Unit 5
Assembly Language Programming

Programming with Intel 8085 microprocessor

Instruction Format

On the basis of size they occupy, the instruction formats are:

i) One byte instruction
   It includes the op-code and operand in the same byte (all register transfer). E.g. MOV A, B

ii) Two byte instruction
   The first byte specifies op-code and second byte specifies the operand. E.g. MOV A, 32H

iii) Three byte instruction
   The first byte specifies op-code and following two byte specifies 16-bit operand. E.g. LXI B, 2032H

Instruction Types

According to their functions, 8085 instructions can be classified as:

1) Data transfer
2) Arithmetic
3) Logical
4) Branching
5) Miscellaneous

1) Data transfer instructions

i) MOV
   Syntax: MOV R_d, R_s
   MOV M, R_s
   MOV R_d, M

ii) MVI
   Syntax: MVI R_d/M, 8-bit data

iii) LXI
   Syntax: LXI Reg. pair, 16-bit data
   e.g. LXI B, 8085

iv) LDA
   Syntax: LDA 16-bit address
   e.g. LDA 8085
   $A \leftarrow M[8085]$

v) STA
   Syntax: STA 16-bit address
   e.g. STA 8085
   $M[8085] \leftarrow A$

vi) LDAX
   Syntax: LDAX B/D register pair
   e.g. LDAX B
   $A \leftarrow M[8050]$ (if B=80, C=50)

vii) STAX
   Syntax: STAX B/D register pair

e.g. STAX B
M [8050]←A (if B=80, C=50)

viii) LHLD
Syntax: LHLD 16-bit address
e.g. LHLD 8085
L←M [8085]
H←M [8086]

ix) SHLD
Syntax: SHLD 16-bit address
e.g. SHLD 8085
M [8085]←L
M [8086]←H

x) XCHG
Syntax: XCHG ; interchanges between D and H; and E and L
H←D and L←E
D←H and E←L

xi) IN
Syntax: IN 8-bit port address
e.g. IN 15 ; data received from port address 15 is transferred into Accumulator.

xii) OUT
Syntax: OUT 8-bit port address
e.g. OUT 15 ; content of Accumulator is transferred to port address 15.

2) Arithmetic Instructions

i) ADD
Syntax: ADD R/M
e.g. ADD B ; A←A+B

ii) ADC
Syntax: ADC R/M
e.g. ADC B ; A←A+B+CY

iii) ADI
Syntax: ADI 8-bit data
e.g. ADI 45 ; A←A+45

iv) ACI
Syntax: ACI 8-bit data
e.g. ACI 45 ; A←A+45+CY

v) SUB
Syntax: SUB R/M
e.g. SUB B ; A←A-B

vi) SBB
Syntax: SBB R/M
e.g. SBB B ; A←A-B-CY

vii) SUI
Syntax: SUI 8-bit data
e.g. SUI 45 ; A←A-45

viii) SBI

Syntax: SBI 8-bit data

e.g. SBI 45 ; A←A-45-CY

ix) DAD

Syntax: DAD register pair ; contents of register pair is added to HL and the sum is saved in HL pair.

e.g. DAD B ; HL←HL+BC

x) INR

Syntax: INR R/M

e.g. INR B ; B←B-1

xi) DCR

Syntax: DCR R/M

e.g. DCR C ; C←C-1

xii) INX

Syntax: INX register pair

e.g. INX H ; HL←HL+1

xiii) DCX

Syntax: DCX register pair

e.g. DCX B ; BC←BC-1

3) Logical instructions

i) ANA

Syntax: ANA reg./M

e.g. ANA B ; A←A /\ B

ii) ANI

Syntax: ANI 8-bit data

e.g. ANI 23H ; A←A /\ 23H

iii) ORA

Syntax: ORA Reg./M

e.g. ORA B ; A←A \/ B

iv) ORI

Syntax: ORI 8-bit data

e.g. ORI 23H ; A←A \/ 23H

v) XRA

Syntax: XRA Reg./M

e.g. XRA B ; A←A ⊕ B

vi) XRI

Syntax: XRI 8-bit data

e.g. XRI 23H ; A←A ⊕ 23H

vii) CMA

e.g. CMA ; complements the content of accumulator

viii) CMP

Syntax: CMP Reg./M

The contents of the operand (register/memory) are compared with the content of accumulator and both contents are preserved and the comparison is shown by setting the flags.

