KTU STUDY MATERIALS

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

DOWNLOAD KTU ASSIST APP FROM PLAYSTORE

Addressing Modes of 8086

Addressing Mode
The method of specifying the data to be operated by the instruction.

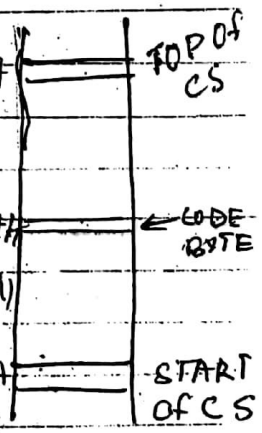
→ 12 A.M's and they can be classified into 5 groups.

Group 1
Addressing Modes for register & Immediate data.

1) Register Addressing
2) Immediate Addressing.

Group 2
Addressing Modes for memory data

3) Direct Addressing.
4) Register Indirect Addressing.
5) Based Addressing.
6) Indexed Addressing.
7) Based Index Addressing
8) String Addressing



Group 3
Addressing modes for I/O ports

9) Direct I/O port Addressing.
10) Indirect I/O port Addressing.

Group 4

11) Relative Addressing mode
12) Relative Addressing Mode

CS 348A0
4214
38A54

Group 5

① Register A.M

The instr. will specify "the name of the register" which holds the data to be operated by the instruction.

eg: MOV CL, DH

(CL) ← (DH)

MOV BX, DX

(BX) ← (DX)

② Immediate A.M

an 8-bit or 16-bit data is specified as part of the instruction.

eg: MOV DL, 08H (DL) ← 08_H

MOV AX, 0A9FH (AX) ← $0A9F_H$

③ Direct Addressing

an unsigned 16 bit displacement or signed 8 bit displacement will be specified in the instruction.

Displacement → Effective Address

8 bit displacement

EA is obtained by sign extending the 8-bit displacement to 16-bit.

Memory Address (20 bit physical Addr of m/y) = BA + EA

eg: MOV DX, [08H]

EA = 0008H (sign extended 8-bit displacement)

BA = (DS) × 16_{10} ; MA = BA + EA

× 10H (DX) ← (MA)

eg: MOV AX, [089DH]

EA = 089D$_H$; BA = (DS) × 16$_{10}$

MA = BA + EA

(AX) ← (MA)

4) Register Indirect Addressing

The instruction will specify "the name of the register, which holds the Effective Address (EA).

↓

BX, SI, DI

eg: MOV CX, [BX]

EA = (BX)

BA = (DS) × 16$_{10}$

MA = BA + EA

(CX) ← (MA)

MOV AX, [SI]

EA = (SI)

BA = (DS) × 16$_{10}$

MA = BA + EA

(AX) ← (MA)

5) Based A.M

In this A.M, BX or BP register is used to hold a base value of E.A and a signed 8 bit or unsigned 16 bit displacement, will be specified in the instruction.

eg: MOV AX, [BX + 08H]

0008H ← sign extend 08H ; EA = (BX) + 0008H.

BA = (DS) × 16$_{10}$

MA = BA + EA

(AX) ← (MA)

Segment
BX → DS
BP → SS

6) Indexed Addressing Mode

In this A.M, SI or DI register is used to hold an index value for any data and a signed 8-bit displacement or unsigned 16-bit displacement will be specified in the instruction.

eg: MOV CX, [SI + 08H]

0008H ← sign extend 08H ; EA = (SI) + 0008H

BA = (DS) × 16_{10} ; MA = BA + EA

(CX) ← (MA)

7) Based Indexed Addressing Mode

In this A.M, the E.A is given by sum of base index (BX or BP), index value (SI or DI) and an 8-bit or 16-bit displacement in the instruction.

eg: MOV DX, [BX + SI + 0AH]

000AH ← sign extend 0AH ; EA = (BX) + (SI) + 000A$_H$

BA = (DS) × 16_{10} ; MA = BA + EA

(DX) ← (MA)

8) String A.M (operate on string data)

In this A.M, the E.A of Src data stored in SI reg. & the E.A of Dest. data → DI reg.

⇒ Seg. reg. used for calculating base address

for Src data → DS

for dest. data → ES.

eg: MOVS BYTE REP MOVSB

Source; EA = (SI) ; BA = (DS) × 16_{10} ; MA = BA + EA

Dest; EA_E = (DI) ; BA_E = (ES) × 16_{10} ; MA_E = BA_E + EA_E

(MA$_E$) ← (MA)

if DF = 1, (SI) ← (SI) − 1 & (DI) ← (DI) − 1 | if DF = 0, (SI) ← (SI) + 1 & (DI) ← (DI) + 1

⑨ Direct I/O Port Addressing Mode

(used to access data from I/O devices or port)

.. an 8-bit port Address is directly specified inthe instruction.

eg: IN AL, [09H]

(PORT) addr = 09H

(AL) ← (PORT)

⑩ INDIRECT I/O Port A.M

(used to access data from I/O devices or port)

The instru. will specify "the name of the reg which holds the port address".

In 8086, 16 bit port address is stored in DX reg.

eg: OUT [DX], AX

Port addr = (DX) ; (PORT) ← (AX)

The content of AX is moved to port whose address is specified by DX registes.

⑪ Relative A.M

The E.A of a pgm instru. is specified relative to IP by an 8-bit signed displacement

eg: JZ 0AH

0000A H ← sign extend 0AH

If ZF=1, then (IP) ← (IP) + 0000AH

EA = (IP) + 0000AH

BA = (CS) * 16 10

MA = BA + EA ; then the pgm ctrl jumps to new address as calculated.

BA
CS IP
ES DI
DS SI

If ZF=0 ; then next instr of the pgm is executed.

2 copy

(12) Implied Addressing

In this A.M, the instruction itself will specify the data to be operated by the instruction.

eg: CLC Clear carry
; CF ← 0

8086	8088
* Access m/y in bytes and words * It has 16 bit data lines ($AD_0 - AD_{15}$) & 20 bit address lines * $\overline{BHE}$ signal to access higher byte * It has 6 byte instruction queue * Pin no. 34 is $\overline{BHE}/S7$. During T_2, T_3, T_4 it carries S_7. In max mode 8087 monitors this pin to identify the CPU as 8086 or 8088 & accordingly sets the queue length to 4 or 6 bytes. * M/$\overline{IO}$ for memory or I/O access	* Access m/y in bytes only. * 8 bit data lines & 20 bit address lines * data bus is 8 bit wide, it doesn't have $\overline{BHE}$ signal * 4 byte instr queue Dve to 8 bit databus instruction fetching is slow and 4 bytes are sufficient for queue. * Pin no. 34 is $\overline{SSO}$. It acts as S_0 in the minimum mode of 8088 µP. This signal is combined with IO/$\overline{M}$ and DT/$\overline{R}$ to decode the function of the current bus cycle. * IO/$\overline{M}$ for memory or I/O access.

Assembler Directives (~~Pseudo instructions~~)

→ A.D. are the instructions to the assembler regarding the pgm being assembled.

functions

- ✓ Used to specify start & end of a pgm
- ✓ attach value to variable
- ✓ allocate storage locations to i/p / o/p data
- ✓ to define ~~start and end of segments~~, procedures, macros etc.

① ASSUME

Indicates the name of each segment to the assembler.

eg: ASSUME CS: CODE, DS: DATA

↳ informs the assembler that the instrns of the programs are stored in the segment CODE & data are stored in the segment DATA.

② DB

Used to define byte type variable

~~input~~ general form — variable DB value / values

eg. AREA DB 45

③ ~~DD~~/DW

Used to define word type variable

variable DW value / values

eg: WEIGHT DW 1250

④ DD / DQ / DT

↳ used to define double word (32 bit), Quad word (64 bit), Ten bytes of a variable respectively.

⑤ SEGMENT AND ENDS

SEGMENT → Used to indicate the beginning of a code/data/stack segment.

ENDS → Used to indicate the end of code/data/stack segment

general form

```
segname SEGMENT
        .
        .
        .
segnam  ENDS
```

(Pgm code or Data defining statements)

⑥ ORG, EVEN AND EQU

ORG → used to assign the starting address (EA) for a pgm/data segment.

ORG 1000H

EVEN → Inform the assembler to store pgm/data in segment starting from even address

EQU → Used to attach a value to a variable.

eg: PORT1 EQU 08H

(The value of variable PORT1 is F8H.)

⑦ PROC, FAR, NEAR, ENDP

PROC → beginning of a procedure

FAR → intersegment call

NEAR → intrasegment call

ENDP → indicate end of a procedure

general form

```
procname  PROC [NEAR/FAR]
          .
          .
          .
          RET
procname  ENDP
```

(Pgm statements in procedure)

(8) MACRO AND ENDM

MACRO → indicate the beginning of a macro

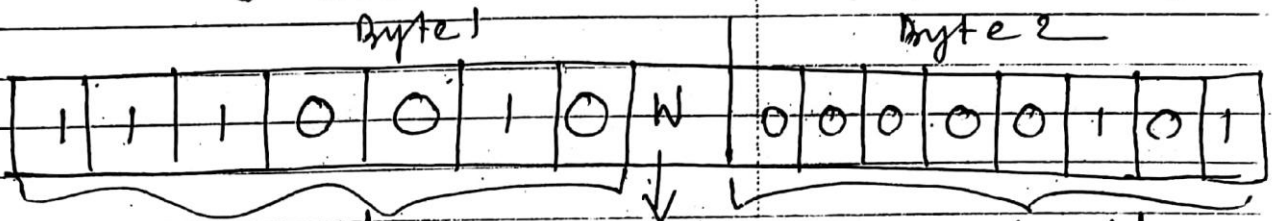
ENDM → end of a macro

general form

```
macroname MACRO [Arg1, Arg2, ...]
        .
        .
        .
        definitions
        declarations
        pgm statements
macroname ENDM
```

Coding template for 8086 IN

opcode ↓ 1110010

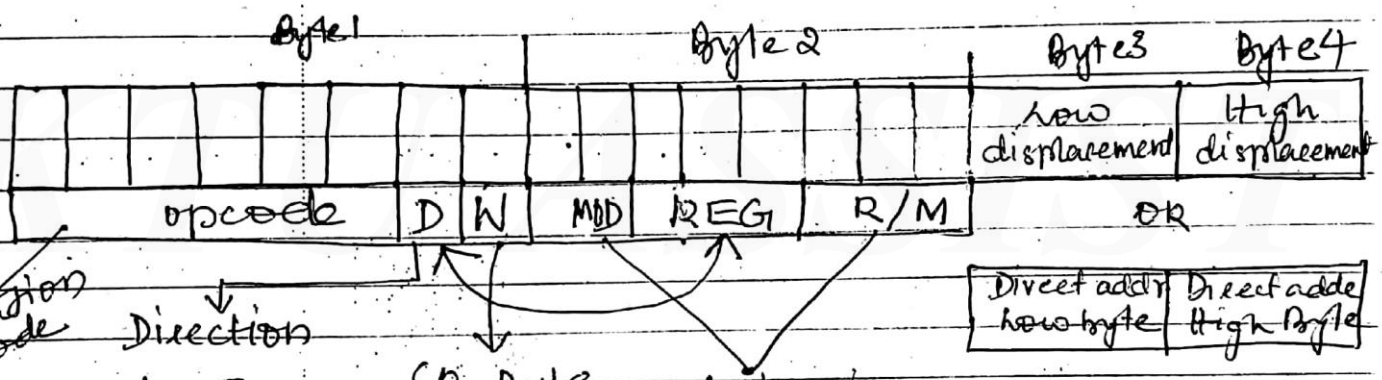


W: 0 - Byte, 1 - Word

tells whether it should input a byte to AL or word to AL

eg: IN AL, 05H

Coding template for 8086 MOV instruction



opcode

Direction: 0 - From, 1 - TO

0 - Byte, 1 - Word

Addressing Mode

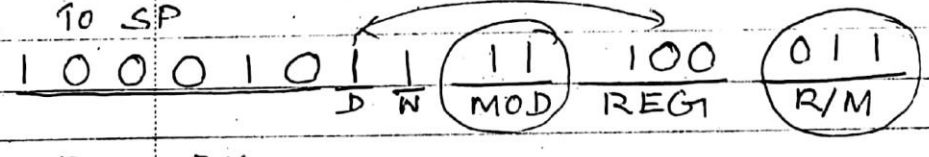
MOD → used to indicate whether the addr specification in the instr. contains displacement

R/M → " which reg contain E.A

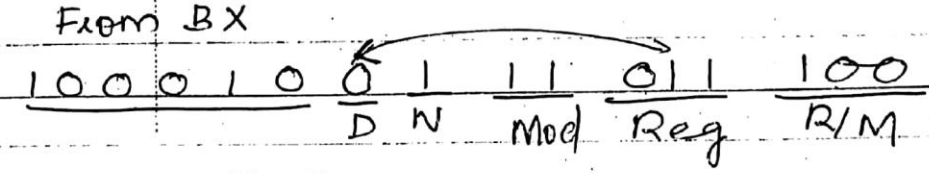
opcode → 100010

Case 1 MOV SP, BX — src & dest are regs.

(a) To SP

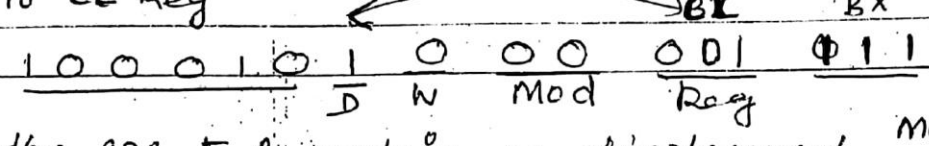


(b) From BX



Case 2 MOV CL, [BX]

To CL Reg



(If the spe. E.A contain no displacement MOD → 00

(3) MOV 43H + [SI], DH

From DH

						Byte 2	
100010	0	0	01	110	100	0100	001
opcode	D	W	Mod	Reg	R/M	4	3

(4) MOV CX, [437AH]

To CX

						Byte 3	Byte 4
100010	1	1	00	001	110	01111010	01000011
opcode	D	W	Mod	Reg	R/M	7A	43

Instruction Set

The 8086 instruction set can be classified into 6 groups.

(1) DATA TRANSFER INSTRUCTION

⇒ Includes instructions for moving data between registers, register and memory, register and stack m/y, accumulator and I/O device.

* General-purpose byte or word transfer Instructions
  MOV, PUSH, POP, & XCHG
* Simple input and o/p port transfer Instructions
  IN, OUT
* Special Address transfer Instructions
  LEA, LDS, LES
* Flag transfer Instructions
  LAHF, SAHF, PUSHF, POPF

(2) ARITHMETIC INSTRUCTION

⇒ Includes instructions for addition & subtraction of binary, BCD & Instructions for multiplication & division of signed & unsigned binary data.

* Addition Instructions
  ADD, ADC, INC, AAA, DAA
* Subtraction Instructions
  SUB, SBB, DEC, NEG, CMP, AAS, DAS
* MULTIPLICATION INSTRUCTIONS
  MUL, IMUL, AAM
* DIVISION INSTRUCTIONS
  DIV, IDIV, AAD

(3) BIT MANIPULATION INSTRUCTIONS

⇒ Includes Instructions for performing logical operations

* Logical Instructions

*Rotate Instructions

ROL, ROR, RCL, RCR

(4) STRING INSTRUCTION ⇒ Includes instructions for moving string data b/w two memory locations and comparing string data word by word or byte by byte.

REP (prefix), REPE (a), REPZ (a), REPNE (b), REPNZ (b)

MOVS (prefix)/MOVSB (a)/MOVSW (b)

COMPS, COMPSB, COMPSW

(5) Program Execution transfer Instructions / control transfer group

* Unconditional transfer instructions:

CALL, RET, JMP

* Conditional

JA, JAE, JB, JBE, JC, JE, JG, JGE, JL, JLE

JNC, JNZ, JNO, JNP, JNS, JO, JP, JS

* Location control Instructions

LOOP, LOOPZ, LOOPNZ, JCXZ

* Interrupt Instructions

INT, INTO, IRET

⇒ Includes instructions to call a procedure/subroutine in the main program and to jump from one part of a program to another part either conditionally (after checking flags) or unconditionally (without checking flags).

