Systems Programming

Page No.
Date

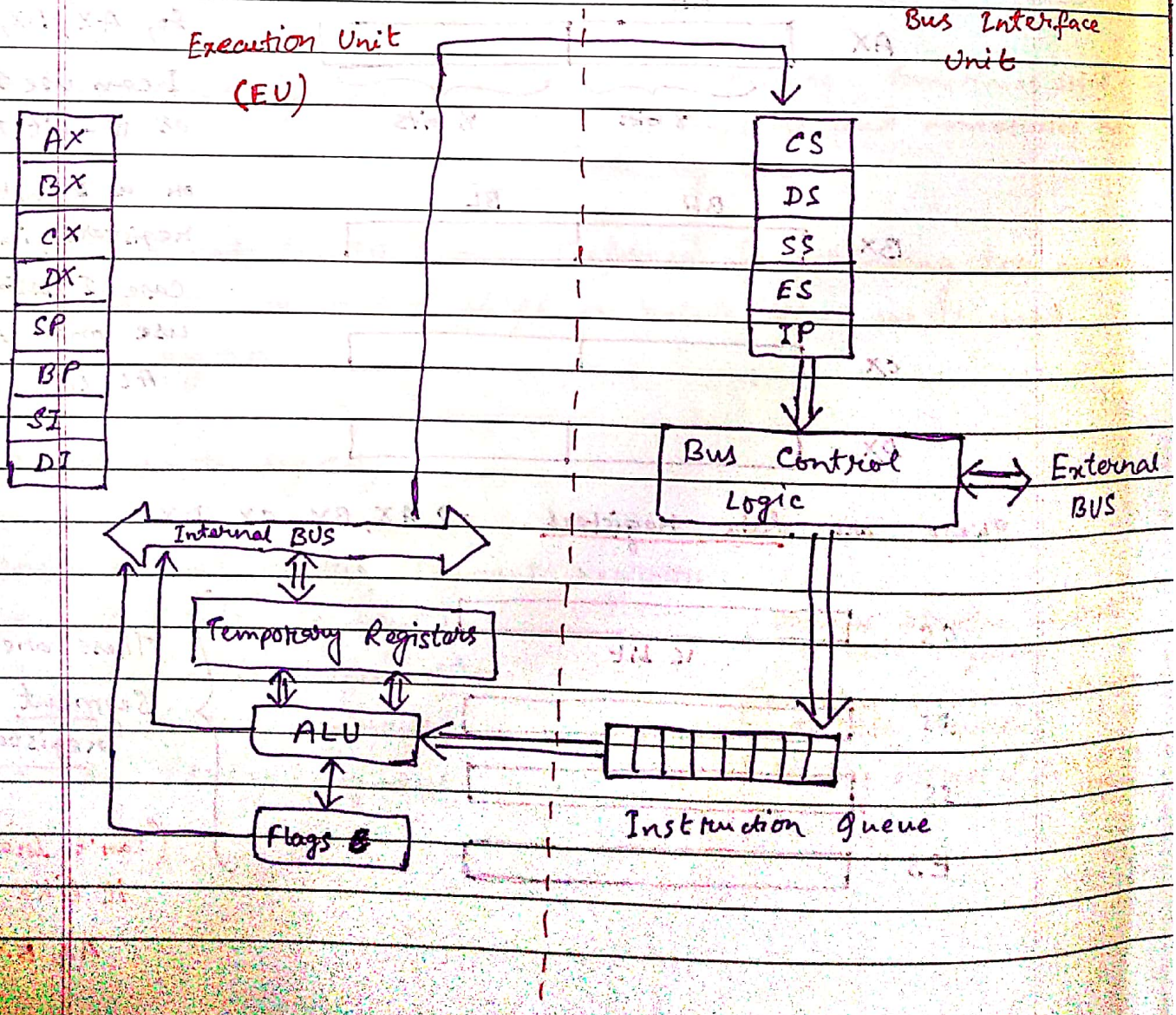
Books:

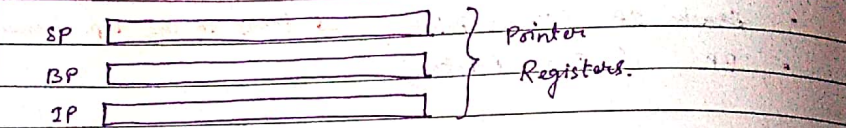
1) Systems Programming – John J. Donoran
2) Microprocessors and Interfacing – Douglas V. Hall
   Programming and Hardware
3) Assembly Language programming and Organization of IBM PC – Ytha Yu and Charles Marut. (Initially this)
4) Systems Software – L. Beck (→ later on (Design part))

* Assembly Language programs depend on the Processor.

In this course, we'll cover 8086 and 8088 microprocessor.

Organization of an 8086 microprocessor :





One more register :

(Flag register) → No specific name now, however its bits have some name. [We'll see later]

- AX (Accumulator Register) → Preferred register for performing arithmetic, logical operations and data transfer.

If I use AX, it will generate the shortest machine code.

- BX (Base Register) → This is multi-purpose. Sometimes used as Data registers and sometimes as Address registers.
- CX (Count Register) → Preferred for performing loop operations.

* Whenever we want to shift or rotate bits, we'll use CL register.

- DX (Data Register)

Now, we come to the Segment registers :

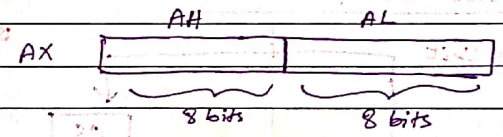
- CS (Code Segment) [Contains starting address of code segment]
- DS (Data Segment) [contains starting address of Data segment]
- SS (Stack Segment) [Starting address of stack seg.]
- ES (Extra Segment)

* Registers are just like memory locations, except that we usually refer to them by name, rather than address.

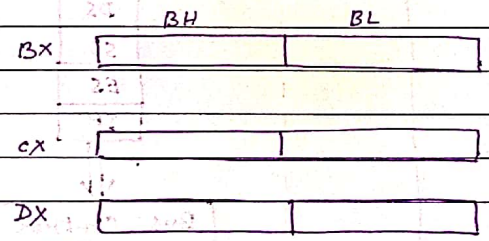
* There are mainly two phases while performing any instruction :
  1) Fetch (Fetch instruction from memory + Decode it to determine operation + fetch the data from memory, if necessary).
  2) Execute

* We can classify the registers as three types :
  1) Data Register
  2) Address Register → Segment Register / Pointer Register / Index Register
  3) Status Register / Flag Register

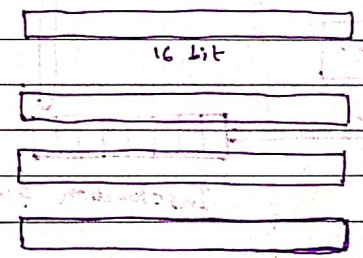
All these are 16 bit registers.



