Paper : Microprocessors and Computer Programming

Lesson: 8085 Instructions

Author: Dr. Anant Pandey

College/Department: Sri Venkateswara College, University of Delhi

Table of Contents

Chapter : 8085 Instructions

Chapter: 8085 Instructions

Introduction

A binary pattern of 0s and 1s that is already designed inside a microprocessor for carrying out a specific task is called an *instruction*. And it is the entire group of such binary patterns of the microprocessor, known as its *instruction set*, which determines what different functions it can perform. Moreover, an *instruction* can also be thought of as a command that is given to the microprocessor to perform a specific task on a specified data. It is this command in the binary form given to the microprocessor by the user through a program, which is matched by the microprocessor with its internally designed binary patterns (read *instruction set*) that subsequently the appropriate action specified in the binary pattern is performed by the microprocessor.

Before we study the entire *instruction set* of the 8085 microprocessor in detail it will only be proper on our part to get a fair idea about the various types of languages that programmers can use to communicate with computers. Broadly speaking, computer languages can be classified into two types, namely *low-level languages* and *high-level languages*.

Computer languages that are machine independent such as Basic, Fortran, Pascal, C, etc are called *high-level languages*. A program written in a *high-level language* can run in any computer irrespective of its microprocessor provided the computer has the appropriate compiler or interpreter for generating the object code (that is the binary equivalent of the program that the microprocessor can understand). *Low-level languages* on the other hand are machine specific. Such languages are called low-level since each instruction in such a language performs a much lower level task compared to a *high-level language* instruction. Therefore a *low-level language* program will be much bigger in size compared to a *high-level language* program written for the same task. The two examples of *low-level languages* are the *machine language* and the *assembly language*.

Machine Language

Every computer has its own set of instructions in the binary form called its *instruction set*. It is this set of instructions that forms the *machine language* of the computer (also sometimes referred to as the *binary language*). An 8-bit microprocessor (i.e. a microprocessor that recognizes a word length of 8 bits) can have a maximum of 2^8 (= 256) different instructions. Our own 8085 microprocessor which is also an 8-bit microprocessor has 74 different instructions with 246 bit patterns designed to perform various operations. Given below are a couple of (8-bit) instructions from the machine language of the 8085 microprocessor:

$(10101111)_2$: this instruction logically XORs the contents of a register called the *accumulator* (denoted by the symbol *A*) with itself and stores the result in *accumulator*.

$(10000000)_2$: this instruction adds the contents of a register called *B* with the contents of the *accumulator* and stores the result in the *accumulator*.

In the single board microcomputers that are generally available in our labs the instructions are entered as hexadecimal codes (through hex-keyboards) and not as binary numbers as mentioned above for the two instructions. The *hexcodes* (short for "hexadecimal codes") for the two binary instructions are *AF*H and *80*H respectively. A program stored in the ROM (read only memory) of the microcomputer called the *Monitor Program* translates the *hexcodes* into their binary equivalent.

Assembly Language

Since practically it is impossible for human beings to communicate with computers in the *binary language*, all binary instructions are given symbolic codes or abbreviated English type names called *mnemonics* that suggest the tasks that are to be performed by the instructions. And it is the set of *mnemonics* of a computer that forms its *assembly language*. Thus the *assembly language* like the *machine language* is also specific to each computer. Therefore a program written in the *assembly language* of the 8085 microprocessor cannot be made to run in a computer having a different microprocessor say the Motorolla 6800. In the *assembly language* of the 8085 microprocessor the *mnemonics* for the earlier mentioned binary instructions are given in the table below suggestive of the operations that the instructions can perform:

Binary Instruction	**Hexcode**	**Mnemonic**
10101111	*AF*H	*XRA A*
10000000	*80*H	*ADD B*

Here *XRA* stands for the logical operation *exclusive-OR* (i.e. *XOR*) and the symbol *A* for the *accumulator*. Similarly *ADD* stands for the arithmetic operation of addition and symbol *B* for register B.

A program written in *mnemonics* is called an *assembly language program*. By the end of this chapter we will be in a position to write ourselves some simple programs in the *assembly language* of the 8085 microprocessor.

Instruction Set and Format

The *instruction set* of the 8085 microprocessor as mentioned before consists of 74 instructions with 246 different bit patterns and all these instructions can be broadly classified into five functional groups depending upon the operations that they can perform. These functional groups are: *data transfer* (or *copy*) operations; *arithmetic* operations; *logic* operations; *branching* operations; and *machine-control* operations.