(6) Processor control Instruction

* Flag set/clear Instructions:

STC, CLC, CMC, STD, CLD, STI, CLI

* External hardware synchronization

HLT, WAIT, ESC, LOCK

* No Operation Instruction

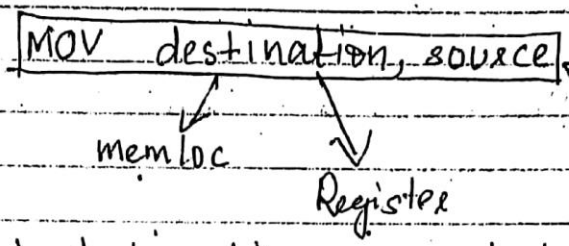
NOP

⇒ Includes instructions to set/clear the flags, to delay and halt the processor execution.

(1) DATA TRANSFER INSTRUCTION

a) General purpose byte or word transfer Instructions

(a.1) MOV destination, source



Source and destination cannot both be m/y loc in an instruction.

⇒ Copy byte or word from specified src to Specified dest

(a.2) PUSH (Copy specified word to ~~top of Stack~~)

PUSH [reg16 / mem / segreg]
(GPreg)
(Source)

(SP) ← (SP) − 2
MA_S = (SS) × 16_10 + (SP)
(MA_S ; MA_S+1) ← reg16 / mem / segreg.
word only

Note :- The stack pointer is decremented by 2 and the [content of (16-bit) reg/mem/segment register] is pushed to stack memory pointed by SP. (Stack segment)

Uses

* To save data on the stack, so that it will not be destroyed by a procedure
* Used to put data on the stack so that a procedure can access it there as needed.

PUSHF

Decrements the stack pointer by 2 and copies the word in the flag reg. to the m/y loc pointed to by the stack pointer.

POPF : Illegible

(copy word from TOS to specified location)

(A.3) POP Destination

Destination → General Purpose Register, mem, Seg reg.

Syntax → POP reg16/mem/segreg	$MA_S \leftarrow SS \times 16_{10}{}^{+} (SP)$
	reg16/mem/segreg $\leftarrow (MA_S ; MA_S{}^{+1})$
Note :- The content of (16 bit) Stack memory pointed by SP is moved to segment register and stack pointer is incremented by 2.	$SP \leftarrow SP+2$

:- POP CS is illegal

⇒ POPF means the content of (16-bit) Stack memory pointed by SP is moved to flag register and the stack pointer is incremented by 2.

eg: POP DX : Copy a word from TOS to DX & Increment SP by 2

POP DS : Copy a word from TOS to DS & Increment SP by 2.

(A.4) XCHG

XCHG Destination, Source

Destination → G.P.reg, mem; Source → G.P.reg, mem

Note
:- Used to exchange the contents of a register with the contents of another register OR the contents of a register with the contents of a memory location.

:- XCHG cannot directly exchange the contents of two memory locations.

:- Src and dest must both be words or both be bytes.

eg: XCHG AX, DX
XCHG BL, CH

(c) Special ... Instructions

(c.1) LEA (Load Effective Address)

LEA Register, Source

→ Determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16 bit register.

eg: LEA BP, SS:STACK_TOP ; Load BP with offset of STACK_TOP in SS

LEA CX, [BX][DI] ; Load CX with EA = (BX) + (DI)

(c.2) LDS (Load Register and DS with words from memory)

LDS Register, Memory Addr. of First Word

This Instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two m/y locations into DS register.

eg: LDS BX, [4326]

Copy contents of m/y at displacements 4326 in DS to BL, contents of 4327 to BH. Copy contents at displacement of 4328H and 4329H in DS to DS register.

(c.3) LES (Load Register and ES with words from m/y)

LES Register, Memory Addr. of First Word

The word from the first two memory location is copied into the specified Register and the word from the next two m/y locs is copied into the ES register.

eg: LES BX, [789AH]; Contents of m/y at displacements 789AH and 789BH in DS copied to BX. Then the contents of m/y at displacements 789CH & 789DH in DS copied to ES reg.

(d) Flag transfer Instructions

(d.1) LAHF

Used to copy low byte of flag register to AH

(d.2) SAHF

Used to copy AH Register to low Byte of flag Register.

(d.3) PUSHF

Decrements the SP by 2 and copies the word in the flag reg. to the m/y loc pointed by the stack pointer (stack m/y)

(d.4) POPF

The content of stack m/y pointed by SP is moved to flag reg. and the SP is incremented by 2.

(2) ARITHMETIC INSTRUCTION

a) Addition Instructions

ADD, ADC, INC, AAA, DAA

(a.1) ADD [Add Specified byte to byte or Specified word to word]

ADD Destination, Source

Destination: mem, reg. Source: mem, reg, number

Note :→ Src & dest cannot both be m/y locations.
→ Src & dest must both be byte locations or word locations.

(a.2) ADC : (Add with carry) --

ie: Add byte + byte + carry flag OR
word + word + carry flag

ADC Destination, Source

Destination: mem, reg
Source: mem, reg, an immediate number

eg: ADC CL, BL ; (CL) ← (CL) + (BL) + Carry

(a.3) INC [Increment specified byte or specified word by 1]

INC Destination

Destination: reg, memloc

eg: a) INC BL ; Add 1 to contents of BL register

b) INC CX ; Add 1 to contents of CX register.

c) INC BYTE PTR [BX] ; Increment byte in DS at offset contained in BX

d) INC WORD PTR[BX] ; Increment word in DS at offset contained in BX

(b) SUBTRACTION INSTRUCTION

(b.1) SUB

SUB Destination, Source

Destination: memloc, reg
Source: memloc, reg, an immediate number

Note ÷ This Instruction subtract the number in the indicated source from the number in the indicated destination & put the result in the indicated destination

ie. (destination) ← (dest) − (src)

÷ Carry flag will be set after a subtraction if the number in the specified source is larger than the number in the specified destination

ie. If (src) > (dest) then CF is set.

÷ Src. and dest. cannot both be m/y locations.

÷ src and dest must both be byte locs or word locs.

eg: SUB CX, BX ; (CX) ← (CX) − (BX)
SUB CH, AL ; (CH) ← (CH) − (AL)

(b.2) SBB (Subtract with Borrow)

SBB Destination, Source
(Destination: memloc, reg; Source: memloc, reg, an immediate number)

Subtract byte and carry flag from byte OR word and carry flag from word.

eg: SBB CH, AL ; (CH) ← (CH) − (AL) − (CY)
; Subtract contents of AL and contents of CF from contents of CH. Result in CH.

(b.3) DEC [Decrement specified byte or specified word by 1]

DEC Destination
(Destination: memloc, reg)

eg: DEC CL : Subtract 1 from contents of CL register.
DEC BP : Subtract 1 from contents of BP register.
DEC BYTE PTR [BX] ; Subtract 1 from byte at offset [BX] in DS
DEC WORD PTR [BP] ; Subtract 1 from a word at offset [BP] in SS.

(b.4) NEG

NEG Destination
(Destination: reg, memloc)

CMP Destination, Source

memloc → reg, reg → memloc, reg → an immediate number

Note :

⇒ Src and Dest cannot both be memory location

⇒ Comparison is done by subtracting the source byte or word from the destination byte or word.

eg: CMP CX BX

	CF	ZF	SF	
CX = BX	0	1	0	; Result of subtraction is 0.
CX > BX	0	0	0	; No borrow required so CF = 0
CX < BX	1	0	1	; Subtraction required borrow so CF = 1

eg: CMP AL, 01H

(c) MULTIPLICATION INSTRUCTION

(c-1) MUL

Multiply unsigned byte by byte or unsigned word by word.

MUL Source

memloc, reg

Note :

① This instruction multiplies an unsigned (8 bit) byte from some source times an unsigned (8 bit) byte in AL; then the result is put in AX (16 bit destination)

AH (MSB) AL (LSB)

eg: MUL BH ; AL times BH, result in AX

(AX) ← (AL) × (BH)

16 bit word, 8 bit byte, 8 bit byte

② Multiplies an unsigned word from some source times an unsigned word in AL; then the result is put in DX AX (32 bit destination)

(MSB) DX, AX (LSB)

eg: MUL CX ; (DX)(AX) ← (AX) × (CX)

double word, word, word

(c.2) IMUL

IMUL Source

(a) Source is a signed byte

(Signed byte in AL) x (signed byte in Source)
= Product a signed word

eg: IMUL BH ; (AX) ← (AL) x (BH) in AX.

(b) Source is a signed word

(signed word in AX) x (signed word in source)
= Product in signed double word

eg: IMUL CX ; (DX)(AX) ← (AX) x (CX)

(d) DIVISION INSTRUCTION

(d.1) DIV (unsigned Division)

DIV Source (divisor)
↙ mem loc ↘ reg.

Dividend
AX DXAX

(a) unsigned word in (AX) / unsigned byte (source) ; Quotient → AL, Remainder → AH

eg: DIV BL ; (AX) / (BL) ; Q = (AL), R = (AH)

(b) unsigned double word in (DX)(AX) / unsigned word (source) ; Quotient → AX, Remainder → DX

eg: DIV CX ; (DX)(AX) / (CX) ; Q → (AX), R → (DX)

(d.2) IDIV (signed Division)

IDIV Source

signed byte in Source ; remainder → AH

eg: IDIV BL ; (AX) ; Q → AL
(BL) ; R → AH

(b) signed double word in (DX)(AX)
signed word in Source ; Quotient → AX
Remainder → DX

eg: IDIV BP ; (DX)(AX) ; Q → (AX)
(BP) ; R → (DX)

(3) BIT MANIPULATION INSTRUCTION

(a) Logical Instruction

(a.1) NOT (Invert each bit of a byte or word)

NOT Destination
(memloc, reg)

eg: NOT BX ; complement contents of BX register.

NOT BYTE PTR [BX] ; complement m/y byte at offset [BX] in data segment.

(a.2) AND (AND Corresponding Bits of two Operands in source and destination of the result is put in the specified destination.)

AND Destination, Source
(reg, memloc) (reg, memloc, an immediate number)

eg: AND CX, [SI] ; AND word in DS at offset [SI] with word in CX and result in CX register.

AND BH, CL ; AND byte in CL with byte in BH & result in BH.

(a.3) OR { Logically OR corresponding bits of two operand in Src and destination and the result is put in destination }

OR destination, Source

memloc reg memloc reg an immediate number.

eg: OR AH, CL ; the content of CL register ORed with the content of AH, result in AH & CL not changed.

OR BP, SI ; the content pointed by SI in DS ORed with BP, result in BP.

(a.4) XOR (Exclusive OR corresponding Bits of two Operands)

0 0 → 0
0 1 → 1
1 0 → 1
1 1 → 0

XOR destination, Source

memloc reg memloc reg an immediate number.

eg: XOR CL, H ; Byte in BH XORed with byte in CL and result in CL.

XOR BX, CX ; (BX) ← (BX) ⊕ (CX)

(b) SHIFT INSTRUCTIONS

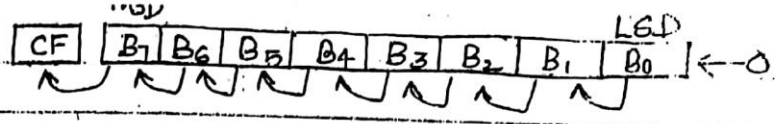
(b.i) SHL / SAL

SHL Destination, Count

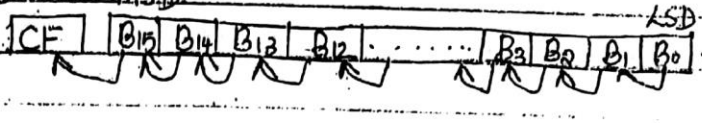
memloc Reg.

CF ← MSB ← ... ← LSB ← 0

Note :-



b) reg16/mem16



(b.2) SHR (Shift Operand Bits Right, Put Zero in MSB)

SHR Destination, Count

(mem or reg)

Note:-

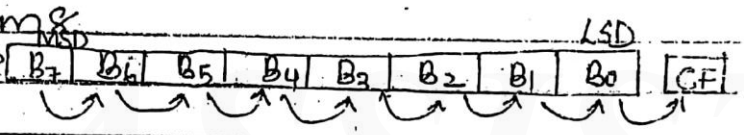
① eg: SHR BP, 1

② eg: MOV CL, 04H

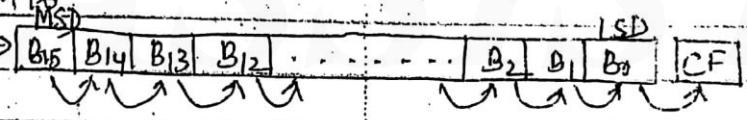
SHR AL, CL

0 → MSB ——————→ LSB → C

a) reg8/mem8



b) reg16/mem16



(b.3) SAR (Shift Operand Bits Right, New MSB = Old MSB)

MSB → MSB ——————→ LSB → CF

SAR Destination, Count

(mem or reg)

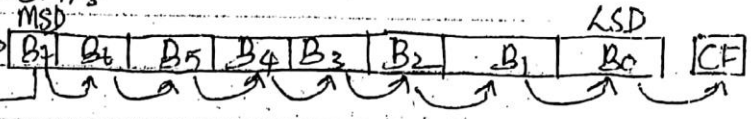
Note:-

① eg: SAR AL, 1

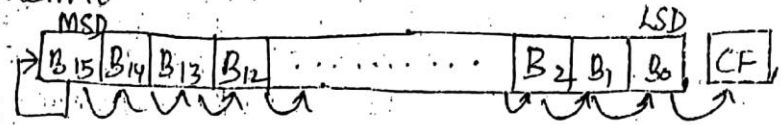
② eg: MOV CL, 02H

SAR BH, CL

a) reg8/mem8



b) reg16/mem16



(C) Rotate Instruction

(C.1) ROL (Rotate All Bits of Operand Left, MSB to LSB)

ROL Destination, Count
(memloc / reg)

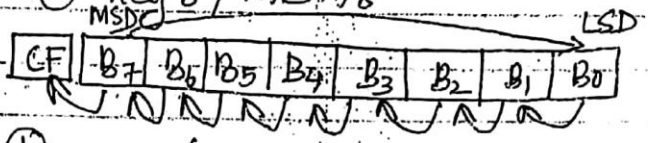
CF ← MSB ← LSB

Note:-

(1) ROL AX, 1

(2) MOV CL, 04H
ROL BL, CL

(a) reg 8 / mem 8



(b) reg 16 / mem 16

(C.2) ROR (Rotate All Bits of Operand Right, LSB to MSB)

ROR Destination, Count
(memloc / reg)

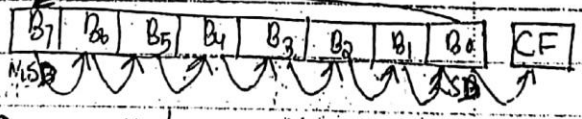
MSB → LSB → CF

Note:-

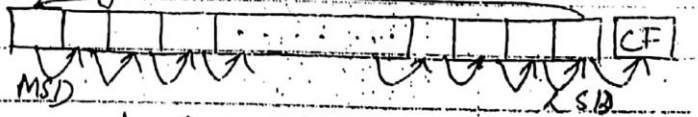
(1) ROR BH, 1

(2) MOV CL, 08H
ROR AL, CL

(a) reg 8 / mem 8



(b) reg 16 / mem 16

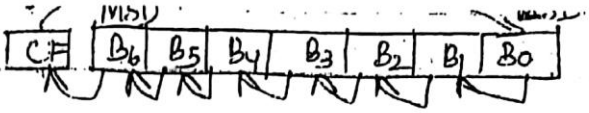


(C.3) RCL (Rotate Operand Around to the left through CF)

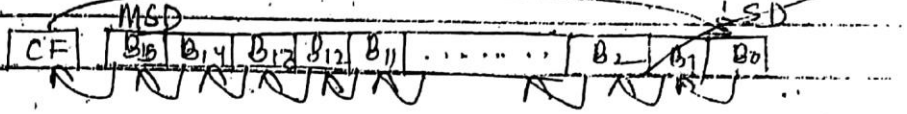
RCL Destination, Count
(memloc / reg)

Note :- (1) RCL DX, 1

(2) ....



