## Push / Pop

→ They store and retrieve data from LIFO stack memory.
→ 6 forms of PUSH and POP instructions
1. Register
2. Memory
3. Immediate
4. Segment register
5. Flags
6. All registers.

① Register → allows contents of any 16-bit register to be transferred to or from the stack.

② Memory addressing → store the contents of 16-bit memory location (32 bit on 80386) on the stack or stack data into memory location

③ Immediate addressing → allows immediate data to be pushed onto the stack but not popped off the stack.

④ Segment register addressing → allows contents of any segment register to be pushed onto the stack or removed from stack.

⑤ Flags & contents of all registers may be pushed or popped

## Push

① 8086-80286 Push instruction always transfers 2 bytes of data to the stack

80386 & above → transfer 2 or 4 bytes depending on registers or size of memory location.

PUSHA → this instruction copies the contents of internal register set except segment register, to the stack.

PUSHA (push All) instruction copies the registers to stack in following order: AX, CX, DX, BX, SP, BP, SI & DI.

PUSHF (push Flags) copies the contents of flag register to stack.

PUSHAD and POPAD → push and pop the contents of 32-bit register set found in 80386 through Pentium 4.

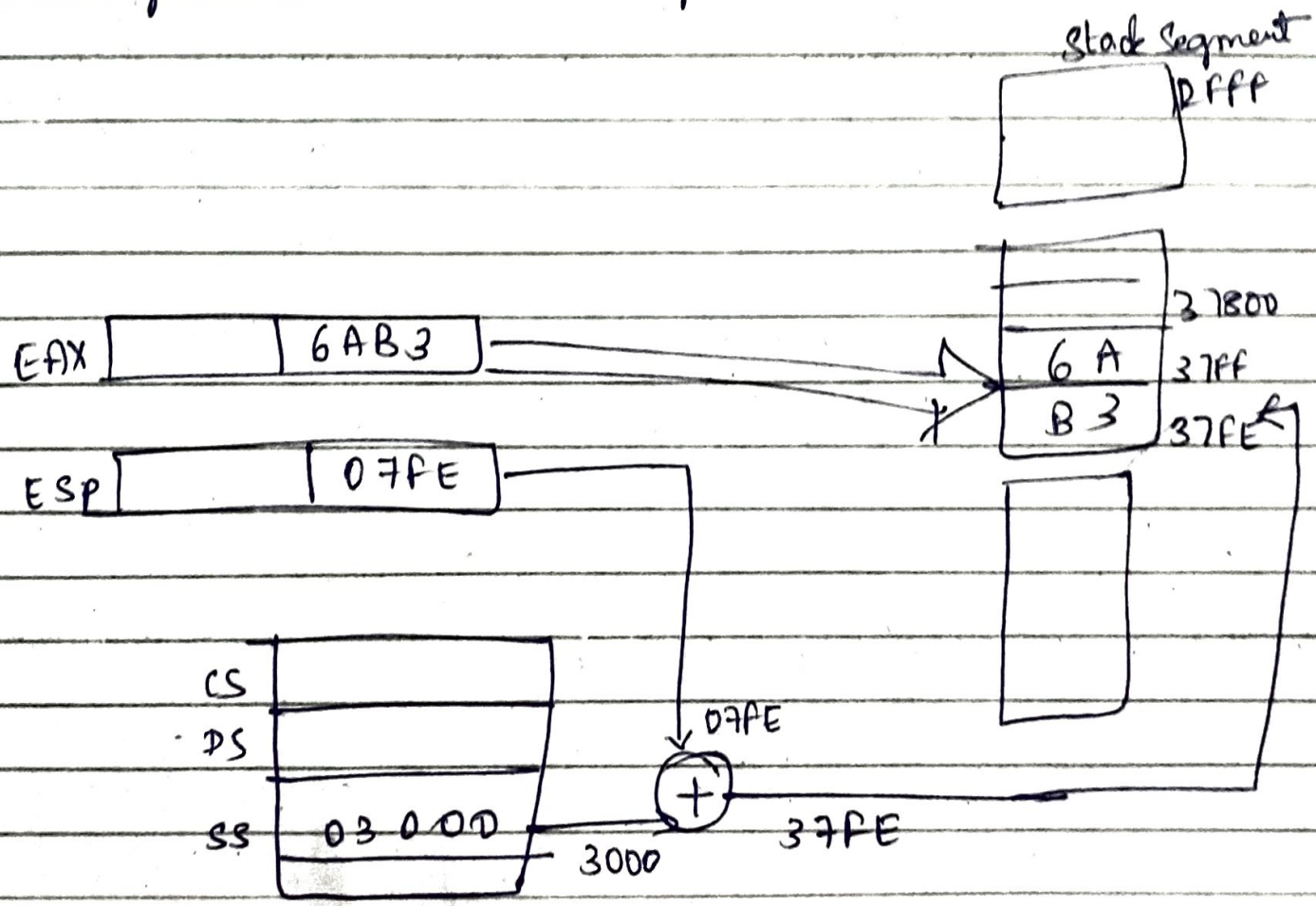
Operation:

① wherever data are pushed onto stack, first (most significant) data byte moves to stack segment memory location addressed by SP-1

② Second (least significant) data byte moves into stack

Segment memory location addressed by SP-2.

③ After data is stored by a PUSH, the contents of SP register decremented by 2.




## PUSH AX

Instruction PUSH AX copies contents of AX onto the stack where address SS:[SP-1] = AH, SS:[SP-2] = AL. and afterwards SP = SP-2.

## POP

It performs the inverse operation of PUSH instruction. POP instruction removes data from the stack and places it into target place.

POPF (Pop flags) removes a 16-bit number from stack and places it into flag register

POPFD → removes 32-bit number from stack and places it into flag registers.

POPA (pop all) → removes 16 bytes of data from the stack and places into following registers in the order DI, SI, BP, SP, BX, DX, CX & AX.

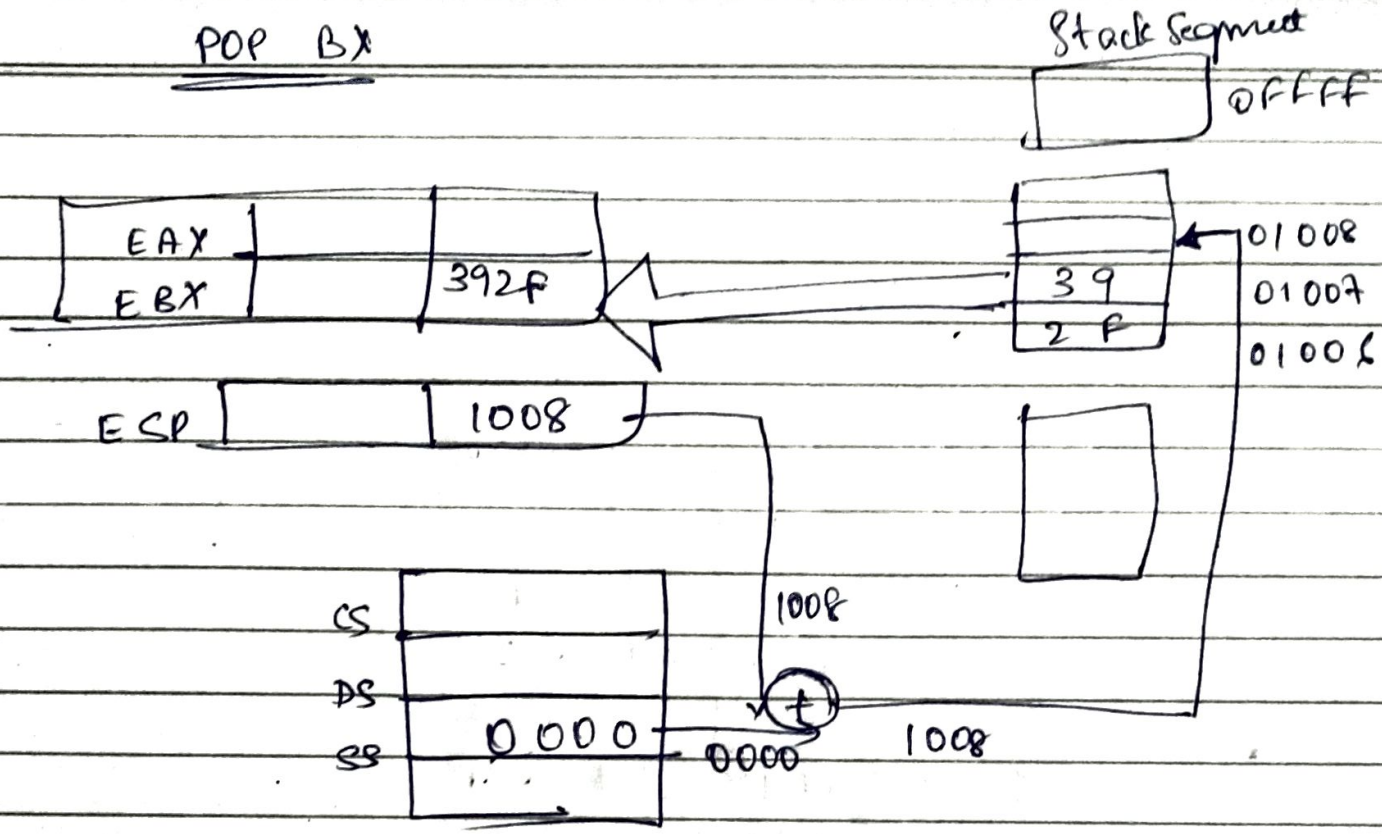
**Operation:** POP BX

① First byte of data removed from stack moves into register BL.

② Second byte is removed from stack memory location SP+1 and placed into register BH.

③ After both bytes are removed from stack, SP register is incremented by 2.