Let us see the operations that the various instructions of each of these functional groups are capable of performing:

(i) *Data transfer (or copy) operations*: data transfer is more of a misnomer for such instructions since these instructions copy data (and not transfer them) from one location called the source to another location called the destination without affecting the contents of the source location. Data transfer operations can further be classified into five different types. They are:

- (a) Transfer of data between registers: example is the instruction *MOV B,C* which copies data from register *C* into register *B*.
- (b) Placing of specific data byte into a register or a memory location: example is the instruction *MVI B*, *67*H (or *MVI M*, *4F*H) which places the data byte *67*H (or the data byte *4F*H) into the register *B* (or the memory location *M* whose address is contained in the register pair *HL*).
- (c) Transfer of data between a memory location and a register: example is the instruction *LDA 2005*H which copies data from a memory location having the address *2005*H into the accumulator.
- (d) Transfer of data between and I/O device and the accumulator: example is the instruction *IN 4B*H which copies data from an input device having port address *4B*H into the accumulator.
- (e) Transfer of data between a register pair and the stack: example is the instruction *PUSH B* which places the two data bytes from

register pair *BC* into the user defined memory locations called the *stack*.

(ii) *Arithmetic operations:* instructions falling under this category perform arithmetic operations such as addition, subtraction, increment (by one) and decrement (by one).

(a) For addition any 8-bit number (for example by using the instruction *ADI* 8-bit), or the contents of a register (using the instruction *ADD R*) or of a memory location (using the instruction *ADD M*) can be added to the contents of the accumulator and the sum is stored in the accumulator. It is to be noted that no two 8-bit registers can be added directly such as adding the contents of register *B* with that of register *D* is not permissible. However, you will see later that the instruction *DAD* R_p (where R_p stands for register pair) is an exception to this rule. It is used for the addition of two 16-bit data directly in register pairs.

(b) Like addition even for subtraction any 8-bit number (for example by using the instruction *SUI* 8-bit), or the contents of a register (using the instruction *SUB R*) or of a memory location (using the instruction *SUB M*) can be subtracted from the contents of the accumulator and the difference is stored in the accumulator. The subtraction is performed by the 2's complement method and the difference if negative is expressed in the 2's complement form. Again direct subtraction of two registers not involving the accumulator is not permissible. The accumulator has to be the minuend in all subtractions.

(c) For increment or decrement the 8-bit contents of a register (using the instruction *INR R* or *DCR R*) or of a memory location (using the instruction *INR M* or *DCR M*) can be incremented or decremented by 1. Moreover, the 16-bit data of a register pair (such as *BC*, *DE*, *HL* or *SP*) can also be incremented or decremented by 1 using the instructions *INX* R_p or *DCX* R_p respectively.

(iii)*Logic operations:* instructions under this category perform various logical operations with the contents of the accumulator such as:

(a) OR, AND, XOR: any 8-bit number (using the instructions ORI 8-bit, ANI 8bit, XRI 8-bit), or the contents of a register/memory location (using the instructions ORA R/M, ANA R/M, XRA R/M) can be logically ORed, ANDed, or Exclusively-ORed with the contents of the accumulator and the result is stored in the accumulator.

(b) Rotate: each bit in the accumulator can be shifted one position to the left (using the instruction RLC) or to the right (using the instruction RRC).

(c) Compare: any 8-bit number or the contents of a register/memory location (using instructions like *CPI 8-bit* and *CMP R/M*) can be compared for greater than, less than or equality with the contents of the accumulator.

(d) Complement: the accumulator contents can be complemented (that is performing the NOT operation, logical inversion) using the instruction *CMA* where all 1s are replaced by 0s and vice-versa.

(iv)*Branching operations:* instructions belonging to this group change the sequence of program execution either unconditionally or after testing a certain condition. Following are the two types of branching operations:

(a) Jump: there are unconditional (like *JMP*) as well as conditional Jump instructions (like *JC* and *JNC*) that alter the program

sequence within the main program either unconditionally or conditionally (depending on the status of a flag).

(b) Call, Return and Restart: such instructions alter the program sequence either by calling a subroutine (or restarting at a service routine) or by returning to the main program from a subroutine. Both the Call and the Return operations have unconditional as well as conditional instructions such as *CALL 16-bit*, *CNC 16-bit*, *RET* and *RNC*. Details about these instructions and others of such type are discussed later.

(v) *Machine control operations:* instructions of this category basically control the machine functions such as *HLT* (indicating the end of program) and *NOP* (indicating no operation to be performed by the computer).

