

Introduction to NASM

A Study Material for
CS2093 - Hardware Laboratory

Prepared By:
Muhammed Yazar Y
BTech, CSE, NIT Calicut

Under The Guidance of:
Mr. Saidalavi Kalady
Assistant Professor
CSE, NIT Calicut



Department Of Computer Science & Engineering
National Institute of Technology, Calicut

2012

Thanks to

Saidalavi Sir,
Jayaraj Sir,
Meera Sudharman

Without their support there won't be even a single
page in this.....

-yazar

Basics of Computer Organization

In order to program in assembly language it is necessary to have basic knowledge of Computer organization and processor architecture. This chapter gives you the basic idea of how a processor works, how it access data from main memory, how it reads or writes information from other I/O (Input / Output) devices etc.

The basic operational design of a computer is called architecture. 80X86 series of processors follow Von Newmann Architecture which is based on the stored program concept.

1. Processor:

Processor is the brain of the computer. It performs all mathematical, logical and control operations of a computer. It is that component of the computer which executes all the instructions given to it in the form of programs. It interacts with I/O devices, memory (RAM) and secondary storage devices and thus implements the instructions given by the user.

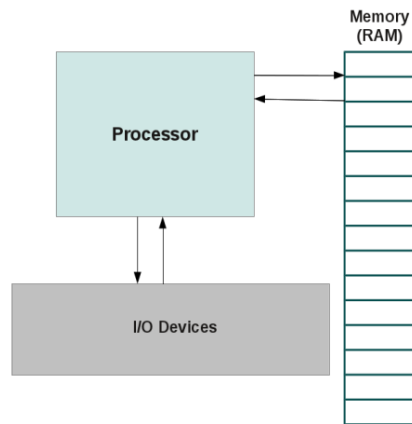


Fig: 1.1 Basic Operation of a processor

2. Registers

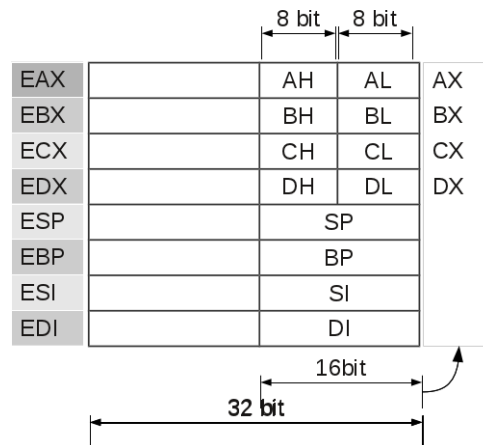
Registers are the most immediately accessible memory units for the processor. They are the fastest among all the types of memory. They reside inside the processor and the processor can access the contents of any register in a single clock cycle. It is the working memory for a processor, i.e , if we want the processor to perform any task it needs the data to be present in any of its registers.

The series of processors released on or after 80186 like 80186, 80286, 80386, Pentium etc are referred to as x86 or 80x86 processors. The processors released on or after 80386 are called I386 processors. They are 32 bit processors internally and externally. So their register sizes are generally 32 bit. In this section we will go through the I386 registers.

Intel maintains its backward compatibility of instruction sets, i.e, we can run a program designed for an old 16 bit machine in a 32bit machine. That is the reason why we can install 32-bit OS in a 64 bit PC. The only problem is that, the program will not use the complete set of registers and other available resources and thus it will be less efficient.

i. General Purpose Registers

There are eight general purpose registers. They are EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP. We can refer to the lower 8 and 16 bits of these registers (E.g.: AX, AH, AL) separately. This is to maintain the backward compatibility of instruction sets. These registers are also known as scratchpad area as they are used by the processor to store intermediate values in a calculation and also for storing address locations.

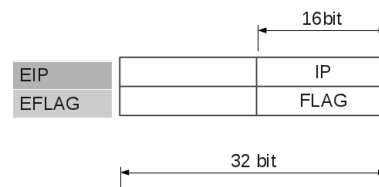


I-386 General Purpose Registers

The General Purpose Registers are used for :

- ⤴ EAX: Accumulator Register – Contains the value of some operands in some operations (E.g.: multiplication).
- ⤴ EBX: Base Register – Pointer to some data in Data Segment.
- ⤴ ECX: Counter Register – Acts as loop counter, used in string operations etc.
- ⤴ EDX: Used as pointer to I/O ports.
- ⤴ ESI: Source Index – Acts as source pointer in string operations. It can also act as a pointer in Data Segment (DS).
- ⤴ EDI: Destination Index- Acts as destination pointer in string operations. It can also act as a pointer in Extra Segment (ES)
- ⤴ ESP: Stack Pointer – Always points to the top of system stack.
- ⤴ EBP: Base Pointer – It points to the starting of system stack (ie. bottom/base of stack).

ii. Flags and EIP



Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	0	0	0	0	0	0	0	0	0	0	I D	V IP	V IF	A C	V M	R F	1	N T	IOPL		O F	D F	IF	T F	S F	Z F	0	A F	0	P F	1	C F

Intel x86 FLAGS register			
Bit #	Abbreviation	Description	Category
FLAGS			
0	CF	Carry flag	S
1	1	Reserved	
2	PF	Parity flag	S
3	0	Reserved	
4	AF	Adjust flag	S
5	0	Reserved	
6	ZF	Zero flag	S
7	SF	Sign flag	S
8	TF	Trap flag (single step)	X
9	IF	Interrupt enable flag	C
10	DF	Direction flag	C
11	OF	Overflow flag	S
12, 13	1,1 / IOPL	I/O privilege level (286+ only) always 1 on 8086 and 186	X
14	1 / NT	Nested task flag (286+ only) always 1 on 8086 and 186	X
15	1 on 8086 and 186, should be 0 above	Reserved	
EFLAGS			
16	RF	Resume flag (386+ only)	X
17	VM	Virtual 8086 mode flag (386+ only)	X
18	AC	Alignment check (486SX+ only)	X
19	VIF	Virtual interrupt flag (Pentium+)	X
20	VIP	Virtual interrupt pending (Pentium+)	X
21	ID	Able to use CPUID instruction (Pentium+)	X
22	0	Reserved	
23	0	Reserved	
24	0	Reserved	
25	0	Reserved	
26	0	Reserved	
27	0	Reserved	
28	0	Reserved	
29	0	Reserved	
30	0	Reserved	
31	0	Reserved	
RFLAGS			
32-63	0	Reserved	

Source : www.wikipedia.org

X – System Flags, S – Status Flags, C – Control Flags

FLAGS are special purpose registers inside the CPU that contains the status of CPU / the status of last operation executed by the CPU.

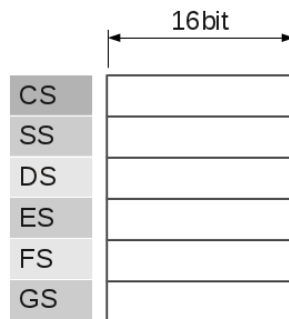
Some of the bits in FLAGS need special mention:

- ⤴ Carry Flag: When a processor do a calculation, if there is a carry then the Carry Flag will be set to 1.
- ⤴ Zero Flag: If the result of the last operation was zero, Zero Flag will be set to 1, else it will be zero.
- ⤴ Sign Flag : If the result of the last signed operation is negative then the Sign Flag is set to 1, else it will be zero.
- ⤴ Parity Flag: If there are odd number of ones in the result of the last operation, parity flag will be set to 1.
- ⤴ Interrupt Flag: If interrupt flag is set to 1, then only it will listen to external interrupts.

EIP:

EIP is the instruction pointer, it points to the next instruction to be executed. In memory there are basically two classes of things stored: (a) Data. (b)Program. When we start a program, it will be copied into the main memory and EIP is the pointer which points to the starting of this program in memory and execute each instruction sequentially. Branch statements like JMP, RET, CALL, JNZ (we will see this in chapter 3) alter the value of EIP.

iii. Segment Registers



In x86 processors for accessing the memory basically there are two types of registers used – Segment Register and Offset. Segment register contains the base address of a particular data section and Offset will contain how many bytes should be displaced from the segment register to access the particular data. CS contains the base address of Code Segment and EIP is the offset. It keeps on updating while executing each instruction. SS or Stack Segment contains the address of top most part of system stack. ESP and EBP will be the offset for that. Stack is a data structure that follows LIFO ie. Last-In-First-Out. There are two main operations associated with stack: push and pop. If we need to insert an element into a stack, we will push it and when we give the pop instruction, we will get the last value which we have pushed. Stack grows downward. So SP will always points to the top of stack and if we push an element, ESP (Stack Pointer) will get reduced by sufficient number of bytes and the data to be stored will be pushed over there. DS, ES, FS and GS acts as base registers for a lot of data operations like array addressing, string operations etc. ESI, EDI and EBX can act as offsets for them.

Unlike other registers, Segment registers are still 16 bit wide in 32-bit processors.

In modern 32 bit processor the segment address will be just an entry into a descriptor table in memory and using the offset it will get the exact memory locations through some manipulations. This is called segmentation.

