Introduction to NASM

Prepared By:

Muhammed Yazar Y

Updated By:

Sonia V Mathew
Govind R
Darshana Suresh - Dheeraj Mohan - Jyothsna Shaji
Lakshmi Alwin - Naveen Babu - Nikhil Sojan
Nileena P.C. - Sanju Alex Jacob - Vrindha K

B Tech
Dept: of CSE - NIT Calicut

Under The Guidance of:

Mr. Jayaraj P B
Mr. Saidalavi Kalady

Assistant Professors

Dept: of CSE
NIT Calicut

Department Of Computer Science & Engineering

NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

2019

Contents

Acknowledgement

I would like to express my gratitude to Saidalavy Kalady Sir and Jayaraj P B Sir (Assistant Professors, Dept: of CSE, NIT Calicut) for guiding me throughout while making this reference material on NASM Assembly language programming. Without their constant support and guidance this work would not have been possible. Thanks to Lyla B Das madam (Associate Professor, Dept: of ECE, NIT Calicut) for encouraging me to bring out an updated version of this work. I would also wish to thank Meera Sudharman, my classmate who helped me in verifying the contents of this document. Special thanks are due to Govind R and Sonia V Mathew (BTech 2011-2015 Batch) for updating the contents and adding more working examples to it. I am extremely grateful to Dheeraj Mohan, Nikhil Sojan, Darshana Suresh, Jyothsna Shaji, Lakshmi Alwin, Vrindha K, Nileena P.C., Sanju Alex Jacob and Naveen Babu (2016 - 2020 batch) for restructuring the manual with different learning strategies and working examples. I want to Thanks to all my dear batch mates and juniors who have been supporting me through the work of this and for providing me with their valu- able suggestions.

Muhammed Yazar

Chapter 1

Basics of Computer Organization

Machine Language

Machine language consists of instructions in the form of 0âĂŹs and 1âĂŹs. Every CPU has its own machine language. It is very difficult to write programs using the combination of 0âĂŹs and 1âĂŹs. So we rely upon either assembly language or high level language for writing programs.

Assembly Language

An assembly language is a low-level programming language for microprocessors and other programmable devices. Assembly language uses a mnemonic to represent each low-level machine instruction.

Why Assembly Language ?

The symbolic programming of Assembly Language is easier to understand and saves a lot of time and effort of the programmer. 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. Some portions of Operating System (Eg: Linux kernel) and some system software are written in assembly language. In programming languages like C, C++ we can even em-

bed assembly language instructions into it using functions like asm( ); (Refer to Chapter 8 of 'The Intel Microprocessors' by Barry B. Brey for more details)

What is NASM ?

The Netwide Assembler is an assembler and disassembler for the Intel X86 architecture (explained in subsequent section). It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. NASM is considered to be one of the most popular assemblers for Linux.

Computer Architecture

The basic operational design of a computer is called architecture. It is a set of rules and methods that describe the functionality, organization and implementation of computer systems. X86 architecture follows Von Neumann architecture which is based on stored program concept.

Von Neumann Architecture

Von Neumann architecture machine, designed by physicist and mathematician John Von Neumann (1903 - 1957) is a theoretical design for a stored program computer that serves as the basis for almost all modern computers. A Von Neumann machine consists of a central processor with an arithmetic/logic unit and a control unit, a memory, mass storage, and input and output.

X86 Architecture

The x86 architecture is an Instruction Set Architecture (ISA) series for computer processors. Developed by Intel Corporation, x86 architecture defines how a processor handles and executes different instructions passed from the operating system (OS) and software programs. The 'x' in x86 denotes ISA version.
Key features include:

- Provides a logical framework for executing instructions through a processor.
- Allows software programs and instructions to run on any processor in the Intel 8086 family.

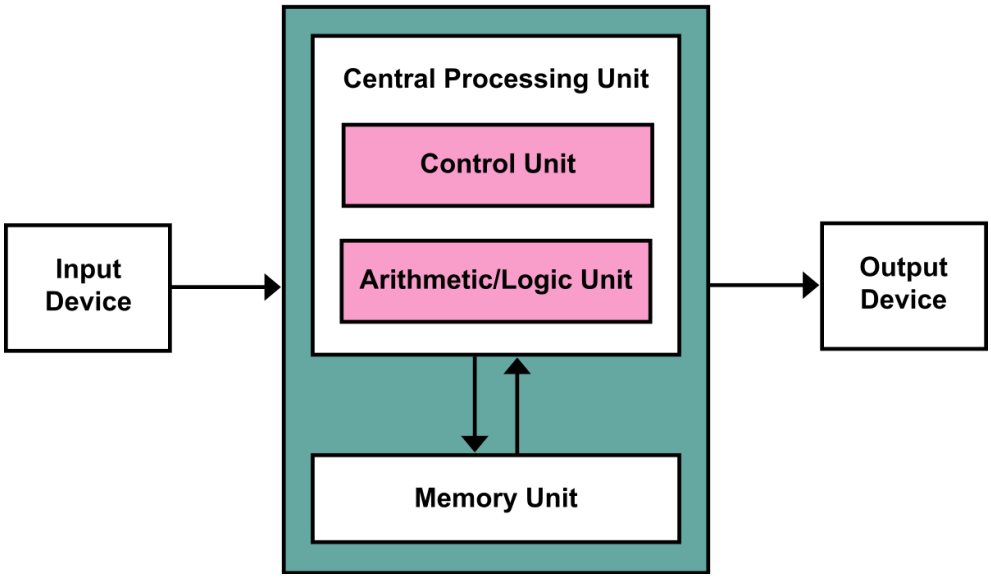


Figure 1.1: Von Neumann Architecture

- Provides procedures for utilizing and managing the hardware components of a central processing unit (CPU).

Historically there have been 2 types of Computers:
Fixed Program Computers - Their function is very specific and they can't be programmed, e.g. Calculators.
Stored Program Computers - These can be programmed to carry out many different tasks, applications are stored on them, hence the name.

Processor

Processor is the brain of the computer. It performs all mathematical, logical and control operations of a computer. It is the main 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.

The term processor is used interchangeably with the term central processing unit (CPU), although strictly speaking, the CPU is not the only processor in a computer.

Figure 1.2: Intel Core i9

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.

What is a clock cycle?

In computers, the clock cycle is the amount of time between two pulses of an oscillator. It is a single increment of the central processing unit (CPU) clock during which the smallest unit of processor activity is carried out. The clock cycle helps in determining the speed of the CPU, as it is considered the basic unit of measuring how fast an instruction can be executed by the computer processor. height.

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 canrun 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 onlyproblem is that, the program will not use the complete set of registers andother available resources and thus it will be less efficient.

