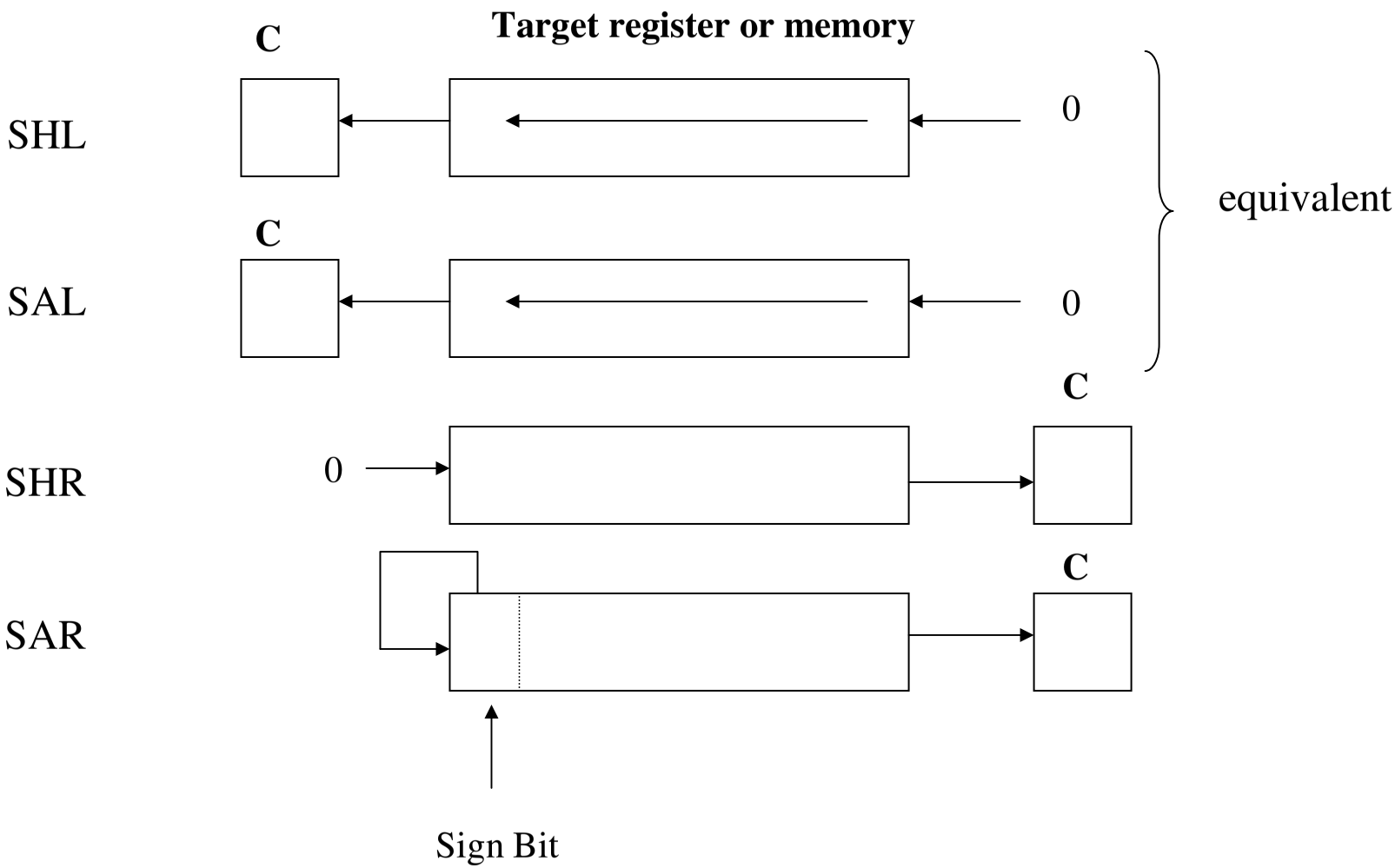
Weeks 6

8088/8086 Microprocessor Programming

Shift


C
Target register or memory
SHL
0
equivalent
C
SAL
0
C
SHR
0
C
SAR
Sign Bit

Examples

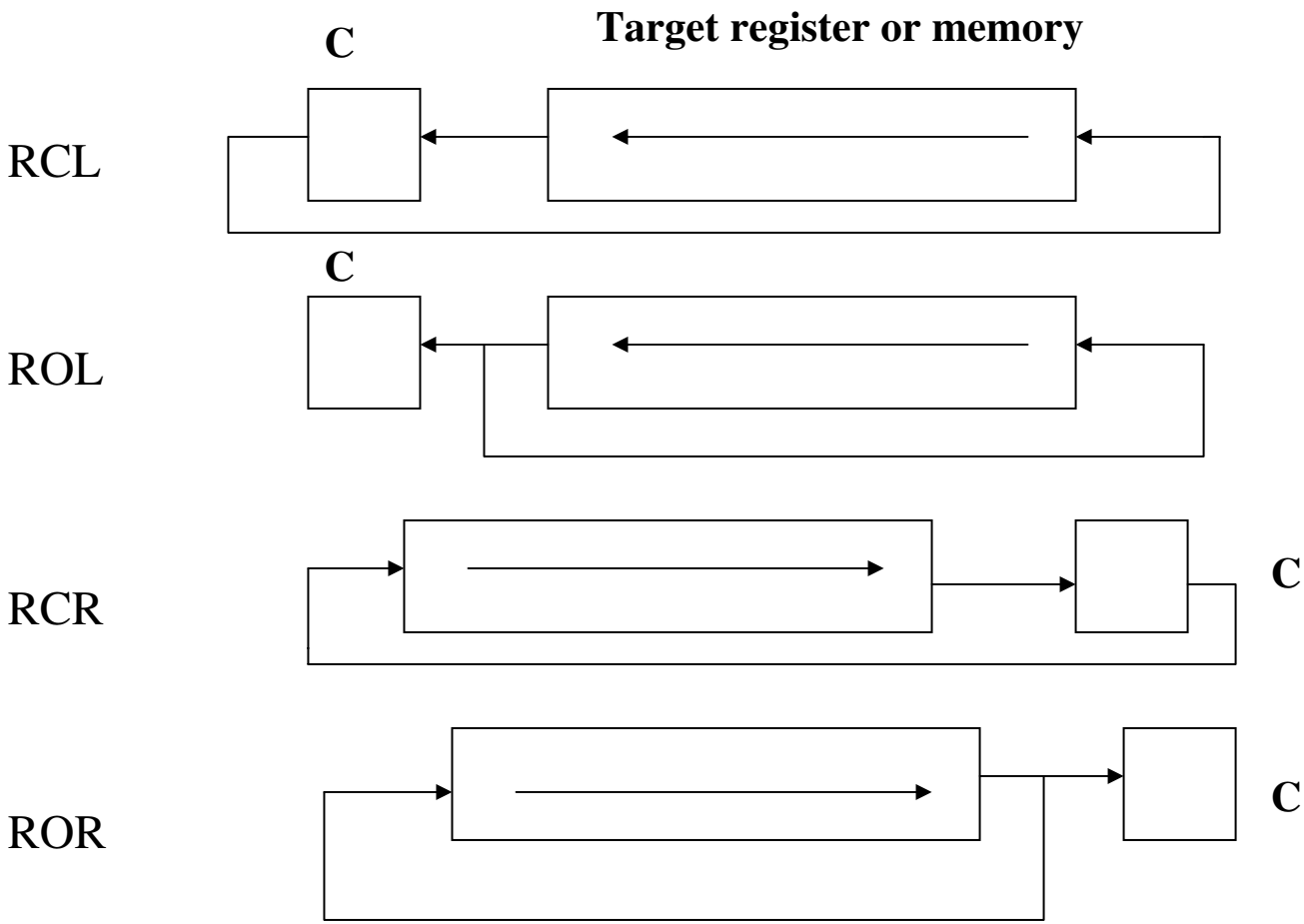
Examples
```
SHL AX,1
SAL DATA1, CL   ; shift count is a modulo-32 count
```

Ex.
```
; Multiply AX by 10
        SHL AX, 1
        MOV BX, AX
        MOV CL,2
        SHL AX,CL
        ADD AX, BX
```

Ex. What are the results of SAR CL, 1 if CL initially contains B6H?

Ex. What are the results of SHL AL, CL if AL contains 75H and CL contains 3?

Rotate



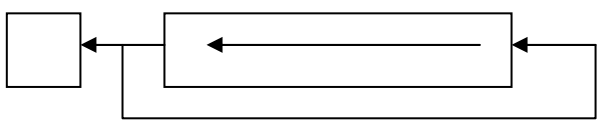
Ex. What is the result of ROL byte ptr [SI], 1 if this memory location 3C020 contains 41H?

What is the result of ROL word ptr [SI], 8 if this memory location 3C020 contains 4125H?

Example

Write a program that counts the number of 1's in a byte and writes it into BL

```
DATA1   DB 97              ; 61h
        SUB    BL,BL       ;clear BL to keep the number of 1s
        MOV    DL,8        ;rotate total of 8 times
        MOV    AL,DATA1
AGAIN:  ROL    AL,1        ;rotate it once
        JNC    NEXT        ;check for 1
        INC    BL          ;if CF=1 then add one to count
NEXT:   DEC    DL          ;go through this 8 times
        JNZ    AGAIN       ;if not finished go back
        NOP
```



BCD and ASCII Numbers

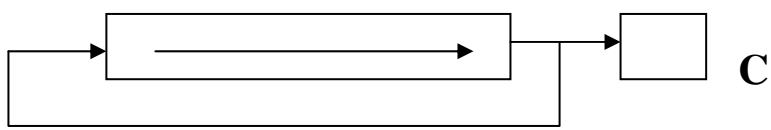
- BCD (Binary Coded Decimal)
  - Unpacked BCD: One byte per digit
  - Packed BCD: 4 bits per digit (more efficient in storing data)
- ASCII to unpacked BCD conversion
  - Keyboards, printers, and monitors all use ASCII.
  - Digits 0 to 9 are represented by ASCII codes 30 – 39.

- Example. Write an 8086 program that displays the packed BCD number in register AL on the system video monitor
  - The first number to be displayed should be the MS Nibble
  - It is found by masking the LS Nibble and then rotating the MS Nibble into the LSD position
  - The result is then converted to ASCII by adding 30h
  - The BIOS video service is then called to display this result.

ASCII Numbers Example

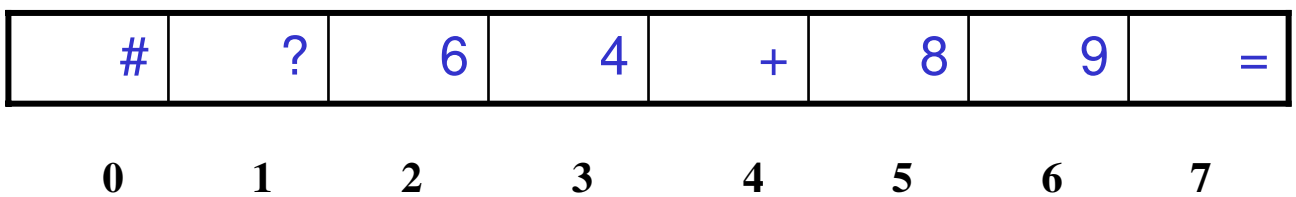
```
MOV BL,AL; save
AND AL,F0H
MOV CL,4
ROR AL,CL
ADD AL,30H
MOV AH,0EH
INT 10H ;display single character

MOV AL,BL; use again
AND AL,0FH
ADD AL,30H
INT 10H
INT 20H          ; RETURN TO DOS
```



Example

- Write an 8086 program that adds two packed BCD numbers input from the keyboard and computes and displays the result on the system video monitor
- Data should be in the form 64+89= The answer 153 should appear in the next line.



Example Continued

```
Mov dx, offset bufferaddress
Mov ah,0a
Mov si,dx
Mov byte ptr [si], 6
Int 21
Mov ah,0eh
Mov al,0ah
Int 10
; BIOS service 0e line feed  position cursor

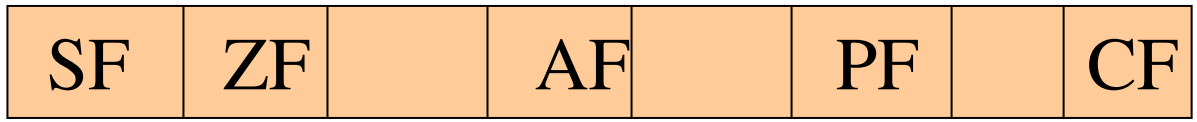
 sub byte ptr[si+2], 30h
 sub byte ptr[si+3], 30h
 sub byte ptr[si+5], 30h
 sub byte ptr[si+6], 30h
```

6	?	6	4	+	8	9	=
0	1	2	3	4	5	6	7

```
Mov cl,4
Rol byte ptr [si+3],cl
Rol byte ptr [si+6],cl
Ror word ptr [si+5], cl
Ror word ptr [si+2], cl

 Mov al, [si+3]
 Add al, [si+6]
 Daa
 Mov bh,al
 Jnc display
 Mov al,1
 Call display
 Mov al,bh
 Call display
 Int 20
```

Flag Control Instructions



- **LAHF** Load AH from flags (AH) ← (Flags)
- **SAHF** Store AH into flags (Flags) ← (AH)
  - Flags affected: SF, ZF, AF, PF, CF

Bulk manipulation of the flags

- **CLC** Clear Carry Flag (CF) ← 0
- **STC** Set Carry Flag (CF) ← 1
- **CLI** Clear Interrupt Flag (IF) ← 0
- **STI** Set interrupt flag (IF) ← 1

Individual manipulation of the flags

- Example (try with debug)

```
LAHF
MOV AX,0000
ADD AX,00
SAHF
```

  - Check the flag changes!

Compare

Mnemonic	Meaning	Format	Operation	Flags Affected
CMP	Compare	CMP D,S	(D) − (S) is used in setting or resetting the flags	CF, AF, OF, PF, SF, ZF

(a)

Unsigned Comparison		
Comp Operands	CF	ZF
Dest > source	0	0
Dest = source	0	1
Dest < source	1	0

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(b)

Signed Comparison		
Comp Operands	ZF	SF,OF
Dest > source	0	SF=OF
Dest = source	1	x
Dest < source	0	SF<>OF

Compare Example

```
DATA1        DW     235Fh
…

MOV AX, CCCCH
CMP AX, DATA1
JNC OVER
SUB AX,AX
OVER: INC DATA1
```

CCCC – 235F = A96D => Z=0, CF=0 =>
CCCC > DATA1

Compare (CMP)

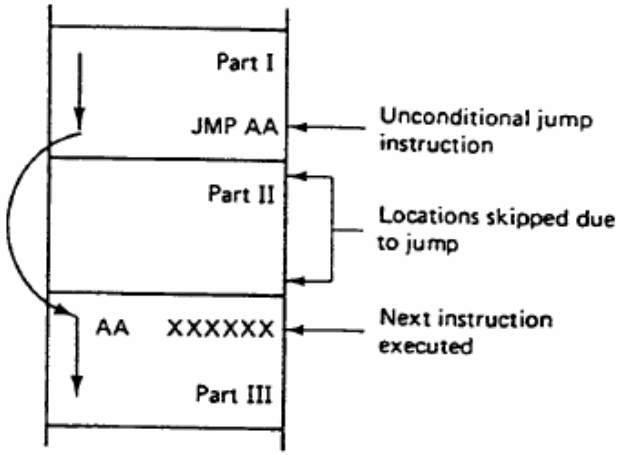
For ex: CMP CL,BL ; CL-BL; no modification on neither operands

Write a program to find the **highest** among 5 grades and write it in **DL**

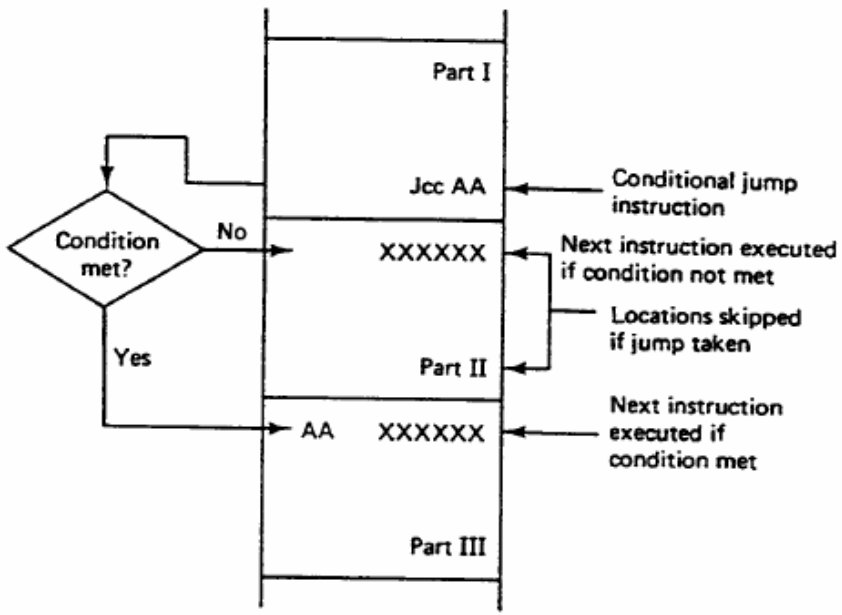
```
DATA    DB       51, 44, 99, 88, 80      ;13h,2ch,63h,58h,50h
        MOV      CX,5                    ;set up loop counter
        MOV      BX, OFFSET DATA         ;BX points to GRADE data
        SUB      AL,AL                   ;AL holds highest grade found so far
AGAIN:  CMP      AL,[BX]                 ;compare next grade to highest
        JA       NEXT                    ;jump if AL still highest
        MOV      AL,[BX]                 ;else AL holds new highest
NEXT:   INC      BX                      ;point to next grade
        LOOP AGAIN                       ;continue search
        MOV  DL, AL
```

Jump Instructions

- Unconditional vs conditional jump



(a)



(b)

Conditional Jump

These flags are based on general comparison

Mnemonic	Description	Flags/Registers
JZ	Jump if ZERO	ZF = 1
JE	Jump if EQUAL	ZF = 1
JNZ	Jump if NOT ZERO	ZF = 0
JNE	Jump if NOT EQUAL	ZF = 0
JC	Jump if CARRY	CF = 1
JNC	Jump if NO CARRY	CF = 0
JCXZ	Jump if CX = 0	CX = 0
JECXZ	Jump if ECX = 0	ECX = 0

Conditonal Jump based on flags

Mnemonic	Description	Flags/Registers
JS	JUMP IF SIGN (NEGATIVE)	SF = 1
JNS	JUMP IF NOT SIGN (POSITIVE)	SF = 0
JP	Jump if PARITY EVEN	PF = 1
JNP	Jump if PARITY ODD	PF = 0
JO	JUMP IF OVERFLOW	OF = 1
JNO	JUMP IF NO OVERFLOW	OF = 0

Jump Based on Unsigned Comparison

These flags are based on unsigned comparison

Mnemonic	Description	Flags/Registers
JA	Jump if above op1>op2	CF = 0 and ZF = 0
JNBE	Jump if not below or equal op1 not <= op2	CF = 0 and ZF = 0
JAE	Jump if above or equal op1>=op2	CF = 0
JNB	Jump if not below op1 not <opp2	CF = 0
JB	Jump if below op1<op2	CF = 1
JNAE	Jump if not above nor equal op1< op2	CF = 1
JBE	Jump if below or equal op1 <= op2	CF = 1 or ZF = 1
JNA	Jump if not above op1 <= op2	CF = 1 or ZF = 1

Jump Based on Signed Comparison

These flags are based on signed comparison

Mnemonic	Description	Flags/Registers
JG	Jump if GREATER op1>op2	SF = OF AND ZF = 0
JNLE	Jump if not LESS THAN or equal op1>op2	SF = OF AND ZF = 0
JGE	Jump if GREATER THAN or equal op1>=op2	SF = OF
JNL	Jump if not LESS THAN op1>=op2	SF = OF
JL	Jump if LESS THAN op1<op2	SF <> OF
JNGE	Jump if not GREATER THAN nor equal op1<op2	SF <> OF
JLE	Jump if LESS THAN or equal op1 <= op2	ZF = 1 OR SF <> OF
JNG	Jump if NOT GREATER THAN op1 <= op2	ZF = 1 OR SF <> OF

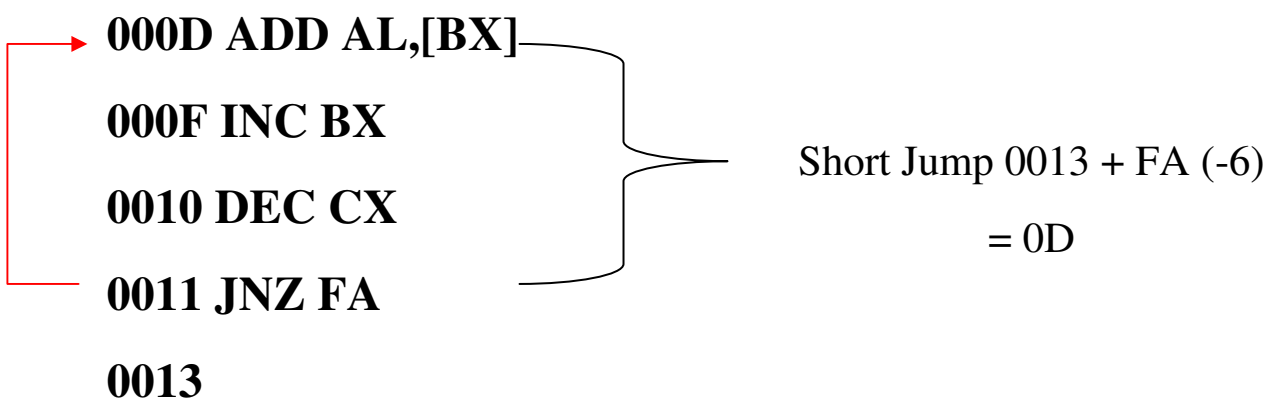
Control Transfer Instructions (conditional)

- It is often necessary to transfer the program execution.
  - Short
    - A special form of the direct jump: “short jump”
    - **All conditional jumps are short jumps**
    - Used whenever target address is in range +127 or –128 (single byte)
    - Instead of specifying the address a relative offset is used.