So, AX, BX, CX, DX I can use them as 16-bit registers or as 8-bit registers, in which case I may use only AH, BH, AL, BL, etc.



These are data registers. → AX, BX, CX, DX.



These are called Segment registers.

(Can't break it like those AX, BX etc.)

The syntax of any statement is like this:

Syntax: name operation operand(s) comment

e.g. S1 : MOV CX, 5 ; initialize counter (Anything after semicolon is a comment)

MAIN PROC ; Directive

Variables:

Types: DB → define byte (8 bits)
DW → define word (2 bytes → 16 bits)
DD → define double word (32 bits)
DQ → define quad word (64 bits)
DT → define Ten bytes (80 bits)

e.g. ALP DB 4 ; ALP=4 ALP → variable name
data type = DB value = 4

ALP DB ? ; Uninitialized ALP variable. Use ? for this.

B_ARR DB 10H, 20H, 30H
B_ARR → B_ARR+1 → B_ARR+2

LETTER DB 'ABC' ; LETTER DB 41H, 42H, 43H ;
→ These both are same / equivalent
∵ 41H = 65D → ASCII code of 'A'.

MSG DB 'HELLO', 0AH, 0DH, '$'
N1 DW 1234D

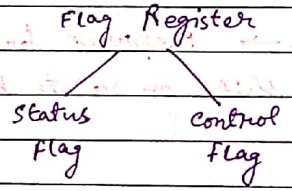
- SP (Stack Pointer) → Points to the top element of a stack.
- BP (Base Pointer)

- SI (Source Index) → Used to point to memory location in the Data Segment.
Format = DS : SI
↑ Segment add. ↖ Offset add.

- DI (Destination Index) → Performs same operation as SI.

- IP (Instruction Pointer) → Instructions are stored in Code segment
Format = CS : IP

So IP contains the offset address of the instruction to be executed.
IP contains the address of the next instruction to be executed.



Assembly Language Programming (ALP)

MASM is an assembler → Microsoft Macro Assembler
(TASM is also similar. It can also be used)

A statement in an Assembly language program can be:
i) Instruction, or
ii) Assembler Directive → To perform some specific task which has been defined beforehand.

- **NEG :**

NEG destination

```
W1 = -10
NEG W1      // W1 = 10.
```

→ How to write certain expressions :??

(i) B = A →
```
MOV BX, A
MOV B, BX
```

(ii) A = 5 - A →
```
MOV AX, 5
SUB AX, A
MOV A, AX
```

(iii) A = B - 2A →
```
MOV AX, B
SUB AX, A
SUB AX, A
MOV A, AX
```

```
MOV AX, B
MOV BX, A
ADD BX, A
SUB AX, BX
MOV A, AX
```

* **Memory Model** → Effectively, what is the size of the code segment, the size of the data segment, that is determined by the specific model.

(i) Small → Code is in 1 segment and data is also in 1 segment

(ii) Medium → Code is in more than 1 segment but data is in 1 segment

(iii) Compact → Code is in 1 segment and data is in more than 1 segment.

(iv) Large → Code and Data both are in more than 1 segment and array size < 64k bytes

(v) Huge → Code and Data both are in more than 1 segment and array size > 64k bytes.

- **Program Structure**

Memory Model

.MODEL m_model ⇒ SMALL
MEDIUM
COMPACT
LARGE
HUGE

We'll generally use .MODEL SMALL

Lec 2

- **Named Constant**

```
name EQU constant_value      // EQU : Equals
LF EQU 0AF ; LF = 0AF        // If we assign using EQU, that variable's value can't be changed during the program. It remains constant.
```

- **MOV**

MOV destination, source

```
MOV AX, W1      MOV AH, W1   // W1 → word type variable, only lower byte of W1 is copied to AH in this case.
MOV AX, BX      MOV W1, W2   // Error → both can't be memory variables
MOV AH, 'A'
MOV W1, AH
```

(* First low byte comes, then high byte)

destination, source can be both registers or one register and one memory variable.
→ both can't be memory variables.

- **XCHG**

XCHG destination, source // XCHG = Exchange.
// Here also, both can't be memory variables.

e.g. XCHG AX, W1

- **ADD / SUB**

ADD/SUB destination, source

```
ADD W1, AX ;   // W1 = W1 + AX
SUB AX, DX ;   // AX = AX - DX
ADD B1, B2 X   // Error. Both can't be memory variables
MOV AL, B2
ADD B1, AL
```

- **INC** (Increment)

INC destination

```
INC AX
INC W1
```
// content of destination will be incremented by 1.

- **DEC** (Decrement)

DEC destination

```
DEC W1
```

② AH=2 :

Input: AH=2
DL = ASCII code of character to be displayed.

Output : AL=DL and Display.

ASCII codes

7 → beep sound
8 → back space
9 → tab
10 → line feed // Go to end of line
13 → carriage return // Next line's beginning

e.g. Program

(Program to take an input character and display it in the next line)

```
TITLE PG1 : ECHO ASM          // Non-executable line
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AH,2
    MOV DL, '?'
    INT 21H
    MOV AH,1
    INT 21H
    MOV BL, AL
    MOV AH,2
    MOV DL,0DH
    INT 21H
    MOV DL, 0AH
    INT 21H
                  => Moves cursor to the beginning of the next line.
    MOV DL, BL
    INT 21H
    MOV AH, 4CH
    INT 21H
                  => To return to the DOS.
MAIN ENDP
END MAIN
```

③ AH=9 : String output

Input : AH=9
DX = offset address of the string
i.e. beginning address of the string.

Output : Display

(String should end with a $ sign)

* Instruction for storing offset address :

LEA (Load Effective Address)

LEA destination, source

// Here, destination should be a general register.
// Source should be a memory location

e.g. LEA DX, MSG

DATA Segment

```
.DATA
    W1 DW 2
    W2 DW 10
    MSG DB 'HELLO!'
    MASK EQU 10011111B
```

STACK Segment

.STACK size e.g. .STACK 100H

CODE Segment

```
.CODE c_name
p_name PROC
    ; body
p_name ENDP
.CODE
MAIN PROC
    ; body
MAIN ENDP
```

Overall structure

```
.MODEL SMALL
.STACK 100H
.DATA
    ; body
.CODE
MAIN PROC
    ; body
MAIN ENDP
; other procedures
END MAIN
```

Lec 3

- Input and Output instructions

INT 21H → Its behaviour depends on what value is there inside AH.

