END OF AUX. FPB, END OF BASIC INPUT BUFFER

80A8 (80F0)            ORG     80F0H

80F0 (0002)   DFDV:    DS      2         ;DEFAULT DEVICE (ASCII)
80F2 (0001)   DFUN:    DS      1         ;DEFAULT UNIT (ASCII)
```

```
 80F3 (0002)    FPBP:   DS      2       ;FILE PARAMETER BLOCK POINTER
 80F5 (0001)    OCODE:  DS      1       ;OPEN TYPE CODE
 80F6 (0001)    OVERS:  DS      1       ;ORIGINAL VERSION

                ; SYSTEM FILE PARAMETER BLOCK ALLOCATION :

 80F7 (0001)    FPB:    DS      1       ;OPEN TYPE CODE
 80F8 (0001)    FATR:   DS      1       ;ATTRIBUTE BYTE
 80F9 (0006)    FNAM:   DS      6       ;FILE NAME
 80FF (0003)    FTYP:   DS      3       ;FILE TYPE
 8102 (0001)    FVER:   DS      1       ;FILE VERSION NUMBER
 8103 (0002)    FSBK:   DS      2       ;STARTING BLOCK NUMBER
 8105 (0002)    FSIZ:   DS      2       ;NUMBER OF BLOCKS
 8107 (0001)    FLBC:   DS      1       ;BYTE COUNT OF LAST BLOCK
 8108 (0002)    FLAD:   DS      2       ;LOAD ADR. FOR "IMAGE" FILE
 810A (0002)    FSAD:   DS      2       ;START ADR. FOR "IMAGE" FILE
 810C (0001)            DS      1       ;SPARE
 810D (0001)    FDBK:   DS      1       ;DIRECTORY BLOCK NUMBER
 810E (0001)    FDEN:   DS      1       ;DIRECTORY ENTRY NUMBER
 810F (0002)    FAUX:   DS      2       ;NEW FILE CLOSING SIZE, OR ...
                                        ;... AUX. BYTE COUNT FOR SEQUENTIAL ROUT
INES
 8111 (0002)    FHAN:   DS      2       ;HANDLER ADDRESS
 8113 (0001)    FFCN:   DS      1       ;HANDLER FUNCTION CODE
 8114 (0001)    FDRV:   DS      1       ;DRIVE NUMBER
 8115 (0002)    FBLK:   DS      2       ;BLOCK NO. FOR TRANSFER
 8117 (0002)    FBUF:   DS      2       ;BUFFER POINTER FOR TRANSFER
 8119 (0002)    FXBC:   DS      2       ;BYTE COUNT FOR TRANSFER
 811B (0002)    FPTR:   DS      2       ;BBUF PNTR FOR SEQUENTIAL ROUTINES
 811D           FPBE:                   ;END OF SYSTEM FPB

 811D           DBF:                    ;DIR BLOCK BUFFER
 811D (0001)    DBLK:   DS      1       ;"THIS" DIR BLOCK NUMBER
 811E (0001)    MDBLK:  DS      1       ;MAX. DIR BLOCK NUMBER
 811F (007E)            DS      126     ;REMAINDER OF 128. BYTE DIR BLOCK BUFFER
 819D           DBFE:                   ;END OF DIR BLOCK BUFFER

                ; THE FOLLOWING 18. BYTES ARE THE DIR BLOCK BUFFER EXTENSION
                ; USED BY CLOSE WHEN THE "FREE" ENTRY MOVES TO THE NEXT BLOCK:

 819D (0002)    XFHAN:  DS      2       ;AUX. HANDLER ADDRESS
 819F (0001)    XFFCN:  DS      1       ;AUX. HANDLER FUNCTION CODE
 81A0 (0001)    XFDRV:  DS      1       ;AUX. DRIVE NUMBER
 81A1 (0002)    XFBLK:  DS      2       ;AUX. BLOCK NUMBER
 81A3 (0002)    XFBUF:  DS      2       ;AUX. BUFFER POINTER
 81A5 (0002)    XFXBC:  DS      2       ;AUX. BYTE COUNT
 81A7 (0004)            DS      4       ; *** USED BY COPY ***
 81AB (0002)    TMP1:   DS      2       ; USED BY COPY & MAYBE OTHERS ?
 81AD 78        EMESS:  MOV     A,B     ;COPY STATUS CODE

      (26A5)    RESET   EQU     26A5H
      (26E7)    CKEND   EQU     26E7H
      (2C0C)    GETTO   EQU     2C0CH
      (2C86)    OPENX   EQU     2C86H
      (2C89)    OPENY   EQU     2C89H

                ; DIRECTORY ACCESS ROUTINES :
```

```
        ;+
        ; OPDIR - "OPEN" DIRECTORY FOR A SCAN.
        ;
        ; INPUTS  - D&E: PNTR TO FHAN IN FPB
        ; OUTPUTS - <C>  : DIRECTORY READ ERROR,
        ;                  OR NO VOLUME ENTRY
        ;           <NC> : NO ERRORS AND :
        ;           A    : ATR BYTE OF THIS ENTRY (=41H)
        ;           B    : INTERNALLY MAINTAINED ENTRY COUNTER
        ;           D&E  : PNTR TO FHAN IN FPB (UNCHANGED)
        ;           H&L  : PNTR TO "VOLUME" ENTRY IN DIRECTORY
        ;-

(2D60)  OPDIR   EQU     2D60H
        ;+
        ; GNDE  - GET NEXT DIRECTORY ENTRY
        ;
        ; INPUTS  - B    : INTERNALLY MAINTAINED ENTRY COUNTER
        ;           D&E  : PNTR TO FHAN IN FPB
        ;           H&L  : PNTR TO CURRENT DIRECTORY ENTRY
        ; OUTPUTS - <C>  : DIRECTORY READ ERROR
        ;           <Z>  : END OF DIRECTORY
        ;        <NC,NZ> : NO ERRORS AND :
        ;           A    : ATTRIBUTE BYTE OF THIS ENTRY
        ;           B    : UPDATED, MUST BE PRESERVED
        ;           D&E  : PNTR TO FHAN IN FPB (UNCHANGED)
        ;           H&L  : PNTR TO NEXT DIRECTORY ENTRY
        ;-

(2D86)  GNDE    EQU     2D86H

        ;+
        ; OPEN - OPEN A FILE FOR INPUT OR OUTPUT
        ;-

(2DAB)  OPEN    EQU     2DABH
        ;+
        ; READ - READ FILE. CURRENTLY ONLY "IMAGE"
        ;        TYPE FILES ARE SUPPORTED.
        ;        "OPEN" MUST BE CALLED BEFORE CALLING "READ".
        ;
        ; INPUTS  - B&C: MAX. ALLOWABLE BYTE COUNT
        ;           D&E: IF ZERO, THEN FILE WILL BE READ IN TO
        ;                MEMORY AT LOAD ADDRESS SPECIFIED IN
        ;                DIRECTORY. IF NON-ZERO, THEN D&E IS
        ;                USED AS THE LOAD ADDRESS.
        ;           H&L: PNTR TO FPB.
        ;
        ;-

(2EA3)  READ    EQU     2EA3H
        ;+
        ; WRITE - FILE WRITE. CURRENTLY ONLY "IMAGE" TYPE
        ;         FILES ARE SUPPORTED. "OPEN" MUST BE
        ;         CALLED BEFORE CALLING "WRITE".
        ;
```

```
        ;-

(2ECC)  WRITE   EQU     2ECCH

(2EF8)  WR      EQU     2EF8H

(2EFB)  RD      EQU     2EFBH
        ;+
        ; CLOSE - FILE CLOSE ROUTINE.
        ;
        ;-

(2F26)  CLOSE   EQU     2F26H

        ;+
        ; PDV - PARSE DEVICE NAME
        ; <C> : INVALID DEVICE
        ; <Z> : NO DEVICE, DEFAULT USED
        ; OTHERWISE: GOT VALID DEVICE
        ;-

(2FDE)  PDV     EQU     2FDEH
(3077)  PFSPC   EQU     3077H

        ;+
        ;    THIS IS AN INITIAL SET OF ROUTINES TO FACILITATE FILE
        ; OPERATIONS WITH THE FCS SYSTEM. ROUTINES ARE PROVIDED
        ; FOR SEQUENTIAL BYTE ACCESS, SEQUENTIAL RECORD ACCESS,
        ; AND BLOCK ACCESS.
        ;
        ;    CERTAIN PARAMETERS IN THE FILE PARAMETER BLOCK (FPB)
        ; MUST BE SET UP BEFORE USING ANY OF THESE ROUTINES. THESE
        ; PARAMETERS SHOULD BE SET UP AFTER CALLING 'OPEN' TO OPEN
        ; THE FILE, BUT BEFORE ANY CALL TO ANY OF THESE ACCESS
        ; ROUTINES.
        ;
        ;    THE PARAMETERS ARE :
        ;
        ; FBUF - SHOULD BE SET TO THE ADDRESS OF THE USER-PROVIDED
        ;         BLOCK BUFFER.
        ;
        ; FXBC - FOR SEQUENTIAL ACCESS, SHOULD BE SET TO THE SIZE
        ;         (NUMBER OF BYTES) OF THE USER-PROVIDED BLOCK BUFFER.
        ;         THE SIZE SHOULD BE A MULTIPLE OF THE SYSTEM STANDARD
        ;         BLOCK SIZE, 128 DECIMAL.
        ;        FOR DIRECT BLOCK ACCESS, SHOULD BE SET TO THE NUMBER
        ;         OF BYTES TO BE TRANSFERRED.
        ;
        ;    WHEN USING SEQUENTIAL BYTE OR RECORD ACCESS, THESE
        ; PARAMETERS SHOULD NOT BE CHANGED DURING THE I/O OPERATIONS.
        ;    WHEN USING DIRECT BLOCK ACCESS, THESE PARAMETERS MAY
        ; BE SET TO THE DESIRED VALUES FOR THE TRANSFER, PRIOR TO
        ; EACH CALL TO RBLK, RBLKI, WBLK, OR WBLKI. THEY'RE VALUES
        ; ARE PRESERVED BY THE BLOCK I/O ROUTINES, SO THEY
        ; DO NOT NEED TO BE RESET PRIOR TO EACH CALL UNLESS
        ; YOU SPECIFICALLY WANT TO CHANGE EITHER THE LOCATION
        ; OR THE SIZE OF THE BUFFER.
```

```
        ;
        ;   SEE THE INDIVIDUAL DESCRIPTION FOR DETAILS ON EACH
        ; OF THE FOLLOWING ACCESS ROUTINES.
        ;-

        ;+
        ; *** RWSEQI *** - "REWIND SEQUENTIAL INPUT" ROUTINE
        ;
        ;  RWSEQI IS USED TO "REWIND" A SEQUENTIAL INPUT FILE.
        ;
        ;  RWSEQI MUST BE CALLED BEFORE THE FIRST CALL TO ANY
        ;  OF THE SEQUENTIAL BYTE OR RECORD INPUT ROUTINES !
        ;
        ; INPUTS:  HL => FPB
        ;
        ; OUTPUTS: A - LOST
        ;          BC,DE - UNCHANGED
        ;          HL => FPB
        ;
        ; STATUS:  NONE
        ;-

(30C6)  RWSEQI  EQU     30C6H

        ;+
        ; *** INSEQO *** "INITIALIZE SEQUENTIAL OUTPUT" ROUTINE
        ;
        ;  INSEQO IS USED TO INITIALIZE A NEWLY CREATED OPEN FILE
        ;  FOR SEQUENTIAL BYTE OR RECORD OUTPUT OPERATIONS.
        ;
        ;  INSEQO MUST BE CALLED BEFORE THE FIRST CALL TO ANY OF
        ;  THE SEQUENTIAL BYTE OR RECORD OUTPUT ROUTINES !
        ;
        ; INPUTS:  HL => FPB
        ;
        ; OUTPUTS: A - LOST
        ;          BC,DE - UNCHANGED
        ;          HL => FPB
        ;
        ; STATUS:  NONE
        ;-

(30E7)  INSEQO  EQU     30E7H

        ;+
        ; *** CLSEQO *** "CLOSE SEQUENTIAL OUTPUT" ROUTINE
        ;
        ;  CLSEQO IS USED TO CLOSE A NEWLY CREATED SEQUENTIAL
        ;  OUTPUT FILE. THE REMAINING UNWRITTEN PART OF THE BLOCK
        ;  BUFFER IS WRITTEN OUT, IF ANY, AND THE FINAL FILE SIZE
        ;  IS CALCULATED AND APPROPRIATE INFORMATION UPDATED IN
        ;  THE FPB. THEN 'CLOSE' IS CALLED TO ENTER THE FILE INTO
        ;  THE DIRECTORY.
        ;
        ; INPUTS:  HL => FPB
        ;
        ; OUTPUTS: A,BC,DE - LOST
```

```
        ;           HL => FPB IF NO ERRORS, ELSE HL LOST
        ;
        ; STATUS:   <NC> - NO ERRORS, B=0
        ;           <C>  - FILE WRITE ERROR OR DIRECTORY WRITE ERROR,
        ;                  WITH B = SYSTEM ERROR CODE
        ;-

