Introduction to MASM Programming

Machine-Level and Systems Programming

Dr. Rahul Raman

Mr Sukesh Babu V S

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, DESIGN AND MANUFACTURING KANCHEEPRAM

Types of Codes:

- Machine codes: Processor can understands only 0 and 1, ie, binary values, since processor is purely an electronic device.
  - Instructions and data's are stored in memory as bytes.
  - Each instruction has two parts: numeric code and operands.

what operation to be performed → **OPCODE** | **OPERAND** ← *on which data the operation to be performed.*

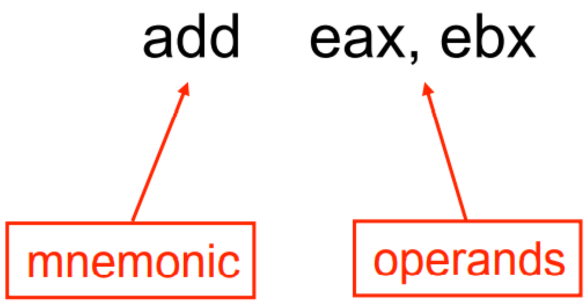
Add A, B

General format of an instruction

Example:

- On x86 there is an instruction to add the content of EAX to the content of EBX and to store the result back into EAX.
- This instruction is encoded (in hex) as: 03C3.

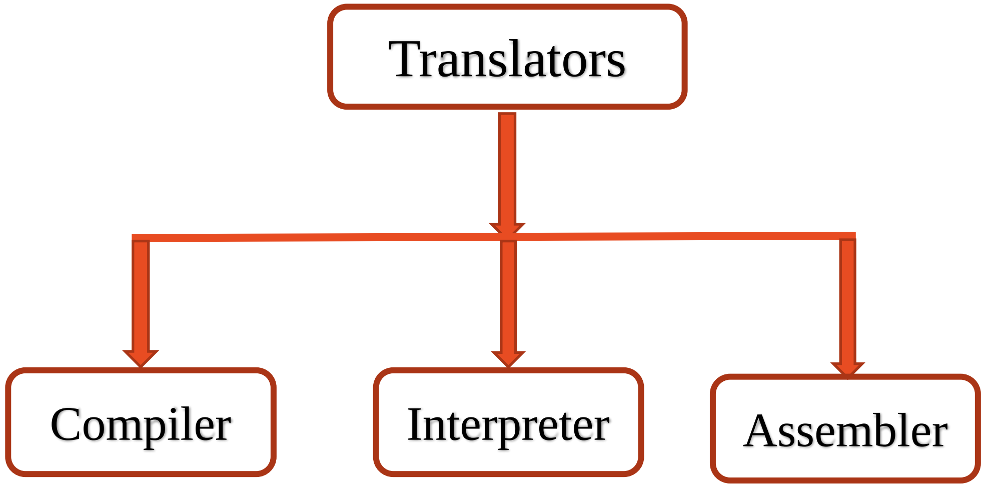
- Assembly codes:
  - Each assembly instruction corresponds to exactly one machine instruction.
  - The instruction (EAX = EAX + EBX) is written simply as:

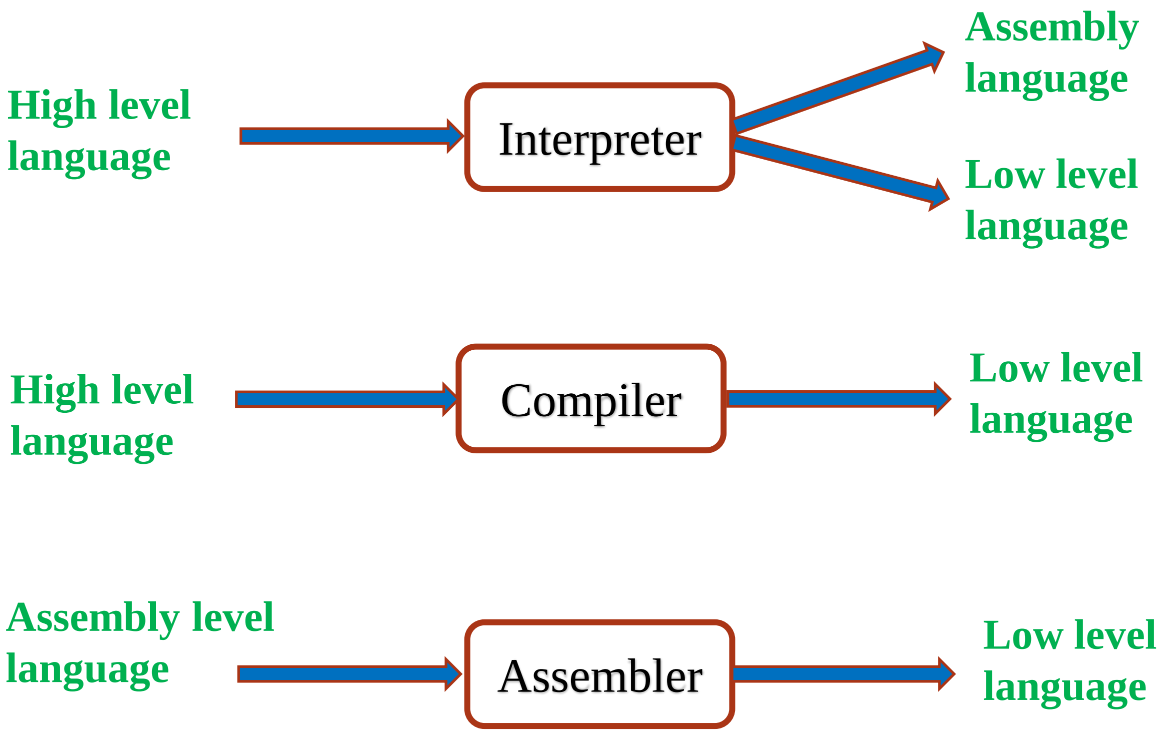


  - Assembler :An assembler translates assembly code into machine code.
  - Assembly code is NOT portable across architectures.
  - For different ISAs, different assembly languages are used.
- High Level Codes: High level programming language like C, C++, java etc are used.(example : a=b+c;)

Language translators:

- Converts programing source code into language that the computer processor can understand





Types of assembler

- **MASAM** (Microsoft Macro Assembler): DOS/Windows-based, produces 16-bit/32-bit/64-bit output
- **TASM** (Borland Turbo Assembler): DOS/Windows-based, produces 16-bit/32-bit output
- **NASM** (Netwide Assembler): produces 16-bit/32-bit/64-bit output
- All of those assemblers take the x86 instruction set as input

Assembly Memory Segments:

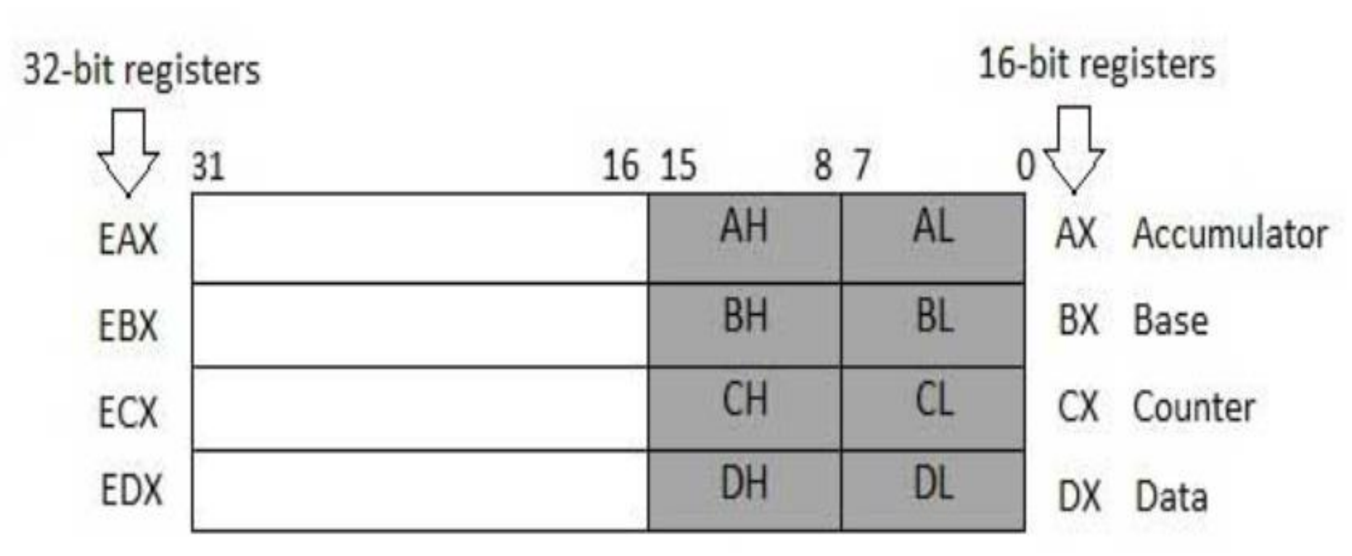
- **Memory Segments** : A segmented memory model divides the system memory into groups of independent segments, referenced by pointers located in the segment registers.
- Each segment is used to contain a specific type of data.
- **Data segment**
  - to declare the memory region where data elements are stored for the program.
- **Code segment**
  - This defines an area in memory that stores the instruction codes. This is also a fixed area.
- **Stack segment** - this segment contains data values passed to functions and procedures within the program.

Assembly Registers:

- To speed up the processor operations, the processor includes some internal memory storage locations, called registers.
- The registers stores data elements for processing without having to access the memory.
- A limited number of registers are built into the processor chip.
- **Processor Registers**
  - General registers
    - Data registers
    - Pointer registers
    - Index registers
  - Control registers
  - Segment register

Data Registers

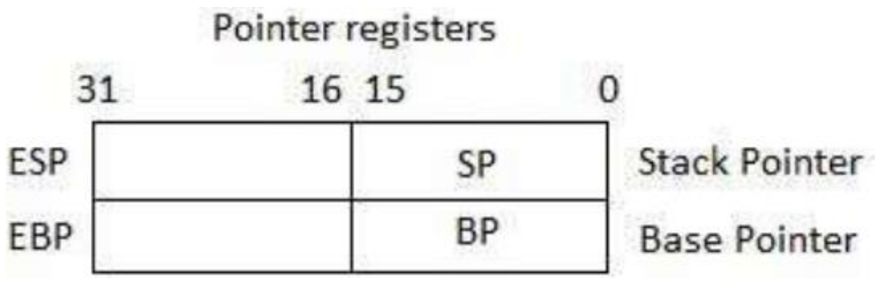
- Four 32-bit data registers are used for arithmetic, logical and other operations.



- These 32-bit registers can be used in three ways:
  - As complete 32-bit data registers: EAX, EBX, ECX, EDX.
  - Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
  - Lower and higher halves of the four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL.