AH
1 → Single key input
2 → Single character output
9 → String output

① AH=1 :

Input: AH=1

Output: AL = ASCII of the input character

- **Flag Register :**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

The flags can be mainly characterised into two types :

Status Flags

1) CF (Carry Flag) → CF=1 ⇒ There is a carry out of MSB on addition or there is a borrow on MSB for subtraction

2) PF (Parity Flag) → PF=1 if low byte of a result has even number of 1s
else PF=0 if odd no. of 1s
e.g. result after some addition is FFFEH → Then PF=0 as it has 7 1s in low or byte

3) AF (Auxiliary Flag) → AF=1 if there is a carry out of/ borrow to at bit 3 upon addition/subtraction

4) ZF (Zero Flag) → ZF=1 for zero result, and ZF=0 for non-zero result

5) SF (Sign Flag) → SF=0 if MSB of a result is 0 i.e. result is +ve
else SF=1 if result is -ve,
i.e. SF=1 if MSB = 1

6) OF (Overflow Flag) → OF=1 if signed overflow occurred.

Control Flags

1) TF (Trap Flag)

2) IF (Interrupt Flag)

3) DF (Direction Flag)

- **Flow Control Instructions :-** (3 types)

① Signed Conditional Jump (Both are similar) → JG = Jump if greater than
→ JNLE = Jump if Not less than or equal to
(when : ZF=0 and SF=OF)

- **Program Segment Prefix (PSP) :**

At the beginning of the program, DS and ES both also point to the CS, i.e. beginning of the code segment.
In this case, if we access some data variable, it will point somewhere else, and give garbage value. To solve this problem, after Main procedure, we should write these two lines:

```
MOV AX, @DATA   // AX is needed. MOV DS, @DATA
MOV DS, AX      // can't be done directly.
```

E.g.: 1) In one variable, we'll store "HELLO!" and display it.

```
. MODEL SMALL
. STACK 100H
. DATA
  MSG DB 'HELLO!$'
. CODE
  MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    LEA DX, MSG
    MOV AH, 9
    INT 21H
    MOV AH, 4CH
    INT 21H
  MAIN ENDP
  END MAIN
```

In Data segment, all the variables/data are stored in consecutive bytes. →

Here, $ was not there in MSG2, so without even explicitly calling CHAR, until $ is encountered, it prints. So, here A is printed.

If MSG2 ....
V1 DB 5 was written,
CHAR .... $ output would've been:
In upper case,... it is : 5A

2) : Program to take input a lowercase letter and output it in uppercase.

```
. MODEL SMALL
. STACK 100H
. DATA
    CR EQU 0DH
    LF EQU 0AH
    MSG1 DB 'Enter lowercase letter : $'
    MSG2 DB 0DH,0AH,'In upper case it is :'
    CHAR DB ?,'$'
. CODE
  MAIN PROC
    MOV AX, @DATA
    MOV DS, AX
    LEA DX, MSG1
    MOV AH,9
    INT 21H
    MOV AH,1
    INT 21H
    SUB AL,20H
    MOV CHAR, AL
    LEA DX, MSG2
    MOV AH,9
    INT 21H          // Didn't explicitly print CHAR
    MOV AH, 4CH
    INT 21H
  MAIN ENDP
  END MAIN
```

* Program to print all 256 characters.

```
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AH,2
    MOV CX,256
    MOV DL,0
P_LOOP:
    INT 21H
    INC DL
    DEC CX
    JNZ P_LOOP
```

NOTE: JMP is an unconditional jump statement

Lec 4

- CMP (Compare)
- CMP destination, source // Both can't be memory variables. destination - source is computed. The result is not stored anywhere but the flags are affected.

```
CMP AX,BX
JG BELOW
```
// Suppose AX = 7FFFH, BX = 0001H. Then it will jump to "Below".

// Suppose AX = 7FFFH, BX = 8000H. Then it will again jump to "BELOW" since BX = 0

∵ JG, JL etc. perform signed comparison, addition, subtraction.

However,
```
CMP AX,BX
JA BELOW
```
// AX = 7FFFH, BX = 8000H. Then it won't jump to "BELOW"

* if AX < 0 then AX = -AX endif ⇒

```
    CMP AX,0
    JNL L1
    NEG AX
L1: ----
```

* if AL ≤ BL display AL else display BL endif ⇒

```
    MOV AH,2
    CMP AL,BL
    JG L1
    MOV DL,AL
    JMP D1
L1: MOV DL,BL
D1: INT 21H
```

→ JGE / JNL (SF = OF)
[JGE = Jump if Greater than or Equal to
JNL = Jump if Not less than]

→ JL / JNGE (SF < > OF)
[JL = Jump if less than
JNGE = Jump if Not Greater than or Equal to]

→ JLE / JNG (ZF=1 or SF < > OF)
[JLE = Jump if less than or equal to
JNG = Jump if Not greater than]

② Unsigned Conditional Jump :

→ JA / JNBE (CF=0 and ZF=0)
[JA = Jump if above
JNBE = Jump if Not Below or equal]

→ JAE / JNB (CF=0)
[JAE = Jump if above or equal
JNB = Jump if Not below]

→ JB / JNAE (CF=1)

→ JBE / JNA (CF=1 or ZF=1)

③ Single Flag Jump :

→ JE / JZ (ZF=1)
[JE = Jump if equal
JZ = Jump if zero]

→ JNE / JNZ (ZF=0)
[JNE = Jump if not equal
JNZ = Jump if not zero]

→ JC (CF=1)
[JC = Jump if carry]

→ JNC (CF=0)
[JNC = Jump if not carry]

→ JO (OF=1)
[JO = Jump if overflow]

→ JNO (OF=0)

→ JS (SF=1)
[JS = Jump if signed (-ve sign)]

→ JNS (SF=0)

→ JP (PF=1)
[JP = Jump if parity even]

→ JNP (PF=0)

* if AL = 1 or 3 → display "0"
if AL = 2 or 4 → display "E"

```
CMP AL,1
JE L1
CMP AL,3
JE L1
CMP AL,2
JE E1
CMP AL,4
JE E1

L1 : MOV DL, '0'
     JMP D1
L2 : MOV DL, 'E'
D1 : MOV AH, 2
     INT 21H
```

* FOR loop :

```
LOOP destination_label   // In this line, CX is automatically decremented and if CX ≠ 0, control goes to next line of loop instruction
```

Note that CX must store the no. of times loop shall run.

* Program to print : * * * ... * (10 times.)

```
MOV CX, 10
MOV AH, 2
MOV DL, '*'
L1: INT 21H
LOOP L1
```

→ One problem occurs if initially CX = 0.

```
MOV CX, 0
:
L1 : INT 21H
LOOP L1
```

This will run 2^{16} times until CX again becomes 0.