We will again be returning to the these five categories of operations and discussing the instructions belonging to each of these categories in much greater detail, but before we do that we need to get a fair idea about the way instructions are designed and formatted. As was mentioned earlier an instruction is a command to the microprocessor to carry out a task on some data. This means that an instruction should have two parts in it: one should be the task (or the operation) to be performed and the other should be the data on which the operation is to be performed. Let us call the first part as *opcode* (short for operation code) and the second as *operand*. The *operand* (that is the data) can be specified directly as an 8-bit or a 16-bit data or indirectly through a register or a register pair or a memory location. Sometimes the *operand* is implicit in the *opcode* (that is included in the *opcode* itself). Now depending on the word size the instructions of the 8085 microprocessor are classified into three categories: 1-byte, 2-byte, and 3-byte instructions.

(a) 1-byte instructions: a 1-byte instruction includes both the operation as well as the data in a single byte called the *opcode*. Such instructions need one memory location to store in the user memory. Given below are a few examples of such instructions:

Instruction	**Operation**	**Operand**	**Hex-code**
MOV B,A	Move content of *A* into *B.*	Data specified through register *A.*	*47*H (*opcode* for the instruction)
DCR B	Decrement content of *B* by 1.	Data specified through register *B.*	*05*H (*opcode* for the instruction)
CMA	Complement the content of *A.*	Data specified through register *A* (implicit in *CMA*).	*2F*H (*opcode* for the instruction)

(b) 2-byte instructions: in a 2-byte instruction the first byte is always the *opcode* of the instruction and the second byte a data byte. Such instructions need two memory locations to store in the user memory. Given below are a few examples of such instructions:

Instruction	**Operation**	**Operand**	**Hex-code**
MVI B,8-bit	Move immediate *8-bit* data into *B*.	8-bit data is specified as the second byte of the instruction.	*06*H (*opcode* for the instruction)
ADI 8-bit	Add immediate *8-bit* data with	8-bit data is specified as the	*C6*H (*opcode* for the instruction)

	content of *A*. Store the sum in A.	second byte of the instruction.	

(c) 3-byte instructions: in a 3-byte instruction the first byte is always the *opcode* of the instruction and the second and third bytes are data bytes. The second byte is the lower order data byte and the third byte is the higher order data byte of a 16-bit data (generally a 16-bit address of a memory location). Such instructions need three memory locations to store in the user memory. Given below is an example of such instructions:

Instruction	**Operation**	**Operand**	**Hex-code**
JMP 16-bit	Jump to memory location having the *16-bit* memory address.	16-bit data (memory address) is specified as the second byte (lower order byte of the address) and third byte (higher order byte of the address) of the instruction.	*C3*H (*opcode* for the instruction)

Now let us see how instructions are formatted in the 8085 microprocessor that is how *opcodes* are designed. All registers, register pairs and operations have been given specific binary codes. Given below are some of these codes:

Register	**Code**
B	000
C	001
D	010
E	011
H	100
L	101
A	111

Register Pair	**Code**
BC	00
DE	01
HL	10

Operation	**Code**	**Comment**
Rotate each bit of accumulator by one position to the left.	00000111	The operation has an 8-bit code.
Add the contents of a register to that of the accumulator.	10000sss	The operation ADD has a 5-bit code 10000. The 3 bits sss are reserved for the code of a register.
Move the contents of the source register R_s to the destination register R_d.	01dddsss	The operation MOVE has a 2-bit code 01. The 3 bits ddd are reserved for the code of

		destination register R_d and the 3 bits sss are reserved for the code of source register R_s.

As an example let us see how the *opcodes* for various ADD and MOVE instructions are designed:

Mnemonic	*ADD B*	*ADD H*	*MOV C,A*	*MOV D,E*
Binary instruction (*Opcode*)	*10000 000*	*10000 100*	*01 001 111*	*01 010 011*
Hex Code (*Opcode* in hexadecimal)	*80*H	*84*H	*4F*H	*53*H

Now with the knowledge on the instructions of the 8085 microprocessor that we have gained till now we are in a position to understand and write simple and basic programs in its assembly language. A program in general is a sequence of instructions that tells the computer to perform a particular task. In the assembly language program the instructions are to be chosen from the instruction set of the 8085 microprocessor. Let us suppose that we want to write a simple program to add two bytes of data (say *45*H and *A3*H) and store the sum in register *H*. The logical steps required for performing this task should be:

(i) Load *45*H in the accumulator.
(ii) Load *A3*H in some other register say register *B*.
(iii) Add the contents of register *B* to that of the accumulator.
(iv) Store the answer in register *H*.
(v) Halt the program.