## POP BX




## load Effective Address

→ LEA loads any 16-bit register with offset address as determined by addressing mode selected for the instructions.

LEA instruction loads a 16 or 32-bit register with offset address of the data specified by the operand.

Eg: lea AX, NUMB

Here operand address NUMB is loaded into register AX, not the contents of NUMB.

LEA BX, [DI] → loads offset address specified by [DI] into BX.

MOV BX, [DI] → loads data stored at memory location addressed by [DI] into BX.

① MOV BX, OFFSET LIST → performs same function
lea BX, LIST

Eg:
```
• model small
• data
  data1 dw 2000H
  data2 dw 3000H
• Code
  lea SI, data1
  mov DI, offset DATA2
  mov BX, [SI]
  mov CX, [DI]
  MOV [SI], CX
  MOV [DI], BX
end.
```

## why LEA instruction when OFFSET directive is there

① OFFSET only functions with simple operands such as LIST. It may not be used for operands such as [DI], LIST[SI] etc.

② OFFSET directive is more efficient than LEA instruction for simple operands.

③ It takes microprocessor longer to execute LEA BX, LIST instruction than MOV BX, OFFSET LIST.

LEA BX, LIST → 8086 requires 2 clock cycles

MOV BX, OFFSET LIST → only one clock cycle.

## LDS, LES, LFS, LGS and LSS

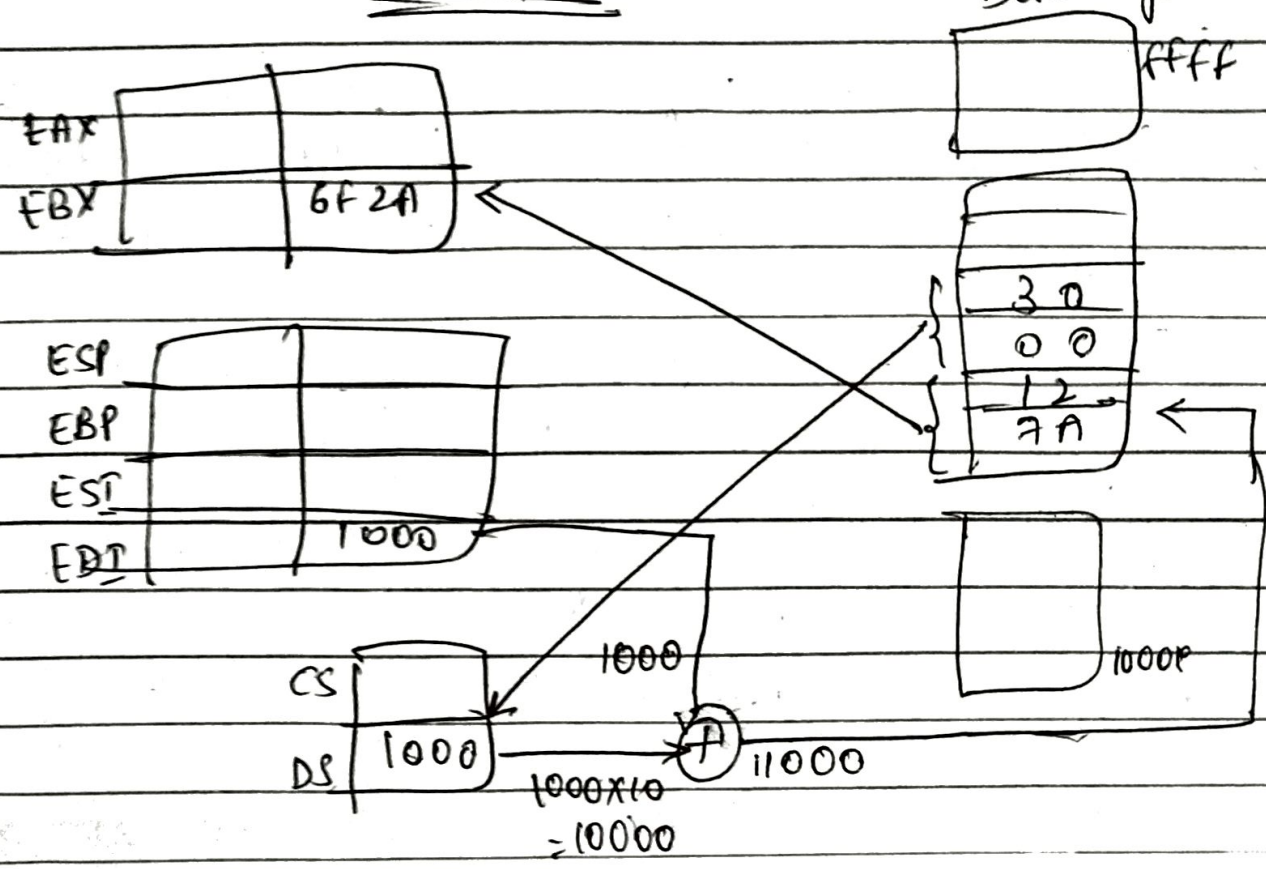
These instructions load any 16-bit or 32-bit register with an offset address and DS, ES, FS, GS and SS segment register with a segment address.

Eg: LDS BX, [DI] → transfers 32-bit number, addressed by the DI in the data segment, into BX and DS registers.

BX = 6F2A
DI = 1000
DS = 1000

→ operation:

## LDS BX, [DI]




## String Data Transfers

There are 5 string data transfer instructions

1. LODS
2. STOS
3. MOVS
4. INS
5. OUTS

## Direction flag

→ D flag present in flag register selects auto increment or auto decrement operation for DI and SI registers during string operations.

D = 0 → auto increment

D = 1 → auto decrement

* D flag is used only with string instructions.

* The CLD instruction clears D flag (D = 0) Hence auto increment
STD instruction sets D flag (D = 1) Hence auto decrement

## DI and SI

* During string instruction execution, memory access occurs through either or both DI and SI registers.

* DI offset address accesses data in extra segment for all string instructions
SI offset address access data, by default, in data segment.

## LODS

* The LODS instruction loads AL, AX and EAX with data stored at data segment offset address indexed by SI register.

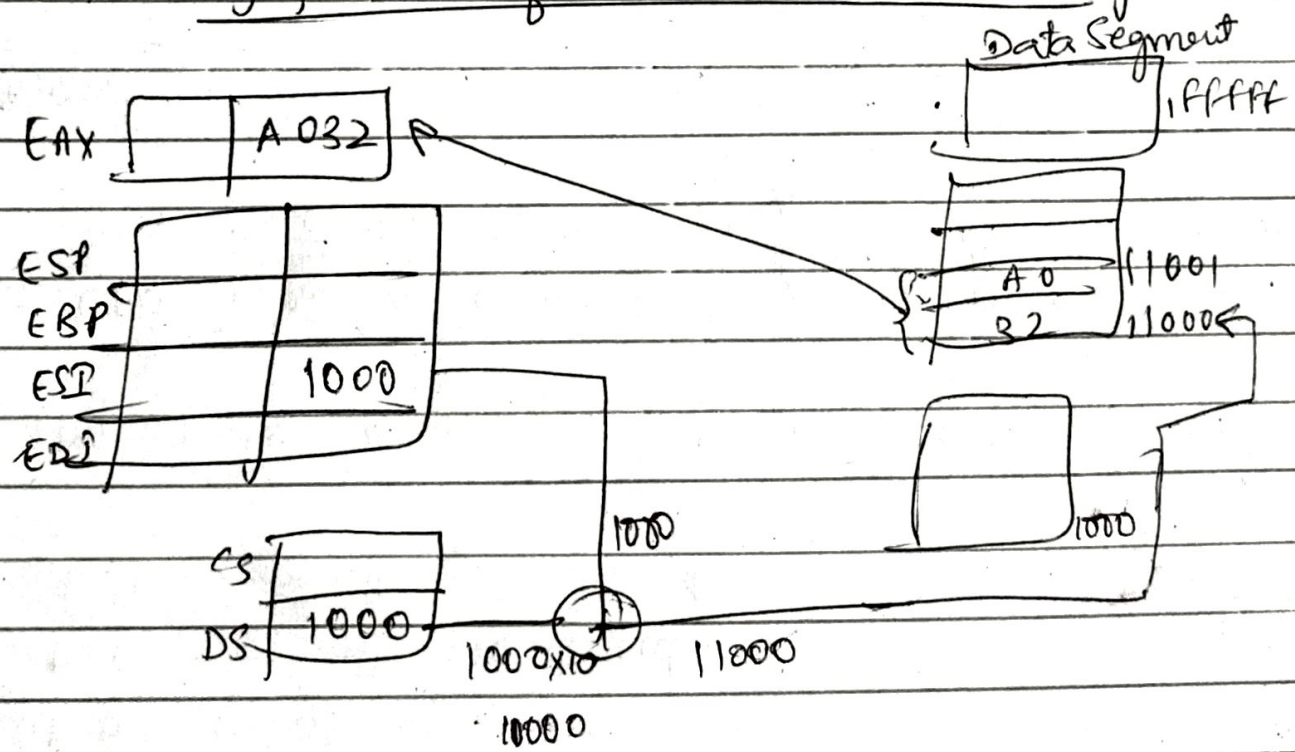
* After loading AL with a byte, AX with a word or EAX with a doubleword, contents of SI increment if D=0 or decrement if D=1.

eg: LODSB  AL = DS: [SI]; SI = SI ± 1

LODSW  AX = DS: [SI]; SI = SI ± 2

LODS LIST  AL = DS: [SI]; SI ± 1 (if LIST is a byte

Operation of LODSW if DS=1000H, D=0,
This instruction is shown after AX is loaded from memory, but before SI incremented by 2



## STOS

① The instruction stores AL, AX or EAX at extra segment memory location addressed by DI register.