To solve this, we do:

```
MOV CX, 0
MOV AH, 2
MOV DL, '*'
JCXZ S1      // Jump if CX is 0, to S1
L1: INT 21H
LOOP L1

S1: ....
```

Lec - 5

- **WHILE Loop :**

Structure:
```
count=0
read a char
while char <> CR        // CR = Carriage Return i.e. Enter key.
DO
    count = count + 1
    read a char
end while
```

⇓

```
MOV CX,0
MOV AH,1
INT 21H
L1: CMP AL,0DH
    JE E1
    INC CX
    INT 21H
    JMP L1
E1:
```

- **DO ... WHILE loop**

```
DO
    read a char
until-char is blank
```

⇓

```
    MOV AH,1
R1: INT 21H
    CMP AL,' '      // OR, CMP AL,0
    JNE R1
```

- **Logical Instructions** → AND, OR, XOR, NOT

```
AND/ destination, source
OR/
XOR
```

// Flags are also affected.
// destination must be a register or memory variable.

```
XOR AX,AX  ≡  MOV AX,0
OR CX,CX   ≡  CMP CX,0
```

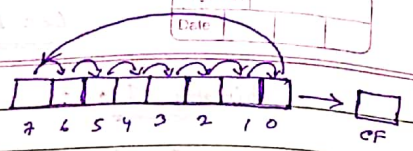
[∵ Flags are affected by OR operation also (ZF flag)]

```
SUB AL,30H
AND AL,0FH
```

} converts digit characters '0' ... '9' to the corresponding digit value, e.g. '8' → 8

NOT AX → Invert all the bits of AX

② ROR : ROR destination, 1/CL

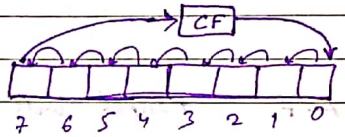


counting no. of set bits in BX and storing the result in AX.

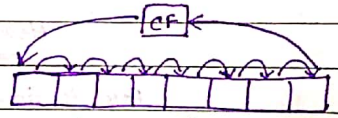
```
XOR AX,AX   // AX=0
MOV CX, 16
L1: ROL BX, 1
    JNC N1
    INC AX
N1: LOOP L1
```

③ RCL : RCL destination, 1/CL

→ This also shifts the bits to left. Shifts MSB to CF and the previous value of CF is shifted to LSB.



④ ~~ROR~~ RCR :



RCR destination, 1/CL

Program to reverse the bits of AL : e.g. AL = 10010110 → becomes : AL = 01101001

```
MOV CX,8
R1: SHL AL,1
    RCR BL,1
    LOOP R1
    MOV AL,BL
```

- TEST instruction : (Flags are affected)

TEST destination, source // Performs AND operation between destination and source but does not store it in destination.

```
TEST AL,1
JZ B2
```

↳ Checks if no. in AL is even or odd. If even, then label goes to B2.

- Shift Instructions :

① SAL :

SHL destination, 1 → LSB becomes 0, left shift occurs and MSB is shifted to CF. i.e. Carry Flag is affected.

② SAL : (Shift Arithmetic Left)

SAL AX, 3 ≡ SHL AX, 8 // i.e. $2^3=8$

③ SHR :

SHR destination, no. → MSB becomes 0, right shift occurs and LSB is shifted to CF

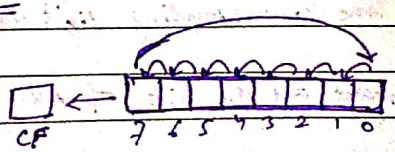
e.g. DH = 8AH , CL = 2
SHR DH, CL ; → Then DH = 22H

④ SAR :

SAR destination, CL

- Rotate Instructions :

① ROL : ROL destination, 1/CL



MSB value goes to LSB and is also stored in CF.

- **Stack :** (SP contains the offset address of the top of the stack)

① **PUSH :** PUSH source // source = 16-bit register or, memory word variable

PUSH AX . This does two things :

1) SP → SP - 2
2) A copy of value at AX will go to the address SS : SP

② **PUSHF :** PUSHF // No arguments required.

→ It just pushes the content of the flag register into the stack.

③ **POP :** POP destination

POP BX . This does two things :

1) Content of SP will be moved to destination (here BX)
2) SP → SP + 2

④ **POPF :** POPF ; → Moves the top of the stack to the flag register. and increments SP by 2.

Lec 7

- **Applications of Stack :**

Q: Program that reads characters until CR is pressed and prints the characters in reverse order.

Lec 6

Binary I/O :

Input

```
XOR BX, BX
MOV AH, 1
INT 21H
W1: CMP AL, 0DH
JE E1
AND AL, 0FH   // Same as SUB AL, 48 for digit inputs
SHL BX, 1
OR BL, AL     // NOTE: OR BX, AL X as AL is 8-bit.
INT 21H
JMP W1
E1:
```

Binary output Algo

For 16 times, do :
Rotate left 0x
If CF = 1
: print '1' (Write code HW)
else print '0'

Hex Input

0 to 9 and A - F → Input will be 4 times [∵ 4×4 = 16 bits]

```
XOR BX, BX
MOV CL, 4
MOV AH, 1
INT 21H
W1: CMP AL, 0DH
JE E1
CMP AL, 39H
JG L1
AND AL, 0FH
JMP S1
L1: SUB AL, 37H
S1: SHL BX, CL
OR BL, AL
INT 21H
JMP W1
E1:
```

Hex Output

For 4 times Do :
Mov BH to DL
Shift DL 4 times to the right
If DL < 10
convert to '0' ... to .. '9'
else
convert to 'A' .. to .. 'F'
endif
Display character
Rotate BX 4 times to the left.
end For.

(Write Code HW)

* RET : RET pop-value (pop-value is optional)

If we write only RET, then the top of the stack will be popped out and stored in IP. → That's what is desired as stack top contained the address of ~~where we started~~ the next line after call statement.

RET N // N is an integer

→ Pops the top of the stack and stores it in IP. Then also pops additional N bytes from the stack. (i.e. SP = SP + N)

* Multiplication of two numbers using addition and bit shifting:

e.g. A = 0111 = 7 , B = 1101 = 13

Prod = 0
if LSB of B is 1
Prod = Prod + A
endif
Shift left A
Shift right B

Repeat this. → (Write a code using Procedure)

```
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AX,7
    MOV BX, 13
    CALL MULTIPLY
    MOV AH, 4CH
    INT 21H
MAIN ENDP

MULTIPLY PROC
    ; input → AX and BX   output → DX = product
    PUSH AX
    PUSH BX
    XOR DX,DX
R1: TEST BX,1
    JZ E1
    ADD DX, AX
E1: SHL AX,1
    SHR BX,1
    JNZ R1
    POP BX
    POP AX
    RET
MULTIPLY ENDP
END MAIN
```

```
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AH,2
    MOV DL, '?'
    INT 21H
    XOR CX, CX
    MOV AH, 1
    INT 21H
W1: CMP AL, 0DH
    JE E1
    PUSH AX        // We wanted to push AL, but we have to push 16-bit register
    INC CX
    INT 21H
    JMP W1
E1: MOV AH,2
    MOV DL, 0DH
    INT 21H
    MOV DL, 0AH
    INT 21H
    JCXZ E2
T1: POP DX
    INT 21H
    LOOP T1
E2: MOV AH, 76
    INT 33
MAIN ENDP
END MAIN
```

* Procedure Declaration :

```
p-name PROC type
    ; body
    RET
p-name ENDP
```

// type → is optional. Give it or don't give it

type → NEAR.
type → FAR

NEAR : the procedure and from where we are calling the procedure, both are in the same segment.