A register may hold an instruction, a storage address, or any kind of data (such as a bit sequence or individual characters). A register must be large enough to hold an instruction - for example, in a 64-bit computer, a register must be 64 bits in length.

1. General Purpose Registers
   General purpose registers are used to store temporary data within the microprocessor. 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 (see image). 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.
   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).

2. Flags and EIP
   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

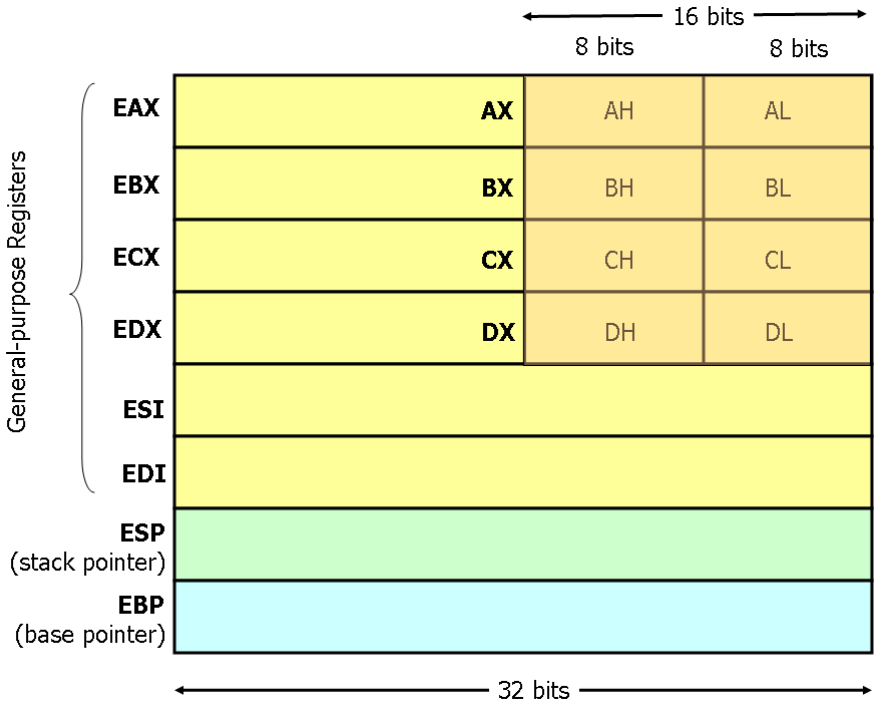


Figure 1.3: i386 Registers

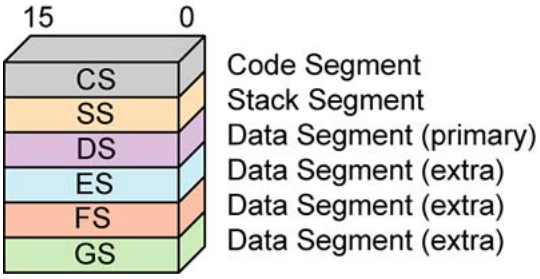


Figure 1.4: Segment Registers

in FLAGS need special mention:

- Carry Flag: When a processor does 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:

- Data
- 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 shortly) alter the value of EIP.

3. 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

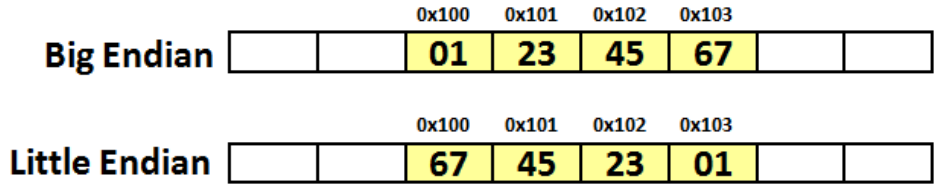
base address of a particular data section and Offset will contain how many bytes should be displaced from the segment register to access the particular data. CS contains the base address of Code Segment and EIP is the offset. It keeps on updating while executing each instruction. SS or Stack Segment 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.

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.

Little Endian and Big Endian

Endianness refers to the sequential order in which bytes are arranged into larger numerical values when stored in memory or when transmitted over digital links. Endianness is of interest in computer science because two conflicting and incompatible formats are in common use: words may be represented in big-endian or little-endian format, depending on whether bits or bytes or other components are ordered from the big end (most significant bit) or the little end (least significant bit).
Big Endian Byte Order: The most significant byte (the "big end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.
Little Endian Byte Order: The least significant byte (the "little end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

Suppose an integer is stored as 4 bytes(32-bits), then a variable with value 0x01234567(Hexa-decimal representation) is stored as four bytes 0x01, 0x23, 0x45, 0x67, on Big-endian while on Little-Endian (Intel x86), it will be stored in reverse order:



Bus

Bus is a name given to any communication medium, that transfers data between two components. (A bus is a subsystem that is used to connect computer components and transfer data between them). A bus may be parallel or serial. Parallel buses transmit data across multiple wires. Serial buses transmit data in bit-serial format. We can classify the buses associated with the processor into three.

- 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

- Address Bus :
  The address bus is used by the CPU or a direct memory access (DMA) enabled device to locate the physical address to communicate read/write commands. Memory management Unit (MMU) 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. All address buses are read and written by the CPU in the form of bits. An address bus is measured by the amount of memory a system can retrieve. A system with a 32-bit address bus can address 4 gigabytes of

memory space.
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 GB of RAM in a 32 bit PC, system cannot find more than 4 GB of available memory.

Processor	Address Bus Width	Maximum Addressable RAM size
8088, 8086, 80186, 80188	20	1 MB
80386SX, 80286	24	16 MB
80486, 80386DX, Pentium, Pentium Override	32	4 GB
Pentium II, Pentium Pro	36	64 GB

- 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 MMU 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 data bus will be written to that location. When the write bit is enabled, MMU will write the data in the data bus into the address location in the address bus.

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.

Here interrupt can be seen as a signal from a device, such as the keyboard, to the CPU, telling processor to immediately stop whatever it is currently doing and do something else. For example, the keyboard controller sends an interrupt when a key is pressed. To know how to call on the kernel when a specific interrupt arise, the CPU has a vector table setup by the OS, and stored in memory. There are 256 interrupt vectors on x86 CPUs, numbered from 0 to 255 which act as entry points into the kernel.The number of interrupt vectors or entry points supported by a CPU differs based on the CPU architecture.

System call

System calls are Application Programmer's Interface to the kernel space. In a NASM program, input has to be taken from the standard input device (Keyboard) and output has to be given to the standard output device (monitor). This is implemented using the Operating System's **read** and **write** system call respectively. Interrupt number 80h(in hexadecimal) is given to the software generated interrupt in Linux Systems. Applications invoke the System Calls using this interrupt number. 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 a system call is,

- System call number is stored in **EAX** register.
- Other parameters needed to implement the system call is stored in other general purpose registers.
- Trigger the 80h interrupt using the instruction **INT 80h**.