Let us see how these steps can now be translated into an assembly language program which can then be loaded into the user memory for execution:

Mnemonics	**Comments**	**User Memory Location**	**Hex Code**
*MVI A,45*H	Load *45*H in the accumulator.	*2000*H *2001*H	*3E*H *45*H
*MVI B,A3*H	Load *A3*H in register *B.*	*2002*H *2003*H	*06*H *A3*H
ADD B	Add the contents of register *B* to that of the accumulator. Result is stored in the accumulator.	*2004*H	*80*H
MOV H,A	Move the result in register *H*.	*2005*H	*67*H
HLT	Halt the program.	*2006*H	*76*H

For running the program in a computer we need to first look up for the *opcode* for each instruction in the instruction set given in the manual of the 8085 microprocessor. Once we have written the program with all the *hex codes* in proper sequence (matching one to one with the *mnemonics*) we can then load the program in the user memory sequentially starting preferably with its first memory location (*2000*H in the above example). After the program has been loaded in the user memory execution should begin from the first memory location. Initially the first *hex code* (which will be the *opcode* of the first instruction) will be fetched by the microprocessor from the user memory, decoded and

then finally the appropriate action will be executed. This cycle of fetch-decode-execute will then carry on sequentially, one instruction after the other, till the microprocessor encounters the halt instruction.

Addressing Modes

The different ways of specifying the data (operand) in instructions are called *addressing modes*. There are four addressing modes namely *immediate*, *register*, *direct* and *indirect*.

(i) *Immediate addressing mode*: in this mode the data itself is specified in the instruction. For example in the instruction *MVI R,8-bit* the data (*8-bit*) is itself specified. Similarly in the instructions *ADI 8-bit* or *LXI* R_p*,16-bit* the data (*8-bit* or *16-bit*) again is itself specified.

(ii) *Register addressing mode*: in this mode the data is specified in the instruction by the register in which it is present. For example in the instruction *MOV* R_d*,*R_s (or *ADD R*) the data is specified by the register R_s (or *R*) where it is stored.

(iii) *Direct addressing mode*: in this mode the address of a memory location or of an I/O device in which the data is present (or is to be moved) is directly specified in the instruction. For example in the instruction *LDA 16-bit* (or *IN 8-bit*) the 16-bit address of a memory location (or 8-bit port address of an input device) is directly specified in the instruction. Data from this memory location (or input device) is to be loaded into the accumulator.

(iv) *Indirect addressing mode*: in this mode the address of a memory location in which the data is present (or is to be moved) is specified indirectly by means of a register pair. For example in the instruction *MOV R,M* which transfers data from a memory location whose address is present in the register pair *HL* into register *R* the address of the memory location *M* is indirectly specified by the register pair *HL*. Similarly in the instruction *LDAX* R_p the data stored in a memory location whose address is present in the register pair R_p is to be transferred to the accumulator.

Instruction Types

As mentioned before, based on the types of operations that can be performed, instructions are classified into five different categories. They are: *data transfer*, *arithmetic*, *logical*, *branching* and *machine control*. Let us now study in more detail with appropriate examples each of these categories (please remember that the first byte of any instruction is its *opcode*):

(a) *Data transfer*: instructions of this type transfer data from a source which can be a register or a memory location or an I/O device into a destination which again can be a register or a memory location or an I/O device. In this transfer of data the contents of the source are not modified. Moreover, data transfer instructions do not affect any of the flags. Examples of commonly used instructions of such type are:

Mnemonic	**Type of instruction**	**Comments**
MOV R_d*,*R_s *(Here* R_d *and* R_s *can be any one of the registers A, B, C, D, E, H, or L)*	One byte instruction. Register addressing mode instruction.	Moves data from register R_s into register R_d without modifying the contents of register R_s. For example the instruction *MOV A,B* will copy the content of register *B* into the accumulator (i.e.