(3136)  CLSEQO  EQU     3136H
        ;+
        ; *** RBLK *** "READ BLOCK" ROUTINE
        ; *** WBLK *** "WRITE BLOCK" ROUTINE
        ;
        ;  RBLK AND WBLK ARE USED TO READ/WRITE TO/FROM A
        ;  SPECIFIED VIRTUAL BLOCK NUMBER IN A FILE.
        ;
        ; INPUTS:  HL => FPB
        ;        FBLK =  DESIRED STARTING VIRTUAL BLOCK NUMBER
        ;        FBUF => BLOCK BUFFER
        ;        FXBC = NUMBER OF BYTES TO READ/WRITE
        ;
        ; OUTPUTS: A - LOST
        ;          BC,DE - UNCHANGED
        ;          HL => FPB
        ;          FBLK,FBUF,FXBC - UNCHANGED
        ;
        ; STATUS:  <NC><Z> - NO ERRORS:
        ;            FAUX= NUMBER OF BYTES TRANSFERRED = (FXBC).
        ;          <NC><NZ> - TRANSFER TRUNCATED BY END-OF-FILE
        ;            FAUX= NUMBER OF BYTES TRANSFERRED < (FXBC).
        ;          <C><Z> - VIRTUAL BLOCK NOT WITHIN FILE:
        ;            FAUX UNCHANGED.
        ;          <C><NZ><M> - READ/WRITE ERROR:
        ;            FAUX = NUMBER OF BYTES ATTEMPTED.
        ;-

(317F)  WBLK    EQU     317FH
(3182)  RBLK    EQU     3182H

        ;+
        ; *** RBLKI *** "READ BLOCK & INCREMENT" ROUTINE
        ; *** WBLKI *** "WRITE BLOCK & INCREMENT" ROUTINE
        ;
        ;  RBLKI AND WBLKI ARE IDENTICAL IN FUNCTION TO
        ;  RBLK AND WBLK EXCEPT: IF NO ERRORS OCCURRED, THEN
        ;  FBLK IS SET TO THE NEXT VIRTUAL BLOCK NUMBER
        ;  FOR SEQUENTIAL ACCESS. IF ANY ERRORS OCCURRED,
        ;  THEN FBLK IS UNCHANGED.
        ;-

(31F6)  WBLKI   EQU     31F6H
(31F9)  RBLKI   EQU     31F9H

        ;+
        ; *** GTBYT *** "GET BYTE" ROUTINE
        ;
        ;  GTBYT IS USED TO READ SEQUENTIAL BYTES FROM AN OPEN
        ;  FILE.
```

```
        ;
        ;  RWSEQI MUST HAVE BEEN CALLED BEFORE THE FIRST CALL
        ;  TO GTBYT !
        ;
        ; INPUTS:  HL => FPB
        ;
        ; OUTPUTS: A = THE BYTE, IF NO ERRORS
        ;          BC,DE - UNCHANGED
        ;          HL => FPB
        ;
        ; STATUS:  <NC> - NO ERRORS, A= THE BYTE
        ;          <C><Z> - END OF FILE
        ;          <C><NZ><M> - READ ERROR
        ;-
(322C)  GTBYT   EQU     322CH

        ;+
        ; *** PTBYT *** "PUT BYTE" ROUTINE
        ;
        ;  PTBYT IS USED TO WRITE SEQUENTIAL BYTES TO AN OPEN FILE.
        ;
        ;  INSEQO MUST BE CALLED AFTER OPEN AND BEFORE THE FIRST
        ;  CALL TO PTBYT !
        ;
        ; INPUTS:  A = THE BYTE
        ;          HL => FPB
        ;
        ; OUTPUTS: A = THE BYTE
        ;          BC,DE - UNCHANGED
        ;          HL => FPB
        ;
        ; STATUS:  <NC> - NO ERRORS
        ;          <C><Z> - FILE FULL, BYTE NOT WRITTEN
        ;          <C><NZ><M> - WRITE ERROR
        ;-

(324A)  PTBYT   EQU     324AH

        ;+
        ; *** GAREC *** "GET ASCII RECORD" ROUTINE
        ;
        ;  GAREC IS USED TO READ SEQUENTIAL RECORDS FROM AN
        ;  ASCII FILE. A RECORD IS A STRING OF ASCII CHARACTERS
        ;  TERMINATED BY EITHER A LINE FEED (10.) OR A
        ;  FORM FEED (12.) .
        ;
        ;  RWSEQI MUST HAVE BEEN CALLED BEFORE THE FIRST CALL
        ;  TO GAREC !
        ;
        ; INPUTS:  HL => FPB
        ;          BC => RECORD BUFFER
        ;          DE =  RECORD BUFFER LENGTH (BYTES)
        ;
        ; OUTPUTS: HL => FPB
        ;          BC => PAST LAST BYTE STORED
        ;          DE = NUMBER OF BYTES READ
        ;          A - LOST
```

```
                ;
                ; STATUS:  <NC> - NO ERRORS
                ;          <C><Z> - END OF FILE
                ;          <C><NZ><M> - READ ERROR
                ;          <C><NZ><P> - BUFFER WAS FILLED, BUT A VALID
                ;                       TERMINATOR WAS NOT SEEN. THE NEXT
                ;                       CALL TO GAREC WILL START WITH THE
                ;                       NEXT SEQUENTIAL BYTE.
                ;-

       (3257)   GAREC   EQU     3257H
                ;+
                ; *** PVREC *** "PUT VARIABLE LENGTH RECORD" ROUTINE
                ;
                ;  PVREC IS USED TO PUT A VARIABLE LENGTH RECORD INTO A
                ;  SYSTEM STANDARD 'VARIABLE LENGTH RECORD, SEQUENTIAL' FILE.
                ;  WITHIN THE FILE, EACH RECORD CONSISTS OF A TWO BYTE BYTE
                ;  COUNT, LOW BYTE FIRST, FOLLOWED BY THAT NUMBER OF DATA
                ;  BYTES.
                ;
                ; INPUTS:  HL => FPB
                ;          BC => RECORD BUFFER
                ;          DE = RECORD LENGTH (BYTES)
                ;
                ; OUTPUTS: HL => FPB
                ;          BC => PAST LAST BYTE IN RECORD BUFFER
                ;          DE = 0 IF NO ERRORS
                ;          A - LOST
                ;
                ; STATUS: <NC> - NO ERRORS
                ;          <C><Z> - TRANSFER TERMINATED BY END OF FILE - FILE FU
LL
                ;          <C><NZ><M> - WRITE ERROR
                ;-

       (327B)   PVREC   EQU     327BH

                ;+
                ; *** PTREC *** "PUT UNFORMATTED RECORD" ROUTINE
                ;
                ;  PTREC IS USED TO PUT AN 'UNFORMATTED' RECORD INTO A
                ;  SEQUENTIAL FILE. OPERATION OF PTREC IS IDENTICAL TO
                ;  PVREC, ABOVE, EXCEPT THAT THE TWO BYTE BYTE COUNT
                ;  IS NOT WRITTEN INTO THE FILE.
                ;-

       (3285)   PTREC   EQU     3285H
  81AE (0000)           END
```

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.
nnnnB	nnnn represents a number in binary format.
nnnnD	nnnn represents a number in decimal format.
nnnnO	nnnn represents a number in octal format.
nnnnQ	nnnn represents a number in octal format.
nnnnH	nnnn represents a number in hexadecimal format.
0 0 1 1 | rp | 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.

TABLE OF CONTENTS

Table of Content

LIST OF FIGURES

CHAPTER 1 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.

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:

Register Pair	Registers Referenced
B	B and C (0 and 1)
D	D and E (2 and 3)
H	H and L (4 and 5)
PSW	See below

Register pair PSW (Program Status Word) refers to register A (7) and a special byte which reflects the current status of the machine flags. This byte is described in detail in Chapter 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 Chapter 2 in the section on Data Statements.

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 this chapter under Computer Program Representation in Memory.

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.*

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 Chapter 2 under Input/Output Instructions.

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 Chapter 2.

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 1-1.

Memory Address	Instruction Number	Program Counter Contents
0212	1	0213
0213		0215
0214	2	
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 1-1. 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 1-1, 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 1-1, 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 Chapter 2, 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.

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.

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.

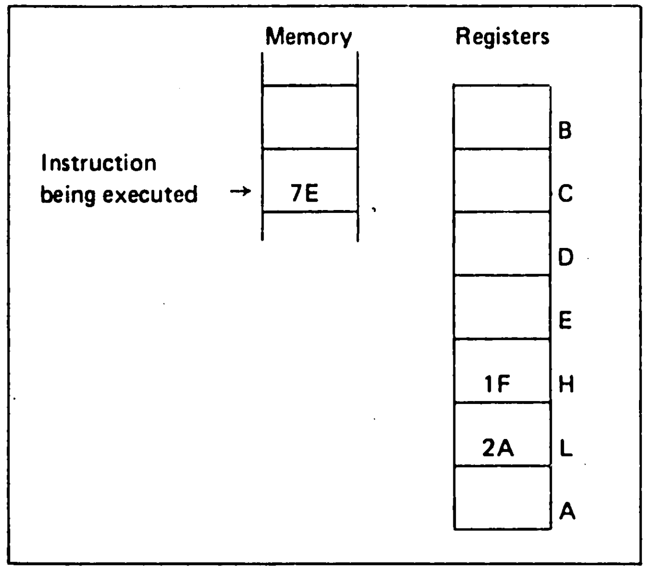
This would appear in memory as follows:

Memory Address	Memory	
any	3A	instruction being executed
any + 1	2A	
any + 2	1F	

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

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 Chapter 2 under Data Transfer Instructions.

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

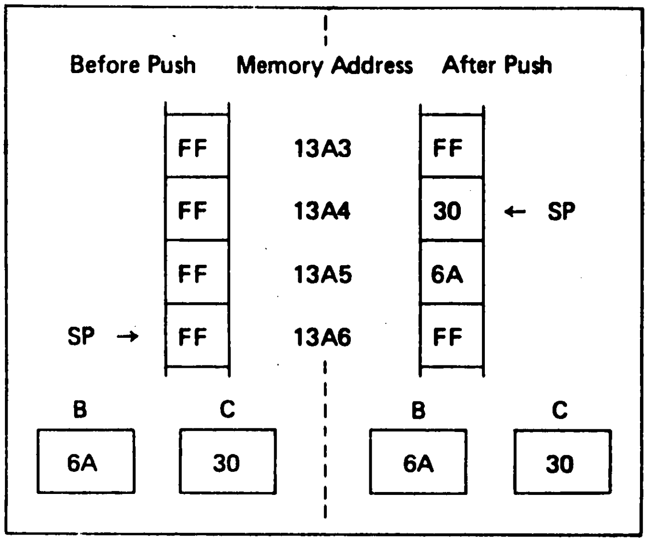
16 bits of data are transferred to a memory area (called a stack) from a register pair or the 16 bit program counter during any stack push operation. The addresses of the memory area which is to be accessed during a stack push operation are determined by using the stack pointer as follows:

(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:



STACK POP OPERATION

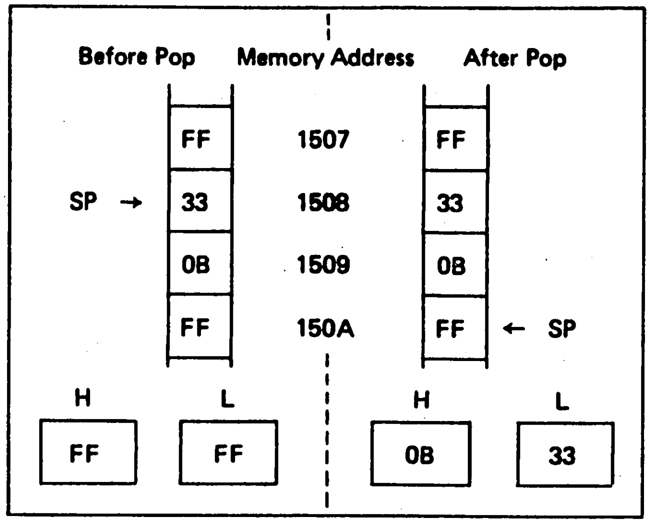
16 bits of data are transferred from a memory area (called a stack) to a register pair or the 16-bit program counter during any stack pop operation. The addresses of the memory area which is to be accessed during a stack pop operation are determined by using the stack pointer as follows:

(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:



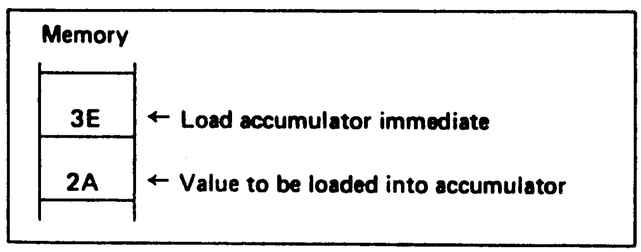
The programmer loads the stack pointer with any desired value by using the LXI instruction described in Chapter 2 under Load Register Pair-Immediate. The programmer must initialize the stack pointer before performing a stack operation, or erroneous results will occur.

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:

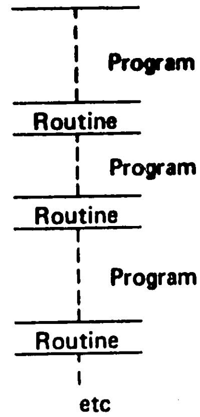


Immediate instructions do not reference memory; rather they contain data in the memory byte following the instruction code byte.

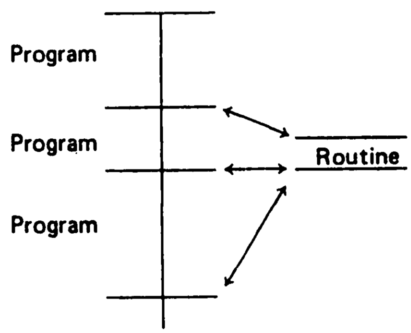
Subroutines and Use of the Stack for Addressing

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:

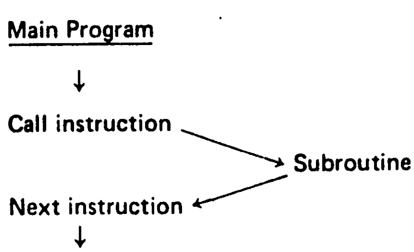


A more efficient means of accessing the routine would be to store it once, and find a way of accessing it when needed:



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, 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		Push address of next instruction (0C06H) onto the stack and branch to subroutine starting at 0F02H
0C03	CALL SUBROUTINE	
0C04	02	
0C05	0F	
0C06	NEXT INSTRUCTION	
0F00		
0F01		
0F02	FIRST SUBROUTINE INSTRUCTION	
0F03		
—		
—	Body of subroutine	
—		Pop return address (0C06H) off stack and return to next instruction
—		
0F4E		
0F4F	RETURN	

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.

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 Chapter 2 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.

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 a carry out of the high-order bit:

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

→ carry-out = 1, sets Carry Bit = 1

An addition operation that results in a carry out of the high-order bit will set the Carry bit; an addition operation that could have resulted in a carry out 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 Chapter 2 under Two's Complement Representation and the individual instruction descriptions in Chapter 2 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.

Auxiliary Carry Bit

The Auxiliary Carry bit indicates carry out of bit 3. 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 Chapter 2) 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

  2E=    0 0 1 0 1 1 1 0
+ 74=    0 1 1 1 0 1 0 0
  A2     1 0 1 0 0 0 1 0
         └→Carry=0 └→Auxiliary Carry=1
```

