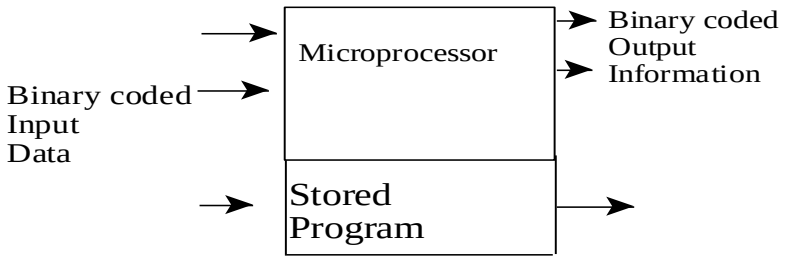
Table of Contents

ASSEMBLY LANGUAGE PROGRAMMING

1.0. INTRODUCTION- TERMINOLOGIES

Program is a set of instructions written in a specific sequence for a computer to accomplish a given task. The process of developing a program is therefore referred to as ***programming.***

A Microcomputer may be described as a digital device that reads binary coded data from its inputs, manipulates this data according to a program stored within its memory, and subsequently produces information at its output.



1.1. Programming Languages.

There are basically TWO types of computer programming languages:

1.1.1. Low Level Languages

This is a medium of communication with a computer that is machine-dependent (or specific to a given computer). The machine and Assembly languages of a computer are considered as low level languages.

(i) ***Machine Language*** is the binary medium of communication with a microprocessor (computer) through a designed set of instruction (instruction set) specific to each microprocessor (CPU). Machine language is written using strings of 0's and 1's.

For example, for 8085 CPU Microprocessor,

Instruction		**Comments**
(i)	01000101	Is the instruction for move. It instructs the 8085 CPU to Move the data in the Register L to the Register B
(ii)	0000011000011000	Instructs the CPU to load 18H into Register B This replaces the original contents of the register.

(iii) 001100101000010000001010 Instructs the computer the CPU to store the contents of the accumulator to the memory location whose address is specified by byte 2 and byte 3.

Programming directly in machine Languages is extremely tedious, time consuming and the process is error-prone since every bit must be perfect.

Instruction types

On traditional architectures, an instruction includes an opcode that specifies the operation to be performed, such as add contents of memory to register—and zero or more operands pecifiers, which may specify registers, memory locations, or literal data.

A Machine instruction may be 1, 2, or 3 bytes in length for 8085 CPU or even 4 bytes for some other microprocessors like Z-80.

In any of these types of instructions, the first byte indicates the operation to be performed (Always referred to as the **operation code or opcode**).The second and third byte, if present, contains either the operand or address of the operand on which the operation is to be performed.

The opcode is referred to as the *instruction field* while the other bytes are referred to as *the Data/Address field*.

One Byte Instruction

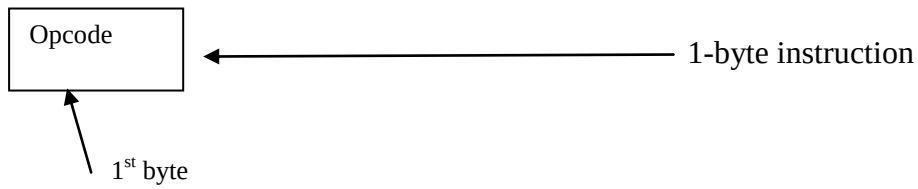
This type of instruction has only op code part of one byte and no operand is given.

The instruction length is only of one byte. It can be stored only in one memory location.

For example MOV C,B

ANA C

RAR etc.



If ‘MOV C,B’ instruction is to be stored in some location say 8400H, then its op code of one byte is to be fed in this memory location.

i.e. 8400H 48H

where 48H is the op code of the instruction ‘MOV C,B’.

Two Byte Instruction

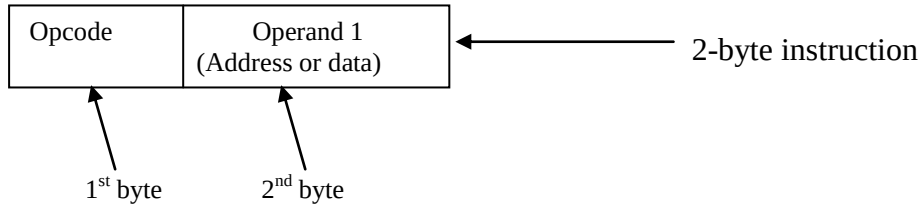
In a two byte instruction, first byte of the instruction is its op code and second byte is the given data.

Such instruction is stored in two consecutive memory locations.

For example MVI A, 06H

OUT 03H

ANI 76H etc.



In order to store the instruction say 'MVI A, 06 H' in the memory locations of the computer, two consecutive memory locations should be used. In one memory location the op code of MVI A is to be stored and in the second location the data 06H is to be stored.

This type of instruction to be stored in two locations say in 2101H and 2102H is given below:

2101H 3EH (op code of MVI A)

2102H 06H (given data)

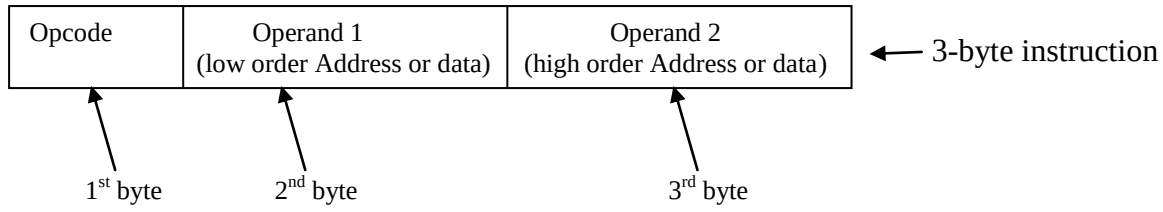
Three Byte Instruction

In a three byte instruction, first byte is used for its op code and second and third bytes are used for 16 bit address. Such an instruction is stored in three consecutive memory locations.