		register *A*).
MVI R,8-bit (Here R can be any one of the registers A, B, C, D, E, H, or L)	Two byte instruction. Immediate addressing mode instruction.	Move immediate *8-bit* data into register *R*. For example the instruction *MVI C,7FH* will load register *C* with the data byte *7FH*.
OUT 8-bit	Two byte instruction. Direct addressing mode instruction.	Send the content of accumulator to an output device whose 8-bit port address is mentioned as the second byte of the instruction. For example the instruction *OUT 87H* will load the output device having port address *87H* with the data byte of the accumulator.
IN 8-bit	Two byte instruction. Direct addressing mode instruction.	Accept the data into the accumulator from an input device whose 8-bit port address is mentioned as the second byte of the instruction. For example the instruction *IN 4EH* will load the accumulator with the data byte present at the input device having port address *4EH*.
LXI R_p, 16-bit (Here R_p can be any one of the register pairs BC, DE, HL, or SP)	Three byte instruction. Immediate addressing mode instruction.	Load register pair R_p immediate with 16-bit data. Here the second byte of the instruction goes to the lower order register and the third byte to the higher order register of the register pair R_p. For example the instruction *LXI D,207FH* will load the lower order byte *7FH* into the lower order register *E* and the higher order byte *20H* into the higher order register *D*.
MOV R,M (Here R can be any one of the registers A, B, C, D, E, H, or L)	One byte instruction. Indirect addressing mode instruction.	Moves data from a memory location *M* whose address is present in register pair *HL* into the register *R* without modifying the contents of the memory location *M*. For example

		the instruction *MOV D,M* will copy the content of a memory location whose address is present in register pair *HL* into the register *D*.
MOV M,R (Here R can be any one of the registers A, B, C, D, E, H, or L)	One byte instruction. Indirect addressing mode instruction.	Moves data from the register *R* into a memory location *M* whose address is present in register pair *HL* without modifying the contents of the register *R*. For example the instruction *MOV M,D* will copy the content of the register *D* into a memory location whose address is present in register pair *HL*.
LDAX R_p (Here R_p can be any one of the register pairs BC or DE)	One byte instruction. Indirect addressing mode instruction.	Load accumulator from memory location indirectly through register pair R_p. For example the instruction *LDAX B* will load the accumulator with the data byte present in a memory location whose 16-bit address is specified by the register pair *BC*.
STAX R_p (Here R_p can be any one of the register pairs BC or DE)	One byte instruction. Indirect addressing mode instruction.	Store the content of the accumulator into a memory location indirectly through register pair R_p. For example the instruction *STAX D* will store the data byte present in the accumulator into a memory location whose 16-bit address is specified by the register pair *DE*.
LDA 16-bit	Three byte instruction. Direct addressing mode instruction.	Load accumulator with the contents of a memory location whose 16-bit address is directly specified in the instruction. For example the instruction *LDA 207DH* will load the accumulator with the data byte present in the memory location *207DH*.

STA 16-bit	Three byte instruction. Direct addressing mode instruction.	Store the content of the accumulator in a memory location whose 16-bit address is directly specified in the instruction. For example the instruction *STA 20B4H* will store the accumulator data byte in the memory location *20B4H*.
LXI R_p, 16-bit *R_p can be BC, DE, HL, or SP.*	Three byte instruction.	Loads the register pair R_p with the 16-bit mentioned in the instruction (as the second and third bytes).
PUSH R_p *R_p can be BC, DE, HL, or PSW*	One byte instruction.	Pushes the contents (two bytes) of the specified register pair onto the stack.
POP R_p *R_p can be BC, DE, HL, or PSW*	One byte instruction.	Pops the top two bytes from a stack to the specified register pair.
LHLD 16-bit	Three byte instruction.	Copy the data of memory location with the *16-bit* address into register *L* and the data of the next memory location into register *H*. No flags are modified.
SHLD 16-bit	Three byte instruction.	Copy the data of register *L* into the memory location with the *16-bit* address and the data of register *H* into the next memory location. No flags are modified.
PCHL	One byte instruction.	Load Program Counter with the 16-bit data of register pair *HL*. No flags are affected.
SPHL	One byte instruction.	Load the Stack Pointer with the 16-bit data of register pair *HL*. No flags are affected.
XCHG	One byte instruction.	Exchange data of registers *H* and *L* with *D* and *E* respectively. No flags are modified.
XTHL	One byte instruction.	Exchange data of registers *H* and *L* with top two memory locations of Stack. No flags are modified.

(b) *Arithmetic*: the various arithmetic operations that the 8085 microprocessor performs through instructions are addition, subtraction, increment and decrement. Some of the commonly used instructions with their mnemonics are as given below:

Mnemonic	**Type of instruction**	**Comments**
ADD R *(Here R can be any one of the registers A, B, C, D, E, H, or L)*	One byte instruction. Register addressing mode instruction.	Adds the data in register *R* with the data in the accumulator and stores the sum in the accumulator. All flags are modified reflecting the data conditions of the result in the accumulator. The data in register *R* remains unchanged.
ADI 8-bit	Two byte instruction. Immediate addressing mode instruction.	Adds the second byte of the instruction (first being the opcode) with the data in the accumulator and stores the sum in the accumulator. All flags are modified reflecting the data conditions of the result in the accumulator.
SUB R *(Here R can be any one of the registers A, B, C, D, E, H, or L)*	One byte instruction. Register addressing mode instruction.	Subtracts the data in register *R* from the data in the accumulator and stores the difference in the accumulator. All flags are modified reflecting the data conditions of the result in the accumulator. The data in register *R* remains unchanged.
SUI 8-bit	Two byte instruction. Immediate addressing mode instruction.	Subtracts the second byte of the instruction (first being the opcode) with the data in the accumulator and stores the difference in the accumulator. All flags are modified reflecting the data conditions of the result in the accumulator.
ADD M *SUB M*	One byte instructions. Indirect addressing mode instructions.	Add/subtract the data in a memory location whose address is present in the register pair *HL* with the data in the accumulator and store the result in the accumulator. All flags are affected.
INR R *DCR R*	One byte instructions. Register addressing mode instructions.	Increment/decrement the data in the register *R* by 1. These instructions affect all flags except the

		Carry flag.
INX R_p *DCX* R_p	One byte instructions. Register addressing mode instructions.	Increment/decrement the data in the register pair R_p by 1. These instructions do not affect the flags.
INR M *DCR M*	One byte instructions. Indirect addressing mode instructions.	Increment/decrement the data in a memory location whose address is present in register pair *HL* by 1. These instructions affect all flags except the Carry flag.
ADC R/M	One byte instruction.	Add the data in register *R* or memory location *M* along with the bit in carry flag with the data in the accumulator and store the result in the accumulator. All flags are affected.
SBB R/M	One byte instruction.	Subtract the data in register *R* or memory location *M* along with the bit in carry flag (borrow) from the data in the accumulator and store the result in the accumulator. All flags are affected.
ACI 8-bit	Two byte instruction.	Add 8-bit data along with CY flag bit to the data in accumulator and store the result in accumulator. All flags are affected.
DAD R_p R_p *can be BC, DE, HL, or SP.*	One byte instruction.	Add the specified register pair R_p data with that of *HL* and store the result in *HL*. Only CY flag is affected.

(c) *Logical*: the various logical operations that the 8085 microprocessor performs through instructions are are as given below:

Mnemonic	**Type of instruction**	**Comments**
ANA R/M	One byte instruction.	Logically AND the data in register *R* or memory location *M* with the data in the accumulator and store the result in the accumulator. All flags except *CY* and *AC* are modified reflecting the data conditions of the result in the

		accumulator. *CY* flag is reset and *AC* flag is set.
ANI 8-bit	Two byte instruction.	Logically AND the second byte of the instruction with the data in the accumulator and store the result in the accumulator. All flags except *CY* and *AC* are modified reflecting the data conditions of the result in the accumulator. *CY* flag is reset and *AC* flag is set.
ORA R/M	One byte instruction.	Logically OR the data in register *R* or memory location *M* with the data in the accumulator and store the result in the accumulator. All flags except *CY* and *AC* are modified reflecting the data conditions of the result in the accumulator. Both *CY* flag and *AC* flag are reset.
ORI 8-bit	Two byte instruction.	Logically OR the second byte of the instruction with the data in the accumulator and store the result in the accumulator. All flags except *CY* and *AC* are modified reflecting the data conditions of the result in the accumulator. Both *CY* flag and *AC* flag are reset.
XRA R/M	One byte instruction.	Logically XOR the data in register *R* or memory location *M* with the data in the accumulator and store the result in the accumulator. All flags except *CY* and *AC* are modified reflecting the data conditions of the result in the accumulator. Both *CY* flag and *AC* flag are reset.
XRI 8-bit	Two byte instruction.	Logically XOR the second byte of the instruction with the data in the accumulator and

		store the result in the accumulator. All flags except *CY* and *AC* are modified reflecting the data conditions of the result in the accumulator. Both *CY* flag and *AC* flag are reset.
CMA	One byte instruction.	Complement (logical NOT) the contents of the accumulator.
RLC	One byte instruction.	Rotate each bit in the accumulator by one position to the left with the MSB shifting to the LSB position as well as to the CY flag.
RAL	One byte instruction.	Rotate each bit in the accumulator by one position to the left with the MSB shifting to the CY flag and the CY flag bit shifting to the LSB position.
RRC	One byte instruction.	Rotate each bit in the accumulator by one position to the right with the LSB shifting to the MSB position as well as to the CY flag.
RAR	One byte instruction.	Rotate each bit in the accumulator by one position to the right with the LSB shifting to the CY flag and the CY flag bit shifting to the MSB position.
CMP R/M	One byte instruction.	Compare the data in register *R* or memory location *M* with the data in the accumulator for equality, greater than or less than. The flags are modified according to the subtraction result of *A–R/M*. However, the accumulator retains its earlier value and not the difference.
CPI 8-bit	Two byte instruction.	Compare the second byte of the instruction with the data in the accumulator for equality, greater than or less than.