Short Jumps

- **Conditional Jump is a two byte instruction.**
- **In a jump backward the second byte is the 2's complement of the displacement value.**
- **To calculate the target the second byte is added to the IP of the instruction after the jump.**

Ex:




Hello2.exe

SJ Example

MS-DOS Prompt - DEBUG


```
-q

C:\>cd irvine

C:\Irvine>debug hello2.exe
-u 0 25
16EF:0000 B8F116        MOV     AX,16F1
16EF:0003 8ED8          MOV     DS,AX
16EF:0005 B400          MOV     AH,00
16EF:0007 CD16          INT     16
16EF:0009 3C61          CMP     AL,61
16EF:000B 720F          JB      001C
16EF:000D 3C7A          CMP     AL,7A
16EF:000F 770B          JA      001C
16EF:0011 B409          MOV     AH,09
16EF:0013 BA1200        MOV     DX,0012
16EF:0016 B409          MOV     AH,09
16EF:0018 CD21          INT     21
16EF:001A CD20          INT     20
16EF:001C BA3A00        MOV     DX,003A
16EF:001F B409          MOV     AH,09
16EF:0021 CD21          INT     21
16EF:0023 B8004C        MOV     AX,4C00
-
```

```
.model small
.stack 100h
.data
org 0010
message1 db "You now have a small letter
entered !",0dh,0ah,'$'
org 50
message2 db "You have NON small letters
",0dh,0ah,'$'
.code
    main proc
        mov ax,@data
        mov ds,ax
        mov ah,00h
        int 16h
        cmp al,61h
        jb next
        Cmp al,7Ah
        ja next
        mov ah,09h
        mov dx,offset message1
        mov ah,09h
        int 21h
        int 20h
        next: mov dx,offset message2
        mov ah,09h
        int 21h
        mov  ax,4C00h
        int  21h
    main endp
end main
```

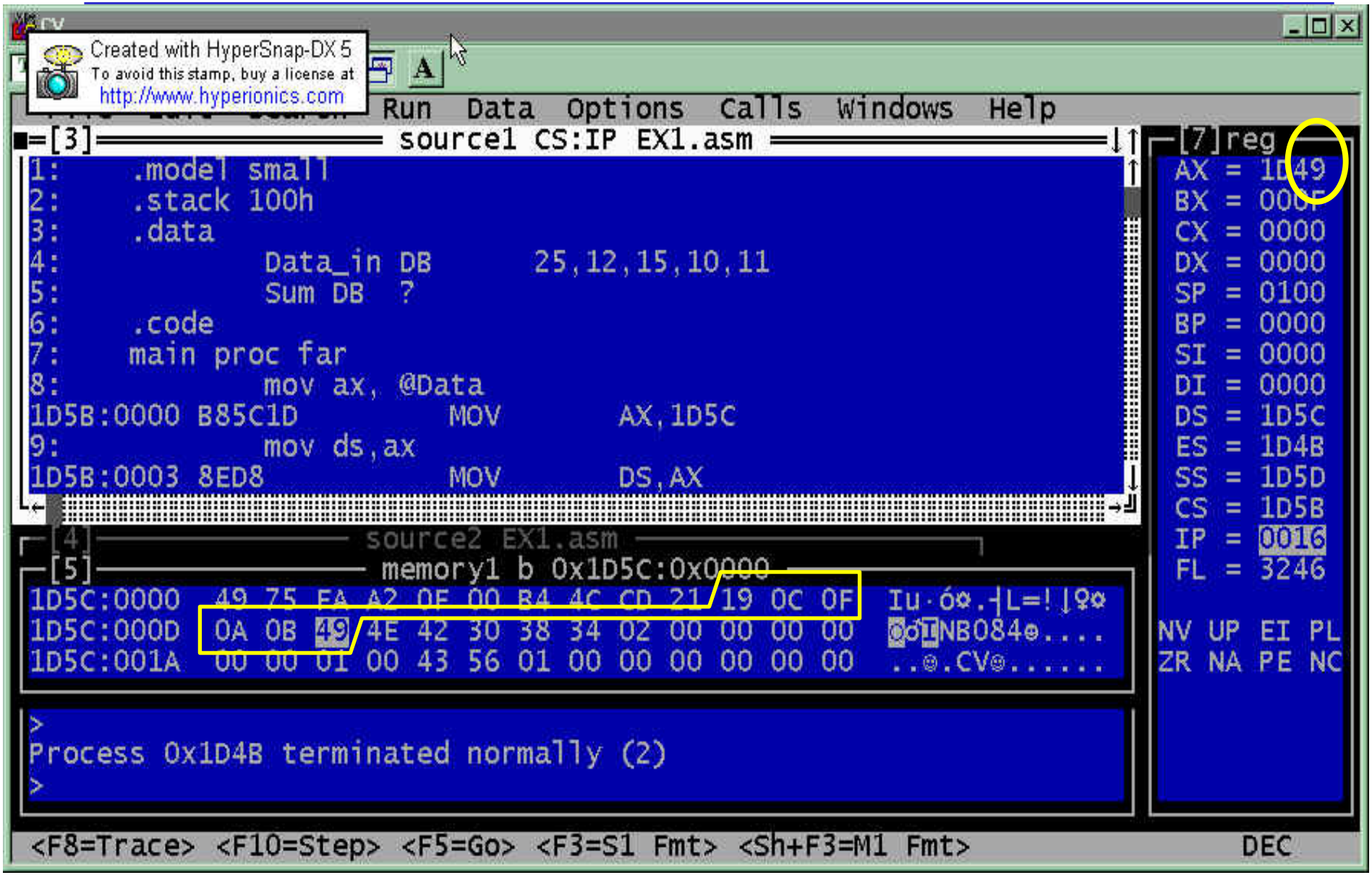
A Simple Example Program finds the sum

- Write a program that adds 5 bytes of data and saves the result. The data should be the following numbers: 25,12,15,10,11

```
.model small
.stack 100h
.data
        Data_in DB 25,12,15,10,11
        Sum DB  ?
.code
main proc far
        mov ax, @Data
        mov ds,ax
        mov cx,05h
        mov bx,offset data_in
        mov al,0
```

```
Again: add al,[bx]
        inc bx
        dec cx
        jnz Again
        mov sum,al
        mov ah,4Ch
        INT 21H
Main    endp
end main
```

Example Output



Unconditional Jump

❖Short Jump: jmp short L1 (8 bit)

❖Near Jump: jmp near ptr Label

If the control is transferred to a memory location within the current code segment (intrasegment), it is NEAR. IP is updated and CS remains the same

➢The displacement (16 bit) is added to the IP of the instruction following jump instruction. The displacement can be in the range of –32,768 to 32,768.

➢The target address can be register indirect, or assigned by the label.

➢**Register indirect JMP:** the target address is the contents of two memory locations pointed at by the register.

➢Ex: JMP [SI] will replace the IP with the contents of the memory locations pointed by DS:DI and DS:DI+1 or JMP [BP + SI + 1000] in SS

❖Far Jump: If the control is transferred to a memory location outside the current segment. Control is passing outside the current segment both CS and IP have to be updated to the new values. ex: JMP FAR PTR label = EA 00 10 00 20

jmp far ptr Label ; this is a jump out of the current segment.

Near Jump

```
0B20:1000 jmp 1200
0B20:1003
-u 1000
0B20:1000 E9FD01          JMP     1200
0B20:1003 200B            AND     [BP+DI],CL
```

Jumps to the specified IP with +/- 32K distance from the next instruction following the jmp instruction

Far Jump

```
0B20:1000 jmp 3000:1200
0B20:1005
-u 1000
0B20:1000 EA00120030    JMP     3000:1200
0B20:1005 FF750B        PUSH    [DI+0B]
```

Jumps to the specified CS:IP

XLAT

- Adds the contents of AL to BX and uses the resulting offset to point to an entry in an 8 bit translate table.
- This table contains values that are substituted for the original value in AL.
- The byte in the table entry pointed to by BX+AL is moved to AL.

- XLAT [tablename] ; optional because table is assumed at BX

- Table db '0123456789ABCDEF'

Mov AL,0A; index value

Mov bx,offset table

Xlat; AL=41h, or 'A'

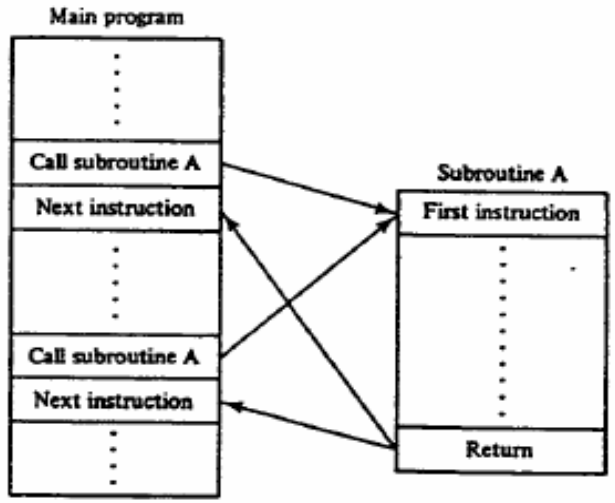
Subroutines and Subroutine Handling Functions

✓A subroutine is a special segment of a program that can be called for execution from any point in the program

✓A RET instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment

Examples. **Call 1234h**
Call BX
Call [BX]

Two calls
•intrasegment
•intersegment



(a)

Mnemonic	Meaning	Format	Operation	Flags Affected
CALL	Subroutine call	CALL operand	Execution continues from the address of the subroutine specified by the operand. Information required to return back to the main program such as IP and CS are saved on the stack.	None

(b)

Operand
Near-proc
Far-proc
Memptr16
Regptr16
Memptr32

(c)

Figure 6–20 (a) Subroutine concept. (b) Subroutine call instruction. (c) Allowed operands.

Calling a NEAR proc

✓The CALL instruction and the subroutine it calls are in the same segment.

✓Save the current value of the IP on the stack.

✓load the subroutine's offset into IP (nextinst + offset)

Calling Program	Subroutine	Stack

```
Main proc          sub1 proc
001A: call sub1    0080: mov ax,1
001D: inc ax       …
.                  ret
Main endp          sub1 endp
```

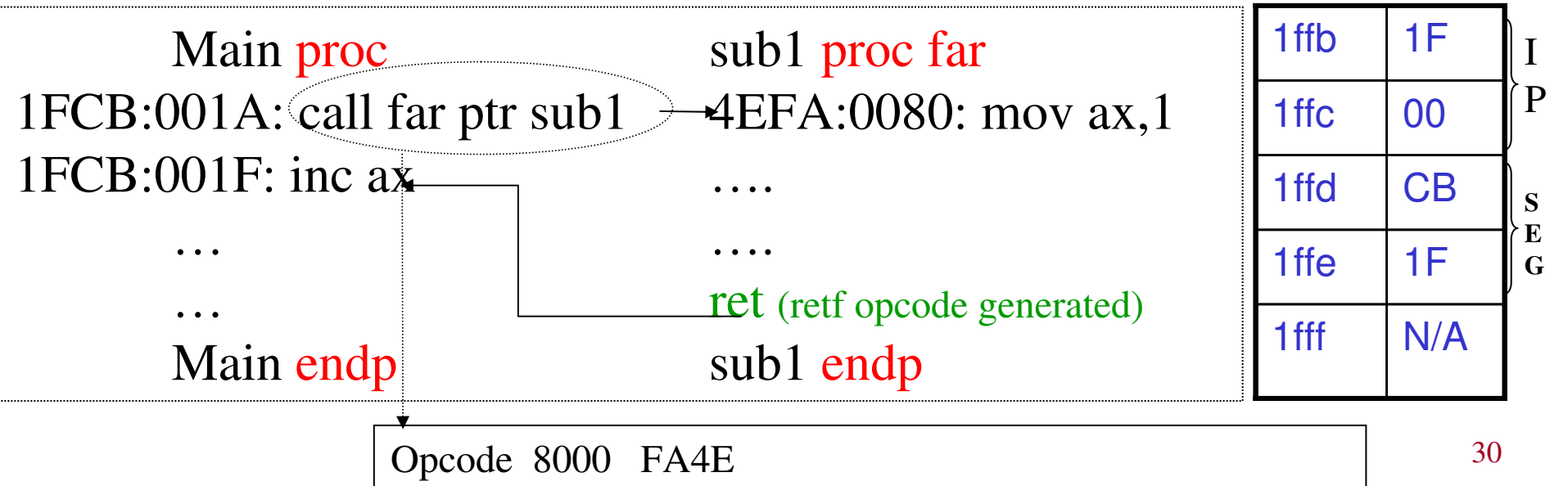
1ffd	1D
1ffe	00
1fff	(not used)

Calling a FAR proc

✓The CALL instruction and the subroutine it calls are in the "Different" segments.

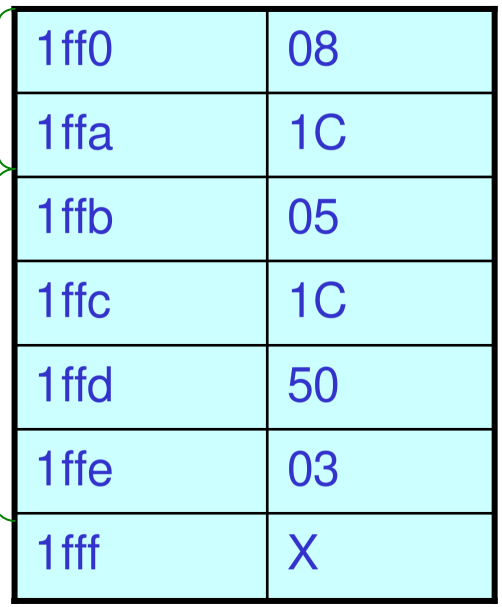
✓Save the current value of the CS and IP on the stack.

✓Then load the subroutine's CS and offset into IP.



Example on Far/Near Procedure Calls

0350:1C00 Call FarProc
0350:1C05 Call NearProc
0350:1C08 nop



Nested Procedure Calls

A subroutine may itself call other subroutines.

Example:

```
                main proc
000A            call subr1
000C            mov ax,…
…

                main endp
```

```
                subr1  proc
0030            nop
                …
                call subr2
0040            ret …

                subr1 endp
```

```
                subr2  proc
0050            nop
                …
                call subr3
0060            ret …

                subr2 endp
```

```
                subr3  proc
0070            nop
                …
0079            nop
007A            ret

                subr3 endp
```

Q: show the stack contents at 0079?

1ff0	60
1ffa	00
1ffb	40
1ffc	00
1ffd	0c
1ffe	00
1fff	X

Do NOT overlap Procedure Declarations

Push and Pop Instructions

To save registers and parameters on the stack

```
PUSH XX
PUSH YY
PUSH ZZ
```

Main body of the subroutine

```
.
.
.
.
.
.
```

To restore registers and parameters from the stack

```
POP  ZZ
POP  YY
POP  XX
```

Return to main program

```
RET
```

Push S (16/32 bit or Mem)
(SP) ← (SP) - 2
((SP)) ← (S)

Pop D (16/32 bit or Mem)
(D) ← ((SP))
(SP) ← (SP) + 2

Loop and Loop Handling Instructions

Mnemonic	Meaning	Format	Operation
LOOP	Loop	LOOP Short-label	(CX) ← (CX) − 1 Jump is initiated to location defined by short-label if (CX) ≠ 0; otherwise, execute next sequential instruction
LOOPE/LOOPZ	Loop while equal/ loop while zero	LOOPE/LOOPZ Short-label	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 1; otherwise, execute next sequential instruction
LOOPNE/ LOOPNZ	Loop while not equal/ loop while not zero	LOOPNE/LOOPNZ Short-label	(CX) ← (CX) − 1 Jump to location defined by short-label if (CX) ≠ 0 and (ZF) = 0; otherwise, execute next sequential instruction

Figure 6–28 Loop instructions.

Loop

```
        MOV CX,COUNT        Load count for the number of repeats
NEXT:       .
            .
            .
            .               Body of routine that is repeated
            .
            .
            .
        LOOP NEXT           Loop back to label NEXT if count not zero
```

(a)

```
        MOV     AX,DATASEGADDR
        MOV     DS,AX
        MOV     SI,BLK1ADDR
        MOV     DI,BLK2ADDR
        MOV     CX,N
NXTPT:  MOV     AH,[SI]
        MOV     [DI],AH
        INC     SI
        INC     DI
        LOOP    NXTPT
        HLT
```

(b)

Nested Loops

single Loop

```
MOV CX,A
BACK: …
…
…
…
LOOP BACK
```

Nested Loops

```
        MOV CX,A
OUTER:  PUSH CX
        MOV CX, 99
INNER:  NOP
        …
        …
        …
        LOOP INNER
        POP CX
        LOOP OUTER
```

How many times will the loop execute, if JCXZ wasn't there

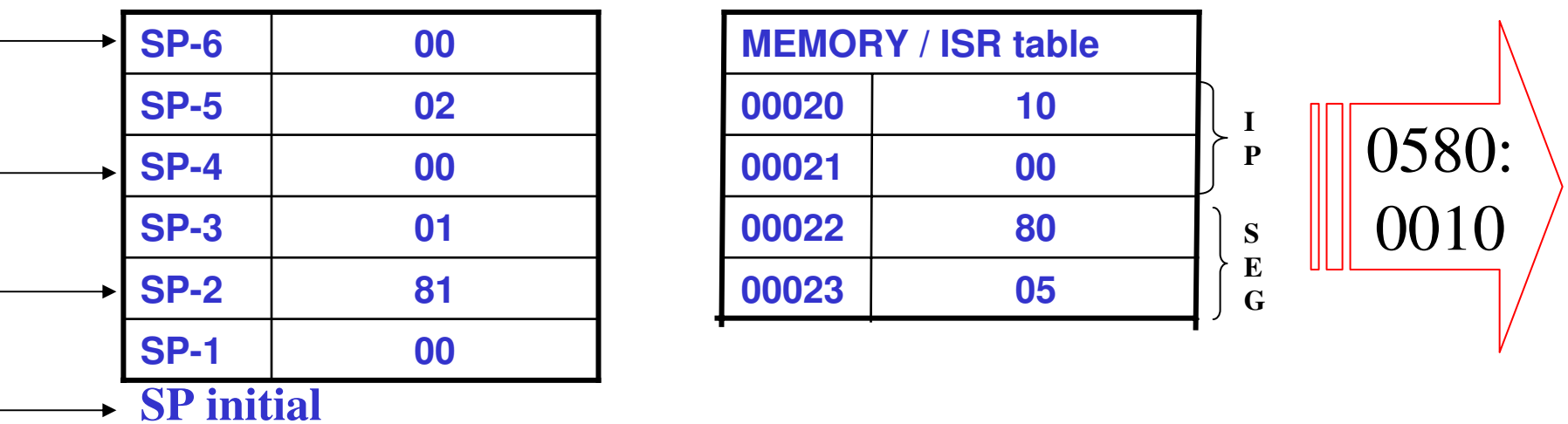
```
MOV CX,0
DLOOP: JCXZ SKIP ;guarding
BACK: MUL AX,2H
ADD AX,05H
LOOP BACK
SKIP: INC AX; if CX=0
```

INT

INT operates similar to Call

- ❖Processor first pushes the flags
- ❖Trace Flag and Interrupt-enable flags are cleared
- ❖Next the processor pushes the current CS register onto the stack
- ❖Next the IP register is pushed

Example: What is the sequence of events for INT 08? If it generates a CS:IP of 0100:0200. The flag is 0081H.



IRET

- IRET must be used for special handling of the stack.
- Must be used at the end of an ISR

SP-6	00
SP-5	02
SP-4	00
SP-3	01
SP-2	81
SP-1	00

SP initial

Return address + flags are loaded

String Instructions

80x86 is equipped with special instructions to handle string operations
String: A series of data words (or bytes) that reside in consecutive memory locations
Operations: move, scan, compare

String Instruction:
Byte transfer, SI or DI increment or decrement by 1
Word transfer, SI or DI increment or decrement by 2
DWord transfer, SI or DI increment or decrement by 4

String Instructions - D Flag

The Direction Flag: Selects the auto increment D=0 or the auto decrement D=1 operation for the DI and SI registers during string operations. D is used only with strings

Mnemonic	Meaning	Format	Operation	Flags Affected
CLD	Clear DF	CLD	(DF) ← 0	DF
STD	Set DF	STD	(DF) ← 1	DF

CLD → Clears the D flag / STD → Sets the D flag

String Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
MOVS	Move string	MOVSB/MOVSW	((ES)0 + (DI)) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None
CMPS	Compare string	CMPSB/CMPSW	Set flags as per ((DS)0 + (SI)) − ((ES)0 + (DI)) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
SCAS	Scan string	SCASB/SCASW	Set flags as per (AL or AX) − ((ES)0 + (DI)) (DI) ← (DI) ± 1 or 2	CF, PF, AF, ZF, SF, OF
LODS	Load string	LODSB/LODSW	(AL or AX) ← ((DS)0 + (SI)) (SI) ← (SI) ± 1 or 2	None
STOS	Store string	STOSB/STOSW	((ES)0 + (DI)) ← (AL or AX) ± 1 or 2 (DI) ← (DI) ± 1 or 2	None

```
        MOV     AX,DATASEGADDR
        MOV     DS,AX
        MOV     ES,AX
        MOV     SI,BLK1ADDR
        MOV     DI,BLK2ADDR
        MOV     CX,N
        CLD
NXTPT:  MOVSB
        LOOP    NXTPT
        HLT
```

Repeat String REP

Basic string operations must be repeated in order to process arrays of data; this is done by inserting a repeat prefix.

Prefix	Used with:	Meaning
REP	MOVS STOS	Repeat while not end of string CX ≠ 0
REPE/REPZ	CMPS SCAS	Repeat while not end of string and strings are equal CX ≠ 0 and ZF = 1
REPNE/REPNZ	CMPS SCAS	Repeat while not end of string and strings are not equal CX ≠ 0 and ZF = 0

Figure 6–36 Prefixes for use with the basic string operations.

Example. Find and replace

- Write a program that scans the name “Mr.Gohns” and replaces the “G” with the letter “J”.


search.asm

```
   Data1 db  'Mr.Gones','$`
   .code
   mov es,ds
   cld ;set auto increment bit D=0
   mov di, offset data1
   mov cx,09; number of chars to be scanned
   mov al,'G'; char to be compared against
   repne SCASB; start scan AL =? ES[DI]
   jne Over; if Z=0
   dec di; Z=1
   mov byte ptr[di], 'J'
Over:  mov ah,09
   mov dx,offset data1
  int 21h; display the resulting String
```


Search.exe

Strings into Video Buffer

Fill the Video Screen with a value


```
CLD
MOV AX,0B800H
MOV ES,AX
MOV DI,0
MOV CX,2000H
MOV AL,20h
REP STOSW
```

Example. Display the ROM BIOS Date

- Write an 8086 program that searches the BIOS ROM for its creation date and displays that date on the monitor.
- If a date cannot be found display the message “date not found”
- Typically the BIOS ROM date is stored in the form xx/xx/xx beginning at system address F000:FFF5
- Each character is in ASCII form and the entire string is terminated with the null character (00)
- Add a ‘$’ character to the end of the string and make it ready for DOS function 09, INT 21

UNIT-II
8086 ASSEMBLY LANGUAGE PROGRAMMING

Contents at a glance:

- ✓ 8086 Instruction Set
- ✓ Assembler directives
- ✓ Procedures and macros.

8086 MEMORY INTERFACING:

- ✓ 8086 addressing and address decoding
- ✓ Interfacing RAM, ROM, EPROM to 8086

INSTRUCTION SET OF 8086

The 8086 instructions are categorized into the following main types

(i) **Data copy /transfer instructions:** These type of instructions are used to transfer data from source operand to destination operand. All the store, load, move, exchange input and output instructions belong to this category.