Then OS will implement the system call.

System memory in Linux can be divided into two distinct regions: kernel space and user space. Kernel space is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services.

Chapter 2

Introduction to NASM

Sections in NASM

A NASM program is divided into three sections.

1. section .text :
   This section contains the executable code from where execution starts. It is analogous to the main() function in C.
2. section .bss :
   Here, variables are declared without initialisation.
3. section .data :
   Variables are declared and initialised in this section.
   For declaring space in the memory the following directives are used,

   (a) RESx:
   Reserve just space in memory for a variable without giving any initial values.

   (b) Dx:
   Declaring space in the memory for any variable and also providing the initial values at that moment. Where x can be replaced with different characters as shown in the below table.

x	Meaning	Bytes
b	BYTE	1
w	WORD	2
d	DOUBLE WORD	4
q	QUAD WORD	8
t	TEN BYTE	20

Examples:

```
section .data
var1: db 10  ;Reserve one byte in memory for storing var1 and var1=10
var2 : db 1,2,3,4
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.

```
section .bss
var1: resb 1
var2: resq 1
var3: resw 1
```

TIMES
It 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 each element
will be initialized with the value 1.

Dereferencing in NASM

To access the data stored at an address, the dereferencing operator used is '[ ]'.

Examples:
mov eax, [var] ;Value at address location var would be copied to eax

mov eax,var ;Address location var is copied to eax

Type casting

It is required for the operands for which the assembler cannot predict the number of memory locations to dereference 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 datatype are: BYTE, WORD, DWORD, QWORD, TWORD.
Eg:
MOV dword[ebx], 1
INC BYTE[label]
ADD eax, dword[label]

x86 Instruction Set

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.
   Syntax: 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 register to the variable var1 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
syntax : mov src, dest

- size of dest should be greater than or equal to 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

3. ADD - Addition
   Sytax : add dest, src
   dest = dest + src;
   Used to add the values of two registers/memory variables 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
   Sytax : sub dest, src
   dest = dest - src;
   Used to subtract the values of two registers/memory variables 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 registers/memory variables by 1
   Eg:
   INC eax ; eax++
   INC byte[var]
   INC al

6. DEC - Decrement operation
   Used to decrement the value of a registers/memory variables by 1
   Eg:
   DEC eax ; eax- -
   DEC byte[var]
   DEC al

7. MUL - Multiplication
   Syntax : mul src
   Used to multiply the value of a registers/memory variables 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 (AX*src) 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. It can be used mainly in three different forms.
   Syntax :

   (a) imul src

(b) imul dest, src

(c) imul dest, src1, src2

- If we use imul as in (a) then its working follows the same rules of MUL.
- If we use that in (b) form then dest = dest * src.
- If we use that in (c) form then dest = src1 * scr2.

9. DIV - Division
   Synatx : div src
   Used to divide the value of EDX:EAX or DX:AX or AX register with registers/memory variables in src. 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 divided 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.
    Sytax : NEG op1
    NEG Instruction negates a given registers/memory variables.

11. CLC - Clear Carry
    This instruction clears the carry flag bit in CPU FLAGS.

12. ADC - Add with Carry
    Syntax : 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

Syntax : 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 subtraction of ecx from eax
sbb edx, ebx    ; Subtracting with carry for the higher bits.
```

Branching In x86

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.

15. CMP - Compares the Operands
    Syntax :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 $op1 \neq op2$ then the Zero Flag(ZF) will be set to 1.
    NB: For generating conditional jumps in X86 programming we will first perform the CMP operation between two registers/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

Advanced Conditional Jump Instructions:
In 80x86 processors Intel has added some enhanced versions of the conditional operations which are much more easier than 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.

(a) 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 $op1 \geq op2$

(b) For Signed numbers:

Instruction	Working
JE	Jump if $op1 = op2$
JNE	Jump if $op1 \neq op2$
JG (jump if above)	Jump if op1 >op2
JNG	Jump if $op1 \leq op2$
JL (jump if below)	Jump if op1 <op2
JNL	Jump if $op1 \geq op2$

16. LOOP instruction
    Syntax: 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 $ecx \neq 0$. If so it will jump to the label following that instruction. Else control jumps to the very next statement.

    Converting Standard C/C++ Control Structures to x86:

    (a) If-else

C Code	x86 equivalent
if($eax \leq 5$)	cmp eax, 5 ;Comparing
$eax = eax + ebx$;	ja if
else	else: ;Else part
$ecx = ecx + ebx$;	add ecx, ebx
	jmp L2
	if: ;If part
	add eax, ebx
	L2:

    (b) For loop

C Code	x86 equivalent
eax = 0;	mov eax, 0
for($ebx = 1$ to 10)	mov ebx, 1
$eax = eax + ebx$;	for:
	add eax, ebx
	cmp ebx, 10
	jbe for

(c) While loop

C Code	x86 equivalent
$sum = 0$;	mov dword[sum], 0
$ecx = n$;	mov ecx, dword[n]
while($ecx >= 0$)	addition:
$sum = sum + ecx$;	add [sum], ecx
$ecx - -$;	loop addition ; Decrements ecx and checks if ecx is not equal to 0 , if so it will jump to addition

Boolean Operators

17. AND - (Bitwise Logical AND)
Syntax : AND op1, op2
Performs bitwise logical AND operation of op1 and op2, assign the result to op1.
$op1 = op1 \& op2$; //Equivalent C Statement
Let $x = 10101001b$ and $y = 10110010b$ 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)
Syntax: OR op1, op2
Performs bitwise logical OR operation of op1 and op2, assign the result to op1.
$op1 = op1 || op2$; //Equivalent C Statement
Let $x = 10101001b$ and $y = 10110010b$ 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)
Syntax: XOR op1, op2

Performs bitwise logical XOR operation of op1 and op2, assign the result to op1.
$op1 = op1 \hat{} op2$; //Equivalent C Statement
Let $x = 10101001b$ and $y = 10110010b$ be two 8-bit binary numbers. Then $x \oplus y$

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)
Syntax: NOT op1
Performs bitwise logical NOT of op1 and assign the result to op1.
$op1 = \neg op1$; //Equivalent C Statement

x	1	0	1	0	1	0	0	1
$\neg x$	0	1	0	1	0	1	1	0

21. TEST - (Logical AND, affects only CPU FLAGS)
Syntax: TEST op1, op2

- It performs the bitwise logical AND of op1 and op2 but it wont 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
Syntax : SHL op1, op2
op1 = op1 «op2; //Equivalent C Statement

- SHL performs the bitwise left shift. op1 should be a registers/memory variables 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.

