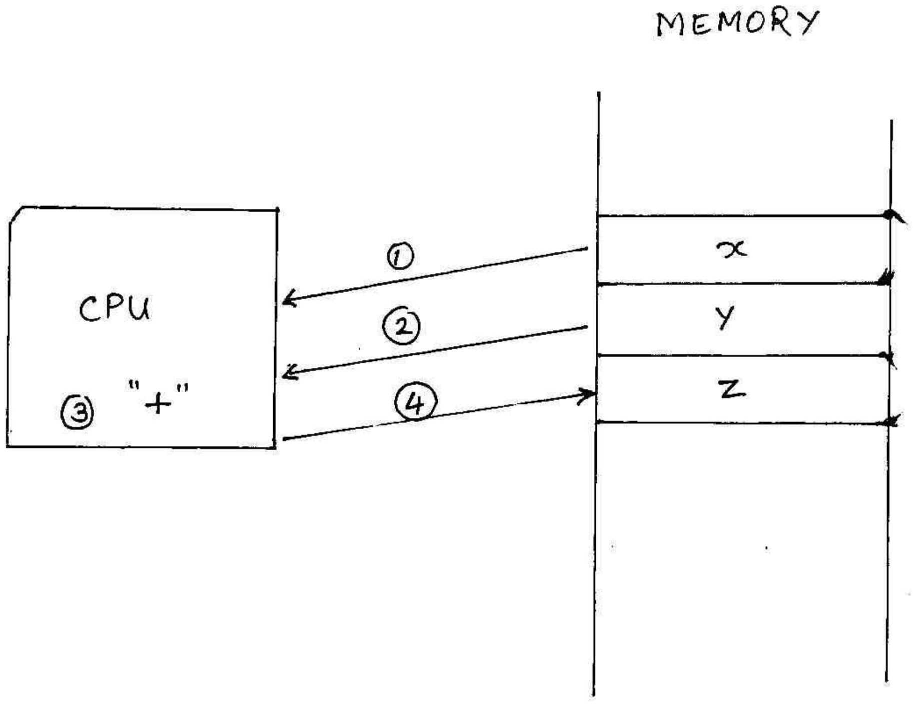
START

Look at the following C program. It adds two numbers, x and y, and stores the result in z.

```
#include <stdio.h>

void main( ) {

   int x=5, y=9, z;

   z  =  x + y;

}
```

- The above program looks simple, just a one-line operation.
- Actually we can break it down into a number of steps:
  - Fetch the value of x from memory
  - Fetch the value of y from memory
  - Add x and y
  - Store x and y in the memory location corresponding to z.
- Look at the diagram below.

- Each step mentioned above can be written into an assembly program as shown on the next page.
- Think of assembly language as a C program broken into smaller steps


MEMORY
CPU
③ "+"
①
②
④
x
y
z

; **<u>My first Assembly program</u>**

; No need to include stdio.h, because we are not doing any i/p or o/p

```
; prog1
.model small
.stack 200
.data
        X db 05h    ;
        Y db 09h    ;
        Z db ?              ; the result has yet to be calculated

.code
start:

        ; Each line below is an "assembly instruction"

        mov al, X   ; al ← X
        mov bl, Y   ; bl ← Y
        add al, bl  ; al ← X+Y or  X+=Y  or al += bl
        mov Z, al   ; Z ← al (X+Y)

        ; Program must end with these 3 lines
        mov ah, 4ch         ;
        int 21h
        end start
```

NOTES:

- Notice the syntax – no operators, only words
- Notice the program variables:- strange names like al, bl, ah etc…
- What are .model, .stack 200, .data, and .code? (directives)
- What is db, and "db ?" ?
- What are start and end start? (labels)
- What are the assembly instructions used here?   mov, add, int 21h
- How do you write comments in assembly?

; **Prog1 revisited using “dw”**

```
; prog1b
.model small
.stack 200
.data
        X db 05h    ;
        Y db 09h    ;
        Z  db ?             ; the result has yet to be calculated

        X2 dw 05h
        Y2 dw 09h
        Z2 dw  ?


.code
start:
        mov ax, X   ; ax ← X
        mov bx, Y   ; bx← Y
        add ax, bx  ; ax ← X+Y or  X+=Y  or ax += bx
        mov Z, ax   ; Z ← ax (X+Y)

        mov ah, 4ch         ; program must end with the follwing 3 lines:-
        int 21h

        end start
```

```
; Prog1 revisited using "sub"

; prog1c
.model small
.stack 200
.data
        X dw 05h   ;
        Y dw 09h   ;
        Z  dw  ?   ; the result has yet to be calculated

.code
start:
        mov ax, X  ; ax ← X
        mov bx, Y  ; bx← Y
        sub ax,    bx     ; ax ← X-Y or  X-=Y  or ax -= bx
        mov Z, ax  ; Z ← ax (X+Y)

        mov ah, 4ch       ; program must end with the follwing 3 lines:-
        int 21h

        end start
```

; **Prog1 revisited using “adc”**

- ADC used for multi-byte addition, i.e. 16bit, 24 bit, 32bit nos.
- Following example adds 2-byte(16bit) nos.
- First no is X: Xlo is lower byte, Xhi is upper byte
- Second number is Y: Ylo is lower byte, Yhi is upper byte
-

```
; prog1d
.model small
.stack 200
.data
        Xlo db 99h
        Xhi db 01h
        Ylo db 12h
        Yhi  db 34h
        Zlo db   ?    ; the result has yet to be calculated
        Zhi db ?
        (0199h + 1234h = 13cdh )

.code
start:
        mov al, Xlo
        mov bl, Ylo
        add al, bl    ; al = al + bl
        mov Zlo, al
        mov al, Xhi
        mov bl, Yhi
        adc al, bl
        mov Zhi, al

        mov ah, 4ch        ; program must end with the following 3 lines:-
        int 21h

        end start
```

; **Prog1 revisited using** “**sub, subb**”

- ADC used for multi-byte addition, i.e. 16bit, 24 bit, 32bit nos.
- Following example adds 2-byte(16bit) nos.
- First no is X: Xlo is lower byte, Xhi is upper byte
- Second number is Y: Ylo is lower byte, Yhi is upper byte
-

```
; prog1e
.model small
.stack 200
.data
        Xlo db 12h
        Xhi db 34h
        Ylo db 99h
        Yhi  db 01h
        Zlo db   ?   ; the result has yet to be calculated
        Zhi db ?
        (0199h - 1234h = 13cdh )

.code
start:
        mov al, Xlo
        mov bl, Ylo
        sub al, bl    ; al = al + bl
        mov Zlo, al
        mov al, Xhi
        mov bl, Yhi
        subb al, bl
        mov Zhi, al

        mov ah, 4ch        ; program must end with the following 3 lines:-
        int 21h

        end start
```

My Second C program – hello, world

```
; Prog 2
#include <stdio.h>

void main( ) {

printf(“Hello, world!\n”);

}
```

The corresponding Assembly program is shown below:

<u>; My second Assembly program</u>

```
; No need to include stdio.h, because we are not doing any i/p or o/p

.model small
.stack 200
.data
        Msg db "Hello, world!", 13,10, '$'

        ; The above line defines a character string.
        ; Msg is the name of the string.
        ; db stands for "define byte" – it reserves a memory byte for
every
        ; character following the word "db"
        ; Character string must be terminated with a '$' sign.
        ; 13 and 10 are ASCII codes for CR '\r' and LF '\lf' characters.

.code

start:
        lea dx, msg1            dx: pointer to the string "msg"
        mov ah, 09h             ah = function number
        int 21h                 actual function call

        mov ah, 4ch             ah = function number
        int 21h                 actual function call

        end start
```

<u>NOTES</u>:

- New instructions – lea
- What is 13, 10
- What is '$'
- What is mov ah, 09h
- What is int 21h

ASCII CODES

- When we use a number key or a letter key, that number or letter is displayed on the screen.
- There are many other keys on the keyboard (total 101 keys on std fullsize keybrd).
  - Special chars like $, %, ^, (,), {,], ”, …
  - Function keys like F1, F2…F12
  - Enter, Shift, Cntrl, Alt, Delete, Backspace …

  How do we tell the display that delete key is pressed?
- Each key on the keyboard is given a unique 8-bit code called ASCII code. This 8-bit code is sent to the display.

Important ASCII codes:

• Numbers 0-9:	30H – 39H
• Lowercase letters a-z:	61H – 7BH
• Uppercase letters A-Z:	41H – 5BH
• CR:	0DH
• LF:	0AH
• ESC:	1BH
• SPACE:	
• TAB:	

In the following program we have defined a character string:

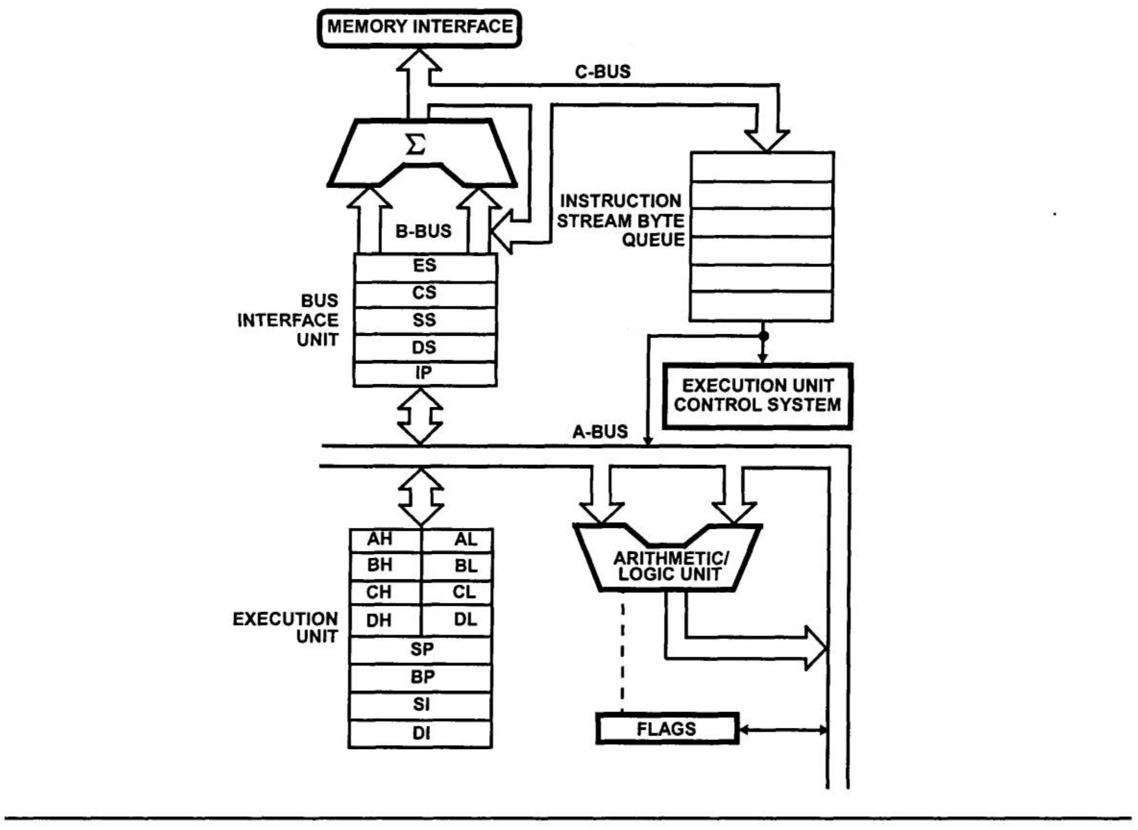
```
.data
Msg1 db “Hello, World”, 0dh,0ah, ‘$’
```

- Here each letter is stored by its ASCII code
- 0dh is ascii for CR
- 0ah is ascii for LF
- Together 0dh and 0ah represent “\n” in C
- In assembly a char struing is terminated by a “$’ sign , so ‘$’ is the ascii code for the $ sign.