Here in the memory ,the Stack Segment starts from the memory location 122 and grows downwards. Here the Stack Pointer ESP is pointing to the location 119. Now if we pop 1 byte of data from stack, we will get $(01010101)_2$ and the ESP will get increased by one

8 bit/1 byte	
Add ress	Memory
123	10101010
122	10111010
121	11001100
120	11111111
119	01010101
118	
117	
116	
115	
114	
113	
112	
111	
110	

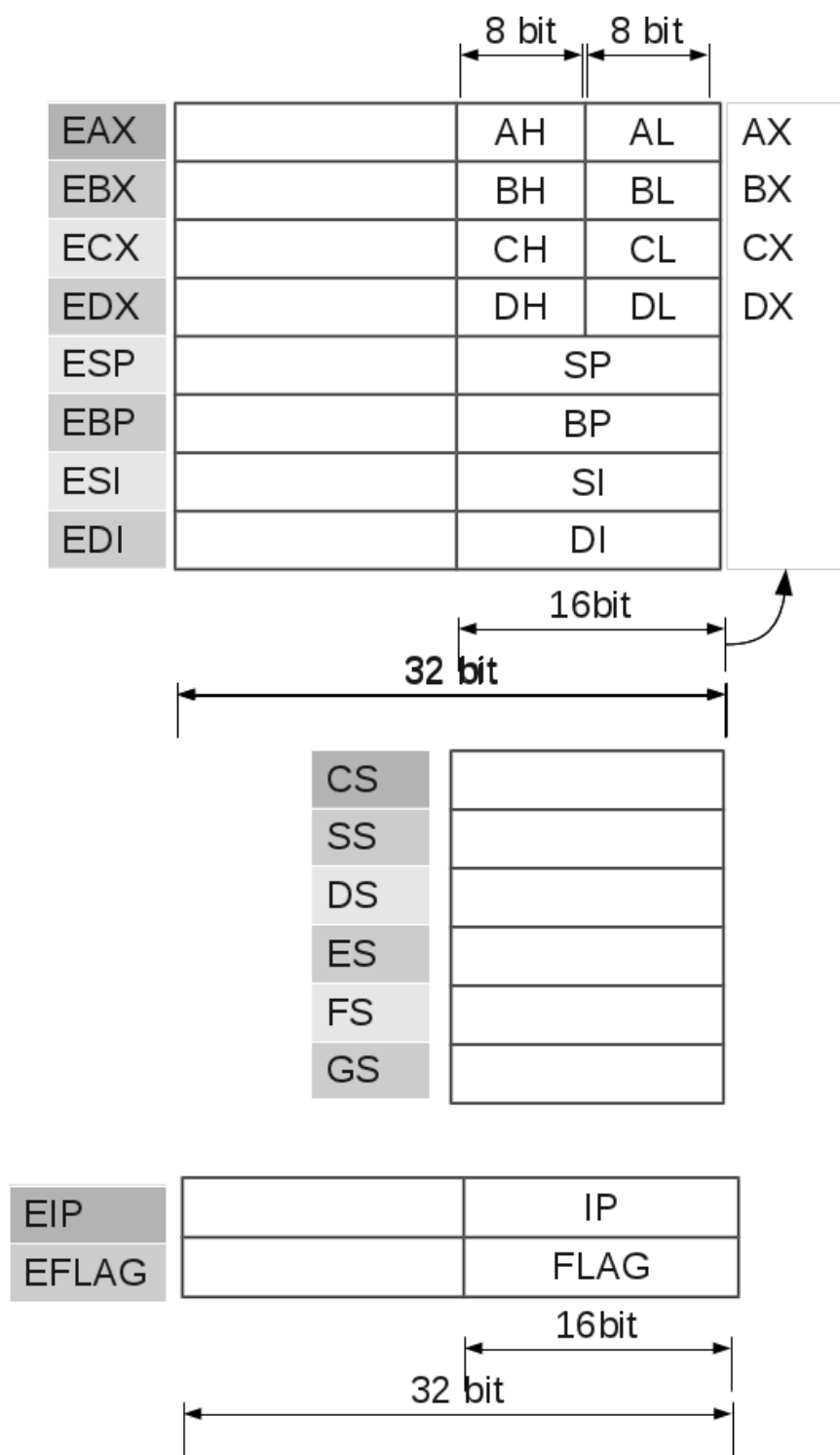
Now suppose we have $(01101100\ 11001111)_2$ in the register ax and we execute the push command:

push ax

Then the ESP will get reduced by two units (because we need two store two bytes of data) and ax will be copied over there:

8 bit/1 byte	
Add ress	Memory
123	10101010
122	10111010
121	11001100
120	11111111
119	01010101
118	01101100
117	11001111
116	
115	
114	
113	
112	
111	
110	

Here we can see that now ESP points to the lower byte of data from the 2 bytes data which we have pushed. In x86 architecture when we push or save some data in memory, the lower bytes of it will be addressed immediately and thus it is said to follow Little Endian Form. MIPS architecture follows Big Endian Form.



2. Bus

Bus is a name given to any communication medium, that transfers data between two components. We can classify the buses associated with the processor into three:

i. Data Bus

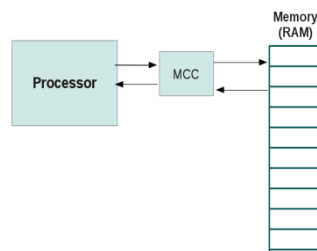
It is the bus used to transfer data between the processor and memory or any other I/O devices (for both reading and writing). As the size of data bus increases, it can transfer more data in a single stretch. The size of data bus in common processors by Intel are given below:

Processor	Bus size
8088, 80188	8 bit
8086, 80816, 80286, 80386SX	16 bit
80386DX, 80486	32 bit
80586, Pentium Pro and later processors	64 bit

ii. Address Bus :

RAM or random access memory is the working memory for a computer. It is a primary storage device, i.e. the data stored in RAM will be erased when the power is gone. All the instructions and data used by the computer is copied to RAM from other secondary storage devices like Hard Disk, Floppy drive, DVD drive etc. When we power on the PC the booting process will happen, which is the process of copying / loading Operating System from a secondary storage device to the main memory or RAM. When we start a program, the associated instructions and data are copied into RAM. Later on when processor needs the data from RAM it will copy that into its registers. RAM consists of contiguous storage units each of which is 8 bit wide (1 byte) and each location in RAM is given a location number starting from 0.

When processor wants to read some data from memory (RAM) or write data into some locations in the memory, it will first place the location in RAM from where the data is to be copied / written in the address bus. If it is a reading process, the data bus will have the data from that memory location. If it is a writing process, processor will place the data to be written into the data bus. MCC or Memory Control Circuitry or Memory Control Unit (MCU) is the set of electronic circuits present in the motherboard which helps the processor in reading or writing the data to or from a location in the RAM.



Address bus is also used for addressing the I/O devices as they can also be viewed as a

source or destination of data. For these operations control buses help a lot.

The maximum size of RAM which can be used in a PC is determined by the size of the address bus. If the size of address bus is n bits, it can address a maximum of 2^n bytes of RAM. This is the reason why even if we add more than 4 G.B of RAM in a 32 bit PC, we cannot find more than 4G.B of available memory. There are also others factors which limit the maximum usable memory size (e.g. some locations are used for I/O addressing).

Processors	Address Bus Width	Maximum addressable RAM size
8088, 8086, 80186, 80286, 80188	20	16 KB
80386SX, 80286	24	16 MB
80486, 80386DX, Pentium, Pentium Overdrive	32	4 GB
Pentium II, Pentium Pro	36	64 GB

iii. Control Bus :

Control bus contains information which controls the operations of processor or any other memory or I/O device. For example, the data bus is used for both reading and writing purpose and how is it that the Memory or MCC knows the data has to be written to a location or has to be read from a location, when it encounters an address in the address bus? This ambiguity is being cleared using read and write bits in the control bus. When the read bit is enabled, the data in the EDB (external data bus or simply data bus) will be written to that location. When the write bit is enabled, MCC will write the data in the EDB into the address location in the address bus.

There are also control bits for interrupts, clocking etc. We will see what an interrupt and clocking is in the coming pages.

3. System Clock

The basic component of a microprocessor or a microcontroller is logic gates. They are examples of digital circuits (i.e. they work in two voltage levels - high and low or 1 and 0). More specifically they are synchronous sequential circuits, i.e, their activities are synchronized with a common clock, which will create square wave pulses at a constant rate. In the control bus, there is a bit for clocking. The processing cycle is divided into four: Fetch, Decode, Execute, Write.

Fetch: At first the processor will fetch the data and instructions to be manipulated from the EDB (External Data Bus)

Decode: It will then decode the binary instruction and decide what action(arithmetic / logic operation) to be done on the operands.

Execute: It will perform the operations on the operands and manipulates the result.

Write: Then it will put the result into the EDB and it will go to the destination depending upon the values in address bus.

How come the processor knows when to do each of these operations? How can the processor conclude that now the data to be fetched is there in the EDB....? All these issues are solved by clocking. During the first clock pulse the processor knows that it is time to take the data from the EDB. During the next cycle it will decode the binary instruction. In the succeeding cycle it will execute the operations on operands and then it will write the result into the EDB. So a single operation takes minimum of four clock pulses to complete. Some operation (especially decode) will take more than one clock cycle to complete. So we can say that, all the activities of a processor are synchronized by a common clocking.

Speed of a processor depends much on its clock speed. It is the maximum speed at which we can make a processor work. But motherboard will always use the processor at a speed lower than or equal to its clock speed and that speed is decided by the system crystals and it is known as System Bus Speed. In earlier days we need to have some jumper settings in motherboard to adjust the system bus speed to clock speed. Modern motherboards will communicate with the processor, get the clock speed and then automatically adjust the System Bus Speed. Nowadays some people use third party software and make the motherboard to use processor at a higher speed than its specified clock speed. This is known as overclocking. Clock speed of a processor is the highest clock rate at which it will work safely, without any problems. So if we overclock the processor, still it may work but there is a very high risk for the processor to burn off due to heat dissipation. So never go for overclocking.

4. Interrupts:

Interrupts are the most critical routines executed by a processor. Interrupts may be triggered by external sources or due to internal operations. In linux based systems 80h is the interrupt number for OS generated interrupts and in windows based systems it is 21h. The Operating System Interrupt is used for implementing systems calls.