Eg: STOSB  AL = ES: [DI] ; DI = DI ± 1

STOSW

STOS LIST

## STOS with a REP

The repeat prefix (REP) is added to any string data transfer instruction, except LODS instruction.

* REP prefix causes CX to decrement by 1 each time the string instruction executes. After CX decrements, the string instruction repeats.
  If CX reaches a value of 0, the instruction terminates and the program continues with next sequential instruction.

Eg: If CX is loaded with 100, and REP STOSB instruction executes, µp automatically repeates STOSB instruction 100 times.

## MOVS

* It transfers data from one memory location to another.

* MOVS instruction transfers a byte, a word or doubleword from data segment location addressed by SI to extra segment location addressed by DI.

Eg: MOV AL, 6+3 — Copies 9 to AL
MOV AL, 4*3 — Copies 12 TO AL
MOV AX, 12/5 — Copies 2 to AX (remainder is lost)

Forms of MOVS instruction

① MOVSB — ES: [DI] = DS [SI]; DI = DI ± 1, SI = SI ± 1
MOVSW
MOVSD

## INS

* INS (Input String) Instruction (not found in 8086/8088) transfers a byte, word or doubleword of data from an I/O device into extra segment memory location addressed by DI register.

Three basic forms of INS.

① INSB → Inputs data from an 8-bit I/O device and stores it in a byte-sized memory location indexed by DI.

② INSW

③ INSD.

I/O device address.

Eg: INSB ES:[DI] = [DX]; DI = DI ± 1 (byte transferred)
INSW

OUTS

OUTS (output string) → not available in 8086 / 8088 → transfers a byte, word or doubleword of data from data segment memory location addressed by SI to an I/O device.

Eg: OUTSB [DX] = DS:[SI]; SI = SI ± 1
OUTSW
OUTSD

## Miscellaneous data transfer instructions

### ① XCHG

→ It exchanges the contents of a register with the contents of any other register or memory location.

→ Exchange instruction cannot exchange segment registers or memory-to-memory data.

| | | |
|---|---|---|
| Eg: | XCHG AL, CL | Exchanges contents of AL with CL |
| | XCHG AL, DATA2 | Exchange contents of AL with data segment memory location DATA2. |

### ② LAHF and SAHF

* LAHF instruction transfers rightmost 8 bits of flag register into AH register.

* SAHF instruction transfers AH register into rightmost 8 bits of the flag register.

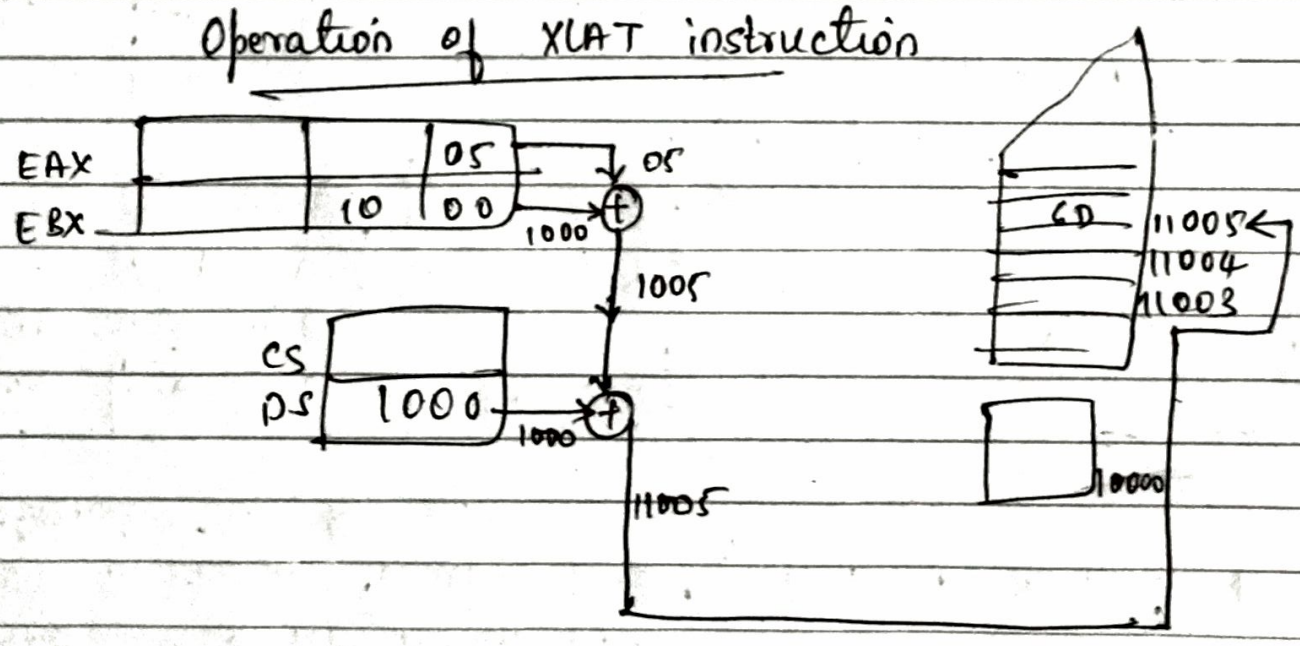
③ XLAT (Translate)

→ This instruction converts contents of AL register into a number stored in a memory table.

* It performs direct lookup technique used to convert one code to another.

* XLAT first adds the contents of AL to BX to form a memory address within data segment.
It then copies the contents of this address into AL.
This is the only instruction that adds a 8-bit number to a 16-bit number.

Operation of XLAT instruction




6DH is loaded into AL.

## ④ IN and OUT

→ These instructions perform I/O operation

* IN instruction transfers data from an external I/O device into AL, AX or EAX
* OUT instruction transfers data from AL, AX or EAX to an external I/O device.

Two forms of I/O device (port) addressing exist for IN and OUT

① Fixed port addressing
② Variable port addressing

* Fixed port allows data transfer between AL, AX or EAX using 8 bit I/O port address.
  It is called fixed port addressing because port number follows instructions opcode.

* Variable port addressing allows data transfers between AL, AX or EAX and a 16-bit port address.
  It is called variable port addressing because the I/O port number is stored in register DX, which can be changed during execution of a program.

Eg: IN AL, p8 → 8 bits are input to AL from I/O port p8

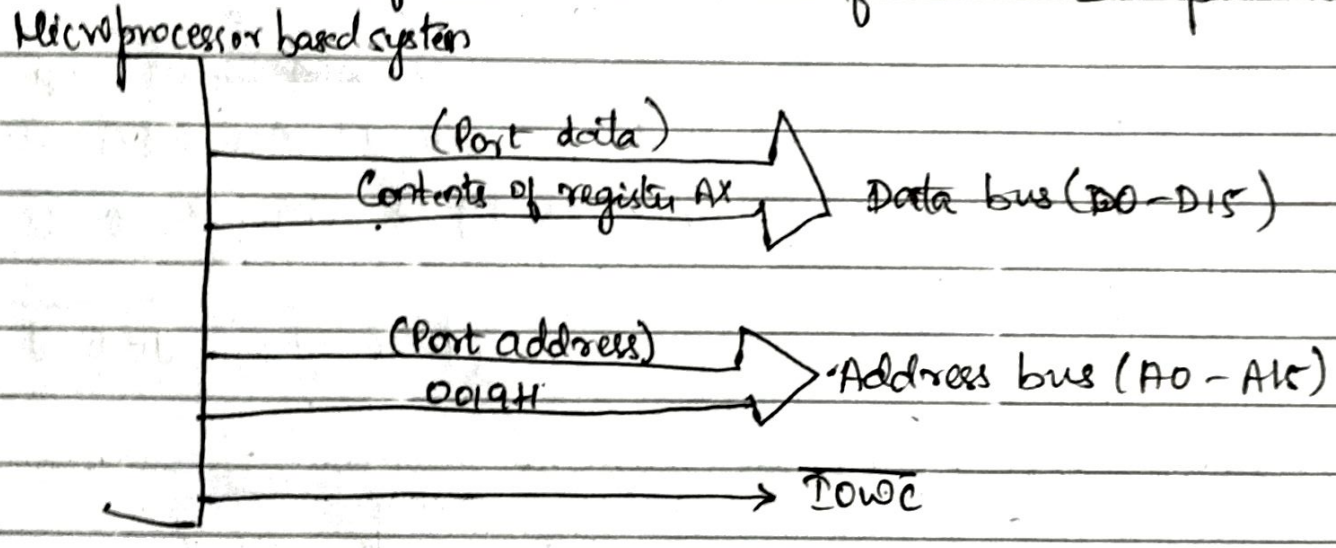
IN EAX, p8 → 32 bits are input to EAX from I/O port p8

IN AX, DX → 16 bits are input to AX from I/O port DX.

OUT p8, AX → 16 bits are output to I/O port p8 from AX

OUT 19H, AX

It transfers the contents of AX to I/O port 19H.




* It transfers contents of AX to I/O port 19H. I/O port numbers appears as a 0019H on 16-bit address bus and that the data from AX appears on the data bus of the μp.
System control signal $\overline{IOWC}$ (I/O write control) is a logic zero to enable the I/O device.

## (5) MOVSX and MOVZX

* MOVSX (Move and sign-extend)
MOVZX (Move and zero-extend)
These instructions move data and at the same time either

sign or zero extend it.

Eg: MOVSX CX, BL — Sign extend BL into CX

MOVSX BX, DATA1 — Sign extend the byte at DATA1 into BX

MOVZX DX, AL — Zero extend AL into DX

MOVZX EBP, DI — Zero extends DI into EBP.

A number is sign-extended when its sign-bit is copied into most significant part.

Eg: if an 8 bit 84H is sign extended into a 16-bit number, it becomes FF84H.