(ii) **Arithmetic and Logical instructions:** All the instructions performing arithmetic , logical, increment, decrement, compare and ASCII instructions belong to this category.

(iii) **Branch Instructions:** These instructions transfer control of execution to the specified address. All the call, jump, interrupt and return instruction belong to this class.

(iv) **Loop instructions:** These instructions can be used to implement unconditional and conditional loops. The LOOP, LOOPNZ , LOOPZ instructions belong to this category.

(v) **Machine control instructions:** These instructions control the machine status. NOP, HLT, WAIT and LOCK instructions belong to this class.

(vi) **Flag manipulation instructions:** All the instructions which directly effect the flag register come under this group of instructions. Instructions like CLD, STD, CLI, STI etc.., belong to this category of instructions.

(vii) **Shift and Rotate instructions:** These instructions involve the bit wise shifting or rotation in either direction with or without a count in CX.

(viii) **String manipulation instructions:** These instructions involve various string manipulation operations like Load, move, scan, compare, store etc..,

1. Data Copy/ Transfer Instructions:

The following instructions come under data copy / transfer instructions:

MOV	**PUSH**	**POP**	**IN**	**OUT**	**PUSHF**	**POPF**	**LEA**	**LDS/LES**	**XLAT**
XCHG	**LAHF**	**SAHF**							

Data Copy/ Transfer Instructions:
MOV: MOVE: This data transfer instruction transfers data from one register / memory location to another register / memory location. The source may be any one of the segment register or other general purpose or special purpose registers or a memory location and another register or memory location may act as destination.

Syntax: 1) MOV mem/reg1, mem/reg2

[mem/reg1] ← [mem/reg2]

Ex: MOV BX, 0210H
MOV AL, BL
MOV [SI], [BX] → is not valid

Memory uses DS as segment register. No memory to memory operation is allowed. It won't affect flag bits in the flag register.

2) MOV mem, data
[mem] ← data

Ex: MOV [BX], 02H
MOV [DI], 1231H

3) MOV reg, data
[reg] ← data

Ex: MOV AL, 11H
MOV CX, 1210H

4) MOV A, mem
[A] ← [mem]

Ex: MOV AL, [SI]
MOV AX, [DI]

5) MOV mem, A
[mem] ← A
A ← : AL/AX

Ex: MOV [SI], AL
MOV [SI], AX

6) MOV segreg,mem/reg
[segreg] ← [mem/reg]

Ex: MOV SS, [SI]

7) MOV mem/reg, segreg
[mem/reg] ← [segreg]

Ex: MOV DX, SS

In the case of immediate addressing mode, a segment register cannot be destination register. In other words, direct loading of the segment registers with immediate data is not permitted. To load the segment registers with immediate data, one will have to load any general-purpose register with the data and then it will have to be moved to that particular segment register.

Ex: Load DS with 5000H
1) MOV DS, 5000H; Not permitted (invalid)

Thus to transfer an immediate data into the segment register, the convert procedure is given below:

2) MOV AX, 5000H
MOV DS, AX

Both the source and destination operands cannot be memory locations (Except for string instructions)

Other MOV instructions examples are given below with the corresponding addressing modes.

3) MOV AX, 5000H; Immediate
4) MOV AX, BX; Register
5) MOV AX, [SI]; Indirect
6) MOV AX, [2000H]; Direct
7) MOV AX, 50H[BX]; Based relative, 50H displacement

PUSH: **Push to Stack**: This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction. The actual current stack-top is always occupied by the previously pushed data. Hence, the push operation decrements SP by two and this store the two-byte contents of the operand onto the stack. The higher byte is pushed first and then the lower byte. Thus out of the two decremental stack addresses the higher byte occupies the higher address and the lower byte occupies the lower address.

Syntax: PUSH reg
[SP] ← [SP]-2
[[S]] ← [reg]

Ex:
1) PUSH AX
2) PUSH DS
3) PUSH [5000H]; content of location 5000H & 5001H in DS are pushed onto the stack.

POP: **Pop from stack**: This instruction when executed, loads the specified register / memory location with the contents of the memory location of which address is formed using the current stack segment and stack pointer as usual. The stack pointer is incremented by 2. The POP instruction serves exactly opposite to the PUSH instruction.
Syntax:

i) POP mem
[SP] ← [SP] +2
[mem] ← [[SP]]

ii) POP reg
[SP] ← [SP] + 2
[reg] ← [[SP]]

Ex:
1. POP AX
2. POP DS
3. POP [5000H]

XCHG: **Exchange**: This instruction exchanges the contents of the specified source and destination operands, which may be registers or one of them may be a memory location. However, exchange of data contents of two memory locations is not permitted.

Syntax:

i) XCHG AX, reg 16
[AX] ⟷ [reg 16]
Ex: XCHG AX, DX

ii) XCHG mem, reg
[mem] ⟷ [reg]
Ex: XCHG [BX], DX

Register and memory can be both 8-bit and 16-bit and memory uses DS as segment register.

iii) XCHG reg, reg
[reg] ⟷ [reg]

Ex: XCHG AL, CL
XCHG DX, BX

Other examples:

1. XCHG [5000H], AX; This instruction exchanges data between AX and a memory location [5000H] in the data segment.
2. XCHG BX; This instruction exchanges data between AX and BX.

I/O Operations:

IN: Input the port: This instruction is used for reading an input port. The address of the input port may be specified in the instruction directly or indirectly AL and AX are the allowed destinations for 8 and 16-bit input operations. DX is the only register (implicit), which is allowed to carry the port address.

Ex: 1. IN AL, DX
[AL] ← [PORT DX]
Input AL with the 8-bit contents of the port addressed by DX

2. IN AX, DX
[AX] ← [PORT DX]
3. IN AL, PORT
[AL]← [PORT]

4. IN AX, PORT
[AX]←[PORT]

5. IN AL, 0300H; This instruction reads data from an 8-bit port whose address is 0300H and stores it in AL.

6. IN AX ; This instruction reads data from a 16-bit port whose address is in DX (implicit) and stores it in AX.

OUT: Output to the Port: This instruction is used for writing to an output port.The address of the output port may be specified in the instruction directly or implicitly in DX. Contents of AX or AL are transferred to a directly or indirectly addressed port after execution of this instruction. The data to an odd addressed port is transferred on D_8 –D_{15} while that to an even addressed port is transferred on D_0-D_7.The registers AL and AX are the allowed source operands for 8-bit and 16-bit operations respectively.

Ex: 1. OUTDX,AL
[PORT DX] ← [AL]
2. OUT DX,AX
[PORT DX] ← [AX]
3. OUT PORT,AL
[PORT] ← [AL]
4. OUT PORT,AX
[PORT] ← [AX]

Output the 8-bit or 16-bit contents of AL or AX into an I/O port addressed by the contents of DX or local port.

5. OUT 0300H,AL; This sends data available in AL to a port whose address is 0300H
6. OUT AX; This sends data available in AX to a port whose address is specified implicitly in DX.

2. Arithmetic Instructions:

ADD	ADC	SUB	SBB	MUL	IMUL	DIV	IDIV	CMP	NEGATE
INC	DEC	DAA	DAS	AAA	AAS	AAM	AAD	CBW	CWD

These instructions usually perform the arithmetic operations, like addition, subtraction, multiplication and division along with the respective ASCII and decimal adjust instructions. The increment and decrement operations also belong

to this type of instructions. The arithmetic instructions affect all the conditional code flags. The operands are either the registers or memory locations immediate data depending upon the addressing mode.

ADD: **Addition:** This instruction adds an immediate data or contents of a memory location specified in the instruction or a register (source) to the contents of another register (destination) or memory location. The result is in the destination operand. However, both the source and destination operands cannot be memory operands. That means memory to memory addition is not possible. Also the contents of the segment registers cannot be added using this instruction. All the condition code flags are affected depending upon the result.

Syntax: i. ADD mem/reg1, mem/reg2
[mem/reg1]← [mem/reg2] + [mem/reg2]

Ex : ADD BL, [ST]
ADD AX, BX

ii. ADD mem, data
[mem]←[mem]+data
Ex: ADD Start, 02H
ADD [SI], 0712H

iii. ADD reg, data
[reg]←[reg]+data
Ex: ADD CL, 05H
ADD DX, 0132H

iv. ADD A, data
[A]←[A]+data
Ex: ADD AL, 02H
ADD AX, 1211H

Examples with addressing modes:

1. ADD AX, 0100H	Immediate
2. ADD AX, BX	Register
3. ADD AX, [SI]	Register Indirect
4. ADD AX, [5000H]	Direct
5. ADD [5000H], 0100H	Immediate
6. ADD 0100H	Destination AX (implicit)

ADC: Add with carry: This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculations) to the result. All the condition code flags are affected by this instruction.

Syntax: i. ADC mem/reg1, mem/reg2
[mem/reg1]←[mem/reg1]+[mem/reg2]+CY

Ex: ADC BL, [SI]
ADC AX, BX

ii. ADC mem,data
[mem]←[mem]+data+CY
Ex: ADC start, 02H
ADC [SI],0712H

iii. ADC reg, data

[reg]←[reg]+data+CY

Ex: ADC AL, 02H
ADC AX, 1211H

Examples with addressing modes:

1. ADC 0100H Immediate(AX implicit)
2. ADC AX,BX Register
3. ADC AX,[SI] Register indirect
4. ADC AX,[5000H] Direct
5. ADC [5000H],0100H Immediate

SUB: Subtract: The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand. Source operand may be a register or a memory location, but source and destination operands both must not be memory operands. Destination operand cannot be an immediate data. All the condition code flags are affected by this instruction.

Syntax: i. Sub mem/reg1, mem/reg2
[mem/reg1]←[mem/reg2]-[mem/reg2]
Ex: SUB BL,[SI]
SUB AX, BX

ii. SUB mem/data
[mem]←[mem]-data
Ex: SUB start, 02H
SUB [SI],0712H

iii. SUB A,data
[A]←[A]-data
Ex: SUB AL, 02H
SUB AX, 1211H

Examples with addressing modes:

1. SUB 0100H Immediate [destination AX]
2. SUB AX, BX Register
3. SUB AX,[5000H] Direct
4. SUB [5000H], 0100 Immediate

SBB: Subtract with Borrow: The subtract with borrow instruction subtracts the source operand and the borrow flag (CF)which may reflect the result of the previous calculations, from the destination operand .Subtraction with borrow ,here means subtracting 1 from the subtraction obtained by SUB ,if carry (borrow) flag is set.

The result is stored in the destination operand. All the conditional code flags are affected by this instruction.

Syntax: i. SBB mem/reg1,mem/reg2
[mem/reg1] ← [mem/reg1]-[mem/reg2]-CY
Ex: SBB BL,[SI]
SBB AX,BX

ii. SBB mem,data
[mem] ← [mem]-data-CY
Ex: SBB Start,02H
SBB [SI],0712H

iii. SBB reg,data
[reg] ← [reg]-data-CY

Ex: SBB CL,05H

SBB DX,0132H

iv. SBB A,data
[A] ← [A]-data-CY

Ex: SBB AL,02H
SBB AX,1211H

INC: Increment: This instruction increments the contents of the specified register or memory location by 1. All the condition flags are affected except the carry flag CF. This instruction adds a to the content of the operand. Immediate data cannot be operand of this instruction.

Syntax: i. INC reg16
[reg 16]←[reg 16]+1
Ex: INC BX

ii. INC mem/reg 8
[mem]←[mem]+1
[reg 8]←[reg 8]+1

Ex: INC BL
INC SI

Segment register cannot be incremented. This operation does not affect the carry flag.

Examples with addressing modes:
1. INC AX Register
2. INC [BX] Register indirect
3. INC [5000H] Direct

DEC: **Decrement:** The decrement instruction subtracts 1 from the contents of the specified register or memory location. All the condition code flags except carry flag are affected depending upon the result. Immediate data cannot be operand of the instruction.

Syntax: i. DEC reg16
[reg 16]←[reg 16]-1
Ex: DEC BX

ii. DEC mem/reg8
[mem]←[mem-1
[reg 8]←[reg 8]-1
Ex: DEC BL

Segment register cannot be decremented.

Examples with addressing mode:
1. DEC AX Register
2. DEC [5000H] Direct

MUL: Unsigned multiplication Byte or Word: This instruction multiplies unsigned byte or word by the content of AL. The unsigned byte or word may be in any one of the general-purpose register or memory locations. The most significant word of result is stored in DX, while the least significant word of the result is stored in AX. All the flags are modified depending upon the result. Immediate operand is not allowed in this instruction. If the most significant byte or word of the result is '0' IF and OF both will be set.

Syntax: MUL mem/reg

For 8X8
[AX]←[AL]*[mem8/reg8]
Ex: MUL BL
[AX]←[AL]*[BL]
For 16X16
[DX][AX]←[AX]*[mem16/reg16]
Ex: MUL BX
[DX][AX]←[AX]*[BX]

higher lower
16-bit 16-bit

Ex: 1. MUL BH ; [AX]←[AL]*[BH]
2. MUL CX ; [DX][AX]←[AX*[CX]
3. MUL WORD PTR[SI];[DX][AX]←[AX]*[SI]

IMUL: Signed Multiplication: This instruction multiplies a signed byte in source operand by a signed byte in AL or signed word in source operand by signed word in AX. The source can be a general purpose register, memory operand, index register or base register, but it cannot be an immediate data. In case of 32-bit results, the higher order word (MSW) is stored in DX and the lower order word is stored in AX. The AF, PF, SF and ZF flags are undefined after IMUL. If AH and DH contain parts of 16 and 32-bit result respectively, CF and OF both will of set. The AL and AX are the implicit operands in case of 8-bit and 16-bit multiplications respectively. The unused higher bits of the result are filled by sign bit and CF, AF are cleared.

Syntax: IMUL mem/reg
For 8X8
[AX]←[AL]*[mem8/reg8]
Ex: IMUL BL
[AX]←[AL]*[BL]

For 16X16
[DX][AX]←[AX*[mem16/reg16]
Ex: IMUL BX
[DX][AX]←[AX]*[BX]

Memory or register can be 8-bit or 16-bit and this instruction will affect carry flag & overflow flag.
Ex: 1. IMUL BH
2. IMUL CX
3. IMUL [SI]

DIV: Unsigned division: This instruction performs unsigned division. It divides an unsigned word or double word by a 16-bit or 8-bit operand. The dividend must be in AX for 16-bit operation and divisor may be specified using any one of the addressing modes except immediate. The result will be in AL (quotient) while AH will contain the remainder. If the result is too big to fit in AL, type 0(divide by zero) interrupt is generated. In case of a double word dividend (32-bit), the higher word should be in DX and lower word should be in AX. The divisor may be specified as already explained. The quotient and the remainder, in this case, will be in AX and DX respectively. This instruction does not affect any flag.

Syntax: DIV mem/reg

Ex: DIV BL (i.e. [AX]/[BX])

For 16 ÷ 8 [AX] / [mem 8/reg 8] → [AH]← Remainder, [AL]← Quotient

[DX] [AX] [DX]← Remainder

For 32 ÷ 16 _____➔

[mem 16/reg 16] [AX]← Quotient

[DX][AX]

Ex: DIV BX (i.e. _____)

[BX]

IDIV: Signed Division: This instruction performs same operation as the DIV instruction, but it with signed operands the results are stored similarly as in case of DIV instruction in both cases of word and double word divisions the results will also be signed numbers. The operands are also specified in the same way as DIV instruction. Divide by zero interrupt is generated, if the result is too big to fit in AX (16-bit dividend operation) or AX and DX (32-bit dividend operation) all the flags are undefined after IDIV instruction.

AAA: ASCII Adjust after addition: The AAA instruction is executed after an ADD instruction that adds two ASCII coded operands to give a byte of result in AL. The AAA instruction converts the resulting contents of AL to unpacked decimal digits. After the addition, the AAA instruction examines the lower 4-bits of AL to check whether it contains a valid BCD number in the range 0 to 9. If it is between 0 to 9 and AF is zero, AAA sets the 4- higher order bits of AL to 0. The AH must be cleared before addition. If the lower digit of AL is between 0 to 9 and AF is set, 06 is added to AL. The upper 4-bits of AL are cleared and AH is incremented by one. If the value of lower nibble of AL is greater than 9 then the AL is incremented by 06, AH is incremented by 1, the AF and CF flags are set to 1, and the higher4-bits of AL are cleared to 0. The remaining flags are unaffected. The AH is modified as sum of previous contents (usually 00) and the carry from the adjustment, as shown in Fig1.7. This instruction does not give exact ASCII codes of the sum, but they can be obtained by adding 3030H to AX.

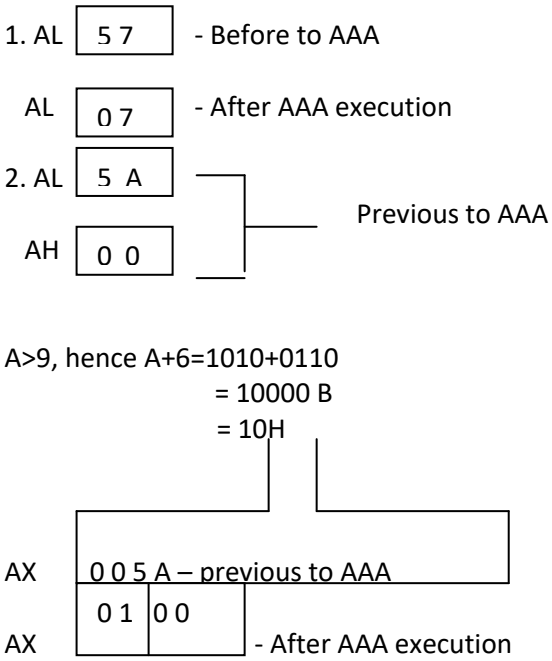


Fig1.7 ASCII Adjust After Addition Instruction

AAS: ASCII Adjust After Subtraction: AAS instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. If the lower 4-bits of AL register are greater than 9 or if the AF flag is one, the AL is decremented by 6 and AH is decremented by 1, the CF and AF are set to 1. Otherwise, the CF and AF are set to 0, the result needs to no correction. As a result, the upper nibble of AL is 00 and the lower nibble may be any number from 0 to 9. The procedure similar to the AAA instruction AH is modified as difference of previous contents (usually 0) of AH and the borrow for adjustment.

AAM: ASCII Adjust after Multiplication: This instruction, after execution, converts the product available in AL into unpacked BCD format. This follows a multiplication instruction. The lower byte of result (unpacked) remains in AL and the higher byte of result remains in AH.

The example given below explains execution of the instruction. Suppose, a product is available in AL, say AL=5D. AAM instruction will form unpacked BCD result in AX. DH is greater than 9, so add of 6(0110) to it D+6=13H. LSD of 13H is the lower unpacked byte for the result. Increment AH by 1, 5+1=6 will be the upper unpacked byte of the result. Thus after the execution, AH=06 and AL=03.

AAD: ASCII Adjust before Division: Though the names of these two instructions (AAM and AAD) appear to be similar, there is a lot of difference between their functions. The AAD instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing number the two unpacked BCD digits in AX by an unpacked BCD byte. PF, SF, ZF are modified while AF, CF, OF are undefined, after the execution of the instruction AAD. The example explains the execution of the instruction.

Let AX contain 0508 unpacked BCD for 58 decimal and DH contain 02H.
Ex:

AX	5	8

AAD result in AL	0	3A

58D=3AH in AL
The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH.

DAA: Decimal Adjust Accumulator: This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL. If the lower nibble is greater than 9, after addition or if AF is set, it will add 06 to the lower nibble in AL. After adding 06 in the lower nibble of AL, if the upper nibble of AL is greater than 9 or if carry flag is set, DAA instruction adds 60H to AL.

The example given below explains the instruction:

i. AL=53 CL=29

```
        ADD AL, CL      ;       AL ←(AL) + (CL)
                        ;       AL←53+29
                        ;       AL←7C
                        ;       AL←7C+06(as C>9)
                        ;       AL←82
```

ii. AL=73 CL=29

```
        ADD AL,CL       ;       AL←AL+CL
                        ;       AL←73+29
                        ;       AL←9C
                        ;       AL←9C
        DAA             ;       AL←02 & CF=1
        AL=73
          +
        CL=29
        _____
          9C
          +6
        _____

          A2
          +60
        _____
        CF=1 02 in AL
```

The instruction DAA affects AF, CF, PF and ZF flags. The OF flag is undefined.

DAS: Decimal Adjust After Subtraction: This instruction converts the results of subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only. If the lower nibble of AL is greater than 9, this instruction will subtract 06 from lower nibble of AL. If the result of subtractions sets the carry flag or if upper nibble is greater than 9, it subtracts 60H from AL. This instruction modifier the AF, CF, PF and ZF flags. The OF is undefined after DAS instruction.
The examples are as follows:

Ex: i. AL=75 BH=46
SUB AL,BH ; AL←2F=(AL)-(BH)
; AF=1
DAS ; AL←29 (as F>9,F-6=9)

ii. AL=38 CH=61
SUB AL, CH ; AL←D7 CF=1(borrow)
DAS ; AL←77(as D>9, D-6=7)
; CF=1(borrow)

NEG: Negate: The negate instruction forms 2's complement of the specified destination in the instruction. For obtaining 2's complement, it subtracts the contents of destination from zero. The result is stored back in the destination operand which may be a register or a memory location. If OF is set, it indicates that the operation could not be completed successfully. This instruction affects all the condition code flags.

CBW: Convert signed Byte to Word: This instruction converts a signed byte to a signed word. In other words, it copies the sign bit of a byte to be converted to all the bits in the higher byte of the result word. The byte to be converted must be in AL. The result will be in AX. It does not affect any flag.

CWD: Convert Signed Word to double Word: This instruction copies the sign bit of AX to all the bits of DX register. This operation is to be done before signed division. It does not affect any other flag.

3. Logical Instructions:

AND	OR	NOT	XOR	TEST

These byte of instructions are used for carrying out the bit by bit shift, rotate or basic logical operations. All the conditional code flags are affected depending upon the result. Basic logical operations available with 8086 instruction set an AND, OR, NOT and XOR.

AND: Logical AND: This instruction bit by bit ANDs the source operand that may be an immediate, a register, or a memory location to the destination operand that may be a register or a memory location. The result is stored in the destination operand. At least one of the operand should be a register or a memory operand. Both the operands cannot be memory locations or immediate operand.

The examples of this instruction are as follows:

Syntax: i. AND mem/reg1, mem/reg2
[mem/reg1]←[mem/reg1]∧[mem/reg2]
Ex: AND BL, CH

ii. AND mem,data
[mem]←[mem] ∧ data
Ex: AND start,05H

iii. AND reg,data
[reg]←[reg] ∧ data

Ex: AND AL, FOH

iv. AND A,data
[A]←[A] ∧ data
A:AL/AX
Ex: AND AX,1021H

OR: Logical OR: The OR instruction carries out the OR operation in the same way as described in case of the AND operation. The limitations on source and destination operands are also the same as in case of AND operation.