⑥ reg16/mem16



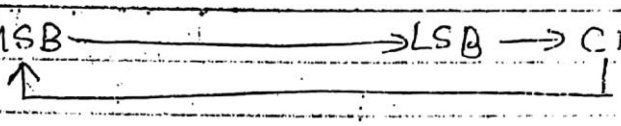
(C4) RCR (Rotate Operand Around to the Right-through CF)

RCR Destination, Count

mem/reg

Note:

① RCR BX, 1

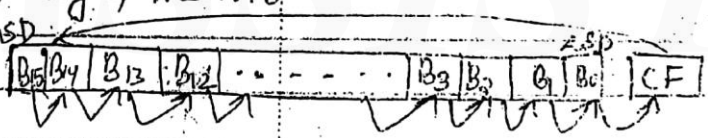


② MOV CL, 02H

RCR BX, CL

ⓐ reg8/mem8

ⓑ reg16/mem16



(4) STRING INSTRUCTION

a) REPZ / REPE (Repeat if Zero / Repeat if Equal)

Used with the compare string Instruction or with the Scan String Instruction.

eg: REPE CMPSB

Note: When a string instruction is prefixed with REPZ/REPE, the instruction execution is repeated if CX ≠ 0 and ZF = 1 & (CX) ← (CX) - 1.

The repeat operation is terminated if CX = 0 or ZF = 1.

REP if CX ≠ 0 ; CX → CX-1

⑥ REPNZ / REPNE (Repeat if not Zero / Repeat if not Equal)

Often used with the Scan string Instruction.

eg: REPNE SCASW

Note:- When a string instruction is prefixed with REPNZ / REPNE, the instruction executed is repeated if CX ≠ 0 and ZF = 0 and (CX) ← (CX) - 1.

The repeat operation is terminated if CX = 0 or ZF = 1.

⑦ MOVSB (Move String Byte)

Note:-

Used to copies a byte of string data stored in data segment into extra segment and SI & DI are automatically INC/DEC by 1 depending on Direction Flag.

MOVSB Destination String Name, Source String Name

Source byte → offset in the DS: SI Register

Destination byte → offset in the ES: DI Register

Symbolic Representation:

(MA) = (DS) × 16_{10} + (SI)

(MAE) = (ES) × 16_{10} + (DI)

(MAE) ← (MA)

If DF = 0 then (DI) ← (DI) + 1; (SI) ← (SI) + 1

If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1

⑧ MOVSW (Move String Word)

Note:-

Used to copies a word of string data stored in data segment into extra segment and SI & DI are automatically INC/DEC by 2 depending on Direction Flag.

MOVSW Destination String Name, Source String Name

Symbolic Representation:

MA = (DS) × 16_{10} + (SI)

MAE = (ES) × 16_{10} + (DI)

(e) CMPSB / CMPSW

(Compare String Bytes or String Words)

Note:-

⇒ One byte/word of a string data in extra segment is subtracted from one byte/word of a string data in the data segment and the result is used to modify flags.

⇒ CMPS instruction can be used with a REPE or REPNE prefix to compare all the elements of a string.

⇒ Source Byte(word) — Destination Byte(word)

Symbolic representation:

MA = (DS) × 16_{10} + (SI)

MA_E = (ES) × 16_{10} + (DI)

Modify flags ← (MA) − (MA_E)

	CF	ZF	SF
(MA) > (MA_E)	0	0	0
(MA) < (MA_E)	1	0	1
(MA) = (MA_E)	0	1	0

For byte Operation

If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1

If DF = 1, then (DI) ← (DI) − 1; (SI) ← (SI) − 1

For word Operation

If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2

If DF = 1, then (DI) ← (DI) − 2; (SI) ← (SI) − 2

(5) Program Execution Transfer Instructions

(a) Unconditional transfer Instructions :-

(a.1) CALL (Call a procedure & save return address on stack)

2 types

- Intra-Segment / near call: Calling a procedure stored in the same code segment m/y in which main program (calling pgm) resides.
- Inter Segment / Far call: Calling a procedure stored in different code segment m/y than that of Main pgm.

Call Near / Intra-segment (direct / indirect)

While executing near call, the SP is decremented by 2 and IP is pushed into stack [ie return address because this is the address that execution will return to after the procedure executes.]. At last the effective address (disp 16) of the procedure to be executed is loaded in IP, from the reg/memory.

eg: CALL disp16/reg/mem
(direct / indirect)

Call instruction will also load the instruction pointer with the offset of the first instruction in the procedure.

and at the end of the procedure, a RET instruction will return execution.

```
SP ← SP − 2
MAS = SS × 16_10 + (SP)
MAS ← (IP) ; return address
(IP) ← disp16/(reg)/(mem)
```

Call Far

While executing far call, the program control is transferred to another segment. The offset and segment base address of the procedure to be executed are directly given in the instruction.

eg: CALL add_offset, add_base

```
(SP) ← (SP) − 2
MAS ← (SS) × 16_10 + (SP)
MAS ← (IP)
(IP) ← add_offset
(SP) ← (SP) − 2
MAS ← (SS) × 16_10 + (SP)
MAS ← (CS)
(CS) ← add_base
```

Direct Intersegment

eg: CALL mem

```
(SP) ← (SP) − 2
MAS ← (SS) × 16_10 + (SP)
MAS ← IP
IP ← (mem)  (offset address)
(SP) ← (SP) − 2
MAS = (SS) × 16_10 + (SP)
(MAS) ← (CS)
(CS) ← (mem + 2)  (base address)
```

Indirect Intersegment

(iv) RET (Return Execution from Procedure to Calling Program)

1) Return from within

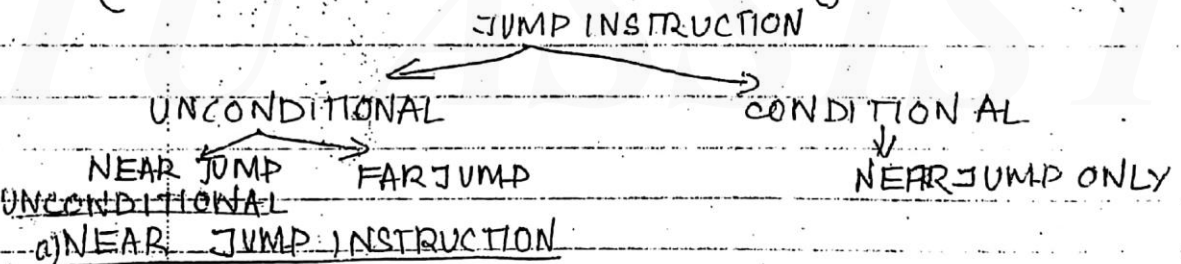
Return will be done by replacing the IP with a word from the Top of stack ie. the offset of the next instruction after the call.

2) Return from intersegment call

Return will be done by replacing IP with a word (ie. the offset of the next instru. after the call.) from the TOS. Then the SP is incremented by 2 and code segment CS is replaced with a word (ie. segment base address of the return address) from the TOS.

(18) JUMP Instruction

(Go to Specified address to get next Instruction)



UNCONDITIONAL

a) NEAR JUMP INSTRUCTION

INSTRUCTION	SYMBOLIC REPRESENTATION	EXPLANATION
a) JMP disp16	(IP) ← (IP) + disp16	The 16-bit value (disp16) given in the instruction is added to instruction pointer (IP).
b) JMP disp8	disp16 ← disp8 (IP) ← (IP) + disp16	The 8-bit value (disp8) given in the instruction is sign extended to 16 bit and added to instruction pointer (IP).
c) JMP reg/mem	(IP) ← (IP) + (reg) (IP) ← (IP) + (mem)	The 16 bit value stored in the reg/mem is added to IP.

b) FAR JUMP INSTRUCTIONS

INSTRUCTION	SYMBOLIC REPRESENTATION	EXPLANATION
a) JMP addr offset, addr base	(IP) ← addr offset (CS) ← addr base	The offset address given in the instruction is loaded in IP and the base address given in the instruction is loaded in CS reg.
b) JMP mem	(IP) ← (mem) (CS) ← (mem+2)	The content of (16-bit) m/y is moved to IP and the next word in the m/y is moved to CS register

CONDITIONAL JUMP INSTRUCTION

J <condition> disp8

If <condition> is TRUE then,

disp16 ← sign extend disp8 ; (IP) ← (IP) + disp16.

INSTRUCTION		EXPLANATION
a) JE/JZ disp8	Jump if Equal/ Jump if zero flag	Jump if ZF = 1 (zero flag)
b) JL/JNGE disp8	Jump if Less than/ Jump if not greater than or Equal.	Jump if SF ≠ OF (sign flag) (over flag)
✓ c) JLE/JNG disp8	Jump if less than or Equal/ Jump if not greater than	Jump if SF ≠ OF or ZF = 1
d) JB/JC/JNAE disp8	Jump if Below/ Jump if carry/ Jump if not above or equal.	Jump if CF = 1
✓ e) JBE/JNA disp8	Jump if below or equal/ Jump	Jump if CF = 1

INSTRUCTION		EXPLANATION
f) JP/JPE disp8	Jump if parity / Jump if parity even	Jump if PF = 1
g) JNB / JAE / JNC disp8	Jump if not Below / Jump if above or Equal / Jump if Not carry	Jump if CF = 0
h) JNBE disp8 / JA	Jump if Not Below or Equal / Jump if above	Jump if CF = 0 and ZF = 0
(i) JNP / JPO disp8	Jump if not Parity / Jump if Parity Odd	Jump if PF = 0
(j) JNO disp8	Jump if not over flow	Jump if OF = 0
k) JNS disp8	Jump if not sign	Jump if SF = 0
l) JO disp8	Jump if Overflow	Jump if OF = 1
m) JS disp8	Jump if Sign	Jump if SF = 1
n) JNE/JNZ disp8	Jump if not Equal / Jump if not zero	Jump if ZF = 0
o) JNL / JGE disp8	Jump if not less than / Jump if greater than or Equal	Jump if SF = OF
P) JNLE/JG disp8	Jump if not less than or Equal / Jump if greater than	Jump if CF = OF and ZF = 0
q) JCXZ disp8		Jump if (CX) = 0

(5.c) ITERATION CONTROL INSTRUCTION

Loop Instructions

→ Used to execute a group of instructions

→ No. of times to be execute the instructions as specified by a count value stored in CX register. (decremented by 1 afr each exe. of looped instr.)

* disp8 in the instruction (displacement).
  ↳ no. of instructions to be looped
* +ve displacement → Instructions below the loop instruction are executed.
* -ve displacement → Instructions above the loop instruction are executed.

→ E.A (Effective Address) of first Instruction of the loop is obtained by sign extending disp8 to 16-bit and adding to IP.

eg: LOOP disp8	Loop if (CX) ≠ 0 (CX) ← (CX) − 1
LOOPZ disp8 (LOOPE disp8)	Loop if (CX) ≠ 0 and ZF = 1 (CX) ← (CX) − 1
LOOPNZ disp8 (LOOPNE disp8)	Loop if (CX) ≠ 0 and ZF = 0 (CX) ← (CX) − 1

(5.d) INTERRUPT INSTRUCTIONS

→ INT Instructions are called software interrupts and used to call a procedure or subroutine on interrupt basis.

→ INT Instruction is accompanied by a type number which can be in the range of 0 to 255.

address for IP and a 16-bit base address for CS are needed.

Therefore, for each INT instruction 4 m/y locs are reserved in the first 1Kb: address space of m/y. ✓ First 2 locs ⇒ IP
✓ Next " ⇒ CS

✓ The address of the reserved m/y loc is called Vector address: ⇒ obtained by multiplying the type number by 4.
(0 – 255)

When an 8086 executes an INT instruction, it will

1) Decrement the SP by 2 & push the flags onto the stack.
2) Decrement the SP by 2 & push the contents of CS onto the stack.
3) Decrement the SP by 2 & push the contents of IP onto the stack.
4) Get a new value for IP from an absolute memory address, then multiplying the type number specified in the instruction by 4.

eg: INT 8

$\Rightarrow 4 \times 8 = 32_{10} = 20H$ $\quad \frac{32|0}{16|2}$

The new IP will be read from address 00020H.

5) Get a new value for CS from an absolute memory address, then multiplying the type number specified in the instruction by 4 and plus 2.

eg: INT 8 $\quad \frac{34|2}{16|2}$

$4 \times 8 = 32_{10} + 2 = 2216H$

The new CS value will be read from address

$= 00022H$

6) Reset both IF and TF. Other flags are not affected.

Symbolic Representation

a) INT type

(SP) ← (SP) − 2 ; (MAs) ← flags
IF ← 0 ; TF ← 0
(SP) ← (SP) − 2 ; (MAs) ← (CS)
(SP) ← (SP) − 2 ; (MAs) ← (IP)
(IP) ← (00000 : (type × 4))
(CS) ← (00000 : (type × 4))

For each push operation the stack m/y addle :-

$MA_S = (SS) \times 16_{10} + (SP)$

b) INTO

If overflow flag is 1, then type 4 interrupt is performed.

(SP) ← (SP) − 2 ; (MAs) ← flags
IF ← 0 TE ← 0
(SP) ← (SP) − 2 ; (MAs) ← (CS)
(SP) ← (SP) − 2 ; (MAs) ← (IP)
(IP) ← (0010)H ; (CS) ← (0012)H

Note :- $4 \times 4 = 16_{10} = 10_H$; $16 + 2 = 18_{10} = 12_H$

c) IRET (Interrupt Return)

→ Used to terminate an interrupt service procedure and transfer the program control back to main program.
→ On execution of this instruction the content of TOS are moved to IP, CS and flag registers one by one. After every pop operation the SP is incremented by 2.

(IP) ← (MAs) ; (SP) ← (SP) + 2
(CS) ← (MAs) ; (SP) ← (SP) + 2
Flag ← (MAs) ; (SP) ← (SP) + 2

Note :- ... (CD)

INSTRUCTION	Symbolic Representation	Explanation
a) CLC	CF ← 0	The carry-flag is reset to zero
b) CMC	CF ← ~CF	The carryflag is complemented
c) STC	CF ← 1	The carryflag is set to one.
d) CLD	DF ← 0	The Direction flag is reset to zero
e) STD	DF ← 1	The Direction flag is set to one.
f) CLI	IF ← 0	The Interrupt flag is reset to zero
g) STI	IF ← 1	The Interrupt flag is set to one.
h) HLT	Halt program execution.	Used to terminate a program On execution of this instruction, the processor enter into an idle state and perform no operation until an interrupt occurs.
(i) WAIT	Wait for test line active.	This instruction causes the processor to enter into an idle state or wait state & continue to remain in that state until a signal is asserted on TEST i/p pin or until a valid interrupt signal is received on the INTR or NMI interrupt input pin.
(j) LOCK	Ensures that the shared s/m resources are not taken over by other bus masters in the middle of instruction execution.	
k) NOP	No operation is performed for 3 clock periods, when this instruction is executed the processor waits for 3 clock periods and then next instruction is executed.	

END TYPE 1
NOTE

MORE NOTE
BELOW

(1) Study of Intel 8086 Microprocessor

Aim: To study the given Microprocessor Trainer Kit and to familiarize with the various Hardware and Software specifications

Hardware specifications:

(1) FREQUENCY: CPU gets Clock signal from Clock generator 8284 which uses a quartz Crystal of frequency, 14.318 MHz. Processor Clock frequency = (Crystal frequency)/3 = 14.318/3 = 4.77 MHz. PCLK signal frequency of VXT Bus connector = (4.77 MHz)/2 = 2.385 MHz.

(2) MEMORY:

(a) EPROM:

Intel 2764 (8 KB Chip) – 2 Nos.

Memory Map: F0000 H to F3FFF H (Total 16KB)

(b) SRAM (Static RAM):

6264 (8 KB Chip) – 2 Nos.