Example:
shl al,3

al	1	0	1	0	1	0	0	1
al«3	0	1	0	0	1	0	0	0

23. SHR - Shift Right
Syntax : SHR op1, op2
op1 = op1»op2; //Equivalent C Statement

- SHR performs the bitwise right shift. op1 should be a registers/memory variables 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.

Example:
shr al,3

al	1	0	1	0	1	0	0	1
al»3	0	0	0	1	0	1	0	1

24. ROL - Rotate Left
Syntax : ROL op1, op2
ROL performs the bitwise cyclic left shift. op1 could be a registers/memory variables but op2 must be an immediate(constant) value.
Example : rol al,3

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

25. ROR - Rotate Right
Syntax : ROR op1, op2
ROR performs the bitwise cyclic right shift. op1 could be a registers/memory variables but op2 must be an immediate(constant) value.
Example: 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
Syntax : 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
Syntax : 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. It decreases the value of ESP and copies the value of a register/constant into the system stack
    Syntax: PUSH register/constant
    Examples
    PUSH ax ; $ESP = ESP - 2$ and copies value of ax to [ESP]
    PUSH eax ; $ESP = ESP - 4$ and copies value of ax to [ESP]
    PUSH ebx
    PUSH dword 5
    PUSH word 258

29. POP
    Pops off a value from the system stack.POP Instruction takes the value stored in the top of system stack to a register and then increases the value of ESP.
    Examples:
    POP bx ; ESP= ESP + 2 and copies value from stack to bx
    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
    Pops off the value of all general purpose registers which we have pushed before using PUSHA 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.

---

34. 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

Comments in NASM

Only single line comments are available in NASM. A semicolon (;) is inserted in front of the line to be commented.
Ex: ; A program to find the largest of 2 numbers

NASM Installation

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.

Compilation

1. Assembling the source file
   nasm -f elf filename.asm
   This creates an object file, filename.o in the current working directory.

2. Creating an executable file
   For a 32 bit machine
   ld filename.o -o output filename

   For 64 bit machine
   ld -melf_i386 filename.o -o output filename

   This creates an executable file of the name output filename.

3. Program execution
   ./output filename

   For example, if the program to be run is first.asm
   nasm -f elf first.asm
   ld first.o -o output
   ./output

Chapter 3

Basic I/O in NASM

In this chapter, we will learn how to obtain user input and how to return the output to the output device. 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
    - This system call is used to exit from the program
    - 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.

(a) Reading a single digit number

```
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 [digit1]
```

(b) 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
int 80h                  omit enter key press as well

sub byte[digit1], 30h
sub byte[digit2], 30h

;Getting the number from ASCII
; 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

   - This system call can be used to print the output to the monitor
   - 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 sting resides

to ecx reg.

- We need to copy the number of characters in the string to edx reg.
- Then we will trigger INT 80h.

  See the below given example which is equivalent to the C statement printf()

  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 print.

Chapter 4

Introduction to Programming in NASM

Now that we have gone through all the basics required for creating a program, let's begin. Like all programming lessons, we will first learn how to create a Hello World program. Programming in NASM is easy if you understand how each construct is used and implemented. A transition from C like language can be frustrating at first, but let's take it step by step. We will learn to program by translating code snippets in C language to assembly code.

Hello World program

As discussed earlier, the whole program is divided into 3 sections, namely code section (section .text), section for uninitalised variables (section .bss) and section for initialised variables (section .data).

section .text is the place from where the execution starts in NASM program, analogous to the main() function in C-Programming.

1. Let's store the string "Hello World" into a variable named str. We are going to print that string as the output

   Pseudo Code:

   ```
   char str[13] = "Hello World";
   ```

   NASM Code:

   ```
   section .data                    ;For Storing Initialized Variables
   ```

```
string: db 'Hello World', 0Ah    ;String Hello World followed by a
newline character
length: equ 13                   ; Length of the string stored to a constant.
```

NB: Using equ we declare constants, ie. their value wonâĂŹt change during execution. $ -string will return the length of string variables in bytes (ie. number of characters)

2. Now we need to print this variable onto the screen. For that we use the write system call.

   Pseudo Code:

   ```
   printf("%s",str);
   ```

   NASM Code:

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

3. Now since we have printed the text, let's exit the program. We use the exit system call for the same

   Pseudo Code:

   ```
   return 0;
   ```

   NASM Code:

   ```
   ;Exit System Call
   mov eax, 1
   mov ebx, 0
   int 80h                 ; Length of the string stored to a constant.
   ```

4. section .text is the place from where the execution starts in NASM program, analogous to the main() function in C-Programming. So let's put the print and exit parts into the section

   The final program looks like this:

   ```
   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
newline character
length: equ 13                   ; Length of the string stored to a constant.
```

Compilation

1. Assembling the source file
   nasm -f elf filename.asm
   This creates an object file, filename.o in the current working directory.

2. Creating an executable file
   For a 32 bit machine
   ld filename.o -o output_filename

   For 64 bit machine
   ld -melf_i386 filename -o output_filename

   This creates an executable file of the name output filename.

3. Program execution
   ./output_filename

   For example, if the program to be run is first.asm
   nasm -f elf first.asm
   ld first.o -o output
   ./output

It doesn't seem as difficult as you felt in the beginning, right? In the same pattern, we'll see how both simple and complex constructs are formed in assembly language.

Chapter 5

Integer Handling

Integers are stored as characters in NASM. This chapter describes how to read and print integers. We have already skimmed through the basic operations in integer handling (add, sub, mul, div) in Chapter 3. Here, we will look at sample programs that will make use of these operations.

1. **Declaring an integer**

   ```
   section.bss
   var: resb/resw/resd 10
   ```

   Here we have reserved 10 bytes/words/double words for the memory location 'var'

2. **Reading a single digit**

   To read a single digit, we will read it as a single character and then subtract 30h from it (ASCII of 0 = 30h). This is to get the actual value of that number in that variable.

   ```
   mov eax, 3 ; Syscall number for read
   mov ebx, 0 ; Source Keyboard
   mov ecx, var ; Pointer to memory location 'var'
   mov edx, byte[size] ; Size of the input
   int 80h ; Triggering OS Interrupt
   sub byte[var], 30h ; Subtracting 30h to get integer value
   ```

   Note that the single digit taken as input here is stored in the first byte of 'var'. If 'var' was declared as a word/double word, the size of the input stored

in the edx register should be changed accordingly to word[size] or dword[size].

3. **Reading a multi-digit number**

   To read a multi-digit number, we read the input character by character (here, digit by digit) until we obtain the enter-key. Each digit is multiplied with 10 and added to the next digit to obtain the original number.