Syntax: i. OR mem/reg1, mem/reg2
[mem/reg1]←[mem/reg1] ∨ [mem/reg2]
Ex: OR BL, CH

ii. OR mem,data
[mem←[mem] ∨ data
Ex: OR start, 05H

iii. OR Start,05H
[reg]←[reg] ∨ data
Ex: OR AL, FOH

iv. OR A, data
[A]←[A] ∨ data
Ex: OR AL, 1021H
A: AL/AX.

NOT: Logical Invert: The NOT instruction complements (invents) the contents of an operand register or a memory location bit by bit.

Syntax: i. NOT reg
[reg] ←[reg]'
Ex: NOT AX

ii. NOT mem
[mem]←[mem]'
Ex: NOT [SI]

XOR: Logical Exclusive OR: The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on high output, when the 2 input bits are dissimilar. Otherwise, the output is zero.

Syntax: i. XOR mem/reg1, mem/reg2
[mem/reg1]←[mem/reg1] ⊕ [mem/reg2]
Ex: XOR BL, CH

ii. XOR mem,data
[mem] ← [mem] ⊕ data
Ex: XOR start, 05H

iii. XOR reg, data
[reg]← [reg] ⊕ data
Ex: XOR AL, FOH

iv. XOR A, data
[A]← [A] ⊕ data

A: AL/AX

Ex: XOR AX, 1021H

CMP: Compare: This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending on the result of subtraction. If both the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset.

Syntax: i. CMP mem/reg1, mem/reg2
[mem/reg1] – [mem/reg2]
Ex: CMP CX, BX

ii. CMP mem/reg, data
[mem/reg] – data
Ex: CMP CH, 03H

iii. CMP A, data
[A]- data
A: AL/AX
Ex: CMP AX, 1301H

TEST: Logical Compare Instruction: The TEST instruction performs a bit by bit logical AND operation on the two operands. Each bit of the result is then set to 1, if the corresponding bits of both operands are1, else the result bit is rest to 0. The result of this and operation is not available for further use, but flags are affected. The affected flags are OF, CF, ZF and PF. The operands may be register, memory or immediate data.

Syntax: i. TEST mem/reg1, mem/reg2
[mem/reg1] $\wedge$ [mem/reg2]
Ex: Test CX,BX
ii. TEST mem/reg, data
[mem/reg] $\wedge$ data
Ex: TEST CH, 03H

iii. TEST A, data
[A] $\wedge$ data
A: AL/AX
Ex: TEST AX, 1301H

4.Shift Instructions:

SHL/SAL	SHR	SAR

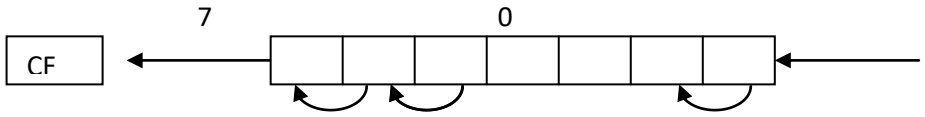
SHL/SAL: Shift Logical/ Arithmetic Left: These instructions shift the operand word or byte bit by bit to the left and insert zeros in the newly introduced least significant bits. In case of all the SHIFT and ROTATE instructions, the count is either 1 or specified by register CL. The operand may reside in a register or memory location but cannot be immediate data. All flags are affected depending on the result.

Ex:

BIT POSITIONS: CF 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OPERAND: 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1

SHL 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1 0
RESULT1st
SHL 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1 0 0
RESULT 2nd

Syntax: i. SAL mem/reg,1
Shift arithmetic left once



ii. SAL mem/reg, CL

Shift arithmetic left a byte or word by shift count in CL register.

iii. SHL mem/reg,1
Shift Logical Left
Ex: SHL BL, 01H

iv. SHL mem/reg, CL
Shift Logical Left once a byte or word in mem/reg.

SHR: Shift Logical Right: This instruction performs bit-wise right shifts on the operand word or byte that may reside in a register or a memory location, by the specified count in the instruction and inserts zeros in the shifted positions. The result is stored in the destination operand. This instruction shifts the operand through carry flag.

Ex:

BIT POSITIONS: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 CF
OPERAND : 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1

Count=1 0 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1

Count=2 0 0 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0

SAR: Shift Arithmetic Right: This instruction performs right shifts on the operand word or byte, that may be a register or a memory location by the specified count in the instruction and inserts the most significant bit of the operand the newly inserted positions. The result is stored in the destination operand. All the condition code flags are affected. This shift operation shifts the operand through carry flag.

Ex:

BIT POSITIONS: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 CF
OPERAND: 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1

Count=1 1 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1

inserted MSB=1

Count=2 1 1 1 0 1 0 1 1 0 0 1 0 1 0 0 1 0

inserted MSB=1

Immediate operand is not allowed in any of the shift instructions.

Syntax: i. SAR mem/reg,1
ii. SAR mem/reg, CL

5.Rotate Instructions:

ROR	**ROL**
RCR	**RCL**

ROR: Rotate Right without Carry: This instruction rotates the contents of the destination operand to the right (bit-wise) either by one or by the count specified in CL, excluding carry. The least significant bit is pushed into the carry flag and simultaneously it is transferred into the most significant bit position at each operation. The remaining bits are shifted right by the specified positions. The PF, SF, and ZF flags are left unchanged by the rotate operation. The operand may be a register or a memory location but it can't be an immediate operand. The destination operand may be a register (except a segment register) or a memory location.

Syntax: i. mem/reg, 01
Ex: ROR BL, 01

ii. ROR mem/reg, CL
Ex: ROR BX, CL

Ex:

BIT POSITIONS:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CF
OPERAND:	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1	
Count=1		1	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1
Count=2	0	1	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	

Execution of ROR Instruction.

ROL: Rotate Left without Carry: This instruction rotates the content of the destination operand to the left by the specified count (bit-wise) excluding carry. The most significant bit is pushed into the carry flag as well as the least significant bit position at each operation. The remaining bits are shifted left subsequently by the specified count positions. The PF, SF and ZF flags are left unchanged by this rotate operation. The operand may be a register or a memory location.

Syntax: i. ROL mem/reg, 1
Rotate once left

ii. ROL mem/reg, CL
Rotate once left a byte or a word in mem/reg.

Ex:

BIT POSITIONS:	CF	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPERAND :		1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1
SHL RESULT 1st:	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1
SHL RESULT 2nd:	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1	0

Execution of ROL instruction

RCR: Rotate Right Through Carry: This instruction rotates the contents (bit-wise) of the destination operand right by the specified count through carry flag (CF) For each operation, the carry flag is pushed into the MSB of the operand, and the LSB is pushed into carry flag. The remaining bits are shifted right by the specified count positions. The SF, PF, ZF are left unchanged. The operand may be a register or memory location.

Syntax: i. RCL mem/reg, 1
Ex: RCL BL, 1

ii. mem/reg, CL
Ex: RCL BX, CL

Rotate through carry left once a byte or word in mem/reg.

Ex:

BIT POSITIONS:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	CF
OPERAND :	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1	0
Count=1		0	1	0	1	0	1	1	1	1	0	1	0	1	1	1	0	1

Execution of RCR Instruction

RCL: Rotate Left through Carry: This instruction rotates (bit-wise) the contents of the destination operand left by the specified count through the carry flag (CF) For each operation, the carry flag is pushed into LSB, and the MSB of the operand is pushed into carry flag. The remaining bits are shifted left by the specified positions. The SF, PF, ZF are left unchanged. The operand may be a register or a memory location.

Ex:

BIT POSITIONS :	CF	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OPERAND :	0	1	0	0	1	1	1	0	1	1	0	1	1	0	1	0	1
Count=1		1	0	0	1	1	1	0	1	1	0	1	1	0	1	0	1	0

Execution of RCL Instruction

The count for rotation or shifting is either 1 or is specified using register CL, in case of all the shift and rotate instructions.

6.String Manipulation Instructions:

A series of data bytes or words available in memory at consecutive locations, to be referred to collectively or individually are called as **byte strings** or **word strings**.

REP	MOVSB/MOVSW	CMPSB/CMPSW	SCASB/SCASW
STOSB/STOSW	LODSB/LODSW		

REP: Repeat Instruction Prefix: This instruction is used as a prefix to other instructions. The instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one) When CX becomes zero, the execution proceeds to the next instruction in sequence. There are two more options of the REP instruction. The first is REPE/REPZ (i.e. repeat operation which equal/zero. The second is REPNE/REPNZ allows for repeating the operation which not equal/not zero. These options are used for CMPS, SCAS instructions only, as instruction prefixes.

MOVSB/MOVSW: Move String Byte or String Word: Suppose a string of bytes, stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of the source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents. The starting address of the source string is 10H * DS + [SI] while the starting address of the destination string is 10H * ES + [DI]. The MOVSB/MOVSW instruction thus, moves a string of bytes/words pointed to by DS:SI pair (source) to the memory location pointed to by ES:DI pair (destination)

After the MOVS instruction is executed once, the index registers are automatically updated and CX is decremented. The incrementing or decrementing of the pointers, i.e. SI and DI depend upon the direction flag DF. If DF is 0, the index registers are incremented, otherwise, they are decremented, in case of all string manipulation instructions.

Ex:

(a)

```
 DATA   SEGMENT
        TEST-MESS       DB "IT'S TIME FOR A NEW HOME"        ;string to move
                        DB  100  DUP(?) ;stationary block of text
        NEW-LOC         DB   23   DUP(0) ;string destination.
DATA  ENDS

CODE    SEGMENT
        ASSUME          CS:CODE,DS:DATA,ES:DATA
        MOV     AX,DATA         ;initialize data segment register
        MOV     DS,AX
        MOV     ES,AX           ;initialize extra segment register
        LEA     SI,TEST-MESS    ;point SI at source string
        LEA     DI,NEW-LOC              ;point DI at destination string
        MOV     CS,23                   ;use CX register as counter
        CLD                             ;clear DF, so pointers auto increment
REP     MOVSB                   ;after each string element is moved
                                ;move string byte until all moved
CODE    ENDS
        END
```

(b)

Fig : program for moving a string from one location to another in memory
(a) Memory map (b) AL program.

Here, the REPEAT-UNTIL loop then consists of moving a byte, incrementing the pointers to point to the source and destination for next byte, and decrementing the counter to determine whether all bytes have been moved.

The single 8086 instruction MOVSB will perform all the actions in the REPEAT-UNTIL loop. The MOVSB instruction will copy a byte from the location pointed to by the DI register. It will then automatically increment SI to point to next destination location. The repeat (REP) prefix in front of the MOVSB instruction, the MOVSB instruction will be repeated and CX decremented until CX is counted down to zero. In other words, the REP MOVSB instruction will move the entire string from the source location to the destination location if the pointers are properly initialized.

CMPSB/CMPSW: Compare String Byte or String Word: The CMPS instruction is used to compare two strings of bytes or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero flag is set. The flags are affected in the same way as CMP instruction. The DS:SI and ES:DI point to the two strings. The REP instruction prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP prefix is false.

The following string of instructions explain the instruction. The comparison of the string starts from initial or word of the string, after each comparison the index registers are updated depending on the direction flag and the counter is decremented. This byte by byte or word by word comparison continues till a mismatch is found. When, a mismatch is found, the carry and zero flags are modified appropriately and the execution proceeds further.

Ex:

```
MOV AX, SEG1        ; Segment address of String1, i.e. SEG1 is moved to AX.
MOV DS, AX                  ; Load it to DS.
MOV AX, SEG2        ; segment address of STRING2, i.e. SEG@ is moved to AX.
MOV ES, AX                  ; Load it to ES.
MOV SI, OFFSET STRING1      ; Offset of STRING1 Is moved to SI.
MOV DI, OFFSET STRING2      ; Offset of string2 is moved to DI.
MOV CX, 0110H       ; Length of string is moved to CX.
CLD                         ; clear DF, i.e. set auto increment mode.
REPE CMPSW                  ; Compare 010H words of STRING1 And STRING2, while  they are equal, IF a
                            mismatch is found, modify the flags and proceed with further execution.
```

If both strings are completely equal, i.e. CX becomes zero, the ZF is set, otherwise ZF is reset.

SCAS: Scan String BYTE or String Word: This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The string is pointed to by ES:DI register pair. The length of the string is stored in CX. The DF controls the mode for scanning of the string as stated in case of MOVSB instruction. Whenever a match to the specified operand, is found in the string, execution stops and the zero flag is set. If no match is found, the zero flag is reset. The REPNE prefix is used with the SCAS instruction. The pointers and counters are updated automatically, till a match is found.

Ex:

```
MOV AX, SEG          ; Segment address of the string, i.e. SEG is moved to AX.
MOV ES, AX           ; Load it to ES.
MOV DI, OFFSET       ; String offset, i.e. OFFSET is moved to DI.
MOV CX,010H          ; Length of the string is moved to CX.
MOV AX, WORD         ; The word to be scanned for, i.e. WORD is in AL.
CLD                  ; Clear DF
REPNE SCASW          ; Scan the 010H bytes of the string, till a match to WORD is found.
```

This string of instructions finds out, if it contains WORD. IF the WORD is found in the word string, before CX becomes zero, the ZF is set, otherwise the ZF is reset. The scanning will continue till a match is found. Once a match is found the execution of the program proceeds further.

LODS: Load string Byte or String word: The LODS instruction loads the AL/AX register by the content of a string pointed to by DS:SI register pair. The SI is modified automatically depending on DF. If it is a byte transfer (LODSB), the SI is modified bye one and if it is a word transfer (LODSW), the SI is modified by two. No other flags are affected by this instruction.

STOS: Store String Byte or String Word: The STOS instruction stores the AL/AX register contents to a location in the string pointed by ES:DI register pair. The DI is modified Accordingly. No flags are affected by this instruction.

The direction flag controls the string instruction execution. The source index SI and destination index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode. In this mode, SI and DI are decremented automatically after each iteration (by1 or 2 depending on byte or word operations) Hence, in auto decrementing mode, the string are referred to by their ending addresses. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically (by 1 or 2 depending on byte or word operation) After each iteration, hence the strings, in this case, are referred to by their starting addresses.

7.Control Transfer or Branching Instruction:

The control transfer instructions transfer the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location where the flow of execution is going to be transferred.

This type of instructions are classified in two types:

i. **Unconditional control Transfer (Branch) Instructions:**

In case of unconditional control transfer instructions; the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

ii. **Conditional Control Transfer (Branch) Instructions:**

In the conditional control transfer instructions, the control is transferred to the specified location provided the result of the previous operation satisfies a particular condition, otherwise, the execution continues in normal flow sequence. The results of the previous operations are replicated by condition code flags. In other words, using type of instruction the control will be transferred to a particular specified location, if a particular flag satisfies the condition.

Unconditional Branch Instructions:

CALL	**RET**	**JUMP**	**IRET**
INT N	**INT O**	**LOOP**	

CALL: Unconditional Call: This instruction is used to call a subroutine procedure from a main program. The address of the procedure may specify directly or indirectly depending on the address mode.
There are again two types of procedures depending on whether it is available in the same segment (Near CALL, i.e. + 2K displacement) or in another segment (Far CALL, i.e. anywhere outside the segment). The modes for them are respectively called as intrasegment and intersegment addressing (i.e. address of the next instruction) and CS onto the stack along with the flags and loads the CS and IP registers, respectively, with the segment and offset addresses of the procedure to be called.

RET: Return from the Procedure: At each CALL instruction, the IP and CS of the next instruction is pushed onto stack, before the control is transferred to the procedure. At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with flags are retrieved into the CS, IP and flag registers from the stack and the execution of the main program continues further. In case of a FAR procedure the current contents of SP points to IP and CS at the time of return. While in case of a NEAR procedure, it points to only IP. Depending on the byte of procedure and the SP contents, the RET instruction is of four types:

i. Return within a segment.
ii. Return within a segment adding 16-bit immediate displacement to the SP contents.
iii. Return intersegment.
iv. Return intersegment adding 16-bit immediate displacement to the SP contents.

INT N: Interrupt Type N: In the interrupt structure of 8086/8088, 256 interrupts are defined corresponding to the types from 00H to FFH. When an INT N instruction is executed, the TYPE byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from the hexadecimal multiplication (N * 4) as offset address and 0000 as segment address. In other words, the multiplication of type N by 4 (offset) points to a memory block in 0000 segment, which contains the IP and CS values of the interrupt service routine. For the execution of this instruction, the IP must be enabled.

Ex: The INT 20H will find out the address of the interrupt service routine follows:
INT 20H
Type * 4 = 20 X 4 = 80H
Pointer to IP and CS of the ISR is 0000:0080H
The arrangement of CS and IP addresses of the ISR in the interrupt rector table is as follows.

INTO: Interrupt on overflow: This is executed, when the overflow flag OF is set. The new contents of IP an CS are taken from the address 0000:0000 as explained in INT type instruction. This is equivalent to a type 4 instruction.

JMP: Unconditional Jump: This instruction unconditionally transfer the control of execution to the specified address using an 8-bit or 16-bit displacement (intrasegment relative, short or long) or CS:IP (intersegment direct for) No flags are affected by this instruction. Corresponding to the three methods of specifying jump address, the JUMP instruction has the following three formats.

JUMP	DISP 8-bit		Intrasegment, relative, near jump
JUMP	DISP 16-bit	DISP 16-bit	Intrasegment, relative, For jump
JUMP	IP(LB) IP(UB)	CS(LB) CS(UB)	Intersegment, direct, jump

IRET: Return from ISR: When interrupt service routine is to be called, before transferring control to it, the IP, CS and flag register are stored on to the stack to indicate the location from where the execution is to be continued, after the ISR is executed. So, at the end of each ISR, when IRET is executed, the valuesof IP, CS and flags are retrieved from the stack to continue the execution of the main program. The stack is modified accordingly.

LOOP: Loop unconditionally: This instruction executes the part of the program from the label or address specified in the instruction up to the loop instruction, CX number of times. At each iteration, CX is decremented automatically, in other words, this instruction implements DECREMENT counter and JUMP IF NOT ZERO structure.

Ex:

```
        MOV CX,0005H ; Number of times in CX
        MOV BX, 0FF7H ; Data to BX
Label   MOV AX, CODE1
        OR BX,AX
        AND DX,AX
        LOOP Label
```

The execution proceeds in sequence, after the loop is executed, CX number of times. IF CX is already 00H, the execution continues sequentially. No flags are affected by this instruction.

Conditional Branch Instructions:

LOOPE/LOOPZ	LOOPNE/LOOPNZ

When these instructions are executed, they transfer execution control to the address specified relatively in the instruction, provided the condition in the opcode is satisfied, otherwise, the execution continues sequentially. The conditions, here means the status of the condition code flags. These type of instructions don't affect any flags. The address has to be specified in the instruction relatively in terms of displacement, which must lie within – 80H to 7FH (or –128 to 127) bytes from the address of the branch instruction. In other words, only short jumps can be implemented using conditional branch instructions. A label may represent the displacement, if it has within the above-specified range.

The different 8086/8088 conditional branch instructions and their operations are listed in Table1

SL.No	Mnemonic	Displacement	Operation
1	JZ/JE	Label	Transfer execution control to address 'Label', if ZF=1
2	JNZ/JNE	Label	Transfer execution control to address 'Label', if ZF=0
3	JS	Label	Transfer execution control to address 'Label', if SF=1
4	JNS	Label	Transfer execution control to address 'Label', if SF=0
5	JO	Label	Transfer execution control to address 'Label', if OF=1
6	JNO	Label	Transfer execution control to address 'Label', if OF=0
7	JP/JPE	Label	Transfer execution control to address 'Label', if PF=1
8	JNP	Label	Transfer execution control to address 'Label', if PF=0
9	JB/JNAE/JC	Label	Transfer execution control to address 'Label', if CF=1
10	JNB/JNE/JNC	Label	Transfer execution control to address 'Label', if CF=0
11	JBE/JNA	Label	Transfer execution control to address 'Label', if CF=1 or ZF=1
12	JNBE/JA	Label	Transfer execution control to address 'Label', if CF=0 or ZF=0
13	JL/JNGE	Label	Transfer execution control to address 'Label', if neither SF=1 nor OF=1
14	JNL/JGE	Label	Transfer execution control to address 'Label', if neither SF=0 nor OF=0
15	JNE/JNC	Label	Transfer execution control to address

			'Label', if ZF=1or neither SF nor OF is 1
16	JNLE/JE	Label	Transfer execution control to address 'Label', if ZF=0 or at least any are of SF & OF is 1

Table:1 Conditional branch instructions.

8. Flag Manipulation and Processor Control Instructions:

These instructions control the functioning of the available hardware inside the processor chip.

These are categorized into 2 types:

a) flag manipulation instructions
b) Machine control instructions.

The flag manipulation instructions directly modify same of the flags of 8086.

The flag manipulation instructions and their functions are as follows:

CLC – clear carry flag
CMC – Complement carry flag
STC – Set carry flag
CLD – clear direction flag
STD - Set direction flag
CLI – clear interrupt flag
STI – Set interrupt flag

These instructions modify the carry (CF), Direction (DF) and interrupt (IF) flags directly. The DF and IF, which may be the processor operation; like interrupt responses and auto increment or auto-decrement modes. Thus the respective instructions may also be called as machine or processor control instructions. The other flags can be modified using POPF and SAHF instructions, which are termed as data transfer instructions. No direct instructions are available for modifying the status flags except carry flags. The machine control instructions don't require any operational.

The **machine control instructions** supported by 8086/8088 are listed as follows along with their functions:

WAIT	– Wait for Test input pin to go low
HLT	– Halt the processor
NOP	– No operation
ESC	– Escape to external device like NDP
LOCK	– Bus lock instruction prefix.

ASSEMBLER DIRECTIVES

- Assembler directives are the commands to the assembler that direct the assembly process.
- They indicate how an operand is treated by the assembler and how assembler handles the program.
- They also direct the assembler how program and data should arrange in the memory.
- ALP's are composed of two type of statements.

(i) The instructions which are translated to machine codes by assembler.

(ii) The directives that direct the assembler during assembly process, for which no machine code is generated.

1. ASSUME: Assume logical segment name.

The ASSUME directive is used to inform the assembler the names of the logical segments to be assumed for different segments used in the program .In the ALP each segment is given name.

Syntax: ASSUME segreg:segname,…segreg:segname

Ex: ASSUME CS:CODE

ASSUME CS:CODE,DS:DATA,SS:STACK

2. DB: Define Byte

The DB directive is used to reserve byte or bytes of memory locations in the available memory.

Syntax: Name of variable DB initialization value.

Ex: MARKS DB 35H,30H,35H,40H

NAME DB "VARDHAMAN"

3. **DW:** Define Word

The DW directive serves the same puposes as the DB directive,but it now makes the assembler reserve the number of memory words(16-bit) instead of bytes.

Syntax: variable name DW initialization values.

Ex: WORDS DW 1234H,4567H,2367H

WDATA DW 5 Dup(522h)

(or) Dup(?)

4. DD: Define Double:

The directive DD is used to define a double word (4bytes) variable.

Syntax: variablename DD 12345678H

Ex: Data1 DD 12345678H

5. DQ: Define Quad Word

This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

Syntax: Name of variable DQ initialize values.