For example LDA 2100H

STA 3000H

JMP 2500H etc.



In order to store the instruction LDA 2100H three consecutive memory locations are to be used. In the first memory location op code of the instruction is stored, in second location lower byte of the address is to be stored and in the third byte upper byte of the address is to

be stored. This instruction loaded in three consecutive memory location 2000H, 2001H and 2002H is given below:

2000H 3AH (op code of LDA)

2001H 00H (Lower byte of address 2100 H)

2002H 21H (Upper byte of address 2100 H)

(ii) ***Assembly Language:*** is a medium of communication with a computer in which programs are written in mnemonics. An assembly language is specific to a given CPU-i.e. they are not transferable from one CPU to another except through a specifically designed translator.

Mnemonics

In general, a mnemonic is a memory aid/tool, such as an abbreviation, rhyme or mental image that helps to remember something *or Mnemonics are memory devices that help learners recall larger pieces of information, especially in the form of lists like characteristics, steps, stages, parts, phases, etc.*The technique of developing these remembering devices is called "mnemonics." Mnemonics can be used to remember phone numbers, names of workmates etc.

In computer assembly language, a mnemonic is a symbolic name for a single executable machine language instruction (an opcode) and there is at least one opcode mnemonic defined for each machine language instruction. It's entered in the operation code field of each assembly language program instruction.

In other words, it is a symbol representing an opcode or symbolic opcode. It is a combination of letters which suggests the operation of an instruction or It is a memory aid with a “sound” suggesting its meaning.

E.g. LD might stand for LoaD
LDA might stand for LoaD Accumulator.
STA might stand for Store Accumulator.
etc.

Thus the mnemonics for a particular instruction consists of letters which suggest the operation to be performed by the instruction.

INR A (INR = INcRement, A=Accumulator)

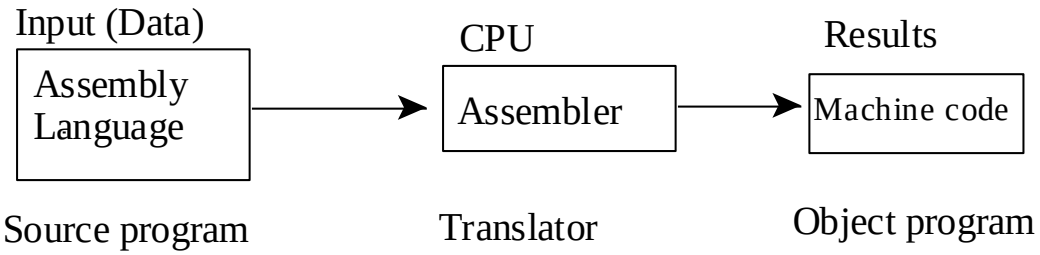
Programmers can write programs called *assembly language program* using these words. Before a program written in assembly language can be executed by a microprocessor, it needs to be translated into an equivalent machine language program. This translation could be done

manually (hand assembly) or on a computer using software called **An Assembler**.

An Assembler is a computer program that translates an assembly language program from mnemonics to the binary machine code of the microprocessor (computer).

Each microprocessor has its own assembler because the mnemonics and machine codes are specific to the microprocessor being used. It accepts an assembly language program as data, converts the mnemonics codes into their numeric equivalent, assign symbolic address to memory location and produces, as output, the required machine code program.

The assembly language program is termed as the ***Source Program*** and the Final Machine code program is called the ***Object Program***.



The assembler has three main functions:

1. Replacement of symbolic opcode (mnemonics) with binary code
2. Assign storage to instructions and output
3. Set the operand addresses.

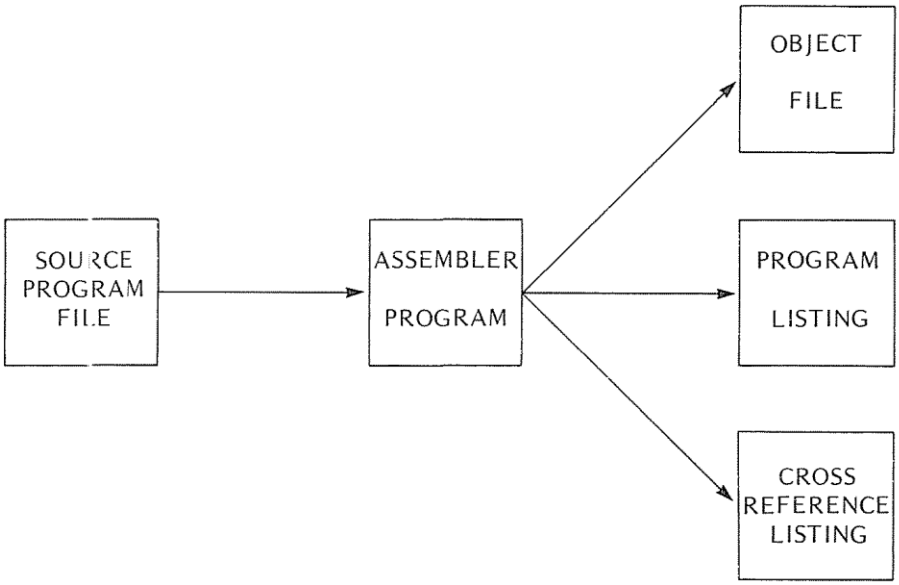
The assembler program performs the clerical task of translating symbolic code into *object code* which can be executed by the 8085 microprocessors. Assembler output consists of three possible files:

i. the *object file* containing source program translated into object code;

ii. the *list file* printout of the source code, the assembler generated object code, and the symbol table; the list file provides a permanent record of both the source program and the object code, The assembler's list file also provides diagnostic messages for common programming errors in the program listing.