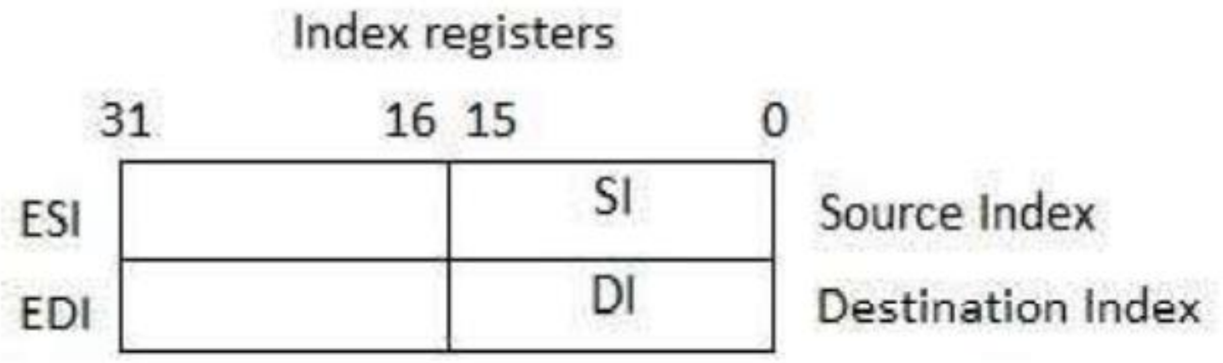
Pointer Registers:

- **Instruction Pointer (IP)** - the 16-bit IP register stores the offset address of the next instruction to be executed.
- **Stack Pointer (SP)** - the 16-bit SP register provides the offset value within the program stack.
- **Base Pointer (BP)** - the 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine.



Index Registers:

- **Source Index (SI)** - it is used as source index for string operations.
- **Destination Index (DI)** - it is used as destination index for string operations.



- **Processor Registers**
  - General registers
    - Data registers , Pointer registers, Index registers
  - Control registers
  - Segment register

Control Registers:

- The 32-bit instruction pointer register and 32-bit flags register combined are considered as the control registers.
- The common flag bits are:
  - Overflow Flag (OF): indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
  - Direction Flag (DF): determines left or right direction for moving or comparing string data.
  - When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.

- Interrupt Flag (IF): determines whether the external interrupts like, keyboard entry etc. are to be ignored or processed.
  - It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
- Sign Flag (SF): shows the sign of the result of an arithmetic operation.
  - A positive result clears the value of SF to 0 and negative result sets it to 1.
- Zero Flag (ZF): indicates the result of an arithmetic or comparison operation.
  - A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
- Auxiliary Carry Flag (AF): contains the carry from bit 3 to bit 4 following an arithmetic operation.
- Parity Flag (PF): indicates the total number of 1-bits in the result obtained from an arithmetic operation.
- Carry Flag (CF): contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation

Segment Registers:

- Segments are specific areas defined in a program for containing data, code and stack.
- There are three main segments:
  - Code Segment: it contains all the instructions to be executed.
    - A 16 - bit Code Segment register or CS register stores the starting address of the code segment.
  - Data Segment: it contains data, constants and work areas.
    - A 16 - bit Data Segment register of DS register stores the starting address of the data segment.
  - Stack Segment: it contains data and return addresses of procedures or subroutines.
    - It is implemented as a 'stack' data structure.
    - The Stack Segment register or SS register stores the starting address of the stack

MASM Program Structure:

```
.8086
.model small
.stack 100h
.data
        msg db "sample string $"
.code
        mov ax, @data
        …
        …
        …
end
```

- **.8086** limits the assembler to 8086 instruction set.
- While using model small, the program should contain only one data and code segment
- .stack 100h:This specifies the stack size for dynamic memory allocation needed by the program
- .data: data segment

  msg db "sample string$" is used to create a byte and assign it the string value
- Code: code segment, contains the instructions
- End: end of the program

Comments:

- With MASM, comments are added after a ‘;’
- Example: add eax, ebx ; y = y + b

Variables

- There are various define directives to allocate space for variables for both initialized and uninitialized data.
  1. To allocate storage space to Initialized data:

     Syntax:

     variable-name define-directive initial-value

  Example:

  msg db “sample string$”

2. To allocate storage space to un-initialized data:

Define Directive	Description	Allocated Space
DB	Define Byte	1 byte
DW	Define Word	2 bytes
DD	Define Doubleword	4 bytes
DQ	Define Quadword	8 bytes
DT	Define Ten Bytes	10 bytes