Whenever an interrupt occurs, processor will stop its present work, preserve the values in registers into memory and then execute the ISR (Interrupt Service Routine) by referring to an interrupt vector table. ISR is the set of instructions to be executed when an interrupt occurs. By referring to the interrupt vector table, the processor can get which ISR it should execute for the given interrupt. After executing the ISR, processor will restore the registers to its previous state and continue the process that it was executing before. Almost all the I/O devices work by utilizing the interrupt requests.

Register encoding	Not modified for 8-bit operands				Low 8-bit	16-bit	32-bit	64-bit
	Not modified for 16-bit operands							
	Zero-extended for 32-bit operands							
0			AH†	AL	AX	EAX	RAX	
3			BH†	BL	BX	EBX	RBX	
1			CH†	CL	CX	ECX	RCX	
2			DH†	DL	DX	EDX	RDX	
6				SIL‡	SI	ESI	RSI	
7				DIL‡	DI	EDI	RDI	
5				BPL‡	BP	EBP	RBP	
4				SPL‡	SP	ESP	RSP	
8				R8B	R8W	R8D	R8	
9				R9B	R9W	R9D	R9	
10				R10B	R10W	R10D	R10	
11				R11B	R11W	R11D	R11	
12				R12B	R12W	R12D	R12	
13				R13B	R13W	R13D	R13	
14				R14B	R14W	R14D	R14	
15				R15B	R15W	R15D	R15	
	63	32 31	16 15	8 7	0			

† Not legal with REX prefix

‡ Requires REX prefix

Registers in 64-bit architecture(Source: <http://www.tortall.net/>)

How to Start

We can classify the programming languages into three categories :

- i. *Machine Language* : Machine language consists of instructions in the form of 0's and 1's. Every CPU has its own machine language. It is very difficult to write programs using the combination of 0's and 1's. So we rely upon either assembly language or high level language for writing programs.
- ii. *Assembly Language* : Assembly language, when compared with machine language is in more human readable form. The general syntax of an assembly language instruction is:

mnemonic operand(s)

Eg: `add eax, ebx`
 `mov al, ah`
 `inc byte[data1]`

Corresponding to each assembly language instruction, there will be a machine language instruction (ie. a hardware implementation). An assembler converts an assembly language code into machine language. We will be using the 'Netwide Assembler' (NASM). It is freely available on the internet. It works for both 32bit and 64bit PCs. It can be installed in Linux as well as Windows. Examples of other assemblers are Microsoft Assembler (MASM) and Borland Assembler(TASM).

- iii. *High Level Languages* : They are in more human readable forms when compared to assembly language and machine language. They resemble natural languages like English very much. Eg: C, C++, Java, Perl, Fortan etc. A compiler or an interpreter converts a high level program into machine language.

Installing NASM

NASM is freely available on internet. You can visit : www.nasm.us . It's documentation is also available there.

In order to install NASM in windows you can download it as an installation package from the site and install it easily.

In Ubuntu Linux you can give the command : `sudo apt-get install nasm` and in fedora you can use the command: `su -c 'yum install nasm'` in a terminal and easily install nasm.

Why Assembly Language ?

- ⤴ When you study assembly language, you will get a better idea of computer organization and how a program executes in a computer.
- ⤴ A program written in assembly language will be more efficient than the same program written in a high level language. The code size will also be smaller. So it is preferred to program using assembly language for embedded systems, where memory is a great constraint.
- ⤴ Some portions of Linux kernel and some system softwares are written in assembly language. In programming languages like C, C++ we can even embed assembly language instructions into it using functions like `asm()`;

First Program :

Now let us write our first program in NASM to print the message “Hello World”.

1. Go to the terminal.
2. If you are using gedit as your text editor type the following command:

gedit hello.asm &

This will create a file `hello.asm` in the present working directory and open it in background. ie. you can still use the terminal for running some other commands, without closing the gedit which we have opened. This happens since we put an `'&'` sign at the end of our command.

3. Now type the following program into it :

```
;Program to print Hello World
;Section where we write our program
section .text
    global _start:
_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, string
    mov edx, length
    int 80h

    ;System Call to exit
    mov eax, 1
    mov ebx, 0
    int 80h

;Section to store uninitialized variables
section .data
    string: db 'Hello World', 0Ah
    length: equ 13

section .bss
    var: resb 1
```

We will go through each portion of the above code in the next few chapters.

Sections in NASM:

A typical NASM Program contain different sections. They are mainly:

Section .text: This is the part of a NASM Program which contains the executable code. It is the place from where the execution starts in NASM program, analogous to the `main()` function in C-Programming.

section .bss : This is the part of program used to declare variables without initialization

section .data : This is the part of program used to declare and initialize the variables in the program.

Eg:

```
section .data
    var1:      db      10
    str1:      db      "Hello World!.."

section .bss
    var3:      resb  1
    var4:      resq  1
```

- **RESx** directive is used to reserve just space in memory for a variable without giving any initial values.
- **Dx** directive is used for declaring space in the memory for any variable and also providing the initial values at that moment.

x	Meaning	No: of Bytes
b	BYTE	1
w	WORD	2
d	DOUBLE WORD	4
q	QUAD WORD	8
t	TEN WORD	20

Eg:

```
var1:      resb  1  ;Reserves 1 byte for storing var1
var2:      dw    25 ;Reserve 1 word in memory for storing var2 and initial value
              ;of var2 = 25
```

NB:

- ‘;’ – Semicolon is used to give comments in NASM.
- 011101b – Represents a number $(011101)_2$ – Binary Number.
- 31h – Represents a number $(31)_{16}$ – Hexadecimal Number.
- 23415o – Represents a number $(23415)_8$ – Octal Number.

Decalring Multiple Elements Together(Arrays)

```
var:  db    10,5,8,9    ;Reserves 4 bytes in memory for var and stores the  
                        ;values 10, 5, 8, 9 respectively in that.
```

Strings:

```
string:      db    "Hello"  
string2:     db    "H", "e", "l", "l", "o"
```

Here both string and string2 are identical. They are 5 bytes long and stores the String "Hello". Each character in the string will be first converted to ASCII Code and that numeric value will be stored in each byte location.

TIMES:

TIMES – is used to create and initialize large arrays with a common initial value for all its elements.

Eg:

```
var:  times 100  db    1; Creates an array of 100 bytes and ach element will  
                        ; be initialized with the value 1
```

Dereferencing Mechanism in NASM:

In NASM if we have some operands in memory, we need the address of that memory location in some variables or in any of the general purpose registers. Let us assume that we have an address in the variable label. If we need to get the value at that address then we need to use the dereferencing operator '[] '.

Eg:

```
mov  eax,  [label];Value stored in the address location will be copied to eax  
mov  ebx,  label; The address location will be copied to ebx reg.
```

We need to do the type casting operations to instructions for the operands for which the assembler won't be able to predict the number of memory locations to be dereferenced to get the data(like INC , MOV etc). For other instructions (like ADD, SUB etc) it is not mandatory. The directives used for specifying the data type are: **BYTE, WORD, DWORD, QWORD, TWORD**.

Eg:

```
MOV  dword[ebx], 1  
INC  BYTE[label]  
ADD  eax,        dword[label]
```


X86 - Basic Instruction Set

In this chapter we will explore the syntax of basic instructions in NASM. We will see few examples of each instruction.

1. MOV – Move/Copy

Copy the content of one register/memory to another or change the value of a register / memory variable to an immediate value.

sy: *mov dest, src*

- src should be a register / memory operand
- Both src and dest cannot together be memory operands.

Eg:

```
mov    eax, ebx    ;Copy the content of ebx to eax
mov    ecx, 109    ; Changes the value of ecx to 109
mov    al, bl
mov    byte[var1], al;Copy the content of al reg to the variable var in memory
mov    word[var2], 200
mov    eax, dword[var3]
```

2. MOVZX – Move and Extend

Copy and extend a variable from a lower spaced memory / register location to a higher one

sy: *mov src, dest*

- size of dest should be \geq size of src
- src should be a register / memory operand
- Both src and dest cannot together be memory operands.
- Works only with signed numbers.

Eg:

```
movzx    eax, ah
movzx    cx, al
```

- For extending signed numbers we use instructions like CBW (Convert Byte to Word), CWD (Convert Word to Double).
- CBW extends the AL register to AX
- CWD extends the AX register to DX:AX register pair

3. ADD – Addition

sy: *add dest, src*
dest = dest + src;

Used to add the values of two register / memory variable and store the result in the first operand.

- src should be a register / memory operand
- Both src and dest cannot together be memory operands.
- Both the operands should have the same size.

Eg:

```
add    eax, ecx ; eax = eax + ecx
add    al, ah ; al = al + ah
add    ax, 5
add    edx, 31h
```

4. SUB - Subtraction

sy: *sub* *dest, src*
 dest = dest - src;

Used to subtract the values of two reg / memory var and store the result in the first operand.

- src should be a register / memory operand
- Both src and dest cannot together be memory operands.
- Both the operands should have the same size.

Eg:

```
sub    eax, ecx ; eax = eax - ecx
sub    al, ah ; al = al - ah
sub    ax, 5
sub    edx, 31h
```

5. INC – Increment operation

Used to increment the value of a reg / memory variable by 1

Eg:

```
INC    eax ; eax++
INC    byte[var]
INC    al
```

6. DEC – Decrement operation

Used to decrement the value of a reg / memory variable by 1

Eg:

```
DEC    eax ; eax--
DEC    byte[var]
DEC    al
```

7. MUL – Multiplication

Sy: *mul* *src*

Used to multiply the value of a reg / memory variable with the EAX / AX / AL reg. MUL works according to the following rules.