FAR : both are in different segments.

[Default value of type is NEAR]

Q. How to call a procedure?

→ CALL p-name . This does two things:

1. Return address of the next executable line after CALL statement is pushed on top of the stack
2. CS:IP stores the first executable line of the procedure being called.

* Byte form: Dividend has to be 16-bit and must be stored in AX. After division:
  Quotient → AL
  Remainder → AH

* Word form: Dividend has to be 32-bit and must be stored in DX:AX. After division:
  Quotient → AX
  Remainder → DX.

If dividend is 8-bit in Byte form, then:
DIV → AH should be cleared i.e. MOV AH, 0
IDIV → CBW ; // Convert Byte to Word e.g. 10000111 becomes
↳ Signed extension. 1000000010000111

Similarly, if dividend is 16-bit in word form, then:
DIV → DX should be cleared
IDIV → CWD ; Convert word to Double word.

e.g. XBY ÷ (-7) where XBY is a byte type variable

```
MOV AL, XBY
CBW          //Signed extension of AL into AX
MOV BL, -7
IDIV BL
```

Q. Write a ~~prog~~ procedure to print the number in AX in signed decimal format.

```
if AX<0
    print '-' and AX=-AX
endif
```

Get the digits in AX in decimal representation. // Use stack
Convert these digits into characters and print.

• Multiplication:
MUL source → unsigned multiplication
IMUL source → signed multiplication

If we multiply two bytes, the result will be of word type.
If we multiply two words, the result will be of double word type.

For Byte multiplication → one no. is in source, other must be in AL.
→ And after product, the 16-bit result will be stored in AX.

For Word multiplication → One no. is in source, other must be in AX.
The 32-bit product will be stored in DX:AX.
The most significant 16 bits in DX and rest in AX.

Q. Suppose A and B are two word variables. Write instructions to perform A = 5A - 12B (Assume that all multiplication results would fit in 16 bits)

```
MOV AX, 5
IMUL A
MOV A, AX
MOV AX, 12
IMUL B
SUB A, AX
```

Q. Write a procedure to compute N! (factorial)

```
FACT PROC
; input : CX=N , ouput : AX=N!
PUSH CX
MOV AX, 1
T1: MUL CX      // we know N>0, so we shall use MUL
LOOP T1            i.e. unsigned.
```

• Division:
DIV divisor → unsigned division
IDIV divisor → signed division

- **One Dimensional Array:**

```
GAMMA  DB  100  DUP (0)   // GAMMA is an array of 100 byte type variables, all initialized with 0.
DELTA  DW  212  DUP (?)
```

Suppose W1 is a word array. How to exchange the 10th and 25th element of W1 ? [Note: For XCHG, both can't be memory variables]

```
MOV  AX, W1+18          // 1st ele = W1
XCHG W1+48, AX          // 2nd ele = W1 + 1x2  (each no. = 2 bytes)
MOV  W+18, AX           // 10th ele = W1+9x2
```

Lec 9

Different ways of accessing elements of an array:

① **Register Indirect Mode:**

We use : [Register]

where Register stores the offset address of the element

Allowed registers are: BX, SI, DI , BP

Segment address for these : DS

SS ← Segment address for BP

e.g. `MOV AX, [SI]`

Suppose W is a word array : Find the sum of its elements.

```
W DW 10,20,30,40,50,60
        MOV CX,6
        XOR AX,AX
        LEA SI, W
L1:     ADD AX, [SI]
        ADD SI, 2          // Adding 2 because each element of W is of 2 bytes size.
        LOOP L1
```

Q. Write a procedure to reverse the elements of an array.

e.g. 1,2,3,...,N becomes N, N-1, ..., 2, 1

e.g. AX = 24168

AX ÷ 10 ⇒	q = 2416	R = 8
AX ÷ 10 ⇒	q = 241	R = 6
AX ÷ 10 ⇒	q = 24	R = 1
AX ÷ 10 ⇒	q = 2	R = 4
AX ÷ 10 ⇒	q = 0	R = 2

} Push these in stack

Name this as OUTDEC PROC and filename as OUTDEC.ASM.

Then, in another file, write this! (Run and Submit on Monday)

```
TITLE PG1 : Decimal Output
.MODEL SMALL
.STACK 100H
.CODE
MAIN PROC
    MOV AX, 1234
    CALL OUTDEC
    MOV AH, 4CH
    INT 21H
MAIN ENDP
INCLUDE C:\OUTDEC.ASM
END MAIN
```

Lec 8

- **Decimal Input:**

```
INDEC PROC
    ; output: AX = No.
    PUSH BX
    PUSH CX
    PUSH DX
B1: MOV AH,2
    MOV DL, '?'
    INT 21H
    XOR BX, BX
    XOR CX,CX
    MOV AH,1
    INT 21H
    CMP AL, '-'
    JE M1
    CMP AL, '+'
    JE P1
    JMP R1
M1: MOV CX,1
P1: INT 21H
R1: CMP AL,'0'
    JL B1
    CMP AL, '9'
    JG B1
    AND AX, 000FH
    PUSH AX
    MOV AX,10
    MUL BX
    POP BX
    ADD BX,AX
    MOV AH, 1
    INT 21H
    CMP AL,0DH
    JNE R1
    MOV AX, BX
    OR CX,CX
    JZ E1
    NEG AX
E1: POP DX
    POP CX
    POP BX
    RET
```

```
MOV CX, 5
XOR BX, BX
XOR AX, AX
L1 : ADD AX, W[BX]
   ADD BX, 2
   LOOP L1
```

Q. Suppose I have a msg with lowercase letters. Convert them into upper case letters.

```
MSG DB 'this is a message'
MOV CX, 17
XOR SI, SI
L1: CMP MSG[SI], ' '
   JE N1
   AND MSG[SI], 0DFH
N1: INC SI
   LOOP L1
```

- **PTR operator :** type PTR address_expression
  - type → BYTE / WORD
  - address_expression → some memory variable

(used to override the declared type of an address expression)

e.g.

```
DOLLAR DB 1AH
CENT DB 52H

MOV AX, DOLLAR    (Wrong)
MOV AX, WORD PTR DOLLAR   // Then, AL = 1AH
                          //       AH = 52H
```

- **LABEL :**

```
MONEY LABEL WORD
DOLLAR DB 1AH
CENT DB 52H
  :  :
MOV AX, MONEY
```

These two do the same thing.

depending on this type, those many bytes will be moved to AX during MOV AX, MONEY. MONEY points to DOLLAR.