   - Pseudo Code

```
while(temp!=enterkey)
   num=0
   read(temp)
   num=numx10+temp
endwhile
```

   NASM Code:

```
read_num:
;;push all the used registers into the stack using pusha
pusha
;;store an initial value 0 to variable 'num'
mov word[num], 0

loop_read:
;; read a digit
mov eax, 3
mov ebx, 0
mov ecx, temp
mov edx, 1
int 80h
;;check if the read digit is the end of number, i.e, the enter-key whose ASCII
cmp byte[temp], 10
je end_read
mov ax, word[num]
mov bx, 10
mul bx
mov bl, byte[temp]
sub bl, 30h
mov bh, 0
add ax, bx
```

```
mov word[num], ax
jmp loop_read

end_read:
;;pop all the used registers from the stack using popa
popa
ret
```

4. **Printing a number**

   Unlike when we're reading a number, there is no end character like the enter-key marking the number's last digit. Here, we extract each digit from the number and push it to the stack. We also keep a count of the number of digits in the number. Using this count, the digits are popped out of the stack and printed in order.

   - Pseudo Code

```
count=0
while (num!=0)
   temp=num%10
   count++
   push(temp)
   num=num/10
endwhile
while(count!=0)
   temp=pop()
   print(temp)
   count- -
endwhile
```

   NASM Code:

```
print_num:
mov byte[count],0
pusha

extract_no:
cmp word[num], 0
je print_no
inc byte[count]
mov dx, 0
mov ax, word[num]
```

```
    mov bx, 10
    div bx
    push dx
    mov word[num], ax
    jmp extract_no

    print_no:
    cmp byte[count], 0
    je end_print
    dec byte[count]
    pop dx
    mov byte[temp], dl
    add byte[temp], 30h
    mov eax, 4
    mov ebx, 1
    mov ecx, temp
    mov edx, 1
    int 80h
    jmp print_no

    end_print:
    mov eax,4
    mov ebx,1
    mov ecx,newline
    mov edx,1
    int 80h
    ;;The memory location 'newline' should be declared with the ASCII key for new
    popa
    ret
```

Example Program:

Read 2 numbers and find their GCD.

Pseudo Code:

```
read(num1)
read(num2)
while(1)
   if(num1%num2==0)
   goto endloop
```

```
   temp=num1%num2
   num1=num2
   num2=temp
endwhile
endloop:
print(num2)
```

NASM Code

```
section .data
newline: db 10

section .bss
num1:resw 10
num2 resw 10
temp:resb 10
num:resw 10
nod:resb 10
count:resb 10

section .text
global _start
_start:

call read_num
mov cx,word[num]
mov word[num1],cx
call read_num
mov cx,word[num]
mov word[num2],cx
mov ax,word[num1]
mov bx,word[num2]

loop1:
mov dx,0
div bx
cmp dx,0
je end_loop
mov ax,bx
mov bx,dx
jmp loop1
```

```
end_loop:
mov word[num],bx
call print_num

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

Do not forget to add the read_num, print_num functions

Chapter 6

Subprograms

We have now covered the basics of programming in assembly language. Often we might have to repeat the same code in the program and that is when we use functions in languages like C. Let's see how it is done in assembly language.

The concept of subprograms can be used heavily in assembly language, since input and output operations require more than one line of code. Making them into subprograms not only makes the code smaller, but also helps you understand and debug the code better. It is highly reccommended that you start using subprograms in your code from the very start of programming in assembly language.

CALL & RET Statements:

In NASM, subprograms are implemented using the call and ret instructions. The general syntax is as follows:

```
;main code....
-------------
-------------

call function_name
-------------
-------------
;rest of the code....


function_name:            ;Label for subprogram
```

```
----------
----------
;function code....
----------
----------
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.
Now, let's see an example on how to use subprograms effectively to simplify your code. Below is a program which uses subprograms to calculate the sum of 10 input integers.

NB: *One major advantage of subprograms that you can exploit is by creating subprograms for input and output, as those components appear in almost all the programs you are going to write. As the codes are quite lengthy, repeating the code for each input and output will make your code even lengthier, making debugging and understanding the code a tedious task. In the program given below, three subprograms are used, one for input, one for output and the last one for calculating the sum. It is suggested that you use subprograms for input and ouput from now on.*

```
section .data
msg1: db 0Ah,"Enter a number"
size1: equ $-msg1
msg2: db 0Ah,"the sum= "
```

```
size2: equ $-msg2
newline: db 0Ah

section .bss
num: resw 1
sum: resw 1
digit1: resb 1
digit0: resb 1
digit2: resb 1
temp: resb 1
counter:resb 1
count: resb 1

section .text
global _start
_start:

mov byte[counter],0
mov byte[sum],0

read_and_add:

call read_num ;; Instead of the code for reading a number,
;; we just call the subprogram

call add

;loop condition checking

inc byte[counter]
cmp byte[counter],10
jl read_and_add


mov ax, word[sum]
mov word[num], ax
call print_num ;; Instead of the code for printing a number,
;; we just call the subprogram
```

```
;exit
mov eax,1
mov ebx,0
int 80h

;sub-program add to add a number to the existing sum
add:
mov ax,word[num]
add word[sum],ax
ret

;;subprogram to print a number

print_num:
mov byte[count],0
pusha
extract_no:
cmp word[num], 0
je print_no
inc byte[count]
mov dx, 0
mov ax, word[num]
mov bx, 10
div bx
push dx
mov word[num], ax
jmp extract_no
print_no:
cmp byte[count], 0
je end_print
dec byte[count]
pop dx
mov byte[temp], dl
add byte[temp], 30h
mov eax, 4
mov ebx, 1
mov ecx, temp
mov edx, 1
int 80h
jmp print_no
```

```
end_print:
mov eax,4
mov ebx,1
mov ecx,newline
mov edx,1
int 80h

;;The memory location 'newline' should be declared with the ASCII key for
;;new line in section.data.

popa
ret

;; subprogram for inputting a number

read_num:
;;push all the used registers into the stackusing pusha
pusha
;;store an initial value 0 to variable 'num'
mov word[num], 0
loop_read:
;; read a digit
mov eax, 3
mov ebx, 0
mov ecx, temp
mov edx, 1
int 80h
;;check if the read digit is the end of number,i.e, the enter-key whose ASCII key
cmp byte[temp], 10
je end_read
mov ax, word[num]
mov bx, 10
mul bx
mov bl, byte[temp]
sub bl, 30h
mov bh, 0
add ax, bx
mov word[num], ax
jmp loop_read
end_read:
```

```
;;pop all the used registers from the stackusing popa
popa
ret
```

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. A program to print fibonacci series up to a number using recursive subprogram is given in the appendix.

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 Cś 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 t
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

Chapter 7

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.