- If src is 1 byte then AX = AL * src
- If src is 1 word (2 bytes) then DX:AX = AX * src (ie. Upper 16 bits of the result will go to DX and the lower 16 bits will go to AX)
- If src is 2 words long(32 bit) then EDX:EAX = EAX * src (ie. Upper 32 bits of the result will go to EDX and the lower 32 bits will go to EAX)

8. IMUL – Multiplication of signed numbers

IMUL instruction works with the multiplication of signed numbers. can be used mainly in three different forms.

Sy:

- i. `imul src`
- ii. `imul dest, src`
- iii. `imul dest, src1, src2`

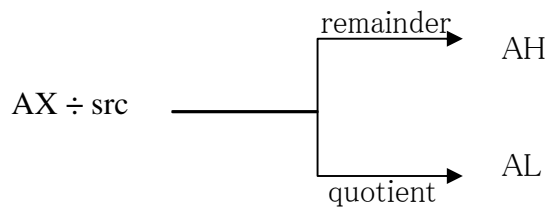
- If we use imul as in (i) then it's working follows the same rules of MUL
- If we use that in (ii) form then `dest = dest * src`
- If we use that in (iii) form then `dest = src1 * src2`

9. DIV – Division

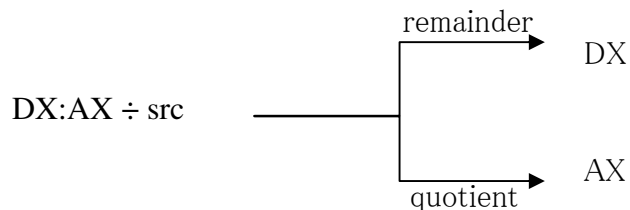
Sy: `div src`

Used to divide the value of EDX:EAX / DX:AX / AX reg with reg / memory variable with the. DIV works according to the following rules.

- If src is 1 byte then AX will be divide by src, remainder will go to AH and quotient will go to AL

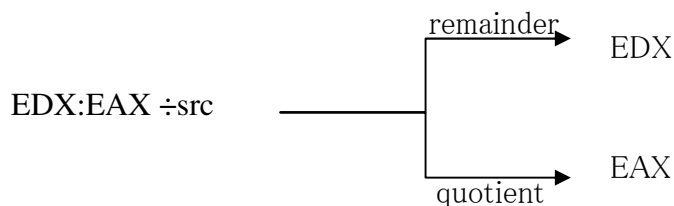


- If src is 1 word (2 bytes) then DX:AX will be divide by src, remainder will go to DX and quotient will go to AX



- If src is 2 words long(32 bit) then EDX:EAX will be divide by src, remainder will go to EDX and quotient will go to EAX

•



10. NEG – Negation of Signed numbers.

Sy: *NEG opl*

NEG Instruction negates a given reg / memory variable.

11. CLC - Clear Carry

This instruction clears the carry flag bit in CPU FLAGS.

12. ADC – Add with Carry

sy: *ADC dest, src*

ADC is used for the addition of large numbers. Suppose we want to add two 64 bit numbers. We keep the first number in EDX:EAX (ie. most significant 32 bits in EDX and the others in EAX) and the second number in EBX:ECX. Then we perform addition as follows

Eg:

```
clc ; Clearing the carry FLAG
add  eax, ecx ; Normal addition of eax with ecx
adc  edx, ebx ; Adding with carry for the higher bits.
```

13. SBB – Subtract with Borrow

sy: *SBB dest, src*

SBB is analogous to ADC and it is used for the subtraction of large numbers. Suppose we want to subtract two 64 bit numbers. We keep the first numbers in EDX:EAX and the second number in EBX:ECX. Then we perform subtraction as follows

Eg:

```
clc ; Clearing the carry FLAG
sub  eax, ecx ; Normal addition of eax with ecx
sbb  edx, ebx ; Adding with carry for the higher bits.
```

Branching In NASM:

14. JMP – Unconditionally Jump to label

JMP is similar to the goto label statements in C / C++. It is used to jump control to any part of our program without checking any conditions.

Eg:

```
label:
-----
-----
    JMP label
-----
-----
    JMP exit
-----
-----
exit:
```

15. CMP – Compares the Operands

sy: *CMP op1, op2*

When we apply CMP instruction over two operands say op1 and op2 it will perform the operation $op1 - op2$ and will not store the result. Instead it will affect the CPU FLAGS. It is similar to the SUB operation, without saving the result. For example if $op1 == op2$ then the Zero Flag(ZF) will be set to 1.

NB: For generating conditional jumps in NASM we will first perform the CMP operation between two reg / memory operands and then we use the following jump operations which checks the CPU FLAGS.

Conditional Jump Instructions:

Instruction	Working
JZ	Jump If Zero Flag is Set
JNZ	Jump If Zero Flag is Unset
JC	Jump If Carry Flag is Set
JNC	Jump If Carry Flag is Unset
JP	Jump If Parity Flag is Set
JNP	Jump If Parity Flag is Unset
JO	Jump If Overflow Flag is Set
JNO	Jump If Overflow Flag is Unset

Eg: Converting the given If statement into NASM Code

```
if( eax >= ebx)
    eax++;
else
    ebx++;

                                cmp    eax, ebx
                                JNC    if
                                INC    ebx      ;Else Part
                                JMP     L1
                                if:      ;If Part
                                INC     eax
                                L1:
                                -----
```

Advanced Conditional Jump Instructions:

In 80x86 processors Intel has added some enhanced versions of the conditional operations which are much more easier to use compared to traditional Jump instructions. They are easy to perform comparison between two variables.

First we need to use CMP op1, op2 before even using these set of Jump instructions. There is separate class for comparing the signed and unsigned numbers.

i) For Unsigned numbers:

Instruction	Working
JE	Jump if $op1 == op2$
JNE	Jump if $op1 \neq op2$
JA (Jump if above)	Jump if $op1 > op2$
JNA	Jump if $op1 \leq op2$
JB (Jump if below)	Jump if $op1 < op2$
JNB	Jump if $op1 \geq op2$

ii) For Signed numbers:

Instruction	Working
JE	Jump if op1 == op2
JNE	Jump if op1 ≠ op2
JG (Jump if greater)	Jump if op1 > op2
JNG	Jump if op1 ≤ op2
JL (Jump if lesser)	Jump if op1 < op2
JNL	Jump if op1 ≥ op2

Eg:

```
CMP    ax, bx           ;Comparing
JA     L1               ;Jumping to label
      INC    ax         ;Else Part
      JMP    L2
L1:    ;If Part
      inc    bx
L2:    -----
      -----
```

16. LOOP Instruction

Sy: loop label

When we use Loop instruction ecx register acts as the loop variable. Loop instruction first decrements the value of ecx and then check if the new value of ecx ≠ 0. If so it will jump to the label following that instruction. Else control jumps to the very next statement.

Eg:

```
mov     ecx, 10
mov     eax, 0

add:
      add    eax, ecx

loop    add
```

Converting Standard C/C++ Control Structures to NASM:

i.

```
//If-Else
if(eax > 5)
    eax = eax + ebx;
else
    ecx = ecx + ebx;

;NASM Statement
cmp    eax, 5    ; Comparing
ja     if
else:   ;Else Part
    add    ecx, ebx
    jmp L2
if:     ;If Part
    add    eax, ebx
L2:
```

ii.

```
//For Loop
eax = 0;
for(ebx = 1 to 10)
    eax = eax + ebx;

;NASM Code
mov    eax, 0
mov    ebx, 1
for:
    add    eax, ebx
    cmp    ebx, 10
    jbe    for
-----
-----
```

iii.

```
//While-Loop
sum = 0;
ecx = n;
while(ecx > 0)
    sum = sum + ecx;
;NASM Code
mov    dword[sum], 0
mov    ecx, dword[n]

add:
    add    [sum], ecx
loop add    ; Decrements ecx and checks if ecx != 0, if so it will jump to
add
```

Boolean Operators:

17. AND –Bitwise Logical AND

sy: *AND op1, op1*

Performs bitwise logical ANDing of op1 and op2 , assign the result to op1.

op1 = op1 & op2; //Equivalent C Statement

Let x = 1010 1001b and y= 10110 010b be two 8-bit binary numbers.

Then x & y

x	1	0	1	0	1	0	0	1
y	1	0	1	1	0	0	1	0
x AND y	1	0	1	0	0	0	0	0

18. OR – Bitwise Logical OR

sy: *OR op1, op1*

Performs bitwise logical ORing of op1 and op2 , assign the result to op1.

op1 = op1 | op2; //Equivalent C Statement

Let x = 1010 1001b and y= 10110 010b be two 8-bit binary numbers.

Then x | y

x	1	0	1	0	1	0	0	1
y	1	0	1	1	0	0	1	0
x OR y	1	0	1	1	1	0	1	1

19. XOR – Bitwise Logical Exclusive OR

sy: *XOR op1, op1*

Performs bitwise logical XORing of op1 and op2 , assign the result to op1.

op1 = op1 ^ op2; //Equivalent C Statement

Let x = 1010 1001b and y= 10110 010b be two 8-bit binary numbers.

Then x ^ y

x	1	0	1	0	1	0	0	1
y	1	0	1	1	0	0	1	0
x XOR y	0	0	0	1	1	0	1	1

20. NOT – Bitwise Logical Negation

sy: *NOT op1*

Performs bitwise logical NOT of op1 and assign the result to op1.

op1 = ~op1; //Equivalent C Statement

Let x = 1010 1001b and y= 10110 010b be two 8-bit binary numbers.

Then ~x

x	1	0	1	0	1	0	0	1
~x	0	1	0	1	0	1	1	0

21. TEST – Logical AND, affects only CPU FLAGS

sy: *TEST op1, op2*

- It performs the bitwise logical AND of op1 and op2 but it won't save the result to any registers. Instead the result of the operation will affect CPU FLAGS.
- It is similar to the CMP instruction in usage.