00		01		02		03	♥	04	♦	05	♣	06	♠	07		08		09		0A		0B		0C		0D		0E		0F	
10		11		12		13		14		15		16		17		18		19		1A		1B		1C		1D		1E		1F	
20		21	!	22	"	23	#	24	$	25	%	26	&	27	'	28	(	29	)	2A	*	2B	+	2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7	38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G	48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W	58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g	68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w	78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	
80	ç	81	ü	82	é	83	â	84	ä	85	à	86	å	87	ç	88	ê	89	ë	8A	è	8B	ï	8C	î	8D	ì	8E	Ä	8F	Å
90	É	91	æ	92	Æ	93	ô	94	ö	95	ò	96	û	97	ù	98	ÿ	99	ö	9A	ü	9B	¢	9C	£	9D	¥	9E	₧	9F	ƒ
A0	á	A1	í	A2	ó	A3	ú	A4	ñ	A5	Ñ	A6	a̲	A7	o̲	A8	¿	A9	⌐	AA	¬	AB	½	AC	¼	AD	¡	AE	<	AF	>
B0	░	B1	▒	B2	▓	B3	│	B4	┤	B5	╡	B6	╢	B7	╖	B8	╕	B9	╣	BA	║	BB	╗	BC	╝	BD	╜	BE	╛	BF	┐
C0	└	C1	┴	C2	┬	C3	├	C4	─	C5	┼	C6	╞	C7	╟	C8	╚	C9	╔	CA	╩	CB	╦	CC	╠	CD	═	CE	╬	CF	╧
D0	╨	D1	╤	D2	╥	D3	╙	D4	╘	D5	╒	D6	╓	D7	╫	D8	╪	D9	┘	DA	┌	DB	█	DC	▄	DD	▌	DE	▐	DF	▀
E0	α	E1	ß	E2	Γ	E3	π	E4	Σ	E5	σ	E6	µ	E7	γ	E8	Φ	E9	Θ	EA	Ω	EB	δ	EC	∞	ED	\	EE	ε	EF	∩
F0	≡	F1	±	F2	≥	F3	≤	F4	⌠	F5	⌡	F6	÷	F7	~	F8	[	F9	.	FA	.	FB	√	FC	ⁿ	FD	²	FE	■	FF	

Table A-1 ASCII Character Set

INTERNAL ARCHITECTURE OF 8086



3-143

The BIU (Bus Interface Unit):

It is responsible for all memory accesses and I/O port accesses by the 8086. In other words, BIU communicates with devices outside the CPU.
It also fetches instructions from the memory.

The BIU consists of the IP, the Segment registers, and adder to calculate a 20 bit address and the instruction prefetch queue.

The EU(Execution Unit);

The EU is responsible for executing instructions fetched by the BIU.

It consists of the Instruction Decoder, Control unit, Arithmetic Logic unit (ALU), general purpose registers, flag register and some other registers.

Register Set of 8086

- They are places inside the CPU where operands can be stored.
- Some registers are 8-bit, others are 16 bit. (From 80386 onwards we have 32bit registers also.)
- Together, they are called as the register set of the 8086.

Category	**Bits**	**Register Names**
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (Stack Pointer), BP (Base Pointer)
Index	16	SI (Source Index), DI(Destination Index)
Segment	16	CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment)
Instruction	16	IP (Instruction Pointer)
Flag	16	FR (Flag Register)

General Purpose Registers	**Special Function**
AH+AL = AX	Accumulator
BH+BL = BX	Base Register
CH+CL = CX	Counter
DH+DL = DX	I/O device addressing

Index Registers

BP = Base Pointer

SP = Stack Pointer

SI = Src Index Register
DI = Dest Index Register

Segment Registers
CS = Code Segment Register
DS = Data Segment Register
SS = Stack Segment Register
ES = Extra Segment Register

IP = Instruction Pointer Register
(IP is not under the direct control of the programmer)

Flag Register

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
				O	D	I	T	S	Z		AC		P		CY

O :	Overflow
D:	Direction
I :	Interrupt
T:	Trap
S:	Sign
Z:	Zero
AC:	Auxiliary Carry
P:	Parity
CY:	Carry

Instructions associated with Flags:-

STC	PUSHF
CLC	POPF

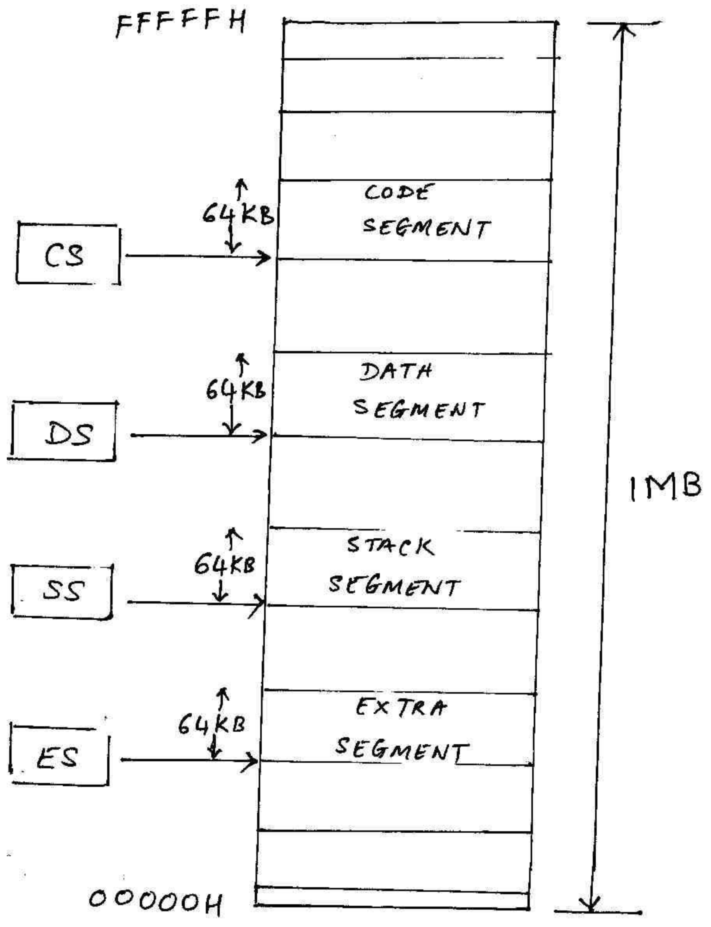
STI
CLI

STD

CLD

Not all flags are user programmable.

MEMORY ORGANIZATION



- The 8086 can access **1MB** of memory.
  1MB = 2^20 i.e. the 8086 has a 20bit address bus, or 20 address pins. The 20 bit address is called the physical address.
- Each address represents one byte of memory.
- At a time the 8086 can work with only 4 blocks of memory, each of size 64 KB. These blocks are called segments.
- The 4 segments can be located anywhere in the 1MB memory space. The segments can even overlap each other partially or fully.

- The 4 segments are Code Segment, Data Segment, Stack Segment and Extra Segment.
- There are 4 Segment Registers which point to the start of each segment. These registers are 16 bit registers.
- The exact location of a byte inside a segment is called its segment offset. But the size of each segment is 64 KB. 64KB = 2^16, hence a segment offset is 16bits.
- The 20 bit address of a byte in memory is calculated as follows:-
  (Segment address << 4) + segment offset = 20 bit phy addr.
- The segment registers always hold the segment address. The segment offset (or simply offset) is held in different registers for different segments:

Segment:	**Offset register**
CS	IP
DS	SI, DI ,BP
SS	SP, BP
ES	SI, DI, BP

Ex1: Find the 20 bit physical address if CS=EF00h and IP=9999h.
(answer on next page)

```
Phy addr = (EF00 << 4) + 9999h
         =    1110_1111_0000_0000_0000
            +          1001_1001_1001_1001
         ---------------------------------------------
                1111_1000_1001_1001_1001

         =  F8999H
```

Practice:

a. DS = a5a5h, BP = 1212h (ans: a6c62h)
b. C000:FFFF ans=?
c. ES=1234h, di:5678 ans=?
d. Find the starting and ending address of data segment if the value of DS is 3700H.

Initializing Segment registers:

- Segment registers cannot be initialized directly with a constant value
- We must write the constant value to another register, then copy that register to the segment register.
- E.g.

```
mov ax, @data
Mov ds, ax
```

Memory models

Model	No of code segments	No of data segments
Small	1	1
Medium	More than 1	1
Compact	1	More than 1
Large	More than 1	More than 1

There are two more models TINY and HUGE which we shall not discuss.