Sign bit of 84H is 1

## (6) BSWAP (Byte Swap)

* It is available only in 80486 & Pentium 4 µP.
* It takes the contents of any 32 bit register and swaps the first byte with the fourth and second with third.

Eg: BSWAP EAX with EAX = 00112233H swaps bytes in EAX, resulting in EAX = 33221100H

## ⑦ CMOV (Conditional MOV)

* These instructions move data only if the condition is true.

Eg:

| | | |
|---|---|---|
| CMOVB | C = 1 | Move if below |
| CMOVAE | C = 0 | Move if above or equal |
| CMOVL | S! = 0 | Move if less than |

## Segment Override Prefix

* It allows the programmer to deviate from default segment
* It is an additional byte that affords the front of an instruction to select an alternate segment register.

Eg: MOV AX, [DI] instruction accesses data within the data segment by default.

If required by a program, this can be changed by prefixing the instruction.

Eg: MOV AX, ES:[DI]

| | Segment accessed | Default segment |
|---|---|---|
| ① MOV AX, DS:[BP] | DS | SS |
| ② MOV ES:[SI], AX | ES | DS |

# Arithmetic & logic Instructions

## Addition

1. Add instruction with 8, 16, 32 bit binary addition.
2. Add-with-Carry (ADC)
3. INC instruction

### Examples of addition instructions

1. ADD AL, BL — AL = AL + BL
2. ADD CL, 44H — CL = CL + 44H
3. ADD [BX], AL — AL adds to byte contents of data segment memory location addressed by BX with the sum stored in same memory location.

* Only types of addition not allowed are memory-to-memory and segment register.

### Register Addition

* Adds the contents of several registers.

```
ADD AX, BX
ADD AX, CX
ADD AX, DX
```

* whenever arithmetic and logic instructions execute, the contents of flag register change.
* Only flags located in the rightmost 8 bits of flag register and overflow flag change.

  These rightmost flags denote the result of arithmetic or logic operation.

## Immediate Addition

* It is used when constant or known data are added.

```
MOV DL, 12H
ADD DL, 33H
```

* DL is first loaded with 12H by using an immediate move instruction.
* Next, 33H is added to 12H in DL by an immediate addition instruction
* After addition, the sum (45H) moves into register DL and flag changes as follows.

Z = 0 (result not zero)
C = 0 (no carry)
A = 0 (no half carry)
S = 0 (result positive)
P = 0 (odd parity)
O = 0 (no overflow)

## Memory-to-Register Addition

* Used when memory data is added to register.

```
MOV DI, OFFSET NUMB ; address NUMB
MOV AL, 0           ; clear sum
ADD AL, [DI]        ; add NUMB
ADD. AL, [DI+1]     ; add NUMB+1
```

* First instruction loads destination index register (DI) with offset address NUMB.
* ~~DX~~ register After clearing the sum to zero,

ADD AL, [DI] instruction adds the contents of memory location NUMB to AL.

* ADD AL, [DI + 1] instruction adds the contents of memory location NUMB plus 1 byte to AL register.
* After both ADD instructions execute, the result appears in AL register as the sum of the contents of NUMB plus the contents of NUMB + 1.

<u>Array addition</u>

* Used for adding contents of array elements.

```
MOV AL, 0              ; clear sum
MOV SI, 3              ; address element 3
ADD AL, ARRAY[SI]      ; add element 3
ADD AL, ARRAY[SI+2]    ; add element 5
ADD AL, ARRAY[SI+4]    ; add element 7.
```

* First clears AL to 0 so it can be used to accumulate the sum.
* Register SI is loaded with a 3 to initially address array element 3.
* ADD AL, ARRAY[SI], adds contents of array element 3 to sum in AL.
* It then adds array elements 5 and 7 to sum in AL.

## Increment Addition

* INC adds 1 to a register or a memory location
* The INC instruction adds 1 to any register or memory location, except a segment register.

Examples of increment instructions

① INC BL — BL = BL + 1

② INC EAX — EAX = EAX + 1

③ INC BYTE PTR[BX] — Adds 1 to the byte contents of the data segment memory location addressed by BX.

④ INC DATA1 — Adds 1 to the contents of data segment memory location DATA1.

* The assembler program cannot determine if INC [DI] instruction is given as whether it is a byte, word or doubleword sized increment. Hence it must be described by using BYTE PTR, WORD PTR, DWORD PTR or QWORD PTR directives.

```
MOV DI, OFFSET NUMB      ; address NUMB
MOV AL, 0                ; clear sum
ADD AL, [DI]             ; add NUMB
INC DI                   ; increment DI
ADD AL, [DI]             ; add NUMB + 1
```

⇓

modified instructions of the example given for memory-to-register addition.

## Addition with Carry (ADC)

* It adds the bit in Carry flag to the operand data.

Examples:

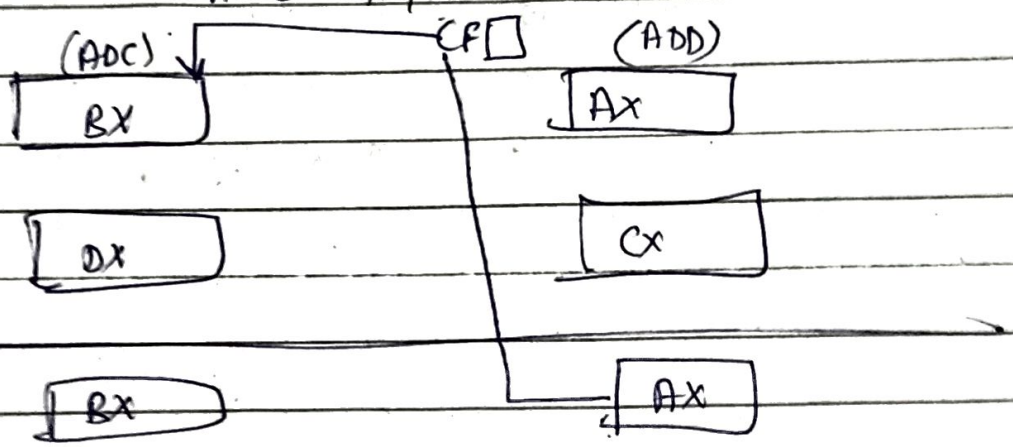
① ADC AL, AH — AL = AL + AH + Carry

② ADC RBX, 0 — RBX = RBX + 0 + Carry

③ ADC ECX, [EBX] — doubleword contents of data segment memory location addressed by EBX add to ECX with sum stored in ECX.

ADD AX, CX

ADC BX, DX




## Exchange and ADD (XADD)

* It adds the source to the destination and stores the sum in destination. After addition, the original value of destination is copied into source operand.

Eg: If BL = 12H and DL = 02H

XADD BL, DL

BL register contains 14H and DL becomes 12H.

## Subtraction (SUB)

① Used with any 8, 16 or 32 bit data.
② Decrement (DEC)
③ SBB (Subtract-with-borrow)

## Register Subtraction

SUB BX, CX
SUB BX, DX

It subtracts the 16-bit contents of registers CX and DX from the contents of register BX.

* After each subtraction, the µp modifies the contents of flag register.

Eg: SUB CL, BL  CL = CL - BL
SUB DH, 6FH  DH = DH - 6FH

## Immediate Subtraction

* It allows immediate operands for the subtraction of constant data.

MOV CH, 22H
SUB CH, 44H

* First load the 22H into CH using immediate move. Next SUB instruction, using immediate data 44H, subtract 44H from 22H. After subtraction, difference

moves into CH register.

The flags change as follows for this subtraction

Z = 0 (result not zero)
C = 1 (borrow)
A = 1 (half - borrow)
S = 1 (result negative)
P = 1 (even parity)
O = 0 (no overflow)

Here 44H (+68) subtracted from 22H (+34)
Result = -34 (0DEH)

<u>Decrement Subtraction</u>

* It subtracts 1 from a register or the contents of a memory location.

<u>Eg:</u> DEC BH  BH = BH - 1
DEC R14  R14 = R14 - 1

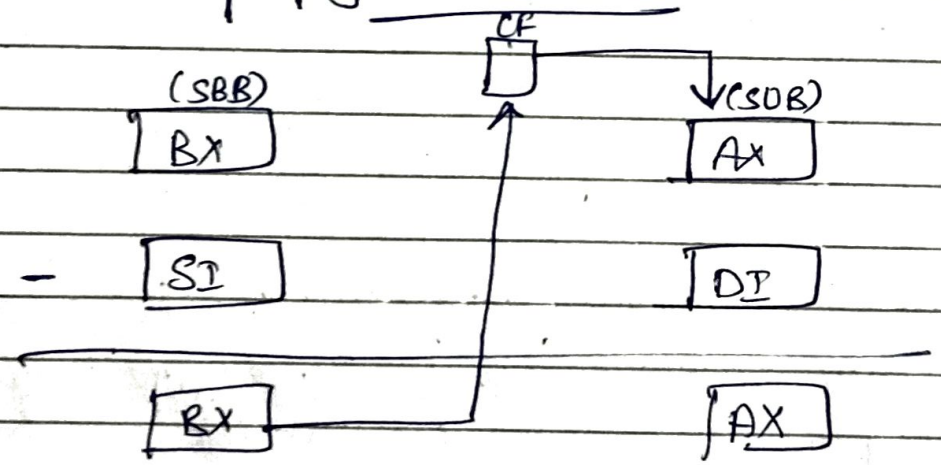
* Decrement indirect memory data instructions require BYTE PTR, WORD PTR, DWORD PTR or QWORD PTR

## Subtraction with Borrow (SBB)

- SBB instruction functions as a regular subtraction except the carry flag (C), which holds the borrow, also subtracts from the difference.

Eg: SBB AH, AL = AH = AH - AL - Carry
SBB CL, 2 CL = CL - 2 - Carry

Subtraction with borrow showing how Carry flag propagate the borrow




## Comparison (CMP)

- It is a subtraction that changes only flag bits, the destination operand never changes.