Ex: Data1 DQ 123456789ABCDEF2H

6. DT: Define Ten Bytes

The DT directive directs the assembler to define the specified variable requiring 10 bytes for its storage and initialize the 10-bytes with the specified values.

Syntax: Name of variable DT initialize values.

Ex: Data1 DT 123456789ABCDEF34567H

7. END: End of Program

The END directive marks the end of an ALP. The statement after the directive END will be ignored by the assembler.

8. ENDP: End of Procedure

The ENDP directive is used to indicate the end of procedure. In the AL programming the subroutines are called procedures.

Ex: Procedure Start

:

Start ENDP

9. ENDS: End of segment

The ENDS directive is used to indicate the end of segment.

Ex: DATA SEGMENT

:

DATA ENDS

10.EVEN: Align on Even memory address

The EVEN directives updates the location counter to the next even address.

Ex: EVEN

Procedure Start

:

Start ENDP

- The above structure shows a procedure START that is to be aligned at an even address.

11.EQU: Equate

The directive EQU is used to assign a label with a value or symbol.

Ex: LABEL EQU 0500H

ADDITION EQU ADD

12.EXTRN: External and public

- The directive EXTRN informs the assembler that the names, procedures and labels declared after this directive have been already defined in some other AL modules.
- While in other module, where names, procedures and labels actually appear, they must be declared public using the PUBLIC directive.

Ex: MODULE1 SEGMENT

PUBLIC FACT FAR

MODULE1 ENDS

MODULE2 SEGMENT

EXTRN FACT FAR

MODULE2 END

13.GROUP: Group the related segments

This directive is used to form logical groups of segments with similar purpose or type.

Ex: PROGRAM GROUP CODE, DATA, STACK

*CODE, DATA and STACK segments lie within a 64KB memory segment that is named as PROGRAM.

14.LABEL: label

The label is used to assign name to the current content of the location counter.

Ex: CONTINUE LABEL FAR

The label CONTINUE can be used for a FAR jump, if the program contains the above statement.

15.LENGTH: Byte length of a label

This is used to refer to the length of a data array or a string

Ex : MOV CX, LENGTH ARRAY

16.LOCAL: The labels, variables, constant or procedures are declared LOCAL in a module are to be used only by the particular module.

Ex : LOCAL a, b, Data1, Array, Routine

17.NAME: logical name of a module

The name directive is used to assign a name to an assembly language program module. The module may now be refer to by its declared name.

Ex : Name “addition”

18.OFFSET: offset of a label

When the assembler comes across the OFFSET operator along with a label, it first computing the 16-bit offset address of a particular label and replace the string ‘OFFSET LABEL’ by the computed offset address.

Ex : MOV SI, offset list

19.ORG: origin

The ORG directive directs the assembler to start the memory allotment for the particular segment, block or code from the declared address in the ORG statement.

Ex: ORG 1000H

20.PROC: Procedure

The PROC directive marks the start of a named procedure in the statement.

Ex: RESULT PROC NEAR

ROUTINE PROC FAR

21.PTR: pointer

The PTR operator is used to declare the type of a label, variable or memory operator.

Ex : MOV AL, BYTE PTR [SI]

MOV BX, WORD PTR [2000H]

22.SEG: segment of a label

The SEG operator is used to decide the segment address of the label, variable or procedure.

Ex : MOV AX, SEG ARRAY

MOV DS, AX

23.SEGMENT: logical segment

The segment directive marks the starting of a logical segment

Ex: CODE SEGMENT

:

CODE ENDS

24.SHORT: The SHORT operator indicates to the assembler that only one byte is required to code the displacement for jump.

Ex : JMP SHORT LABEL

25.TYPE: The TYPE operator directs the assembler to decide the data type of the specified label and replaces the TYPE label by the decided data type.

For word variable, the data type is 2.

For double word variable, the data type is 4.

For byte variable, the data type is 1.

Ex : STRING DW 2345H, 4567H

MOV AX, TYPE STRING

AX=0002H

26.GLOBAL: The labels, variables, constants or procedures declared GLOBAL may be used by other modules of the program.

Ex : ROUTINE PROC GLOBAL.

27.FAR PTR: This directive indicates the assembler that the label following FAR PTR is not available within the same segment and the address of the label is of 32-bits i.e 2-bytes of offset followed by 2-bytes of segment address.

Ex : JMP FAR PTR LABEL

28.NEAR PTR: This directive indicates that the label following NEAR PTR is in the same segment and needs only 16-bit i.e 2-byte offset to address it

Ex : JMP NEAR PTR LABEL

CALL NEAR PTR ROUTINE

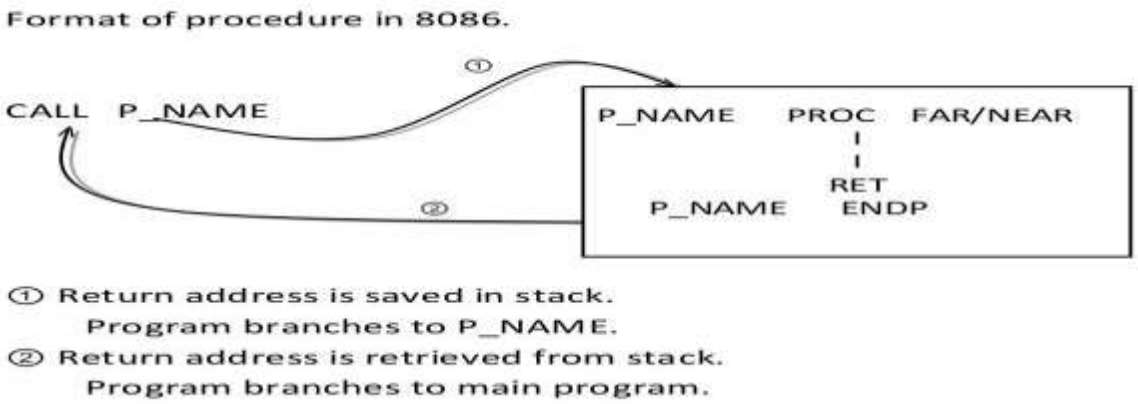
Procedures and Macros:

- When we need to use a group of instructions several times throughout a program there are two ways we can avoid having to write the group of instructions each time we want to use them.
  1. One way is to write the group of instructions as a separate **procedure.**
  2. Another way we can use **macros.**

Procedures:

- The procedure is a group of instructions stored as a separate program in the memory and it is called from the main program whenever required using CALL instruction.
- For calling the procedure we have to store the return address (next instruction address followed by CALL) onto the stack.
- At the end of the procedure RET instruction used to return the execution to the next instruction in the main program by retrieving the address from the top of the stack.

- Machine codes for the procedure instructions put only once in memory.
- The procedure can be defined anywhere in the program using assembly directives PROC and ENDP.



- **The four major ways of passing parameters to and from a procedure are:**
  1. In registers
  2. In dedicated memory location accessed by name
  3 .With pointers passed in registers
  4. With the stack
- The type of procedure depends on where the procedure is stored in the memory.
- If it is in the same code segment where the main program is stored the it is called near procedure otherwise it is referred to as far procedure.
- For near procedure CALL instruction pushes only the IP register contents on the stack, since CS register contents remains unchanged for main program.
- But for Far procedure CALL instruction pushes both IP and CS on the stack.

Syntax:

Procedure name PROC near

instruction 1

instruction 2

RET

Procedure name ENDP

Example:

near procedure:

ADD2 PROC near

ADD AX,BX

RET

ADD2 ENDP

far procedure:

Procedures segment

Assume CS : Procedures

ADD2 PROC far

ADD AX,BX

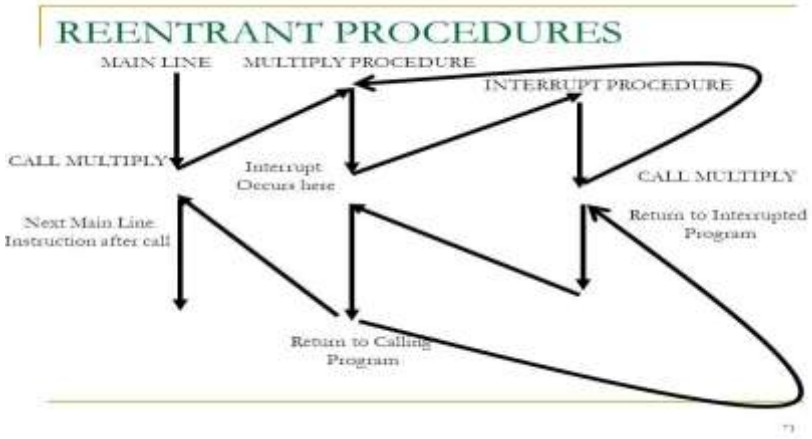
RET ADD2 ENDP

Procedures ends

- Depending on the characteristics the procedures are two types
  1. Re-entrant Procedures
  2. Recursive Procedures

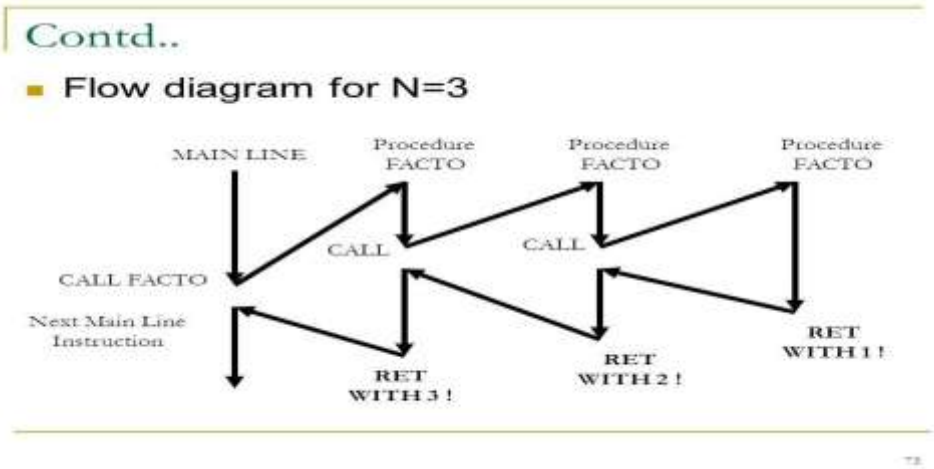
Reentrant Procedures

- The procedure which can be interrupted, used and "reentered" without losing or writing over anything.



Recursive Procedure

- A recursive procedure is procedure which calls itself.



-

ALP for Finding Factorial of number using procedures

```
CODE SEGMENT
ASSUME CS:CODE
START: MOV AX,7
CALL FACT
MOV AH,4CH
INT 21H
FACT PROC NEAR
MOV BX,AX
DEC BX
BACK: MUL BX
DEC BX
JNZ BACK
RET
ENDP
CODE ENDS
END START
```

Macros:

- A **macro** is a group of repetitive instructions in a program which are codified only once and can be used as many times as necessary.
- A macro can be defined anywhere in program using the directives **MACRO** and **ENDM**
- Each time we call the macro in a program, the assembler will insert the defined group of instructions in place of the call.
- The assembler generates machine codes for the group of instructions each time the macro is called.
- Using a macro avoids the overhead time involved in calling and returning from a procedure.

Syntax of macro:

```
macroname MACRO

instruction1

instruction2

.

.

ENDM
```

- **Example:**

```
Read MACRO
 mov ah,01h
int 21h
ENDM
```

```
Display MACRO
mov dl,al
 Mov ah,02h
int 21h
 ENDM
```

ALP for Finding Factorial of number using procedures

```
CODE SEGMENT
ASSUME CS:CODE
    FACT MACRO
     MOV BX,AX
     DEC BX
    BACK: MUL BX
     DEC BX
     JNZ BACK
    ENDM
START: MOV AX,7
     FACT
     MOV AH,4CH
     INT 21H
     CODE ENDS
     END START
```

Advantage of Procedure and Macros:

Procedures:

Advantages

- The machine codes for the group of instructions in the procedure only have to be put once.

Disadvantages

- Need for stack
- Overhead time required to call the procedure and return to the calling program.

Macros:

Advantages

- Macro avoids overhead time involving in calling and returning from a procedure.

Disadvantages

- Generating in line code each time a macro is called is that this will make the program take up more memory than using a procedure.

Differences between Procedures and Macros:

PROCEDURES	MACROS
Accessed by CALL and RET mechanism during program execution	Accessed by name given to macro when defined during assembly
Machine code for instructions only put in memory once	Machine code generated for instructions each time called
Parameters are passed in registers, memory locations or stack	Parameters passed as part of statement which calls macro
Procedures uses stack	Macro does not utilize stack
A procedure can be defined anywhere in program using the directives PROC and ENDP	A macro can be defined anywhere in program using the directives MACRO and ENDM
Procedures takes huge memory for CALL (3 bytes each time CALL is used) instruction	Length of code is very huge if macro's are called for more number of times

8086 MEMORY INTERFACING:

- Most the memory ICs are byte oriented i.e., each memory location can store only one byte of data.
- The 8086 is a 16-bit microprocessor, it can transfer 16-bit data.
- So in addition to byte, word (16-bit) has to be stored in the memory.
- To implement this , the entire memory is divided into two memory banks: Bank0 and Bank1.
- Bank0 is selected only when A0 is zero and Bank1 is selected only when BHE' is zero.
- A0 is zero for all even addresses, so Bank0 is usually referred as even addressed memory bank.
- BHE' is used to access higher order memory bank, referred to as odd addressed memory bank.

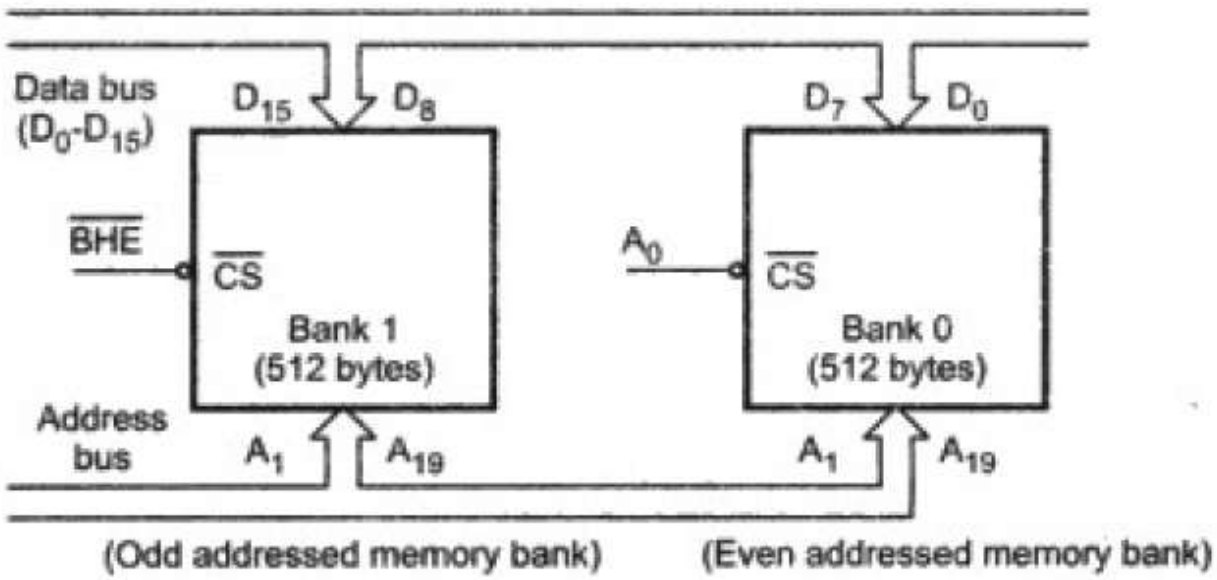


Fig. 5.2 Memory interfacing

- Every microprocessor based system has a memory system.
- Almost all systems contain two basic types of memory, read only memory (ROM) and random access memory (RAM) or read/write memory.
- ROM contains system software and permanent system data such as lookup tables, IVT..etc.
- RAM contains temporary data and application software.
- ROMs/PROMs/EPROMs are mapped to cover the CPU's reset address, since these are non-volatile.
- When the 8086 is reset, the next instruction is fetched from the memory location FFFF0H.
- So in the 8086 system the location FFFF0H must be in ROM location.

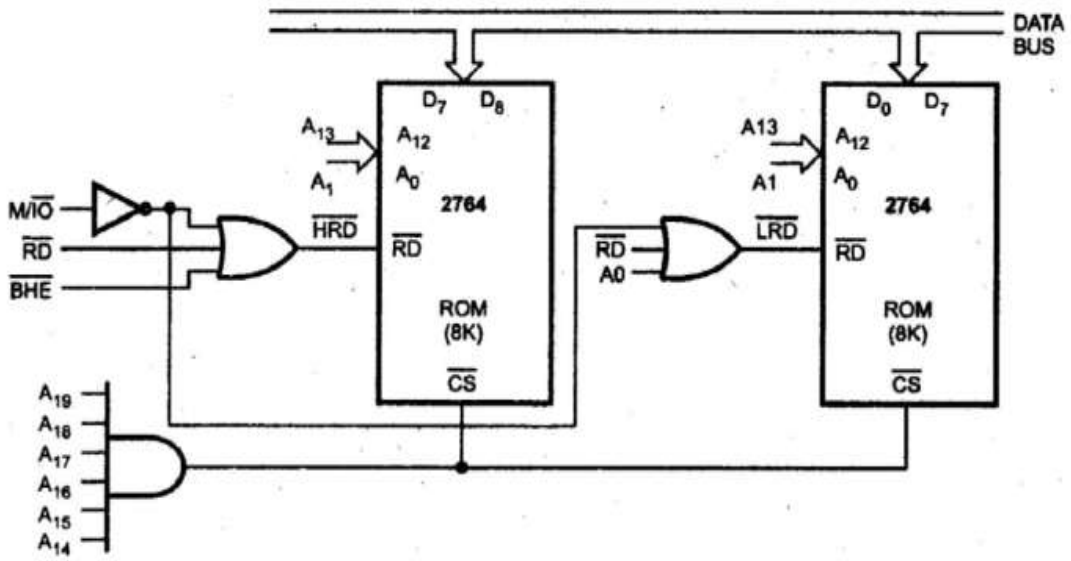
Address Decoding Techniques

1. Absolute decoding
2. Linear decoding
3. Block decoding

1. Absolute Decoding:

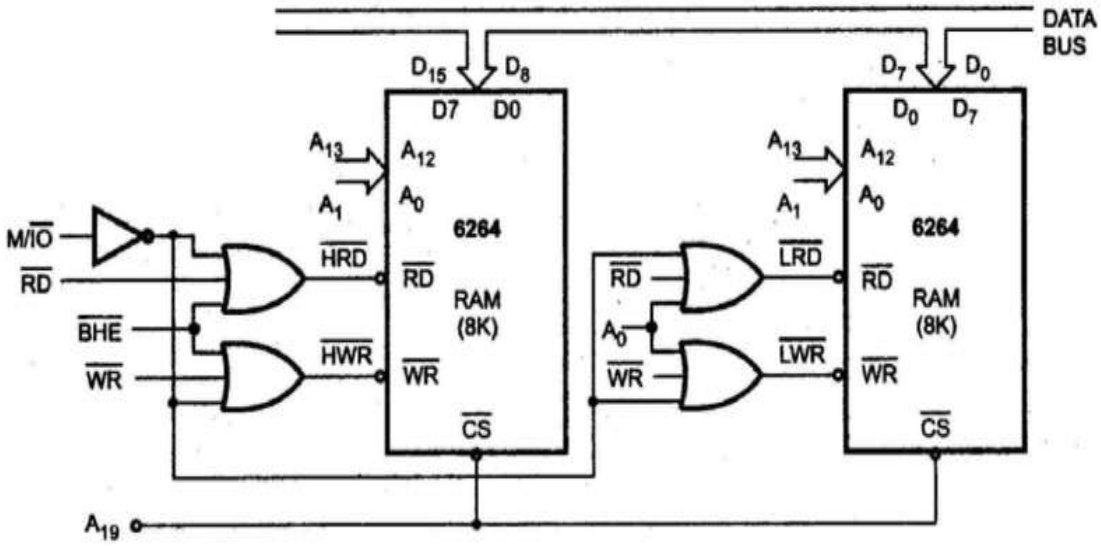
- In the absolute decoding technique the memory chip is selected only for the specified logic level on the address lines: no other logic levels can select the chip.
- Below figure the memory interface with absolute decoding. Two 8K EPROMs (2764) are used to provide even and odd memory banks.
- Control signals BHE and A0 are use to enable output of odd and even memory banks respectively. As each memory chip has 8K memory locations, thirteen address lines are required to address each locations, independently.

- All remaining address lines are used to generate an unique chip select signal. This address technique is normally used in large memory systems.



Linear Decoding:

In small system hardware for the decoding logic can be eliminated by using only required number of addressing lines (not all). Other lines are simple ignored. This technique is referred as linear decoding or partial decoding. Control signals BHE and Ao are used to enable odd and even memory banks, respectively. Figure shows the addressing of 16K RAM (6264) with linear decoding.

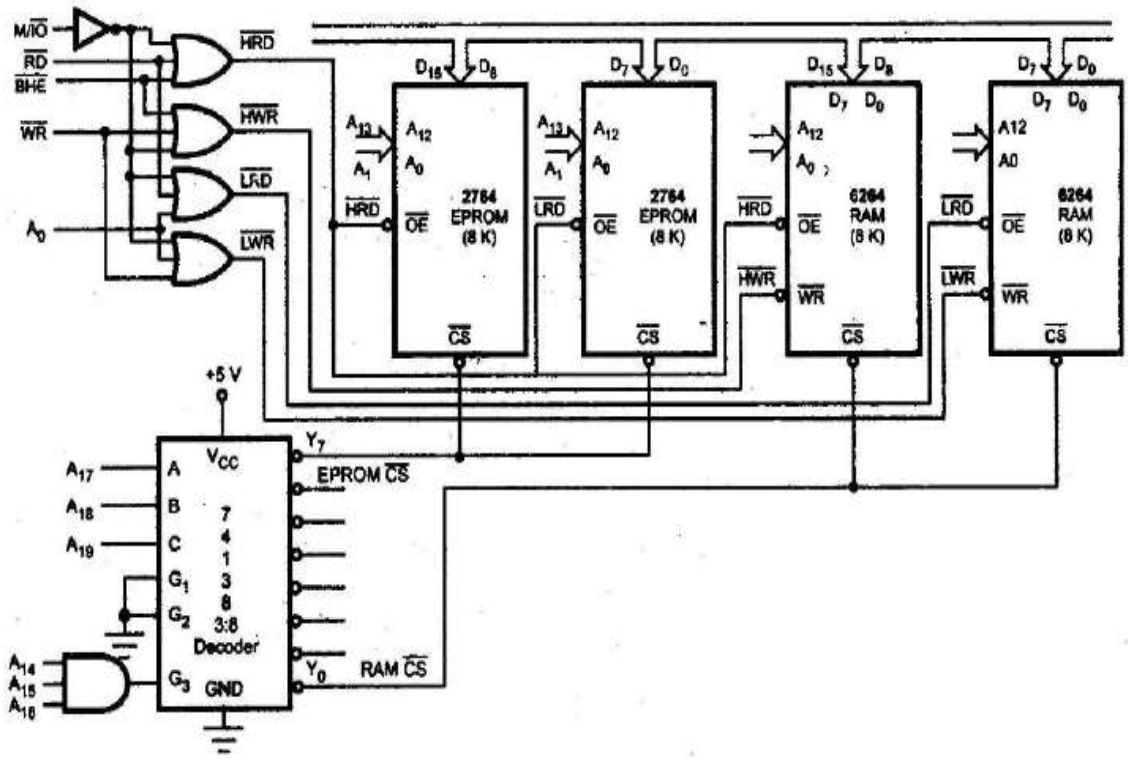


The address line A19 is used to select the RAM chips. When A19 is low, chip is selected, otherwise it is disabled. The status of A14 to A18 does not affect the chip selection logic. This gives you multiple addresses (shadow addresses). This technique reduces the cost of decoding circuit, but it gas drawback of multiple addresses.