- **Segment override :**

We know by default, BX, SI, DI have their segment address coming from DS during array indexing.

We can explicitly override the (default) segment address.

Syntax is → Segment_Reg : [Pointer_Reg]

eg. MOV AX, [SI] ; → moves DS : SI

MOV AX, ES : [SI] ; → moves ES : SI.

```
REVERSE PROC
; input : SI = offset address of array
; BX = no. of elements
; output : Reversed array
PUSH AX
PUSH BX
PUSH CX
PUSH SI
PUSH DI
MOV DI, SI
MOV CX, BX
SHR CX, 1 ; CX = n/2
DEC BX ; BX = (n-1)
SHL BX, 1 ; BX = 2(n-1)
ADD DI, BX ; [DI] = last element

L1: MOV AX, [SI]
XCHG AX, [DI]
MOV [SI], AX
ADD SI, 2
SUB DI, 2
LOOP L1
POP DI
POP SI
  :
RET
```

② **Based and Indexed Addressing mode :**

We can use : [Reg. + displacement] or [displacement + Reg.] (Register or Memory Location (in all three cases))

or [Reg.] + displacement // displacement may be positive, or may be negative.

Again, allowed registers are :

BX, BP, SI, DI // Their segment addresses come from ... as specified for previous case.

If we use BX or BP, we call it Base Addressing Mode.

Indexed Addressing Mode

// Let BX = 4

```
MOV AX, [W + BX]
MOV AX, [BX + W]
MOV AX, W + [BX]
MOV AX, W[BX]
```

all these are valid. Using any of these, 3rd element of W is moved to AX.

W + 2 × (3-1) = W + 4

Q. Given a word array, find the sum of all its elements and store the result in AX.

Variable [base-reg.][index-reg] → for accessing array elements.

e.g. Mov AX, w[BX][SI]
Mov AX, w[BX+SI]
Mov AX, [w+BX+SI]
} All these are equivalent.

w[BX][SI] → accesses element at address w+BX+SI

NOTE:

Q. Suppose A is a 5×7 word array. Write code to clear all the row 3 elements of A.

```
Mov BX, 28      ;  A[3][0] → A + 2×{2×7+0}
Xor SI, SI          = A + 28
Mov CX, 7
L1: Mov A[BX][SI], 0
ADD SI, 2
Loop L1
```

Q. Code to clear all the column 4 elements of A.

```
Xor BX, BX
Mov SI, 6       ;  A[1][4] = A + 2×{0×7+3} = A+6
Mov CX, 5
L1: Mov A[BX][SI], 0
ADD BX, 14
LOOP L1
```

Lec 11

- XLAT : Before using XLAT instruction, BX contains some address and AL contains some value.

XLAT ; (No operand instruction) → goes to the address BX+AL and stores the content of that address in AL. (BX remains unchanged)
→ So only AL is changed.

Q. I want to push the top 3 words of the stack in AX, BX and CX. The stack should remain as it is, so don't use POP.

Method 1:
```
Mov SI, SP
Mov AX, SS:[SI]
Mov BX, SS:[SI+2]
Mov CX, SS:[SI+4]
```
[We can only use BX, BP, SI, DI for addressing.]

Method 2: (Wrong! Can't use SP)
```
Mov AX, [SP]
Mov BX, [SP+2]
Mov CX, [SP+4]
```
// Here, as we are using [SP], so no need to write SS: because default segment for SP is SS.

Here, in Method 2 we are not changing SP's value. However, it's a good practice that we operate the addresses in stack using BP (so as to ensure that SP doesn't change and always point to the top of the stack).

Method 3:
```
Mov BP, SP      ; Now BP points to the top of the stack
Mov AX, [BP]    ; Again no need to write SS:
Mov BX, [BP+2]
Mov CX, [BP+4]
```

HW Write an Assembly code to sort an array and display in main.

Lec - 10

- Two Dimensional Array : e.g. B DW 10, 20, 30, 40
DW 50, 60, 70, 80
DW 90, 100, 110, 120

Let $[A]_{M \times N}$ matrix.

→ Can be stored in Row major form or column major form.

$$A[i][j] = \begin{cases} A + \{(i-1) \times N + (j-1)\} \times S & \text{(for Row Major)} \\ A + \{(i-1) + (j-1) \times M\} \times S & \text{; (for Column Major)} \end{cases}$$

where S = bytes size. For DW, S = 2
For DB, S = 1

Suppose I have :

```
.DATA
STR1 DB 'HELLO'
STR2 DB 5 DUP(?)
```

Q. How to copy the contents of STR1 into STR2 ?

```
MOV AX, @DATA
MOV DS, AX
MOV ES, AX      ; So that ES also points to the Data segment
                  and ES:DI works correctly
LEA SI, STR1
LEA DI, STR2
CLD             ; DF=0 → for moving left to right
MOV CX,5
REP MOVSB       ; REP MOVSB is similar to L: MOVSB
                                           LOOP L
```

Q. How to copy the contents of STR1 into STR2 in reverse order ? (i.e. we want STR2 = 'OLLEH')

```
LEA SI, STR1+4   ; DB type → So STR+4 for last
LEA DI, STR2
STD
MOV CX,5
L1: MOVSB
ADD DI,2         ; So that DI = DI+1 effectively and SI = SI-1
LOOP L1
```

Now for word type :

Suppose we have : ARR DW 10,20,40,50,60, ?

Q. How to insert 30 in between 20 and 40 ?

→ Shift 40,50,60 to right by 1 place and make space for 30.

```
LEA SI, ARR+08H   ; SI = second last   } Initialize
LEA DI, ARR+0AH   ; DI = last
STD
MOV CX,3
REP MOVSW
MOV WORD PTR [DI], 30
```

e.g. T1 DB 30H, 31H, ..., 46H

```
MOV AL, 0CH     ; 0CH = 12
LEA BX, T1
XLAT            ; goes to T1 + 12 address
                  → value there = 42H
                                = 66 = 'B'
```

(BX is unchanged). ; So sets AL = 'B'.

- String Instructions :

DF (Direction Flag) is used to determine in which direction to move while processing strings.

If DF = 0 → SI and DI move in increasing memory address (i.e. left to right)

If DF = 1 → SI and DI move in decreasing memory address (i.e. right to left)

Command for doing DF = 0 → CLD
Command for doing DF = 1 → STD
} Both are no operand commands.

* MOVSB command → 'SB' → is for byte type string.

MOVSB is also a no operand command.