- A CMP is normally followed by a conditional jump instruction, which tests the condition of flag bits.

Eg: CMP CL, BL CL - BL
CMP AX, 2000H AX - 2000H

```
CMP AL, 10H
JAE SUBER
```

* The contents of AL are compared with 10H. Conditional JAE instruction follows the comparison. It causes the program to continue at memory location SUBER if the value in AL is 10H or above.

Compare and Exchange (CMPXCHG)

* It compares destination operand with the accumulator. If they are equal, the source operand is copied into the destination, if they are not equal, the destination operand is copied into accumulator.

Eg: CMPXCHG CX, DX

* The instruction first compares the contents of CX with AX. If CX equals AX, DX is copied into AX. If CX is not equal to AX, CX is copied into AX.

# Multiplication and Division

## Multiplication

* It is performed on bytes, words or doublewords.
* Signed integer multiplication (IMUL)
  Unsigned integer multiplication (MUL)
* Product after multiplication is always a double-width product
* If two 8-bit number are multiplied, a 16-bit product is generated. It is same for 16 and 32-bit numbers.

* Some flag bits (overflow and carry) change when the multiply instruction executes and produce predictable outcomes.
* flag bits show that the result is 8 bits wide (C=0) or 16-bits wide (C=1)

## 8-bit Multiplication

* Here, the multiplicand is always in AL register, whether signed or unsigned.
* Multiplier can be any 8-bit register or any memory location.
* The multiplication instruction contains one operand because it always multiplies the operand times the contents of register AL.

Eg: MUL BL

It multiplies unsigned contents of AL by the unsigned contents of BL.

After multiplication, unsigned product is placed in AX — a double width product.

Eg:

```
MOV BL, 5        } load data
MOV CL, 10
MOV AL, CL       ; position data
MUL BL           ; multiply
MOV DX, AX       ; position product
```

## 16-bit Multiplication

* Here multiplicand is in AX.
* 32-bit product appears in DX-AX
  DX → contains most significant 16 bits of the product
  AX → contains least significant 16 bits.

Eg: MUL CX → AX is multiplied by CX, product is in DX-AX
IMUL DI → AX is multiplied by DI, signed product is in DX-AX.

IMUL CX, DX, 12H → it multiplies 12H times DX and leaves a 16-bit signed product in CX.

## 32-bit multiplication

* 32 bit multiplication can be signed or unsigned using IMUL and MUL instructions
* Contents of EAX are multiplied by the operand specified with the instruction.
* Product is found in EDX-EAX where EAX contains least significant 32 bits of the product.

Eg: MUL ECX  EAX is multiplied by ECX.
Unsigned product is in EDX-EAX.

IMUL EDI  EAX is multiplied by EDI
Signed product is in EDX-EAX.

## 64-bit multiplication

* Result of 64-bit multiplication appears in RDX.

Eg: MUL RCX  RAX is multiplied by RCX.
Unsigned product is in RDX-RAX.

## Division

1) Occurs on 8 or 16-bit numbers.
These numbers are signed (IDIV) or unsigned (DIV) integers.
* Dividend is always a double width dividend that is divided by operand.
8 bit division divides a 16-bit number by an 8-bit number.

## 8-bit Division

* It uses AX register to store dividend that is divided by Content of any 8-bit register or memory location.
* Quotient moves into AL.
  AH contains remainder.

Eg: If AX = 0010H (+16) and BL = 0FDH (-3)
IDIV BL executes AX = 01FBH.
This represents a quotient of -5 (AL) with remainder (AH).

$-16 / +3$ = Quotient = -5 (AL)
Remainder = -1 (AH)

Eg: DIV CL  AX is divided by CL
Unsigned quotient is in AL and unsigned remainder is in AH.

## 16-bit division

* Same as 8-bit division except that instead of dividing into AX, 16-bit number is divided into $\underbrace{DX - AX}_{\text{32-bit dividend.}}$
* Quotient appears in AX and remainder in DX.

Eg: DIV CX : DX-AX is divided by CX
Unsigned quotient is AX and
unsigned remainder in DX.

## 32-bit division

* 64-bit contents of EDX-EAX are divided by operand specified by the instruction, leaving 32-bit quotient in EAX and 32-bit remainder in EDX.

Eg: DIV ECX — EDX-EAX is divided by ECX
Unsigned quotient in EAX
Unsigned remainder in EDX.

## 64-bit division

* It uses RDX : RAX register pair to hold dividend and quotient in RAX and remainder in RDX.

Eg: DIV RCX — RDX-RAX is divided by RCX.
Unsigned quotient is in RAX
Unsigned remainder is in RDX.

# BCD and ASCII Arithmetic

① ASCII representation

* Numbers are stored as a string of ASCII characters

Example: 1234 is stored as 31 32 33 34 H

## AAA

1) used after the addition of two ASCII numbers or two unpacked BCD numbers.

working

## AAA

* Used after the addition of two ASCII numbers or two unpacked BCD numbers.

**Working**

① If lower 4 bits of AL > 9 or Auxiliary flag = 1, then

(AL) ← AL + 06

(AH) ← (AH) + 01

Initialize AH = 0 before this step.

C = AF = 1

② clear MS 4 bits of AL register

(AL) ← (AL) and 0FH

**Example:** Two ASCII numbers 39 and 36 are added by microprocessor and result is adjusted by AAA instruction.

Two numbers are added as below

| | | |
|---|---|---|
| 39 H | C 0011 | AC 1001 |
| 36 H | 0011 | 0110 |
| 6FH | 0110 | 1111 |
| Hexa decimal addition | 6 | F result |
| | | Binary addition |

PAGE :
DATE : / /

The above result is not accepted. The above result can be adjusted by using AAA instruction as follows.

**Step1:** AL = 6FH + 06 = 75H
AH = AH + 01 AH = 1
C = AF = 1

**Step2:** clear M.S 4 bits of AL register
AL = AL and 0FH
75H and 0FH
05H

| | AH | AL | AC | C | |
|---|---|---|---|---|---|
| AX | 01 | 05H | 1 | 1 | Flag |

**Example 2:** 39 and 38

Expected result = 39+38 = 0107
[39 and 38 = 09 and 08 in BCD
09 + 08 = 0107 in unpacked BCD]

C 0

39
38
71H

because 9+8 = 17
for 15 it is F
16 it is 0, 17 it is 1.

0 AC
C

00 11 1001
0011 1000
0 1 11 0001
7 1

Achiever

**Step 1:** AL = 71 + 06 = 77H

AH = AH + 01 = AH = 01

C = AC = 1

**Step 2:** clear ms 4 bits of AL register

AL = AL and 0FH

77H and 0FH

07H

| | AH | AL |
|---|---|---|
| AX | 01 | 07H |

| AC | C |
|---|---|
| 1 | 1 |

Flag:

## Shift and Rotate

* They are used for manipulating binary numbers at the binary bit level.

## Shift

* shift instructions position or move numbers to the left or right within a register or memory location.

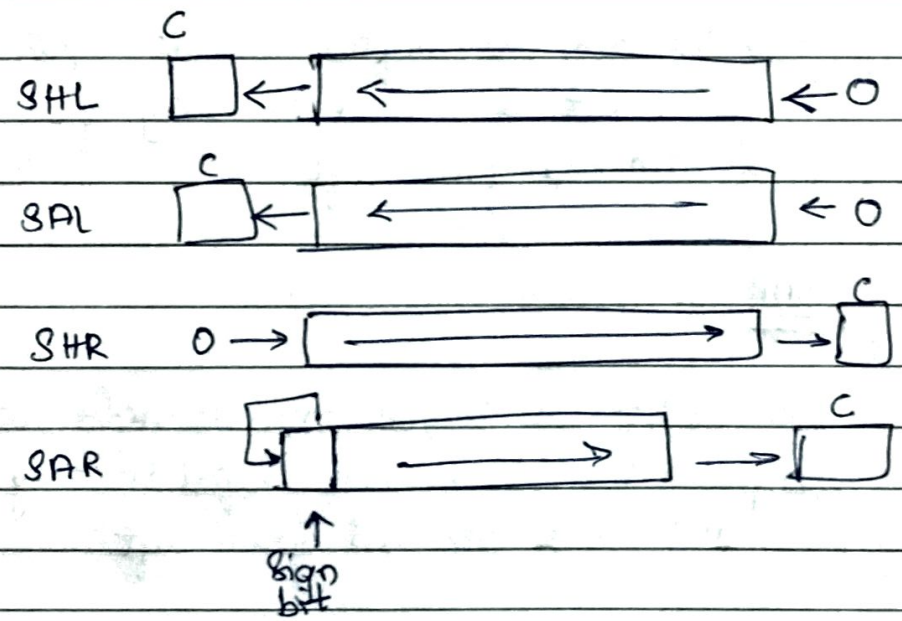
* They also perform simple arithmetic such as multiplication by powers of $2^{+n}$ (left shift) and division by powers of $2^{-n}$ (right shift)

Four difference shift instructions are there

1. Two logical shifts
2. Two arithmetic shifts

* logical shift moves a 0 into rightmost bit position for a logical left shift and a 0 into leftmost bit position for a logical right shift.

* Arithmetic right shift and logical right shift are different because the arithmetic right shift copies the sign-bit through the number whereas logical right shift copies a 0 through the number.




Eg: SHL AX, 1 — AX is logically shifted left 1 place
SHR BX, 12 — BX is logically shifted right 12 places

* logical shift operates with <u>unsigned numbers</u>

* Arithmetic shift → signed numbers.

* CL must hold shift count.

Eg: SHL DX, 14

(or)

MOV CL, 14
SHL DX, CL

SHR

when the bits are shifted towards right, least significant bit is shifted to carry flag and 0 is attached from left.

