1-Basics

*printing hello world

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

_start:
                                mov eax, 4
                                mov ebx, 1
                                mov ecx, string
                                mov edx, length
                                int 80h
; Using int 80h to implement write() sys_call
;Exit System Call
                                mov eax, 1
                                mov ebx, 0
                                int 80h

section .data
string: db 'Hello World', 0Ah                                              ;newline
character
length: equ $-string                                                          ;For
Storing Initialized Variables

;String Hello World followed by a

; Length of the string stored to a constant
```

Running the code

1.)Assembling the source file
nasm -f elf filename.asm

2.)For a 32 bit machine
ld filename.o -o output_filename

3.)program execution
./output_filename

push and pop operations

```

push ebx
;pushing ebx into the stack
;then decrementing the stack pointer(ESP) by 4
;the value of ebx is not changed

pop edx
;poping the value from the stack and putting it into edx
```

```
;then incrementing the value of ESP by 4
```

2-Arithmetic operations

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).

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.

3-procedures or subprograms

Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.
Procedures is the name given for the call return function it acts similar to calling a function in a main function but any changes made to memory and registers (not necessarily)in the called function is reflected in main function.

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.

Pushes all general purpose registers onto the stack in the following order: (E)AX, (E)CX, (E)DX, (E)BX, (E)SP, (E)BP, (E)SI, (E)DI.
***The value of SP is the value before the actual push of SP**

The ret instruction transfers control to **the return address located on the stack**.
This address is usually placed on the stack by a call instruction
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

read_num

```
read_num:
                        pusha
                        mov word[num],0
readloop:
                        mov eax,3
                        mov ebx,1
                        mov ecx,temp
                        mov edx,1
                        int 0x80

                        mov cl,byte[temp]
                        cmp cl,10
                        je end_loop
                        cmp cl,32
                        je end_loop

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

```
jmp readloop

end_loop:
popa
ret
```

print_num

```
print_num:
                pusha
                mov word[nod],0
        cmp word[num],0
        je print_zero
extract_num:

                cmp word[num],0
                je print_loop

                mov bx,10
                mov ax,word[num]
                mov dx,0
                div bx
                push dx
                mov word[num],ax
                inc word[nod]

                jmp extract_num

print_loop:
                pop dx

                mov byte[temp],dl
                add byte[temp],30h

                mov eax,4
                mov ebx,1
                mov ecx,temp
                mov edx,1
                int 0x80

                dec word[nod]
                mov cx,word[nod]

                cmp cx,0
                je exit_print

                jmp print_loop

exit_print:
                popa
                ret
```

```
print_zero:
                mov eax,4
                mov ebx,1
                mov ecx,'0'
                mov edx,1
                int 0x80

                jmp exit_print
```

4-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.

Accesing array elements
address of a memory location is always 32 bits so we first store the address of the declared array in 32 bit registers(eg EAX).
mov eax,array;

The label which we use to create array(eg: 'array')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.

To access an element at the "*i*"th index we add (i x (size))+base_address
array[5]
To access the element at fifth index we can use the following commands
mov ebx,array
mov eax,5
//we are storing number of size 2 in the array so size=2;
mov cx,word[ebx+2 * eax] ;we use word since the number stores at that memory location has a size of 2 bytes

Reading an array

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

We the perform the same operation in nasm by calling the subprogram read_array

```

;;Here eax is the counter
;;Function to read an array of n numbers
read_array:
                mov ebx,array
                mov eax,0
                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
        jmp read_loop

;;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
```

Printing array

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

The nasm procedure for this is

```

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
```

Strings

read_string

```
read_string:
                pusha
                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
            ;; 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
```

print_string

```
print_array:
            pusha
            mov ebx, string

printing:
            mov al, byte[ebx]
            mov byte[temp], al
            cmp byte[temp], 0
            je end_printing
            ;; checks if the character is NULL character
            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
```

5-2D matrix

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**
But we mostly use row major form.

Declararion is same as that of arrays.