22. SHL – Shift Left

sy: *SHL op1, op2*

op1 = op1 << op2; //Equivalent C Statement

- SHL performs the bitwise left shift. op1 should be a reg / memory variable but op2 must be an immediate(constant) value.
- It will shift the bits of op1, op2 number of times towards the left and put the rightmost op2 number of bits to 0.

Eg:

shl eax, 5

al	1	0	1	0	1	0	0	1
al << 3	0	1	0	0	1	0	0	0

23. SHR – Right Shift

sy: *SHR op1, op2*

op1 = op1 >> op2; //Equivalent C Statement

- SHR performs the bitwise right shift. op1 should be a reg / memory variable but op2 must be an immediate(constant) value.
- It will shift the bits of op1, op2 number of times towards the right and put the leftmost op2 number of bits to 0.

Eg:

shr eax, 5

al	1	0	1	0	1	0	0	1
al >> 3	0	0	0	1	0	1	0	1

24. ROL – Rotate Left

sy: *ROL op1, op2*

- ROL performs the bitwise cyclic left shift. op1 could be a reg / memory variable but op2 must be an immediate(constant) value.

Eg:

rol eax, 5

al	1	0	1	0	1	0	0	1
rol al, 3	0	1	0	0	1	1	0	1

25. ROR – Rotate Right

sy: *ROR op1, op2*

- ROR performs the bitwise cyclic right shift. op1 could be a reg / memory variable but op2 must be an immediate(constant) value.

Eg:

```
ror    eax, 5
```

al	1	0	1	0	1	0	0	1
ror al, 3	0	0	1	1	0	1	0	1

26. RCL – Rotate Left with Carry

sy: *RCL op1, op2*

- Its working is same as that of rotate left except it will consider the carry bit as its left most extra bit and then perform the left rotation.

27. RCR – Rotate Right with Carry

sy: *RCR op1, op2*

- Its working is same as that of rotate right except it will consider the carry bit as its left most extra bit and then perform the right rotation.

Stack Operations

28. PUSH – Pushes a value into system stack

PUSH decreases the value of ESP and copies the value of a reg / constant into the system stack

sy: *PUSH reg/const*

Eg:

```
PUSH    ax      ;ESP = ESP - 2 and copies value of ax to [EBP]
PUSH    eax     ;ESP = ESP - 4 and copies value of ax to [EBP]
PUSH    ebx
PUSH    dword 5
PUSH    word  258
```

29. POP – Pop off a value from the system stack

POP Instruction takes the value stored in the top of system stack to a reg and then increases the value of ESP

Eg:

```
POP     bx      ; ESP= ESP + 2
POP     ebx     ; ESP= ESP + 4
POP     eax
```

30. PUSHA – Pushes the value of all general purpose registers

PUSHA is used to save the value of general purpose registers especially when calling some subprograms which will modify their values.

31. POPA – POP off the value of all general purpose registers which we have pushed before using PUSH instruction

32. PUSHF – Pushes all the CPU FLAGS

33. POPF – POP off and restore the values of all CPU Flags which have been pushed before using PUSHF instructions.

NB: It is important to pop off the values pushed into the stack properly. Even a minute mistake in any of the PUSH / POP instruction could make the program not working.

31. Pre-processor Directives in NASM

In NASM `%define` acts similar to the C's preprocessor directive `#define`. This can be used to declare constants.

Eg: `%define SIZE 100`

Basic I/O in NASM

The input from the standard input device (Keyboard) and Output to the standard output device (monitor) in a NASM Program is implemented using the Operating System's read and write system call. Interrupt no: 80h is given to the software generated interrupt in Linux Systems. Applications implement the System Calls using this interrupt. When an application triggers int 80h, then OS will understand that it is a request for a system call and it will refer the general purpose registers to find out and execute the exact Interrupt Service Routine (ie. System Call here). The standard convention to use the software 80h interrupt is, we will put the system call no: in eax register and other parameters needed to implement the system calls in the other general purpose registers. Then we will trigger the 80h interrupt using the instruction 'INT 80h'. Then OS will implement the system call.

1. Exit System Call

- System call number for exit is 1, so it is copied to eax reg.
- Output of a program if the exit is successful is 0 and it is being passed as a parameter for exit() system call. We need to copy 0 to ebx reg.
- Then we will trigger INT 80h

```
mov    eax, 1    ;System Call Number
mov    ebx, 0    ;Parameter
int     80h      ;Triggering OS Interrupt
```

2. Read System Call

- Using this we could read only string / character
- System Call Number for Read is 3. It is copied to eax.
- The standard Input device(keyboard) is having the reference number 0 and it must be copied to ebx reg.
- We need to copy the pointer in memory, to which we need to store the input string to ecx reg.
- We need to copy the number of characters in the string to edx reg.
- Then we will trigger INT 80h.
- We will get the string to the location which we copied to ecx reg.

```
mov    eax, 3    ;Sys_call number for read
mov    ebx, 0    ;Source Keyboard
mov    ecx, var   ;Pointer to memory location
mov    edx, dword[size] ;Size of the string
int     80h      ; Triggering OS Interrupt
```

- This method is also used for reading integers and it is a bit tricky. If we need to read a single digit, we will read it as a single character and then subtract 30h from it(ASCII of 0 = 30h). Then we will get the actual value of that number in that variable.

```
mov    eax, 3
mov    ebx, 0
mov    ecx, digit1
mov    edx, 1
int     80h
sub    byte[digit1], 30h ;Now we have the actual number in [var]
```

Reading a two digit number:

```
;Reading first digit
mov eax, 3
mov ebx, 0
mov ecx, digit1
mov edx, 1
int 80h

;Reading second digit
mov eax, 3
mov ebx, 0
mov ecx, digit2
mov edx, 2 ;Here we put 2 because we need to read and
           ;omit enter key press as well
int 80h

sub byte[digit1], 30h ;Getting the number from ASCII
sub byte[digit2], 30h

; num = (10* digit1) + digit2

mov al, byte[digit1] ; Copying first digit to al
mov bl, 10
mul bl                ; Multiplying al with 10
movzx bx, byte[digit2] ; Copying digit2 to bx
add ax, bx
mov byte[num], al ;We are sure that no less than 256, so we can omit
                  ;higher 8 bits of the result.
```

3. Write System Call

- Using this we could write only string / character
- System Call Number for Write is 4. It is copied to eax.
- The standard Output device(Monitor) is having the reference number 1 and it must be copied to ebx reg.
- We need to copy the pointer in memory, where the output string resides to ecx reg.
- We need to copy the number of characters in the string to edx reg.
- Then we will trigger INT 80h.

Eg:

```
mov eax, 4      ;Sys_call number
mov ebx, 1      ;Standard Output device
mov ecx, msg1   ;Pointer to output string
mov edx, size1  ;Number of characters
int 80h        ;Triggering interrupt.
```

- This method is even used to output numbers. If we have a number we will break that into digits. Then we keep each of that digit in a variable of size 1 byte. Then we add 30h (ASCII of 0) to each, doing so we will get the ASCII of character to be printed.

Sample Programs:

1. Hello World Program

```
section .text                ;Code Section
    global _start:

_start:
    mov eax, 4                ;Using int 80h to implement write() sys_call
    mov ebx, 1
    mov ecx, string
    mov edx, length
    int 80h

    ;Exit System Call
    mov eax, 1
    mov ebx, 0
    int 80h

section .data                ;For Storing Initialized Variables
    string: db 'Hello World', 0Ah    ;String Hello World followed by a new
line character
    length: equ 13                ; Length of the string stored to a constant.
```

NB: Using equ we declare constants, ie. their value won't change during execution.
\$-string will return the length of string variables in bytes (ie. number of characters)

2. Program to add two one two digit numbers

```
;Initialized variables
section .data
    message: db "Enter a two digit number : "
    message_length: equ $-message

;Un-initialized variables

section .bss
    digit1: resb 1
    digit2: resb 1
    digit3: resb 1
    digit4: resb 1
    num: resb 1
    num1: resb 1
    num2: resb 1
    sum: resw 1
```

```

section .text
    global _start

_start:

    ;Printing prompt message
    mov eax, 4
    mov ebx, 1
    mov ecx, message
    mov edx, message_length
    int 80h

    ;Reading first digit
    mov eax, 3
    mov ebx, 0
    mov ecx, digit1
    mov edx, 1
    int 80h

    ;Reading second digit
    mov eax, 3
    mov ebx, 0
    mov ecx, digit2
    mov edx, 2      ;Read and ignore an extra character as the system will read enter
                    ;press as well
    int 80h

    ;Calculating the number from digits
    sub byte[digit1], 30h
    sub byte[digit2], 30h

    movzx ax, byte[digit1]
    mov bl, 10
    mul bl
    movzx bx, byte[digit2]
    add ax, bx
    mov byte[num], al ;We are sure that no less than 256...

    mov word[sum], 0      ;Initializing sum to zero
    movzx ecx, byte[num]  ;Initializing loop variable(ecx) to number

;Loop for adding ecx to num
adding:
    add word[sum], cx ; Assuming maximum value in ecx will use only 16 bits
    Loop adding

;The result could be maximum of 4 digits.....
;In the remaining section of the code, we will break the
;number so as to print each digit one by one

;First splitting the number into two num1, num2 each having maximum 2 digits each
    mov ax, word[sum]
    mov bl, 100
    div bl
    mov byte[num1], al
    mov byte[num2], ah

```

```

;Copying each digits to digit1, digit2, digit3 and digit4
movzx ax, byte[num1]
mov bl, 10
div bl
mov byte[digit4], al
mov byte[digit3], ah

movzx ax, byte[num2]
mov bl, 10
div bl
mov byte[digit2], al
mov byte[digit1], ah

;Converting the digit to its ASCII by adding 30h
add byte[digit1], 30h
add byte[digit2], 30h
add byte[digit3], 30h
add byte[digit4], 30h

;Printing each digits.....
mov eax, 4
mov ebx, 1
mov ecx, digit4
mov edx, 1
int 80h

mov eax, 4
mov ebx, 1
mov ecx, digit3
mov edx, 1
int 80h

mov eax, 4
mov ebx, 1
mov ecx, digit2
mov edx, 1
int 80h

mov eax, 4
mov ebx, 1
mov ecx, digit1
mov edx, 1
int 80h

;Exit code
mov eax, 1
mov ebx, 0
int 80h

```

NB: Problem with the above code is we need to give the input as two digits. If we need to input 9 we need to give it as 09 ↴

Output will be printed in 4 digits. ie even if the sum is just 45 the output will be : 0045

We can correct this by reading each digits by pushing to a stack and the popping out when new line character is being encountered in the input. In your programs you are expected to use this method for reading and writing numbers.