iii. and the *symbol-crass-reference file,* a listing of the symbol-cross-reference records. The symbol-crass-reference list file is another of the diagnostic tools provided by the assembler. Assume, for example, that the developed program manipulates a data field

named DATE, and that testing reveals a program logic error in the handling of this data. The symbol-crass-reference listing simplifies debugging this error because it Points to each instruction that references the symbol DATE.



Assembler Outputs

Some assemblers generates only the object file and the list file

This is a source file (source program)

```
PORT1   EQU     21H     ;Input port address
PORT2   EQU     22H     :Output port address
PORTR   EQU     20H     ;Control register
        ORG 8400H       ;Starting address for program
                        ; assembling
START:  MVI A,05H       ;Control word
        OUT PORTR       ;Command register
REPEAT: MVI A,00H
        OUT PORT2       ;Switch off all LEDs
        MVI A,0FFH
        OUT PORT2       ;Switch on all LEDs
        IN PORT1        ;Read state if input switches
        OUT PORT2       ;display on LEDs
        JMP REPEAT      ;go back and repeat
        END 01H         ;end of assembling process
```

This is a object file (object program)

```
:108400003E05D3203E00D3223EFFD322DB21D322E0
:03841000C304841E
:00000101FE
```

This is a print file (or list file)

```
LINE ADDR CODE
0001 0021          PORT1   EQU     21H ;Input port address
0002 0022          PORT2   EQU     22H ;Output port address
0003 0020          PORTR   EQU     20H ;Control register
0004 8400                  ORG 8400H   ;Starting address for

0005                                   ; program assembling
0006 8400 3E05     START:  MVI A,05H   ;Control word
0007 8402 D320             OUT PORTR   ;Command register
0008 8404 3E00     REPEAT: MVI A,00H
0009 8406 D322             OUT PORT2   ;Switch off all LEDs
0010 8408 3EFF             MVI A,0FFH
0011 840A D322             OUT PORT2   ;Switch on all LEDs
0012 840C DB21             IN PORT1    ;Read state if input switches
0013 840E D322             OUT PORT2   ;display on LEDs
0014 8410 C30484           JMP REPEAT  ;go back and repeat
0015 8413                  END 01H
```

1.1.2 High level languages:

Is a medium of communication that is machine independent (is independent of a given computer).Programs are written in English-like words, and they can be executed on a computer after translation .Commonly used translators are **compilers and interpreters.** High level languages are problem oriented languages unlike low level languages which are machine oriented.

1.2 Classification of instructions sets

1.2.1 Instructions set

These are also called a command set, the basic set of commands, or instructions, that a particular microprocessor understands.

1.2.2 Instruction set architecture (ISA)

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor

Two important factors in instruction set design are decoder simplicity and code density. The decoder reads the next instruction from memory, and then routes the component pieces of that instruction appropriately. The design of the instruction set can dramatically affect the complexity and, therefore, the speed of decoding the instructions.

Code density is the combined size of the instructions needed to perform a particular task. Higher density improves memory utilization.

There are generally two classification of the instruction set. One of the principal characteristics that separate the two ISAs is the size and complexity of the instruction set.

(a) *A complex instruction set computer (CISC)* - has many specialized instructions, some of which may only be rarely used in practical programs. A computer microprocessor that is based on CISC architecture has many instructions built into the chip. This makes the processing time for work more efficient since the required instructions are available directly to the microprocessor and do not have to be loaded from the computer's memory or RAM. Although the CISC architecture helps speed up the programs' execution, the number of instructions loaded on the processor negatively impacts processor performance. As a result, more transistors are built into the microprocessor to improve performance, which can result in an increase in the unit's cost and power consumption. CISC processors are most commonly used in personal computers (PCs) using Intel and AMD processors.

(b) *A reduced instruction set computer (RISC)* –is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions.

It simplifies the processor design by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as

subroutines, having their resulting additional processor execution time offset by infrequent use.

The RISC architecture has limited instruction sets built into the processor, resulting in fewer transistors having to be used in processor construction. This saves space on the microprocessor and results in a lower overall cost for the unit. To do work that is equivalent to what a CISC processor can do, a greater amount of computer RAM or memory is required. RISC processors were initially favored for scientific purposes since these applications did not require as large of an instruction set as general purpose computers. They are also used in Apple computers and mobile devices.

In recent years, however, an increasing number of general purpose PCs using RISC microprocessors have been produced.

1.2.2 The 8085 Instruction set

Each microprocessor is designed to execute a particular instruction set. The 8085 has an instruction set containing 74 basic instructions. Many, however, have variations increasing the actual number of distinct operations to 246.

Instructions for the 8085 are 1 to 3 bytes in length. The bit pattern of the first byte is the opcode. This bit pattern is decoded in the instruction register and provides information used by the timing and control sections to generate a sequence of elementary operations-*microinstructions*- that implement the instruction. The second and the third bytes, the instruction operands, are either addresses or constants.

1.2.3. Classification of 8085 instructions

For convenience to programmers, the 8085 instructions have been classified into the following five groups:

(a) ***Data Transfer Group***.