If A>Reg./M, CY=0, Z=0

If A=Reg./M, CY=0, Z=1

If A<Reg./M, CY=1, Z=0

ix) CPI
Syntax: CPI 8-bit data
If A>data, CY=0, Z=0
If A=data, CY=0, Z=1
If A<data, CY=1, Z=0

x) RLC
Each binary bit of the accumulator is rotated left by one position. Bit D7 is placed in the position of D0 as well as in the carry flag (CY). Other flags are not affected.
Syntax: RLC
Before RLC: 10100111, CY=0
After RLC: 01001111; CY=1

xi) RRC
Syntax: RRC
Before RLC: 10100111, CY=0
After RLC: 11010011; CY=1

xii) RAL
Each binary bit of the accumulator is rotated left by one position through the carry flag (CY). Other flags are not affected.
Syntax: RAL
Before RAL: 10100111, CY=0
After RAL: 01001110; CY=1

xiii) RAR
Each binary bit of the accumulator is rotated right by one position through the carry flag. Other flags are not affected.
Syntax: RAL
Before RAL: 10100111, CY=0
After RAL: 01010011; CY=1

xiv) DAA
Decimal Adjust Accumulator
The contents of accumulator are changed from binary value to two binary-coded decimal (BCD) digits. This is the only instruction that uses the AC flag to perform binary to BCD conversion. All flags are affected.

4) Branching instructions

a) Jump instructions

i) Unconditional jump
The program sequence is transferred to the memory location specified by the 16-bit address without checking any condition.
Syntax: JMP 16-bit address
e.g. JMP 8085

ii) Conditional jump

Op-code	**Description**	**Flag status**
JC	Jump on carry	CY=1
JNC	Jump on no carry	CY=0
JP	Jump on positive	S=0
JM	Jump on minus	S=1
JPE	Jump on even parity	P=1
JPO	Jump on odd parity	P=0

JZ	Jump on zero	Z=1
JNZ	Jump on non-zero	Z=0

b) Call instructions / Return instructions

i) Unconditional call/return

Syntax: CALL 16-bit address
```
        :
        :
        :
        RET
```

ii) Conditional Call/return

Op-code	Description	Flag status
CC/RC	Call/Return on carry	CY=1
CNC/RNC	Call/Return on no carry	CY=0
CP/RP	Call/Return on positive	S=0
CM/RM	Call/Return on minus	S=1
CPE/RPE	Call/Return on even parity	P=1
CPO/RPO	Call/Return on odd parity	P=0
CZ/RZ	Call/Return on zero	Z=1
CNZ/RNZ	Call/Return on non-zero	Z=0

c) Restart instructions

The RST instructions are equivalent to 1-byte call instructions to one of the eight memory locations on page 0. The instructions are generally used in conjunction with interrupts and inserted using external hardware. However, these can be used as software instructions in a program to transfer program execution to one of the eight locations.

Op-code	Restart Address (H)
RST 0	0000
RST 1	0008
RST 2	0010
RST 3	0018
RST 4	0020
RST 5	0028
RST 6	0030
RST 7	0038

5) Miscellaneous instructions

i) PUSH

Syntax: PUSH reg. pair

e.g. PUSH B

Before instruction	After instruction

B=32, C=57
SP=2099
	M [2097]=57
	M [2098]=32
	SP=2097

ii) POP

Syntax: POP reg. pair

e.g. POP H

Before instruction	After instruction

SP=2090
M [2090]=F5
M [2091]=01
	H=01
	L=F5
	SP=2092

iii) EI
Enabling interrupts

iv) DI
Disabling interrupts

v) PCHL
The contents of registers H and L are copied to the program counter.

vi) SPHL
The contents of registers H and L are copied to the stack pointer register.

vii) XTHL
Exchange H and L with Top of Stack

6) Machine control instructions

i) HLT
Halt and enter wait state. The contents of the registers are unaffected during the HLT state.

ii) NOP
No operation is performed. The instruction is fetched and decoded; however, no operation is executed. The instruction is used to fill in time delays or to delete and insert instructions while troubleshooting.

iii) RIM
Read Interrupt Mask. This is a multipurpose instruction used to read the status of interrupts 7.5, 6.5, 5.5 and to read serial data input bit. The instruction loads 8 bits in the accumulator with the following interpretations:

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

D7=serial input data bit
D6, D5, D4=interrupts pending if bit=1
D3=interrupt enable; flip-flop is set if bit=1
D2, D1, D0=interrupt masked if bit=1
e.g. After the execution of instruction RIM, the accumulator contained 49H. Explain
(A): 49H = 0 1 0 0 1 0 0 1
RST 7.5 is pending.
Interrupt enable flip-flop is set.
RST 7.5 and 6.5 are enabled. RST 5.5 masked.