Memory Map: 00000 H to 03FFF H (Total 16KB)

User area of RAM - 01000 H to 03FFF H

	Memory Usage	Memory storage (capacity)	Memory Map (Linear form)	Memory Map (Segment:Offset form)	Memory Chip used
			FFFFF -- -- F4000	F000:FFFF -- -- F000:4000	(this area is Not Used)
	Monitor EPROM	16 KB	F3FFF -- -- F0000	F000:3FFF -- -- F000:0000	EPROM Intel 2764 × 2 Nos. 8K + 8K = 16 KB
			EFFFF -- -- 04000	0000:4000	(this area is Not Used)
16 KB	**User area of RAM**	12 KB	**03FFF** -- -- **01000**	**0000:3FFF** -- -- **0000:1000**	SRAM 6264 × 2 Nos. 8K + 8K = 16 KB
	System Stack (System Scratch Pad area)	3 KB	00FFF -- -- 00400	0000:0FFF -- -- 0000:0400	
	Interrupt Vector Table (IVT)	1 KB	003FF -- -- 00000	0000:03FF -- -- 0000:0000	

400H = 1K; 800H = 2K; 1000H = 4K; 2000H = 8K; 4000H = 16K and so on.

(3) INPUT/OUTPUT PERIPHERALS:

Parallel: 48 I/O lines using two numbers of 8255 PPI

Serial: One number of RS232C Serial Interface using 8251 USART *(RS stands for Recommended Standard)*

Timer: 3 Channel 16-bit programmable Timer 8253. Channel 0 is used as baud rate generator for the serial port

Interrupts: 8 Interrupt lines using 8259 Programmable Interrupt Controller

(4) DISPLAY:

20 × 4 Alphanumeric LCD Display

(5) Keyboard: IBM PC-AT Keyboard

(6) On-board Battery Back-up for SRAM: 3.6 V, Ni-Cd cell

(7) Internal Power Supply of the Kit: +5V DC, 3A (regulated)

(8) Mains AC Input: 230 V, 50 Hz AC

(9) BUS Expansion: An expansion Slot called VXT BUS is used in the Kit which facilitates addition of extra hardware on to the Kit. Fully buffered Address bus, Data bus and Control bus are brought out to this expansion bus for interfacing with external devices.

Important Commands and description:

- is the Command Prompt

<cr> - Carriage Return (Enter Key)

1) **Assemble Command:** – will generate op-code from the mnemonic entered

#A <cr>

After entering the 'A' command, the Kit displays: 'Line Assembler'

Then it asks the starting address. Here enter the starting address. It can be any number in the range 1000 to 3FFF (i.e., user area of RAM)

2) **Unassemble (Disassemble) Command:** - will generate mnemonic from the op-code

#U <cr>

After entering the 'U' command, the Kit displays: 'Disassembler'

Then it asks the starting address. Here enter the starting address. It can be any number in the range 1000 to 3FFF (ie, user area of RAM)

3) **Substitute memory Command:** (to modify or view memory content – byte or word)

#SB <addr> <cr> ; view or modify byte

#SW <addr> <cr> ; view or modify word

The address (addr) can be entered in either 'Segment:Offset' form or only 'Offset form'.

Use Enter Key for incrementing memory address

Use – (minus) Key for decrementing memory address

Dot (.) command can be used for terminating a Command and to return to the Command Prompt.

4) **Register view /modify Command:–** to view/modify register contents
#R <cr>

Initial values of Register Contents:
AX = 0000H; BX = 0000H; CX = 0000H; DX = 0000H

SP = 0800H; BP = 0000H; SI = 0000H;; DI = 0000H

CS = 0000H; DS = 0000H; SS = 0000H; ES = 0000H

IP = 1000H; FL = 0000H;

5) **Trace Command (Single step Execution): -** This helps the user to execute programs in steps (i.e., Single step mode)
That is, executing Instruction by Instruction. This is helpful in debugging the programs.
#TR <addr> <cr>
Procedure:
i) Enter the Trace command as above.
ii) Now the first instruction is executed and the registers are automatically updated.
iii) The next instruction to be executed and the memory locations are displayed.
iv) To view the register contents after execution of each instruction, Press the dot (.) key for command termination and to return control to the monitor program. Here type the # R command to view the Register contents, or #SB or #SW command to view the memory contents. Again Press the dot key for command termination and to return control to the monitor program. To continue single step execution, from where it has stopped, enter the Command '#TR' alone, without any argument.

6) **Normal Execution Command: - GO command**
GO command is used to RUN a program. This command transfers control of the CPU from the Monitor program to User Program.
Syntax: **GO <start addr> <cr>**
Procedure:
i) Type GO command and the address from where execution should start.
ii) Now the control is transferred to the User program at the entered address location and the display shows 'Executing …', since the Last Instruction used in the program is **HLT**. Or, equivalently a JMP instruction can be used instead of HLT, which will put the execution in an infinite Loop. For example: HERE: JMP HERE
iii) To Exit from execution and to return control to the monitor program, press **RES** (Reset key) or **INT** (Interrupt key). But in this case the contents of registers will be cleared and hence cannot be observed.

'Dot' command will not work for terminating the execution, in this situation.

Note1: RES Key can also be used for breaking an infinite loop execution. INT key can also be used for Single Step execution
Note2: In the Keyboard, in the case of Keys having two characters written on a Key (for example the #Key), the upper Character is obtained by depressing the Shift Key first, then releasing the finger and then depressing the Character Key. Simultaneous depressing of the Shift Key and the Character Key will not work.

7) **Execution with Break Point: - (ie, GO command with break point)**

Syntax: **GB <start addr> <end addr> <cr>**
When the execution reaches the end address, execution automatically stops and the control is returned to the Command Prompt. Now the register contents can be observed using #R command or the memory contents by the #SB or #SW commands. This command is useful in observing register contents as they are not cleared since we are not using the RES or INT keys for terminating execution in this case.

Aim: To add two 16-bit unsigned integer numbers stored in memory locations and to store the 16-bit sum into memory locations.

Algorithm:

Assumption: It is assumed that the numbers to be added are stored in consecutive memory locations.

Step 1: Start.
Step 2: Initialise a register to hold carry, to zero.
Step 3: Bring the first number from memory into another register
Step 4: Add the second number from memory to the contents of the above register.
Step 5: If a carry is not generated, go to Step 7.
Step 6: Increment the Carry holding register.
Step 7: Store the result of addition into the next consecutive memory location.
Step 8: Store the contents of the Carry holding register into the next memory location.
Step 9: Stop

Assembly language Program:

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000	C7, C1, 00, 00		MOV CX, 0000H	CX holds the Carry and is initialised to 0.
1004	8B, 06, 00, 12		MOV AX, [1200]	
1008	03, 06, 02,12		ADD AX, [1202]	
100C	73, FE		JNC L1	
100E	41		INC CX	
100F	89, 06, 00, 13	L1:	MOV [1300], AX	
1013	89, 0E, 02, 13		MOV [1302], CX	
1017	F4		HLT	

Sample Input:

First set of data:
NUM1: [1200, 1201] = 1234H
NUM2: [1202, 1203] = 5613H
SUM: [1300, 1301, 1302, 1303] = 0000 6847H

```
1234 +
5613
----------
00006847
----------
```

Second set of data:
NUM1: [1200, 1201] = FFFFH
NUM2: [1202, 1203] = FFFFH
SUM: [1300, 1301, 1302, 1303] = 0001 FFFE H

```
FFFF +
FFFF
----------
0001FFFE
----------
```

Note 1: Forward JMP and Forward CALL instructions: In this case the 'Target Address'. JMP and CALL instructions are not known before hand until we reach the Target address lo To resolve such problem, use the 'Self Address' of the instruction temporarily as the 'target and proceed till the end. Later on replace the 'Self Address' by the actual 'Target Address' b the JMP or CALL instructions using the Assemble Command '#A', making use of the act address.

But there is no such problem in the case of Backward JMP and Backward CALL ir

Note 2: The destination Label (ie, the target address) of all Jump instructions must be in the range of -128 bytes (ie,80H) to +127 (ie,7F H) bytes relative to the address of the instruction immediately after the Jump instruction. That is, from the current IP location. The relative address of the current IP location is taken as 00H.

Note 3: While entering mnemonics of instructions, the operands should be separated by comma, without a space. For example, MOV AX,BX (no space permitted between AX and BX). The Hexadecimal numbers can be entered with or with out suffixing of 'H'. For example, MOV AX,1234H or MOV AX,1234 (both are valid Hexadecimal numbers).

2 (b) Subtraction of Two 16-bit numbers

Aim: To subtract two 16-bit unsigned integer numbers stored in memory locations and to store the 16-bit difference into memory locations. Borrow is to be used as a word, indicating the sign (+ or -).

Algorithm:

Assumption: It is assumed that the numbers to be subtracted are stored in consecutive memory locations.

Step 1: Start.
Step 2: Initialise a register to hold borrow, to zero.
Step 3: Bring the first number from memory into another register
Step 4: Subtract the second number from memory from the contents of the above register.
Step 5: If a borrow is not generated, go to Step 8.
Step 6: Increment the Borrow holding register.
Step 7: Take two's complement of the result.
Step 8: Store the result of subtraction into the next consecutive memory location.
Step 9: Store the contents of the Borrow holding register into the next memory location.
Step 10: Stop

Assembly language Program:

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV CX, 0000H	CX holds the Borrow and is initialised to 0.
			MOV AX, [1200]	
			SUB AX, [1202]	
			JNB L1	
			INC CX	
			NEG AX	Taking 2's complement
		L1:	MOV [1300], AX	
			MOV [1302], CX	
			HLT	

Sample Input:

First set of data:
NUM1: [1200, 1201] = 5678H
NUM2: [1202, 1203] = 1101H
DIFFERENCE: [1300, 1301, 1302, 1303] = 0000 4577H
(0000 indicates +ve result)

```
5678 -
1101
----------
00004577
----------
```

Second set of data:
NUM1: [1200, 1201] = 1101H
NUM2: [1202, 1203] = 5678H
DIFFERENCE: [1300, 1301, 1302, 1303] = 0001 4577H

```
1101 -
5678
----------
00014577
----------
```

(0001 indicates -ve result)

3 (a) Addition of Two 64-bit numbers (Multi-byte addition)

Aim: To add two 64-bit unsigned integer numbers stored in memory locations and to store the 80-bit sum into memory locations.

Theory: Here the addition can be done 'byte by byte' or 'word by word' starting from the least significant byte or word. The instruction for addition is ADC, operating on Byte register or Word register. While using ADC the initial Carry should be cleared.

Algorithm:

Assumption: It is assumed that the numbers to be added are stored in consecutive memory locations. Also it is assumed that a Word by Word addition is done.

Step 1: Start.
Step 2: Initialise a register to hold the word count value, to 4. (The preferred register is CX)
Step 3: Clear the Carry flag.
Step 4: Initialise three memory pointer registers to point to the least significant word locations of the First number, the second number and the result location, in memory.
Step 5: Bring the word pointed by the first pointer into a register.
Step 6: Add the word pointed by the second pointer in memory to the contents of the above register using **add with carry** (ADC) instruction. Store the partial result in memory, making use of the corresponding memory pointer.
Step 7: Increment the three memory pointes to point to the next word locations. Decrement the word counter. If the counter value is not zero, go to step 5.
Step 8: Store the Carry generated by the addition of the most significant words, into the result memory locations.
Step 9: Stop

Assembly language Program:

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV CX, 0004	CX holds the word count value
			CLC	Initial carry is cleared
			MOV SI, 2000	SI is used as a Memory pointer and is initialized to point to location 2000
			MOV DI, 3000	DI is used as a Memory pointer and is initialized to point to location 3000
			MOV BX, 3500	BX is used as a Memory pointer and is initialized to point to location 3500 (Instead of BX, BP can also be used as a pointer)
		L1:	MOV AX, [SI]	
			ADC AX, [DI]	Do not use ADD instruction (think why?)
			MOV [BX], AX	
			INC SI	
			INC SI	
			INC DI	
			INC DI	
			INC BX	
			INC BX	
			LOOP L1	
			MOV AX,0000	
			ADC AX,0000	
			MOV [BX],AX	Store the carry generated from the addition of the most significant words into memory
			HLT	

(Alternatively, the Last 4 lines of the above can be replaced by):

	MOV AX,0000
	JNC L2
	MOV AX,0001
L2:	MOV [BX],AX
	HLT

Sample Data:

NUM1: [2000-2007] → [SI] 4444 3333 2222 1111 (2000)
NUM2: [3000-3007] → [DI] F987 F678 F345 F123

SUM: [3500-3509] → [BX] 0001 3DCC 29AC 1568 0234

3 (b) Subtraction of Two 64-bit numbers

Aim: To subtract two 64-bit unsigned integer numbers stored in memory locations and to store the 64-bit difference into memory locations.

Algorithm:

Assumption: It is assumed that the numbers to be subtracted are stored in consecutive memory locations. Also it is assumed that a 'Word by Word' subtraction is done and that the Subtrahend is smaller than the Minuend.

Step 1: Start.
Step 2: Initialize a register to hold the word count value, to 4. (The preferred register is CX)
Step 3: Clear the Borrow flag (ie, same as Carry flag)
Step 4: Initialize three memory pointer registers to point to the least significant word locations of the First number, the second number and the result location, in memory.
Step 5: Bring the word pointed by the first pointer into a register.
Step 6: Subtract the word pointed by the second pointer in memory from the contents of the above register using Subtract with borrow instruction. Store the partial result in memory, making use of the corresponding memory pointer.
Step 7: Increment the three memory pointers to point to the next word locations. Decrement the word counter. If the counter value is not zero, go to step 5.
Step 8: Stop

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV CX, 0004	CX holds the word count value
			CLC	
			MOV SI, 2000	SI is used as a Memory pointer and is initialized to point to location 2000
			MOV DI, 3000	DI is used as a Memory pointer and is initialized to point to location 3000
			MOV BX, 3500	BX is used as a Memory pointer and is initialized to point to location 3500 (Instead of BX, BP can also be used as a pointer)
		L1:	MOV AX, [SI]	
			SBB AX, [DI]	

			MOV [BX], AX	
			INC SI	
			INC SI	
			INC DI	
			INC DI	
			INC BX	
			INC BX	
			LOOP L1	
			HLT	

Sample Data:

NUM1: [2000-2007] → [SI] F987 F678 F345 F123 -
NUM2: [3000-3007] → [DI] 4444 3333 2222 1111

Difference: [3500-3507] → [BX] B543 C345 D123 E012

CODE CONVERSIONS

[Note: Read Chapter 10 (Code conversion) of Microprocessor Architecture, Programming and Applications with 8085 by Ramesh Gaonkar, for understanding the need for Code conversions and the algorithms]

The need for Code conversion

In microcomputer applications, various number systems and codes are used to input data or to display results. The ASCII (American Standard Code for Information Interchange) keyboard is a commonly used input device for microcomputers. Similarly, alphanumeric characters (letters and numbers) are displayed on a display device using the ASCII code. However, inside the microprocessor, data processing is usually performed in binary. In some instances, arithmetic operations are performed in BCD numbers. Therefore data must be converted from one code to another code.

The programming techniques used for code conversion fall into four general categories:

(i) Conversion based on the position of a digit in a number (eg. BCD to binary and Binary to BCD conversions).
(ii) Conversion based on sequential order of digits (eg, Binary to ASCII and ASCII to binary).
(iii) Conversion based on hardware consideration (eg, Binary to 7-segment code using table look-up procedure).

4 (a) BCD to Binary Conversion

Aim: To convert the given two digit packed BCD number to its equivalent binary number.

Theory: BCD represents each of the digits of an unsigned decimal as the 4-bit binary equivalents. Unpacked BCD representation contains only one decimal digit per byte. The digit is stored in the lower nibble; the upper nibble in not relevant to the value of the represented number (usually the upper nibble is zero). Packed BCD representation packs two decimal digits into a single byte.