Define Directive	Description
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Sample program in assembly

```
.model small
.stack 100h
.data
      msg db   "Hello world$"     ; defining string
.code
      mov ax, @data   ;initialize the data segment
      mov ds, ax

      mov ah, 09h     ;printing the message
      lea dx, msg
      int 21 h

      mov ax, 4c00h  ; exit from the program
      int 21h
end
```

.data **msg db "hello world$"**	• Reserve a byte of memory and initialize it with the string specified and give it a label 'msg'. • All memory initializations should be done in the **.data** segment. • Any reference to a memory initialization in the code segment should be declared in the data segment to avoid errors.
.code **mov ax, @data** **mov ds, ax**	• To initialize the data segment. • This should be the first lines in the code segment to use the data segment, without this we cannot access the data segment.

Sample program in assembly

```
.model small
.stack 100h
.data
    msg db  "Hello world$"     ; defining string
.code
    mov ax, @data   ;initialize the data segment
    mov ds, ax

    mov ah, 09h     ;printing the message
    lea dx, msg
    int 21 h

    mov ax, 4c00h  ; exit from the program
    int 21h
end
```

load effective address

mov ax, 09h **lea dx, msg** **int 21h**	• To print the string into standard output. • **09h** is the DOS interrupt code to write a string to STDOUT. • Store this interrupt value in the AX using **mov** instruction. • Then load string which is labelled 'msg' into DX using **lea.** • Call the ISRusing **int** instruction, which will execute the interrupt code stored in accumulator, which is **09h**.
mov ax, 4c00h **int 21h**	• To safely exit from the program. • **4c00h** is the interrupt code to exit from a program.

interrupt

Print two message

```
.model small
.stack 100h
.data
    msg1 db     "Good morning $"     ; defining string
    msg2 db     "Welcome to IITDM$"    ; defining string
.code
    mov ax, @data   ;initialize the data segment
    mov ds, ax
    mov ah, 09h     ;printing the message
    lea dx, msg1
    int 21 h

    mov ah, 09h     ;printing the message
    lea dx, msg2
    int 21 h
    mov ax, 4c00h  ; exit from the program
    int 21h
end
```

Output

Good morningWelcome to IIITDM

```
.model small
.stack 100h
.data
      msg1 db       "Good morning $"     ;
      defining string
      msg2 db        "Welcome to IITDM$"     ;
      defining string
.code
      mov ax, @data   ;initialize the data segment
      mov ds, ax
      mov ah, 09h     ;printing the message
      lea dx, msg1
      int 21 h

      mov dl,10    ;ascii 10-new line
      mov ah,02h
      int 21h
```

```
      mov ah, 09h     ;printing the message
      lea dx, msg2
      int 21 h
      mov ax, 4c00h  ; exit from the program
      int 21h
  end
```

Output

Good morning
Welcome to IIITDM

Program to add two numbers:

```
.model small
.stack 100h
.data
        msg1 db "The sum is$"
.code
        mov ax, @data
        mov ds,ax
        mov al,03h
        mov bl,04h
        add al, bl
             add al,30h ; convert to ascii
             mov cl, al ; mov result to cl
```

```
        mov ah, 09h
         lea dx, msg1
         int 21h

         mov dl,cl     ; move the result to dl
         mov ah, 02h
         int 21h

         mov ax, 4c00h
         int 21h
end
```

Output

The sum is7

```
.model small
.stack 100h
.data
        msg1 db "The sum is$"
        n1 db 05h
        n2 db 04h
.code
        mov ax, @data
        mov ds,ax
        mov al,n1
        mov bl,n2
        add al, bl
            add al,30h ; convert to ascii
            mov cl,al
```

```
mov ah, 09h
lea dx, msg1  ; for printing the result or
int 21h   ;message  should be in dx or dl
        mov dl,10
        mov ah, 02h
        int 21h
        mov dl,cl     ; move the result to dl
        mov ah, 02h
        int 21h
        mov ax, 4c00h
        int 21h
end
```