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

Sign Bit

As described in Chapter 2 under Two's Complement Representation, 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 Chapter 2), the Sign bit will be set to the condition of the most significant bit of the answer (bit 7).

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]  0 0 0 0 0 0 0 0

Carry out /           Zero answer
of bit 7./
                      Zero bit set to 1.
```

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.

CHAPTER 2 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, Chapter 2 describes individual instructions with examples and machine code equivalents.

ASSEMBLY LANGUAGE

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	New Instruction
1434	C4	C3
1435	14	C5
1436	•	14
1437	•	•
14C3	FF	•
14C4	2E	FF
14C5	36	2E
14C6	77	37
14C7		77

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 its operator's manual.

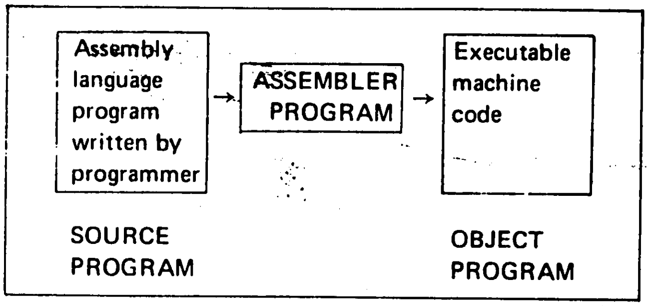


Figure 2-1. Assembler Program Converts Assembly Language Source Program to Object Program

As illustrated in Figure 2-1, 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:

```
                                        One Possible
                                        Version of the
Source Program                          Object Program

NOW:  MOV   A,B                             78
      CPI   'C'   → is converted        →   FE43
      JZ    LER     by the                  CA7C3D
       :            Assembler                :
LER:  MOV   M,A     to                      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 later in this chapter.

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.

Statement Syntax

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 a name used to reference the instruction's address.

Field 2 is the CODE field. It specifies the operation that is to be performed.

Field 2 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:

Label	Code	Operand	
HERE:	MVI	C,0	; Load the C register with 0
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.

Label Field

This is an optional field, which, if present, may be ANY NUMBER OF characters long. The first character of the label must be a letter of the alphabet

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 and pseudo-instructions are given later in this chapter and Appendix A.

Here are some examples of valid label fields:

LABEL:

F14F:

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

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.

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 later in this chapter are each identified by a mnemonic label which must appear in the code field. For example, since the "jump" instruction is identified by the letters "JMP," these letters must appear in the code field to identify the instruction as "jump."

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

```
HERE:      JMP      THERE
```

is legal, but:

```
HERE       JMPTHERE
```

is illegal.

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	
Information required	Ways of specifying
(a) Register	(1) Hexadecimal Data
(b) Register Pair	(2) Decimal Data
(c) Immediate Data	(3) Octal Data
(d) 16-bit Memory Address	(4) Binary Data
	(5) Program Counter ($)
	(6) ASCII Constant
	(7) Labels assigned values
	(8) Labels of instructions
	(9) Expressions
	(10) Register or Register Pair Specification

The ten 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:

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

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

Example:

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

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

Example:

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

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

Example:

Label	Code	Operand	Comment
NOW:	MVI	10B,11110110B	; Load register two ; (the 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:

Label	Code	Operand
GO:	JMP	$+6

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

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

Example:

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

(7) Labels that have been assigned a numeric value by a SET or EQU directive

Example:

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

Label	Code	Operand
A1:	MVI	D, VALUE
A2:	MVI	D, 9FH

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

Example:

Label	Code	Operand	Comment
HERE:	JMP	THERE	; Jump to instruction ; at THERE

THERE:	MVI	D, 9FH	

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

All operators treat their arguments as 15-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. Zeros 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.

Examples:

Label	Code	Operand	Arbitrary Memory Address
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.

Label	Code	Operand
NEXT:	MVI	D, 34+4 OH/2

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

Operators cause expressions to be evaluated in the following order:

1. Parenthesized expressions
2. *,/M, 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 ANDOFH

is invalid. See Page 11 A

Using some or all of the above ten 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 –

Specification	Register
B	B
C	C
D	D
E	E
H	H
L	L
M	Memory Reference
A	A (accumulator)

Example:

Label	Code	Operand
INS1:	MVI	A, 2EH
INS2:	MVI	B, 17
INS3:	MVI	C, XYZ

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

Specification	Register Pair
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

(10) **A register or register pair specification is a letter or group of letters used to specify a register, register pair, or memory reference operand.**

Specification	Meaning
A	A register
B	B register or B and C register pair
C	C register
D	D register or D and E register pair
E	E register
H	H register or H and L register pair
L	L register
M	a memory reference (the memory address specified by the contents of the H and L register pair.)
SP	The 16-bit stack pointer register
PSW	The "register pair" consisting of the A register and the state of the condition bits

Example:

Label	Code	Operand	Comment
	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:

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

Here are some examples of the form DATA could take:

ADDR AND 0FFH (where ADDR is a 16-bit address)
127
'*'
VALUE (where VALUE has been equated to a number)
3EH=10/2 (2 AND 2)

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

Example:

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

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
;
```

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 00011111.

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.

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:

0 = 00000000B = 0H
1 = 00000001B = 1H
•
•
•
126D = 01111110B = 7EH
127D = 01111111B = 7FH

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.

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

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

Take the two's complement of 0CH=F4H

Add the result to the minuend:

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

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

```
   0 = 0 0 0 0 0 0 0 0 B = 0H
   1 = 0 0 0 0 0 0 0 1 B = 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. *Thus, the Carry bit when set indicates the occurrence of a "borrow."*

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

```
                197D = 1 1 0 0 0 1 0 1 = C5H
                -98D = 1 0 0 1 1 1 1 0 = 9EH
                       ---------------
carry out →   1        0 1 1 0 0 0 1 1 = 63H = 99D
```

Since the carry 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.

```
                12D = 0 0 0 0 1 1 0 0 = 0CH
               -15D = 1 1 1 1 0 0 0 1 = 0F1H
                      ---------------
carry out →   0       1 1 1 1 1 1 0 1 = -3D
```

Since the carry 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 indicating that a "borrow" occurred.

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

```
   +12D = 0 0 0 0 1 1 0 0
+(-15D) = 1 1 1 1 0 0 0 1
          ---------------
      0   1 1 1 1 1 1 0 1 = -3D
          causes carry to be reset.
```

"SUB" +15D from +12D

```
   +12D = 0 0 0 0 1 1 0 0
-(+15D) = 1 1 1 1 0 0 0 1
          ---------------
      1   1 1 1 1 1 1 0 1 = -3D
          causes carry to be set
```

DB Define Byte(s) of Data

Label	Code	Operand
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:

Instruction	Assembled Data (hex)
HERE: DB 0A3H	A3
WORD1: DB 5*2, 2FH-0AH	0A25
WORD2: DB 5ABCH SHR 8	5A
STR: DB 'STRINGSpl'	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.

DW Define Word (Two Bytes) of Data

Format:

Label	Code	Operand
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.

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

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

DS Define Storage (Bytes)

Format:

Label	Code	Operand
oplab:	DS	exp

"exp" is a single arithmetic or logical expression.

Description: The value of EXP specifies the number of memory bytes to be reversed 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
```

CARRY BIT INSTRUCTIONS

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

0	0	1	1	X	1	1	1

X: 0 for STC, 1 for CMC

The general assembly language format is:

Label	Code	Operand
LABEL:	OP	

- LABEL: Optional instruction label
- OP: STC or CMC
- Operand: not used

CMC Complement Carry

Format:

Label	Code	Operand
oplab:	CMC	—

0	0	1	1	1	1	1	1

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

STC Set Carry

Format:

Label	Code	Operand
oplab:	STC	—

0	0	1	1	0	1	1	1

Description: The Carry bit is set to one.

Condition bits affected: Carry

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.

INR Increment Register or Memory

Format:

Label	Code	Operand
oplab:	INR	reg

↑ B,C,D,E,H,L,M or A

0	0	reg			1	0	0

- 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 ref. M
- 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

DCR Decrement Register or Memory

Format:

Label	Code	Operand
oplab:	DCR	reg

↑ B,C,D,E,H,L,M or A

0	0	reg			1	0	1

- 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 ref. M
- 111 for register A

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

Condition bits affected: Zero, Sign, Parity, Auxiliary Carry

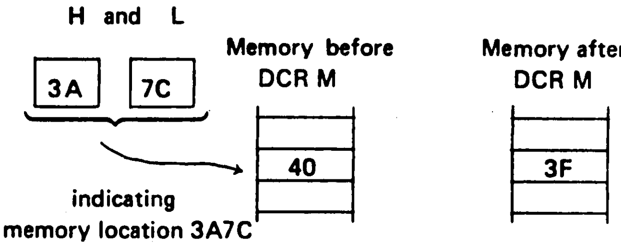
Example:

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



CMA Complement Accumulator

Format:

Label	Code	Operand
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

$\overline{\text{Accumulator}}$ = 1 0 1 0 1 1 1 0 = AEH

DAA Decimal Adjust Accumulator

Format:

Label	Code	Operand
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 represents 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 reset. 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

Accumulator = 1 0 0 1   1 0 1 1 = 9BH
  +6        =           0 1 1 0
              1 0 1 0   0 0 0 1 = A1H
                        ↓
                         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

Accumulator = 1 0 1 0 0 0 0 1 = A1H
          +6 = 0 1 1 0
          1  0 0 0 0 0 0 0 1
          └→ 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 Chapter 4.

NOP INSTRUCTIONS

The NOP instruction occupies one byte.

Format:

Label	Code	Operand
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

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:

```
|0|1| dst | src |
```

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:

```
|0|0|0|x|x|0|1|0|
```

0 for register pair B — 0 for STAX
1 for register pair D — 1 for LDAX

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:

Label	Code	Operand
oplab:	MOV	dst, src

dst, src: A,B,C,D,E,H,L, or M (dst and src not both = M)
oplab: Optional instruction label

– or –

Label	Code	Operand
oplab:	OP	rp

rp: B or D
OP: STAX or LDAX
oplab: Optional instruction label

MOV Instruction

Format:

Label	Code	Operand
oplab:	MOV	dst, src

```
|0|1| dst | src |
```

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 remains unchanged.

Condition bits affected: None

Example 1:

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

NOTE: Any of the null operation instructions MOV X,X can also be specified as 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.

STAX Store Accumulator

Format:

Label	Code	Operand
oplab:	STAX	rp

0	0	0	X	0	0	1	0

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.

LDAX Load Accumulator

Format:

Label	Code	Operand
oplab:	LDAX	rp

0	0	0	X	1	0	1	0

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.

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:

1	0	op	reg

op		reg	
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 M
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:

Label	Code	Operand
oplab:	op	reg

- reg: A,B,C,D,E,H,L, or M
- op: ADD, ADC, SUB, SBB, ANA, XRA, ORA or CMP
- oplab: Optional instruction label

ADD ADD Register or Memory To Accumulator

Format:

Label	Code	Operand
oplab:	ADD	reg

1	0	0	0	0	reg

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 A_3, the Auxiliary Carry bit is set. The accumulator now contains 9AH.

Example 2:

The instruction:

ADD A

will double the accumulator.

ADC ADD Register or Memory To Accumulator With Carry

Format:

Label	Code	Operand
oplab:	ACD	reg