It copies the data from address DS:SI to ES:DI and then does (SI = SI ±1, DI = DI ±1) or (SI = SI ∓ 1, DI = DI ∓ 1) according to whether DF = 0 or DF = 1.

Note: SI = SI ± (1) → because byte type string.

* MOVSW command → Copies the data from address DS:SI to ES:DI and then does SI = SI ±2, DI = DI ±2 according as DF = 0 or DF = 1. [update by 2 because it is for word type string]

- LODSB → Moves the content of at address DS:SI into AL and SI is incremented / decremented depending on DF = 0 or 1.

[Note: LODSB : DS:SI —to→ AL]
(Source, so SI)

- LODSW → DS:SI —to→ AX and SI ± = 2 depending on DF = 0 or 1.

Q. Write a program / procedure to display the contents of a string, given that BX contains the no. of characters in the string and SI contains starting address of string.

```
.MODEL SMALL
.DATA
   STR1 DB "HELLO"
.STACK 100H
.CODE
   MAIN PROC
      MOV AX, @DATA
      MOV DS, AX
      MOV BX, 5 ------> LEA SI, STR1
      CALL DISP_STR
      MOV AH, 76
      INT 33
   MAIN ENDP

   DISP_STR PROC
      PUSH AX
      PUSH CX ------> PUSH SI
      MOV AH, 2
      MOV CX, BX
      CLD
   L1: LODSB
      MOV DL, AL
      INT 33
      LOOP L1
      POP SI
      POP CX
      POP AX
      RET
   DISP_STR ENDP
END MAIN
```

Lec 12

- STOSB → STOSB ; (No operand instruction of AL) It moves the content at address ES:DI and after that DI will be incremented or decremented by 1, depending on whether DF = 0 or 1.

[Note: STOSB : AL —to→ ES:DI]
(Destination, so DI)

- STOSW → Moves the content of AX into ES:DI and then DI is incremented / decremented by 2 depending on DF = 0 or 1.

[STOSW : AX —to→ ES:DI]

Q. Write a procedure to read a string until carriage return is entered and store in STR1. It can also read backspace, in which case it goes back and removes the last character.

```
READ_STR PROC
   ; input : DI = offset address of string STR1
   ; output: DI = offset address of string STR1
   ;       : BX = # characters in the string.
   PUSH AX
   PUSH DI
   CLD
   XOR BX, BX
   MOV AH, 1
   INT 21H
W1: CMP AL, 0DH
   JE E1
   CMP AL, 08H   ; 08H = ascii of backspace
   JNE E2
   DEC DI
   DEC BX
   JMP R1
E2: STOSB
   INC BX
R1: INT 21H
   JMP W1
E1: POP DI
   POP AX
   RET
READ_STR ENDP
```

We can make val. at last DI after loop ends, as '$' so that backspaced characters are not printed

Lec 13

- CMPSB → Subtracts the content at address DS:SI from the content at address ES:DI and sets the flag register according to the operation. The result of the operation is not stored anywhere.

And, after the operation, both SI and DI are incremented/decremented by 1, depending on DF = 0 or 1.

[CMPSB calculates [ES:DI] - [DS:SI] and sets flag register]

* Similarly we also have CMPSW for word size operations.

e.g.
```
.DATA
    STR1 DB 'ACD'
    STR2 DB 'ABC'
.CODE
    MOV AX, @DATA
    MOV DS, AX
    MOV ES, AX
    CLD
    LEA SI, STR1
    LEA DI, STR2
    CMPSB ;   ZF = 1  and  SI += 1 , DI += 1
    CMPSB ;   ZF = 0  and  SI += 1 , DI += 1
    :
```

* We can also do:

REPE CMPSB ;

→ Loops and performs CMPSB until ZF = 0 (i.e. there is a mismatch) or CX = 0 (loop ends).

- MACROS : Declared as:

```
macro_name MACRO d1, d2, ..., dn   // where d1, d2, ..., dn are parameters that are passed while calling
    ; body
ENDM
```

and are used inside macro.

- SCASB → Subtracts the content at address ES:DI from the content of AL and ~~address~~

- SCASB → Subtracts the content of AL from the content at address ES:DI, does not store the result of subtraction anywhere, but set the flag register depending upon the value of the subtraction.

[SCASB does [ES:DI] = [ES:DI] - AL]

And then DI = DI ± 1 depending on DF = 0 or 1.

e.g.
```
STR1 DB 'ABC'
  :
MOV AX, @DATA
MOV DS, AX
MOV ES, AX
  :
CLD
LEA DI, STR1
MOV AL, 'B'
SCASB ;   ZF = 0  and  DI += 1
SCASB ;   ZF = 1  as STR1[1] = 'B' same as AL.
```

- REPNE SCASB ;
  ↳ Loops and repeatedly performs SCASB until ZF = 1 (i.e. character matched) or CX becomes 0.

Note: First CX is decremented, then the SCASB operation is done.

e.g. for the previous example, if we do:

```
MOV CX, 3
  :
REPNE SCASB ;   // then CX = 1 after this loop ends.
```

* Similar to SCASB, we also have SCASW for word size operations.

is transferred, and also procedures are stored at different memory location.]

So here, multiple instances of the label E1 will be created if we call the macro multiple times, which results in error.

To solve this issue, we use the keyword LOCAL in the macro.

```
GET_BIG MACRO W1,W2
    LOCAL E1
    MOV AX,W1
    :
E1: ENDM
```

// Using the LOCAL keyword, if each time the macro is called, it assigns a unique ID to the label E1.

[Note: If there are multiple labels inside a macro, we can write LOCAL E1, E2, ...]

* We can also call a macro from inside another macro.

e.g.
```
SAVE_REGS MACRO R1,R2,R3
    PUSH R1
    PUSH R2
    PUSH R3
ENDM

RESTORE_REGS MACRO S1,S2,S3
    POP S1
    POP S2
    POP S3
ENDM

COPY MACRO SOU, DES, LEN
    SAVE_REGS CX, SI, DI
    LEA SI, SOU
    LEA DI, DES
    CLD
    MOV CX, LEN
    REP MOVSB
    RESTORE_REGS DI,SI,CX
ENDM
    :
COPY A, B, 10   ; will copy first 10 bytes of A into B.
```

[We can write all macros in a file MACROS and include
INCLUDE C:\MACROS]

* How to call a macro?

```
macro_name a1,a2,...,an   // a1, a2, ... are actual parameters
                          // d1, d2, ..., dn are formal parameters.
```

* Note that, we can pass parameters to macros but not to procedures.
* Also, macros are much faster than procedures.