iv) SIM
Set Interrupt Mask. This is a multipurpose instruction and used to implement the 8085 interrupts (RST 7.5, 6.5, and 5.5) and serial data output.
The instruction interprets the accumulator contents as follows:

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

D7=serial output data
D6=serial data enable (1=enable and 0=disable)
D4=if 1, reset RST 7.5 flip-flop
D3=if 1, mask set enable

D2, D1, D0=masks interrupts if bits=1

- **SOD** – Serial Output Data: Bit D7 of the accumulator is latched into the SOD output line and made available to a serial peripheral if bit D6=1.
- SDE – Serial Data Enable: If this bit = 1, it enables the serial output. To implement serial output, this bit needs to be enabled.
- **XXX** – Don't care.
- **R7.5** – Reset RST 7.5: If this bit = 1, RST 7.5 flip-flop is reset. This is an additional control to reset RST 7.5.
- **MSE** – Mask Set Enable: If this bit is high, it enables the functions of bits D2, D1, D0. This is a master control over all the interrupt masking bits. If this bit is low, bits D2, D1, and D0 do not have any effect on the masks.
- **M7.5** – D2 = 0, RST 7.5 is enabled.
  = 1, RST 7.5 is masked or disabled.
- **M6.5** – D1 = 0, RST 6.5 is enabled.
  = 1, RST 6.5 is masked or disabled.
- **M5.5** – D0 = 0, RST 5.5 is enabled.
  = 1, RST 5.5 is masked or disabled.

Simple Sequence Programs, Branching, Looping

- Explained inside the class and lab.

Programming with Intel 8086 microprocessor

Assembly Instruction Format: Op-codes, mnemonics and operands

Machine Language:

There is only one programming language that any computer can actually understand and execute: its own native binary machine code. This is the lowest possible level of language in which it is possible to write a computer program. However, binary code is not native to humans and it is very easy for error to occur in the program. These bugs are not easy to determine in a pool of binary digits. Also for CPU like 8086 it is tedious and virtually impossible to memorize thousands of binary instructions codes.

Example

Program memory address	Content (binary)	Content (Hex)
00100H	0000 0100	04h

Assembly Language:

Programs in assembly language are represented by certain words representing the operation of instruction. Thus programming gets easier. Assembly language statements are generally written in a standard form that has four fields.

- Label field
- Op-code field (Instruction or Mnemonic)
- Operand field
- Comment field

Let us consider a simple example to add two numbers

```
Label   Mnemonic    Operand     Comment
Start:  MVI         A, 10h      ; Move 10h into accumulator
        MVI         B, 20h      ; Move 20h into register B
        ADD         B           ; Add the contents of register B with accumulator
```

Thus, we see the ease with the assembly language rather than machine language.

Mnemonic is a short alphabetic code used in assembly language for microprocessor operation. These are words (usually two-to-four letter) used to represent each instruction.

Assembler is software (program module) which converts assembly language code (source module) into a machine language code.

Types of Assemblers

One Pass Assembler:

- Goes (scans) through the program once
- Can resolve backward reference
- Cannot resolve forward reference

Two Pass Assembler:

- Goes (scans) through the assembly language program twice
- First pass generates the table of the symbol, which consists of labels with addresses assigned to them
- Second pass uses the symbol table generated in the first pass to complete the object code for each instruction thus producing machine code
- It can resolve forward and backward reference both.

Backward reference	Forward reference
L1: …..	JMP L1
…..	…..
…..	…..
…..	…..
JMP L1	L1: …..

Linker is a program that links object file created by assembler into a single executable file.

However, it must be remembered that for execution these codes are converted to machine codes. This is done by assembler. An assembler translates a program written in assembly language into machine language program (object code). Assembly language program are called "source codes". Machine language programs are known as object codes. A translator converts source codes to object codes and then into executable formats. The process of
converting source code into object code is called compilation and assembler does it. The process of converting object codes into executable formats is called linking and ***linker*** does it.

*.asm ~~assembler~~ *.obj ~~linker~~ *.exe

Macro assembler

Macro assembler is an assembly language that allows macros to be defined and used. The Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows.