7.1 One Dimensional Arrays

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 though the elements of the array.

1. **Declaring/ Initializing an array**

```
section .bss
arr: resb/resw/resd 50              ;Declares an array of size 50
array1: db 2, 5, 8, 10, 12, 15      ;Initializes array
                                    ;{2,5,8,10,12,15}
array2: dw 191, 122, 165, 165       ;An array of 4 words
array3: dd 111, 111, 111            ;An array of 4 dwords
```

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
```

2. **Reading an n-size array**

Pseudo Code:

```
i=0
while(i<n)
   read(num)
   *(arr+i)=num
   i++
endwhile
```

NASM Code:

```
;;Initialization


mov ebx,array
mov eax,0          ;;Here eax is the counter


;;Function to read an array of n numbers

read array:
pusha
read loop:
cmp eax,dword[n]
je end read
call read num

;;read num stores the input in 'num'

mov cx,word[num]
```

```
mov word[ebx+2*eax],cx
inc eax

;;Here, each word consists of two bytes, so the counter should be
incremented by multiples of two. If the array is declared in bytes
do mov byte[ebx+eax],cx

end read:
popa
ret
```

3. **Printing an n-size array**

   Pseudo Code:

```
while(i<n)
   print *(arr+i)
   i++
endwhile
```

   NASM Code:

```
print array:
pusha

print loop:
cmp eax,dword[n]
je end print1
mov cx,word[ebx+2*eax]
mov word[num],cx

;;The number to be printed is copied to 'num'
before calling print num function

call print num
inc eax
jmp print loop
end print1:
popa
ret
```

   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.
```

Example Program

Read an array and find the average of the numbers in the array

```
section .data
space:db ' '
newline:db 10

section .bss
nod: resb 1
num: resw 1
temp: resb 1
counter: resw 1
num1: resw 1
num2: resw 1
n: resd 10
array: resw 50
matrix: resw 1
count: resb 10

section .text
global _start
_start:

call read_num
```

```
mov cx,word[num]
mov word[n],cx
mov ebx,array
mov eax,0
call read_array
mov ebx,array
mov eax,0
mov dx,0

loop1:
mov cx,word[ebx+2*eax]
add dx,cx
inc eax
cmp eax,dword[n]
jb loop1
mov ax,dx
mov bx,word[n]
mov dx,0
div bx
mov word[num],ax
call print_num
mov eax,4
mov ebx,1
mov ecx,newline
mov edx,1
int 80h
mov eax,0
mov ebx,array
call print_array

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

Do not forget to add the print_num, read_num, read_array, print_array functions.

The general syntax of using array offset is:

[basereg + factor *indexreg + constant]

basereg should be general purpose register containing the base address of the array.
factor: It can be 1, 2, 4 or 8.
indexreg: 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]
```

A sample program to search an array for an element (Traversal) is given in the appendix

7.2 Strings

Strings are stored in memory as array of characters. Each character in English alphabet has an 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.
Now let's get started to work with strings.

1. **Declaring/ Initializing a string**

   ```
   section .bss

   string: resb 50
   ```

2. **Reading a string**

   Pseudo Code:

   ```
   i=0
   while(num!='\n')
   read(num)
   *(arr+i)=num
   i++
   endwhile
   ```

NASM Code

```
read_array:

pusha

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

pop ebx
cmp byte[temp], 10   ;; check if the input is 'Enter'
je end_reading
inc byte[string_len]
mov al,byte[temp]
mov byte[ebx], al
inc ebx
jmp reading

end_reading:
;; Similar to putting a null character at the end of a string
mov byte[ebx], 0
mov ebx, string
popa
ret
```

3. **Printing a string**

Pseudo Code:

```
while(*(arr+i)!='\n')
print(*(arr+i))
i++
endwhile
```

NASM Code:

```
print_array:
pusha
mov ebx, string

printing:
mov al, byte[ebx]
mov temp, al
cmp byte[temp], 10    ;; checks if the character is NULL character
je end_printing

push ebx
mov eax, 4
mov ebx, 1
mov ecx, temp
mov edx, 1
int 80h
pop ebx
inc ebx
jmp printing

end_printing:
popa
ret
```

Sample Program - To count the number of 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
```

7.3 Two-Dimensional Array / Matrices

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 we first store 1st row, then the 2nd, and so on. In column major form we store the 1st column, then the 2nd, and so 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]

a) 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

1. **Declaration/ Initialization**

```
section .bss
num: resw 1       ;Number to be read/printed
nod: resb 1       ;For storing the number of digits
temp: resb 2
```

```
matrix1: resw 200
m: resw 1          ;Number of rows m
n: resw 1          ;Number of columns n
i: resw 1
j: resw 1

section .data
tab: db 9          ;ASCII for vertical tab
new line: db 10    ;ASCII for new line
```

2. **Read elements into a matrix**

   Pseudo Code:

```
read(m)
read(n)
i=0
k=0
while(i<m)
   j=0
   while(j<n)
     read(num)
     *(matrix+k)=num
     ++j
     ++k
   endwhile
   ++i
endwhile
```

   NASM Code:

```
section .text
global _start
_start:

mov ecx, 0
call read_num
mov cx, word[num]
mov word[m], cx
;Reads the number of rows and stores it in m

mov ecx, 0
```

```
call read_num
mov cx, word[num]
mov word[n], cx
;Reads the number of columns into n

mov eax, 0
mov ebx, matrix1
mov word[i], 0

i_loop:
mov word[j], 0
j_loop:
call read_num
mov dx , word[num]
mov word[ebx + 2 * eax], dx

;eax will contain the array index and each element is 2 bytes(1 word) long

inc eax
inc word[j]
mov cx, word[j]
cmp cx, word[n]
jb j_loop

inc word[i]
mov cx, word[i]
cmp cx, word[m]
jb i_loop
```

3. **Print elements from a matrix**

   Pseudo Code:

```
i=0
k=0
while(i<m)
   j=0
   while(j<n)
      num=*(matrix+k)
      print(num)
      ++j
      ++k
```

```
   endwhile
   ++i
endwhile
```

NASM Code:

```
mov eax, 0
mov ebx, matrix1
mov word[i], 0

i_loop2:
mov word[j], 0
j_loop2:

;eax will contain the array index and each element is 2 bytes(1 word) long

mov dx,word[ebx + 2 * eax]
mov word[num],dx
call print_num
;Printing a space after each element
pusha
mov eax, 4
mov ebx, 1
mov ecx, tab
mov edx, 1
int 80h
popa
inc eax
inc word[j]
mov cx, word[j]
cmp cx, word[n]
jb j_loop2

pusha
mov eax, 4
mov ebx, 1

mov ecx, new_line
mov edx, 1
int 80h
popa
```

```
inc word[i]
mov cx, word[i]
cmp cx, word[m]
jb i_loop2

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

Don't forget to add the necessary sub-functions for the program

A program to read and print an m X n matrix is given in the appendix

7.4 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
```

$LODSW$
$AX = word[DS : ESI]$
$ESI = ESI\pm; 2$

$LODSD$
$EAX = dword[DS : ESI]$
$ESI = ESI\pm; 4$

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
   ```