Output

The sum is

7

Assembly Numbers:

- Numerical data is generally represented in decimal(internally binary) system.
- Arithmetic instructions operate on binary data.
- When numbers are displayed on screen or entered from keyboard, they are in ASCII form.
- So this input data in ASCII form should be converted to binary for arithmetic calculations and converted the result back to binary.

```
mov al, '3'  ; 3 is in asci form
sub al, 30h  ; convert it into binary
mov bl, 05h   ; 05h is in hexa
add al, bl
add al, 30h ; result is in binary, so
            ; convert it into ascii for display
```

```
mov al, '3'  ; 3 is in asci form
sub al, 30h  ; convert it into binary
mov bl, '5'   ; convert it to binary
sub bl, 30h
add al, bl
add al, 30h ; result is in binary, so
            ; convert it into ascii for display
```

```
.model small
.stack 100h
.data
        msg1 db "The sum is$"
.code
        mov ax, @data
        mov ds,ax
        mov al,'3' ;3 in ascii form
        sub al, 30h ; convert to binary
        mov bl,5h
        add al, bl
        add al,30h ; convert to ascii
        mov cl, al
```

```
mov ah, 09h
lea dx, msg1
int 21h
    mov dl,10
    mov ah, 02h
    int 21h
mov dl,cl    ; move the result to dl
      mov ah, 02h
      int 21h

      mov ax, 4c00h
      int 21h
end
```

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

add/sub 38h(hexa)
or
add/sub 48(decimal)
or
add/sub 60(octal)
or
Add/sub ‘0’(char)

Program to add two numbers from keyboard:

```
.model small
.stack 100h
.data
        msg1 db "The sum is$"
.code
        mov ax, @data
        mov ds,ax

        mov ah,1  ; for reading from keyboard
        int 21h
        sub al, 30h  ; convert to binary
        mov bl,al
        mov ah,1
        int 21h
        sub al, 30h
        add al, bl
```

```
add al,30h
mov cl,al
        mov ah, 09h
        lea dx, msg1  ; for printing the result
        int 21h       ; should be in dx or dl
        mov dl,10
        mov ah, 02h  ; for new line
        int 21h

        mov dl,cl    ; move the result to dl
        mov ah, 02h  ; result printing
        int 21h
       mov ax, 4c00h
        int 21h
end
```

Assembly system calls:

Print the message

```
mov ah, 09h     ; load the system call number 9 to ah
lea dx, msg     ; load the EA of message in to dx
int 21 h        ;  call the interrupt
```

System read call

```
mov ah, 01h     ; load the system call number 1 to ah
int 21 h        ;  call the interrupt
                ; the entered value is in accumulator
```

System write call

```
mov ah, 02h     ; load the system call number 1 to ah
int 21 h        ;  call the interrupt
                ; the value in accumulator is displayed
```

Exit system call

```
mov ax, 4c00h
int 21h
```

Addressing Modes:

- The opcode field of an instruction specifies the operation to be performed.
- This operation must be executed on some data stored.
- The way the operands are chosen during program execution depends on addressing modes.
- The different ways in which the location of an operand is specified in an instruction is called as Addressing mode.

- **Implied mode**
- **Immediate mode**
- **Register mode**
- **Register indirect mode**
- **Auto increment/decrement mode**
- **Direct addressing mode**
- **Indirect addressing mode**
- **Relative addressing mode**
- **Indexed addressing mode**
- **Base register addressing mode.**

- The three basic modes of addressing are:
  1. Register addressing
  2. Immediate addressing
  3. Memory addressing

```
mov ax, bx
add al, bl
```

Register addressing

```
mov al, 03h
add al, 04h
```

Immediate addressing

```
mov al, 100
add al, n1
```

Direct addressing

```
mov al, [100]
```

Indirect addressing