•

SOME I/O FUNCTIONS

```
#include <stdio.h>

void main( ) {


char *msg1 = “Hello, world!\n”;
char msg2[50];
char x,y;

printf(“%c”,  x);             ; Print a character

scanf(“%c”, &y);             ; Get a character
-OR-
getch();

printf(“%s”, &msg1);      ; Print a string

scanf(%s”, &msg2[0]);  ; Input a string from the keyboard

}
```

ASSEMBLY I/O FUNCTIONS:

```
;  I/O Functions in Assembly

; No need to include stdio.h, because we are not doing any i/p or o/p

.model small
.stack 200
.data
        Msg1 db “Hello, world!”, 13,10, ‘$’
        Msg2 db 50 dup (?)
        X db ?
        Y db ?
```

```
.code

start:

        ; PRINT A CHAR

.code
Start:
        Mov ax, @data
        Mov ds, ax

        Mov dl, ‘e’
        Mov ah, 02
        Int 21h

        Mov ah, 4ch
        Int 21h

        End start

        ; GET (INPUT A CHAR)

        mov ah, 01h              ; keyboard input with echo
        int 21h
        ; Here AL = contains the ASCII  code of the key pressed

        mov ah, 08h              ; keyboard input without echo (e.g.
password)
        int 21h
        AL = contains the ASCII  code of the key pressed

        mov ah, 06h              ; alternate function
        mov dl, 0ffh
        int 21h
        AL = contains the ASCII  code of the key pressed
```

```
; PRINT A STRING
        lea dx, msg1            dx: pointer to the string “msg”
        mov ah, 09h             ah = function number
        int 21h             actual function call


        ; INPUT A STRING

        my_str db 80 dup(10)

        lea dx, my_str
        mov ah, 8
        mov [my_str+1], ah
        mov ah, 0ah
        int 21h
```

The CMP instruction:

CMP r1, r2 ; Compare two 8 bit registers r1 and r2
–OR –
Compare two 16 bit registers r1 and r2
-OR-
Compare a register and a memory location

e.g.

```
CMP CL, DL
CMP SI, DI
```

The CMP instruction does r1-r2, but doesn’t save the result.

<u>Instead it sets the **CY flag** or the **Z flag** depending on the result.</u>

If r1==r2, Z flag (zero flag) is set
If r1 < r2, CY flag is set
If r1>r2, CY=Z=0 (reset)

The JMP instruction:

The program branches to or jumps to or starts executing from a different location.

In C lang we use "goto"

```
 main() {

part1:
                printf("msg1");
                goto part2;
                printf("msg2");
                        :
                        :
part3:
                printf("msg3");
}
```

In assembly:

```
        Start:

        Part1:
                Mov al, 03
                Jmp part2
                Add bl, 04
                Shr cl
                        :
                        :
                        :
        Part2:
                Add cl,05h
                        :
                        :
```

JMP instruction has three attributes:- NEAR, FAR, SHORT
These depend on relative positions of part1 and part2

If part1 and part2 are in different code segments we use FAR.
If part1 and part2 are in the same code segment but spaced by more than 256 bytes, we use NEAR.
If part1 and part2 are in the same code segment but spaced by less than 256 bytes, we use SHORT.

JMP is an unconditional jump.

Conditional Jump Instructions:

Unsigned data:

JE / JZ	: Jump if equal, Jump if zero flag is set	: ZF
JNE/JNZ	: Jump if not equal, Jump if zero flag is not set	: ZF
JA/JNBE	: Jump if above, Jump if not below or equal	:CF,ZF
JAE/JNB	: Jump if above or equal, Jump if not below	:CF
JB/JNAE	: Jump if below, Jump if not above or equal	: CF
JBE/JNA	: Jump if below or equal, Jump if not above	: CF,AF

Signed data:

JE / JZ	: Jump if equal, Jump if zero flag is set	: ZF
JNE/JNZ	: Jump if not equal, Jump if zero flag is not set	: ZF
JG/JNLE	: Jump if above, Jump if not below or equal	:SF,ZF, OF
JGE/JNL	: Jump if above or equal, Jump if not below	:SF, OF
JL/JNGE	: Jump if below, Jump if not above or equal	: SF, OF
JLE/JNG	: Jump if below or equal, Jump if not above	: ZF,SF,OF

JS	/	JNS
JC	/	JNC
JO	/	JNO
JP	/	JPE
JNP	/	JPO

Conditional Jmps are short jmps i.e. will jump to a location +127 bytes or -128 bytes from the current location.

INC and DEC Instructions:

- INC = Increment register (similar to ++)
- DEC is decrement (-- operator)
- E.g. if CL=08H, then after

```
        INC CL  =>        CL = 09h
        DEC CL =>         CL = 07h
```

- NOTE: INC and DEC do not change the CY flag as ADD and SUB do.

SUMMARY
So far …..

Instructions:

- mov
- add, sub, adc, subb
- lea
- int 21h
- cmp
- jmp
- inc, dec

80x86 internal architecture:

- diagram
- biu, eu
- register set
- flag register

memory organization:

- segmented memory
- role of IP,SP
- 20 address bits
- Physical address calculation
-

- **Exercise1: A program which inputs characters from the keyboard and stops when the ESC character i.e. ESC key is pressed.**

```
.model tiny
.stack 200h
.data
        ESC equ 27
.code
        ; actual program begins here
Start:
        Mov ax, @data   ; strange looking instruction
        Mov ds, ax      ; we'll skip this for now.

        ; input a character
loop1:
        mov ah, 01h             ; keyboard input with echo
        int 21h
        ; Now AL will contain the ASCII  code of the key pressed
        ;
        ; check if it is the ESC key
        cmp al, 27
        Je prog_over    ; yes, it is the ESC key

        ; no its not, so repeat
        Jmp  loop1

        ;quit the program
Prog_over:
        Mov ah, 4ch
        Int 21h
        End start
```

- **Exercise2: A program which finds the largest of 3 numbers x1, x2 and x3.**

```
.model tiny
.stack 200h

my_data segment
        x1 dw 07
        x2 dw 09
        x3 dw 01
        largest dw ?
my_data ends

my_code segment
        Assume cs:my_code, ds:my_data

        ; actual program begins here
Start:
        Mov ax, @my_data      ; strange looking instruction
        Mov ds, ax            ; we'll skip this for now.

        Mov ax, x1
        Mov bx, x2
        Mov cx, x3
x1_x2:
        cmp ax, bx
        Jge x1_x3       ; what does jge do?
x2_x3:
        cmp  bx, cx
        jge bx_largest
        jmp cx_largest  ; what does jmp do?
x1_x3:
        cmp ax, cx
        jge ax_largest
        jmp cx_largest

ax_largest:
        mov largest, ax
        jmp prog_over
```

```
bx_largest:
        mov largest, bx
        jmp prog_over

cx_largest:
        mov largest, cx
        jmp prog_over


        ;quit the program
Prog_over:
        Mov ah, 4ch
        Int 21h
my_code ends

        End start
```

Arrays, Indexing, and Pointers

Arrays in C:

int my_arr[10];

int 2d_arr[10][20];

Arrays in Assembly:

We declare a ten-byte array like this:

```
  my_arr db 5, 2, 8, 9, 1, 7, 3, 0, 4, 6
```

No special notation or operator to declare multi-dimensional arrays.

To load the first element of the array into register al is like this:

```
  mov al, byte ptr [my_arr]
```

Accessing the second, the third, and the forth element is like this:

```
  mov al, byte ptr [my_arr+1]
  mov al, byte ptr [my_arr+2]
  mov al, byte ptr [my_arr+3]
  :
```

This is one way to access the array. Here my_arr is the pointer to the first element. This is C-language style addressing.
(In C lang the name of the array acts as a pointer).

The second way is to use two registers, a base register and an index register:

```
  mov  bx, offset my_arr
  mov si, 0                 ; set si to 0, i.e. beginning of the array
```

```
mov  al, byte ptr [bx+si]
inc  si           ; increment index si
  :
mov  al, byte ptr [bx+si]
inc  si           ; increment index si

  :   ; do something here
```

Exercise: Define an array of 10 numbers and find the greatest element

```
.model tiny
.stack 200h
.data
        my_arr db  01, 03, 06, 08, 05, 09, 08, 07, 02, 00
        arr_last db  09
        Max db ?
.code
        ; Actual program begins here
Start:
        Mov ax, @data    ; Initialize data segment
        Mov ds, ax

        ; initialize the variables

        lea bp, offset my_arr
        mov cl, arr_last
        mov al, my_arr[bp]
        mov max, al
loop1:
        inc bp
        mov al, max
        cmp al, my_arr[bp]        ; one of the operands to cmp
                                  ; can be a mem loc
        jge next
        mov al, my_arr[bp]
        mov max, al

next:
        dec cl
        jnz loop1

        ;quit the program
Prog_over:
        Mov ah, 4ch
        Int 21h
        End start
```

Exercise: Modify the above program to use the "loop" instruction

```
.model small
.stack 200h
.data
        my_arr db  01, 03, 06, 08, 05, 09, 08, 07, 02, 00
        arr_last dw  09     ; FIRST CHANGE
        max db ?
.code
        ; actual program begins here
Start:
        Mov ax, @data    ; strange looking instruction
        Mov ds, ax       ; we'll skip this for now.

        ; initialize the variables

        lea bp, offset my_arr
        mov cx, arr_last  ; SECOND CHANGE
        mov al, my_arr[bp]
        mov max, al
loop1:
        inc bp
        mov al, max
        cmp al, my_arr[bp]
        jge next
        mov max,al
next:
        ; The following two instructions are replaced by the loop
instruction
        ;       dec cl;
        ;       jnz loop1

        loop loop1       ; THIRD CHANGE

        ;quit the program
Prog_over:
        Mov ah, 4ch
        Int 21h
        End start
```

- In **loopz**, if CX is not 0 **and** the zero flag is set (i.e. equals to 1), then it takes the jump.
- In **loopnz**, if CX is not 0 **and** zero flag is reset (i.e. equals to 0), then it takes the jump.

Exercise: Write a program to find the length of a string.
(use of loopnz)

```
.model small
.stack

.data
        Str1 db “hello world”,13,10, ‘$’
        length dw ?

.code
        ; actual program begins here
Start:
        Mov ax, @data    ; strange looking instruction
        Mov ds, ax       ; we’ll skip this for now.

        ; initialize the variables

        lea bp, offset str1
        mov si,0
        mov cx, FFFFH    ; set cx to a very high value, because we
                         ; don’t know the length of the string.

Loop1:
        mov al, str1[bp]
        inc si
        cmp al, ‘$’
        loopnz  loop1

        mov length, si
        ;quit the program
Prog_over:
        Mov ah, 4ch
```

```
Int 21h
End start
```

SHORT, NEAR AND FAR Jumps

Different formats for the unconditional Jump instruction:

```
1. JMP  imm8                    ; SHORT

2. JMP    imm16                 ;  NEAR
3. JMP    imm16:imm16           ;  NEAR
4. JMP    r/m16                 ;  NEAR

5. JMP mem32                    ; FAR
```

Short Jumps:

The program branches to a nearby location which is within +127 or -128 bytes of the current location

Near Jumps:

The program branches to a location within the current code segment, but with a displacement more than +/- 127 bytes.