The instruction in this group move data between the various microprocessor registers or between a microprocessor register and a memory location.

```
e.g.    MOV   H,E    ; Causes the contents of register E to be transferred to register H.
        STA 8700H    ; causes the contents of Accumulator to be stored at memory location
                       8700H.
```

(b) ***Arithmetic Group***.

Instruction in the Arithmetic group performs arithmetic operations on data that are either in a

specific processor register or in a memory location.

e.g. ADD L; causes the contents of the accumulator to be added to the content of register L. The result (sum) is stored in the accumulator.

c) ***Logical Group***.

Instruction in the Logical group performs logical operations on data that are either in a specific processor register or in a memory location.

e.g. ORA B ; cause a bit-by-bit OR operation to be performed on the contents of the B register and the accumulator .The result is placed in the accumulator.

Arithmetic and Logical group of instructions are normally called **data manipulation** instructions.

(d). ***Branch (Transfer of control) Group***.

The transfer of control group of instructions provides the computer with the ability to transfer from one sequence of operation to another; based on a variety of conditions.

When such an instruction is encountered, the PC is changed to the address of the location of the new sequence. The next machine cycle begins with the new sequence of instructions.

Some of these transfers of control instructions always force a jump to the new sequence while others base the jump on the result of a previous operation.

If the jump is always made, it is called **an unconditional** branch .If the jump is based on machine data, such as a specific flag status, it is called a **conditional branch**.

E.g.

JMP 2800H ; is an unconditional branch to the memory location 2800H.

JNZ 4760H ; is a conditional branch to the memory location 4760H only if the previous arithmetic (or logical) operation has **NOT** resulted into a zero, i.e. Zero flag is not set.

(e) . **Stack, Input /Output and Machine control group.**

Instructions in **the input/output group** move data between the various I/O ports of the system and an internal register, usually the accumulator.

e.g. OUT 22H causes the contents of the accumulator to be transferred to the output port with address 22H.

The instructions in **the machine control group** affect the state or mode of operation of the processor itself. A common machine control instruction available in most microprocessors is the NOP (no operation).This causes the machine to wait through an instruction cycle.

Others are HLT (or HALT), RIM etc

Stack instructions are used to perform stack related operation.

E.g. PUSH B causes the content of the B-C register pair to be stored in the stack.

Refer to the attached Instruction set

1.2.4 Hand Assembling (Assembly process using lookup tables)

(a) One byte instructions

i.	Mnemonic	ADD B	Translated Machine code	**80**
ii.	Mnemonic	MOV D, C	Translated Machine code	**51**
iii.	Mnemonic	DCR B	Translated Machine code	**05**

(b) Two bytes instructions

i.	Mnemonic	MVI A, 55H	Translated Machine code	**3E**55
ii.	Mnemonic	ADI 44H	Translated Machine code	**C6**44
iii.	Mnemonic	IN 21H	Translated Machine code	**DB**21
iv.	Mnemonic	OUT 45H	Translated Machine code	**D3**45

(c) Three bytes instructions

i.	Mnemonic	CALL 2244H	Translated Machine code	**CD**4422
ii.	Mnemonic	LXI H, 3388H	Translated Machine code	**21**8833
iii.	Mnemonics	STA 8200H	Translated Machine code	**32**0082

1st Byte (bold) in each translated code is the opcode

1.3 Assembly Language Instruction Format

Each instruction in an assembly program has four fields: a label field, a mnemonic field (or the opcode field), an operand field and a comment field. These fields follow each other in the sequence specified. All fields are optional. Furthermore, the comment field is not translated by the assembler during translation of the assembly program into an equivalent machine language program. When it is desired to write only a comment, all other fields may be omitted.

Label	***Mnemonic***	***Operand(s)***	***Comments***
START:	MOV	A,B	; Moves contents of Reg. B to Reg. A

```
POO4  :  LXI        H,5699H     ; Get address 5699H into register-pair HL
etc.
```

Label

Labels are programming aids used with jumps and calls. When writing a program, we often have no idea what address to use in a Jump or Call instruction. By using labels instead of numerical addresses, program writing becomes easy and more flexible. The assembler keeps track of the labels and automatically assigns the correct address to them.

Since labels are symbolic addresses, one can use them in the operand fields.

e.g. JMP LOOP

where LOOP represents the target address. The label field contains a label. A label could be any character string consisting of lower and upper case letters (A-Z or a-z), digits (0-9) and the $ sign. The first character of the label must be a letter or a $ sign. Some assemblers place an upper limit on the number of characters in a label.

A label is separated from the mnemonic by a colon. Similarly, any operand is separated from the mnemonic by at least one space character. The comment is separated from any operand by a semicolon.

Operands.

An explicit operand required by an instruction could be specified in one of several ways:

i) No operand

```
CMC         ; This complements the carry flag.
```

ii) Registers

```
MOV A,B     ; Move contents of Reg. B to Reg. A
```

iii) Label

```
LDA TEMP    ; Get the contents of memory location TEMP and
            ; load them into the accumulator.
```

One may use simple arithmetic expressions in the operand if it is a label.

e.g. STA START+2

Only expressions of the type X ± C are allowed as operands, where X denotes a label and C is a constant.

iv) Immediate Operands (address or data)

```
e.g.    MVI   B, 68H ; Gets the constant 68H into Reg. B
```

```
LDA   4000H; Gets the contents of memory location 4000H into Acc.
```

Constants in Assembly Language Programs

Constants without any suffix will be treated by the assembler as decimal constants. Thus 100, 496 etc are all decimal constants.