Decimal	Binary	BCD	
		Unpacked	Packed
0	0000 0000	0000 0000	0000 0000
1	0000 0001	0000 0001	0000 0001
2	0000 0010	0000 0010	0000 0010
8	0000 1000	0000 1000	0000 1000
9	0000 1001	0000 1001	0000 1001
10	0000 1010	0000 0001 0000 0000	0001 0000
11	0000 1011	0000 0001 0000 0001	0001 0001
12	0000 1100	0000 0001 0000 0010	0001 0010
19	0001 0011	0000 0001 0000 1001	0001 1001
20	0001 0100	0000 0010 0000 0000	0010 0000

Invalid BCD Numbers

These binary numbers are not allowed in the BCD code: 1010, 1011, 1100, 1101, 1110, 1111

The conversion uses the principle of positional weighting in a number. To convert a 2-digit packed BCD number into its equivalent binary, first the digits are to be separated into MSD and LSD and then each digit is multiplied by its position value (place value) and added together.

ie, Binary equivalent = MSD × 0AH + LSD

Algorithm:

1) Start
2) Get the BCD number from memory and store it in a register.
3) Separate the LSD of the two-digit packed BCD number (ie, unpack the packed BCD) by ANDing (masking) the packed BCD number with 0FH.and storing it in a register.
4) Separate the MSD of the packed BCD number by ANDing (masking) the packed BCD number with F0H.and shifting the bits 4 times to the right to get the MSD. Multiply the MSD by the place value 10 (0AH).
5) Add the Product obtained by multiplication in step 3, with the LSD obtained in step 2
6) Stop

ALP

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV BX,1100	BX is used as a memory pointer and is initialized to point to location 1100
			MOV AL,[BX]	
			MOV DL,AL	Value in AL is copied to DL
			AND DL,0F	Now DL contains LSD
			AND AL,0F0	
			MOV CL,04	
			ROR AL,CL	Now AL contains MSD
			MOV DH,0A	
			MUL DH	(AX) ← (AL) × (DH) Now AX contains MSD×0A AH will be zero always, because the maximum value possible is 09×0A which is 5A
			ADD AL,DL	Since AH is zero
			MOV [BX+1],AL	
			HLT	

Input and Output:

First Set of Data:

Input: [1100] = 72 (BCD)

Output: [1101] = 48 (Binary in HEX)

Second set of Data:

Input: [1100] = 99 (BCD)

Output: [1101] = 63 (Binary in HEX)

4 (b) Binary to BCD Conversion

Aim: To convert an 8-bit Binary number to its BCD equivalent.

Theory: An 8-bit unsigned binary number has the range from 00H to FF H. Its BCD equivalent number has the range from 00 to 255 (packed BCD). BCD equivalent number is obtained from the 8-bit binary number by finding the number of 100s (ie, 64Hs), 10s (ie, 0AHs) and 1s (01Hs). Let the most significant digit be BCD_3, the middle digit be BCD_2 and the least significant digit be BCD_1

Algorithm:

1) Start
2) If the binary number is less than 64H (ie, 100dec), go to Step 3; otherwise, divide the binary number by 64H or equivalently, subtract 64H repeatedly until the remainder is less than 64H. The quotient of division is the most significant BCD digit, BCD_3. (It is the same as the number of times the repeated subtraction is done).
3) If the binary number is less than 0AH (ie, 10dec), go to Step 4; otherwise, divide the binary number by 0AH or equivalently, subtract 0AH repeatedly until the remainder is less than 0AH. The quotient of division is the middle BCD digit, BCD_2. (It is the same as the number of times the repeated subtraction is done).
4) The remainder from Step 3 is the least significant BCD digit, BCD_1.
5) Store the three BCD digits obtained in steps 2, 3 and 4 in consecutive memory locations as either (i) unpacked BCD digit for the MSD and two-digit packed BCD digits for the LSD and Middle digit or (ii) three unpacked BCD digits, BCD_3, BCD_2, BCD_1 (This is better).
6) Stop

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV BX,2000	BX is used as a memory pointer and is initialized to point to location 2000
			MOV AL,[BX]	
			MOV DX,0000	DX holds the BCD digits
		HUND:	CMP AL,64	Comparison is done by subtraction
			JC TEN	
			SUB AL,64	
			INC DL	DL holds the number of times subtraction is done (ie, 100s)
			JMP HUND	
		TEN:	CMP AL,0A	
			JC UNIT	
			SUB AL,0A	The last subtraction gives BCD1 in AL
			INC DH	DH holds the number of times subtraction is done (ie, 10s)
			JMP TEN	
		UNIT:	MOV CL,04	
			ROR DH,CL	ROR 4 times, will swap nibbles in a byte or swap bytes in a word. 10s in DH is moved to High Nibble and added to Units.
			ADD AL,DH	
			MOV [BX+1],AL	
			MOV [BX+2], DL	
			HLT	

Input Data: (2000) = FE

Output Result: (2001) = 54 (Packed BCD)

(2002) = 02 (Most significant digit as unpacked BCD)

(But display of 05, 04, 02 is better)

4 (c) HEX digit to ASCII code Conversion

Aim: To convert a given Hexadecimal digit (ie, 0, 1,...9, A, B, C, D, E, F) into its equivalent ASCII code in Hex.

Theory:

Each HEX digit can be converted to an 8-bit ASCII code as shown below. For HEX digit in the range 0 to 9, add 30 to get the ASCII code. For HEX digit in the range A to F, add 37 (ie 30+7) to get the ASCII code.

HEX digit	ASCII Code in HEX	
0	30	Here the difference between the two is **30** (For example, 32-2 = 30)
1	31	
2	32	
3	33	
4	34	
5	35	
6	36	
7	37	
8	38	
9	39	
..	..	
..	..	
A	41	Here the difference between the two is **37 (ie, = 30 + 7)** (For example, 41 - A = 37)
B	42	
C	43	
D	44	
E	45	
F	46	

Algorithm:

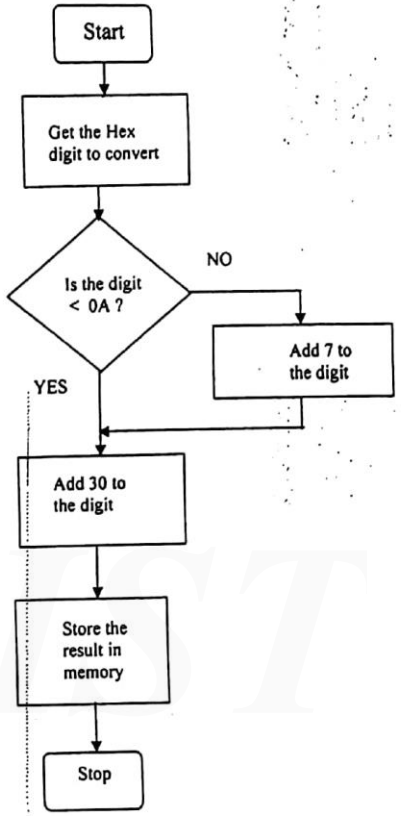
1) Start
2) Compare the given HEX digit with '0A' . If it is less than 0A, add 30 to it and Go to step 4.
3) Else, add 30 and 7 to it.
4) Stop

Input Data (HEX digit): Output Result (ASCII code in HEX):

First sample of data:
(2000) = 07 (2001) = 37H

Second sample of data:
(2000) = 0B (2001) = 42H

FLOW CHART:



OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV SI, 3000H	SI pointer initialized to point to the data location.
			MOV AL,[SI]	
			SUB AL,30H	
			CMP AL,0AH	
			JC L1	
			SUB AL,07H	
		L1:	MOV [SI+1],AL	
			HLT	

Input Data:
First sample data:
(3000) = 33H

Output Result (HEX value)

(3001) = 03H

Second sample data:
(3000) = 42H

(3001) = 0BH

5 (a) BCD addition

Aim: To add two 4-digit (16-bit) BCD numbers stored in memory

Theory: The 8086 processor will perform only binary addition. Hence for BCD addition, the binary addition of BCD data is performed first and then the Sum is corrected to get the result in BCD. The following is the correction to be done:

(iv) If the binary sum of lower nibbles of the BCD numbers stored in AL, exceeds 9 or if there is an Auxiliary carry generated from the lower nibble to the upper nibble of the Sum in AL, then 06 is added to the lower nibble of the Sum in AL.

(v) If the binary sum of upper nibbles of the BCD numbers stored in AL, exceeds 9 or if there is a Carry generated from the upper nibble of the Sum in AL, then 60 is added to the Upper nibble of the Sum in AL.

The correction is automatically done using the DAA instruction.

DAA instruction: Decimal Adjust AL after BCD addition. DAA works on 8-bit only. The result of the addition must be in AL for DAA to work correctly.

Algorithm:

1. Start
2. Get the BCD numbers from the memory locations.
3. Add the lower two digits of the 4-digit BCD numbers and adjust for correction using DAA instruction.
4. Add the upper two digits of the 4-digit BCD numbers along with Carry, if any, generated from Step 3 and adjust for correction using DAA instruction.
5. Store the BCD sum and the final Carry in memory locations
6. Stop

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV SI,2000H	SI pointer initialized to point to the data location.
			MOV CL,00H	To hold the final carry (initialised to 0)
			MOV AX,[SI]	
			MOV BX,[SI+2]	
			ADD AL,BL	
			DAA	
			MOV DL,AL	
			MOV AL,AH	
			ADC AL,BH	
			DAA	
			MOV DH,AL	
			JNC L1	
			INC CL	
		L1:	MOV [SI+4],DX	
			MOV [SI+6],CL	
			HLT	

Input Data:

BCD NUM1: [2000-2001]: 1245

BCD NUM2: [2002-2003]: 9566

```
  1245 +
  9566
------
  010811
------
```

Output Result:

BCD SUM: [2004-2005]: 0811
[2006]: 01

5 (b) BCD subtraction

Aim: To subtract two 4-digit (16-bit) BCD numbers stored in memory

Theory: The 8086 processor will perform only binary subtraction. Hence for BCD subtraction, the binary subtraction of BCD data is performed first and then the Difference is corrected to get the result in BCD. The following is the correction to be done:

(i) If the binary difference of lower nibbles of the BCD numbers stored in AL, exceeds 9 or if there is an Auxiliary carry generated from the lower nibble to the upper nibble of the difference in AL, then 06 is subtracted from the lower nibble of the Difference in AL.

(ii) If the binary difference of upper nibbles of the BCD numbers stored in AL, exceeds 9 or if there is a Carry generated from the upper nibble of the Difference in AL, then 60 is subtracted from the Upper nibble of the Difference in AL.

The correction is automatically done using the DAS instruction.

DAS instruction: Decimal Adjust AL after BCD subtraction. DAS works on 8-bit only. The result of the subtraction must be in AL for DAS to work correctly.

Algorithm:

1. Start
2. Get the BCD numbers from the memory locations.
3. Subtract the lower two digits of the 4-digit BCD numbers and adjust for correction using DAS instruction.
4. Subtract the upper two digits of the 4-digit BCD numbers along with Borrow, if any, generated from Step 3 and adjust for correction using DAS instruction.
5. Store the BCD difference and the final Carry (borrow) in memory locations
6. Stop

OFFSET ADDRESS	MACHINE CODE	LABEL	MNEMONICS	COMMENT
1000			MOV SI,2000H	SI pointer initialized to point to the data location.
			MOV CL,00H	To hold the final carry (initialised to 0)
			MOV AX,[SI]	
			MOV BX,[SI+2]	
			SUB AL,BL	
			DAS	
			MOV DL,AL	
			MOV AL,AH	
			SBB AL,BH	
			DAS	
			MOV DH,AL	
			JNC L1	
			INC CL	
		L1:	MOV [SI+4],DX	
			MOV [SI+6],CL	
			HLT	

Input Data:

BCD NUM1: [2000-2001]: 9123H

BCD NUM2: [2002-2003]: 7368H

```
 9123 -
 7368
------
001755
------
```

Output Result:

BCD Difference: [2004-2005]: 1755
[2006]: 00

Study of MASM and Debug commands and usage of CodeView debugger

Microsoft Macro Assembler **masm.exe** is used for assembling an 8086 assembly language program. The other files associated with it are **link.exe** (linker program), **ml.exe** and **ml.err**. For debugging and execution, a DOS debugger, **debug.exe** or Microsoft debugger **cv.exe** (CodeView) is used. The assembler version used is ver 6.11 and CodeView version used is ver 3.14. DOS debugger gives a command line interface whereas CodeView debugger gives an IDE interface (Integrated Development Environment).

Configuration of Path setting:

Store the files **masm.exe, link.exe, ml.exe, ml.err, debug.exe, cv.exe and cv.hlp** in a folder (named say, masm) and the folder is placed in a drive (for example under c:\.) The absolute path of this folder is included in the path statement as follows for gaining universal access to the above files from any location (in any drive).

My Computer → Properties →Advanced → Environment Variables → System variables → Path → Edit → Edit System variables → Variable value →
Here add, after semi colon, the absolute path of the 'masm' folder, after the last entry in the path statement.
For example **;c:\masm** Click **ok, ok, ok.** (take care not to delete the existing path-do only append).

Alternatively, the above 7 files may be copied in to the 'Current folder' where user programs are stored. But in this case, universal access of 'masm' files are not obtained.

Creating Source file and Assembling:

Create an 8086 ALP source file with extension **.asm** using any text editor such as notepad, edit.com, Norton editor, wordpad etc. The file name should not exceed 8 characters in length and file extension is 3 characters.
Go to the DOS command prompt. In Windows XP: start → run → cmd
Now a default folder is displayed. (For example: c:\Documents and settings\Prog>). Under this folder create a user folder to store the user files namely: **.asm, .lst, .obj and .exe** files.
For assembling the source file, use the following command:
If the source file is **add.asm,** type the command as:

c:\ …\user> **masm.exe /la add.asm**
(here **.exe** extension and .asm extension are optional) (/la means list all)
(That is, you can even type: c:\...\user>masm /la add)
This will assemble the source file and generate two files: **add.lst** and **add.obj**
(Note: For exiting from DOS Command Window, use 'exit' command)
The **command summary** of masm can be obtained by the command: **masm /?**
Usage: masm /options source(.asm),[out(.obj)],[list(.lst)],[cref(.crf)][;]

/c	Generate cross-reference
/D<sym>[=<val>]	Define symbol
/e	Emulate floating point instructions and IEEE format
/I<path>	Search directory for include files
/l[a]	Generate listing, a-list all
/M{lxu}	Preserve case of labels: l-All, x-Globals, u-Uppercase Globals
/n	Suppress symbol tables in listing
/t	Suppress messages for successful assembly
/w{012}	Set warning level: 0-None, 1-Serious, 2-Advisory
/X	List false conditionals
/Zi	Generate symbolic information for CodeView
/Zd	Generate line-number information

Linking the .obj file, debugging and running the .exe file:

Type the command as:
c:\ …… \user> **link.exe add.obj**
(here **.exe** extension and .obj extension are optional)

(That is, you can even type: c:\...\user>link add)
If a semicolon is typed after .obj the prompt options will not be asked.
ie., type c:\ … \user> **link add;**
Here, **add.exe** file is created.
Since .exe files are self executing files, the file name can be given as a dos command in the dos command prompt for running the program and for observing the result. (note:- if a display routine is written in the main program a display can be seen in the screen – other wise nothing can be seen).
Alternatively, the **.exe** file can be loaded into memory, debugged and executed using DOS debugger (debug.exe) or Microsoft debugger cv.exe (CodeView).

CodeView Debugger:

Microsoft debugger **CodeView** (cv.exe) (version 3.14) can be used for loading the .exe file to memory, debugging and executing. CodeView gives an IDE interface (Integrated Development Environment) and is a better debugger than DOS debugger **debug.exe.**
Usage: c:\ …\user> **cv.exe add.exe** (.exe extension of cv and add are optional)
(That is, you can even type: c:\…\user>cv add). In the CV Window, go to the 'Run' menu, click 'Restart' and then click 'Animate' for running the Program.
The **Result** of execution can be observed in Memory by typing the following command in the CV command prompt: For example: **> d ds: 0 8** (where 8 is the no. of bytes 'n' to be displayed).