SAR

when the bits are shifted towards right, least significant bit (LS bit) is shifted to carry flag and MS bit is attached from left.

Double precision shifts

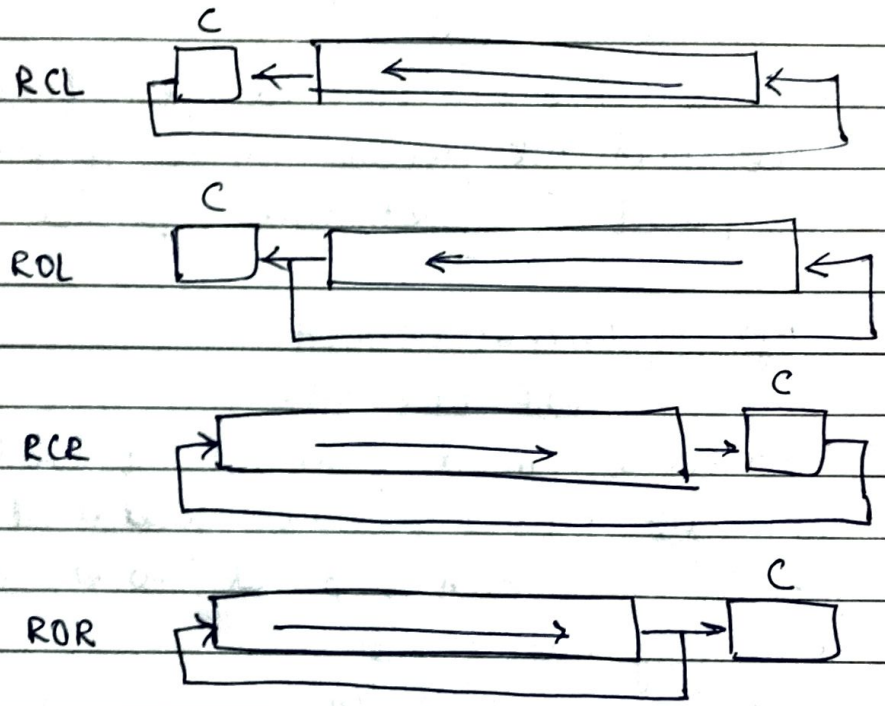
SHRD AX, BX, 12

* It logically shifts AX right by 12 bit positions. Rightmost 12 bits of BX shift into leftmost 12 bits of AX. Contents of BX remain unchanged by this instruction.

## Rotate

Rotate instruction position binary data by rotating information in a register or memory location either from end end to another or through carry flag.

Four available rotate instructions are




* Numbers rotate through a register or memory location, through C flag (carry), or through a register or memory location only.

* A rotate count can be immediate or located in register CL.

Eg: ROL SI, 14 — SI rotates left 14 places
RCL BL, 6 — BL rotates left through carry 6 places.

ACHIEVER

## Bit Scan Instructions

① BSF (Bit Scan Forward)
② BSR (Bit Scan Reverse)

* BSF instruction scans the number from leftmost bit toward the right
* BSR scans number from rightmost bit toward the left.
* If a 1-bit is encountered, zero flag is set and bit position number of 1-bit is placed into destination operand.

Eg: If EAX = 60000000H

BSF EBX, EAX

Number is scanned from leftmost bit toward the right. First 1-bit encountered is at bit position 30, which is placed into EBX and zero flag bit is set.

If same value for EAX is used for BSR instruction, EBX register is loaded with 29 and zero flag bit is set.

## String Comparisons

① SCAS (String Scan Instruction)

It compares the AL register with a byte block of memory, the AX register with a word block of memory or EAX register with a double word block of memory.

PAGE :
DATE : / /

* SCAS instruction subtracts memory from AL, AX or EAX without affecting either the register or memory location.

* opcode for byte comparison → SCASB
  word → SCASW
  doubleword → SCASD

* SCAS instruction use direction flag (D) to select either auto increment or auto decrement operation for DI.

```
MOV DI, OFFSET BLOCK
CLD
MOV CX, 100
XOR AL, AL
REPNE SCASB
```

↓

(Repeat while not Equal)

<u>CMPS</u> (Compare strings Instruction)

* It compares two sections of memory data as bytes (CMPSB), words (CMPSW) or doublewords (CMPSD).

```
MOV SI, OFFSET LINE
MOV DI, OFFSET TABLE
CLD          → auto-increment
MOV CX, 10   → load counter
REPE CMPSB.
```

## AAM Instruction

* AAM follows multiplication instruction after multiplying two one-digit unpacked BCD numbers.

### Working

① Divide AX by 10
remainder → AL
Quotient → AH.

```
MOV AL,09
MOV BL,07
MUL BL
AAM
```

Expected answer: 09 * 07 = 63 (Packed BCD)
0603 (Unpacked BCD)

09 * 07 = 003FH (AX)

## AAD

* AAD instruction appears before a division.
* It requires AX register contains two digit unpacked BCD number before executing.
* After adjusting AX register with AAD, it is divided by an unpacked number to generate a single digit result in AL with any remainder in AH.

$\boxed{ah * 10 + al \rightarrow al}$

PAGE :
DATE : / /

## AAS (ASCII adjust after subtraction)

* After subtracting an ASCII or unpacked decimal number (BCD) from another ASCII or unpacked decimal number, the result will be in binary.

* If AAS is used, result will be converted into unpacked decimal number (BCD).

* Result is available in AL register.

Eg: 09 − 06 = 03
BCD BCD BCD

Working:

① If lower 4 bits of AL > 9 or AF = 1 then
(AL) ← (AL) − 06
(AH) ← (AH) − 01
Set C and AC flag to 1.

② Clear MS 4 bits of AL register
(AL) ← (AL) and 0FH.

Eg: 39 is subtracted from 36.

Expected result = 36 − 39 = −03.

39H = 0 0 1 1 1 0 0 1
1's Complement = 1 1 0 0 0 1 1 0
1
2's Complement = 1 1 0 0 0 1 1 1
C 7

ACHIEVER

$$\begin{array}{r} 36\ H \\ -\ 39\ H \\ \hline FD\ H \end{array}$$

C = 0 AC = 0

$$\begin{array}{r} 0\ 0\ 1\ 1 \quad 0\ 1\ 1\ 0 \\ +\ 1\ 1\ 0\ 0 \quad 0\ 1\ 1\ 1 \\ \hline 1\ 1\ 1\ 1 \quad 1\ 1\ 0\ 1 \end{array}$$

① AL = AL − 06

AL = FDH − 06 = F7

AH = AH − 01

C = AC = 1

② Clear MS 4 bits of AL

AL = AL and 0FH

| | AH | AL |
|---|---|---|
| AX | FF | 07 |

| AC | C |
|---|---|
| 1 | 1 |

Flag.

## DAA

* After adding two binary numbers the result should be stored in AL register.

* After addition, if instruction DAA is used, the result will be converted into BCD and result will be available in AL register.

PAGE :
DATE : / /

Working

① If lower 4 bits of AL > 9 or AF = 1

AL = AL + 06

AF = 01

② Upper 4 bits of AL > 9 or C = 1

AL = AL + 60H

CY = 1

Example: 84 and 96

Expected result : 84 + 96 = 180

```
C=1  84H          C=1  1000      AC=0  0100
     96H        +      1001            0110
     ----             ----            ----
     1AH              0001            1010
                      ----            ----
                        1               A
```

AL = 01A + 06H = 20H and AF = 1

AL = 20H + 60H = 80H and C = 1

Result is AL = 80H and AF = 1 and C = 1

| C | AL |
|---|---|
| 1 | 80 |

DAS (Decimal After Subtraction)

Working

① If lower 4 bits of AL > 9 or AF = 1

AL = AL − 06

Set AF = 1

ACHIEVER

② Upper 4 bits of AL > 9 or Carry is 1
AL = AL - 60H
Set C flag to 1.

Eg: SUB AX, BX
DAS

## Program Control Instructions

### JUMP

* It is a program Control Instruction
* It allows programmer to skip sections of a program and branch to any part of the memory for next instruction.

* Conditional jump instruction allows programmer to make decisions based upon numerical tests.

### Unconditional Jump (JMP)

* Three kinds of unconditional jump are present
  ① Short jump
  ② Near jump
  ③ Far jump

* Short jump is 2-byte instructions that allow jumps or branches to memory locations within +127 and -128 bytes from the address following the jump.

* Near jump is 3-byte which allows a branch or jump within ± 32k bytes from instruction in the current code segment.

* Far jump is 5-byte which allows a jump to any memory location within real memory system.

* Short and near jumps → Intrasegment jumps
Far jump → Intersegment jumps.

## Short jump

* Short jumps are called relative jumps because they can be moved along with their related software to any location in current code segment without a change.

* Here jump address is not stored with opcode, instead a distance or displacement follows opcode.

Short jump

| opcode | |
|---|---|
| EB | Disp |

Near jump

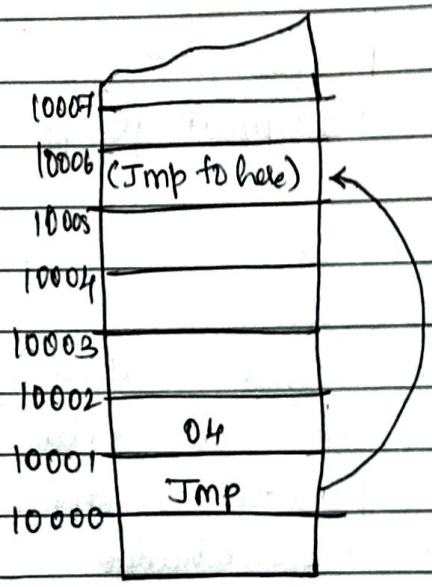
| opcode | | |
|---|---|---|
| E9 | Disp low | Disp high |

Far jump

| opcode | | | | |
|---|---|---|---|---|
| EA | IP low | IP high | CS low | CS high |

* When microprocessor executes a short jump, displacement is signextended and added to instruction pointer (IP/EIP) to generate jump address within current code segment.