Suffixes B, H and O are used to indicate that the constant is binary, hexadecimal and octal respectively.

e.g.

100B - is a binary constant

100H - is a hexadecimal constant

A hexadecimal constant must be preceded by a 0 (zero) if it starts with any of the letters A-F. The assembler will treat A3H as a label (symbol) and not a constant. To force the assembler to treat A3H as a constant, it should be written as 0A3H.

2.0 ADDRESSING MODES

2.1 Definition

An addressing mode is a method of specifying the address of the operand in an instruction.

2.2 Register –Register Addressing

This addressing mode is used for data transfer and manipulation instruction involving the internal registers. The source and the destination register are contained within the instruction itself.

e.g.
```
MOV A,B
ADD C        ---8085 CPU
or
LD H,D
ADD A,D      ----Z80
```

With these instructions, the accumulator is at times implied as a second operand. For example, the instruction CMP E may be interpreted as 'compare the contents of the E register with the contents of the accumulator.

Most of the instructions that use register addressing deal with 8-bit values. However, a few of these instructions deal with 16-bit register pairs. For example, the PCHL instruction exchanges the contents of the program counter with the contents of the H and L registers. Also XCHG instruction which exchanges the contents of H-L register pair with that of D-E register pair.

2.3 Immediate Addressing

In Immediate addressing, the operand to be acted upon is specified within the instruction. It is the byte(s) following the opcode.

e.g.
```
MVI B, 45H
LXI H, 3475H -------8085
or
LD B, 73H
LD HL, 8756H          -----Z80
```

The names of the immediate instructions indicate that they use immediate data. Thus, the

name of an add instruction is ADD; the name of an add immediate instruction is ADI.
All but two of the immediate instructions uses the accumulator as an implied operand, as in the CPI instruction. The MVI (move immediate) instruction can move its immediate data to any of the working registers including the accumulator or to memory. Thus, the instruction MVI D, 0FFH moves the hexadecimal value FF to the D register.

The LXI instruction (load register pair immediate) has its immediate data as a 16-bit value. This instruction is commonly used to load addresses into a register pair.
LXI instruction is normally used to initialize the stack pointer; For example, the instruction LXI SP,30FFH loads the stack pointer with the hexadecimal value 30FF.

2.4 Direct Addressing

In this addressing mode, the address of the operand is specified within (as part of) the instruction. Sometimes it is called *absolute addressing mode*. It is used to access data and write data into system memory.

e.g.

```
LDA 8600H
STA 3450H
OUT 22H
IN 21H          -8085
or
LD HL,(8400H)
LD (7560H),A
OUT (23H),A          -Z80
```

Instructions that include a direct address require three bytes of storage: one for the instruction code, and two for the 16-bit address

2.5 Indirect Addressing Mode.

In this mode, the address of the operand is NOT contained in the instruction. The instruction only contains a pointer that stores the address of the operand.

e.g.

```
MOV M, A
STAX B
or
LD A, (BC)
LD C, (HL)
OUT (C), B
```

Register indirect instructions reference memory via a register pair.

2.6 Implied Addressing

In implied addressing mode, the location of the operand is contained within the opcode. In most cases, it is the accumulator.

e.g. CMA
RAL -8085
or
NEG - Negate Accumulator
CPL -complement accumulator -Z80

Advantages and Disadvantages

1. Implied –are one byte instruction and hence are faster to execute, and occupies less memory.
2. Direct addressing-It is simple to use but since most are 3 byte instructions, it takes longer time to execute and occupies more memory space.
3. Indirect Addressing- allows convenient ways of accessing data stored in sequential memory block. Most are one byte instruction (8085) and hence occupies less memory. But they sometimes take longer to execute.

Some microprocessors use other addressing modes that are not available via single instructions in the 8085. Some of these addressing modes are not supported by 8085 directly because of the nature of 8085's internal architecture. However, these addressing modes can be synthesized y combining two or more8085 instructions.

2.7 Indexed Addressing.

In Index addressing, the instruction specifies both an index register and an address displacement. The content of the register is added to the displacement to produce the final effective address of the operand. In this way, the address could be the beginning of a table in the memory. The index register would then be used to access all the elements of a memory block/table successively in an efficient way, just by incrementing or decrementing the displacement.

e.g. LD A,(IY+d) -Load accumulator with data stored at address pointed by (IY index register + the displacement).

2.8 Relative addressing.

This addressing mode is used to effectively and efficiently implement transfer of control

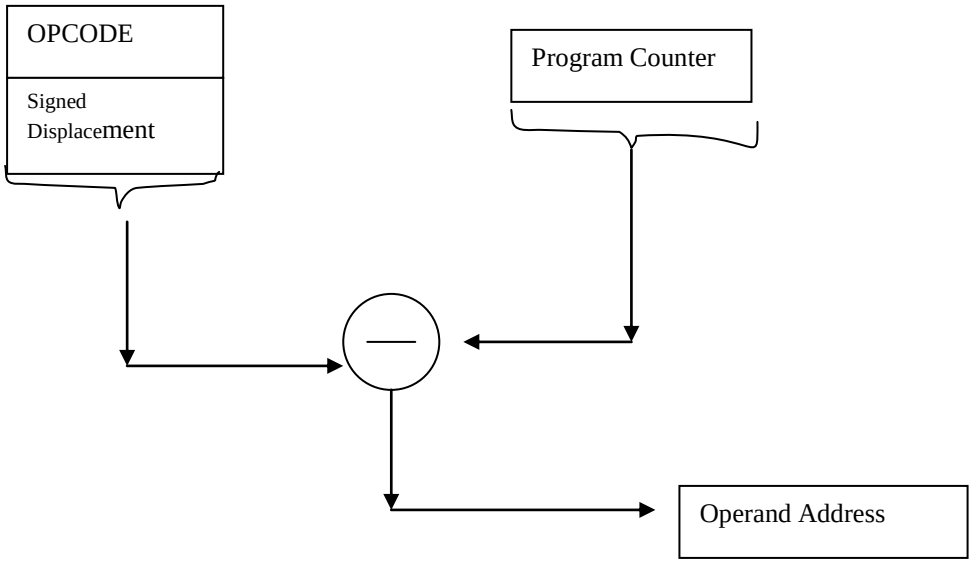
instructions (Jumps). The instruction under this addressing mode uses only two bytes. The first byte is the branch specification,(conditional or unconditional) and the second byte is the displacement. Since the displacement must be positive or negative (2's complement), relative branching can be used to branch forward to 127 locations or to branch backward 128 locations. Because most loops tend to be short, relative branching can be used most of time and results in significantly improved performance.