DOS Debugger:

Alternatively, DOS Debugger can also be used.
Type the command as:
c:\ … \user> **debug.exe add.exe** (here .exe extension of debug is optional)

(That is, you can even type: c:\...\user>debug add.exe)

Now the debug command prompt, hyphen, is displayed.
The debug command summary is obtained by: **-?**
To execute the user program **add.exe,** give the debug command as:
- g.= 0000 xxxx where 0000 is the starting offset address and xxxx denotes the ending offset address. The ending offset address (which is a break point) can be obtained from the offset address of the MOV AH, 4CH instruction displayed in the **.lst** file. The result of execution can be observed in **memory** by the debug command: **-d ds:0000** (if no ending offset address is given, 128 bytes will be displayed by default). The result of execution can also be observed in **registers** using the debug command **–r.**
(Note: For exiting from DOS Debugger Window, use 'q' command (quit))

Example:
(1) To add two 16-bit unsigned numbers stored in memory and to get the 32-bit sum in memory:
add.asm

```
.MODEL SMALL
.STACK 1000H
```

These dot directives define the stack segment. This will avoid the warning indication namely, "Warning: no stack segment" during the Linking process. These are not absolutely necessary.

```
        ;Logical Data Segment DATA1 begins
DATA1   SEGMENT
  NUM1  DW 1234H  ;DW is a directive to tell the assembler to reserve two successive memory
                  ; locations for a word type variable, to store the values in those memory
                  ; locations and to assign the name NUM1 to the first memory location.ie The
                ; First byte. NUM1 can be considered as an array of one word.
  NUM2  DW 0F562H ; Pre-fix 0 for hexadecimal numbers starting with A to F.
```

```
SUM     DW 2 DUP(0) ; Sets aside storage for two words in memory and assigns the name SUM to the
DATA1   ENDS        ;first memory location. The DUP(0) part initializes the two words to all zeros.

;Logical Code Segment CODE1 begins

CODE1   SEGMENT
        ASSUME CS:CODE1, DS:DATA1 ;It is a directive to tell the assembler to use a logical
;segment named CODE1 as the Physical Code Segment and to use a logical segment named DATA1
        as the Physical Data Segment for the user's program
START:  MOV AX, DATA1 ;To initialize the Data Segment register DS. Segment Registers cannot be
        MOV DS, AX    ;directly loaded with an immediate number. The number is to be loaded
                      ; first to a general purpose register and then from that register
                      ;to the Segment register. DS holds the upper 16-bit of the Base Address
                      ;(ie, the upper 16-bit of the Starting address of the physical Data Segment)
        MOV CX, 0000H ; Initializes the Carry holding register
        MOV AX, NUM1
        ADD AX, NUM2
        JNC L1
        INC CX
L1:     MOV SUM, AX
        MOV SUM+2,CX

        MOV AH, 4CH   ;DOS function Call to terminate the execution of the user program and to exit
        INT 21H       ;to DOS command prompt

CODE1   ENDS
        END START     ; END directive indicates end of assembly.
```

1234 H +
F562 H
0001 0796 H (32-bit sum)

eg. ADD AX, NUM2
(AX) ← (AX) + NUM2 (symbolic illustration)

Syntax of ADD instruction: **ADD <destination>, <source>**
where destination is a register or a memory location specified in one of 24 different ways and source is an immediate number or a register or a memory location specified in one of 24 different ways. Destination and Source cannot be both memory locations in the same instruction.

Syntax of MOV instruction: **MOV <destination>, <source>**
where destination is a register or a memory location specified in one of 24 different ways and source is an immediate number or a register or a memory location specified in one of 24 different ways. Destination and Source cannot be both memory locations in the same instruction.

8086 FLAGS

FLAGS	Name	SET symbol (flag value= 1)	CLEAR (RESET) symbol (flag value= 0)
OF	Overflow Flag	OV (overflow)	NV (not overflow)
DF	Direction Flag	DN (down)	UP
IF	Interrupt Flag	EI (enable interrupt)	DI (disable interrupt)
TF	Trap Flag	-	-
SF	Sign Flag	NG (negative)	PL (positive)
ZF	Zero Flag	ZR (zero)	NZ (not zero)
AF	Auxiliary Carry Flag	AC	NA
PF	Parity Flag	PE	PO
CF	Carry Flag	CY (carry)	NC (no carry)

NOTE:

Augend +	Minuend -
Addend	Subtrahend
Sum	Difference

24 Different ways of specifying Memory Locations

Only, index registers **SI** and **DI** and base registers **BX** and **BP** can be used as memory pointers.

Column 1	Column 2	Column 3
[BX]+[SI]	Column 1 + d8	Column 1 + d16
[BX]+[DI]	Column 1 + d8	Column 1 + d16
[BP]+[SI]	Column 1 + d8	Column 1 + d16
[BP]+[DI]	Column 1 + d8	Column 1 + d16
[SI]	Column 1 + d8	Column 1 + d16
[DI]	Column 1 + d8	Column 1 + d16
[d16]	[BP] + d8	[BP] + d16
[BX]	Column 1 + d8	Column 1 + d16

Note1: No stack segment is defined in the above program and is left out because DOS automatically allocates a 128-byte stack for all programs. The Linker program indicates a warning message saying that no stack is present. The instruction that uses stack in the program is INT 21H which Calls a procedure in DOS. This warning can be ignored because the Stack memory used here is fewer than 128 bytes.

Note2: **Signed 8-bit numbers and Overflow Flag:**
D7 is the sign bit. D7 is zero for positive numbers and D7 is 1 for negative numbers. The magnitude of negative number is represented in 2's complement form (ie for D6 to D0 bits). Consider 3 Cases:

Case1:

+96 (dec)
+70 (dec)
+166 (dec)

+60(hex)
+46 (hex)
+A6 (hex)

0 110 0000 (binary)
0 100 0110 (binary)
1 010 0110 (binary)
D7 D0

Here, since D7 is 1, the magnitude of the result (ie, D6 to D0 bits) is in 2's complement form. ie, D6 to D0 namely, 010 0110 is in 2's complement form. To get the answer, take 2's complement of the magnitude. We get 101 1010 . ie 5A or 90 dec. That is, the answer is -90 dec which is Incorrect. Therefore the overflow flag OV is Set to 1.

Case2:

-128 (dec)
-2 (dec)
-130 (dec)

-80(hex)
- 2 (hex)
-82 (hex)

1 000 0000 (binary)
0 000 0010 (binary)
1 000 0010 (binary)
D7 D0

Here, since D7 is 1, the magnitude of the result (ie, D6 to D0 bits) is in 2's complement form. ie, D6 to D0 namely, 000 0010 is in 2's complement form. To get the answer, take 2's complement of the magnitude. We get 000 0011 . ie 03 or 3 dec. That is, the answer is -3 dec which is Incorrect. Therefore the overflow flag OV is Set to 1.

Case3:

-2 (dec)
-5 (dec)
-7 (dec)

-2(hex)
-5(hex)
-7 (hex)

1 000 0000 (binary)
0 000 0010 (binary)
1 000 0010 (binary)
D7 D0

Here, since D7 is 1, the magnitude of the result (ie, D6 to D0 bits) is in 2's complement form. ie, D6 to D0 namely, 000 0010 is in 2's complement form. To get the answer, take 2's complement of the magnitude. We get 000 0011 . ie 03 or 3 dec. That is, the answer is -3 dec which is Incorrect. Therefore the overflow flag OV is Set to 1.

Question: When is the Oveflow Flag Set to 1? It will be Set to 1, if there is a Carry from D6 to D7 or from D7, but not from both.

Contents of the List file **add.lst**

```
Microsoft (R) Macro Assembler Version 6.11                01/14/10 15:51:40
add.asm                                                   Page 1 - 1

0000                        DATA1       SEGMENT
0000 1234                           NUM1  DW 1234H
0002 F562                           NUM2  DW 0F562H
0004 0002 [                         SUM   DW 2 DUP(0)
     0000
     ]
0008                        DATA1       ENDS

0000                        CODE1       SEGMENT
                                        ASSUME CS:CODE1, DS:DATA1

0000 B8 ---- R              START:  MOV AX, DATA1
0003 8E D8                          MOV DS, AX

0005 B9 0000                        MOV CX, 0000H
0008 A1 0000 R                  MOV AX, NUM1
000B 03 06 0002 R                   ADD AX, NUM2
000F 73 01                          JNC L1
0011 41                             INC CX
0012 A3 0004 R              L1:     MOV SUM, AX
0015 89 0E 0006 R                   MOV SUM+2, CX

0019 B4 4C                          MOV AH, 4CH
001B CD 21                          INT 21H

001D                        CODE1       ENDS
                                        END START

Microsoft (R) Macro Assembler Version 6.11                01/14/10 15:51:40
add.asm                                                   Symbols 2 - 1
```

'R' stands for Re-locatable address which Linker program must resolve.

Segments and Groups:

Name	Size	Length	Align	Combine Class
CODE1	16 Bit	001D	Para	Private
DATA1	16 Bit	0008	Para	Private

Symbols:

Name	Type	Value	Attr
L1	L Near	0012	CODE1
NUM1	Word	0000	DATA1
NUM2	Word	0002	DATA1
START	L Near	0000	CODE1
SUM	Word	0004	DATA1

0 Warnings
0 Errors

(2) To subtract two 16 bit unsigned numbers stored in memory and to get the 32 bit difference in memory, the high order word being a sign indicator

;sub.asm

```
DATA1 SEGMENT
      NUM1   DW   1243H
      NUM2   DW   0F231H
      DIFF   DW   2 DUP(0)
DATA1 ENDS


CODE1 SEGMENT
      ASSUME      CS:CODE1, DS:DATA1
START: MOV AX, DATA1
      MOV DS, AX
      MOV CX, 0000H; CX holds borrow
      MOV AX, NUM1
      SUB AX, NUM2
      JNC L1
      INC CX
      NEG AX  ; it takes the 2's complement of the difference, if the result if negative.
 L1:  MOV DIFF, AX
      MOV DIFF+2, CX

      MOV AH, 4CH
      INT 21H

CODE1 ENDS
      END START
```

Syntax: SUB <destination>, <source>

(a reg) or (a mem. loc.) — (an immediate number) or (a reg) or (a mem. loc.)

Destination and Source cannot be both memory locations in the same instruction.

SUB AX, NUM2
(AX) ← (AX) - NUM2

```
        1243 H -
        F231 H
-------------------
        2012 H
```

(the result is a negative number in 2's complement form - taking 2's complement of the result gives the actual answer – 2's complement of 2012 gives DFEE – the borrow flag (ie, carry flag) = 1, is a sign indicator. The actual answer can also be obtained by doing the subtraction 'Larger number' minus 'Smaller number' and prefixing a minus sign. ie, F231 – 1243 → – DFEE

;(3) To multiply two 16 bit unsigned numbers stored in memory and to get the 32 bit product in memory

;mul.asm

```
DATA1 SEGMENT
      MULTIPLICAND  DW   204AH
      MULTIPLIER    DW   3B2AH
      PRODUCT       DW   2 DUP(0)
DATA1 ENDS


CODE1 SEGMENT
      ASSUME  CS:CODE1, DS:DATA1
START: MOV AX, DATA1
      MOV DS, AX
      MOV AX, MULTIPLICAND
      MUL MULTIPLIER
      MOV PRODUCT, AX
      MOV PRODUCT+2, DX

      MOV AH, 4CH
      INT 21H

CODE1 ENDS
      END START
```

Syntax: MUL <source> {byte×byte or word×word multiplication}

(Source is the multiplier which is a byte or a word in a reg) or (is the multiplier which is a byte or a word in a mem. Loc.)

Product ← Multiplicand × Multiplier
For Byte × Byte multiplication, the Multiplicand is in AL and the product is in AX by dafult. For Word × Word multiplication, the Multiplicand is in AX and the product is in DX AX by default.
Eg,

```
        204A H ×
        3B2A H
-------------------
      0776 5A24 H
```

;(4)word by byte division

```
DATA            SEGMENT
     DIVIDEND   DW 75A5H
      DIVISOR   DB 7FH
    QUOTIENT    DB 1 DUP(0)
   REMAINDER    DB 1 DUP(0)
DATA            ENDS

CODE            SEGMENT
                ASSUME CS:CODE, DS:DATA
        START:  MOV AX, DATA
                MOV DS, AX

                MOV AX, DIVIDEND
                DIV DIVISOR

                MOV QUOTIENT, AL
                MOV REMAINDER, AH

                MOV AH, 4CH
                INT 21H

CODE            ENDS
                END START
```

Syntax: DIV <source> {word+byte or doubleword+word division is performed)

(source is the divisor which is a byte or a word in a reg)
or
(a byte or a word in a mem. Loc.)

Word in AX (dividend) / Byte in Source (divisor) → Quotient in AL and Remainder in AH

DoubleWord in DX AX (dividend) / Word in Source (divisor) → Quotient in AX and Remainder in DX

Example (word by byte division):
75A5 H

7F H

Q=ED H =(AL)
R=12 H = (AH)

;(5) double word by word division

```
DATA            SEGMENT
     DIVIDEND   DD 0CF7564CH
      DIVISOR   DW 3567H
    QUOTIENT    DW 1 DUP(0)
   REMAINDER    DW 1 DUP(0)
DATA            ENDS

CODE            SEGMENT
                ASSUME CS:CODE, DS:DATA
        START:  MOV AX, DATA
                MOV DS, AX

                MOV AX, WORD PTR DIVIDEND ;PTR will override the type declaration specified earlier
                MOV DX, WORD PTR DIVIDEND+2
                DIV DIVISOR

                MOV QUOTIENT, AX
                MOV REMAINDER, DX

                MOV AH, 4CH
                INT 21H

CODE            ENDS
                END START
```

Example (Doubleword by Word division):
0CF7564C H

3567 H

Q = 3E28 H = (AX)
R = 0C34 H = (DX)

;(6) To display the message 'Welcome to MBCET, Thiruvananthapuram' on the display monitor
;disp.asm

```
DATA1 SEGMENT
      MSG DB 'Welcome to MBCET, ThiruvananthapuramS'      ; here $ functions as a string terminator
DATA1 ENDS

CODE1 SEGMENT
      ASSUME CS:CODE1, DS:DATA1
START: MOV AX, DATA1
      MOV DS, AX

      MOV AH, 09H
      MOV DX, OFFSET MSG    ; DOS function call for displaying a string on the screen. AH reg. is to be preloaded
      INT 21H               ;with 09H and DX reg is to be preloaded with the offset address of the First Byte of
                            ;the string

      MOV AH, 4CH           ; DOS function Call to terminate the execution of the user program and to exit
      INT 21H               ;to DOS command prompt

CODE1 ENDS
      END START     ;Note: Here, to display the string, run the .exe file as a command in the DOS command prompt.
```

;(7) To move a string from a source location to a destination location, skipping over 100 bytes of memory locations
;movestr.asm

```
DATA1           SEGMENT
      SRC_STR   DB 'Mar Baselios Engineering College'
      STR_LENGTH EQU ($ - SRC_STR)
                DB 100 DUP(?) ;here a name can be
                              ;given such as SKIP_LOC
      DST_STR   DB STR_LENGTH DUP(0)
DATA1           ENDS

CODE1 SEGMENT
      ASSUME CS:CODE1, DS:DATA1, ES:DATA1
START: MOV AX, DATA1
      MOV DS, AX
      MOV ES, AX
      LEA SI, SRC_STR
      LEA DI, DST_STR
      MOV CX, STR_LENGTH
      CLD
 REP MOVSB ; or (REP MOVS DST_STR, SRC_STR)

      MOV AH, 4CH
      INT 21H

CODE1 ENDS
      END START
```

Note: Redo the above program with 'MOVS' instruction instead of 'MOVSB'
ie, substitute 'REP MOVSB' by 'REP MOVS DST_STR, SRC_STR'

Also instead of 'LEA SI, SRC_STR', we can write 'MOV SI, OFFSET SRC_STR'
and similarly instead of 'LEA DI, DST_STR', we can write 'MOV DI, OFFSET DST_STR'

Location Counter:
Here, $ is acting as a location counter. When assembler reads through the source code for a program, it uses a 'location counter' to keep track of the offset of each data item in a segment. The character $ is used to symbolically represent the current value of the location counter at any point. The initial value of the location counter is zero. When the first character (byte) is read, the location counter value becomes 1; when the second character (byte) is read, the value becomes 2 and so on.
Location counter can be used to find the length of a string.