		The flags are modified according to the subtraction result of *A–8-bit* data. However, the accumulator retains its earlier value and not the difference. If (i) A=data, then Z flag is set. (ii) A>data, then CY flag is reset. (iii)A<data, then CY flag is set.
CMC	One byte instruction.	Complement the CY flag bit. No other flag is modified.
STC	One byte instruction.	Set the CY flag. No other flag is modified.
DAA	One byte instruction.	Decimal Adjust Accumulator. It converts the accumulator data from binary to BCD using the AC flag internally.

(d) *Branching*: instructions of this type are used in the 8085 microprocessor for changing the sequence of program execution. They are broadly classified into three types: *Jump* instructions; *Call* and *Return* instructions; and *Restart* instructions. *Call* and *return* instructions are associated with *subroutines* while *restart* instructions with *interrupts*. Let us focus on the *Jump* instructions as of now. They are all three byte instructions which make the program control jump to a specific memory location for further execution either unconditionally or if a certain condition is satisfied. In case of conditional jumping the instructions test the status of one of the four flags *S*, *Z*, *P*, and *CY*. Let us see them in more detail:

Mnemonic	**Type of instruction**	**Comments**
JMP 16-bit	Three byte instruction.	Jump the program control unconditionally to the memory location specified by the 16-bit address mentioned in the instruction (as second and third bytes of the instruction).
JC 16-bit	Three byte instruction.	Jump if Carry flag is set to the 16-bit address.
JNC 16-bit	Three byte instruction.	Jump if Carry flag is not set to the 16-bit address.
JZ 16-bit	Three byte instruction.	Jump if Zero flag is set to the 16-bit address.
JNZ 16-bit	Three byte instruction.	Jump if Zero flag is not set to the 16-bit

		address.
JP 16-bit	Three byte instruction.	Jump on Plus i.e. if sign flag is reset to the 16-bit address.
JM 16-bit	Three byte instruction.	Jump on Minus i.e. if sign flag is set to the 16-bit address.
JPE 16-bit	Three byte instruction.	Jump if Parity flag is set to the 16-bit address.
JPO 16-bit	Three byte instruction.	Jump if Parity flag is reset to the 16-bit address.
CALL 16-bit	Three byte instruction.	Call a subroutine i.e. transfer the program control to starting memory location of a subroutine specified by the 16-bit address mentioned in the instruction. There are conditional Call instructions as well.
RET	One byte instruction.	Return to the main program after completing the subroutine. There are conditional return instructions as well.

(e) *Machine control*: instructions falling under this category, as suggested by the name, control machine operations. A couple of such examples are as given below:

Mnemonic	**Type of instruction**	**Comments**
HLT	One byte instruction.	Halts any further execution and makes the microprocessor enter into a wait state. The buses are placed in high impedance state.
NOP	One byte instruction.	No operation is to be executed. Generally used while trouble shooting a program.

SIM and RIM

Let us now see more closely two multi-purpose one byte instructions primarily associated with the interrupts namely *SIM* and *RIM*. *SIM* stands for Set Interrupt Mask and *RIM* for Read Interrupt Mask.

SIM is used for implementing the interrupts RST 7.5, 6.5, and 5.5 and for serial data output. The instruction reads the 8-bit data in the accumulator and enables/disables the interrupts according to its interpretation of the data. The instruction interprets the 8-bit data of the accumulator as follows:

SOD	SDE	XXX	R7.5	MSE	M7.5	M6.5	M5.5
D7	D6	D5	D4	D3	D2	D1	D0

Bit D7 of the accumulator is latched into the Serial Output Data (SOD) output line (pin number 4) and made available to a serial output device if bit D6 = 1 (i.e. the Serial Data Enable (SDE) bit is high). Bit D5 is XXX i.e. a Don't Care bit.
M7.5 i.e. the RST 7.5 Interrupt Masking bit D2 is used for masking (disabling) the interrupt RST 7.5. If D2 = 0 then RST 7.5 is enabled else masked. Similarly the M6.5 and the M5.5 bits are used to mask the interrupts RST 6.5 and RST 5.5 respectively.
MSE (Mask Set Enable) is a master control over all the interrupt masking bits (D2, D1, and D0) and thus bit D3 should be high for the bits D2, D1, and D0 to be effective.
R7.5 (Reset RST 7.5) i.e. the D4 bit is an additional control to mask the interrupt RST 7.5.