e.g.

JR NC,04H -Jumps forward 4 locations

JR Z,F2H -Jumps backward 14 locations

[F2H=$11110010_{(2,C)}$= $-14_{(10)}$]



2.9 Page Zero Addressing Mode

This is similar to direct addressing except that it requires two-byte instruction.

The Least significant byte of the address containing the operand is contained in byte 2 of the instruction. Byte 1 is the opcode. It is then assumed that the most significant byte of the address are 00H. Using this mode, it is clear that the operand can only be stored within the address range 0000H-00FFH i.e. the first $256_{(10)}$ memory locations.

2.10 Modified Page Zero Addressing.

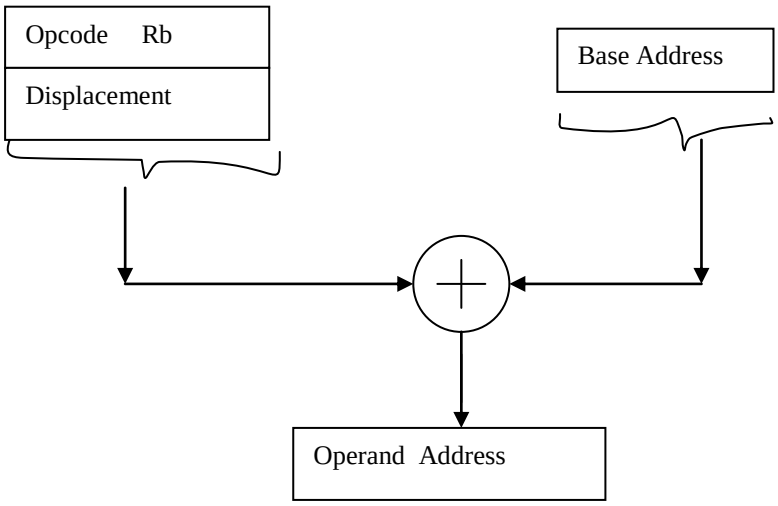
It is provided with one instruction namely RST. The effect of this instruction is to cause a jump to a new address on page zero (0000-00FFH), defined by the value of 'nnn' after

pushing the contents of the PC on to the stack.

Opcode	1 1 n n n 1 1 1

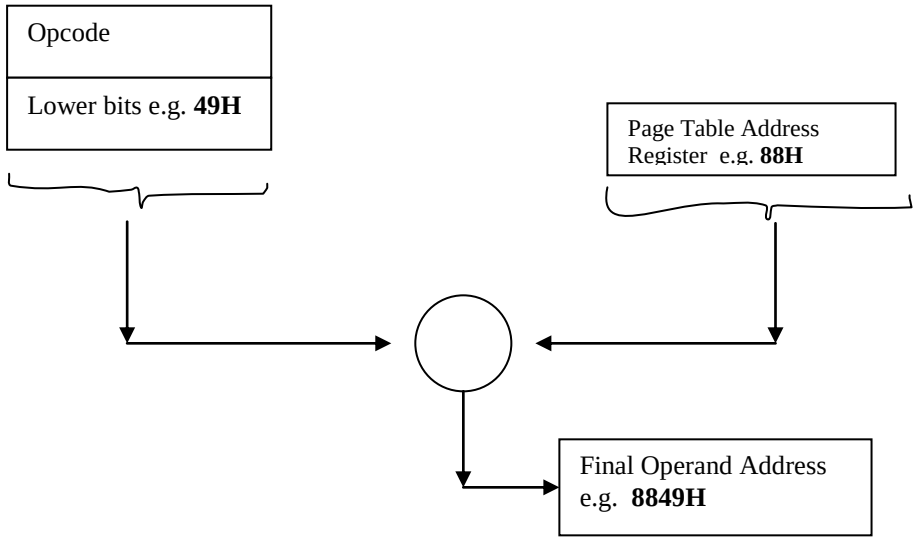
2.11 Base Register Addressing

Base register addressing conceptually, is almost identical to indexing and in several large computing machine, the same hardware is utilized to contain an address (base). The address displacement in the instruction is added to the contents of the base register to obtain the effect address of the operand. The displacement may be a signed binary number so that the operand may be found at an address above or below that contained in the base register.



2.12 Page Addressing

This is a variant of the base register addressing. The main storage is divided into sectors or pages. Each page has a maximum size equal to the addressing range of the displacement. A page /sector address register is preloaded with the high order bits of the effective address. The full effective address is formed by **concanating** the high-order bits in the sector address register with the displacement bits; the displacement forms the least significant bits of the effective address. This technique is often called pagination or sectorization.