Subprograms

Subprograms are independent parts of the code which can be called at various parts of the main code. They are called functions in high level languages. They are used to group together redundant code and call them repeatedly.

CALL & RET Statements:

In NASM subprograms are implemented using the CALL and RET instructions. The general syntax is like this:

```
;main code.....  
-----  
-----  
call func_name  
-----  
-----  
  
;Sub_Program  
func_name:          ;Label for subprogram  
    -----  
    -----  
    -----  
ret
```

- When we use the CALL instruction, address of the next instruction will be copied to the system stack and it will jump to the subprogram. ie. ESP will be decreased by 4 units and address of the next instruction will go over there.
- When we call the ret towards the end of the sub-program then the address being pushed to the top of the stack will be restored and control will jump to that.

Calling Conventions:

The set of rules that the calling program and the subprogram should obey to pass parameters are called calling conventions. Calling conventions allow a subprogram to be called at various parts of the code with different data to operate on. The data may be passed using system registers / memory variables / system stack. If we are using system stack, parameters should be pushed to system stack before the CALL statement and remember to pop off, preserve and to push back the return address within the sub program which will be in the top of the stack.

Recursive Sub-routine:

A subprogram which calls itself again and again recursively to calculate the return value is called recursive sub-routine. We could implement recursive sub-routine for operations like calculating factorial of a number very easily in NASM.

Using C Library functions in NASM:

We can embed standard C library functions into our NASM Program especially for I/O operations. If we would like to use that then we have to follow C's calling conventions given below:

- parameters are passed to the system stack from left to right order.
Eg: printf("%d",x)
Here value of x must be pushed to system stack and then the format string.
- C-Function won't pop out the parameters automatically, so it is our duty to restore the status of stack pointers(ESP and EBP) after the function being called.

Eg: Reading an integer using the C-Functions...

```
section .text
global main
    ;Declaring the external functions to be used in the program.....
    extern scanf
    extern printf
;Code to read an integer using the scanf function
getint:
    push ebp                ;Steps to store the stack pointers
    mov ebp , esp

    ;scanf("%d",&x)

    ;Creating a space of 2 bytes on top of stack to store the int value
    sub esp , 2
    lea eax , [ ebp-2]
    push eax                ;Pushing the address of that location

    push fmt1               ;Pushing format string
    call scanf              ;Calling scanf function

    mov ax, word [ebp-2]
    mov word[num], ax

    ;Restoring the stack registers.
    mov esp , ebp
    pop ebp
    ret

putint:
    push ebp                ;Steps to store the stack pointers
    mov ebp , esp

    ;printf("%d",x)

    sub esp , 2 ;Creating a space of 2 bytes and storing the int value there
    mov ax, word[num]
    mov word[ebp-2], ax

    push fmt2               ;Pushing pointer to format string
    call printf              ;Calling printf( ) function

    mov esp , ebp           ;Restoring stack to initial values
    pop ebp
    ret
```

```

main:                                ;Main( ) Function
    mov eax, 4
    mov ebx, 1
    mov ecx, msg1
    mov edx, size1
    int 80h

    call getint
    mov ax, word[num]
    mov bx, ax
    mul bx
    mov word[num], ax
    call putint

exit:
    mov ebx, 0
    mov eax, 1
    int 80h

section .data
    fmt1 db "%d",0
    fmt2 db "Square of the number is : %d",10
    msg1: "Enter an integer : "
    size1:      db    $-msg1

section .bss
    num: resw 1

```

NB: *Assembling and executing the code...*

- First we have to assemble the code to object code with NASM Assembler, then we have to use gcc compiler to make the executable code.
- `nasm -f elf -o int.o int.asm`
- `gcc int.o -o int`
- `./int`

Arrays and Strings

An Array is a continuous storage block in memory. Each element of the array have the same size. We access each element of the array using:

- i) Base address / address of the first element of the array.
- ii) Size of each element of the array.
- iii) Index of the element we want to access.

In NASM there is no array element accessing / dereferencing operator like [] in C / C++ / Java using which we can access each element. Here we compute the address of each element using an iterative control structure and traverse through the elements of the array.

Declaring / Initializing an array:

We can declare and initialize arrays in the data section using the normal notations and listing out each elements.

Eg:

```
array1:      db      2, 5, 8, 10, 12, 15          ;An array of 6 bytes
array2:      dw      191, 122, 165, 165          ;An array of 4 words
array3:      dd      111, 111, 111              ;An array of 4 dwords and
                                                ;each having the same value
```

We can also use TIMES keyword to repeat each element with a given value and thus easily create array elements:

Eg:

```
array1:      TIMES   100 db  1  ;An array of 100 bytes with each element=1
array2:      TIMES   20 dw  2  ;An array of 20 dwords
```

We can declare array in .bss section using RESx keyword.

Eg:

```
array1:      resb    100          ;An array of 100 bytes
array2:      resw     20
```

The label which we use to create array(eg: 'array1') acts as a pointer to the base address of the array and we can access each element by adding suitable offset to this base address and then dereferencing it.

Eg:

```
;Let array1 have elements of size 1 byte
mov  al,  byte[array1]      ;First element of the array copied to al reg
mov  cl,  byte[array1 + 5]  ;array1[5], ie. 6th element copied to cl reg

;Let array2 have elements of size 1 word(2bytes)
mov  ax,  word[array2]      ;First element of the array copied to ax reg
mov  dx,  word[array2 + 8]  ;array2[4], ie 5th element of the array copied
                           ;to dx reg.
```

The general syntax of using array offset is:

$$[base_reg + factor * index_reg + constant]$$

base_reg: It should be general purpose register containing the base address of the array.

factor: It can be 1, 2, 4 or 8.

index_reg: It can also be any of the general purpose registers.

constant: It should be an integer.

Eg:

```
byte[ebx+12]
word[ebp + 4 * esi]
dword[ebx - 12]
```

Sample Program - To search an array for an element(Traversal):

- First we read n, the number of elements.
- Then we store the base address of array to ebx reg.
- Iterate n times, read and keep the ith element to [ebx].
- Then read the number to be searched.
- Iterate through the array using the above method.
- Print success if the element is found.

```
section .bss
    digit0: resb 1
    digit1: resb 1
    array:  resb 50          ;Array to store 50 elements of 1 byte each.
    element: resb 1
    num:    resb 1
    pos:    resb 1
    temp:   resb 1
    ele:    resb 1

section .data
    msg1: db "Enter the number of elements : "
    size1: equ $-msg1
    msg2: db "Enter a number:"
    size2: equ $-msg2
    msg3: db "Enter the number to be searched : "
    size3: equ $-msg3
    msg_found: db "Element found at position : "
    size_found: equ $-msg_found
    msg_not: db "Element not found"
    size_not: equ $-msg_not

section .text
    global _start
```

```

_start:

;Printing the message to enter the number
mov eax, 4
mov ebx, 1
mov ecx, msg1
mov edx, size1
int 80h

;Reading the number
mov eax, 3
mov ebx, 0
mov ecx, digit1
mov edx, 1
int 80h

mov eax, 3
mov ebx, 0
mov ecx, digit0
mov edx, 1
int 80h

mov eax, 3
mov ebx, 0
mov ecx, temp
mov edx, 1
int 80h

sub byte[digit1], 30h
sub byte[digit0], 30h

mov al, byte[digit1]
mov dl, 10
mul dl
mov byte[num], al
mov al, byte[digit0]
add byte[num], al
mov al, byte[num]

mov byte[temp], al

mov ebx, array

reading:

push ebx ;Preserving The value of ebx in stack

;Printing the message to enter each element
mov eax, 4
mov ebx, 1
mov ecx, msg2
mov edx, size2
int 80h

```

```

;Reading the number
mov eax, 3
mov ebx, 0
mov ecx, digit1
mov edx, 1
int 80h

mov eax, 3
mov ebx, 0
mov ecx, digit0
mov edx, 2
int 80h

sub byte[digit1], 30h
sub byte[digit0], 30h
mov al, byte[digit1]
mov dl, 10
mul dl
add al, byte[digit0]

;al now contains the number
pop ebx

mov byte[ebx], al

add ebx, 1

dec byte[temp]
cmp byte[temp], 0      ;Comparing loop variable
jg reading             ;Loop using branch statements

;Reading the number to be searched :.....

mov eax, 4
mov ebx, 1
mov ecx, msg3
mov edx, size3
int 80h

;Reading the number
mov eax, 3
mov ebx, 0
mov ecx, digit1
mov edx, 1
int 80h

mov eax, 3
mov ebx, 0
mov ecx, digit0
mov edx, 2
int 80h

sub byte[digit1], 30h
sub byte[digit0], 30h
mov al, byte[digit1]
mov dl, 10
mul dl

```

```

    add al, byte[digit0]

    mov byte[ele], al

    movzx ecx, byte[num]

    mov ebx, array
    mov byte[pos], 1

searching:

    push ecx

    mov al, byte[ebx]

    cmp al, byte[ele]
    je found

    add ebx, 1

    pop ecx
    add byte[pos], 1

loop searching

    mov eax, 4
    mov ebx, 1
    mov ecx, msg_not
    mov edx, size_not
    int 80h

exit:
    mov eax, 1
    mov ebx, 0
    int 80h

found:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_found
    mov edx, size_found
    int 80h

    movzx ax, byte[pos]
    mov bl, 10
    div bl
    mov byte[digit1], al
    mov byte[digit0], ah

    add byte[digit0], 30h
    add byte[digit1], 30h

```



```

mov eax, 4
mov ebx, 1
mov ecx, digit1
mov edx, 1
int 80h

mov eax, 4
mov ebx, 1
mov ecx, digit0
mov edx, 1
int 80h
jmp exit

```

Strings:

Strings are stored in memory as array of characters. Each character in English alphabet has a 8-bit unique numeric representation called ASCII. When we read a string from the user, the user will give an enter key press at the end of the string. When we read that using the read system call, the enter press will be replaced with a new line character with ASCII code 10 . Thus we can detect the end of the string.