The instruction *RIM* on the other hand is used to load the accumulator with 8-bits indicating the current status of the interrupt masks, the interrupt enable, pending interrupts, and serial input data according to the format given below:

SID	I7	I6	I5	IE	7.5	6.5	5.5
D7	D6	D5	D4	D3	D2	D1	D0

SID (Serial Input Data) i.e. bit D7 will come from pin number 5. I7, I6, I5 will be high if corresponding interrupt is found pending. If Interrupt Enable flip flop is set then IE bit is high. Bits D2, D1, D0 will be high if corresponding interrupt is found masked.

Summary

So we have seen in this chapter how the various operations that the microprocessor can perform through instructions can be classified into five different categories. Also towards the end we did a couple of instructions that are primarily associated with the externally initiated operations called the interrupts. Now it is time that we did some exercises to further strengthen our understanding of this subject.

Exercises

1. The accumulator contains C5H and the CY flag is set. What will the accumulator and CY flag contain following each of the instruction given below?
   (i) XRA A
   (ii) ADI 91H
   (iii) RRC
2. Write instructions to enable interrupt RST6.5 and mask other interrupts.
3. After the execution of instruction RIM, the accumulator contained 49H. Explain the accumulator contents.
4. How many bytes instructions are the following:
   (i) MOV A,B
   (ii) MOV A,M
5. How many bytes instructions are the following:
   (i) JNC 2060H
   (ii) LXI H, 2072H
6. How many bytes instructions are the following:
   (i) IN 84H
   (ii) ADI 6AH
7. Suppose after executing the instruction ADD the flag register contains 45H. What does it indicate?
8. How many T states are used in executing the instruction MOV B,A?
9. How many T states are used in executing the instruction MOV B,M?
10. Which of the following instructions is/are direct addressing mode instruction(s)?

(i) LXI
(ii) LDAX
(iii) STA
(iv) STAX

11. Which of the following instructions is/are indirect addressing mode instruction(s)?
    (i) MOV A,B
    (ii) MOV M,A
    (iii) STA
    (iv) LDA
12. Which of the following instructions is/are immediate addressing mode instruction(s)?
    (i) LXI
    (ii) MVI
    (iii) IN
    (iv) ADI
13. Which of the following instructions is/are register addressing mode instruction(s)?
    (i) MOV A,H
    (ii) MOV A,M
    (iii) LXI
    (iv) MVI
14. What instruction can be used for 16-bit addition in register pair HL?
15. What instruction can be used to retain the accumulator data but clear the carry flag at the same time?
16. Which out of the following is used for incrementing the value stored in a register pair?
    (i) INR
    (ii) INX
17. Which out of the following is used for decrementing the value stored in an 8-bit register?
    (i) DCR
    (ii) DCX
18. Can the data present in memory location 20A0H be transferred to the accumulator using a single instruction?
19. How is the instruction RAL different from RLC?
20. How many different memory locations can be addressed by the 8085 microprocessor?

Hints:

1. (i) A=00H; CY=0; (ii) A=56H; CY=1; (iii) A=E2H; CY=1
2. MVI A, 0DH
   SIM
3.

0	1	0	0	1	0	0	1
D7	D6	D5	D4	D3	D2	D1	D0

4. (i) 1; (ii) 2
5. (i) 3; (ii) 3
6. (i) 2; (ii) 2
7.

0	1	0	0	0	1	0	1
S	Z	X	AC	X	P	X	CY

8. 4
9. 7
10. (iii)
11. (ii)
12. (i), (ii), and (iv)
13. (i)

14. DAD
15. ORA A or ADI 00H
16. (ii)
17. (i)
18. Yes; LDA 20A0H
19. In RAL the bits rotate left through carry flag. MSB moves to the carry flag and the carry flag bit moves to LSB position. In RLC the MSB moves to the carry flag as well as to the LSB position. The previous carry flag bit is lost.
20. 65,536 or 64 k

References

1. Microprocessor Architecture, Programming and Applications with the 8085 By Ramesh S. Gaonkar (Prentice Hall, 2002).
2. Microprocessor Architecture, Programming and Systems featuring the 8085 By William A. Routt (Thomson Delmar Learning, 2006)
3. Microprocessors and Programmed Logic, 2nd Edition by Kenneth L Short (P.H.I., 1988).