Far Jumps:

The program branches to a location which is part of another code segment.

NOTE: All Conditional Jumps (JGE, JZ, JC, JNP, JA etc.) are always SHORT Jumps.

```
;JMPDEMO.ASM

.model small
.stack

data1 segment
msg1 db "this is a short jump", 13,10,'$'
msg2 db "this is a near jump", 13,10,'$'
data1 ends

code1 segment
assume cs:code1, ds:data1

start:
        org 0000h
        mov ax, @data
        mov ds, ax

part1:
        org 1000h
        mov al, 08
        mov bl, 09
        cmp al,bl
        je part2            ; SHORT JUMP
        inc al
        jmp part1           ; NEAR JUMP

part2:
        jmp code2:quit  ; FAR JUMP

code1 ends

code2 segment
assume cs:code2
quit:
    mov ah, 4ch
    int 21h
code2 ends
```

end start

```
; JMPDEMO.LST


Microsoft (R) Macro Assembler Version 6.11                02/19/09
14:38:36
jmpdemo.asm                                          Page 1 - 1


                              .model small
                              .stack

 0000                         data1 segment
 0000         msg1 db "this is a short jump", 13,10,'$'
       73 20 61 20 73 68
       6F 72 74 20 6A 75
       6D 70 0D 0A 24
 0017         msg2 db "this is a near jump", 13,10,'$'
       73 20 61 20 6E 65
       61 72 20 6A 75 6D
       70 0D 0A 24

 002D                              data1 ends

 0000                         code1 segment
                              assume cs:code1, ds:data1

 0000                         start:
                                   org 0000h
 0000  B8 ---- R                   mov ax, @data
 0003  8E D8                            mov ds, ax


 0005                         part1:
                                   org 1000h
 1000  B0 08                            mov al, 08
 1002  B3 09                            mov bl, 09
 1004  38 D8                            cmp al,bl
 1006  74 05                            je part2     ; SHORT JUMP
 1008  FE C0                            inc al
 100A  E9 EFF8                          jmp part1    ; NEAR JUMP
```

```
100D                          part2:
100D  EA ---- 0000 R                 jmp code2:quit ; FAR JUMP

1012                     code1 ends


0000                     code2 segment
                         assume cs:code2
0000                     quit:
0000  B4 4C                        mov ah, 4ch
0002  CD 21                        int 21h
0004                     code2 ends


                         end start
```

```
Microsoft (R) Macro Assembler Version 6.11                02/19/09
14:38:36
jmpdemo.asm                                          Symbols 2 - 1




Segments and Groups:

             N a m e             Size     Length   Align   Combine Class

DGROUP . . . . . . . . . . . . .   GROUP
_DATA  . . . . . . . . . . . . .   16 Bit        0000  Word         Public
'DATA'
STACK  . . . . . . . . . . . . .   16 Bit        0400  Para         Stack
'STACK'
_TEXT  . . . . . . . . . . . . .   16 Bit        0000  Word         Public
'CODE'
code1  . . . . . . . . . . . . .   16 Bit        1012  Para         Private
code2  . . . . . . . . . . . . .   16 Bit        0004  Para         Private
data1  . . . . . . . . . . . . . 16 Bit        002D         Para         Private


Symbols:

             N a m e             Type     Value    Attr

@CodeSize  . . . . . . . . . . .   Number      0000h
@DataSize  . . . . . . . . . . .   Number      0000h
@Interface . . . . . . . . . . .   Number      0000h
@Model . . . . . . . . . . . . .   Number      0002h
@code  . . . . . . . . . . . . .   Text        _TEXT
@data  . . . . . . . . . . . . .   Text        DGROUP
@fardata?  . . . . . . . . . . .   Text        FAR_BSS
@fardata . . . . . . . . . . . .   Text        FAR_DATA
@stack . . . . . . . . . . . . .   Text        DGROUP
msg1 . . . . . . . . . . . . . .   Byte   0000  data1
msg2 . . . . . . . . . . . . . .   Byte   0017  data1
part1  . . . . . . . . . . . . . L Near        0005  code1
part2  . . . . . . . . . . . . . L Near        100D         code1
```

```
quit . . . . . . . . . . . . . . .  L Near      0000  code2
start  . . . . . . . . . . . . . .  L Near      0000  code1

           0 Warnings
           0 Errors
```

MUL Instr

- The 8086 can multiply two 8 bit numbers or two 16 bit numbers

8 bit multiplication:

- One number must always be in the AL register.
- The other number can be in another 8 bit register or a memory location.
- The **16 bit result** is always in the **AX** register
- AL is an implied operand.
- e.g.
  - mul cl ➔ AX = AL * CL
  - mul byte ptr ds:[0005] ➔ AX = AL * 8-bit data at DS:[0005]

16 bit multiplication:

- One number must always be in the AX register.
- The other number can be in another 16 bit register or a 16 memory location.
- The **32 bit result** is always in the DX:**AX** registers i.e.
  DX = upper 16 bits of the result
  AX = lower 16 bits of the result
- AX is an implied operand.
- e.g.
  - mul cx ➔ DX:AX = AX * CX
  - mul **word** ptr ds:[0005] ➔ DX:AX = AX * 16 bit data at DS:[0005]

Example1:

AL = 20h (32 decimal), DL = 14h (20 decimal)
Then AL * DL = 32 * 20 = 640 decimal or 0280h
After executing “mul dl”,
AX = 0280h or
AH = 02h, AL = 80h

Example2:

AX = 100h (256 decimal), BX = 100h (256 decimal)
Then AX * BX = 256 * 256 = 65,536 decimal or 10000h
After executing “mul bx”,
AX = 0000h and
DX = 0001h

Exercise: Find the factorial of a 16 bit number

```
.model small
.stack
.data
        my_num dw 05h (use a small number, assume the answer is
16 bits).
        Answer dw ?
.code
        ; actual program begins here
Start:
        Mov ax, @data    ; strange looking instruction
        Mov ds, ax       ; we'll skip this for now.

        ; initialize the variables
        Mov cx, my_num
        Mov ax, 0001h
        Mov dx, 0000h

repeat:
        mul cx
        loop repeat

;quit the program
        Mov ah, 4ch
        Int 21h
        End start
```

DIV Instr

- The 8086 can divide a 16 bit number by an 8 bit number –OR divide 32 bit number by a 16 bit number

Dividing a 16 bit number by an 8 bit number

- The dividend i.e. the 16 bit number must always be in AX
- The divisor or 8 bit number can be in an 8 bit register or in a memory location
- E.g.

- div bl ➔ ax / bl
  quotient in al
  remainder in ah

Example 1:
AX = 64h (100 decimal) and BL = 0Fh (15 decimal)
After “div bl”,
al = 06h
ah = 0ah (10 decimal)

Dividing a 32 bit number by a 16 bit number

- The dividend i.e. the 32 bit number must always be in DX:AX i.e. the upper 16 bits in DX and lower 16 bits in AX
- The divisor or 16 bit number can be in a 16 bit register or in a memory location
- E.g.

- div bx ➔ dx:ax / bx
  quotient in ax
  remainder in dx

Example 2:
DX:AX = 11223344h (287,454,020 decimal) and
BX = 100h (256 decimal)

After “div bx”

DX = 44h (remainder)
AX = 112233h (quotient)

Exercise: Define an array of 10 numbers and find the average.

```
.model small
.stack
.data
        my_arr db  01, 03, 06, 08, 05, 09, 08, 07, 02, 00
        average db ?
        remainder db ?
.code
        ; actual program begins here
Start:
        Mov ax, @data    ; strange looking instruction
        Mov ds, ax       ; we'll skip this for now.
```

; initialize the variables

```
        lea bp, offset my_arr     bp = pointer to array
        mov cl, 09
        mov ax, 0000     ;        ax = sum of 10 nos
        mov dl, 00
```

Part1: Find the sum of 10 numbers

```
repeat:
        add al, my_arr[bp]
        adc ah, dl
        inc bp
        loop repeat
```

Part2: Find the average

```
        Mov cl, 10
        Div cl
        Mov average, al
        Mov remainder, ah
```

;quit the program

```
        Mov ah, 4ch
        Int 21h
        End start
```

Practice Problems

1. Write an ALP to check if a given number is odd or even

2. Write an ALP to check if a given number is prime

3. Write an ALP to add two 32-bit numbers using the ADC instruction.

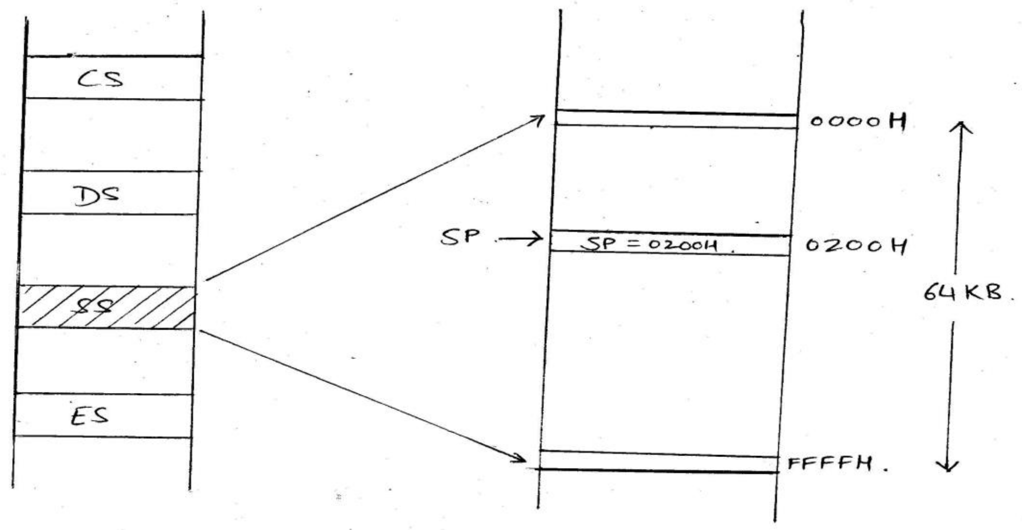
4.