Block Decoding:

In a microcomputer system the memory array is often consists of several blocks of memory chips. Each block of memory requires decoding circuit. To avoid separate decoding for each memory block special decoder IC is used to generate chip select signal for each block.

Figure shows the Block decoding technique using 74138, 3:8 decoder



Interfacing RAM, ROM, EPROM to 8086:

- The general procedure of static memory interfacing with 8086

1. Arrange the available memory chips so as to obtain 16-bit data bus width.
    - The upper 8-bit bank is called 'odd address memory bank'.
    - The lower 8-bit bank is called 'even address memory bank'.
2. Connect available memory address lines of memory chips with those of the microprocessor and also connect the RD and WR inputs to the corresponding processor control signals.
3. Connect the 16-bit data bus of memory bank with that of the microprocessor 8086.
4. The remaining address lines of the microprocessor, BHE and A_0 are used for decoding the required chip select signals for the odd and even memory banks. The CS of memory is derived from the output of the decoding circuit.

Problem 1:

Interface two 4Kx8 EPROM and two 4Kx8 RAM chips with 8086. Select suitable maps.

Solution:

We know that, after reset, the IP and CS are initialized to form address FFFF0H. Hence, this address must lie in the EPROM. The address of RAM may be selected anywhere in the 1MB address space of 8086, but we will select the RAM address such that the address map of the system is continuous.

Memory Map Table

Address	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A09	A08	A07	A06	A05	A04	A03	A02	A01	A00
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
EPROM								8K X 8												
FE000H	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
FDFFFH	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
RAM								8K X 8												
FC000H	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Total 8K bytes of EPROM need 13 address lines A0-A12 (since z13 = 8K).

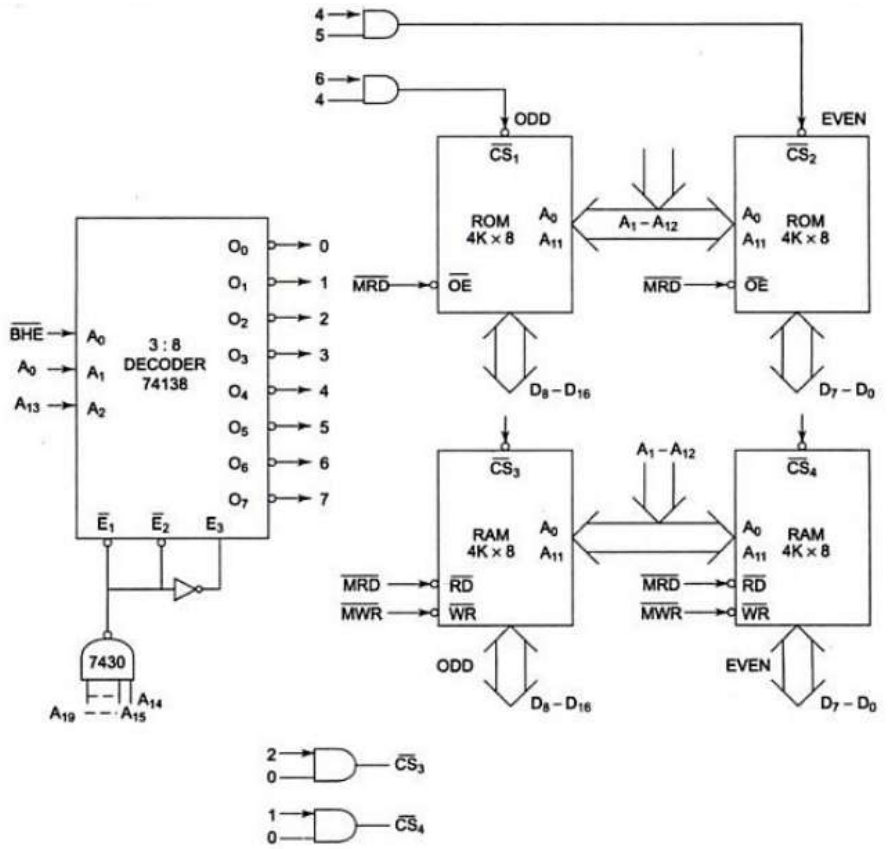
Address lines A13 - A19 are used for decoding to generate the chip select.

The $\overline{BHE}$ signal goes low when a transfer is at odd address or higher byte of data is to be accessed.

Let us assume that the latched address, $\overline{BHE}$ and demultiplexed data lines are readily available for interfacing.

The memory system in this problem contains in total four 4K x 8 memory chips.

The two 4K x 8 chips of RAM and ROM are arranged in parallel to obtain 16-bit data bus width. If A0 is 0, i.e., the address is even and is in RAM, then the lower RAM chip is selected indicating 8-bit transfer at an even address. If A0 is i.e., the address is odd and is in RAM, the $\overline{BHE}$ goes low, the upper RAM chip is selected, further indicating that the 8-bit transfer is at an odd address. If the selected addresses are in ROM, the respective ROM chips are selected. If at a time A0 and $\overline{BHE}$ both are 0, both the RAM or ROM chips are selected, i.e., the data transfer is of 16 bits. The selection of chips here takes place as shown in table below.



Memory Chip Selection Table:

Decoder I/P --> Address/$\overline{BHE}$ -->	A2 A13	A1 A0	A0 $\overline{BHE}$	Selection/ Comment
Word transfer on D0 - D15	0	0	0	Even and odd address in RAM
Byte transfer on D7 - D0	0	0	1	Only even address in RAM
Byte transfer on D8 - D15	0	1	0	Only odd address in RAM
Word transfer on D0 - D15	1	0	0	Even and odd address in RAM
Byte transfer on D7 - D0	1	0	1	Only even address in RAM
Byte transfer on D8 - D15	1	1	0	Only odd address in ROM

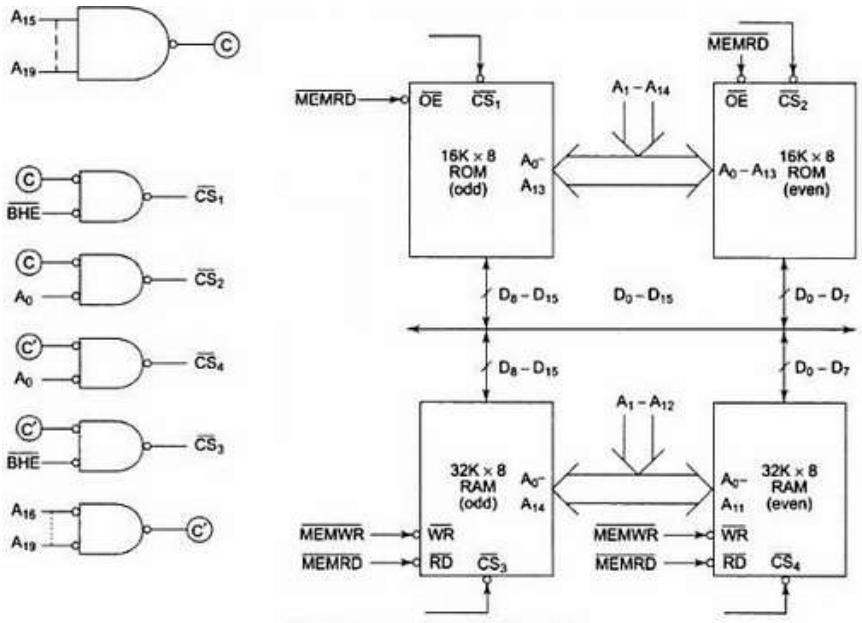
Problem2: Design an interface between 8086 CPU and two chips of 16K×8 EPROM and two chips of 32K×8 RAM. Select the starting address of EPROM suitably. The RAM address must start at 00000 H.

Solution: The last address in the map of 8086 is FFFFF H. after resetting, the processor starts from FFFF0 H. hence this address must lie in the address range of EPROM.

Address Map for Problem

Addresses	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
						32KB EPROM														
F8000H	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0FFFFH	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
						64KB RAM														
00000H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

It is better not to use a decoder to implement the above map because it is not continuous, i.e. there is some unused address space between the last RAM address (0FFFF H) and the first EPROM address (F8000 H). Hence the logic is implemented using logic gates.



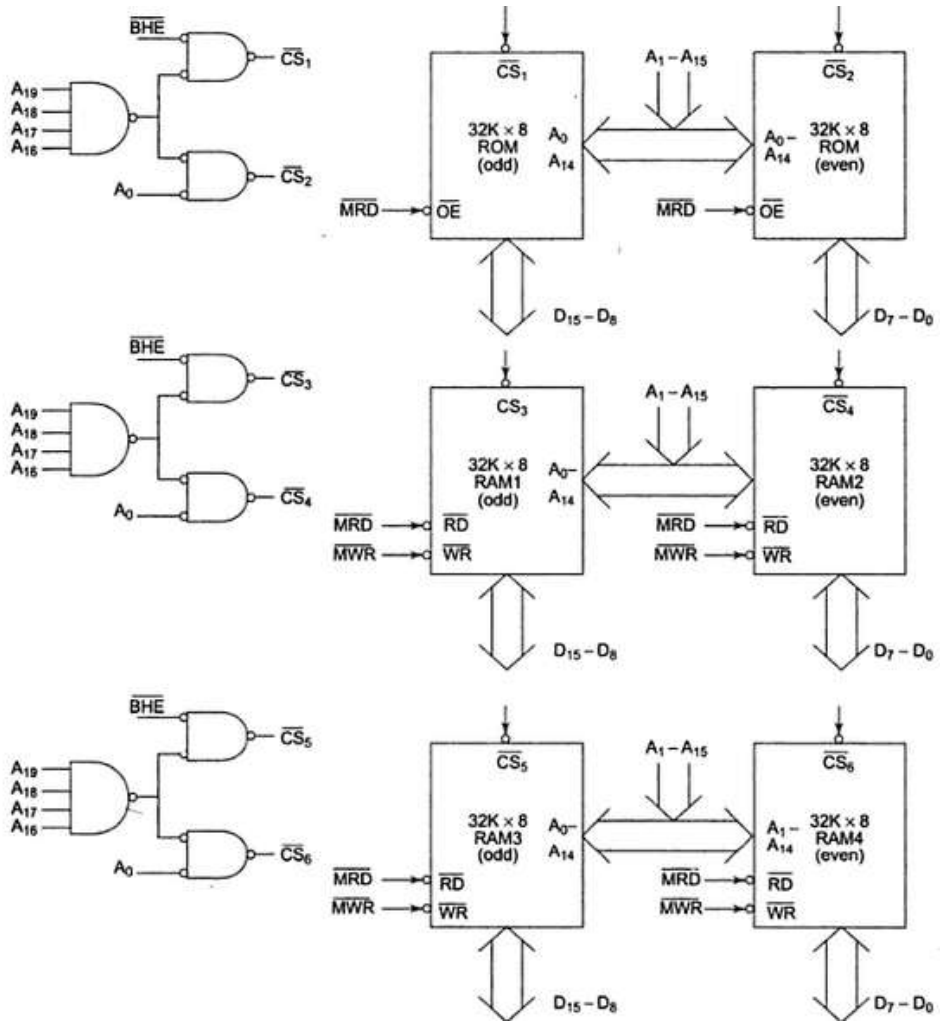
Problem3: It is required to interface two chips of 32K×8 ROM and four chips of 32K×8 RAM with 8086, according to following map.

ROM 1 and ROM 2 F0000H - FFFFFH, RAM 1 and RAM 2 D0000H - DFFFFH, RAM 3 and RAM 4 E0000H - EFFFFH. Show the implementation of this memory system.

Solution:

Address Map for Problem

Address	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{0}
F0000H	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ROM 1and2										64K										
FFFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
D0000H	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RAM 1and2										64K										
DFFFFH	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
E0000H	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RAM 3and4										64K										
EFFFFH	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



Methods of Interfacing I/O Devices

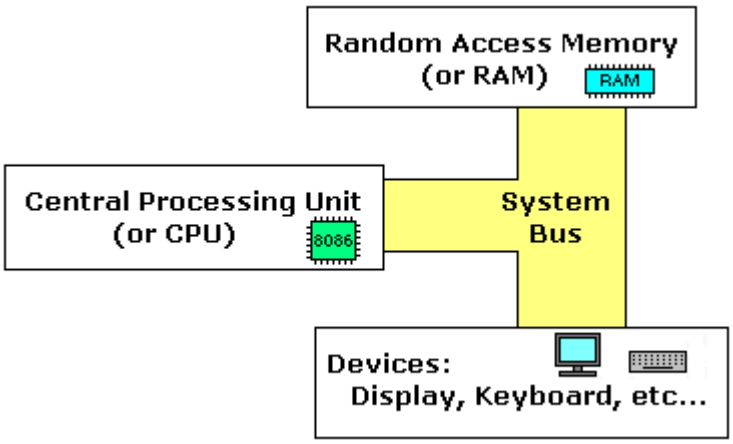
Memory Mapping	IO mapping
1. 20-bit addresses are provided for IO devices.	1. 8-bit or 16-bit address are provided for IO devices
2. The IO ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transfer.	2. Only IN and OUT instructions can be used for data transfer between IO device and the processor.
3. In memory mapped ports, the data can be moved from any register to port and vice versa	3. In IO mapped ports, the data transfer can take only between the accumulator and the ports
4. When memory mapping is used for IO devices, the full memory address space cannot be used for addressing memory.	4. When IO mapping is used for IO devices, then the full address space can be used for addressing memory.

8086 assembler tutorial for beginners (part 1)

This tutorial is intended for those who are not familiar with assembler at all, or have a very distant idea about it. of course if you have knowledge of some other programming language (basic, c/c++, pascal...) that may help you a lot. but even if you are familiar with assembler, it is still a good idea to look through this document in order to study emu8086 syntax.
It is assumed that you have some knowledge about number representation (hex/bin), if not it is highly recommended to study **numbering systems tutorial** before you proceed.

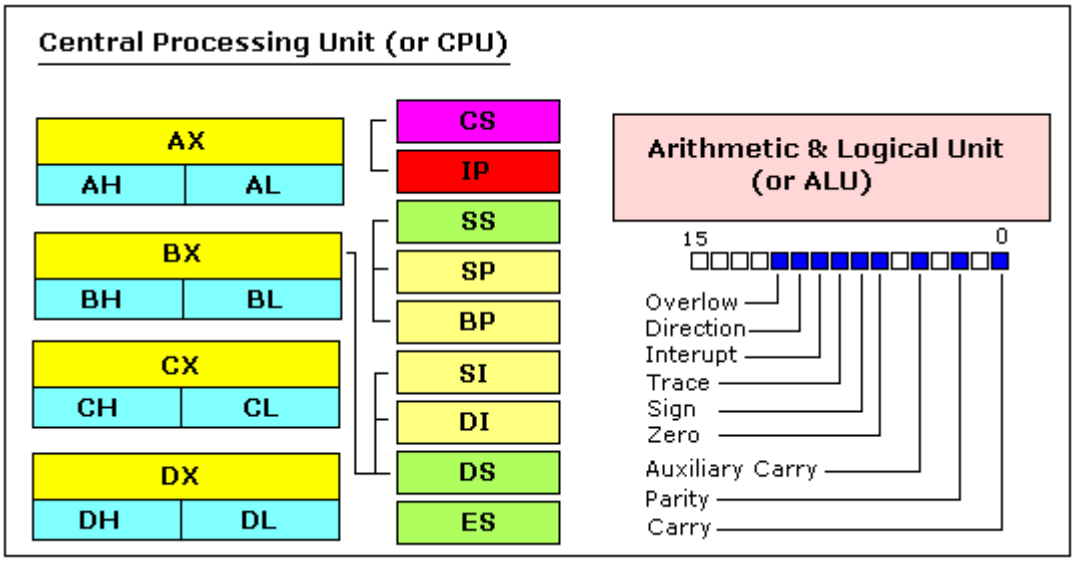
what is assembly language?

assembly language is a low level programming language. you need to get some knowledge about computer structure in order to understand anything. the simple computer model as i see it:



the **system bus** (shown in yellow) connects the various components of a computer. the **CPU** is the heart of the computer, most of computations occur inside the **CPU**.
RAM is a place to where the programs are loaded in order to be executed.

inside the cpu



general purpose registers
8086 CPU has 8 general purpose registers, each register has its own name:

- **AX** - the accumulator register (divided into **AH / AL**).
- **BX** - the base address register (divided into **BH / BL**).
- **CX** - the count register (divided into **CH / CL**).
- **DX** - the data register (divided into **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

despite the name of a register, it's the programmer who determines the usage for each general purpose register. the main purpose of a register is to keep a number (variable). the size of the above registers is 16 bit, it's something like: **0011000000111001b** (in binary form), or **12345** in decimal (human) form.

4 general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= **0011000000111001b**, then AH=**00110000b** and AL=**00111001b**. therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. the same is for other 3 registers, "H" is for high and "L" is for low part.

because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. therefore, you should try to keep variables in the registers. register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

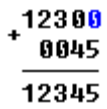
segment registers

- **CS** - points at the segment containing the current program.
- **DS** - generally points at segment where variables are defined.
- **ES** - extra segment register, it's up to a coder to define its usage.
- **SS** - points at the segment containing the stack.

although it is possible to store any data in the segment registers, this is never a good idea. the segment registers have a very special purpose - pointing at accessible blocks of memory.

segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, since this way we can access much more memory than with a single register that is limited to 16 bit values.
CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h * 10h + 45h

= 12345h):



the address formed with 2 registers is called an **effective address**.
by default **BX, SI** and **DI** registers work with **DS** segment register;
BP and **SP** work with **SS** segment register.
other general purpose registers cannot form an effective address!
also, although **BX** can form an effective address, **BH** and **BL** cannot.

special purpose registers

- **IP** - the instruction pointer.
- **flags register** - determines the current state of the microprocessor.

IP register always works together with **CS** segment register and it points to currently executing instruction.
flags register is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.
generally you cannot access these registers directly, the way you can access AX and other general registers, but it is possible to change values of system registers using some tricks that you will learn a little bit later.

Memory Access

---

to access memory we can use these four registers: **BX, SI, DI, BP**. combining these registers inside **[ ]** symbols, we can get different memory locations. these combinations are supported (addressing modes):

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

d8 - stays for 8 bit signed immediate displacement (for example: 22, 55h, -1, etc...)

d16 - stays for 16 bit signed immediate displacement (for example: 300, 5517h, -259, etc...).

displacement can be a immediate value or offset of a variable, or even both. if there are several values, assembler evaluates all values and calculates a single immediate value..

displacement can be inside or outside of the **[ ]** symbols, assembler generates the same machine code for both ways.

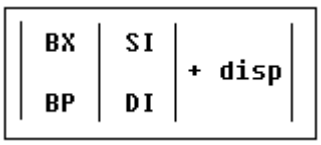
displacement is a **signed** value, so it can be both positive or negative.

generally the compiler takes care about difference between **d8** and **d16**, and generates the required machine code.

for example, let's assume that **DS = 100**, **BX = 30**, **SI = 70**.
The following addressing mode: **[BX + SI] + 25**
is calculated by processor to this physical address: **100 * 16 + 30 + 70 + 25 = 1725**.

by default **DS** segment register is used for all modes except those with **BP** register, for these **SS** segment register is used.

there is an easy way to remember all those possible combinations using this chart:



you can form all valid combinations by taking only one item from each column or skipping the column by not taking anything from it. as you see **BX** and **BP** never go together. **SI** and **DI** also don't go together. here are an examples of a valid addressing modes: **[BX+5]** , **[BX+SI]** , **[DI+BX-4]**

---

the value in segment register (CS, DS, SS, ES) is called a **segment**,
and the value in purpose register (BX, SI, DI, BP) is called an **offset**.
When DS contains value **1234h** and SI contains the value **7890h** it can be also recorded as **1234:7890**. The physical address will be 1234h * 10h + 7890h = 19BD0h.

if zero is added to a decimal number it is multiplied by 10, however **10h = 16**, so if zero is added to a hexadecimal value, it is multiplied by 16, for example:

7h = 7
70h = 112

---

in order to say the compiler about data type,
these prefixes should be used:

byte ptr - for byte.
word ptr - for word (two bytes).

for example:

```
byte ptr [BX]    ; byte access.
   or
word ptr [BX]    ; word access.
```

assembler supports shorter prefixes as well:

b. - for **byte ptr**
w. - for **word ptr**

in certain cases the assembler can calculate the data type automatically.

MOV instruction

- copies the **second operand** (source) to the **first operand** (destination).
- the source operand can be an immediate value, general-purpose register or memory location.
- the destination register can be a general-purpose register, or memory location.
- both operands must be the same size, which can be a byte or a word.

these types of operands are supported:

MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

for segment registers only these types of **MOV** are supported:

MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
SREG: DS, ES, SS, and only as second operand: CS.

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

The **MOV** instruction cannot be used to set the value of the **CS** and **IP** registers.

here is a short program that demonstrates the use of **MOV** instruction:

```
ORG 100h          ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h    ; set AX to hexadecimal value of B800h.
MOV DS, AX        ; copy value of AX to DS.
MOV CL, 'A'       ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh      ; set BX to 15Eh.
MOV [BX], CX      ; copy contents of CX to memory at B800:015E
RET               ; returns to operating system.
```

you can **copy & paste** the above program to emu8086 code editor, and press [**Compile and Emulate**] button (or press **F5** key on your keyboard).