Syntax of MOV instruction related to string:
There are three forms:
(1) MOVS <dest_string>, <src_string>
(2) MOVSB
(3) MOVSW
here, no additional arguments are to be given
For string instructions, the default association of 'segment:offset' is as follows:
DS:SI (for source string)
ES:DI (for destination string)
(refer page-17 for more about the syntax)

; (8) Program for adding a series of 8-bit numbers – addser.asm

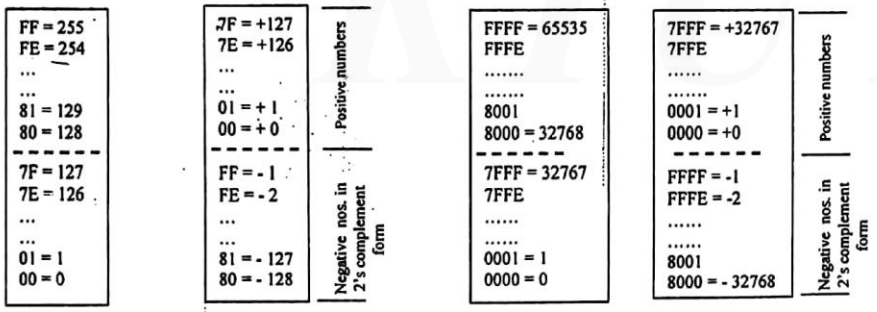
```
DATA    SEGMENT
        LIST   DB    25H, 23H, 12H, 0FFH, 0FFH, 86H
        COUNT  EQU   06H ; or write COUNT EQU ($ - LIST) for automatic computation of count
        SUM    DB    02H DUP(?)
DATA    ENDS

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV   AX, DATA
        MOV   DS, AX
        MOV   CX, COUNT
        MOV   AX, 0000H
        MOV   SI, OFFSET LIST
L2:     ADD   AL, [SI]
        JNC   L1
        INC   AH
L1:     INC   SI
        DEC   CX
        JNZ   L2
        MOV   SUM, AL
        MOV   SUM+1, AH
        MOV   AH, 4CH
        INT   21H
CODE    ENDS
        END   START
```

Registers used as Memory Pointers:
1)Base Registers: BX, BP
2) Index Registers: SI, DI
No other registers can be used for Memory Pointers
(refer p-5 for the 24 different ways of specifying memory locations)

Rewrite the program making use of Loop instruction instead of JNZ

Unsigned and Signed Numbers



Unsigned 8-bit Numbers | Signed 8-bit Numbers | Unsigned 16-bit Numbers | Signed 16-bit Numbers

;(9) to find the largest Signed number - largests.asm

```
DATA    SEGMENT
        LIST   DB    7FH, 0F3H, 80H, 8FH
        COUNT  EQU   04H  ;number of elements
        LARGEST DB   01H DUP(?)
DATA    ENDS

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV AX, DATA
        MOV DS, AX              ;Data Segment register initialised
        MOV SI, OFFSET LIST     ;SI memory pointer initialised to point to the first data location
        MOV CX, COUNT-1         ;counter register initialised to n-1
        MOV AL, [SI]
L2:     CMP AL, [SI+1]          ;compares ith element with (i+1)th element
        JGE L1                  ;Jump if ith element is Greater than or Equal to (i+1)th element
        MOV AL, [SI+1]          ;AL holds always the Greater element
L1:     INC SI
        DEC CX
        JNZ L2
        MOV LARGEST, AL         ;result stored in memory location named LARGEST
        MOV AH, 4CH             ;DOS function call invoked for exiting to DOS
        INT 21H
CODE    ENDS
        END START
```

While comparing **Signed** numbers using 'CMP' instructions, use the following 'Jump' instructions for making decision for branching:
JGE – Jump if First no. is **Greater than or Equal** to Second no.
JLE – Jump if First no. is **Less than or Equal** to Second no.
Here, 'Greater Than' means more positive and 'Less Than' means less positive

Rewrite the program making use of Loop instruction instead of JNZ

Syntax: CMP <destination>, <source>
- <destination>: (a reg) or (a mem. loc.)
- <source>: (an immediate number) or (a reg) or (a mem. loc.)

Destination and Source cannot be both memory locations in the same instruction.

;(10) to find the smallest Signed number - smallets.asm

```
DATA    SEGMENT
        LIST   DB    0FFH, 0F2H, 85H, 62H
        COUNT  EQU   04H
        SMALLEST DB  01H DUP(?)
DATA    ENDS

CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV AX, DATA
        MOV DS, AX
        MOV SI, OFFSET LIST
        MOV CX, COUNT-1
        MOV AL, [SI]
L2:     CMP AL, [SI+1]
        JLE L1          ;Jump if ith element is Less than or Equal to (i+1)th element
        MOV AL, [SI+1]  ;AL holds always the Smaller element
L1:     INC SI
        DEC CX
        JNZ L2
        MOV SMALLEST, AL  ;result stored in memory location named SMALLEST
        MOV AH, 4CH
        INT 21H
CODE    ENDS
        END START
```

; (11) to find the largest Unsigned number - largestu.asm

```
DATA        SEGMENT
      LIST  DB    00H, 7FH, 80H, 0F3H
     COUNT  EQU   04H
   LARGEST  DB    01H  DUP(?)
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
    START:  MOV AX, DATA
            MOV DS, AX
            MOV SI, OFFSET LIST
            MOV CX, COUNT-1
            MOV AL, [SI]
       L2:  CMP AL, [SI+1]
            JAE L1          ;Jump if the ith element is Above or Equal to the (i+1)th element
                            ;It is the same as JNC
            MOV AL, [SI+1] ; AL holds always the Greater element
       L1:  INC SI
            DEC CX
            JNZ L2
            MOV LARGEST, AL
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

While comparing Unsigned numbers using 'CMP' instructions, use the following 'Jump' instructions for making decision for branching:
JAE – Jump if First no. is Above or Equal to Second no.
JBE – Jump if First no. is Below or Equal to Second no.
Here, 'Above' means larger in magnitude and 'Below' means smaller in magnitude.

Rewrite the program making use of Loop instruction instead of JNZ

;(12) to find the smallest Unsigned number - smalletu.asm

```
DATA        SEGMENT
      LIST  DB    0FFH, 0F2H, 80H, 90H
     COUNT  EQU   04H
  SMALLEST  DB    01H  DUP(?)
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE,  DS:DATA
    START:  MOV AX, DATA
            MOV DS, AX
            MOV SI, OFFSET LIST
            MOV CX, COUNT-1
            MOV AL, [SI]
       L2:  CMP AL, [SI+1]
            JBE L1      ;JBE is almost same as JC- both will give same result-but JBE is better
                        ; Jump if the ith element is Below or Equal to the (i+1)th element
            MOV AL, [SI+1] ; AL holds always the Smaller element
       L1:  INC SI
            DEC CX
            JNZ L2
            MOV SMALLEST, AL
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```



```
            MOV SI, OFFSET LIST
       L2:  MOV AL, [SI]       ; because CMP [SI], [SI+1] is invalid (two mem. Locations)
            CMP AL, [SI+1]
            JAE L1             ;Jump if the ith no. is Above or Equal to the (i+1)th no.
            XCHG AL, [SI+1]    ; because XCHG [SI], [SI+1] is invalid (two mem. Locations)
            MOV [SI], AL
       L1:  ADD SI, 01  ;or use INC SI
            DEC CX
            JNZ L2
            DEC BX
            JNZ L3

            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Syntax; XCHG <destination>, <source>
(a reg) or (a mem. loc.) — (a reg) or (a mem. loc.)
Destination and Source cannot be both memory locations in the same instruction.

;(15) to sort 8-bit unsigned nos in ascending order - ascendu.asm

```
DATA        SEGMENT
      LIST  DB    7FH, 00H, 0FFH, 80H
     COUNT  EQU   04H
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
    START:  MOV AX, DATA
            MOV DS, AX

            MOV BX, COUNT-1   ;BX is the Pass counter
       L3:  MOV CX, BX        ;CX is the Comparison counter which is initialised to the Pass counter
                              ; value at the beginning of each Pass
            MOV SI, OFFSET LIST
       L2:  MOV AL, [SI]
            CMP AL, [SI+1]
            JBE L1            ;Jump if the ith no.is Below or Equal to the (i+1)th no.
            XCHG AL, [SI+1]
            MOV [SI], AL
       L1:  ADD SI, 01     ;or use INC SI
            DEC CX
            JNZ L2
            DEC BX
            JNZ L3

            MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

;(16) to sort 8-bit signed nos in descending order - descends.asm

```
DATA        SEGMENT
      LIST  DB    80H, 0FFH, 00H, 7FH
     COUNT  EQU   04H
DATA        ENDS
```

(13) to find out the number of even and odd numbers from a given set of 16-bit hex nos.- evenodd.asm

```
DATA        SEGMENT
      LIST  DW    1234H, 0FCD7H, 8526H, 6278H
     COUNT  EQU   04H
   EVENNOS  DB    01H DUP (?)
    ODDNOS  DB    01H DUP (?)
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
     START: MOV AX, DATA
            MOV DS, AX
            MOV BL, 00H    ; (or Write XOR BL,BL)   -BL stores no. of even nos.
            MOV DL, 00H    ; (or Write XOR DL,DL)   -DL stores no. of odd nos.
            MOV SI, OFFSET LIST  ; or  LEA  SI, LIST
            MOV CX, COUNT

     AGAIN: MOV AX, [SI]
            ROR AX, 01
            JC  ODD
            INC BL
            JMP NEXT
     ODD:   INC DL
     NEXT:  ADD SI, 02H ; (SI) ← (SI) + 02  {is equivalent to INC SI;  INC SI}
            DEC CX
            JNZ AGAIN

            MOV EVENNOS, BL ;result stored in memory location named EVENNOS
            MOV ODDNOS,  DL ;result stored in memory location named ODDNOS
            MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

The simplest logic to decide whether a binary number is even or odd, is to check the LSB of the number. If the LSB is zero, the number is even; if it is one, the number is odd. Check the LSB by rotating the number through Carry flag and increment the 'Even' or 'Odd' counter accordingly.

Syntax: XOR <destination>, <source>

<destination>	<source>	
(a reg) or (a mem. loc.)	(an immediate number) or (a reg) or (a mem. loc.)	Destination and Source cannot be both memory locations in the same instruction.

Syntax: ROR <destination>, <count>

<destination>	<count>
(a reg) or (a mem. loc.)	1 or CL reg content. For count > 1, load CL reg. eg. MOV CL, 05H

This gives 8-bit rotation for byte or 16 bit rotation for word. To rotate more than one bit position, load the desired number in the CL register.

---

(14) to sort 8-bit unsigned nos in descending order - descendu.asm

```
DATA        SEGMENT
      LIST  DB    80H, 0FFH, 00H, 7FH
     COUNT  EQU   04H   ; or  COUNT EQU ($ - LIST)
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
     START: MOV AX, DATA
            MOV DS, AX
            MOV BX, COUNT-1   ;BX is the Pass counter
     L3:    MOV CX, BX        ;CX is the Comparison counter which is initialised to the Pass counter
                              ;value at the beginning of each Pass
```



```
CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
     START: MOV AX, DATA
            MOV DS, AX

            MOV BX, COUNT-1   ;BX is the Pass counter
     L3:    MOV CX, BX        ;CX is the Comparison counter which is initialised to the Pass counter
                              ; value at the beginning of each Pass
            MOV SI, OFFSET LIST
     L2:    MOV AL, [SI]
            CMP AL, [SI+1]
            JGE L1            ;Jump if the ith no.is Greater than or Equal to the (i+1)th no.
            XCHG  AL, [SI+1]
            MOV   [SI], AL
     L1:    ADD SI, 01        ;or use INC SI
            DEC CX
            JNZ  L2
            DEC BX
            JNZ L3

            MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

;(17) to sort 8-bit signed nos in ascending order - ascends.asm

```
DATA        SEGMENT
      LIST  DB    7FH, 00H, 0FFH, 80H
     COUNT  EQU   04H
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
     START: MOV AX, DATA
            MOV DS, AX
            MOV BX, COUNT-1   ;BX is the Pass counter
     L3:    MOV CX, BX        ;CX is the Comparison counter which is initialised to the Pass counter
                              ;value at the beginning of each Pass
            MOV SI, OFFSET LIST
     L2:    MOV AL, [SI]
            CMP AL, [SI+1]
            JLE L1            ;Jump if the ith no.is Less than or Equal to the (i+1)th no.
            XCHG  AL, [SI+1]
            MOV   [SI], AL
     L1:    ADD SI, 01        ;or use INC SI
            DEC CX
            JNZ  L2
            DEC BX
            JNZ L3

            MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

;(18) addition of two 64-bit nos – 64bitadd.asm (for multibyte addition use ADC instead of ADD – think why?)

```
DATA        SEGMENT
   NUM1     DB  0EFH, 0CDH, 0ABH, 89H, 67H, 45H, 23H, 0F1H
   NUM2     DB  01H, 23H, 45H, 67H, 89H, 0ABH, 0CDH, 0EFH
   NUM3     DB  09H DUP(00) ;to store the result
  COUNT     EQU 08H
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
   START:   MOV AX, DATA
            MOV DS, AX

            MOV CX, COUNT
            MOV SI, OFFSET NUM1
            MOV DI, OFFSET NUM2
            MOV BX, OFFSET NUM3
            XOR AX, AX ;AX is initialized to Zero and Carry flag is Reset – MOV AX, 0000H will not
                       ;reset Carry Flag – if MOV AX, 0000H is used, CLC instruction can be used for
   NEXT:    MOV AL, [SI]    ; resetting carry flag
            ADC AL, [DI]
            MOV [BX], AL
            INC SI
            INC DI
            INC BX
            DEC CX
            JNZ NEXT
            JNC NOCARRY
            MOV [BX], 01
NOCARRY:    MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

;(19) addition of two 3×3 matrices - matadd.asm

```
DATA        SEGMENT
   MAT1     DB 0FFH, 02H, 03H, 04H, 05H, 06H, 07H,08H, 09H
   MAT2     DB 0FFH, 02H, 03H, 04H, 05H, 06H, 07H, 08H, 09H
   MAT3     DW 09H DUP(00) ; to store the result matrix – byte addition may give word type value
                           ; and hence DW is used
   COUNT    EQU 09H
DATA        ENDS


CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
   START:   MOV AX, DATA
            MOV DS, AX

            MOV CX, COUNT
            MOV SI, OFFSET MAT1
            MOV DI, OFFSET MAT2
            MOV BX, OFFSET MAT3
   NEXT:    XOR AX, AX  ; AX is initialized to zero and Carry flag is reset – MOV AX, 0000H can also
                        ;be used here since we need not have to consider Carry as we are using ADD
```



```
                        ; instead of ADC
            MOV AL, [SI]
            ADD AL, [DI]  ;since it is not a multibyte addition, ADD can be used instead of ADC
            JNC L1
            INC AH
   L1:      MOV [BX], AX
            INC SI
            INC DI
            ADD BX, 02    ; add 02 since the result matrix elements are of type word – or use INC BX
                          ;twice – CY is automatically taken care of here since the result
                          ;elements are of word type
            DEC CX
            JNZ NEXT

            MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

String Instructions

A string is a series of bytes or words stored in successive memory locations. Often a string consists of a series of ASCII character codes.

1) **Move String Instructions**

MOVS/MOVSB/MOVSW

These are used for moving (copying) a byte or a word from a location in the data segment to a location in the extra segment.
There are three forms for the syntax:

a) MOVS <dst_str>, <src_str> ; dst_str can be a reg or a mem.loc and src_str can be a reg or mem.loc or an immediate number (Can both be memory locations? Can both be registers? – verify)
b) MOVSB
c) MOVSW } - No operands specified here

The offset address of the source byte or word in the **data-segment** must be in the **SI** Register.
(ie, SI should point to a byte or word in the Data Segment)
The offset address of the destination location in the **extra-segment** must be contained in the **DI** Register.
(ie, DI should point to a byte or word in the Extra Segment)
For multiple-byte or multiple-word moves, the number of elements to be moved is put in the **CX** register and **CX** acts as a down counter.

After a byte or a word is moved, SI and DI are automatically adjusted to point to the next source and next destination locations (ie, it can be either **auto-increment** or **auto-decrement**, depending on the value of the Direction Flag, DF)

If the Direction Flag is reset (= 0) using CLD instruction, then SI and DI will be auto-incremented by 1 for byte movement and by 2 for word movement.
If the Direction Flag is set (= 1) using STD instruction, then SI and DI will be auto-decremented by 1 for byte movement and by 2 for word movement.