5. Class Notes

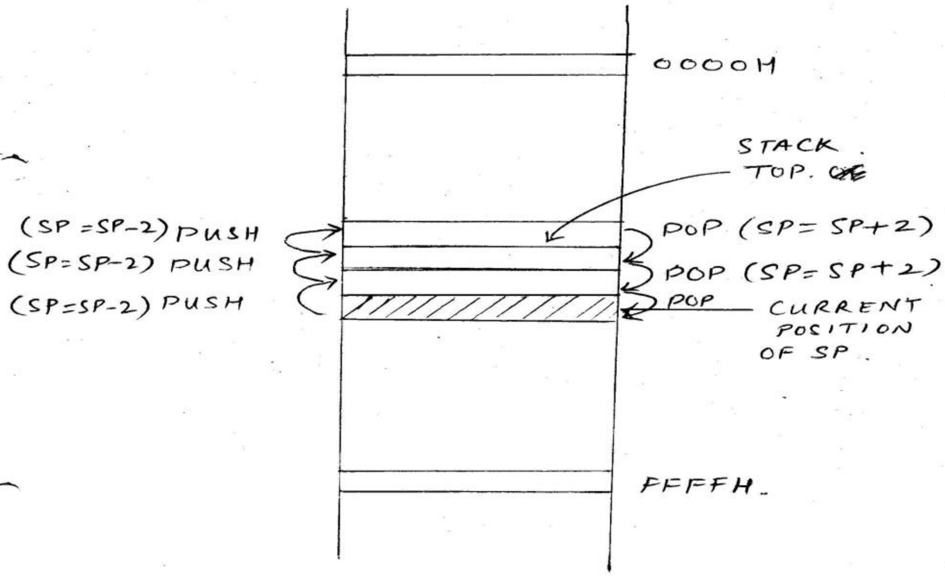
1. Go to http://groups.yahoo.com
2. Sign In or get a Yahoo useriD if you don't have one (its free)
3. Search for the group nhce_mp_4a_4b
4. Join this group
5. Open the folder called CLASS NOTES
6. Download the Word document
7. This document will be updated approximately once a week

STACK SEGMENT, STACK and STACK POINTER

- Stack segment is one of the 4 segments used by a program
- SS is the Stack segment register
- SP is the stack pointer, which points to the "top" of the stack.



- A stack is a FILO (First –in last-out) area of memory used to store temporary data. E.g. like a pile of books.
- We add data to the stack memory, or “stack” by using the push instruction. (Data can be added or removed in multiples of two bytes i.e. we cannot add a single byte to the stack).
- We remove data from the stack by using the pop instruction.



REGISTERS associated with the STACK

SS, SP and BP are the 3 registers most often used with stack.

- **SS** points to the start of the stack segment.
- **SP** points to the current “top” of the stack. i.e. the data that will be first removed from the stack.
- **BP** also used as an index register for stack operations, especially passing parameters to functions or subroutines.

INSTRUCTIONS used to operate the STACK

- **Most common stack operations** -- **push** or **pop**.

  The PUSH instruction push will add data on to the stack, while POP will take it out. The syntax is like this:

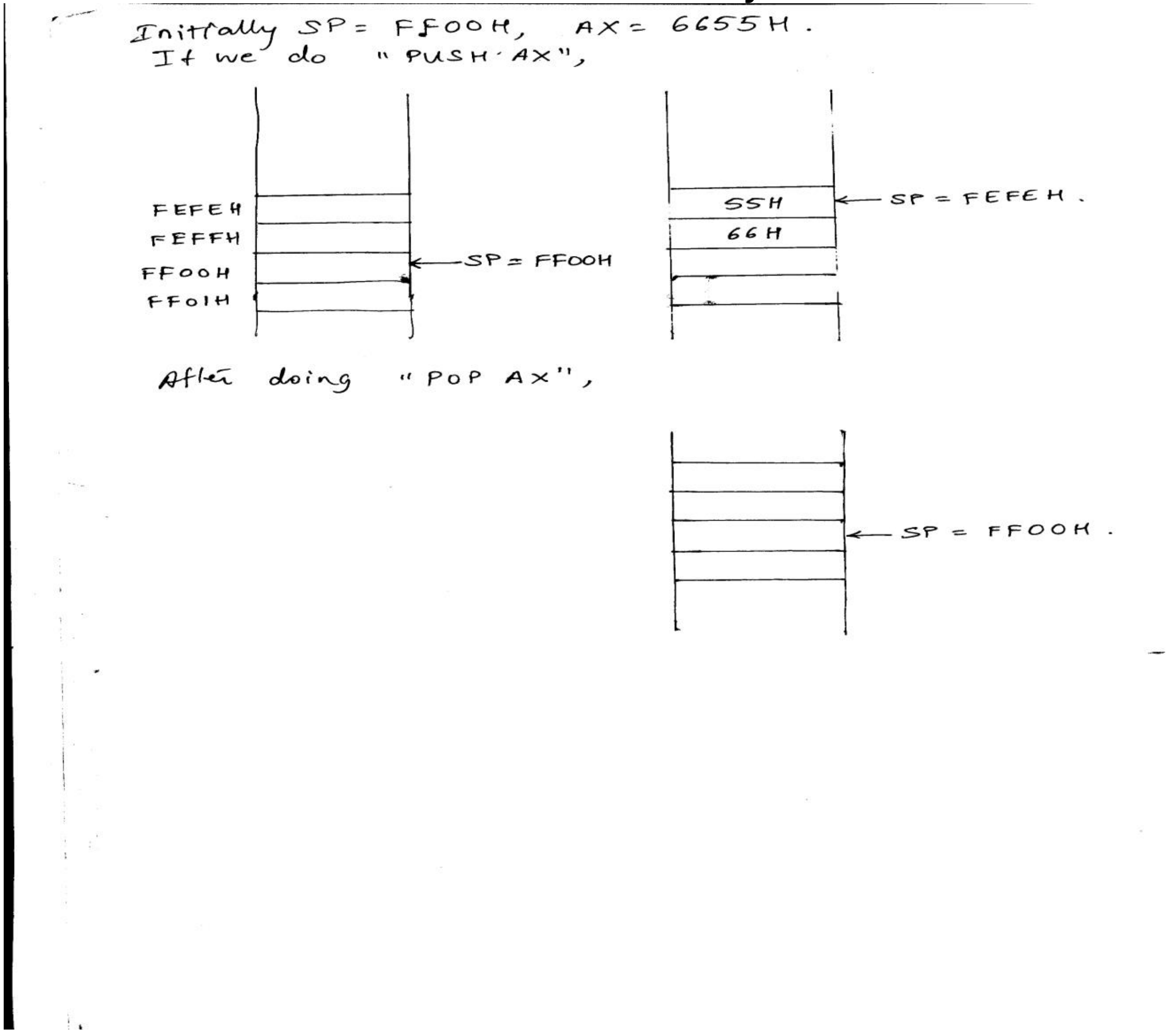
```
push  x    ; push x on the stack
 pop  x    ; pop  x out of the stack
```

where x is a 8 bit register, a 16 bit register or a memory location.

- The PUSH instruction adds data to the stack by subtracting SP by 2, and then storing the new data at the new address in SP. The new data becomes the new “top of the stack”, as shown below.
- The POP instruction reads data from the “top” of the stack and writes that data to the destination register. It then increments the SP by 2, effectively “deleting” data from the top of the stack.

- Can we push a constant?

  In 8086- NO. **In 80286 or above yes.**



NOTE:

1. **The lower byte 55h is pushed first, then the upper byte 66h**
2. **See the previous diagram also**

IMPORTANT USES OF STACK

1. To call functions or subroutines and save their return addresses (using the instructions CALL, RET, INT, IRET)

2. To pass parameters to subroutines (using the instructions PUSH, POP)

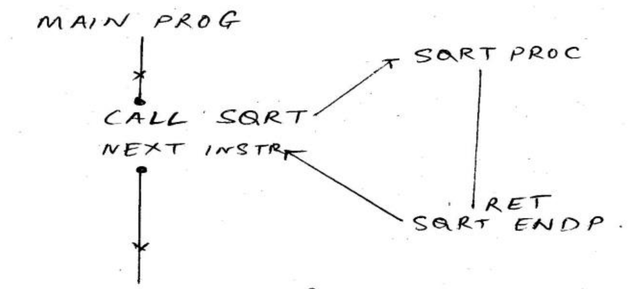
3. To save processor flag status (using the instructions PUSHF, POPF)

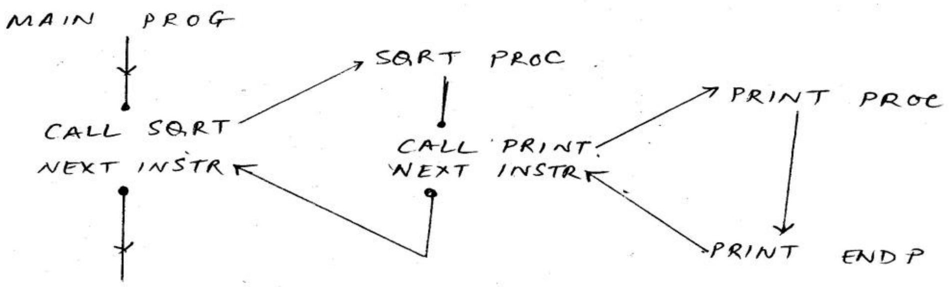
4. To define local or temporary variables.

5. To preserve original register values if we change them in a subroutine (using the instructions PUSH, POP, **PUSHA, POPA**)

We shall now study each of the above in more detail.

CALL and RET Instructions





- What are subroutines, functions and procedures
- Functions usually return a value.
- Subroutines usually are longer and are used to perform specific tasks.
- Both functions and subroutines can call other functions or subroutines. (nested fns, recursive fns)
- Another name for functions and subroutines is procedures.