Sample Program - To count the number of each vowels in a string :

```

section .data
    a_cnt: db 0
    e_cnt: db 0
    i_cnt: db 0
    o_cnt: db 0
    u_cnt: db 0
    string_len: db 0
    msg1: db "Enter a string : "
    size1: equ $-msg1
    msg_a: db 10 , "No: of A : "
    size_a: equ $-msg_a
    msg_e: db 10, "No: of E : "
    size_e: equ $-msg_e
    msg_i: db 10, "No: of I : "
    size_i: equ $-msg_i
    msg_o: db 10, "No: of O : "
    size_o: equ $-msg_o
    msg_u: db 10, "No: of U : "
    size_u: equ $-msg_u

section .bss
    string: resb 50
    temp: resb 1

section .data
    global _start

```

```

_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg1
    mov edx, size1
    int 80h

    mov ebx, string

reading:
    push ebx

    mov eax, 3
    mov ebx, 0
    mov ecx, temp
    mov edx, 1
    int 80h

    pop ebx
    cmp byte[temp], 10
    je end_reading
    inc byte[string_len]
    mov al, byte[temp]
    mov byte[ebx], al
    inc ebx
    jmp reading

end_reading:
    mov byte[ebx], 0

    mov ebx, string
counting:
    mov al, byte[ebx]
    cmp al, 0
    je end_counting

    cmp al, 'a'
    je inc_a
    cmp al, 'e'
    je inc_e
    cmp al, 'i'
    je inc_i
    cmp al, 'o'
    je inc_o
    cmp al, 'u'
    je inc_u

    cmp al, 'A'
    je inc_a
    cmp al, 'E'
    je inc_e
    cmp al, 'I'
    je inc_i
    cmp al, 'O'
    je inc_o
    cmp al, 'U'
    je inc_u

```

```

next:
    inc ebx
    jmp counting

end_counting:

;Printing the no of a
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_a
    mov edx, size_a
    int 80h

    add byte[a_cnt], 30h

    mov eax, 4
    mov ebx, 1
    mov ecx, a_cnt
    mov edx, 1
    int 80h

;Printing the no of e
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_e
    mov edx, size_e
    int 80h

    add byte[e_cnt], 30h

    mov eax, 4
    mov ebx, 1
    mov ecx, e_cnt
    mov edx, 1
    int 80h

;Printing the no of i
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_i
    mov edx, size_i
    int 80h

    add byte[i_cnt], 30h

    mov eax, 4
    mov ebx, 1
    mov ecx, i_cnt
    mov edx, 1
    int 80h

```

```

;Printing the no of o
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_o
    mov edx, size_o
    int 80h

    add byte[o_cnt], 30h

    mov eax, 4
    mov ebx, 1
    mov ecx, o_cnt
    mov edx, 1
    int 80h

;Printing the no of u
    mov eax, 4
    mov ebx, 1
    mov ecx, msg_u
    mov edx, size_u
    int 80h

    add byte[u_cnt], 30h

    mov eax, 4
    mov ebx, 1
    mov ecx, u_cnt
    mov edx, 1
    int 80h

exit:
    mov eax, 1
    mov ebx, 0
    int 80h

inc_a:
    inc byte[a_cnt]
    jmp next

inc_e:
    inc byte[e_cnt]
    jmp next

inc_i:
    inc byte[i_cnt]
    jmp next

inc_o:
    inc byte[o_cnt]
    jmp next

inc_u:
    inc byte[u_cnt]
    jmp next

```

Two / Multi-Dimensional Arrays:

Memory / RAM is a continuous storage unit in which we cannot directly store any 2-D Arrays/Matrices/Tables. 2-D Arrays are implemented in any programming language either in row major form or column major form. In row major form first we store the row 1 then row 2 then row 3 and it goes on. In column major form we store column 1 first, then column 2 then column 3 and goes on till the last element of last column. For example if we have a 2 x 3 matrix say A of elements 1 byte each. Let the starting address of the array be 12340. Then the array will be stored in memory as:

a) Row Major Form

Address	Element
12340	A[0][0]
12341	A[0][1]
12342	A[0][2]
12343	A[1][0]
12344	A[1][1]
12345	A[1][2]

b) Column Major Form

Address	Element
12340	A[0][0]
12341	A[1][0]
12342	A[0][1]
12343	A[1][1]
12344	A[0][2]
12345	A[1][2]

Using this concept we can implement the 2-D array in NASM Programs.

Array / String Operations:

x86 Processors have a set of instructions designed specially to do array / string operations much easily compared with the traditional methods demonstrated above. They are called String Instructions. Even though it is termed as string instructions, it work well with general array manipulations as well. They use index registers(ESI & EDI) and increments / decrements either one or both the registers after each operation. Depending on the value of Direction Flag(DF) it either increments or decrements the index register's value. The following instructions are used to set the value of DF manually:

- i) CLD - Clears the Direction Flag. Then the string instruction will increment the values of index registers.
- ii) STD - Sets the Direction Flag to 1. Then the string instructions will decrement the values of index registers.

NB: Always make sure to set the value of Direction Flags explicitly, else it may lead to unexpected errors.

For string operations we must make sure to have DS to be the segment base of Source string and ES to be the segment base of Destination String. As we are using the protected mode we need not set them manually. But in real mode we have to set the register values to the base address of the suitable segments properly.

1. Reading an array element to reg(AL/AX/EAX):

LODSx: $x = B / W / D$ - Load String Instruction

This instruction is used to copy one element from an array to the register. It can transfer an element of size 1 Byte / 1 Word / 4 Bytes at a time.

LODSB

```
AL = byte[DS:ESI]
ESI = ESI ± 1
```

LODSW

```
AX = word[DS:ESI]
ESI = ESI ± 2
```

LODSD

```
EAX= dword[DS:ESI]
ESI = ESI ± 4
```

2. Storing a reg(AL/AX/EAX) to an array:

STOSx: $x = B / W / D$ - Load String Instruction

This instruction is used to copy one element from a register to an array. It can transfer an element of size 1 Byte / 1 Word / 4 Bytes at a time.

STOSB

```
byte[ES:EDI] = AL
EDI = EDI ± 1
```

STOSW

```
word[ES:EDI] = AX
EDI = EDI ± 2
```

STOSD

```
dword[ES:EDI] = EAX
EDI = EDI ± 4
```

NB: ESI - Source Index reg is used when the array acts as a source ie. A value is copied from that
EDI - Destination Index reg is used when the array acts as a destination ie. A value is copied to that.

Eg: Program to increment the value of all array elements by 1

```
section .data
    array1: db 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

section .text
    global _start
_start:
    CLD          ;Clears the Direction Flag
    mov esi, array1      ;Copy Base address of array to index registers
    mov edi, array1
    mov ecx, 10      ;No: of element in the array

increment:
    LODSB
    INC al
    STOSB
    loop increment

-----
-----
-----
```

3. Memory Move Instructions:

These instructions are used to copy the elements of one array/string to another.

MOVSx: $x = B / W / D$ - *Move String Instruction*

MOVSB

```
byte[ES:EDI] = byte[DS:ESI]
ESI = ESI ± 1
EDI = EDI ± 1
```

MOVSW

```
word[ES:EDI] = word[DS:ESI]
ESI = ESI ± 2
EDI = EDI ± 2
```

MOVSD

```
dword[ES:EDI] = dword[DS:ESI]
ESI = ESI ± 4
EDI = EDI ± 4
```

Eg: Program to copy elements of an array to another

```
section .data
    array1: dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
section .bss
    array2: resd    10

section .text
    global _start
```

```

_start:
    CLD          ;Clears the Direction Flag
    mov esi, array1      ;Copy Base address of array to index registers
    mov edi, array2
    mov ecx, 10      ;No: of element in the array

copy:
    MOVSD
loop copy

-----
-----
-----

```

4. REP - Repeat String Instruction

Sy: *REP* <string-instruction>

Repeats a string instruction. The number of times repeated is equal to the value of ecx register(just like loop instruction)

Eg: Previous program can also be written as follows using REP instruction:

```

section .data
    array1: dd      1, 2, 3, 4, 5, 6, 7, 8, 9, 10
section .bss
    array2: resd    10

section .text
    global _start
_start:
    CLD          ;Clears the Direction Flag
    mov esi, array1      ;Copy Base address of array to index registers
    mov edi, array2
    mov ecx, 10      ;No: of element in the array

REP     MOVSD

-----
-----
-----

```

5. Compare Instructions

CMPSx : x = B / W / D - Compares two array elements and affects the CPU Flags just like CMP instruction.