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

Reading matrix

```
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
```

The matrix representation is read in nasm as

```
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
```

Printing matrix

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

The matrix can be printed using nasm

```
                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
```

6-sample program

```
void insert(int a[], int n) /* function to sort an aay with insertion sort */
{
    int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;

        while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahead from their current position*/
        {
             a[j+1] = a[j];
             j = j-1;
         }
         a[j+1] = temp;
     }
 }
```

```
section .data

array1 : dw 12,67,34,5,30,45,22
length : dd 7
space : db " "


section .bss

num :resw 1
temp:resb 1
nod :resw 1
key:resw 2
i: resw 1
j: resw 1

section .text
global _start
_start:
    mov ebx,array1
    mov word[i],0

i_loop:
   inc word[i]
   mov ebx,array1
   mov cx,word[length]
   cmp cx,word[i]
   je endloop
```

```
    mov ecx,0
    mov cx,word[i]

    mov eax,0
    add ax,cx
    mov cx,word[ebx+2*eax]
    mov word[key],cx
    mov cx,word[i]
    mov word[j],cx;
    dec word[j]

    movzx eax,word[j]

j_loop:
    mov cx,word[key]
    cmp word[ebx+2*eax],cx
    jl new

    mov cx,word[ebx+2*eax]
    mov word[ebx+2*eax+2],cx
    dec eax
    dec word[j]

    cmp word[j],0xff
    jne j_loop
    cmp word[j],0
    jge j_loop

new:
    mov cx,word[key]
    mov [ebx+2*eax+2],cx

    jmp i_loop

endloop:

    call print_array

    mov eax,1
    mov ebx,1
    int 0x80




print_array:
        pusha
        mov ebx,array1
        mov eax,0
printloop:
        cmp ax,word[length]
        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

        pusha
        mov eax,4
        mov ebx,1
        mov ecx,space
        mov edx,1
        int 0x80
        popa

        inc eax
        jmp printloop
end_print1:
    popa
    ret


print_num:
                pusha
                mov word[nod],0
        cmp word[num],0
        je print_zero
extract_num:

                cmp word[num],0
                je print_loop

                mov bx,10
                mov ax,word[num]
                mov dx,0
                div bx
                push dx
                mov word[num],ax
                inc word[nod]

                jmp extract_num

print_loop:
                pop dx

                mov byte[temp],dl
                add byte[temp],30h

                mov eax,4
                mov ebx,1
                mov ecx,temp
                mov edx,1
                int 0x80

                dec word[nod]
```

```
            mov cx,word[nod]

            cmp cx,0
            je exit_print

            jmp print_loop

exit_print:
            popa
            ret

print_zero:
            mov eax,4
            mov ebx,1
            mov ecx,'0'
            mov edx,1
            int 0x80

            jmp exit_print
```

7-Using Debugger in NASM

1.)Assembling the source file
nasm -f elf filename.asm

2.)For a 32 bit machine
ld filename.o -o output_filename

3.)program execution
./output_filename

To open the debugger we type the command
gdb output_filename

//If your code was giving a segmentation fault,to see when the code crashes type the command
run
//this will show the label of the index where your program has crashed.

break labelname

info all-registers

disassemble

set disassembly-flavor Copy intel

8-Array & String operations

These operations give a much easier way to manipulate strings and arrays in nasm

We use indexing registers **ESI and EDI** to store the base address of the array and increment/decrement either one or both of the index register's value.

Depending on the value of Direction Flag(DF) it either in-
crements or decrements the index register’s value. The following instructions are
used to set the value of DF manually
CLD :-clears the DF(DF to 0),then string instructions will increment the indexing registers.
STD :-sets the DF to 1,then string insructions will decrement the indexing registers.

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

ESI stores the base address of the source array.
EDI stores the base address of the destination array.

1.)LODS

Reading an array element to reg.

LODSB
AL = byte[DS:ESI]
ESI = ESI ±1

LODSW
AX = word[DS : ESI]
ESI = ESI±2

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

2.)STOS

Storing a reg to an array.

STOSB
byte[ES:EDI] = AL
EDI = EDI ±1

STOSW
word[ES : EDI] = AX
EDI = EDI±2

STOSD
dword[ES : EDI] = EAX

EDI = EDI±4

3.)MOVS

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

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

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

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

4.)REP

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

5.) CMPS

Compares two array elements and affects the CPU Flags just like CMP instruction.

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

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

CMPSD
Compares dword[DS : ESI]with dword[ES : EDI]

ESI = ESI ± 4
EDI = EDI ± 4

6.)SCAS

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 AXwithword[ES : EDI]
EDI = EDI ± 2

SCASD
Comparesvalueof EAXwithdword[ES : EDI]
EDI = EDI ± 4