   $STOSW$
   $word[ES : EDI] = AX$
   $EDI = EDI\pm; 2$

   $STOSD$
   $dword[ES : EDI] = EAX$
   $EDI = EDI\pm; 4$

   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 $\pm; 1$
   $EDI = EDI \pm; 1$

   $MOVSW$
   $word[ES : EDI] = word[DS : ESI]$

$ESI = ESI\pm;2$
$EDI = EDI\pm;2$

$MOVSD$
$dword[ES:EDI] = dword[DS:ESI]$
$ESI = ESI\pm;4$
$EDI = EDI\pm;4$
$Eg: Programtocopyelementsofanarraytoanother$

```
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
.............................................
.............................................
.............................................
```

REP - Repeat String Instruction

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
.................................................
.................................................
.................................................
```

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
$EDI = EDI \pm 1$

$CMPSW$
$Compare sword[DS : ESI] with word[ES : EDI]$
$ESI = ESI \pm 2$
$EDI = EDI \pm 2$

$CMPSD$
$Compare sdword[DS : ESI] with dword[ES : EDI]$
$ESI = ESI \pm 4$
$EDI = EDI \pm 4$

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

$SCASW$
$Comparesvalueof AX with word[ES : EDI]$
$EDI = EDI \pm 2$

$SCASD$
$Comparesvalueof EAX with dword[ES : EDI]$
$EDI = EDI \pm 4$

$Eg : Scanninganarrayforanelement$

$section.data$
$array1 : db1, 5, 8, 12, 13, 15, 28, 19, 9, 11$

$section.text$
$globalstart$

$start :$
$CLD; ClearstheDirectionFlag$
$movedi, array1$
$; CopyBaseaddressofarraytoindexregisters$
$movecx, 10$
$; No : ofelementinthearray$
$moval, 15$
$; Valuetobesearched$
$scan :$
$SCASB$
$jefound$
$loopscan$
$jmpnotfound$

Chapter 8

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.

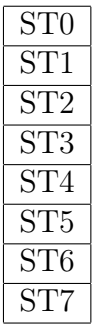
Single Precision	1 bit	8 bits	32bits
Double Precision	1 bit	11 bits	52bits
	S	Exponent	Fraction

S - Sign Bit. If the number is -ve then $S = 1$ else $S = 0$.
The number
$f = (-1)Sx(1 + 0.Fraction) \times 2Exponent - Bias$
$1.0 \leq |Significand| < 2.0$

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. The 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 coprocessor 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.



Floating Point Stack

C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1	C0

Floating Point Control Word

Floating Point Instructions:

Load / Store Instructions:

FLD src : Loads the value of src on to the top of Floating Point stack(ie. ST0). src should be either a floating point register or a single precision/double precision/extended precision floating point 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 Floating Point Stack.

FLDZ : Stores 0 to the top of Floating Point Stack.

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.
$ST0 \longleftrightarrow STn$

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 : $ST0 = (float)src - ST0$,

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 : $ST0 = ST0imes(float)src$, 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 : 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 ÏĂ 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.

Inorder to use C-Funtions make sure to have gcc and gcc-multilib installed
Compile using:

```
nasm -f elf file.asm
gcc -m32 -o file file.o
```

Run with ./file

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
```

```
readnat:
push ebp
mov ebp, esp
sub esp , 2
lea eax , [ebp-2]
push eax
push format3
call scanf
mov ax, word[ebp-2]
mov word[num], ax
mov esp, ebp
pop ebp
ret

main:

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

call readnat

mov ax, word[num]
mov word[num2], ax

fldz
reading:
call read
fadd ST1
dec word[num]
cmp word[num], 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
format3: db "%d", 0
msg1: db "Enter the number of floating numbers : "
len1: equ $-msg1

section .bss
num: resw 1
num2: resw 1
```

Eg: Program to find the circumference of a circle of a given radius:

```
section .data
mes1: db "Enter Radius :"
len1: equ $-mes1
mes4: db "Perimeter is :"
len4: equ $-mes4
format1: db "%lf", 0
format2: db "%lf", 10
float2: dq 2
newline: dw 10

section .bss
float1: resq 1

section .text
global main:
extern scanf
extern printf

main:

mov eax, 4
```

```
mov ebx, 1
mov ecx, mes1
mov edx, len1
int 80h

call read_float

fstp qword[float1]

fldpi
fld qword[float1]
fild qword[float2]
fmul ST1
fmul ST2

mov eax, 4
mov ebx, 1
mov ecx, mes4
mov edx, len4
int 80h

call print_float

ffree ST0
ffree ST1
ffree ST2

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

;;;;;;;;;;;;;;;function to read floating point

read_float:
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

;;;;;;;;;;;;;;;function to print floating point

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

;;;;;;;;;;;;;;;function to print newline

print_newline:
pusha
mov eax, 4
mov ebx, 1
mov ecx, newline
mov edx, 1
int 80h
popa
ret
```

Appendix

Some errors that we often make

- Forgetting exit system call
  We may sometime find that we have written the program perfectly and it is giving correct output except that it shows a âĂIJSegmentation faultâĂİ in the end.
  This can be because we have forgotten to give an exit system call at the end. It's important to tell the operating system exactly where it should begin execution and where it should stop. The program on continuing sequential execution of the next address in memory, it could have encountered anything. We don't know what the kernel tried to execute but it caused it to choke and terminate the process for us instead - leaving us the error message of 'Segmentation fault'. Calling exit system call at the end of all our programs will mean the kernel knows exactly when to terminate the process and return memory back to the general pool thus avoiding an error.

- Adding and subtracting 30h
  We often find that the certain symbols are displayed instead of numbers that should have been printed. There is also a possibility that the input used is wrong in the program.
  This can be a problem of ASCII conversion. The numbers we type through keyboard are stored as ASCII values. 30h (in hexadecimal or 48 in decimal) is the ASCII code for a digit '0'. Then 31h, 32h, ... 39h correspond to '1', '2',... '9'. So to get the exact number for calculation must subtract 30h from the input. When we want to display it on the screen we add 30h so that the number is converted to its corresponding ASCII value.