e.g.
```
MOVW MACRO W1,W2
    PUSH W2
    PUSH W1
    ENDM
```

```
    :
MOVW A,B   ; The macro replaces this part of the program.
```

e.g. We can't write XCHG A,B for two memory variables A and B. But, we can do this by writing a macro:

```
TITLE PG1: Macro Demo
.MODEL SMALL
EXCH MACRO W1,W2
    PUSH AX
    MOV AX,W1
    XCHG AX,W2
    MOV W1,AX
    POP AX
ENDM
```

```
.STACK 100H
.DATA
    A DW 10
    B DW 5
.CODE
MAIN PROC
    MOV AX,@DATA
    MOV DS,AX
    EXCH A,B    ; swaps the contents of A and B.
    :
```

e.g.
```
GET_BIG MACRO W1,W2
    MOV AX,W1
    CMP AX,W2
    JG E1
    MOV AX,W2
E1: ENDM
```

```
.CODE
    :
GET_BIG A,B
    :
GET_BIG C,D
```

The above code will give error if the macro GET_BIG is called multiple times, because in calling macros, the body of the macro replaces the portion where the macro is called. [Note that, in procedures, the control of the program

e.g.
```
SAVE_REG MACRO REGS
    IRP D, <REGS>
        PUSH D
    ENDM
ENDM
```

Then, it is called using:

```
SAVE_REG <AX, BX, CX, DX>
```

↓

This is equivalent to:

```
PUSH AX
PUSH BX
PUSH CX
PUSH DX
```

* The basic difference in IRP is that here, in IRP, the no. of parameters is not fixed.

Example: Hex output using macros, (using REPT and IRP)

[See the algorithm for Hex output earlier]

```
TITLE PG1 : HEX OUTPUT
.MODEL SMALL
SAVE_REGS MACRO REGS
    IRP D, <REGS>
    PUSH D
    ENDM
ENDM
RESTORE_REGS MACRO REGS
    IRP D, <REGS>
    POP D
    ENDM
ENDM
CONVERT_TO_CHAR MACRO BYT
    LOCAL E1, E2
    CMP BYT, 9
    JNLE E1
    OR BYT, 30H
    JMP E2
E1: ADD BYT, 37H
E2: ENDM
DISP_CHAR MACRO BYT
    PUSH AX
    MOV AH,2
    MOV DL, BYT
    INT 21H
    POP AX
ENDM

HEX_OUT MACRO WRD
    SAVE_REGS <BX,CX, DX>
    MOV BX, WRD
    MOV CL,4
    REPT 4
      MOV DL, BH
      SHR DL, CL
      CONVERT_TO_CHAR DL
      DISP_CHAR DL
      ROL BX,CL
    ENDM
    RESTORE_REGS <DX, CX, BX>
ENDM

.STACK 100H
.CODE
MAIN PROC
    :
    HEX_OUT AX
    :
MAIN ENDP
END MAIN
```

We can use macros for writing the lines of code which can be used repetitively.

e.g.
```
DOS_RTN MACRO          (DOS return)
    MOV AH, 4CH
    INT 21H
ENDM
```

or,

```
NEWLINE MACRO
    MOV AH, 2
    MOV DL, 0DH
    INT 21H
    MOV DL, 0AH
    INT 21H
ENDM
```

Lec 14

- REPT → The REPT macro is used to repeat a block of statements.

```
REPT expression
    ; statements
ENDM
```

e.g.
```
FACT MACRO N
    M = 1
    FACTO = 1
    REPT N
    DW FACTO
    M = M+1
    FAC = M* FACTO
    ENDM
ENDM
```

// we can use '=' symbol etc. inside any macro.

[REPT and IRP can only be used inside a macro.]

- IRP → Syntax:
```
IRP d, <a1, a2, ..., an>
    ; statements
ENDM
```

* The IRP instruction expands the statements n times and in its expansion, it replaces d by a_i.

e.g.

```
TITLE PG1 : ERR DEMO
.MODEL SMALL
.STACK 100H
DISP_CH MACRO CH
    IFNB <CH>
    MOV AH,2
    MOV DL,CH
    INT 21H
    ELSE
    .ERR
    ENDIF
ENDM
```

```
.CODE
MAIN PROC
    DISP_CH 'A'
    DISP_CH     // stops the execution and gives forced error.
    :
```

Lec 15

```
procedure_name PROC type    // How to declare a procedure
```

where, type = NEAR or FAR

NEAR, if the procedure and from where it is called are in the same segment.

FAR, if the procedure and from where it is called are in different segment.

- EXTRN → EXTRN external_name_list

  name : type → NEAR / FAR (for procedure) / BYTE / WORD (for global variables)

- PUBLIC → PUBLIC name_list

e.g. we can write the following code : Creating two different files PG1.asm and PG1A.asm

(Using EXTRN and PUBLIC is just another way, if we don't wanna use INCLUDE)

- Conditional Pseudo-operations :

Form	True if
IF exp	exp value is non zero
IFE exp	exp value is zero
IFB <arg>	arguments is missing / blank
IFNB <arg>	arguments is not missing / not blank.
IFDEF sym	sym is defined in the program, or declared as some external variable.
IFNDEF sym.	sym is not defined in the program, or not declared as external variable.
IFIDN <str1>, <str2>	strings str1 and str2 are identical
IFDIF <str1>, <str2>	strings str1 and str2 are different / not identical.

e.g.

```
BLOCK MACRO N,K
    I = 1
    REPT N
        IF K+1-I
        DW I
        I = I+1
        ELSE
        DW 0
        ENDIF
    ENDM
ENDM
:
A1 LABEL WORD      ; means A1 is a word type variable
BLOCK 10,5         ; getting values from this line.
```

// Result of the program will be :

A1 DW 1,2,3,4,5,0,0,0,0,0

- .ERR directive → If a macro is not called with proper no. of arguments then it stops the execution and gives a forced error.

PG1.asm

```
TITLE PG1 : CASE
EXTRN CONVERT
.MODEL SMALL
.STACK 100H
.DATA
    MSG DB 'Enter a lower case
                letter : $'
.CODE
    MAIN PROC
        MOV AX, @data
        MOV DS, AX
        MOV AH,9
        LEA DX, MSG
        INT 21H
        MOV AH,1
        INT 21H
        CALL CONVERT
        MOV AH, 4CH
        INT 21H
    MAIN ENDP
END MAIN
```

PG1A.asm

```
TITLE PG1A : CONVERT
PUBLIC CONVERT
.MODEL SMALL
.DATA
    MSG DB 0DH,0AH, 'In upper case
                it is :'
    CHAR DB , -20H , '$'
.CODE
    CONVERT PROC
        PUSH BX
        PUSH DX
        ADD CHAR, AL
        MOV AH,9
        LEA DX, MSG
        INT 21H
        POP DX
        POP BX
        RET
    CONVERT ENDP
END
```

8. How to execute PG1 in such a case ?

```
MASM PG1 ;
MASM PG1A ;
LINK PG1 + PG1A ;
PG1
```