2.13 Bit Addressing

This addressing mode is used to access specific bits in a register or memory location. E.g. Z80 CPU is equipped with special instruction for setting, resetting and testing specific bits in a memory location or a register.

e.g.

1. BIT b,(HL) -Test bit b of indirectly address memory location. The specified bit of the memory location addressed by the HL register pair is tested and the Z flag is set according to the result. b may be 0,1,2,3,4,---7.

   BIT b,r -Test bit b of register r

   e.g. BIT 6,A –tests bit 6 of register A contents and zero flag is set according to the result.

   r=A,B,C,D,E,H,L,(HL)

2. RES b,s -Reset bit b of operand s

   $s_b \leftarrow 0$

b: 0,1,2,3……7
s: A, B,C,D,E,H,L,

3. SET b,s -set bit b of operand

   sb $\leftarrow$ 1

s: A,B,C,…..H,L,(HL)

b:0,1,2,…..7

The specified bit of the location determined by s is set.

2.14 Combined Addressing Modes

Some instructions use a combination of addressing modes. A CALL instruction, for example, combines direct addressing and register indirect addressing. The direct address in a CALL instruction specifies the address of the desired subroutine; the register indirect address is the stack pointer. The CALL instruction pushes the current contents of the program counter into the memory location specified by the stack pointer.

Timing Effects of Addressing Modes:

Addressing modes affect both the amount of time required for executing an instruction and the amount of memory required for its storage. For example, instructions that use implied or register addressing, execute very quickly since they deal directly with the processor′s hardware or with data already present in hardware registers. Most important, however is that the entire instruction can be fetched with a single memory access. The number of memory accesses required is the single greatest factor in determining execution timing. More memory accesses therefore require more execution time. A CALL instruction for example, requires five memory accesses: three to access the entire instruction and two more to push the contents of the program counter onto the stack.

The processor can access memory once during each processor cycle. Each cycle comprises a variable number of states. The length of a state depends on the clock frequency specified for the system, and may range from 480 nanoseconds to 2 microseconds. Thus, the timing for a four state instruction may range from 1.920 microseconds through 8 microseconds. (The 8085 have a maximum clock frequency of 5 MHz and therefore a minimum state length of 200 nanoseconds.)

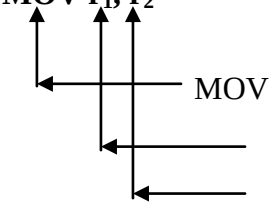
3.0 DATA TRANSFER INSTRUCTIONS.

Data Transfer Instructions move data (generally byte length data), from one location to another e.g. between various processor registers or between a processor register and a memory location. The source of the data is defined in the instruction.

3.1 Moving data between internal processor Registers.

The instruction to move data between processor registers is the mnemonic MOV**. (**MOVe)
This one-byte instruction's operand defines the source and the destination.
e.g.

MOV r_1, r_2



MOV is the operation mnemonics
r_1 is the destination register
r_2 is the source register

MOV C, B -causes the contents of B-register to be transferred to the C-register. The contents of register B remain unchanged.

(C) ← (B) ( ) contents of

For Z80 CPU

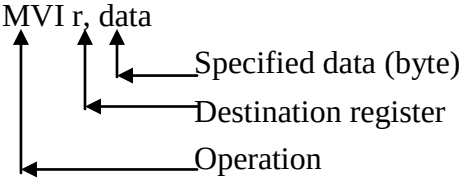
LD r_1, r_2 *Loads the contents of r_2 into r_1*

The 8085 allow limited 16-bit data transfer operation. The instruction dealing with 16-bits is XCHG
This instruction causes the contents of the combined D-E pair and H-L pair to be exchanged
e.g. (H) ⟷ (D) (L) ⟷ (E)
These are one-byte instructions.
For Z80 CPU
EX DE, HL *this instruction causes the contents of the combined D-E pair and H-L pair to be exchanged*
e.g. (H) ⟷ (D) (L) ⟷ (E)

3.2 Moving specified data into an internal register.

I. The mnemonic (operand) that loads a specified 8-bit data into a register is MVI (MoVe Immediate)

e.g. MVI r, data



MVI A, 44H -loads register A with 44H.

(A) ← 44H

MVI B, 6BH -loads register B with 6BH

(B) ← 6BH.

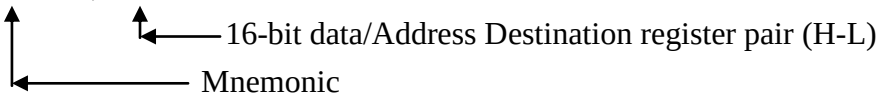
These are two byte instructions

For Z80 CPU

LD r_1, n *Loads n into register r_1*

II The mnemonic that loads a specified 16-bit data /Address into register B-C, D-C, H-L pair and SP is LXI (Load immediate)

e.g. LXI H, 8635H



(H-L) ← 8635H

LXI SP, 8700H

(SP) ← 8700H

Loads SP with 8700H (Initialising the stack)

These are 3 byte instructions.

For Z80 CPU

LD dd, nn *Loads nn into registers pair dd; where dd may be BC, HL, DE, or SP*

3.3 Moving data between Accumulator and specified memory location, and between H-L pair and specified memory locations.

(These are three bytes instructions).

I. **Between accumulator and memory location**

-The Instruction used to load data into the Accumulator from specified memory location is LDA (Load Accumulator)

e.g. LDA 8600H - This causes the A register to be loaded with the contents of the memory location whose address is 8600H.