CMPSB
 Compares byte[DS:ESI] with byte[ES:EDI]
 ESI = ESI ± 1
 EDI = EDI ± 1

CMPSW
 Compares word[DS:ESI] with word[ES:EDI]
 ESI = ESI ± 2
 EDI = EDI ± 2

CMPSD

Compares dword[DS:ESI] with dword[ES:EDI]
ESI = ESI ± 4
EDI = EDI ± 4

6. Scan Instructions

SCASx : $x = B / W / D$ - Compares a register(AL/AX/EAX) with an array element and affects the CPU Flags just like CMP instruction.

SCASB

Compares value of AL with byte[ES:EDI]
EDI = EDI ± 1

SCASW

Compares value of AX with word[ES:EDI]
EDI = EDI ± 2

SCASD

Compares value of EAX with dword[ES:EDI]
EDI = EDI ± 4

Eg: Scanning an array for an element

```
section .data
    array1: db      1, 5, 8 , 12, 13, 15, 28 , 19, 9, 11

section .text
    global _start
_start:
    CLD          ;Clears the Direction Flag
    mov  edi, array1      ;Copy Base address of array to index registers
    mov  ecx, 10          ;No: of element in the array
    mov  al, 15           ;Value to be searched

scan:
    SCASB
    je found
loop scan
    jmp not_found
-----
-----
```

Floating Point Operations

As in the case of characters, floating point numbers also cannot be represented in memory easily. We need to have some conversion to have a unique numeric representation for all floating point numbers. For that we use the standard introduced by IEEE(Institute of Electrical and Electronics Engineers) called IEEE-754 Standard. In IEEE-754 standard a floating point number can be represented either in Single Precision(4 Bytes) or Double Precision(8 Bytes). There is also another representation called Extended Precision which uses 10 Bytes to represent a number. If we convert a double precision floating point number to a single precision number, the result won't go wrong but it will be less precise.

IEEE-754 Standard can be summarized as follows. For further details about this refer some books on Computer Organization / Architecture.

Single Precision	1 bit	8 bits	32 bits
Double Precision	1 bit	11 bits	52 bits
	S	Exponent	Fraction

S - Sign Bit. If the number is -ve then S=1 else S = 0.

The number $f = (-1)^S \times (1 + 0.\text{Fraction}) \times 2^{\text{Exponent}-\text{Bias}}$

$1.0 \leq |\text{Significand}| < 2.0$

Bias: 127 for Single Precision
1203 for Double Precision

Floating Point Hardware:

In the early intel microprocessors there were no built in floating point instructions. If we had to do some floating point operations then we had to do that using some software emulators(which will be about 100 times slower than direct hardware) or adding an extra chip to the PC's mother board called math coprocessor or floating point coprocessor. For 8086 and 8088 the math coprocessor was 8087. For 80286 it was 80287 and for 80386 it was 80387. From 80486DX onwards intel started integrating the math co-processor into the main processor. But still it exists like a separate unit inside the main processor with a separate set of instructions, separate stack and status flags.

There hardware for floating point operations are made for even doing operations in extended precision. But if we are using single / double precision numbers from memory it will be automatically converted while storing or loading. There are eight floating point registers called ST0, ST1, ST2.....ST7. They are organized in the form of a stack with ST0 in the top. When we push or load a value to the stack, value of each registers is pushed downward by one register. Using these floating point registers we implement floating point operations. There is also a 1 word Status Flag for the floating point operations, which is analogous to the CPU Flags. It contains the status of the floating point operations.

ST0
ST1
ST2
ST3
ST4
ST5
ST6
ST7

C ₁₅	C ₁₄	C ₁₃	C ₁₂	C ₁₁	C ₁₀	C ₉	C ₈	C ₇	C ₆	C ₅	C ₄	C ₃	C ₂	C ₁	C ₀
-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

Floating Point Control Word

Floating Point Stack

Floating Point Instructions:

Load / Store Instructions:

FLD **src** : Loads the value of src on to the top of F.P stack(ie. ST0). src should be either a f.p register or a single precision / double precision / extended precision f.p number in memory.

ST0 = src

FILD **src** :Loads an integer from memory to ST0. src should be a word / double word / quad word in memory.

ST0 = (float) src

FLD1 :Stores 1 to the top of F.P Stack.

FLDZ :Stores 0 to the top of F.P Stack.

*Eg: fld dword[temp]
 fld ST5*

FST **dest** :Stores ST0 to dest and will not pop off the value from the stack. dest should be a coprocessor register or a single precision / double precision f.p number in memory.

dest = ST0

FSTP **dest** :Works similar to FST, it also pops off the value from ST0 after storing.

FIST **dest** :Converts the number present in the top of f.p stack to an integer and stores that into dest. dest should be a coprocessor register or a single precision / double precision f.p number in memory. The way the number is being rounded depend on the value of coprocessor flags. But default it will be set in such a way that the number in ST0 is being rounded to the nearest integer.

dest = (float)ST0

FISTP dest :Works similar to FIST and it will also pop off the value from top of the stack.

FXCH STn :The nth coprocessor register will be exchanged with ST0.
 $ST0 \leftrightarrow STn$

FFREE STn :Frees up the nth coprocessor register and marks it as empty.

Arithmetic Operations:

FADD src : $ST0 = ST0 + src$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FIADD src : $ST0 = ST0 + (float)src$. This is used to add an integer with ST0. src should be word or dword in memory.

FSUB src : $ST0 = ST0 - src$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FSUBR src : $ST0 = src - ST0$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FISUB src : $ST0 = ST0 - (float)src$, this is used to subtract an integer from ST0. src should be word or dword in memory.

FISUBR src : $ST0 = (float)src - ST0$, src should be word or dword in memory.

FMUL src : $ST0 = ST0 \times src$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FIMUL src : $ST0 = ST0 \times (float)src$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FDIV src : $ST0 = ST0 \div src$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FDIVR src : $ST0 = src \div ST0$, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FIDIV src : $ST0 = ST0 \div (float)src$, src should be a word or dword in memory.

FIDIVR src : $ST0 = (float)src \div ST0$, src should be a word or dword in memory.

Comparison Instructions:

The usual comparison instructions FCOM and FCOMP affects the coprocessors status word but the processor cannot execute direct jump instruction by checking these values. So we need to copy the coprocessor flag values to the CPU flags in order to implement a jump based on result of comparison. FSTSW instruction can be used to copy the value of coprocessor status word to AX and then we can use SAHF to copy the value from AL to CPU flags.

FCOM src :Compares src with ST0, src should be a coprocessor register or a single precision / double precision f.p number in memory.

FCOMP src :Works similar to FCOM, it will also pop off the value from ST0 after comparison.

FSTSW src :Stores co-processor status word to a dword in memory or to AX register.

SAHF :Stores AH to CPU Flags.

Eg:

```
fld    dword[var1]
fcomp  dword[var2]
fstsw
sahf
ja     greater
```

In Pentium Pro and later processors(like Pentium II, III, IV etc) Intel added two other instructions FCOMI and FCOMIP which affects the CPU Flags directly after a floating point comparison. These instructions are comparatively easier than the trivial ones.

FCOMI src :Compares ST0 with src and affects the CPU Flags directly. src must be coprocessor register.

FCOMIP src :Compares ST0 with src and affects the CPU Flags directly. It will then pop off the value in ST0. src must be a coprocessor register.

Miscellaneous Instructions:

FCHS	: $ST0 = -(ST0)$
FABS	: $ST0 = ST0 $
FSQRT	: $ST0 = \sqrt{ST0}$
FSIN	: $ST0 = \sin(ST0)$
FCOS	: $ST0 = \cos(ST0)$
FLDPI	: Loads value of π into ST0

NB: We use C-Functions to read or write floating point numbers from users. We cannot implement the read and write operations with 80h interrupt method easily.

Eg: Program to find the average of n floating point numbers:

```
section .text
global main
extern scanf
extern printf

print:
    push ebp
    mov ebp, esp
    sub esp, 8
    fst qword[ebp-8]
    push format2
    call printf
    mov esp, ebp
    pop ebp
ret

read:
    push ebp
    mov ebp, esp
    sub esp, 8
    lea eax, [esp]
    push eax
    push format1
    call scanf
    fld qword[ebp-8]
    mov esp, ebp
    pop ebp

ret

main:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg1
    mov edx, len1
    int 80h

    mov eax, 3
    mov ebx, 0
    mov ecx, temp1
    mov edx, 1
    int 80h

    mov eax, 3
    mov ebx, 0
    mov ecx, temp2
    mov edx, 1
    int 80h

    mov eax, 3
    mov ebx, 0
    mov ecx, temp3
    mov edx, 1
    int 80h
```

```

    sub byte[temp1], 30h
    sub byte[temp2], 30h

    mov al, byte[temp1]
    mov bl, 10
    mul bl
    add al, byte[temp2]
    mov byte[num], al
    mov byte[ctr], al

    mov ah, 0
    mov word[num2], ax

    fldz

reading:
    call read
    fadd ST1

    dec byte[ctr]
    cmp byte[ctr], 0
    jne reading

    fidiv word[num2]

    call print

exit:
    mov eax, 1
    mov ebx, 0
    int 80h

section .data
    format1: db "%lf",0
    format2: db "The average is : %lf",10
    msg1: db "Enter the number of numbers : "
    len1: equ $-msg1

section .bss
    temp1: resb 1
    temp2: resb 1
    temp3: resb 1
    num:      resb 1
    ctr:      resb 1
    num2:     resw 1

```

Reference

1. PC Assembly Language Tutorial - *Dr. Paul Carter*
www.drpaulcarter.com/pcasm/
2. The Art of Assembly Language, 2nd Edition - *Randall Hyde*
3. Assembly Language Step-by-Step: Programming with Linux - *Jeff Duntemann*