```
| 1 | 0 | 0 | 0 | 1 | reg |
```

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:

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

The results can be summarized as follows:

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

SUB Subtract Register or Memory From Accumulator

Format:

Label	Code	Operand
oplab:	SUB	reg

```
| 1 | 0 | 0 | 1 | 0 | reg |
```

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

If there is *no* carry 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:

```
      3EH   = 0 0 1 1 1 1 1 0
  +(-3EH)   = 1 1 0 0 0 0 0 1  negate and add one
  +                         1  to produce two's
                               complement
carry  →  1   0 0 0 0 0 0 0 0  Result = 0
```

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

Since there was a carry out of bit A_3, the Auxiliary Carry bit will be set.

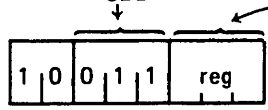
The Parity and Zero bits will also be set, and the Sign bit will be reset.

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

SBB Subtract Register or Memory From Accumulator With Borrow

Format:

Label	Code	Operand
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 the section on Multibyte Addition and Subtraction in Chapter 4.

Condition bits affected: Carry, Sign, Zero, Parity, Auxiliary Carry (see last section 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 procedures:

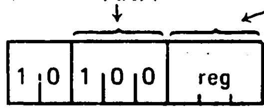
```
Accumulator = 04H = 0 0 0 0 0 1 0 0
                    1 1 1 1 1 1 0 1
                 1] 0 0 0 0 0 0 0 1 = 01H = Result
   carry out = 1 causing the Carry bit to be reset
```

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 a carry out of the high-order bit position. The Auxiliary Carry bit is set since there was a carry out of bit A_3. The Parity and the Sign bits are reset.

ANA Logical and Register or Memory With Accumulator

Format:

Label	Code	Operand
oplab:	ANA	reg



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:

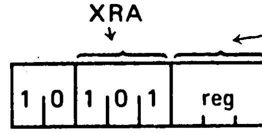
```
Accumulator = 1 1 1 1 1 1 0 0 = 0FCH
C Register  = 0 0 0 0 1 1 1 1 = 0FH
Result in
Accumulator = 0 0 0 0 1 1 0 0 = 0CH
```

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

XRA Logical Exclusive-Or Register or Memory With Accumulator (Zero Accumulator)

Format:

Label	Code	Operand
oplab:	XRA	reg



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, Auxiliary Carry

Example 1:

Since any bit EXCLUSIVE-ORed with itself produces zero, the EXCLUSIVE-OR can be used to zero the accumulator.

Label	Code	Operand
	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

will 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.

```
Label    Code   Operand
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 0 1 0 0 1 0 0
```

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

```
Result  =  0 0 1 0 0 1 0 0
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.

ORA Logical or Register or Memory With Accumulator

Format:

```
Label      Code      Operand
oplab:     ORA       reg

| 1 0 | 1 1 0 | 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.

CMP Compare Register or Memory With Accumulator

Format:

```
Label      Code      Operand
oplab:     CMP       reg

| 1 0 | 1 1 1 | reg |
```

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 carry 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:

```
  Accumulator    =  0AH  =  0 0 0 0 1 0 1 0
+ (-E Register)  =  -5H  =  1 1 1 1 1 0 1 1
                         1] 0 0 0 0 0 1 0 1 = result
  carry = 1, causing the Carry bit to be reset
```

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

Example 2:

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

```
   Accumulator    = 02H = 0 0 0 0 0 0 1 0
 + (-E Register)  = -5H = 1 1 1 1 1 0 1 1
                      0   1 1 1 1 1 1 0 1 = result
                          carry = 0, Carry bit = 1
```

The Zero bit would be reset and the Carry bit set, indicating E greater than A.

Example 3:

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

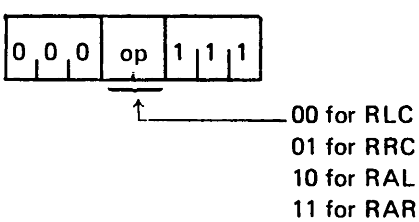
```
   Accumulator    = -1BH  = 1 1 1 0 0 1 0 1
 + (-E Register)  = -5H   = 1 1 1 1 1 0 1 1
                        1   1 1 1 0 0 0 0 0
                     carry = 1, causing carry to be reset
```

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

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:

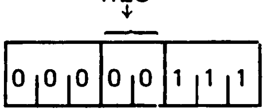
Label	Code	Operand
label:	op	——

- Operand: not used
- Code: RLC, RRC, RAL, or RAR
- Label: Optional instruction label

RLC Rotate Accumulator Left

Format:

Label	Code	Operand
oplab:	RLC	——



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

Condition bits affected: Carry

Example:

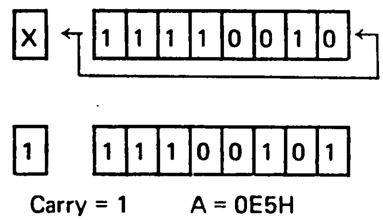
Assume that the accumulator contains 0F2H. Then the instruction:

RLC

acts as follows:

Before RLC is executed: Carry Accumulator

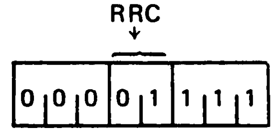
After RLC is executed:



RRC Rotate Accumulator Right

Format:

Label	Code	Operand
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:

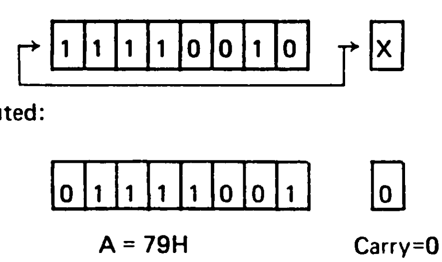
Assume that the accumulator contains 0F2H. Then the instruction:

RRC

acts as follows:

Before RRC is executed: Accumulator Carry

After RRC is executed:



RAL Rotate Accumulator Left Through Carry

Format:

Label	Code	Operand
oplab:	RAL	—

0	0	0	1	0	1	1	1

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 high-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

Carry	Accumulator
0	1 0 1 1 0 1 0 1

After RAL is executed:

Carry	Accumulator
1	0 1 1 0 1 0 1 0
Carry=1	A = 6AH

RAR Rotate Accumulator Right Through Carry

Format:

Label	Code	Operand
oplab:	RAR	—

0	0	0	1	1	1	1	1

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:

Before RAR is executed:

Accumulator	Carry
0 1 1 0 1 0 1 0	1

After RAR is executed:

Accumulator	Carry
1 0 1 1 0 1 0 1	0
A = 0B5H	Carry=0

REGISTER PAIR INSTRUCTIONS

This section describes instructions which operate on pairs of registers.

PUSH Push Data Onto Stack

Format:

Label	Code	Operand
oplab:	PUSH	rp

rp ← B,D,H, or PSW

1	1	rp	0	1	0	1

rp:
00 for registers B and C
01 for registers D and E
10 for registers H and L
11 for flags and register A

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 contents of the A register; the second byte holds the settings of the five condition bits, i.e., Carry, Zero, Sign, Parity, and Auxiliary Carry. The format of this byte is:

7	6	5	4	3	2	1	0
S	Z	0	AC	0	P	1	C

- S: State of Sign bit
- Z: State of Zero bit
- bit 5: always 0
- AC: State of auxiliary Carry bit
- bit 3: always 0
- P: State of Parity bit
- bit 1: always 1
- C: State of Carry bit

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

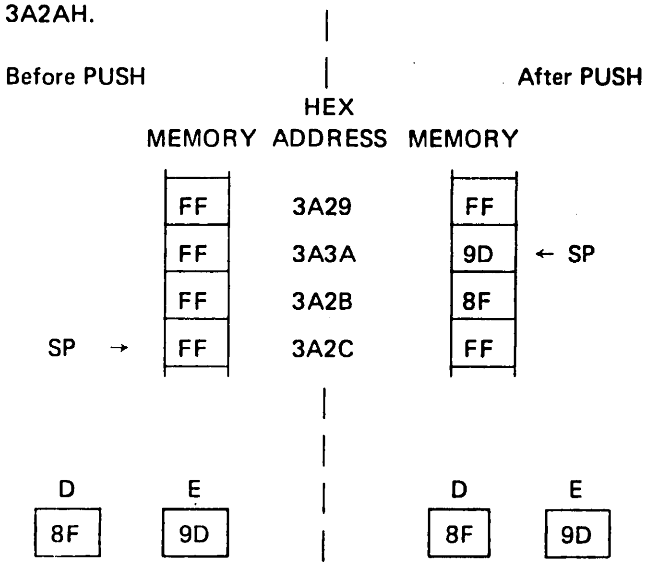
Condition bits affected: None

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.



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:

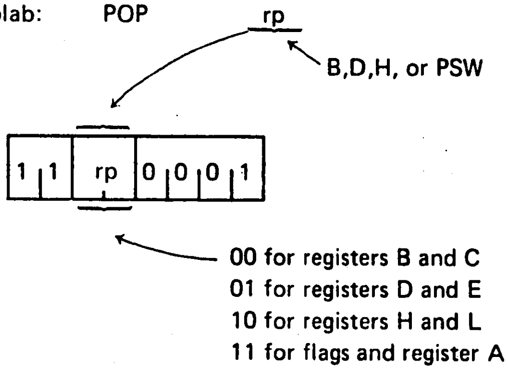
PUSH PSW

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

POP Pop Data Off Stack

Format:

Label	Code	Operand
oplab:	POP	rp



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 the last section.

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

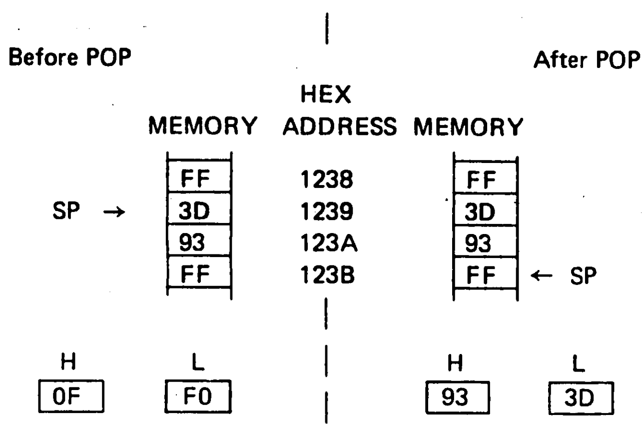
Condition bits affected: If register pair PSW is specified, Carry, Sign, Zero, Parity, and Auxiliary Carry may be changed. Otherwise, none are affected.

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 123BH.

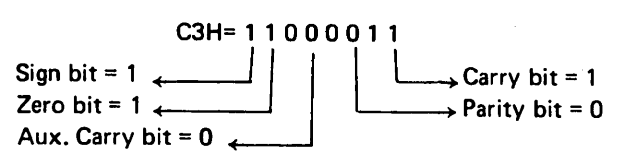


Example 2:

Assume that memory locations 2C00H and 2C01H contain C3H and FFH 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:



DAD Doubie Add

Format:

Label	Code	Operand
oplab:	DAD	rp

rp ← B,D,H, or SP

0	0	rp	1	0	0	1

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

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:

Registers B and C = 339F
\+ Registers H and L = A17B
New contents of H and L = D51A

Register H now contains D5H and register L now contains 1AH. Since no carry out 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).

INX Increment Register Pair

Format:

Label	Code	Operand
oplab:	INX	rp

rp ← B,D,H, or SP

0	0	rp	0	0	1	1

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

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.

DCX Decrement Register Pair

Format:

Label	Code	Operand
oplab:	DCX	rp

rp ← B,D,H, or SP

0	0	rp	1	0	1	1

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

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.

XCHG Exchange Registers

Format:

Label	Code	Operand
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	H	L
00	FF	33	55

XTHL Exchange Stack

Format:

Label	Code	Operand
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:

Before XTHL	MEMORY	HEX ADDRESS	MEMORY	After XTHL
	FF	10AC	FF	
SP →	F0	10AD	3C	← SP
	0D	10AE	0B	
	FF	10AF	FF	

Before XTHL		After XTHL	
H	L	H	L
0B	3C	0D	F0

SPHL Load SP From H And L

Format:

Label	Code	Operand
oplab:	SPHL	——

1	1	1	1	1	0	0	1

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.

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):

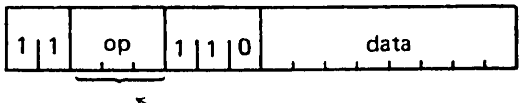
0	0	rp	0	0	0	1	low data	high data

rp:
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):

0	0	reg	1	1	0	data

reg:
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 ref. M
111 for register A



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:

Label	Code	Operand
oplab:	LXI	rp, data

- data: 16-bit data quantity
- rp: B, D, H, or SP
- oplab: Optional instruction label

—or—

Label	Code	Operand
oplab:	MVI	reg, data

- data: 8-bit data quantity
- reg: A,B,C,D,E,H,L, or M
- oplab: Optional instruction label

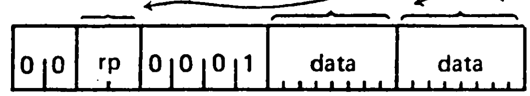
—or—

Label	Code	Operand
oplab:	OP	data

- data: 8-bit data quantity
- OP: ADI,ACI,SUI,SBI,ANI,XRI,ORI, or CPI
- oplab: Optional instruction label

Format:

Label	Code	Operand
oplab:	LXI	rp, data



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:

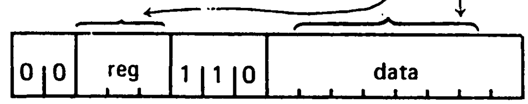
The following instruction loads the stack pointer with the value 3ABCH:

```
LXI  SP,3ABCH
```

MVI Move Immediate Data

Format:

Label	Code	Operand
oplab:	MVI	reg, data



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

Condition bits affected: None

Example

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

The instructions at M1 loads the Hregister with the byte of data at M1 + 1, i.e., 3CH.

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

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

LXI H, 3CF4H

ADI Add Immediate To Accumulator

Format:

Label	Code	Operand
oplab:	ADI	data

1	1	0	0	0	1	1	0
data							

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:

Label	Code	Operand	Assembled Data
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.

ACI Add Immediate To Accumulator With Carry

Format:

Label	Code	Operand
oplab:	ACI	data

1	1	0	0	1	1	1	0
data							

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:

Label	Code	Operand	Assembled Data
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

SUI Subtract Immediate From Accumulator

Format:

Label	Code	Operand
oplab:	SUI	data

1	1	0	1	0	1	1	0
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 carry out of the high-order bit position, and reset if there is a carry out.

Condition bits affected: Carry, Sign, Zero, Parity, Auxiliary Carry

Example:

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

Label	Code	Operand	Assembled Data
	MVI	A, 0	3E00
S1:	SUI	1	D601

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 carry, 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.

SBI Subtract Immediate from Accumulator With Borrow

Format:

Label	Code	Operand
oplab:	SBI	data

1	1	0	1	1	1	1	0	data

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 the section on Multibyte Addition and Subtraction in Chapter 4.

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

Condition bits affected: Carry, Sign, Zero, Parity, Auxiliary Carry

Example:

Label	Code	Operand	Assembled Data
	XRA	A	AF
	SBI	1	DE01

The XRA instruction will zero the accumulator (see example earlier in this chapter). 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
            └→ carry out = 0 causing the Carry bit 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
                   11111111
            ┌──────11111110 = -2H = Result
            └→ carry out = 0 causing the Carry bit to be set
```

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

ANI And Immediate With Accumulator

Format:

Label	Code	Operand
oplab:	ANI	data

1	1	1	0	0	1	1	0	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:

Label	Code	Operand	Assembled Data
	MOV	A, C	79
A1:	ANI	0FH	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) = 0FH = 00001111
                 Result =       00001010 = 0AH
```

XRI Exclusive-Or Immediate With Accumulator

Format:

Label	Code	Operand
oplab:	XRI	data

```
| 1 1 | 1 0 1 | 1 1 0 | data |
```

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
```

ORI Or Immediate With Accumulator

Format:

Label	Code	Operand
oplab:	ORI	data

```
| 1 1 | 1 1 0 | 1 1 0 | data |
```

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

The result is stored in the accumulator. The Carry bit is reset to zero, while the Zero, Sign, and Parity bits are set according to the result.

Condition bits affected: Carry, Zero, Sign, Parity

Example:

Label	Code	Operand	Assembly Data
	MOV	A,C	79
OR1:	ORI	0FH	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 (OR1 Immediate data) = 0FH  = 00001111
                          Result = 10111111 = 0BFH
```

CPI Compare Immediate With Accumulator

Format:

Label	Code	Operand
oplab:	CPI	data

```
| 1 1 | 1 1 1 | 1 1 0 | data |
```

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

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

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

Since a subtract operation is performed, the Carry bit will be set if there is no carry 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:

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

The CPI instruction performs the following operation:

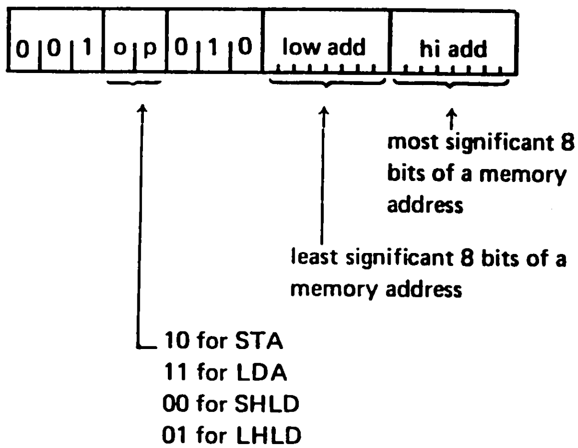
```
         Accumulator = 4AH = 01001010
+(-Immediate data) = -40H = 11000000
                           1] 00001010 = Result
```

carry out = 1 causing the Carry bit 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.

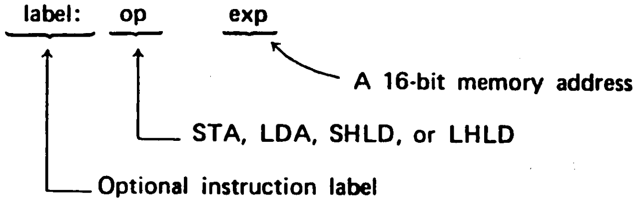
DIRECT ADDRESSING INSTRUCTIONS

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.

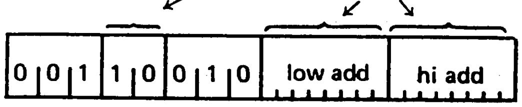
The general assembly language format is:



STA Store Accumulator Direct

Format:

Label	Code	Operand
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:

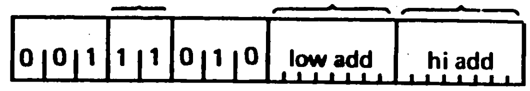
The following instructions will each store the contents of the accumulator at memory address 5B3H:

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

LDA Load Accumulator Direct

Format:

Label	Code	Operand
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:

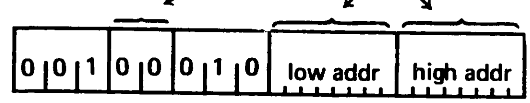
The following instructions will each replace the accumulator contents with the data held at location 300H:

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

SHLD Store H and L Direct

Format:

Label	Code	Operand
oplab:	SHLD	adr



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	109	00
00	10A	29
00	10B	AE
00	10C	00

LHLD Load H And L Direct

Format:

Label	Code	Operand
oplab:	LHLD	adr

```
0 0 1 0 1 0 1 0 | low add | hi add
```

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.

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):

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

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

```
1 1 x x x 0 1 X | low add | hi add
```

- hi add: most significant 8 bits of a memory address
- low add: least significant 8 bits of a memory address
- X: 1 for JMP, 0 otherwise
- xxx:
  - 000 for JMP or JNZ
  - 001 for JZ
  - 010 for JNC
  - 011 for JC
  - 100 for JPO
  - 101 for JPE
  - 110 for JP
  - 111 for JM

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:

Label	Code	Operand
oplab:	PCHL	____

- oplab: Optional instruction label
- Operand: not used

—or—

Label	Code	Operand
label:	op	EXP

- label: Optional instruction label
- op: JMP,JC,JNC,JZ,JNZ,JM,JP,JPE,JPO
- EXP: A 16-bit address

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	ADR	2AC040
		PCHL		E9
		•		•
		•		•
		•		•
4200	LOC:	NOP		00
		•		
		•		
		•		

Program execution begins at STRT. The LHLD instruction loads registers H and L from locations 40C1H and 40C0H; 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.

JMP JUMP

Format:

Label	Code	Operand
oplab:	JMP	adr

1 1	0 0 0	0 1 1	low add	hi add

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	LOAD:	MVI	A, 3	3E03
3D02		JMP	3C03H	C3033C
3E00	CLR:	XRA	A	AF
3E01		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.

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

JC Jump If Carry

Format:

Label	Code	Operand
oplab:	JC	adr

1 1	0 1 1	0 1 0	low add	hi add

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

Condition bits affected: None

For a programming example, see the section on JPO later in this chapter.

JNC Jump If No Carry

Format:

Label	Code	Operand
oplab:	JNC	adr

1 1	0 1 0	0 1 0	low add	hi add

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

Condition bits affected: None

For a programming example see the section on JPO later in this chapter.

JZ Jump If Zero

Format:

Label	Code	Operand
oplab:	JZ	adr

1 1	0 0 1	0 1 0	low add	hi add

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

Condition bits affected: None

JNZ Jump If Not Zero

Format:

Label	Code	Operand
oplab:	JNZ	adr

1 1	0 0 0	0 1 0	low add	hi add

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

Condition bits affected: None

JM Jump If Minus

Format:

Label	Code	Operand
oplab:	JM	adr

1 1	1 1 1	0 1 0	low add	hi add

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

Condition bits affected: None

JP Jump If Positive

Format:

Label	Code	Operand
oplab:	JP	adr

1 1	1 1 0	0 1 0	low add	hi add

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

Condition bits affected: None

JPE Jump If Parity Even

Format:

Label	Code	Operand
oplab:	JPE	adr

1 1	1 0 1	0 1 0	low add	hi add

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

JPO Jump If Parity Odd

Format:

Label	Code	Operand
oplab:	JPO	adr

1 1	1 0 0	0 1 0	low add	hi add

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).

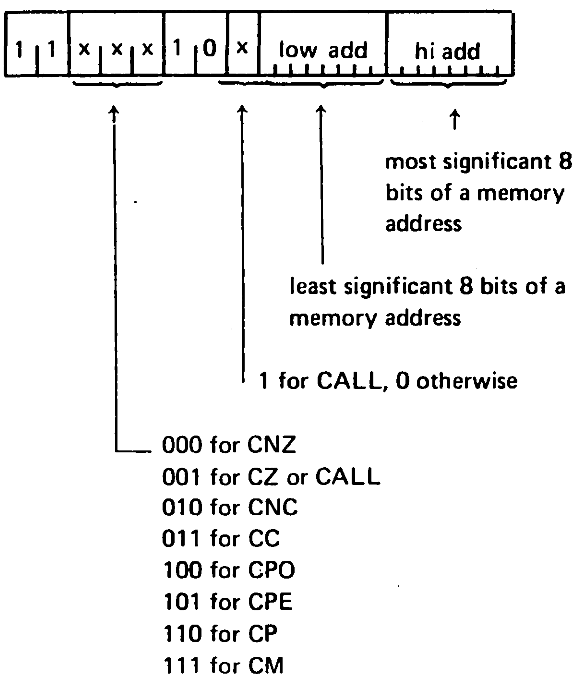
The RLC instruction in block TWO causes the Carry bit to be set equal to the Sign bit of the data byte. If the Sign bit was reset, the JNC instruction causes a jump to PLUS. Otherwise the JMP instruction is executed, unconditionally transferring control to MINUS. (Note that, in this instance, a JC instruction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

CALL SUBROUTINE INSTRUCTIONS

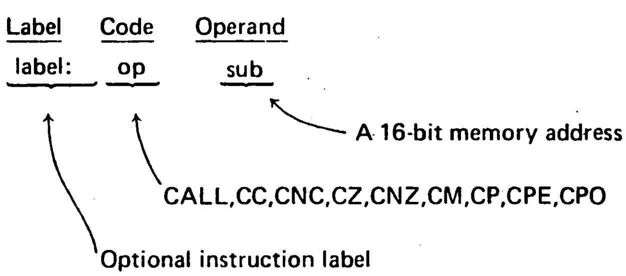
This section describes the instructions which call subroutines. These instructions operate like the jump instructions, causing a transfer of program control. In addition, a return address is pushed onto the stack for use by the RETURN instructions (see Return From Subroutine Instructions later in this chapter).

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:

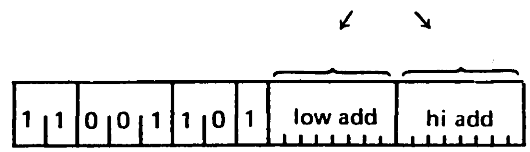


Instructions in this class call subroutines upon certain specified conditions. If the specified condition is true, a return address is pushed onto the stack and program execution continues at memory address SUB, formed by concatenating the 8 bits of HI ADD with the 8 bits of LOW ADD. If the specified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

CALL Call

Format:

Label	Code	Operand
oplab:	CALL	sub



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

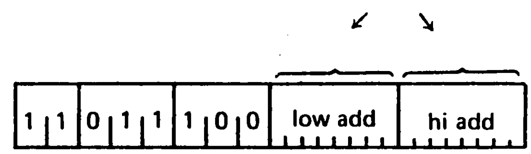
Condition bits affected: None

For programming examples see Chapter 4.

CC Call If Carry

Format:

Label	Code	Operand
oplab:	CC	sub



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

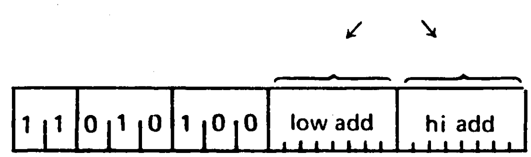
Condition bits affected: None

For programming examples using subroutines, see Chapter 4.

CNC Call If No Carry

Format:

Label	Code	Operand
oplab:	CNC	sub



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 Chapter 4.

CZ Call If Zero

Format:

Label	Code	Operand
oplab:	CZ	sub

1	1	0	0	1	1	0	0	low add	hi add

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 Chapter 4.

CNZ Call If Not Zero

Format:

Label	Code	Operand
oplab:	CNZ	sub

1	1	0	0	0	1	0	0	low add	hi add

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 Chapter 4.

CM Call If Minus

Format:

Label	Code	Operand
oplab:	CM	sub

1	1	1	1	1	1	0	0	low add	hi add

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 Chapter 4.

CP Call If Plus

Format:

Label	Code	Operand
oplab:	CP	sub

1	1	1	1	0	1	0	0	low add	hi add

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 Chapter 4.

CPE Call If Parity Even

Format:

Label	Code	Operand
oplab:	CPE	sub

1	1	1	0	1	1	0	0	low add	low add

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 Chapter 4.

CPO Call If Parity Odd

Format:

Label	Code	Operand
oplab:	CPO	sub

1	1	1	0	0	1	0	0	low add	hi add

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 Chapter 4.

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 1 | x x x | 0 0 | x
```

- 1 for RET, 0 otherwise

- 000 for RNZ
- 001 for RZ or RET
- 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:

Label	Code	Operand
oplab:	op	——

- Label (oplab:): Optional statement label
- Code (op): RET,RC,RNC,RZ,RNZ,RM,RP,RPE,RPO
- Operand: not used

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.