In C lang:

```
int x,y,z;     /* global variables */
int ans;

void main()
{
        printf(“ program to find the biggest of x,y, and z\n”);
        max();
        printf(“program over”);
}

int max()
{
   if((x>=y) &&(x>=z)) ans = x;
   else if((y>=x) &&(y>=z)) ans = y;
   else if((z>=x) &&(z>=y)) ans = z;
}
```

In assembly the same program will look like this:

```
Data1 segment
        a dw 6;
        b dw 10;
        c dw 20;
        ans dw ?
Data1 ends

Code1 segment
        Assume cs:code1, ds:data1
start:
        call max;                        ----- 1

        mov ah, 4ch
        int 21h

        max proc near                    ---- 2
        mov ax, a
        cmp ax, b ; ax < b ?
        jc b_c      ; yes, compare b and c
        cmp ax, c ; ax < c ?
        jc b_c
        mov ans, ax       ; ax is the biggest
        jmp over
b_c:
        mov ax, b
        cmp ax,c
        jc c_biggest
        mov ans, ax
        jmp over
c_biggest:
        mov ax, c
        mov ans, ax
over:
        ret                              ---- 3
        max endp                         ---- 4

code1 ends
        end start
```

- In C we simply write **“max()”** to go to the max function. In assembly we use the call instruction.

- The procedure starts with a **proc** directive.

- In assembly the function ends with a “**ret**” instruction. The “ret” instruction takes the program back to the calling routine. (in this case the main program).

- Note that **multiple ret** may be used in a subroutine if there is more than one exit point as in the next example.

- In addition we also use the **endp** directive to tell the assembler that the subroutine is over.

If the procedure max is located in another segment, then the same program will look like this:

```
Data1 segment
        a dw 6;
        b dw 10;
        c dw 20;
        ans dw ?
Data1 ends

Code1 segment
        Assume cs:code1. ds:data1
start:
        call max;

        mov ah, 4ch
        int 21h
Code1 ends


code2 segment
        assume cs:code2

        max proc far
        mov ax, a
        cmp ax, b ; ax < b ?
        jc b_c      ; yes, compare b and c
        cmp ax, c ; ax < c ?
        jc b_c
        mov ans, ax        ; ax is the biggest
        jmp over
b_c:
        mov ax, b
        cmp ax,c
        jc c_biggest
        mov ans, ax
        jmp over
c_biggest:
        mov ax, c
        mov ans, ax
```

```
over:
        ret
        max endp

code2 ends


end start
```

Notes:

1. The “end start” statement is the last statement of the program, and NOT the last line of the code1 segment.

2. The procedure max should end with a endp directive

3. No parameters are passed using (…,…) like in C.

4. The assume directive should be used with every code segment.

5. Any procedure should always have a ret statement. If you forget to put this ret, the program will hang!

NEAR and FAR calls:

- A function in the same code segment is called using a “NEAR” call

- A function in a different code segment is called using a “FAR” call

- “call max” is a **direct call**

- “call [bx]” where bx contains the starting address of max() is an **indirect call**.

- So we have 4 types of calls-
  - Direct intra-segment (near) calls
  - Indirect intra-segment (near) calls
  - Direct inter-segment (FAR) calls
  - Indirect inter-segment (FAR) calls

Why is it important to write “near” or far”?

Because it matters to the stack!

The stack works differently for near and far calls.

- For NEAR CALLS, only the IP is saved on stack
  (SP is decremented by 2)

- For FAR calls both IP and CS value is saved on the stack
  (SP is decremented by 4)

Example of NEAR and FAR calls

```
.model small

stack1 segment stack
my_stack db 200 dup(0)
stack_top dw 0
stack1 ends

data1 segment
data1 ends

code1 segment
        Assume cs:code1, ds:Data1, ss:stack1
start:
        mov ax, data1
        mov ds, ax

        mov ax, stack1   ;NOTE: We also initialize the stack segment
        mov ss, ax

        call max;          ; near call
        ;
        call binsrch       ; far call
        ;
        mov ah, 4ch
        int 21h

        max proc near
        ; ::
        ; ::
        ; some instructions here
        ; ::
        ; ::
        ret
        max endp

code1 ends
```

```
code2 segment
      assume cs:code2

     binsrch proc far
     ; ::
     ; ::
     ; some instructions here
     ; ::
     ; ::
     ret
     binsrch endp


code2 ends



end start
```

Here is the lst file for the above program:

```
Microsoft (R) Macro Assembler Version 6.11              02/24/09
02:43:59
call.asm                                                Page 1 - 1


                                .model small
                                .stack

 0000                           Data1 segment
 0000                           Data1 ends

 0000                           Code1 segment
                                        Assume cs: code1, ds: data1
 0000                           start:
 0000  E8 0009                                  call max;
                                        ;
 0003  9A ---- 0000 R                           call binsrch
                                        ;
 0008  B4 4C                                    mov ah, 4ch
 000A  CD 21                                    int 21h


 000C                                           max proc near
                                        ; ::
                                        ; ::
                                        ; some instructions here
                                        ; ::
                                        ; ::
 000C  C3                               ret
 000D                                           max endp

 000D                                   code1 ends


 0000                           code2 segment
                                          assume cs:code2

 0000                                   binsrch proc far
```

```
                              ; ::
                              ; ::
                              ; some instructions here
                              ; ::
                              ; ::
0000  CB                      ret
0001                          binsrch endp


0001                      code2 ends



                          end start
```

```
Segments and Groups:

           N a m e             Size    Length   Align   Combine Class

Code1 . . . . . . . . . . . . .  16 Bit  000D   Para   Private
DGROUP . . . . . . . . . . . . . GROUP
_DATA . . . . . . . . . . . . .  16 Bit  0000   Word          Public  'DATA'
STACK . . . . . . . . . . . . .  16 Bit  0400   Para   Stack          'STACK'
Data1 . . . . . . . . . . . . .  16 Bit  0000   Para   Private
_TEXT . . . . . . . . . . . . .  16 Bit  0000   Word          Public  'CODE'
code2 . . . . . . . . . . . . .  16 Bit  0001   Para   Private


Procedures,  parameters and locals:

           N a m e             Type    Value    Attr

binsrch . . . . . . . . . . . .  P Far    0000   code2       Length= 0001 Private
max . . . . . . . . . . . . . .  P Near  000C  Code1       Length= 0001 Private


Symbols:

           N a m e             Type    Value    Attr

@CodeSize . . . . . . . . . . .Number       0000h
@DataSize . . . . . . . . . . . Number       0000h
@Interface . . . . . . . . . . . Number       0000h
@Model . . . . . . . . . . . . . Number       0002h
@code . . . . . . . . . . . . .  Text    _TEXT
@data . . . . . . . . . . . . .  Text    DGROUP
@fardata? . . . . . . . . . . .  Text    FAR_BSS
@fardata . . . . . . . . . . . . Text    FAR_DATA
@stack . . . . . . . . . . . . . Text    DGROUP
start . . . . . . . . . . . . .  L Near         0000    Code1

          0 Warnings
          0 Errors
```

Passing Parameters to Functions.

In C language:

```
void main()
{
        int x1, x2, x3;      /* global variables */
        int ans;

        printf(" program to find the biggest of x1,x2 and x3\n");
        ans = max(x1,x2,x3);
        printf("program over");
}

int max( int a, int b, int c)
{
   int retval;
   if((a>=b) &&(a>=c))  retval = a;
   else if((b>=a) &&(b>=c)) retval = b;
   else if((c>=a) &&(c>=b)) retval = c;
   return(retval);
}
```

In assembly:

Method1: Passing parameters using memory

```
.data
        x1 dw 6;
        x2 w 10;
        x3 dw 20;
        ans dw ?
.code
start:
        mov ax, @data
        mov ds, ax

        call max;

prog_over:
        mov ah, 4ch
        int 21h

        max proc near

        mov ax, x1
x1_x2:
        cmp ax, x2
        Jge x1_x3

x2_x3:
        mov ax, x2
        cmp ax, x3
        jge bx_largest
        jmp cx_largest
x1_x3:
        cmp ax, x3
        jge ax_largest
        jmp cx_largest

ax_largest:
        mov largest, ax
        jmp proc_over

bx_largest:
        mov largest, bx
        jmp proc_over

cx_largest:
        mov largest, cx
        jmp proc_over


        ;quit the program
Proc_over:
        Ret
        Max endp


        end start
```

Advantages:

1. Simple way to pass parameters
2. Uses minimum number of registers, lets other routines use the registers

Disadvantages:

1. More memory accesses needed, therefore slower to execute
2. Dedicated memory locations used (any other program using this function will have to first copy the values to x1, x2. x3 and then call this routine). Thus the program is not “re-entrant” or “portable”
3. Multiple instances of this routine cannot run simultaneously because they may overwrite each others’s values in x1,x2 and x3

Method2: Passing parameters using registers

```
.data
        x1 dw 6;
        x2 w 10;
        x3 dw 20;
        ans dw ?
.code
start:
        mov ax, x1;
        mov bx, x2;
        mov cx, x3;

        call max;

prog_over:
        mov ah, 4ch
        int 21h

        max proc near
x1_x2:
        cmp ax, bx
        Jge x1_x3
x2_x3:
        cmp  bx, cx
        jge bx_largest
        jmp cx_largest
x1_x3:
        cmp ax, cx
        jge ax_largest
        jmp cx_largest
ax_largest:
        mov largest, ax
        jmp proc_over

bx_largest:
        mov largest, bx
        jmp proc_over
```

```
cx_largest:
        mov largest, cx
        jmp proc_over


        ;quit the program
Proc_over:
        Ret
        Max endp


        end start
```

Advantages:

1. Simple way to pass parameters
2. Needs minimum memory accesses, makes program faster

Disadvantages:

1. This method works only for limited number of parameters because the number of registers are limited.
2. Registers which are holding parameters cannot be used as general purpose registers, hence coding becomes difficult and slow.
3. Multiple instances of this routine cannot run simultaneously because they may overwrite each others's registers unless registers are saved by each instance.