It translates a program written in macro language into the machine language. A macro language is the one in which all the instruction sequence can be defined in macro block. A macro is an instruction sequence that appears repeatedly in the program assigned with specific name. The MASM replaces a macro name with appropriate instruction sequence whenever it encounters a macro name. E.g.

```
Initiz macro
        Mov ax, @dataseg
        Mov ds, ax
        Mov es, ax
Endm

OR,     Initiz { Mov ax, @dataseg
                 Mov ds, ax
                 Mov es, ax }
```

There exists little difference between macro program and subroutine program.
When a subroutine program occurs in program, execution jumps out of main program and executes subroutine and control returns to the main program after RET instruction. A macro, in the other hand, does not cause program execution to branch out of main program. Each time a macro occurs, it is replaced with appropriate sequence in the main program.

Advantages of using Macro

- To simplify and reduce the repetitive coding.
- To reduce errors caused by repetitive coding.
- To make assembly language program more readable.

<u>Assembler Directives</u>

The instruction that will give the information to assembler for performing assembling are called assembler directives. They are not executed by MP, so they are called dummy or pseudo instructions. It gives direction to the assembler.

Following are some commonly used directives:

1) **Segment and Ends directive**

```
Segment_name SEGMENT
…………
…………
…………
Segment_name ENDS

e.g.    dataseg segment
        a dw 1234h
        b dw 4321h
        c dw ?
        dataseg ends
```

2) **Proc and Endp directive**

```
Procedure_name proc
……...
………
```

………
Procedure_name endp

3) **END directive**
- Ends the entire program and appears as the last statement.

END procedure_name

4) **ASSUME directive**

Assume CS: CODE_HERE, DS: DATA_HERE

5) **PAGE directive**
- Specifies the maximum number of lines the assembler is to list on a page and the maximum number of characters on a line.

PAGE [length], [width]   ; maximum is page 10 to 255, 60 to 132
e.g. page 60, 132
- length is 60 lines per page and width is 132 characters per line.

6) **TITLE directive**

Title text

7) **EQU directive**
- Is used to assign name to constant used in your program.

pi equ 3.1417 ; pi=3.1417

8) **DUP directive**

Price dw 4 dup(0)
Weight db 100 dup(?)

9) **Data Definition directive**

Directive	**Description**	**Size (byte)**	**Attribute**
DB	Define byte	1	Byte
DW	Define word	2	Word
DD	Define double word	4	Double word
DQ	Define quad word	8	Quad word
DT	Define ten byte	10	Ten byte

10) **ORG directive**

It is used to assign the starting memory location to the program.
Org 3000H

11) **Macro and Endm directive**
- Used to define macro module.

12) **OFFSET directive**

It is an operator which informs the assembler to determine the displacement of named data or variable from the start of the segment, which contains it.

Simplified Segment Directives

.model (memory model)

Memory model	**Description**
Tiny	Code and data together may not be greater than 64K
Small	Neither code nor data may be greater than 64K
Medium	Only the code may be greater than 64K
Compact	Only the data may be greater than 64K
Large	Both code and data may be greater than 64K
Huge	All available memory may be used for code and data

.stack
The .stack directive sets the size of the program stack, which may be any size up to 64K. This segment is addressed by SS and SP registers.

.code
The .code directive identifies the part of the program that contains instructions. This segment is addressed by CS and IP registers.

.data
All variables are defined in this segment area.

Instruction sets

- Provided in the photocopy. Study the following.

1) Data transfer: MOV, IN, OUT, LEA
   i) MOV R/M, R/M/Imm
      Copy byte or word from specified source to specified destination
   ii) IN AL/AX, port no/DX
      Copy a byte or word from specified port number to accumulator
      Second operand is a port number. If required to access port number over 255 - **DX** register should be used.
   iii) OUT port no/DX, AL/AX
      Copy a byte or word from accumulator to specified port number
      First operand is a port number. If required to access port number over 255 - **DX** register should be used.
   iv) LEA R, M
      Load effective address of operand into specified register
2) Arithmetic: ADD, SUB, INC, DEC, MUL, DIV, CMP, DAA, AAA
   i) ADD R/M, R/M/Imm
      Add specified byte to byte or specified word to word
   ii) SUB R/M, R/M/Imm
      Subtract specified byte from byte or specified word from word
   iii) INC R/M
      Increment specified byte or word by 1
   iv) DEC R/M
      Decrement 1 from specified byte or word
   v) MUL R/M
      Multiplies an unsigned multiplicand by an unsigned multiplier

AL or AX is assumed as multiplicand.

vi) DIV R/M

Divides an unsigned dividend (accumulator) by an unsigned divisor (register)

vii) CMP R/M, R/M/Imm

Compare two specified bytes or two specified words

	CF	SF	ZF
Operand1>Operand2	0	0	0
Operand1=Operand2	0	0	1
Operand1<Operand2	1	1	0

viii) DAA

Decimal adjust After Addition.