RET Return

Format:

Label	Code	Operand
oplab:	RET	——

```
1 1 | 0 0 1 | 0 0 | 1
```

Description: A return operation is unconditionally performed.

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

Condition bits affected: None

RC Return If Carry

Format:

Label	Code	Operand
oplab:	RC	——

```
1 1 | 0 1 1 | 0 0 0
```

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

Condition bits affected: None

For programming examples, see Chapter 4.

RNC Return If No Carry

Format:

Label	Code	Operand
oplab:	RNC	——

```
1 1 | 0 1 0 | 0 0 0
```

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

Condition bits affected: None

For programming examples, see Chapter 4.

RZ Return If Zero

Format:

Label	Code	Operand
oplab:	RZ	——

```
1 1 | 0 0 1 | 0 0 0
```

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

Condition bits affected: None

For programming examples, see Chapter 4.

RNZ Return If Not Zero

Format:

Label	Code	Operand
oplab:	RNZ	——

```
1 1 | 0 0 0 | 0 0 | 0
```

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

Condition bits affected: None

For programming examples, see Chapter 4.

RM Return If Minus

Format:

Label	Code	Operand
oplab:	RM	—

`1 1 | 1 1 1 | 0 0 0`

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

Condition bits affected: None

For programming examples, see Chapter 4.

RP Return If Plus

Format:

Label	Code	Operand
oplab:	RP	—

`1 1 | 1 1 0 | 0 0 0`

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

Condition bits affected: None

For programming examples, see Chapter 4.

RPE Return If Parity Even

Format:

Label	Code	Operand
oplab:	RPE	—

`1 1 | 1 0 1 | 0 0 0`

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

Condition bits affected: None

For programming examples, see Chapter 4.

RPO Return If Parity Odd

Format:

Label	Code	Operand
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 Chapter 4.

RST INSTRUCTION

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

Format:

Label	Code	Operand
oplab:	RST	exp

`1 1 | exp | 1 1 1`

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 0 E X P 0 0 0 B

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

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

Condition bits affected: None

Example:

Label	Code	Operand	Comment
	RST	10 - 7	; Call the subroutine at
			; address 24 (011000B)
	RST	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 Chapter 5.

INTERRUPT FLIP-FLOP INSTRUCTIONS

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

1	1	1	1	X	0	1	1

X: 1 for EI, 0 for DI

The general assembly language format is:

Label	Code	Operand
label:	op	—

label: Optional instruction label
op: EI or DI
Operand: not used

EI Enable Interrupts

Format:

Label	Code	Operand
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

DI Disable Interrupts

Format:

Label	Code	Operand
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

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:

1	1	0	1	x	0	1	1	device no.

x: 1 for IN, 0 for OUT

device no.: 8-bit device number

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:

Label	Code	Operand
label:	op	exp

label: Optional instruction label
op: IN or OUT
exp: An 8-bit device number

IN Input

Format:

Label	Code	Operand
oplab:	IN	exp

1	1	0	1	1	0	1	1	exp

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:

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

OUT Output

Format:

Label	Code	Operand
oplab:	OUT	exp

1	1	0	1	0	0	1	1	exp

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

Condition bits affected: None

Example:

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

HLT HALT INSTRUCTION

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

Format:

Label	Code	Operand
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.

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 earlier in this chapter, but does not cause any object code to be generated. It acts merely to provide the assembler with information to be used subsequently while generating object code.

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

Label	Code	Operand	Comment
name	op	opnd	

opnd ← Operand, may be optional

op ← ORG,EQU,SET,END,

name ← name may be required, option, or illegal

NOTE: Names on pseudo-instructions are not followed by a colon, as are labels. *Names* are required in the label field of 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. If present, names may be any number of characters long.

ORG Origin

Format:

Label	Code	Operand
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 0A000H

Example 1:

Hex Memory Address	Label	Code	Operand	Assembled Data
		ORG	1000H	
1000		MOV	A,C	79
1001		ADI	2	C602
1003		JMP	NEXT	C35010
	HERE:	ORG	1050H	
1050	NEXT:	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 the section on DS earlier in this chapter). The following two sections of code are exactly equivalent:

Memory Address	Label	Code	Operand	Label	Code	Operand	Assbl. 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

EQU Equate

Format:

Label	Code	Operand
name	EQU	exp
↑ *Required* name		↑ An expression

Description: The symbol "name" is assigned the value by 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.

SET

Format:

Label	Code	Operand
name	SET	exp
↑ *Required* name		↑ An expression

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

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

Example :

Label	Code	Operand	Assembled Data
IMMED	SET	5	
	ADI	IMMED	C605
IMMED	SET	10H-6	
	ADI	IMMED	C60A

END End Of Assembly

Format:

Label	Code	Operand
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.

CHAPTER 3 PROGRAMMING TECHNIQUES

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

BRANCH TABLES PSEUDO-SUBROUTINE

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

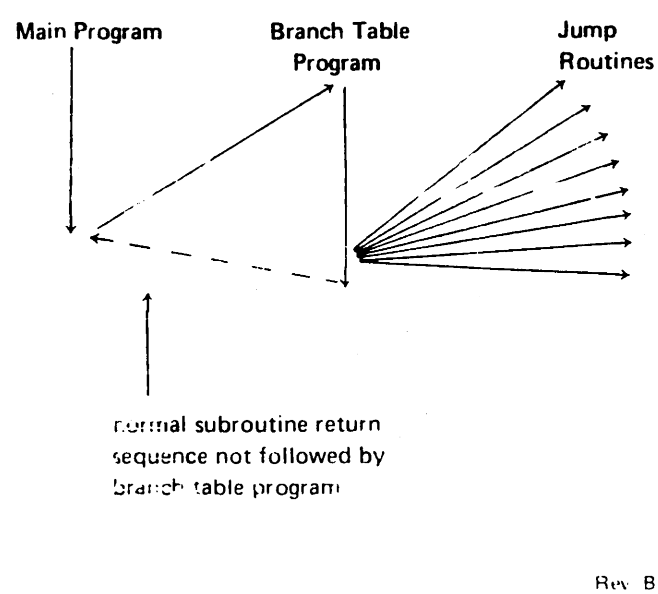
```
CONDITION = CONDITION 1?
IF YES BRANCH TO ROUTINE 1
CONDITION = CONDITION 2?
IF YES BRANCH TO ROUTINE 2
          •
          •
          •
 BRANCH TO ROUTINE N
```

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

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

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

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



```
Label    Code   Operand
START:   LXI    H, BTBL  ; Registers H and L will
                         ; point to branch table.
GTBIT:   RAR
         JC     GETAD
         INX    H        ; (H,L)=(H,L)+2 to
         INX    H        ; point to next address
                         ; in branch table.
         JMP    GTBIT
GETAD:   MOV    E,M      ; A one bit was found.
         INX    H        ; Get address in D and
                         ; E.
         MOV    D,M
         XCHG            ; Exchange D and E
                         ; with H and L.
         PCHL            ; Jump to routine
                         ; address.
         ----
         ----
         ----
BTBL:    DW     ROUT1    ; Branch table. Each
         DW     ROUT2    ; entry 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.

SUBROUTINES

Frequently, a group of instructions must be repeated many times in a program. As we have seen in Chapter 3, 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.

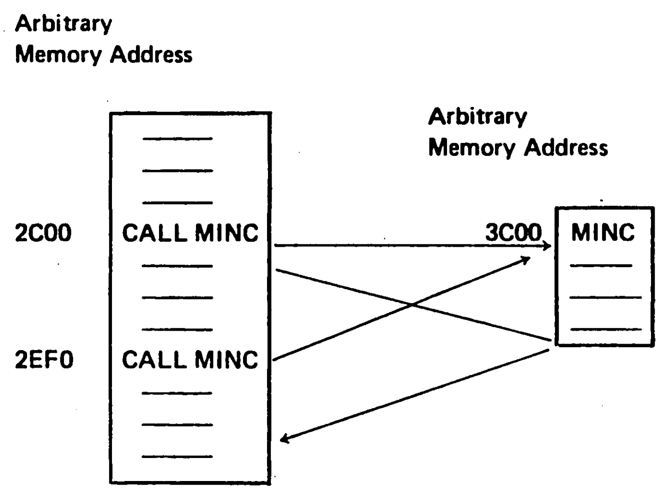
A subroutine is coded like any other group of assembly language statements, and is referred to by its name, which is the label of the first instruction. The programmer references a subroutine by writing its name in the operand field of a CALL instruction. When the CALL is executed, the address of the next sequential instruction after the CALL is pushed onto the stack (see the section on the Stack Pointer in Chapter 1), 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:

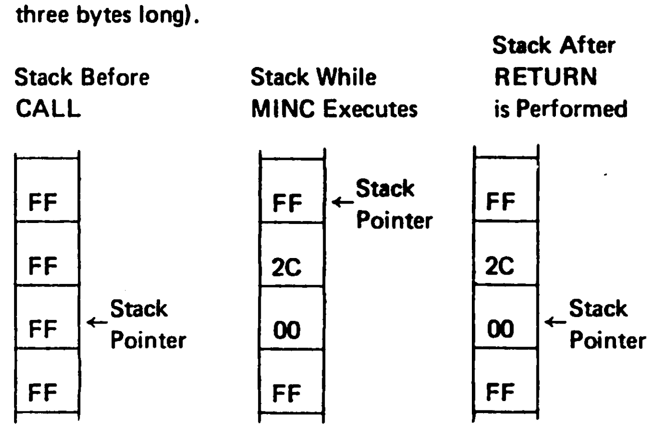
Subroutine MINC increments a 16-bit number held least-significant-byte first in two consecutive memory locations, and then returns to the instruction following the last CALL statement executed. The address of the number to be incremented is passed in the H and L registers.