Method3: Passing parameters using pointers

```
.data
        x1 dw 6;
        x2 w 10;
        x3 dw 20;
        ans dw ?
.code
start:

        push si
        mov si, offset x1  (Similar to lea si, x1)

        call max;

        pop si

prog_over:
        mov ah, 4ch
        int 21h

        max proc near
        mov ax, [si]
x1_x2:
        cmp ax, [si+2]
        Jge x1_x3
x2_x3:
        mov ax, [si+2]
        cmp  ax, [si+4]
        jge bx_largest
        jmp cx_largest
x1_x3:
        cmp ax, [si+4]
        jge ax_largest
        jmp cx_largest

ax_largest:
        mov [si+6], ax
        jmp proc_over
```

```
bx_largest:
        mov [si+6], bx
        jmp proc_over

cx_largest:
        mov [si+6], cx
        jmp proc_over


        ;quit the program
Proc_over:
        Ret
        Max endp


        end start
```

Advantages:

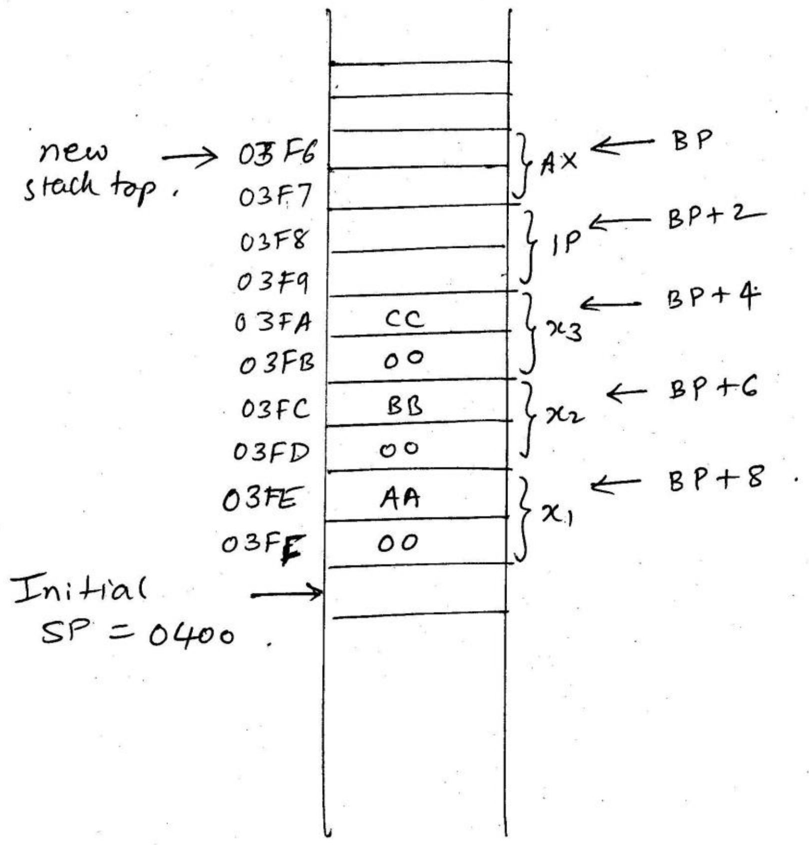
1. Simple way to pass parameters
2. Needs minimum memory accesses, makes program faster

Disadvantages:

3. This method works only for limited number of parameters because the number of registers are limited.
4. Registers which are holding parameters cannot be used as general purpose registers, hence coding becomes difficult and slow.
5. Multiple instances of this routine cannot run simultaneously because they may overwrite each others's registers unless registers are saved by each instance.

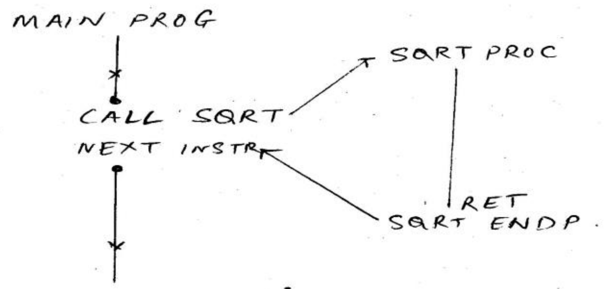
Method4: Passing parameters using stack

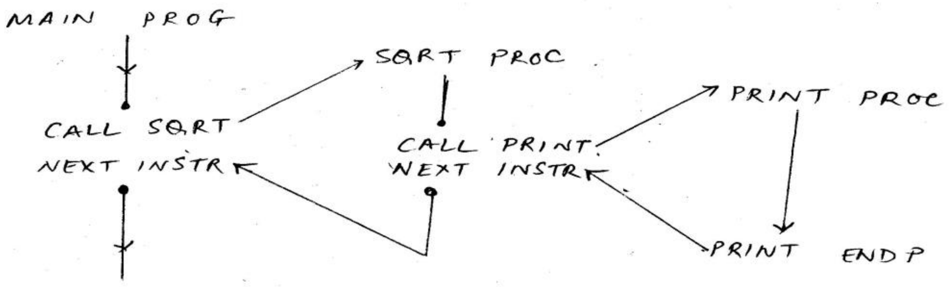
```
.model small
.stack

.data
        x1 dw 000aah;
        x2 dw 000bbh;
        x3 dw 000cch;
        ans dw ?

.code
start:
        mov ax, @data
        mov ds,ax

        push x1
        push x2
        push x3


        call max;

        pop ax
        pop ax
        pop ax

        mov ah, 4ch
        int 21h


        max proc near

        push ax
        mov bp, sp

        ; At this point x1 is [bp+8]
        ;                 x2 is [bp+6]
        ;                 x3 is [bp+4]
        mov ax, word ptr ss:[bp+8]

        cmp ax, ss:[bp+6]   ; is x1 < x2 ?
        jc x2_x3              ; yes, drop x1

        cmp ax, ss:[bp+4] ; is x1 < x3 ?
        jc x2_x3       ; yes, x3 is the biggest

        mov ans, ax          ; no, x1 is the biggest
        jmp over

x2_x3:
        mov ax, ss:[bp+6] ; ax = x2
        cmp ax, ss:[bp+4] ; is x2 < x3 ?
        jc x3_biggest        ; yes, x3 is the biggest
        mov ans, ax          ; no, x2 is the biggest
        jmp over

x3_biggest:
        mov ax, ss:[bp+4]
        mov ans, ax
over:

        pop ax
        ret
        max endp

        end start
```


new stack top.
03F6
03F7
03F8
03F9
03FA
03FB
03FC
03FD
03FE
03FF
CC
00
BB
00
AA
00
AX
IP
x_3
x_2
x_1
BP
BP+2
BP+4
BP+6
BP+8
Initial SP = 0400

NESTED Subroutines

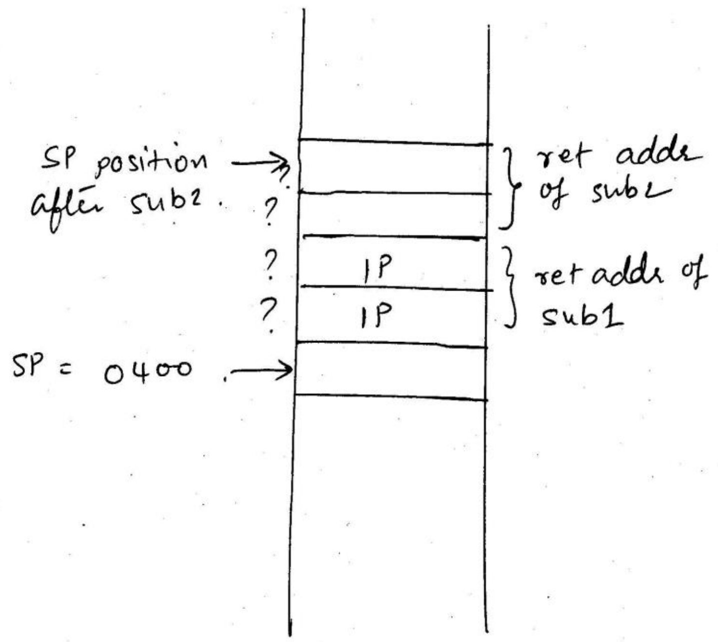
- Main program calls the first subroutine
- This subroutine calls the second subroutine before returning to the main program
- See diag below.





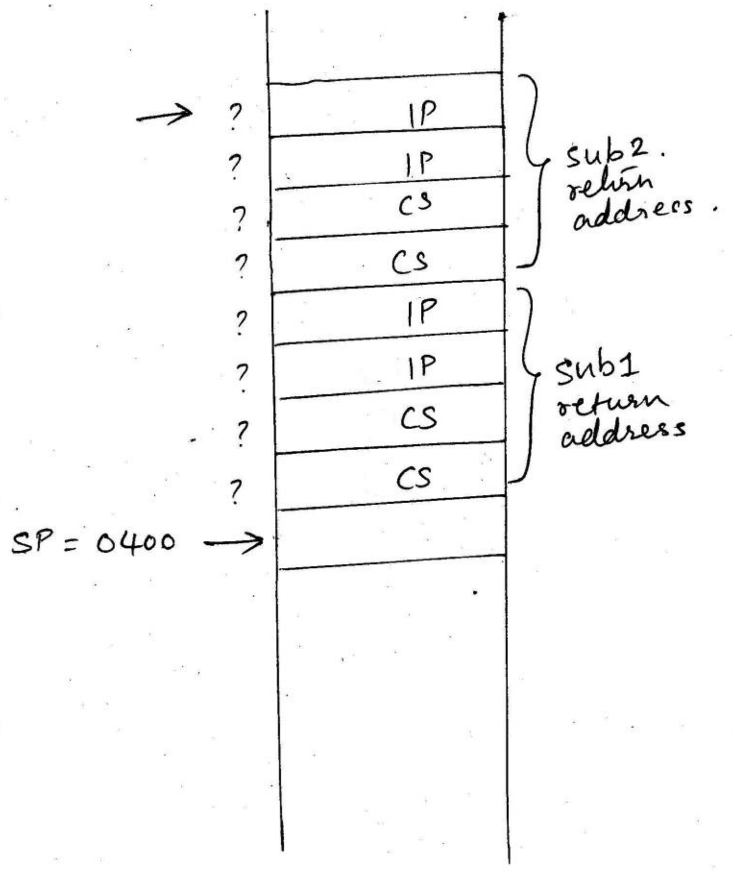
Example1: nested subroutines

```
.model small
.stack

.data

.code

start:

      mov ax, @data
      mov ds, ax
      call sub1
      mov ah, 4ch
      int 21h


sub1 proc near
     call sub2
     ret
sub1 endp

sub2 proc near
     ret
sub2 endp


      end start
```


SP position after sub2.
?
?
?
?
IP
IP
ret addr of sub2
ret addr of sub1
SP = 0400