Short jump to four memory location




CS = 1000H
IP = 0002H
New IP = IP + 4
New IP = 0006H

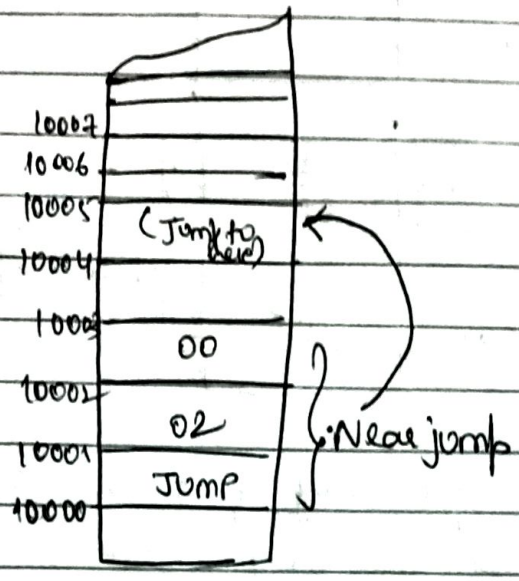
JMP Next instruction jumps to label Next for next instruction.

Near Jump

* Near jump passes control to an instruction in current code segment located within ±32K bytes from near jump instruction.

* Signed displacement adds to instruction pointer to generate jump address.

* Near jump is relocatable because it is also a relative jump




CS = 1000H
IP = 0002H
New IP = 0005H

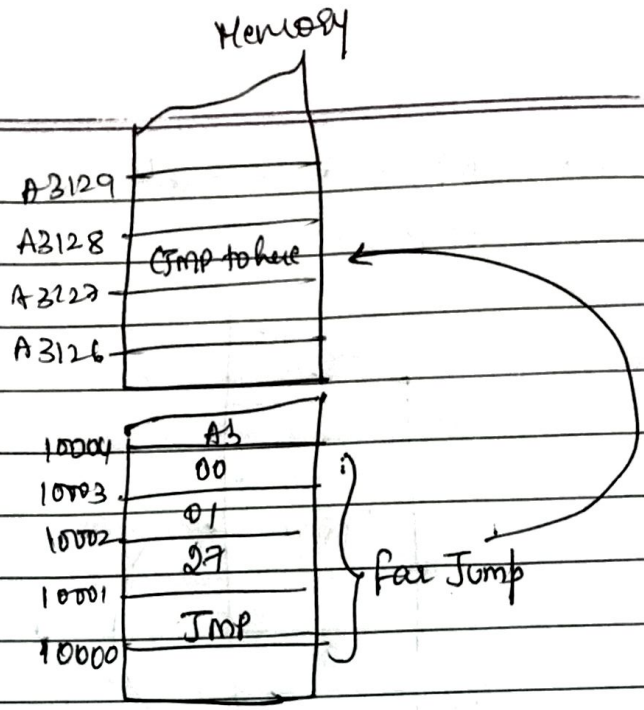
```
        XOR BX, BX
START:  MOV AX, 1
        ADD AX, BX
        JMP NEXT

        <skipped memory locations>

NEXT:   MOV BX, AX
        JMP START
```

## Far Jump

* Far jump obtains a new segment and offset address to accomplish jump.
* Bytes 2 and 3 of the 5-byte instruction contain the new offset address, bytes 4 and 5 contain new segment address.




* label is far only if it is external to current code segment or procedure.

```
EXTRN UP: FAR      → more than one pgm file.
        XOR BX, BX
START:  ADD AX, 1
        JMP NEXT

<skipped memory locations>

NEXT:   MOV BX, AX
        JMP FAR PTR START
        JMP UP
```

## Jump with Register Operands

* Jump instruction can also use a 16 or 32-bit register as an operand.
* This sets up instruction as indirect jump.

* Address of jump is in register specified by jump instruction.

* Indirect jump does not add IP as with short and near jumps.

JMP AX

It copies contents of AX register into IP when jump occurs. This allows a jump to any location within current code segment.

## Indirect Jumps using an Index

* Jump instruction may also use [ ] form of addressing to directly access jump table.

* Jump table can contain offset addresses for near indirect jump or segment & offset address for far indirect jumps.

This type of jump is known as double-indirect jump if register jump is called an indirect jump.

## Conditional Jumps and Conditional Sets

* Conditional jumps are always short jumps in 8086 through 80286 µp.
* So the range of jump is limited to within +127 bytes and -128 bytes from the location following conditional jump.

* In 80386 and above conditional jumps are either short or near (±32K)

* In 64-bit mode of Pentium 4, near jump distance is ±2G for conditional jumps

* Conditional jump test the following flag bits : Sign (S) Parity (P)
  Zero (Z) Overflow (O)
  Carry (C)

If condition is true, a branch to label associated with jump instruction occurs.
If condition is false, next sequential step in the pgm executes.

Eg: JC will jump if carry bit is set

| Assembly language | Tested Condition | Operation |
|---|---|---|
| JA | Z=0 and C=0 | Jump if above |
| JCXZ | CX=0 | Jump if CX=0 |

* The term greater than and less than refer to signed numbers.
* The term above and below refer to unsigned numbers.

```
MOV CX,100      ; load Counter
MOV AL,0AH      ; load AL with 0AH
CLD             ; auto increment
REPNE SCASB     ; search for
STC             0AH
```

set carry if found.

PAGE :
DATE : / /

## Conditional set Instructions

Conditions tested by conditional jumps are put to work with Conditional set instructions

* They set a byte to either 01H or clear a byte to 00H depending on the outcome of Condition under test.

| Assembly language | Tested Condition | Operation |
|---|---|---|
| SETA | Z=0, C=0 | Set if Above |
| SETS | S=1 | Set if Sign (negative) |
| SETNO | O=0 | Set if no overflow |

## LOOP

* loop instruction is a combination of a decrement CX and JNZ conditional jump.

* loop decrements CX
  If CX != 0, it jumps to the address indicated by label.
  If CX = 0, next sequential instruction executes.

## Conditional ~~Loops~~ Loops

* loop instruction also has conditional forms
  ① LOOPE ② LOOPNE

* LOOPE (Loop while Equal) instruction jumps if CX != 0 while an equal condition exists.

* LOOPNE (loop while Not Equal) instruction jumps if CX != 0 while a not equal Condition exists.
  It will exit the loop if condition is equal or if CX register decrements to 0.

## Controlling the flow of the program

* Statements like .IF, .ELSE, .ELSEIF and .ENDIF are used to control the flow of the program.

Eg:
```
.IF AL >= 'A' && AL <= 'F'
        SUB AL, 7
.ENDIF
SUB AL, 30H
```

Relational operators used with .IF statement in assembly language.

| Operator | Function |
|---|---|
| == | Equal or the same as |
| != | Not Equal |
| > | Greater than |
| >= | Greater than or Equal. |

Example that uses conditional .IF statement

A DOS program that reads key and stores the hexadecimal value in memory location TEMP

```
. Model small
. Data
Temp db ?
. Code
. Startup
        mov ah, 1
        int 21h
        .IF AL >='a' && AL <='f'      ; if lowercase
            SUB AL, 57H
    . ELSEIF . IF AL >='A' && AL <= 'F'
            SUB AL, 37H               ; if uppercase
    . ELSE
            SUB AL, 30H               ; if otherwise
    . ENDIF
        mov Temp, al                  ---> Save in Temp
. EXIT
End
```

## while loops

* The .WHILE statement is used to ~~see~~ with a condition to begin the loop, and

• ENDW statement ends the loop.

DOS program that reads a character string from keyboard and then displays it again

```
.model small
.data
    mes db 13,10          ; return & line feed
    Buf db 256 dup(?)     ; character string buffer
.Code
.Startup
    mov ax, dx
    mov es, ax
    cld                   ; select auto-increment
    mov di, offset buf    ; address buffer
.while AL != 0DH          ; loop while not enter
    jmp @C0001
@C0002:
        mov ah, 1         ; read key
        int 21h
        STOSB             ; store key code
.ENDW
@C0001:
        cmp al 0dh
        jne @C0002
        mov byte ptr [DI-1] '$'
        mov dx, offset mes
        mov ah, 9
        int 21h           ; display mes.
.EXIT
end
```

PAGE :
DATE : / /

## Repeat - Until loops

* A series of instruction is repeated until some condition occurs.
  - REPEAT statement defines start of the loop, the end is defined with .UNTIL statement which contains a condition.

## Procedures

* Procedure (subroutine, method or function) is a group of instructions that usually perform one task.
* Procedure is a reusable section of software that is stored in memory once but used as often as necessary.
* This saves memory space and makes it easier to develop software
* CALL instruction links to the procedure, RET (Return) instruction returns from the procedure.

* Stack stores return address whenever a procedure is called during execution of program.
* Call pushes the address of the instruction following CALL (return address) on the stack.
* A procedure begins with PROC directive and ends with ENDP directive.
* Each directive appears with name of the procedure.

Types of procedure

1. Near
2. Far

Eg:

```
SUMS PROC NEAR
        ADD AX, BX
        ADD AX, CX
        ADD AX, DX
        RET
SUMS ENDP

SUMS1 PROC FAR
        Add ax, bx
        add ax, cx
        add ax, dx
        ret
Sums1 Endp

Sum S3 proc near Use bx cx dx
        add ax, bx
        add ax, cx
        add ax, dx
        ret
Sums Endp
```

* Near return removes a 16-bit number from stack and places it into instruction pointer from procedure in current code segment.