```
DATA        SEGMENT
   PWDSTR     DB 'ABCD'                ;source string (stored password)
 PWDSTR_LEN   EQU ($-PWDSTR)           ;length of stored password string
  INPUTSTR    DB 'ABCD'                ;destination string (user input password)
INPUTSTR_LEN  EQU ($-INPUTSTR)         ; length of user input  password string
  CORRECT     DB 'Password is CORRECT$'
  INCORRECT   DB 'Password is INCORRECT$'
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE,  DS:DATA,   ES:DATA
    START:  MOV AX, DATA
            MOV DS, AX
            MOV ES, AX
            MOV AX, PWDSTR_LEN
            CMP AX, INPUTSTR_LEN
            JNZ L1          ; same as JNE
            LEA SI, PWDSTR
            LEA DI, INPUTSTR
            MOV CX, PWDSTR_LEN
            CLD
            REPE CMPSB   ; or (REPE CMPS INPUTSTR,  PWDSTR)
            JNZ L1          ;this checks the zero flag – same as JNE

            DISP   CORRECT
            JMP L2
      L1:   DISP   INCORRECT

      L2:   MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

3) **Scan String Instructions (Scan or search a string for a Byte or a word)**

SCAS / SCASB / SCASW

There are three forms for the syntax:

a) SCAS <dst_str>

b) SCASB
c) SCASW } - No operands specified here

When a match is found (ie, when the two elements are the same) Zero flag is SET

These are used for searching (scanning) through a string to find if it contains a specified string byte or string word. This compares a byte in 'AL' or a word in 'AX' with a byte or word pointed to by DI in Extra Segment. The string to be scanned must be in Extra Segment, and DI must contain the offset of the byte or word to be compared. The length of the string must be stored in CX.
If the Direction Flag is reset (= 0) using CLD instruction, then DI will be auto-incremented by 1 for byte comparison and by 2 for word comparison.

If the Direction Flag is set (= 1) using STD instruction, then DI will be auto-decremented by 1 for byte comparison and by 2 for word comparison.
This instruction is often used with a repeat prefix REPNE (Repeat If Not Equal to – meaning, repeat if the First element is not equal to the Second element) to find the occurrence of a specified byte (in AL) or a specified word (in AX) in a given string of characters. REPNE is same as REPNZ.
This prefix will cause the string instruction to be repeated if 'REPNE' is True and if 'CX' has not counted down to zero. (After each repeated execution, CX is automatically decremented).
In other words, the repeated execution stops when 'REPNE' is False or CX = 0

Usage Example

```
MOV AX, DATA    ; DATA is the name of the Data Segment
MOV DS, AX
MOV  ES, AX ;here Data Segment and Extra Segment are initialized with the same address -
   ;if an Extra Segment is defined, ES can be initialized with Extra Segment base address.
MOV  DI, OFFSET  DST_STR   (or LEA  DI,  DST_STR)
MOV AL, 0DH; Byte to the scanned for in AL
CLD
MOV CX, 80  ;length of the string is stored in CX
REPNE SCASB    ; or   REPNE  SCAS DST_STR
```

(22) Programming Example:

; occur1.asm--to find the number of occurrences of a given character in a given string of ; characters and to display the result in memory or register

```
DATA        SEGMENT
   CHAR     DB   'e'
   STRG     DB   'Mar Baselios College of Engineering'
   STRGLEN  EQU   ($-STRG)
DATA        ENDS

CODE        SEGMENT
            ASSUME  CS:CODE,  DS:DATA, ES:DATA
    START:  MOV AX, DATA
            MOV DS, AX
            MOV ES, AX ;Data segment and Extra segment are initialized to the same base address
            MOV CX, STRGLEN      ;counter CX initialised with the string length
            INC CX          ;this is required to take account of a matching character at the end of the
                            ;string  ie, (n+1)
            MOV DI, OFFSET STRG     ;DI pointer points to the first character in the string
            CLD
            XOR BX, BX      ;or(MOV BX, 0) -- BX stores the no. of occurrences-it is initialized
                            ; to 0
            MOV AL, CHAR
    AGAIN:  REPNE SCASB ; or (REPNE  SCAS  STRG) - DI is auto-incremented and CX is
                            ;auto-decremented after each repetition
            JCXZ  L1        ;jump if CX register is zero - this instruction does not check zero flag –
                    ; exits when end of String has reached (see note below)

            INC BX          ;BX holds the number of occurrences of the given character
            JMP AGAIN
      L1:   MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

Note: We cannot use JZ L1, because when REPNE SCASB is false (ie, the first operand in AL becomes equal to second operand) the Zero flag will be Set and the program will exit to DOS.

(23) to find the number of occurrence of a given character in a string of characters (without using Scan string instruction but only Compare instruction)

; occur2.asm

```
DATA        SEGMENT
   CHAR     DB  'e'
  STRNG     DB  'Mar Baselios'
  STRLN     EQU  ($-STRNG)   (here Extra segment is not required)
DATA        ENDS

CODE        SEGMENT
            ASSUME   CS:CODE,  DS:DATA
   START:   MOV AX, DATA
            MOV DS, AX
            MOV CX, STRLN ;counter CX initialised to the count value
            MOV AL, CHAR
            XOR BX, BX     ; or  MOV BX, 0
            MOV SI, OFFSET STRNG     ;SI pointer points to the first character in the string
   AGAIN:   CMP AL, [SI]
            JNZ SKIP
            INC BX
   SKIP :   INC SI
            LOOP AGAIN
            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

(24) To find the number of occurrences of a given single byte integer in a given series of integers

; occurint.asm

```
DATA        SEGMENT
      BYTE1 DB     24H
      LIST  DB     43H, 02H, 24H, 03H, 24H, 06H
      LEN   EQU  $-LIST
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
      START:MOV AX, DATA
            MOV DS, AX      ;Data Segment register initialised
            MOV CX, LEN
            MOV AL, BYTE1
            XOR BX, BX      ;or MOV BX, 0000H
            MOV SI, OFFSET LIST
      AGAIN:CMP AL, [SI]
            JNZ SKIP
            INC BX
      SKIP: INC SI
            LOOP AGAIN

            MOV AH, 4CH     ;DOS function call invoked for exiting to DOS
            INT 21H
CODE        ENDS
            END START
```


DOS and BIOS interrupts

Software interrupts are the primary means to access the services of an operating system. The DOS (Disk Operating System) and BIOS (Basic Input Output system) provide a large number of functions to access devices, files, memory and process control services that are available to any program which is capable of setting registers and invoking software interrupts.

DOS and BIOS services for device access such as reading from Keyboard, writing to screen, disk read, disk write etc, communicates in ASCII format. For example, the numeric Key '1' on keyboard is read as the ASCII code 31H. The numeric value '1' can be displayed on the screen by writing the ASCII code 31H on to the screen. Therefore the ASCII codes read from the keyboard have to be converted to 'numeric data' for processing and back to ASCII form for displaying on the screen. This process is required for numeric data processing.

The operating system service access interface is called 'System Call'. The services provided by INT 21H (Software Interrupts) are called DOS services or DOS function (system) call. DOS interrupts and the corresponding services are listed below.

DOS INTERRUPTS:

DOS INTERRUPTS (SERVICES)	
Interrupt Type	**Service Name**
INT 20H	Program Terminate
INT 21H	DOS System Call (or DOS Services) (or DOS Function Call)
INT 22H	Terminate Handler Pointer (address)
INT 23H	Control Break Handler Pointer
INT 24H	Critical Error Handler Pointer
INT 25H	Absolute Disk Read
INT 26H	Absolute Disk Write
INT 27H	Terminate and Stay Resident (TSR)
INT 28H	DOS Time Slice
INT 2EH	Perform DOS Command
INT 2FH	Multiplex Interrupts

For example, consider DOS System Call (Function Call) INT 21H:

The DOS services INT 21H are categorized into various 'Functions'. The function numbers vary from Function 00H to Function 63H. Some of the functions are:

INT 21H, Function 00H : Program Terminate
INT 21H, Function 01H : Read Character from Standard Input Devices (ie, Keyboard)
INT 21H, Function 02H : Write Character to Standard Output Devices (ie, Display Monitor)
INT 21H, Function 03H : Read Character from Standard Auxiliary Input Devices (ie, Serial Port COM1)

INT 21H, Function 09H : Display String to Standard Output Devices (ie, Display Monitor)

INT 21H, Function 4CH : Terminate execution with Return Code
etc,

The steps involved in accessing DOS Interrupts are:

(i) Load a Function Number in the AH register. If there is a sub-function, its value is stored in AL register. That is, use instructions such as MOV AH, function_num and MOV AL, sub-function_num
(ii) Load other registers, if required, as specified in the DOS service format.
(iii) Prepare Buffers, ASCIIZ (zero terminated string), and control blocks, if necessary.
(iv) Invoke DOS service INT n
(v) Process Response: Look for the error indicators by examining the carry flag and error code register (generally AX) as set by the DOS service Call. The system call communicates results through register parameters.

BIOS INTERRUPTS:

BIOS is the lowest level software in the PC. Even DOS uses BIOS functions to control the hardware. BIOS service functions are supplied by the manufacturer of the hardware device.

BIOS INTERRUPTS (SERVICES)	
Interrupt Type	**Service Name**
INT 10H	Video Services
INT 11H	Machine Configuration
INT 12H	Usable RAM Memory Size
INT 13H	Disk I/O
INT 14H	Serial Port I/O (RS 232 C)
INT 15H	AT Services
INT 16H	Keyboard I/O
INT 17H	Printer I/O

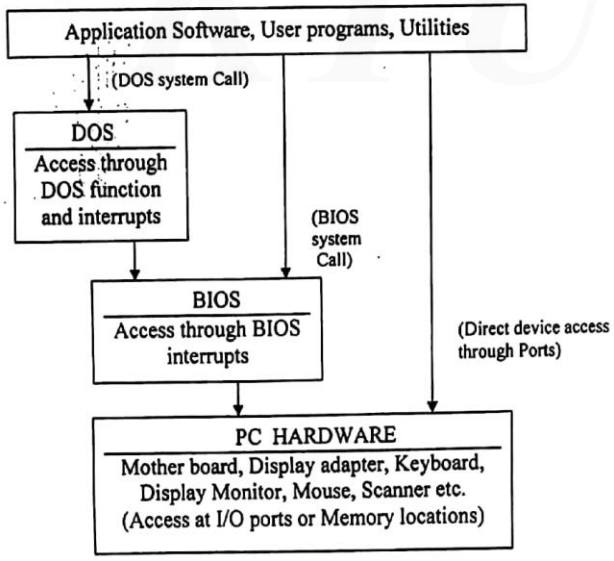
For example, consider BIOS Function Call INT 10H:

The BIOS function call INT 10H are categorized into various 'Functions'. The function numbers vary from Function 00H to Function FFH. Some of the functions are:

INT 10H, Function 00H : Set Video Mode
INT 10H, Function 01H : Set Cursor Shape
INT 10H, Function 02H : Set Cursor Position
INT 10H, Function 03H : Read Cursor Position , etc

The steps involved in accessing BIOS Interrupts are:

(i) Load a Function Number in the AH register. If there is a sub-function, its value is stored in AL register. That is, use instructions such as MOV AH, function_num and MOV AL, sub-function_num
(ii) Load other registers, if required, as specified in the BIOS service format.
(iii) Prepare Buffers, ASCIIZ (zero terminated string), and control blocks, if necessary.
(iv) Invoke BIOS service INT n
(v) Process Response: Look for the error indicators by examining the carry flag and error code register (generally AX) as set by the BIOS service Call. The system call communicates results through register parameters.



SOFTWARE MODEL FOR DEVICE ACCESS

Syntax of Some functions of DOS function Call INT 21H

DOS function Call	Required Preloading of Registers	Action of the Function Call	Usage example
INT 21H, Function 01H	**AH** register is to be preloaded with the Function number **01H.**	This function call Reads an ASCII Character from Standard Input Device (ie, Key Board) with simultaneous echoing of the character on the Screen. The ASCII code of the input Key is returned to **AL** register.	MOV AH, 01H INT 21H

INT 21H, Function 02H	**AH** register is to be preloaded with the Function number **02H.**	This function call Writes an ASCII Character whose ASCII code is stored in **DL** register on to the standard output device-ie., Display monitor (screen)	MOV DL, 'A'
			MOV AH, 02H
			INT 21H

INT 21H, Function 09H	**AH** register is to be preloaded with the Function number **09H** and **DX** register is to be preloaded with the offset address of the starting byte of the string to be displayed. The string should be terminated with a $ sign.	This function call Displays a string terminated with $ sign on to the standard Output device (ie Display monitor)	MOV DX, OFFSET string MOV AH, 09H INT 21H
INT 21H, Function 4CH	**AH** register is to be preloaded with the Function number **4CH.**	This function call terminates execution of the current program and exit to DOS. (It also stores the return code in AL register which can be used to set error level using batch commands).	MOV AH, 4CH INT 21H

(25) DOS function call to Read a character from the keyboard and to echo the character on to the screen, with suitable message string

INT 21H-function 01H (here ASCII code of the input key is returned to AL register)

;int21_01.asm

```
;Macro is defined
DISP_STR    MACRO  MSG      ; 'MSG' is the dummy argument
            MOV  AH, 09H
            MOV DX, OFFSET MSG
            INT 21H
            ENDM
```

Any number of dummy arguments can be used. For example,
MACRO_NAME MACRO <arg1>, <arg2>, ... ,<argn>
MACRO can also be defined without any dummy arguments.

```
DATA        SEGMENT
    MSG1    DB 'Type any key:$'  ;MSG1 is the actual argument
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
    START:  MOV AX, DATA
            MOV DS, AX
            DISP_STR  MSG1    ;here macro is 'Called' - MSG1 is the Actual argument. Actual parameter is
                              ;passed  from Calling program to the Dummy argument in the Macro definition

            MOV AH, 01H  ;This function Reads a Character from Key Board with simultaneous echoing of the character on
            INT 21H      ;the Screen. The ASCII code of the input Key is returned to the  AL register.

            MOV AH, 4CH
            INT 21H

CODE        ENDS
            END START
```

ASCII character	ASCII code (in HEX)
A	41H
a	61H
CR (carriage return)	0DH
LF (Line feed)	0AH
0	30H
9	39H
$	24H
Space /Blank	20H or 2CH

(26) DOS function call to display the ASCII character whose ASCII code is stored in DL register

INT 21H-function 02H

;int21_02.asm

```
DATA        SEGMENT
    MSG     DB 'Hello world - DOS function Call INT 21H, function 02H'
    LEN     EQU ($-MSG)
DATA        ENDS

CODE        SEGMENT
            ASSUME CS:CODE, DS:DATA
    START:  MOV AX, DATA
            MOV DS, AX
            MOV CX, LEN
            MOV SI, OFFSET MSG

            MOV AH, 02H   ;this function call displays the ASCII character whose ASCII code is stored in DL register
NEXT_CHAR:  MOV DL, [SI]  ;The ASCII code pointed by the SI pointer is stored in DL register
            INT 21H

            INC SI
            LOOP  NEXT_CHAR

            MOV AH, 4CH
            INT 21H
CODE        ENDS
            END START
```

END