- Declaration and initialisation of string and its length one after the other

It is always better to declare the length of the string which is already defined beneath the string. Otherwise when we have declarations and initialisation of more than one string we may find that some of the strings are displayed more than once.
When the strings and its length are not given one after the other in section .data, the compiler executes every statements between the string initialisation and length initialisation. Even if some of the strings may not be asked to be displayed but if they are initialised between another string and its length initialisation, they are displayed. Therefore make sure nothing is declared between string and its length declaration. Try the following code and correct it accordingly.

```
section .text \\
global \_start \\
\_start: \\

mov eax,4 \\
mov ebx,1 \\
mov ecx,a \\
mov edx,alen \\
int 80h \\

mov eax,4 \\
mov ebx,1 \\
mov ecx,b \\
mov edx,blen \\
int 80h \\

mov eax,4 \\
mov ebx,1 \\
mov ecx,c \\
mov edx,clen \\
int 80h \\

mov eax,1 \\
mov ebx,0 \\
int 80h \\

section .data \\
c: db 'hi ' \\
```

```
a: db 'hello ' \\
b: db 'world' \\
blen: equ \$ -b \\
clen: equ \$ -c \\
alen: equ \$ -a \\
```

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 specically 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. In this stage, the values are read from the source registers.
- Execute : It will perform the operations on the operands and manipulates the result.
- Memory Access : The memory is accessed at this stage (read/ write).
- Write : In this stage result is written back to the destination register.

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 firstclock 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.

Program to print Fibonacci series

Example program: Printing fibonacci series up to a number using recursive sub-progam

```
section .data

msg1:   db 0Ah,"Enter the number"
size1:  equ $-msg1
msg2:   db 0Ah,"fibonacci series = "
size2:  equ $-msg2
space:  db " "

section .bss

num:  resb 1
sum:  resb 1
digit1:  resb 1
digit0:  resb 1
fiboterm1:  resb 1
fiboterm2:  resb 1
temp:  resb 1

section .text

global _start

_start:

;Reading the number up to which fibonacci series has to be printed

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

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 bl,10
mul bl
add al,byte[digit0]
mov byte[num],al

mov byte[fiboterm1],0        ;first fibonacci term
mov byte[fiboterm2],1        ;second fibonacci term

mov eax,4
mov ebx,1
mov ecx,msg2
mov edx,size2
int 80h

;printing the first fibonacci term

movzx ax,byte[fiboterm1]
mov bl,10
div bl
mov byte[digit1],al
mov byte[digit0],ah

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

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

;printing the second fibonacci term

movzx ax,byte[fiboterm2]
mov bl,10
div bl
mov byte[digit1],al
mov byte[digit0],ah

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

mov eax,4
```

```
mov ebx,1
mov ecx,space
mov edx,1
int 80h

;calling recursive sub-program fibo to print the series

call fibo

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

fibo:                       ;sub program to print fibonacci series

mov al,byte[fiboterm1]
mov bl,byte[fiboterm2]
add al,bl
mov byte[temp],al

cmp al,byte[num]
jng  contd

end:
ret

contd:

;printing the next fibonacci term

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

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

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

movzx ax,byte[fiboterm2]
mov byte[fiboterm1],al
movzx bx,byte[temp]
mov byte[fiboterm2],bl

call fibo

jmp end
```

Traversal Program

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
jg reading

;Comparing loop variable

;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]

;al now contains the number
pop ebx
mov byte[ebx], al
add ebx, 1
dec byte[temp]
cmp byte[temp], 0
jg reading

;Comparing loop variable

;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
```

Program to read and print m * n matrix

Example program: reading and displaying an m x n matrix

```
section .bss
num: resw 1 ;For storing a number, to be read of printed....
nod: resb 1 ;For storing the number of digits....
temp: resb 2
matrix1: resw 200
m: resw 1
```

```
n: resw 1
i: resw 1
j: resw 1


section .data
msg1: db "Enter the number of rows in the matrix : "
msg_size1: equ $-msg1
msg2: db "Enter the elements one by one(row by row) : "
msg_size2: equ $-msg2
msg3: db "Enter the number of columns in the matrix : "
msg_size3: equ $-msg3
tab: db  9 ;ASCII for vertical tab
new_line: db 10 ;ASCII for new line

section .text
global _start

_start:
mov eax, 4
mov ebx, 1
mov ecx, msg1
mov edx, msg_size1
int 80h

mov ecx, 0

;calling sub function read num to read a number
call read_num
mov cx, word[num]
mov word[m], cx

mov eax, 4
mov ebx, 1
mov ecx, msg3
mov edx, msg_size3
int 80h

mov ecx, 0
call read_num
```

```
mov cx, word[num]
mov word[n], cx

mov eax, 4
mov ebx, 1
mov ecx, msg2
mov edx, msg_size2
int 80h

;Reading each element of the matrix........
mov eax, 0
mov ebx, matrix1

mov word[i], 0
mov word[j], 0

i_loop:
mov word[j], 0

j_loop:

call read_num
mov dx , word[num]

;eax will contain the array index and each element is 2 bytes(1 word) long
mov word[ebx + 2 * eax], dx
inc eax   ;Incrementing array index by one....

inc word[j]
mov cx, word[j]
cmp cx, word[n]
jb j_loop

inc word[i]
mov cx, word[i]
cmp cx, word[m]
jb i_loop

;Printing each element of the matrix
mov eax, 0
```

```
mov ebx, matrix1

mov word[i], 0
mov word[j], 0

i_loop2:
mov word[j], 0

j_loop2:

;eax will contain the array index and each element is 2 bytes(1 word) long
mov  dx, word[ebx + 2 * eax]   ;
mov word[num] , dx
call print_num

;Printing a space after each element.....
pusha
mov eax, 4
mov ebx, 1
mov ecx, tab
mov edx, 1
int 80h

popa

inc eax

inc word[j]
mov cx, word[j]
cmp cx, word[n]
jb j_loop2

pusha
mov eax, 4
mov ebx, 1
mov ecx, new_line
mov edx, 1
int 80h

popa
```

```
inc word[i]
mov cx, word[i]
cmp cx, word[m]

jb i_loop2

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

;Function to read a number from console and to store that in num

read_num:

pusha
mov word[num], 0

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

cmp byte[temp], 10
je end_read

mov ax, word[num]
mov bx, 10
mul bx
mov bl, byte[temp]
sub bl, 30h
mov bh, 0
add ax, bx
mov word[num], ax
jmp loop_read
end_read:
```

```
popa

ret

;Function to print any number stored in num...
print_num:
pusha
extract_no:
cmp word[num], 0
je print_no
inc byte[nod]
mov dx, 0
mov ax, word[num]
mov bx, 10
div bx
push dx
mov word[num], ax
jmp extract_no

print_no:
cmp byte[nod], 0
je end_print
dec byte[nod]
pop dx
mov byte[temp], dl
add byte[temp], 30h


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

jmp print_no

end_print:
popa
ret
```