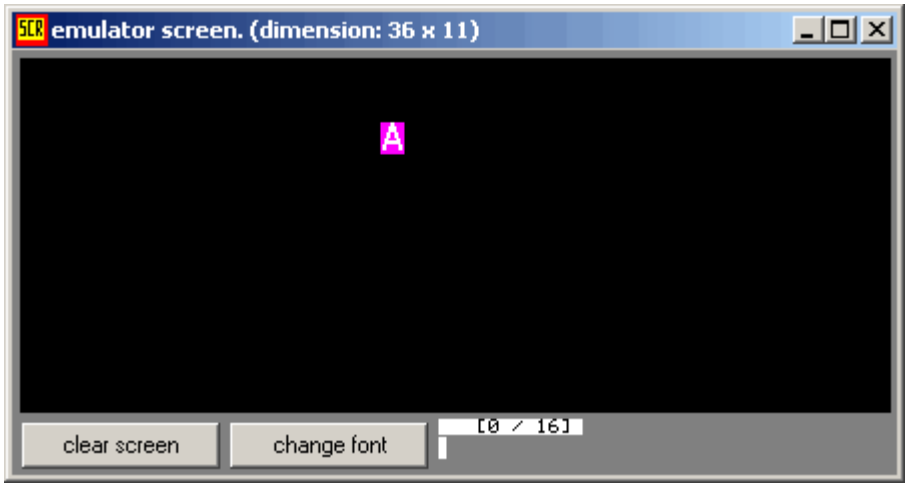
the emulator window should open with this program loaded, click [**Single Step**] button and watch the register values.

how to do **copy & paste**:

1. select the above text using mouse, click before the text and drag it down until everything is selected.
2. press **Ctrl + C** combination to copy.
3. go to emu8086 source editor and press **Ctrl + V** combination to paste.

as you may guess, "**;**" is used for comments, anything after "**;**" symbol is ignored by compiler.

you should see something like that when program finishes:



actually the above program writes directly to video memory, so you may see that **MOV** is a very powerful instruction

Variables

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "**var1**" then at the address 5A73:235B, especially when you have 10 or more variables.

Our compiler supports two types of variables: **BYTE** and **WORD**.

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

DB - stays for Define Byte.
DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

As you probably know from *part 2* of this tutorial, **MOV** instruction is used to copy values from source to destination.
Let's see another example with **MOV** instruction:

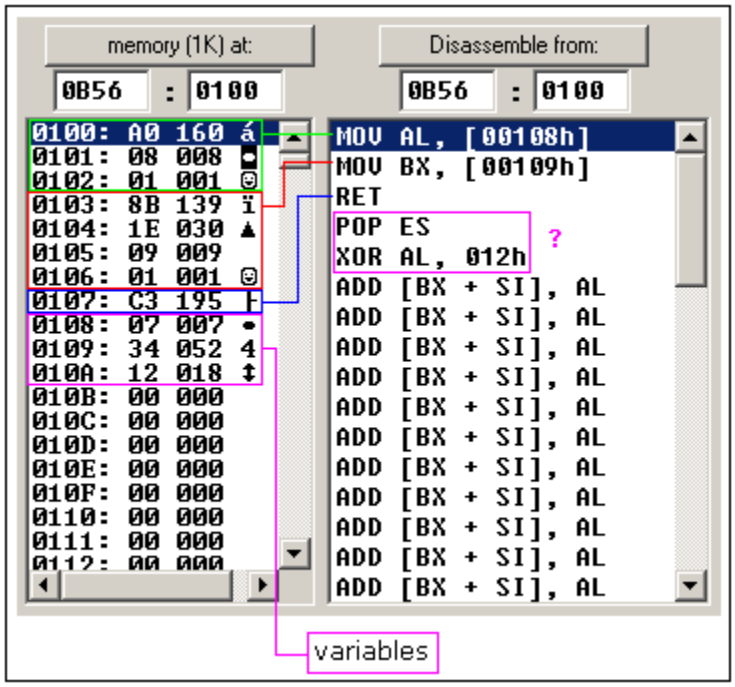
```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; stops the program.

VAR1 DB 7
var2 DW 1234h
```

Copy the above code to emu8086 source editor, and press **F5** key to compile and load it in the emulator. You should get something like:



As you see this looks a lot like our example, except that variables are replaced with actual memory locations. When compiler makes machine code, it automatically replaces all variable names with their **offsets**. By default segment is loaded in **DS** register (when **COM** files is loaded the value of **DS** register is set to the same value as **CS** register - code segment).

In memory list first row is an **offset**, second row is a **hexadecimal value**, third row is **decimal value**, and last row is an **ASCII** character value.

Compiler is not case sensitive, so "**VAR1**" and "**var1**" refer to the same variable.

The offset of **VAR1** is **0108h**, and full address is **0B56:0108**.

The offset of **var2** is **0109h**, and full address is **0B56:0109**, this variable is a **WORD** so it occupies **2 BYTES**. It is assumed that low byte is stored at lower address, so **34h** is located before **12h**.

You can see that there are some other instructions after the **RET** instruction, this happens because disassembler has no idea about where the data starts, it just processes the values in memory and it understands them as valid 8086 instructions (we will learn them later).
You can even write the same program using **DB** directive only:

```
ORG 100h  ; just a directive to make a simple .com file
(expands into no code).

DB 0A0h
DB 08h
DB 01h
```

```
DB 8Bh
DB 1Eh
DB 09h
DB 01h

DB 0C3h

DB 7

DB 34h
DB 12h
```

Copy the above code to emu8086 source editor, and press **F5** key to compile and load it in the emulator. You should get the same disassembled code, and the same functionality!

As you may guess, the compiler just converts the program source to the set of bytes, this set is called **machine code**, processor understands the **machine code** and executes it.

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.
Why executable file is loaded at **offset** of **100h**? Operating system keeps some data about the program in the first 256 bytes of the **CS** (code segment), such as command line parameters and etc.
Though this is true for **COM** files only, **EXE** files are loaded at offset of **0000**, and generally use special segment for variables. Maybe we'll talk more about **EXE** files later.

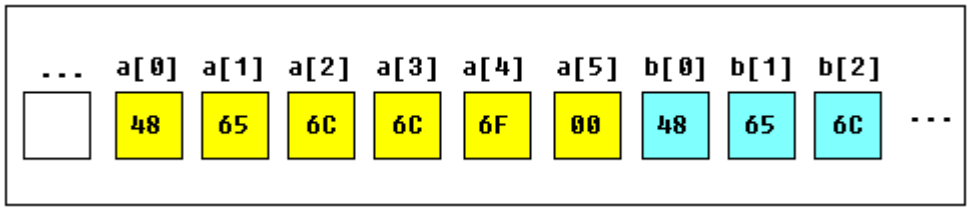
Arrays

Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

b is an exact copy of the *a* array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:



You can access the value of any element in array using square brackets, for example:
MOV AL, a[3]

You can also use any of the memory index registers **BX, SI, DI, BP**, for example:
MOV SI, 3
MOV AL, a[SI]

If you need to declare a large array you can use **DUP** operator.
The syntax for **DUP**:

number DUP (value(s))
number - number of duplicate to make (any constant value).
value - expression that DUP will duplicate.

for example:
c DB 5 DUP(9)
is an alternative way of declaring:
c DB 9, 9, 9, 9, 9

one more example:
d DB 5 DUP(1, 2)
is an alternative way of declaring:
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2

Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings.

Getting the Address of a Variable

There is **LEA** (Load Effective Address) instruction and alternative **OFFSET** operator. Both **OFFSET** and **LEA** can be used to get the offset address of the variable.
LEA is more powerful because it also allows you to get the address of an indexed variables. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

Reminder:

In order to tell the compiler about data type,
these prefixes should be used:

BYTE PTR - for byte.
WORD PTR - for word (two bytes).

For example:

```
BYTE PTR [BX]     ; byte access.
   or
WORD PTR [BX]     ; word access.
```

emu8086 supports shorter prefixes as well:

b. - for **BYTE PTR**
w. - for **WORD PTR**

in certain cases the assembler can calculate the data type automatically.

Here is first example:

```
ORG 100h

MOV    AL, VAR1            ; check value of VAR1 by
moving it to AL.

LEA    BX, VAR1            ; get address of VAR1 in BX.

MOV    BYTE PTR [BX], 44h    ; modify the contents of
VAR1.

MOV    AL, VAR1            ; check value of VAR1 by
moving it to AL.

RET

VAR1   DB  22h

END
```

Here is another example, that uses **OFFSET** instead of **LEA**:

```
ORG 100h

MOV    AL, VAR1            ; check value of VAR1 by
moving it to AL.

MOV    BX, OFFSET VAR1       ; get address of VAR1
in BX.

MOV    BYTE PTR [BX], 44h    ; modify the contents of
VAR1.
```

```
MOV    AL, VAR1          ; check value of VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Both examples have the same functionality.

These lines:
LEA BX, VAR1
MOV BX, OFFSET VAR1
are even compiled into the same machine code: MOV BX, num
num is a 16 bit value of the variable offset.

Please note that only these registers can be used inside square brackets (as memory pointers): **BX, SI, DI, BP**!
(see previous part of the tutorial).

Constants

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants **EQU** directive is used:

name EQU < any expression >

For example:

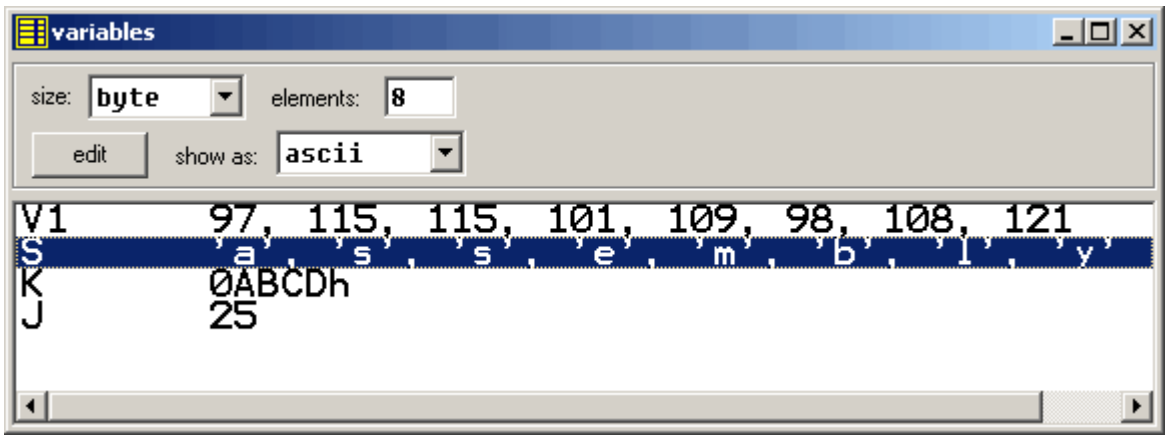
```
k EQU 5

MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

You can view variables while your program executes by selecting "**Variables**" from the "**View**" menu of emulator.



To view arrays you should click on a variable and set **Elements** property to array size. In assembly language there are not strict data types, so any variable can be presented as an array.

Variable can be viewed in any numbering system:

- **HEX** - hexadecimal (base 16).
- **BIN** - binary (base 2).
- **OCT** - octal (base 8).
- **SIGNED** - signed decimal (base 10).
- **UNSIGNED** - unsigned decimal (base 10).
- **CHAR** - ASCII char code (there are 256 symbols, some symbols are invisible).

You can edit a variable's value when your program is running, simply double click it, or select it and click **Edit** button.

It is possible to enter numbers in any system, hexadecimal numbers should have "**h**" suffix, binary "**b**" suffix, octal "**o**" suffix, decimal numbers require no suffix. String can be entered this way:
'hello world', 0
(this string is zero terminated).

Arrays may be entered this way:
1, 2, 3, 4, 5
(the array can be array of bytes or words, it depends whether **BYTE** or **WORD** is selected for edited variable).

Expressions are automatically converted, for example:
when this expression is entered:
5 + 2
it will be converted to **7** etc...

Interrupts

Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character you can simply call the interrupt and it will do everything for you. There are also interrupt functions that work with disk drive and other hardware. We call such functions **software interrupts**.

Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.

To make a **software interrupt** there is an **INT** instruction, it has very simple syntax:
INT value
Where **value** can be a number between 0 to 255 (or 0 to 0FFh),
generally we will use hexadecimal numbers.
You may think that there are only 256 functions, but that is not correct. Each interrupt may have sub-functions.
To specify a sub-function **AH** register should be set before calling interrupt.
Each interrupt may have up to 256 sub-functions (so we get 256 * 256 = 65536 functions). In general **AH** register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.

The following example uses **INT 10h** sub-function **0Eh** to type a "Hello!" message. This functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary.

```
ORG    100h ; directive to make a simple .com file.

; The sub-function that we are using
; does not modify the AH register on
; return, so we may set it only once.

MOV    AH, 0Eh    ; select sub-function.

; INT 10h / 0Eh sub-function
; receives an ASCII code of the
```

```
; character that will be printed
; in AL register.

MOV    AL, 'H'    ; ASCII code: 72
INT    10h        ; print it!

MOV    AL, 'e'    ; ASCII code: 101
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'o'    ; ASCII code: 111
INT    10h        ; print it!

MOV    AL, '!'    ; ASCII code: 33
INT    10h        ; print it!

RET               ; returns to operating system.
```

Copy & paste the above program to emu8086 source code editor, and press [**Compile and Emulate**] button. Run it!

See **list of supported interrupts** for more information about interrupts.

Library of common functions - emu8086.inc

To make programming easier there are some common functions that can be included in your program. To make your program use functions defined in other file you should use the **INCLUDE** directive followed by a file name. Compiler automatically searches for the file in the same folder where the source file is located, and if it cannot find the file there - it searches in **Inc** folder.

Currently you may not be able to fully understand the contents of the **emu8086.inc** (located in **Inc** folder), but it's OK, since you only need to understand what it can do.

To use any of the functions in **emu8086.inc** you should have the following line in the beginning of your source file:

```
include 'emu8086.inc'
```

---

emu8086.inc defines the following **macros**:

- **PUTC char** - macro with 1 parameter, prints out an ASCII char at current cursor position.
- **GOTOXY col, row** - macro with 2 parameters, sets cursor position.
- **PRINT string** - macro with 1 parameter, prints out a string.
- **PRINTN string** - macro with 1 parameter, prints out a string. The same as PRINT but automatically adds "carriage return" at the end of the string.
- **CURSOROFF** - turns off the text cursor.
- **CURSORON** - turns on the text cursor.

To use any of the above macros simply type its name somewhere in your code, and if required parameters, for example:

```
include emu8086.inc

ORG    100h

PRINT 'Hello World!'

GOTOXY 10, 5

PUTC 65          ; 65 - is an ASCII code for 'A'
PUTC 'B'

RET              ; return to operating system.
END              ; directive to stop the compiler.
```

When compiler process your source code it searches the **emu8086.inc** file for declarations of the macros and replaces the macro names with real code. Generally macros are relatively small parts of code, frequent use of a macro may make your executable too big (procedures are better for size optimization).

---

emu8086.inc also defines the following **procedures**:

- **PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.
- **PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction. For example:

```
CALL PTHIS
db 'Hello World!', 0
```

To use it declare: **DEFINE_PTHIS** before **END** directive.

- **GET_STRING** - procedure to get a null terminated string from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. Procedure stops the input when 'Enter' is pressed. To use it declare: **DEFINE_GET_STRING** before **END** directive.
- **CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare: **DEFINE_CLEAR_SCREEN** before **END** directive.
- **SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare: **DEFINE_SCAN_NUM** before **END** directive.
- **PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END** directive.
- **PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.

To use any of the above procedures you should first declare the function in the bottom of your file (but before the **END** directive), and then use **CALL** instruction followed by a procedure name. For example:

```
include 'emu8086.inc'

ORG    100h

LEA    SI, msg1       ; ask for the number
CALL   print_string   ;
CALL   scan_num       ; get number in CX.

MOV    AX, CX         ; copy the number to AX.

; print the following string:
CALL   pthis
DB  13, 10, 'You have entered: ', 0

CALL   print_num      ; print number in AX.

RET                   ; return to operating system.

msg1   DB  'Enter the number: ', 0

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
```

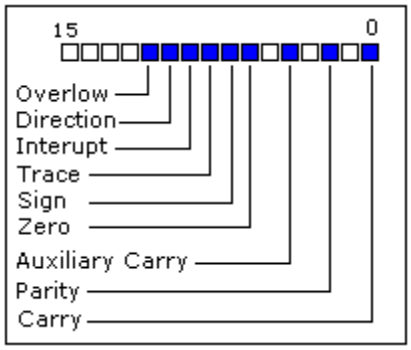
```
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required for print_num.
DEFINE_PTHIS

END                   ; directive to stop the compiler.
```

First compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to **CALL** instruction it replaces the procedure name with the address of the code where the procedure is declared. When **CALL** instruction is executed control is transferred to procedure. This is quite useful, since even if you call the same procedure 100 times in your code you will still have relatively small executable size. Seems complicated, isn't it? That's ok, with the time you will learn more, currently it's required that you understand the basic principle.

Arithmetic and logic instructions

Most Arithmetic and Logic Instructions affect the processor status register (or **Flags**)



As you may see there are 16 bits in this register, each bit is called a **flag** and can take a value of **1** or **0**.

- **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**. For example when you add bytes **255 + 1** (result is not in range 0...255). When there is no overflow this flag is set to **0**.
- **Zero Flag (ZF)** - set to **1** when result is **zero**. For none zero result this flag is set to **0**.
- **Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. Actually this flag take the value of the most significant bit.
- **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).
- **Parity Flag (PF)** - this flag is set to **1** when there is even number of one bits in result, and to **0** when there is odd number of one bits. Even if result is a word only 8 low bits are analyzed!
- **Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).
- **Interrupt enable Flag (IF)** - when this flag is set to **1** CPU reacts to interrupts from external devices.
- **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward, when this flag is set to **1** the processing is done backward.

There are 3 groups of instructions.

First group: **ADD**, **SUB**,**CMP**, **AND**, **TEST**, **OR**, **XOR**

These types of operands are supported:

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

After operation between operands, result is always stored in first operand. **CMP** and **TEST** instructions affect flags only and do not store a result (these instruction are used to make decisions during program execution).

These instructions affect these flags only:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **ADD** - add second operand to first.
- **SUB** - Subtract second operand to first.
- **CMP** - Subtract second operand from first **for flags only**.
- **AND** - Logical AND between all bits of two operands. These rules apply:

  1 AND 1 = 1
  1 AND 0 = 0
  0 AND 1 = 0
  0 AND 0 = 0

  As you see we get **1** only when both bits are **1**.

- **TEST** - The same as **AND** but **for flags only**.
- **OR** - Logical OR between all bits of two operands. These rules apply:

  1 OR 1 = 1
  1 OR 0 = 1
  0 OR 1 = 1
  0 OR 0 = 0

As you see we get **1** every time when at least one of the bits is **1**.

- **XOR** - Logical XOR (exclusive OR) between all bits of two operands. These rules apply:

  1 XOR 1 = 0
  1 XOR 0 = 1
  0 XOR 1 = 1
  0 XOR 0 = 0

  As you see we get **1** every time when bits are different from each other.

Second group: **MUL**, **IMUL**, **DIV**, **IDIV**

These types of operands are supported:

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

MUL and **IMUL** instructions affect these flags only:
CF, **OF**
When result is over operand size these flags are set to **1**, when result fits in operand size these flags are set to **0**.

For **DIV** and **IDIV** flags are undefined.

- **MUL** - Unsigned multiply:

  when operand is a **byte**:
  AX = AL * operand.

  when operand is a **word**:
  (DX AX) = AX * operand.

- **IMUL** - Signed multiply:

  when operand is a **byte**:
  AX = AL * operand.

  when operand is a **word**:
  (DX AX) = AX * operand.

- **DIV** - Unsigned divide:

when operand is a **byte**:
AL = AX / operand
AH = remainder (modulus). .

when operand is a **word**:
AX = (DX AX) / operand
DX = remainder (modulus). .

- **IDIV** - Signed divide:

when operand is a **byte**:
AL = AX / operand
AH = remainder (modulus). .

when operand is a **word**:
AX = (DX AX) / operand
DX = remainder (modulus). .

Third group: **INC**, **DEC**, **NOT**, **NEG**

These types of operands are supported:

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

INC, **DEC** instructions affect these flags only:
ZF, **SF**, **OF**, **PF**, **AF**.

NOT instruction does not affect any flags!

NEG instruction affects these flags only:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **NOT** - Reverse each bit of operand.
- **NEG** - Make operand negative (two's complement). Actually it reverses each bit of operand and then adds 1 to it. For example 5 will become -5, and -2 will become 2.

program flow control

Controlling the program flow is a very important thing, this is where your program can make decisions according to certain conditions.

- **unconditional jumps**

  The basic instruction that transfers control to another point in the program is **JMP**.

  The basic syntax of **JMP** instruction:

  JMP label

  To declare a *label* in your program, just type its name and add "**:**" to the end, label can be any character combination but it cannot start with a number, for example here are 3 legal label definitions:

  label1:
  label2:
  a:

  Label can be declared on a separate line or before any other instruction, for example:

  x1:
  MOV AX, 1

  x2: MOV AX, 2

  here's an example of **JMP** instruction:

```
org   100h

mov   ax, 5       ; set ax to 5.
mov   bx, 2       ; set bx to 2.

jmp   calc        ; go to 'calc'.

back: jmp stop    ; go to 'stop'.

calc:
add   ax, bx      ; add bx to ax.
jmp   back        ; go to 'back'.

stop:

ret               ; return to operating system.
```

Of course there is an easier way to calculate the some of two numbers, but it's still a good example of **JMP** instruction.
As you can see from this example **JMP** is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).

- **Short Conditional Jumps**

  Unlike **JMP** instruction that does an unconditional jump, there are instructions that do a conditional jumps (jump only when some conditions are in act). These instructions are divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned.

  Jump instructions that test single flag

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

-

  as you may already notice there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile **JE** instruction - you will get it disassembled as: **JZ**, **JC** is assembled the same as **JB** etc...

different names are used to make programs easier to understand, to code and most importantly to remember. very offset dissembler has no clue what the original instruction was look like that's why it uses the most common name.

if you emulate this code you will see that all instructions are assembled into **JNB**, the operational code (opcode) for this instruction is **73h** this instruction has fixed length of two bytes, the second byte is number of bytes to add to the **IP** register if the condition is true. because the instruction has only 1 byte to keep the offset it is limited to pass control to -128 bytes back or 127 bytes forward, this value is always signed.

```
    jnc a
    jnb a
    jae a

    mov ax, 4
a:  mov ax, 5
    ret
```

Jump instructions for signed numbers

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

<> - sign means not equal.

Jump instructions for unsigned numbers

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

-

Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).

The logic is very simple, for example:
it's required to compare 5 and 2,
5 - 2 = 3
the result is not zero (Zero Flag is set to 0).

Another example:
it's required to compare 7 and 7,
7 - 7 = 0
the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).

here's an example of **CMP** instruction and conditional jump:

-
-
```
include "emu8086.inc"
```
-

```
org    100h

mov    al, 25     ; set al to 25.
mov    bl, 10     ; set bl to 10.

cmp    al, bl     ; compare al - bl.

je     equal      ; jump if al = bl (zf = 1).

putc   'n'        ; if it gets here, then al <> bl,
jmp    stop       ; so print 'n', and jump to stop.

equal:            ; if gets here,
putc   'y'        ; then al = bl, so print 'y'.

stop:

ret               ; gets here no matter what.
```

try the above example with different numbers for **AL** and **BL**, open flags by clicking on flags button, use single step and see what happens. you can use **F5** hotkey to recompile and reload the program into the emulator.