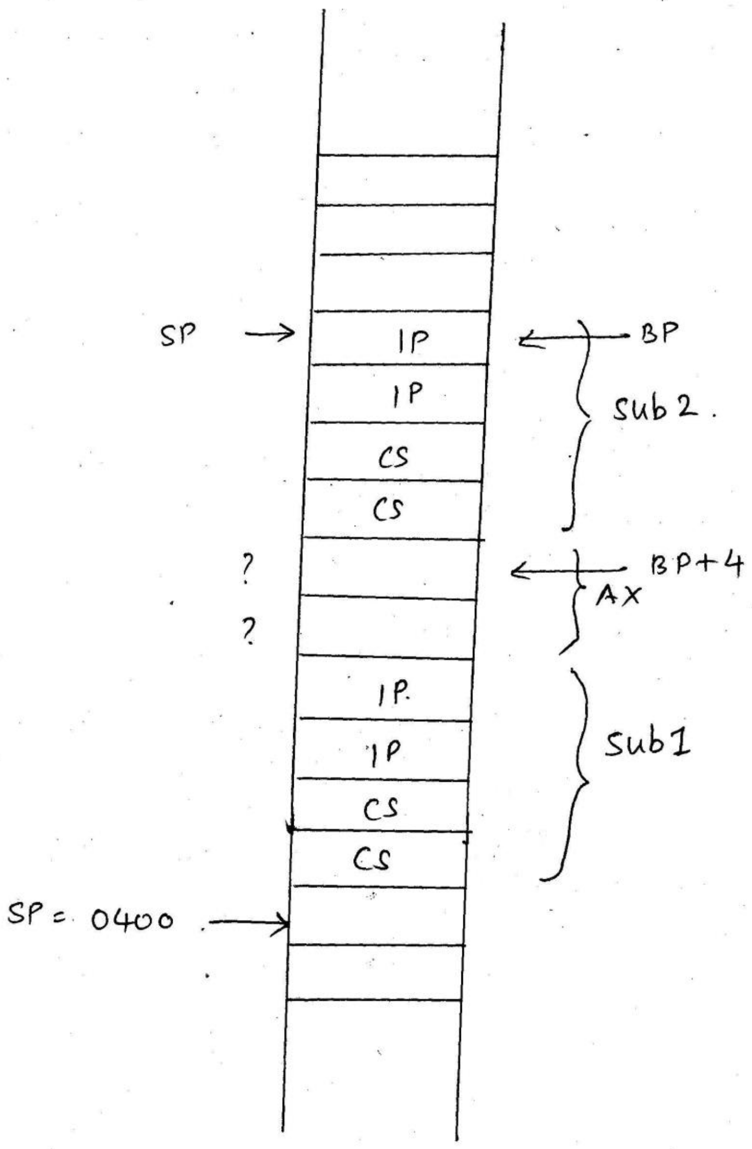
Example2: nested subroutines

```
.model small
.stack

.data

code1 segment
start:

      mov ax, @data
      mov ds, ax
      call sub1
      mov ah, 4ch
      int 21h
code1 ends

code2 segment
     sub1 proc far
     call sub2
     ret
     sub1 endp
code2 ends

code3 segment
     sub2 proc far
     ret
     sub2 endp
code3 ends

      end start
```


?
IP
IP
CS
CS
IP
IP
CS
CS
sub2. return address.
sub1 return address
SP = 0400

Example3: nested subroutines

```
.model small
.stack

.data

code1 segment
start:

      mov ax, @data
      mov ds, ax
      call sub1
      mov ah, 4ch
      int 21h
code1 ends

code2 segment
     sub1 proc far
     push ax ; send a parameter to sub2
             ; similar to sub2(value) in C
     call sub2
     pop ax   ; restore the stack to the orig posn
     ret
     sub1 endp
code2 ends

code3 segment
     sub2 proc far
     ; Get the parameter from the stack
     mov bp, sp
     mov ax, ss:[bp+4]
     ; ::
     ; do something here with the param
     ; ::
     ret
     sub2 endp
code3 ends

      end start
```


SP
IP
IP
CS
CS
BP
Sub 2.
?
?
BP+4
AX
IP.
IP
CS
CS
Sub1
SP = 0400

RECURSIVE function

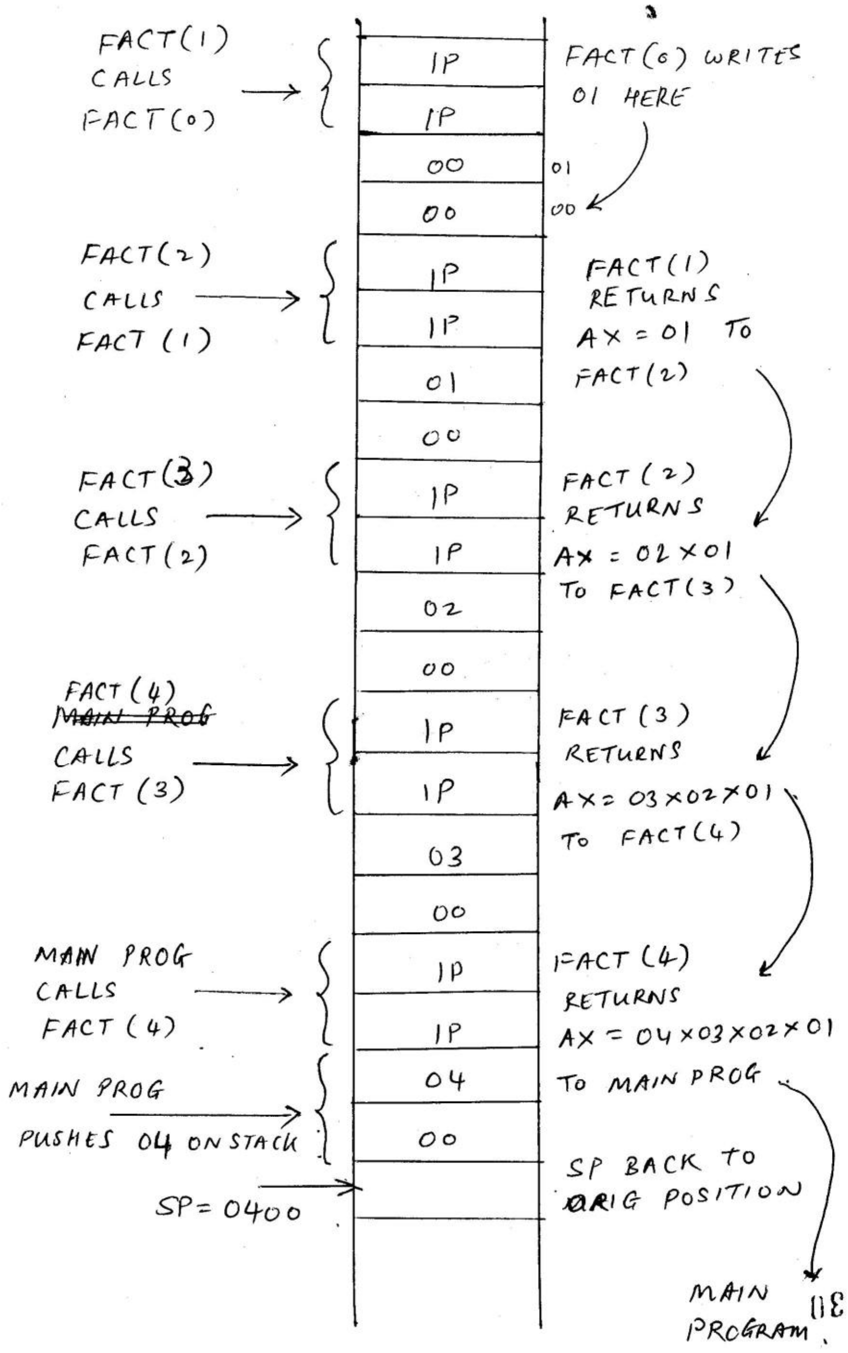
- A function which calls itself
- Common example is factorial.

C version of factorial

```
function fact(N)
          {
                    if(N==1)
                              return 1
                    else
                              return N*fact(N-1)
          }
```

Assembly version

```
; FACT.asm

.model small
.stack
.data
   num dw 4       ; Find factorial of num
   result dw ?
.code
start:
   mov ax, @data   ; Init DS
   mov ds, ax
   mov ax, 01
   mov cx, num     ; If num=0 we have the answer already
   cmp cx, 0
   je exit
   mov bx, cx         ; Pass num as a parameter through BX
   call fact     ;
exit:
   mov result, ax
   mov ah, 4ch
   int 21h

fact proc near
   cmp bx, 1
   jz stop_when_1
   push bx     ; push N, N-1, N-2, till N=1
   dec bx
   call fact
   pop bx      ; pop and multiply
   mul bx
   ret
stop_when_1:
   mov ax, 01
   ret
fact endp

end start
```


FACT(1) CALLS FACT(0)
IP
IP
00
00
01
00
FACT(0) WRITES 01 HERE
FACT(2) CALLS FACT(1)
IP
IP
01
00
FACT(1) RETURNS AX = 01 TO FACT(2)
FACT(3) CALLS FACT(2)
IP
IP
02
00
FACT(2) RETURNS AX = 02×01 TO FACT(3)
FACT(4) ~~MAIN PROG~~ CALLS FACT(3)
IP
IP
03
00
FACT(3) RETURNS AX = 03×02×01 TO FACT(4)
MAIN PROG CALLS FACT(4)
IP
IP
FACT(4) RETURNS AX = 04×03×02×01 TO MAIN PROG
MAIN PROG PUSHES 04 ON STACK
04
00
SP = 0400
SP BACK TO ORIG POSITION
MAIN PROGRAM

Saving Flag register status using PUSHF, POPF

PUSHF : Decrement the SP by 2 and put the contents of the Flag register (16 bits or 2 bytes) at the top of the stack

POPF: Copy the 2 bytes at the top of the stack into the Flag register and increment the Sp by 2.

USE:

When a function is executed the flags will be typically modified.
If the main routine wants to save the current contents of the flags it can use these instructions

```
pushf
Call some_function
popf
```

Saving the processor context using PUSHA, POPA (80186/286+ only)

PUSHA : Push all registers on the stack

POPA: Recover all register values from the stack

USE : Context switch for multitasking

Saving temporary variables on the Stack

Look at the following C program:

```
main()
{
        :
        Function1( a, b,c);
        :
        Function2(d,e,f);
        :
}

Function1(int a, inb, int c)
{
}

Function2 (int d, int e, int f)
{
}
```

- For big applications which call hundreds of subroutines, it is not practical to store all temporary data in data segment.

- Besides we are executing only one subroutine at a time, so no need to save all the local variables.

- Since the scope or life of a temporary variable is limited, ideal to store it on the stack.