```
Label    Code   Operand   Comment
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
```

Assume MINC appears in the following program:



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).



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 ssorage 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.

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

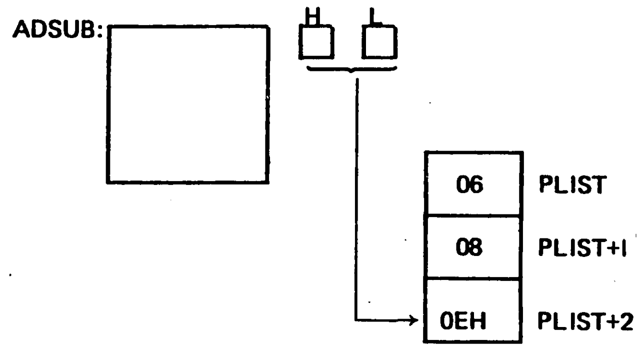
Sometimes it is more convenient and economical to let the subroutine load its own registers. One way to do this is to place a list of the required data (called a parameter list) in some data area of memory, and pass the address of this list to the subroutine in the H and L registers.

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:

```
Label     Code   Operand    Comment
          LXI    H, PLIST   ; Load H and L with
                            ; addresses of the param-
                            ; eter list
          CALL   ADSUB      ; Call the subroutine
RET1:     ----
          ----
PLIST:    DB     6          ; First number to be added
          DB     8          ; Second number to be
                            ; added
          DS     1          ; Result will be stored here
          LXI    H, LIST2   ; Load H and L registers
          CALL   ADSUB      ; for 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
```

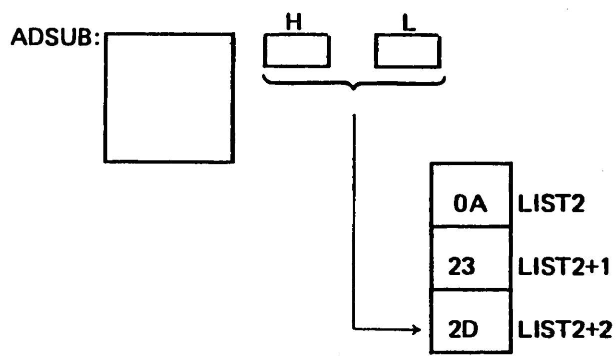
The first time ADSUB is called, it loads the A and B registers from PLIST and PLIST+1 respectively, adds them, and stores the result in PLIST+2. Return is then made to the instruction at RET1.

First call to ADSUB:



The second time ADSUB is called, the H and L registers point to the parameter list LIST2. The A and B registers are loaded with 10 and 35 respectively, and the sum is stored at LIST2 + 2. Return is then made to the instruction at RET2.

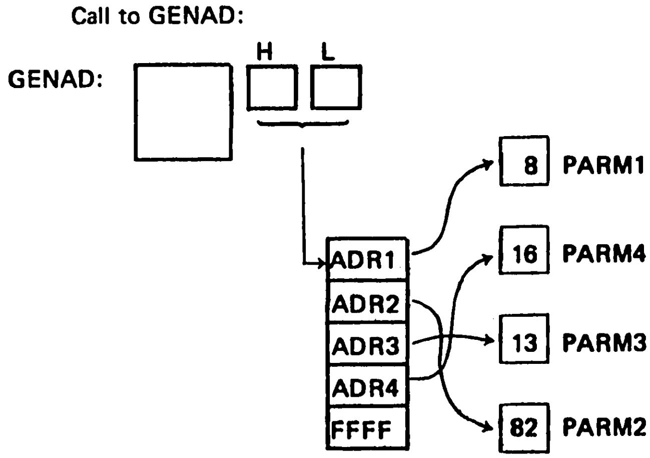
Second call to ADSUB:



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).



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.

```
Label       Code    Operand     Comment

            LXI     H, PLIST    ; Calling program
            CALL    GENAD
            ——
            ——
PLIST:      DW      PARM1       ; List of parameter addresses
            DW      PARM2
            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
                ---
                ---
                ---
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.

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

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

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

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

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

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

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

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

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

The result produced is 09D8.

The process works for the following reason:

The result of any multiplication may be written:

Equation 1: $BIT7 \cdot MCND \cdot 2^7 + BIT6 \cdot MCND \cdot 2^6 + \ldots + BIT0 \cdot MCND \cdot 2^0$

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	=

$0 \cdot 0AH \cdot 2^7 + 0 \cdot 0AH \cdot 2^6 + 0 \cdot 0AH \cdot 2^5 + 0 \cdot 0AH \cdot 2^4 +$

$0 \cdot 0AH \cdot 2^3 + 1 \cdot 0AH \cdot 2^2 + 0 \cdot 0AH \cdot 2^1 + 1 \cdot 0AH \cdot 2^0 =$

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

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

Therefore, step one above produces:

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

Step two produces:

$((BIT0 \cdot MCND \cdot 2^8) \cdot 2^{-1} + (BIT1 \cdot MCND \cdot 2^8)) \cdot 2^{-1}$

$= BIT0 \cdot MCND \cdot 2^6 + BIT1 \cdot MCND \cdot 2^7$

And so on, until step eight produces:

$BIT0 \cdot MCND \cdot 2^0 + BIT1 \cdot MCND \cdot 2^1 + \ldots + BIT7 \cdot MCND \cdot 2^7$

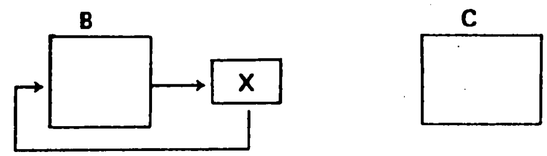
which is equivalent to Equation 1 above, and therefore is the correct result.

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

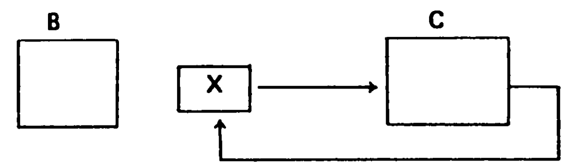
The program uses the B register to hold the most significant byte of the result, and the C register to hold the least significant byte of the result.

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

Zero carry and then rotate B



Then rotate C to complete the shift



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

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

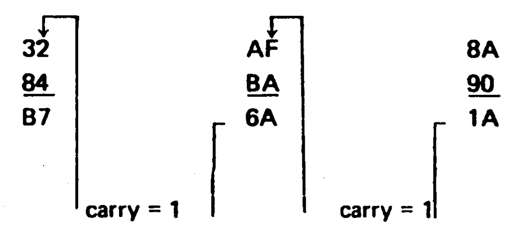
```
DIV:    MVI  E,9        ; Bit counter
        MOV  A,B
DIV0:   MOV  B,A
        MOV  A,C        ; Rotate carry into C
                        ; register; rotate next
                        ; most significant bit
                        ; to carry
        MOV  C,A
        DCR  E
        JZ   DIV2
        MOV  A,B        ; Rotate most significant
        RAL             ; bit to high-order
        JNC  DIV1       ; quotient
        SUB  D          ; Subtract divisor & loop
        JMP  DIV0
DIV1:   SUB  D          ; Subtract divisor. If
                        ; less than high-order
        JNC  DIV0       ; quotient, loop.
        ADD  D          ; Otherwise, add it back
        JMP  DIV0
DIV2:   RAL
        MOV  E,A
        MVI  A,0FFH     ; Complement the quotient
        XRA  C
        MOV  C,A
        MOV  A,E
        RAR
DONE:
```

MULTIBYTE ADDITION AND SUBTRACTION

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

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

This addition may be performed on the 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:

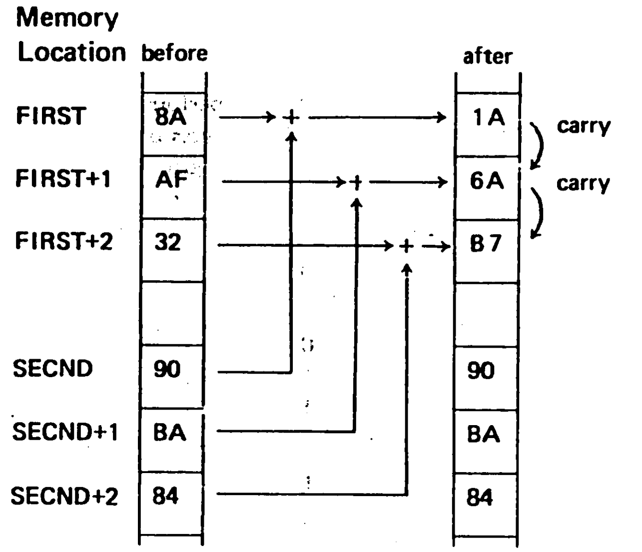


The following routine will perform this multibyte addition, making these assumptions:

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

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

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




```
Label   Code  Operand   Comment
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:

```
  1301
- 0503
  0DFE
```

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):

```
00000001 = 01H
11111101 = -(03H+carry)
11111110 = 0FEH, the low-order result
carry out = 0, setting the Carry bit = 1, indicating a borrow
```

High-order subtraction:

```
00010011 = 13H
11111010 = -(05H+carry)
00001101
carry out = 1, resetting the Carry bit indicating no borrow
```

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

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

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:

```
  2985
+ 4936
  7921
```

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).

```
          85 = 10000101B
          36 = 00110110B
     carry   =        0
             0 10111011B
            /     \
Carry = 0          Auxiliary Carry = 0
```

The accumulator now contains BBH.

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

```
Accumulator = 10111011B
          6 =     0110B
              11000001B
```

Since the leftmost 4 bits are now 910, 6 will be added to these bits, setting the Carry bit.

```
Accumulator = 11000001B
          6 = 0110    B
            1 00100001B
             \
              Carry bit = 1
```

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

(3) Add the next group of two digits:

```
          29 = 00101001B
          49 = 01001001B
     carry   =        1
             0 01110011B
            /     \
Carry = 0          Auxiliary Carry = 1
```

The accumulator now contains 73H.

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

```
Accumulator = 01110011B
          6 =     0110B
            0 01111001B
             \
              carry bit = 0
```

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 the last section, 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.

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:

```
  4358D
- 1362D
  2996D
```

(1) Set carry = 1.

(2) Load accumulator with 99H.

(3) Add zero with carry to the accumulator, producing 9AH.

```
Accumulator = 10011001B
          0 = 00000000B
      Carry =         1
              10011010B = 9AH
```

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

Accumulator = 10011010B
$\overline{62}$H = 10011110B
[1] 00111000B

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

```
Accumulator = 00111000B
        58H = 01011000B
            [0] 10010000B = 90H
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.

Accumulator = 10011001B
$\overline{13}$H = 11101101B
[1] 10000110B

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

```
Accumulator = 10000110B
        43H = 01000011B
            [0] 11001001B = C9H
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.

```
Label   Code    Operand    Comment

DSUB:   LXI     D, MINU    ; D and E address minuend
        LXI     H,SBTRA    ; H and L address subtra-
                           ; hend
        MVI     C, 8       ; Each loop subtracts 2
                           ; digits (one byte),
                           ; therefore program will
                           ; subtract 16 digits.
        STC                ; Set Carry indicating
                           ; no borrow
LOOP:   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
        INX     D          ; Address next byte
                           ; of minuend
        INX     H          ; Address next byte
                           ; of subtrahend
        JMP     LOOP       ; Get next 2 decimal digits
DONE:   NOP
```

CHAPTER 4 INTERRUPTS

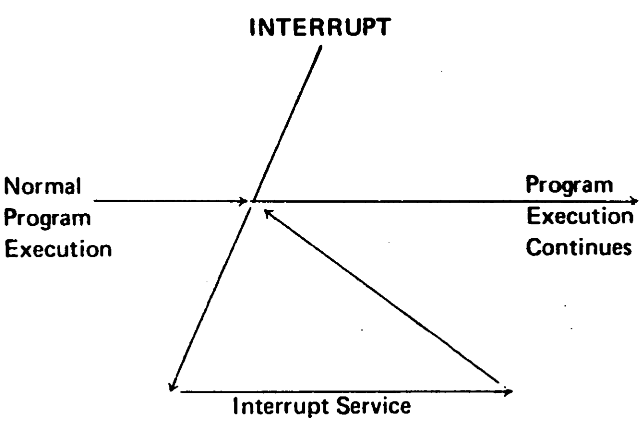
Often, events occur external to the central processing unit which require immediate action by the CPU. For example, suppose a device is receiving a string of 80 characters from the CPU, one at a time, at fixed intervals. There are two ways to handle such a situation:

(a) A program could be written which inputs the first character, stalls until the next character is ready (e.g., 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 Chapter 2. Whenever INTE is equal to 0, the entire interrupt handling system is disabled, and no interrupts 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 Chapter 2), 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:

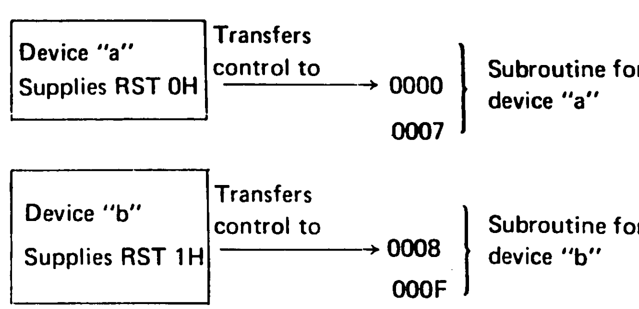
RST 0H

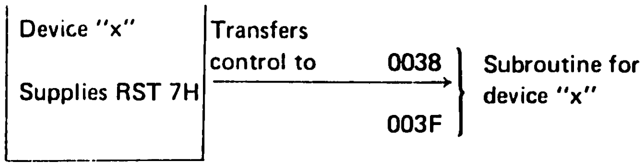
with each teletype input interrupt. Then the subroutine which processes data transmitted from the teletype to the CPU will be called into execution via an eight-byte instruction sequence at memory locations 0000H to 0007H.

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.

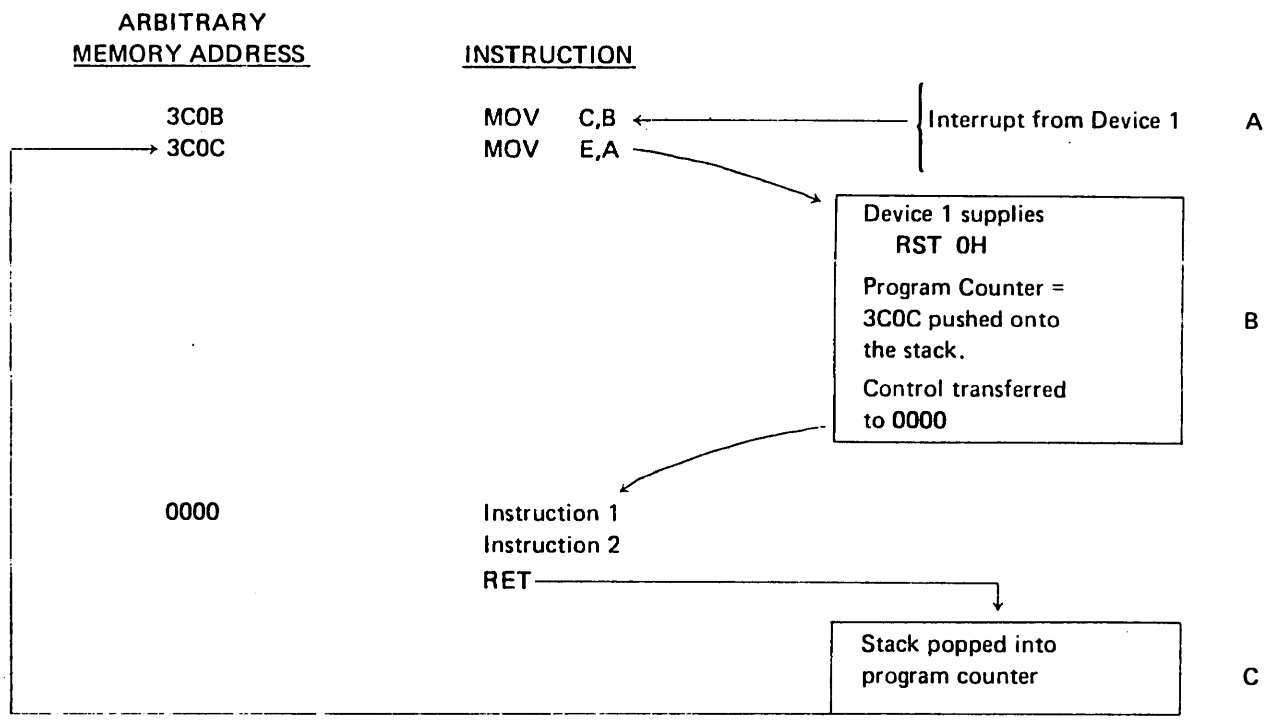
Any device may supply an RST instruction (and indeed may supply any 8080 instruction).

The following is an example of an Interrupt sequence:

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.



Device 1 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 1 is executed. Since this is a call to location zero, 3C0C is pushed onto the stand 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.

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.

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:

```
Code  Operand                Comment
PUSH   PSW   ; Save condition bits and accumulator
EI           ; 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 righthand side of arrow

CARRY BIT INSTRUCTIONS

Format:

```
[LABEL:]                    CODE
```

CODE	DESCRIPTION	
STC	(Carry) ←1	Set carry
CMC	(Carry) ←$(\overline{\text{Carry}})$	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) ← (REGM)+1	Increment register REGM
DCR	(REGM) ← (REGM)-1	Decrement register REGM
CMA	(A) ← $(\bar{A})$	Complement accumulator
DAA	If $(A_0\text{-}A_3) > 9$ or (Aux.Carry)=1, (A) ← (A)+6 Then if $(A_4\text{-}A_7) > 9$ or (Carry)= 1 $(A) = (A) + 6 \cdot 2^4$	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

DATA TRANSFER INSTRUCTIONS

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-ORaccumulator with REGM
ORA	(A) ← (A) OR (REGM)	OR accumulator with REGM
CMP	Condition bits set by (A)-(REGM)	Compare REGM with accumulator

Condition bits affected:

ADD, ADC, SUB, SBB: Carry, Sign, Zero, Parity, Aux. Carry
ANA, XRA, ORA: Sign, Zero, Parity. Carry is zeroed.
CMP: Carry, Sign, Zero, Parity, Aux. Carry. Zero set if (A)=(REGM)
Carry set if (A) $<$ (REGM)
Carry reset if (A) $\geqslant$ (REGM)
Note: CMP treats (A) and (REGM) as unsigned 8-bit quantities.

ROTATE ACCUMULATOR INSTRUCTIONS

Format:

[LABEL:] CODE

CODE	DESCRIPTION	
RLC	(Carry) ← A_7, A_{n+1} ← A_n, A_0 ← A_7	Set Carry = A_7, rotate accumulator left
RRC	(Carry) ← A_0, A_n ← A_{n+1}, A_7 ← A_0	Set Carry = A_0, rotate accumulator right
RAL	A_{n+1} ← A_n, (Carry) ← A_7, A_0 ← (Carry)	Rotate accumulator left through the Carry
RAR	A_n ← A_{n+1}, (Carry) ← A_0, A_7 ← (Carry)	Rotate accumulator right through 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) ← (RP1), ((SP)-2) ← (RP2), (SP) ← (SP)-2	Save RP on the stack RP=PSW saves accumulator and condition bits
POP	(RP1) ← ((SP)+1), (RP2) ← ((SP)), (SP) ← (SP)+2	Restore RP from the stack RP=PSW restores accumulator and condition bits
DAD	(HL) ← (HL) + (RP)	Add RP to the 16-bit number in H and L
INX	(RP) ← (RP)+1	Increment RP by 1
DCX	(RP) ← (RP)-1	Decrement RP by 1
CODE2	**DESCRIPTION**	
XCHG	(H) ⟷ (D), (L) ⟷ (E)	Exchange the 16 bit number in H and L with that in D and E
XTHL	(L) ⟷ ((SP)), (H) ⟷ ((SP)+1)	Exchange the last values saved in the stack with H and L
SPHL	(SP) ← (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

CODE	DESCRIPTION	
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 accumulator with immediate data
ORI	(A) ←(A) OR DATA	OR accumulator with immediate data
CPI	Condition bits set by (A)-DATA	Compare immediate data with accumulator

Condition bits affected:

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 set if (A) < DATA
Carry reset if (A) ≥ DATA
Note: CPI treats (A) and DATA as unsigned 8-bit quantities.

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+!)	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)	Jump to location specified by register H and L
JMP	(PC) ←ADDR	Jump to location ADDR
JC	If (Carry) = 1, (PC) ←ADDR If (Carry) = 0, (PC) ←(PC)+3	Jump to ADDR if Carry set
JNC	If (Carry) = 0, (PC) ←ADDR If (Carry) = 1, (PC) ←(PC)+3	Jump to ADDR if Carry reset
JZ	If (Zero) = 1, (PC) ←ADDR If (Zero) = 0, (PC) ←(PC)+3	Jump to ADDR of Zero set
JNZ	If (Zero) = 0, (PC) ←ADDR If (Zero) = 1, (PC) ←(PC)+3	Jump to ADDR if Zero reset
JP	If (Zero) = 0, (PC) ←ADDR If (Zero) = 1, (PC) ←(PC)+3	Jump to ADDR if plus
JM	If (Sign) = 1, (PC) ←ADDR If (Sign) = 0, (PC) ←(PC)+3	Jump to ADDR if minus
JPE	If (Parity) = 1, (PC) ←ADDR If (Parity) = 0, (PC) ←(PC)+3	Jump to ADDR if parity even
JPO	If (Parity) = 0, (PC) ←ADDR If (Parity) = 1, (PC) ←(PC)+3	Jump to ADDR if parity odd

Condition bits affected: None

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)+1 Return if Carry set
RNC	If (Carry) = 0, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP)+2 If (Carry) = 1, (PC) ← (PC)+1 Return if Carry reset
RZ	If (Zero) = 1, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP)+2 If (Zero) = 0, (PC) ← (PC)+1 Return if Zero set
RNZ	If (Zero) = 0, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP) ← (SP)+2 If (Zero) = 1, (PC) ← (PC)+1 Return if Zero reset
RM	If (Sign) = 1, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP)+2 If (Sign) = 0, (PC) ← (PC)+1 Return if minus
RP	If (Sign) = 0, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP)+2 If (Sign) = 1, (PC) ← (PC)+1 Return if plus
RPE	If (Parity) = 1, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP)+2 If (Parity) = 0, (PC) ← (PC)+1 Return if parity even
RPO	If (Parity) = 0, (PCL) ← ((SP)), (PCH) ← ((SP)+1), (SP) ← (SP)+2 If (Parity) = 1, (PC) ← (PC)+1 Return if parity odd

Condition bits affected: None

RST INSTRUCTION

Format:

[LABEL:] RST EXP

NOTE: 000B ⩽ EXP ⩽ 111B

CODE	DESCRIPTION
RST	((SP)-1) ← (PCH), ((SP)-2) ← (PCL), (SP) ← (SP)+2 (PC) ← 0000000000EXP000B 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 μsec.

Where two numbers of time periods are shown (eq. 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.

MNEMONIC	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	1	1	0	0	11/17
CPO	1	1	1	0	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	0	0	0	0	5/11
RM	1	1	1	1	1	0	0	0	5/11
RPE	1	1	1	0	1	0	0	0	5/11
RPO	1	1	1	0	0	0	0	0	5/11

MNEMONIC	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF TIME PERIODS
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	0	0	0	0	0	0	0	1	10
LXI D	0	0	0	1	0	0	0	1	10
LXI H	0	0	1	0	0	0	0	1	10
LXI SP	0	0	1	1	0	0	0	1	10
PUSH B	1	1	0	0	0	1	0	1	11
PUSH D	1	1	0	1	0	1	0	1	11
PUSH H	1	1	1	0	0	1	0	1	11
PUSH PSW	1	1	1	1	0	0	0	1	11
POP B	1	1	0	0	0	0	0	1	10
POP D	1	1	0	1	0	0	0	1	10
POP H	1	1	1	0	0	0	0	1	10
POP PSW	1	1	1	1	0	0	0	1	10
STA	0	0	1	1	0	0	1	0	13
LDA	0	0	1	1	1	0	1	0	13
XCHG	1	1	1	0	1	0	1	1	4
XTHL	1	1	1	0	0	0	1	1	18
SPHL	1	1	1	1	1	0	0	1	5
PCHL	1	1	1	0	1	0	0	1	5
DAD B	0	0	0	0	1	0	0	1	10
DAD D	0	0	0	1	1	0	0	1	10
DAD H	0	0	1	0	1	0	0	1	10
DAD SP	0	0	1	1	1	0	0	1	10
STAX B	0	0	0	0	0	0	1	0	7
STAX D	0	0	0	1	0	0	1	0	7
LDAX B	0	0	0	0	1	0	1	0	7
LDAX D	0	0	0	1	1	0	1	0	7
INX B	0	0	0	0	0	0	1	1	5
INX D	0	0	0	1	0	0	1	1	5
INX H	0	0	1	0	0	0	1	1	5
INX SP	0	0	1	1	0	0	1	1	5
MOV r_1, r_2	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	D	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	10
INR	0	0	D	D	D	1	0	0	5
DCR	0	0	D	D	D	1	0	1	5
INR A	0	0	1	1	1	1	0	0	5
DCR A	0	0	1	1	1	1	0	1	5
INR M	0	0	1	1	0	1	0	0	10
DCR M	0	0	1	1	0	1	0	1	10
ADD r	1	0	0	0	0	S	S	S	4
ADC r	1	0	0	0	1	S	S	S	4
SUB r	1	0	0	1	0	S	S	S	4
SBB r	1	0	0	1	1	S	S	S	4
AND r	1	0	1	0	0	S	S	S	4
XRA r	1	0	1	0	1	S	S	S	4
ORA r	1	0	1	1	0	S	S	S	4
CMP r	1	0	1	1	1	S	S	S	4
ADD M	1	0	0	0	0	1	1	0	7
ADC M	1	0	0	0	1	1	1	0	7

MNEMONIC	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF TIME PERIODS
SUB M	1	0	0	1	0	1	1	0	7
SBB M	1	0	0	1	1	1	1	0	7
AND M	1	0	1	0	0	1	1	0	7
XRA M	1	0	1	0	1	1	1	0	7
ORA M	1	0	1	1	0	1	1	0	7
CMP M	1	0	1	1	1	1	1	0	7
ADI	1	1	0	0	0	1	1	0	7
ACI	1	1	0	0	1	1	1	0	7
SUI	1	1	0	1	0	1	1	0	7
SBI	1	1	0	1	1	1	1	0	7
ANI	1	1	1	0	0	1	1	0	7
XRI	1	1	1	0	1	1	1	0	7
ORI	1	1	1	1	0	1	1	0	7
CPI	1	1	1	1	1	1	1	0	7
RLC	0	0	0	0	0	1	1	1	4
RRC	0	0	0	0	1	1	1	1	4
RAL	0	0	0	1	0	1	1	1	4
RAR	0	0	0	1	1	1	1	1	4
JMP	1	1	0	0	0	0	1	1	10
JC	1	1	0	1	1	0	1	0	10
JNC	1	1	0	1	0	0	1	0	10
JZ	1	1	0	0	1	0	1	0	10
JNZ	1	1	0	0	0	0	1	0	10
JP	1	1	1	1	0	0	1	0	10
JM	1	1	1	1	1	0	1	0	10
JPE	1	1	1	0	1	0	1	0	10
JPO	1	1	1	0	0	0	1	0	10
DCX B	0	0	0	0	1	0	1	1	5
DCX 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	16
LHLD	0	0	1	0	1	0	1	0	16
EI	1	1	1	1	1	0	1	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

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

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
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