Corrects the result of addition of two packed BCD values

Algorithm:

if low nibble of AL > 9 or AF = 1 then:

- AL = AL + 6
- AF = 1

If AL > 9Fh or CF = 1 then:

- AL = AL + 60h
- CF = 1

Example:

```
MOV AL, 0Fh   ; AL = 0Fh (15)
DAA           ; AL = 15h
RET
```

ix) AAA

ASCII Adjust after Addition.

Corrects result in AH and AL after addition when working with BCD values.

It works according to the following Algorithm:

if low nibble of AL > 9 or AF = 1 then:

- AL = AL + 6
- AH = AH + 1
- AF = 1
- CF = 1

Else

- AF = 0
- CF = 0

in both cases:

clear the high nibble of AL.

Example:

```
MOV AX, 15   ; AH = 00, AL = 0Fh
AAA          ; AH = 01, AL = 05
RET
```

3) Logic: AND, OR, XOR, NOT, ROR, RCR, ROL, RCL, SHL, SHR

i) AND/OR/XOR R/M, R/M/Imm

ii) NOT R/M

Invert each bit of a byte or word

iii) RCL/RCR
Syntax: RCL/RCR R/M, CL/Imm

RCL	**RCR**
Rotate operand1 left through Carry Flag. The number of rotates is set by operand2.	Rotate operand1 right through Carry Flag. The number of rotates is set by operand2.
Algorithm: shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position.	Algorithm: shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position.
Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCL AL, 1 ; AL = 00111001b, CF=0. RET	Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCR AL, 1 ; AL = 10001110b, CF=0. RET

iv) ROL/ROR
Syntax: ROL/ROR R/M, CL/Imm

ROL	**ROR**
Rotate operand1 left. The number of rotates is set by operand2.	Rotate operand1 right. The number of rotates is set by operand2.
Algorithm: shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position.	Algorithm: shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position.
Example: MOV AL, 1Ch ; AL = 00011100b ROL AL, 1 ; AL = 00111000b, CF=0. RET	Example: MOV AL, 1Ch ; AL = 00011100b ROR AL, 1 ; AL = 00001110b, CF=0. RET

v) SHL/SHR
Shift logical left / Shift logical right
Syntax: SHL/SHR R/M, CL/Imm

SHL	**SHR**
Shift operand1 Left. The number of shifts is set by operand2.	Shift operand1 Right. The number of shifts is set by operand2.
Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position.	Algorithm: • Shift all bits right, the bit that goes off is set to CF. • Zero bit is inserted to the left-most position.
Example: MOV AL, 11100000b SHL AL, 1 ; AL = 11000000b, CF=1.	Example: MOV AL, 00000111b SHR AL, 1 ; AL = 00000011b, CF=1.

4) Branching: JMP, CALL, RET, LOOP
   i) JMP label
      Jumps to a designated address
   ii) CALL label
      Call a procedure (subprogram), save return address on stack
   iii) RET
      Returns from a procedure previously entered by a call
   iv) LOOP
      Loop through a sequence of instruction until CX=0

```
Label: …..
       …..
       …..
       Loop Label
```

5) Stack: PUSH, POP
   - Like in 8085
   - E.g. PUSH BX
   - POP BX

INT 21h functions

- Provided in the photocopy
- DOS services
- 01h, 02h, 09h, 0Ah, 4Ch

Function number	**Description**
01h e.g. mov ah,01h int 21h	**Keyboard input with echo:** This operation accepts a character from the keyboard buffer. If none is present, waits for keyboard entry. It returns the character in AL.
02h e.g. mov ah,02h int 21h	**Display character:** Send the character in DL to the standard output device console.
09h e.g. mov ah,09h int 21h	**String output:** Send a string of characters to the standard output. DX contains the offset address of string. The string must be terminated with a '$' sign.
0Ah	**String input**
4Ch e.g. mov ax,4C00h int 21h	**Terminate the current program** (mov ah,4Ch int 21h is also used.)

INT 10h functions

- Provided in the photocopy
- Video display services (BIOS services)
- 00h, 01h, 02h, 06h, 07h, 08h, 09h, 0Ah

Function number	**Description**
00h	Set video mode
01h	Set cursor size
02h	Set cursor position
06h	Scroll window up
07h	Scroll window down
08h	Read character and attribute of cursor
09h	Display character and attribute at cursor
0Ah	Display character at cursor

Simple sequence programs, Branching, Looping

- Discussed in the class and lab.