(A) ⟵ (8600H)

-The instruction used to store the contents of the accumulator into memory location is Store Accumulator, STA.

e.g. STA 8500H causes the contents of the accumulator to be stored in the memory location with address 8500H.

(A) ⟶ (8500H)

For Z80 CPU

LD A, (nn) this causes the A register to be loaded with the contents of the memory location whose address is nn H.

LD (nn), A causes the contents of the accumulator to be stored in the memory location with address nn H

II. Between the H-L pair and specified memory location

-Instruction that loads H-L pair with data from specified memory location is

LHLD. (Load H-L Direct)

e.g. LHLD 6000H

This loads the contents of memory location 6000H into register L while register H is loaded with the contents of the next consecutive location, 6001H.

(L) ⟵ (6000H) (H) ⟵ (6001H)

-Instruction that store the contents of H-L pair into specified memory location is

Store H-L Direct, SHLD

E.g. SHLD 2000H -Stores the contents of H-L pair into memory location with address 2000H and 2001H.

i.e. (L) ⟶ (2000H) (H) ⟶ (2001H).

For Z80 CPU

LD HL, (nn) *this loads the contents of memory location nn into register L while register H is loaded with the contents of the next consecutive location, nn+1.*

LD (nn), HL *this loads the contents of register L into memory location nn while the contents of register H are loaded into the next consecutive location, nn+1.*

3.4 Moving data between any processor register and memory location without specifying the memory address directly.

(The memory address is assumed to be stored in a register pair).

I. When the register pair holding the memory address is H-L, H-L is referenced using the single letter ***M***

(a) *Moving data between register and memory location;*

E.g. MOV B, M

The contents of memory location whose address is in the H-L register pair is moved to register B

i.e. (B) ⟵ ((H-L))

Also

MOV M, B

(B) ⟶ ((H-L))

LD r, (HL) *the contents of memory location whose address is in the H-L register pair is moved to register r where r may be A,B,C,D,E*

(b) *Moving a specified data into memory location whose address is in the H-L pair register*:

MVI M, data

e.g.

MVI M, 86H

((H-L)) ⟵ 86H

LD (HL), n *this loads n into the memory location whose address is in the H-L*

II. Moving data between accumulator and memory location whose address is stored in B-C or DE register pair.

LDAX D -loads the accumulator with data from memory whose address is in the D-E pair registers. (A) ⟵ ((D-E))

LDAX B -loads the accumulator with data from memory whose address is in the B-C register pair. (A) ⟵ ((B-C))

STAX D -stores the contents of accumulator into memory whose address is in D-E register pair. (A) ⟶ ((D-E))

STAX B -stores the contents of the accumulator into memory location whose address is in B-C register pair. (A) ⟶ ((B-C))

For Z80 CPU

LD (BC), A stores the contents of the accumulator into memory location whose address is in B-C register pair.

For Z80 CPU

LD A, (DE) loads the accumulator with data from memory whose address is in the D-E pair registers.

For Z80 CPU

Most data transfer operations, between registers, between register and memories are accomplished by the instruction **LD**.

For example

Between register A, D	LD, D,A
Immediate	LD C,32H
Indirect transfers	LD (HL),56H
Register pair loading	LD BC,45E8H etc

4.0 Programming examples

Q1. Outline, giving examples, the SIX groups of instructions under which the 8085 CPU instruction set may be classified.

Q2. (a) Develop an 8085CPU assembly program to perform the following:

i. Load 48H, F3H, 76H, and 40H into registers A, B, C and H respectively

ii. transfer the contents of register B into register L

iii. store the contents of Register A into memory location with address 3001H

iv. store the contents of register C into memory location pointed by HL register pair.

v. Stop

Solutions

i. MVI A,48H
MVI B,F3H OR MVI B,0F3H
MVI C,76H
MVI H,40H

ii. MOV L,B

iii. STA 3001H

iv. MOV M,C

v. HLT

(b) Draw a trace table for the program listing in Q2 (a) and fill in the contents

Instructions		A	B	C	H	L	3001H	(HL)-40F3H
	Initial state	X	X	X	X	X	X	X
MVI A,48H		48H	X	X	X	X	X	X
MVI B,F3H		48H	F3H	X	X	X	X	X
MVI C,76H		48H	F3H	76H	X	X	X	X
MVI H,40H		48H	F3H	76H	40H	X	X	X
MOV L,B		48H	F3H	76H	40H	F3H	X	X
STA 3001H		48H	F3H	76H	40H	F3H	48H	X
MOV M,C		48H	F3H	76H	40H	F3H	48H	76H
HLT		48H	F3H	76H	40H	F3H	48H	76H

(c) Hand-assemble the program in 2(b) and determine the memory capacity of the program in bits

Solutions

	Mnemonics	**Machine code**
i.	MVI A,48H	3E48
	MVI B,F3H OR MVI B,0F3H	06F3
	MVI C,76H	0E76
	MVI H,40H	2640
ii.	MOV L,B	68
iii.	STA 3001H	320130
iv.	MOV M,C	71
v.	HLT	76

Memory capacity occupied by the program is ***14 bytes***

References

1. Digital Computers Electronics; "An introduction to Microcomputers". 2nd edition by Albert Paul Malvino Ph.D.
2. The MCS-80/85 Family User manual by Intel Corporation
3. "Introduction to Microprocessor 8085", By Dr. D. K. Kaushik, Dhanpat Rai Publishing co., New Delhi
4. "Microprocessor Architecture, Programming and Applications with 8085", 5th Edition, By Ramesh S. Goankar, Prentice Hall
5. Assorted websites