* A far return removes a 32-bit number from stack and places it into both IP and CS to return from the procedure to any memory location.

PAGE :
DATE : / /

* Procedures that are to be used by all software (global) should be written as far procedure.
* Procedures that are used by a given task (local) are normally defined as a near procedure.

## CALL

* Call instruction transfers the flow of the program to the procedure.
* Call differs from jump instruction because a CALL saves a return address on stack.

## Near Call

* Near Call instruction is 3 bytes long.
  First byte contains opcode
  Second and third byte contains displacement or a distance of ±32K.

* When Near Call executes, it first pushes offset address of next instruction onto the stack.

  Offset address of next instruction appears in instruction pointer (IP or EIP).

  After saving this return address, it then adds the displacement from bytes 2 and 3 to the IP to transfer control to the procedure.

## Far Call

* Far Call is 5-byte instruction that conta an opcode followed by next value for IP and CS registers.

* Far Call places the contents of both IP an CS on the stack before jumping to addres indicated by bytes 2 through 5 of the instruction.

## Calls with Register operands

* Call instructions also contain a register op

Eg: CALL BX

Pushes the contents of IP onto stack. It then jumps to offset address located in register BX in current code segment.

Eg:
```
.model tiny
.Code
.Startup
mov bx, offset Disp   ; load BX with offset Disp
mov dl, '0'           ; display 0
Call bx
mov dl, 'K'           ; display K
Call bx
.Exit

Disp Proc Near
     mov ah, 2
     int 21h
```

PAGE :
DATE : / /

```
        ret
disp Endp
end.
```

## Calls with Indirect Memory Addresses

* Used when different subroutines need to be chosen in a program.
* This selection process is often keyed with a number that addresses a CALL address in a lookup table.

## RET

* RET removes a 16-bit number (near return) from the stack and places it into IP or removes a 32-bit number (far return) and places it into IP and CS.

* Both near and far return instructions are defined in procedure's PROC directive which automatically selects proper return instruction.

* One form of return instruction adds a number to the contents of stack pointer (SP) after the return address is removed from stack.

Eg.

```
MOV AX,30
MOV bx,40
push ax
push bx
```

ACHIEVER

```
Call Addm
Addm Proc near
        push bp
        mov bp, sp
        mov ax, [bp+4]
        add ax, [bp+6]
        pop bx
        ret 4
Addm Endp.
```

## Introduction to Interrupts

* An interrupt is either a hardware generated CALL (externally derived from a hardware signal) or a software generated CALL (internally derived from execution of an instruction or by some other internal event.
* Sometimes an internal interrupt is called an exception.

* Into Program is interrupted by calling an interrupt service procedure (ISP) or interrupt handler.

## Interrupt Vectors

* An interrupt vector is a 4-byte number stored in first 1024 bytes of the memory when µP operates in real mode.

PAGE :
DATE : / /

* In protected mode, vector table is replaced by an interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts.

* There are 256 different interrupt vectors and each vector contains the address of an interrupt service provider.

* Each vector contains a value of IP and CS that forms the address of the interrupt service procedure. First 2 bytes contain IP, and last 2 bytes contain CS.

## Interrupt vectors defined by Intel

| Number | Address | Function |
|---|---|---|
| 0 | 0H - 3H | Divide error |
| 1 | 4H - 7H | Single-step |
| 2 | 8-BH | NMI pin |

* Intel reserves first 32 interrupt vectors for the present and future microprocessor products.
* Remaining vectors (32-255) are available for users.

## Interrupt instructions

Three different interrupt instructions are
1. INT
2. INTO
3. INT3

## INTs

* There are 256 software interrupt instructions (INTs)
* Each INT instruction has a numeric operand whose range is 0 to 255.

  Eg: INT 100

  It uses interrupt vector 100

* Address of interrupt vector is determined by multiplying interrupt type number by 4.

  Eg: INT 10H, calls interrupt service procedure whose address is stored beginning at memory location 40H (10H X 4) in real mode.

* In protected mode, interrupt descriptor is located by multiplying type number by 8 because each descriptor is 8 bytes long.

* Each INT instruction is 2 bytes long. First byte contains opcode, second byte contains vector type number.

Whenever a software interrupt instruction executes, it

1. Pushes flags onto the stack
2. Clears T and I flag bits
3. Pushes CS onto the stack
4. Fetches new value for CS from interrupt vector
5. Pushes IP/EIP onto the stack
6. Fetches new IP/EIP from vector
7. Jumps to new location addressed by CS and IP/EIP.

PAGE :
DATE : / /

## IRET / IRETD

* Interrupt return instruction (IRET) is used only with software or hardware interrupt service procedures.

* IRET instruction will
  1. Pop stack data back into IP
  2. Pop stack data back into CS
  3. Pop stack data back into flag register.

## INT3

* It is a special software interrupt designed to function as a breakpoint.
* INT 3 is a 1-byte instruction while others are 2-byte instructions.

* INT3 is inserted in software to interrupt or break the flow of the software. This function is called a breakpoint.

## INTO

* Interrupt on Overflow (INTO) is a conditional software interrupt that tests the overflow flag(O).
  If O = 0 → INTO instruction performs no operation
  O = 1 → INTO instruction executes.
  Eg: It occurs in software that adds or subtracts signed binary numbers.

## Interrupt Control

Two instructions that are used to control INTR pin.

① STI → Set Interrupt flag

↓

It places a 1 into I flag bit which enables INTR pin

② CLI (Clear Interrupt Flag)

↓

It places a 0 into I flag bit, which disables INTR pin.

## Machine Control and Miscellaneous Instructions

### ① Controlling Carry Flag Bit

* The carry flag (c) propogates the carry or borrow in multiple-word / doubleword addition or Subtraction.

* Three instructions control contents of carry flag
  ① STC (Set Carry)
  ② CLC (Clear carry)
  ③ CMC (Complement Carry)

If C=0 → no error occured.
C=1 → error has occurred.

control

it which

which disables

Instructions

Carry or addition and

carry flag

PAGE :
DATE : / /

## WAIT

* WAIT instruction monitors the hardware $\overline{BUSY}$ pin and $\overline{TEST}$ pin 8086/8088.

* If WAIT executes while $\overline{BUSY}$ pin = 1, nothing happens and the next instruction executes.
* If $\overline{BUSY}$ pin = 0 when WAIT executes, µp waits for $\overline{BUSY}$ pin to return to a logic 1.

## HLT

* The halt instruction (HLT) stops execution of software
* There are 3 ways to exit a halt
  1. By an interrupt
  2. By a hardware reset
  3. During a DMA operation.

## NOP

* when µp enters a no operation instruction (NOP) it takes a short time to execute.

## LOCK Prefix

* The LOCK prefix appends an instruction and causes the $\overline{LOCK}$ pin to become a logic 0.
* $\overline{LOCK}$ pin often disables external bus masters or other system components.

  Eg: LOCK: MOV AL, [SI]

ACHIEVER

## ESC

* The escape (ESC) instruction passes instructions to the floating-point coprocessor from the microprocessor.

## BOUND

* It is a Comparison instruction that may cause an interrupt (vector type number 5).
* This instruction compares the contents of any 16-bit or 32-bit register against the contents of two words or doublewords of memory an upper and a lower boundary.

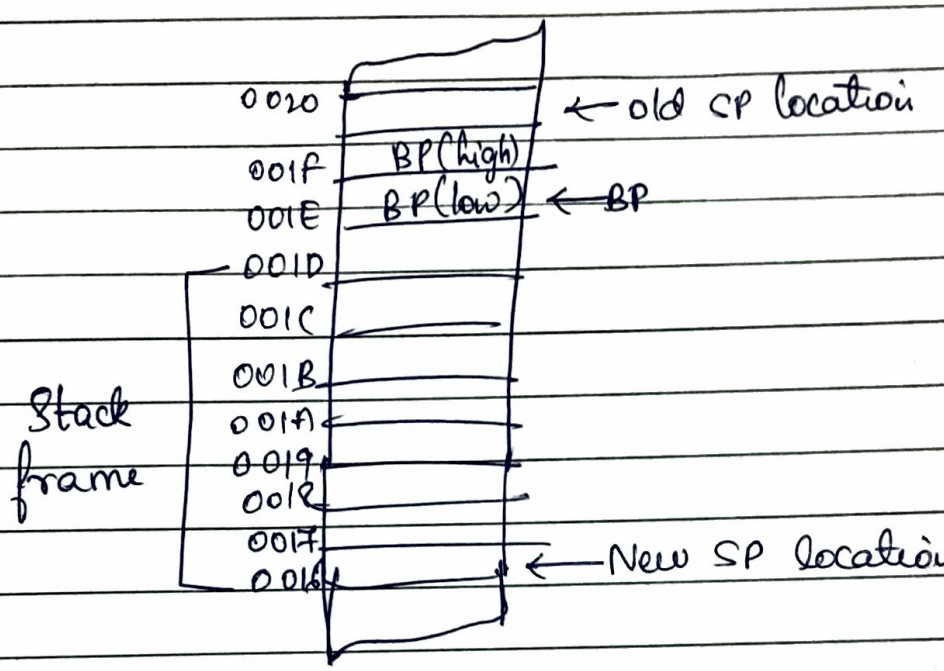
## ENTER and LEAVE

* They are used with stack frames, which are mechanisms used to pass parameters to a procedure through the stack memory.

* Enter instruction creates a stack frame by pushing BP onto the stack and then loading BP with uppermost address of stack frame.
* Enter instruction contains two operands
  1. 1st operand specifies number of bytes to reserve for variables on stack frame
  2. 2nd operand specifies the level of the procedure.

Eg: ENTER 8, 0

It reserves 8 bytes of memory for stack frame. 0 indicates level 0.

PAGE :
DATE : / /

## Stack frame created by ENTER 8,0 instruction