---

loops

instruction	operation and jump condition	opposite instruction
LOOP	decrease cx, jump to label if cx not zero.	DEC CX and JCXZ
LOOPE	decrease cx, jump to label if cx not zero and equal (zf = 1).	LOOPNE
LOOPNE	decrease cx, jump to label if cx not zero and not equal (zf = 0).	LOOPE
LOOPNZ	decrease cx, jump to label if cx not zero and zf = 0.	LOOPZ
LOOPZ	decrease cx, jump to label if cx not zero and zf = 1.	LOOPNZ
JCXZ	jump to label if cx is zero.	OR CX, CX and JNZ

loops are basically the same jumps, it is possible to code loops without using the loop instruction, by just using conditional jumps and compare, and this is just what loop does. all loop instructions use **CX** register to

count steps, as you know CX register has 16 bits and the maximum value it can hold is 65535 or FFFF, however with some agility it is possible to put one loop into another, and another into another two, and three and etc... and receive a nice value of 65535 * 65535 * 65535 ....till infinity.... or the end of ram or stack memory. it is possible store original value of cx register using **push cx** instruction and return it to original when the internal loop ends with **pop cx**, for example:

```
org 100h

mov bx, 0  ; total step counter.

mov cx, 5
k1: add bx, 1
   mov al, '1'
   mov ah, 0eh
   int 10h
   push cx
   mov cx, 5
    k2: add bx, 1
    mov al, '2'
    mov ah, 0eh
    int 10h
    push cx
      mov cx, 5
      k3: add bx, 1
      mov al, '3'
      mov ah, 0eh
      int 10h
      loop k3    ; internal in internal loop.
    pop  cx
    loop  k2     ; internal loop.
  pop cx
loop k1          ; external loop.

ret
```

- bx counts total number of steps, by default emulator shows values in hexadecimal, you can double click the register to see the value in all available bases.

  just like all other conditional jumps loops have an opposite companion that can help to create workarounds, when the address of desired location is too far assemble automatically assembles reverse and long jump instruction, making total of 5 bytes instead of just 2, it can be seen in disassembler as well.

  for more detailed description and examples refer to complete 8086 instruction set

---

All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that

most instructions are assembled into 3 or more bytes).

We can easily avoid this limitation using a cute trick:

- Get an opposite conditional jump instruction from the table above, make it jump to *label_x*.
- Use **JMP** instruction to jump to desired location.
- Define *label_x:* just after the **JMP** instruction.

label_x: - can be any valid label name, but there must not be two or more labels with the same name.

here's an example:

```
include "emu8086.inc"

org    100h

mov    al, 5
mov    bl, 5

cmp    al, bl     ; compare al - bl.


; je equal        ; there is only 1 byte



jne    not_equal  ; jump if al <> bl (zf = 0).
jmp    equal
not_equal:


add    bl, al
sub    al, 10
xor    al, bl

jmp skip_data
db 256 dup(0)     ; 256 bytes
skip_data:


putc   'n'        ; if it gets here, then al <> bl,
jmp    stop       ; so print 'n', and jump to stop.

equal:            ; if gets here,
putc   'y'        ; then al = bl, so print 'y'.

stop:

ret
```

Note: the latest version of the integrated 8086 assembler automatically creates a workaround by replacing the conditional jump with the opposite, and adding big unconditional jump. To check if you have the latest version of emu8086 click **help-> check for an update** from the menu.

Another, yet rarely used method is providing an immediate value instead of label. When immediate value starts with $ relative jump is performed, otherwise compiler calculates instruction that jumps directly to given offset. For example:

```
org    100h

; unconditional jump forward:
; skip over next 3 bytes + itself
; the machine code of short jmp instruction takes 2 bytes.
jmp $3+2
a db 3    ; 1 byte.
b db 4    ; 1 byte.
c db 4    ; 1 byte.

; conditional jump back 5 bytes:
mov bl,9
dec bl      ; 2 bytes.
cmp bl, 0   ; 3 bytes.
jne $-5     ; jump 5 bytes back

ret
```

Procedures

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC

    ; here goes the code
    ; of the procedure ...

RET
name ENDP
```

name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, you already know that **RET** instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

PROC and **ENDP** are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

CALL instruction is used to call a procedure.

Here is an example:

```
ORG    100h

CALL   m1

MOV    AX, 2

RET                ; return to operating system.

m1     PROC
MOV    BX, 5
RET                ; return to caller.
m1     ENDP

END
```

The above example calls procedure **m1**, does **MOV BX, 5**, and returns to the next instruction after **CALL**: **MOV AX, 2**.

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, here is another example of a procedure that receives two parameters in **AL** and **BL** registers, multiplies these parameters and returns the result in **AX** register:

```
ORG    100h

MOV    AL, 1
MOV    BL, 2

CALL   m2
CALL   m2
CALL   m2
CALL   m2

RET                ; return to operating system.

m2     PROC
MUL    BL          ; AX = AL * BL.
RET                ; return to caller.
m2     ENDP

END
```

In the above example value of **AL** register is update every time the procedure is called, **BL** register stays unchanged, so this algorithm calculates **2** in power of **4**,
so final result in **AX** register is **16** (or 10h).

Here goes another example,
that uses a procedure to print a *Hello World!* message:

```
ORG    100h

LEA    SI, msg       ; load address of msg to SI.

CALL   print_me

RET                ; return to operating system.

;
=======================================================
; this procedure prints a string, the string should be null
; terminated (have zero in the end),
; the string address should be in SI register:
print_me     PROC

next_char:
   CMP  b.[SI], 0    ; check for zero to stop
   JE   stop         ;
```

```
    MOV  AL, [SI]     ; next get ASCII char.

    MOV  AH, 0Eh      ; teletype function number.
    INT  10h          ; using interrupt to print a char in AL.

    ADD  SI, 1        ; advance index of string array.

    JMP  next_char    ; go back, and type another char.

stop:
RET                   ; return to caller.
print_me     ENDP
;
================================================================

msg    DB  'Hello World!', 0   ; null terminated string.

END
```

"**b.**" - prefix before [SI] means that we need to compare bytes, not words. When you need to compare words add "**w.**" prefix instead. When one of the compared operands is a register it's not required because compiler knows the size of each register

The Stack

Stack is an area of memory for keeping temporary data. Stack is used by **CALL** instruction to keep return address for procedure, **RET** instruction gets this value from the stack and returns to that offset. Quite the same thing happens when **INT** instruction calls an interrupt, it stores in stack flag register, code segment and offset. **IRET** instruction is used to return from interrupt call.

We can also use the stack to keep any other data,
there are two instructions that work with the stack:

PUSH - stores 16 bit value in the stack.

POP - gets 16 bit value from the stack.

Syntax for **PUSH** instruction:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bit variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Syntax for **POP** instruction:

POP REG
POP SREG
POP memory
REG: AX, BX, CX, DX, DI, SI, BP, SP.

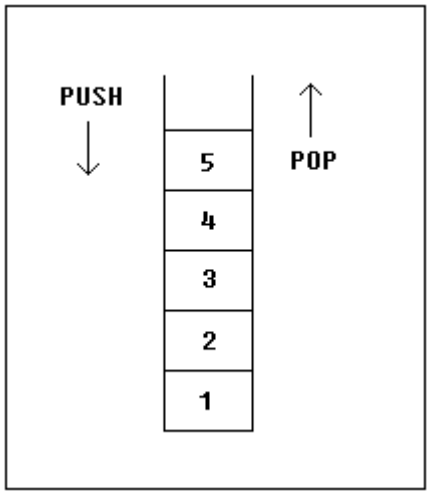
SREG: DS, ES, SS, (except CS).

memory: [BX], [BX+SI+7], 16 bit variable, etc...

Notes:

- **PUSH** and **POP** work with 16 bit values only!
- Note: **PUSH immediate** works only on 80186 CPU and later!

The stack uses **LIFO** (Last In First Out) algorithm,
this means that if we push these values one by one into the stack:
1, 2, 3, 4, 5
the first value that we will get on pop will be **5**, then **4**, **3**, **2**, and only then **1**.



It is very important to do equal number of **PUSH**s and **POP**s, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).

PUSH and **POP** instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register

The Stack

Stack is an area of memory for keeping temporary data. Stack is used by **CALL** instruction to keep return address for procedure, **RET** instruction gets this value from the stack and returns to that offset. Quite the same thing happens when **INT** instruction calls an interrupt, it stores in stack flag register, code segment and offset. **IRET** instruction is used to return from interrupt call.

We can also use the stack to keep any other data,
there are two instructions that work with the stack:

PUSH - stores 16 bit value in the stack.

POP - gets 16 bit value from the stack.

Syntax for **PUSH** instruction:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bit variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Syntax for **POP** instruction:

POP REG
POP SREG
POP memory
REG: AX, BX, CX, DX, DI, SI, BP, SP.

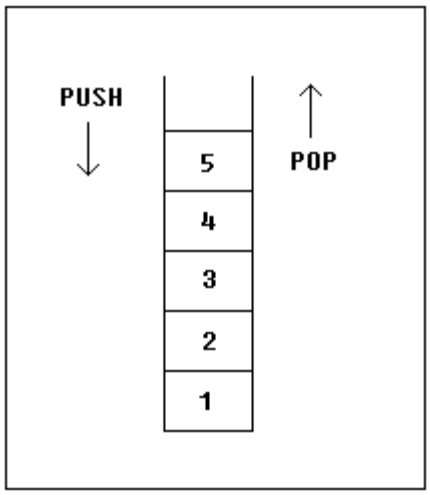
SREG: DS, ES, SS, (except CS).

memory: [BX], [BX+SI+7], 16 bit variable, etc...

Notes:

- **PUSH** and **POP** work with 16 bit values only!
- Note: **PUSH immediate** works only on 80186 CPU and later!

The stack uses **LIFO** (Last In First Out) algorithm,
this means that if we push these values one by one into the stack:
1, 2, 3, 4, 5
the first value that we will get on pop will be **5**, then **4**, **3**, **2**, and only then **1**.



It is very important to do equal number of **PUSH**s and **POP**s, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).

PUSH and **POP** instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register in stack (using **PUSH**).
- Use the register for any purpose.
- Restore the original value of the register from stack (using **POP**).

Here is an example:

```
ORG    100h

MOV    AX, 1234h
PUSH   AX          ; store value of AX in stack.

MOV    AX, 5678h   ; modify the AX value.

POP    AX          ; restore the original value of AX.

RET

END
```

Another use of the stack is for exchanging the values,
here is an example:

```
ORG    100h
```

```
MOV    AX, 1212h   ; store 1212h in AX.
MOV    BX, 3434h   ; store 3434h in BX


PUSH   AX          ; store value of AX in stack.
PUSH   BX          ; store value of BX in stack.

POP    AX          ; set AX to original value of BX.
POP    BX          ; set BX to original value of AX.

RET

END
```

The exchange happens because stack uses **LIFO** (Last In First Out) algorithm, so when we push **1212h** and then **3434h**, on pop we will first get **3434h** and only after it **1212h**.

The stack memory area is set by **SS** (Stack Segment) register, and **SP** (Stack Pointer) register. Generally operating system sets values of these registers on program start.

"**PUSH *source***" instruction does the following:

- Subtract **2** from **SP** register.
- Write the value of ***source*** to the address **SS:SP**.

"**POP *destination***" instruction does the following:

- Write the value at the address **SS:SP** to ***destination***.
- Add **2** to **SP** register.

The current address pointed by **SS:SP** is called **the top of the stack**.

For **COM** files stack segment is generally the code segment, and stack pointer is set to value of **0FFFEh**. At the address **SS:0FFFEh** stored a return address for **RET** instruction that is executed in the end of the program.

You can visually see the stack operation by clicking on [**Stack**] button on emulator window. The top of the stack is marked with "**<**" sign.

Macros

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it. **emu8086.inc** is a good example of how macros can be used, this file contains several macros to make coding easier for you.

```
Macro definition:

name    MACRO  [parameters,...]

        <instructions>

ENDM
```

Unlike procedures, macros should be defined above the code that uses it, for example:

```
MyMacro    MACRO  p1, p2, p3

    MOV AX, p1
    MOV BX, p2
    MOV CX, p3

ENDM

ORG 100h

MyMacro 1, 2, 3

MyMacro 4, 5, DX

RET
```

The above code is expanded into:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

Some important facts about **macros** and **procedures**:

- When you want to use a procedure you should use **CALL** instruction, for example:

  ```
  CALL MyProc
  ```

- When you want to use a macro, you can just type its name. For example:

  ```
  MyMacro
  ```

- Procedure is located at some specific address in memory, and if you use the same procedure 100 times, the CPU will transfer control to this part of the memory. The control will be returned back to the program by **RET** instruction. The **stack** is used to keep the return address. The **CALL** instruction takes about 3 bytes, so the size of the output executable file grows very insignificantly, no matter how many time the procedure is used.

- Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making the output executable file larger and larger, each time all instructions of a macro are inserted.

- You should use **stack** or any general purpose registers to pass parameters to procedure.

- To pass parameters to macro, you can just type them after the macro name. For example:

  ```
  MyMacro 1, 2, 3
  ```

- To mark the end of the macro **ENDM** directive is enough.

- To mark the end of the procedure, you should type the name of the procedure before the **ENDP** directive.

Macros are expanded directly in code, therefore if there are labels inside the macro definition you may get "Duplicate declaration" error when macro is used for twice or more. To avoid such problem, use **LOCAL** directive followed by names of variables, labels or procedure names. For example:

```
MyMacro2    MACRO
            LOCAL label1, label2

            CMP  AX, 2
            JE label1
            CMP  AX, 3
            JE label2
            label1:
                        INC  AX
            label2:
                        ADD  AX, 2
ENDM
```

```
ORG 100h

MyMacro2

MyMacro2

RET
```

If you plan to use your macros in several programs, it may be a good idea to place all macros in a separate file. Place that file in **Inc** folder and use **INCLUDE *file-name*** directive to use macros. See **Library of common functions - emu8086.inc** for an example of such file.

---

making your own operating system

Usually, when a computer starts it will try to load the first 512-byte sector (that's Cylinder **0**, Head **0**, Sector **1**) from any diskette in your **A:** drive to memory location **0000h:7C00h** and give it control. If this fails, the BIOS tries to use the MBR of the first hard drive instead.

This tutorial covers booting up from a floppy drive, the same principles are used to boot from a hard drive. But using a floppy drive has several advantages:

- you can keep your existing operating system intact (windows, dos, linux, unix, be-os...).
- it is easy and safe to modify the boot record of a floppy disk.

example of a simple floppy disk boot program:

```
; directive to create BOOT file:
#make_boot#

; Boot record is loaded at 0000:7C00,
; so inform compiler to make required
; corrections:
ORG 7C00h
```

```
PUSH    CS   ; make sure DS=CS
POP     DS

; load message address into SI register:
LEA SI, msg

; teletype function id:
MOV AH, 0Eh

print:   MOV AL, [SI]
        CMP AL, 0
        JZ done
        INT 10h   ; print using teletype.
        INC SI
        JMP print

; wait for 'any key':
done:      MOV AH, 0
          INT 16h


; store magic value at 0040h:0072h:
;   0000h - cold boot.
;   1234h - warm boot.
MOV     AX, 0040h
MOV     DS, AX
MOV     w.[0072h], 0000h ; cold boot.

JMP          0FFFFh:0000h             ; reboot!


new_line EQU 13, 10

msg DB  'Hello This is My First Boot Program!'
   DB  new_line, 'Press any key to reboot', 0
```

copy the above example to the source editor and press **emulate**. the emulator automatically loads **.bin** file to **0000h:7C00h** (it uses supplementary .binf file to know where to load).

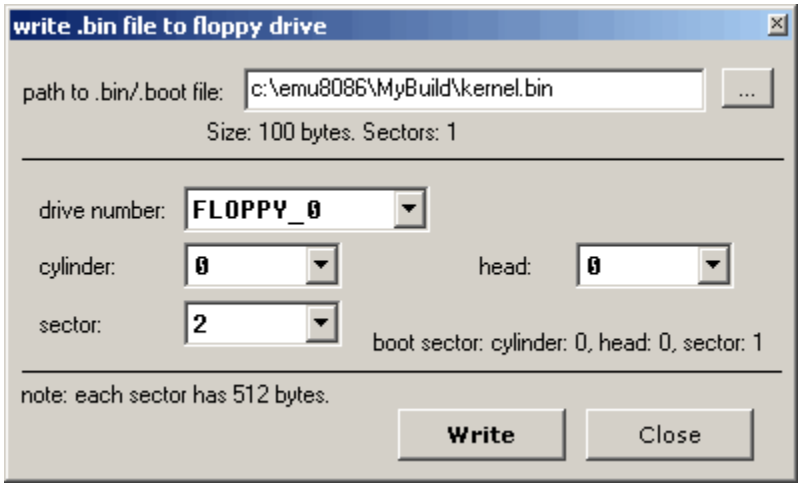
you can run it just like a regular program, or you can use the **virtual drive** menu to **write 512 bytes at 7c00h to boot sector** of a virtual floppy drive (it's "**FLOPPY_0**" file in c:\emu8086). after your program is written to the virtual floppy drive, you can select **boot from floppy** from **virtual drive** menu.

.bin files for boot records are limited to 512 bytes (sector size). if your new operating system is going to grow over this size, you will need to use a boot program to load data from other sectors (just like *micro-os_loader.asm* does). an example of a tiny operating system can be found in c:\emu8086\examples and "**online**":

micro-os_loader.asm
micro-os_kernel.asm

To create extensions for your Operating System (over 512 bytes), you can use additional sectors of a floppy disk. It's recommended to use "**.bin**" files for this purpose (to create "**.bin**" file select "**BIN Template**" from "**File**" -> "**New**" menu).

To write "**.bin**" file to virtual floppy, select **"Write .bin file to floppy..."** from **"Virtual drive"** menu of emulator, you should write it anywhere but the boot sector (which is Cylinder: **0**, Head: **0**, Sector: **1**).



you can use this utility to write **.bin** files to virtual floppy disk ("**FLOPPY_0**" file), instead of "**write 512 bytes at 7c00h to boot sector**" menu. however, you should remember that **.bin** file that is designed to be a boot record should always be written to cylinder: **0**, head: **0**, sector: **1**

Boot Sector Location:
Cylinder: 0
Head: 0
Sector: 1

to write .bin files to real floppy disk use writebin.asm, just compile it to com file and run it from command prompt. to write a boot record type: **writebin loader.bin** ; to write kernel module type: **writebin kernel.bin /k**
/k - parameter tells the program to write the file at sector 2 instead of sector 1. it does not matter in what order you write the files onto floppy drive, but it does matter where you write them.

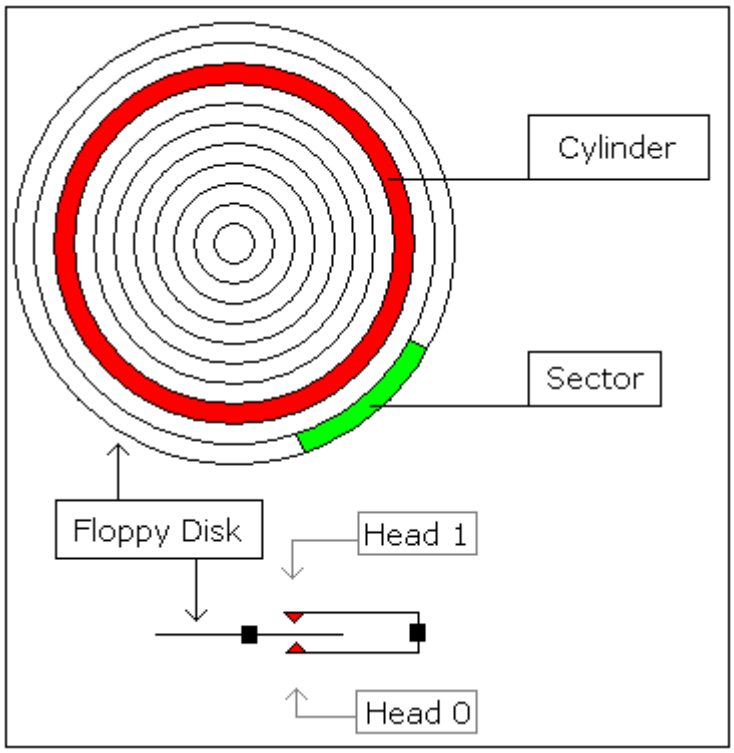
---

mote: this boot record is not MS-DOS/Windows compatible boot sector, it's not even Linux or Unix compatible, operating system may not allow you to

read or write files on this diskette until you re-format it, therefore make sure the diskette you use doesn't contain any important information. however you can write and read anything to and from this disk using low level disk access interrupts, it's even possible to protect valuable information from the others this way; even if someone gets the disk he will probably think that it's empty and will reformat it because it's the default option in windows operating system... such a good type of self destructing data carrier :)

---

idealized floppy drive and diskette structure:



for a **1440 kb** diskette:

- floppy disk has 2 sides, and there are 2 heads; one for each side (**0..1**), the drive heads move above the surface of the disk on each side.
- each side has 80 cylinders (numbered **0..79**).
- each cylinder has 18 sectors (**1..18**).
- each sector has **512** bytes.
- total size of floppy disk is: 2 x 80 x 18 x 512 = **1,474,560** bytes.

note: the MS-DOS (windows) formatted floppy disk has slightly less free space on it (by about 16,896 bytes) because the operating system needs place to store file names and directory structure (often called FAT or file system allocation table). more file names - less disk space. the most efficient way to

store files is to write them directly to sectors instead of using file system, and in some cases it is also the most reliable way, if you know how to use it.

to read sectors from floppy drive use **INT 13h / AH = 02h**.

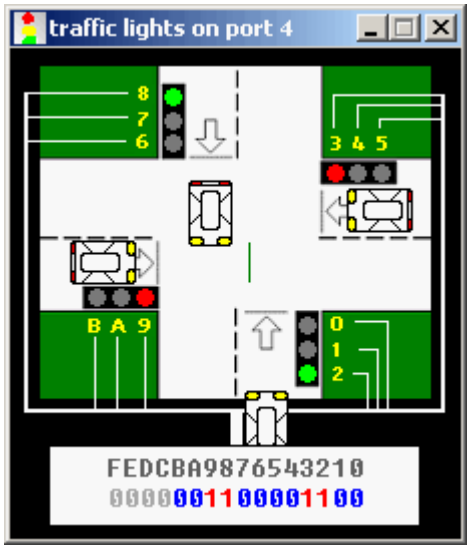
Controlling External Devices

There are 7 devices attached to the emulator: traffic lights, stepper-motor, LED display, thermometer, printer, robot and simple test device. You can view devices when you click "**Virtual Devices**" menu of the emulator.

For technical information refer to **I/O ports** section of emu8086 reference.

In general, it is possible to use any x86 family CPU to control all kind of devices, the difference maybe in base I/O port number, this can be altered using some tricky electronic equipment. Usually the "**.bin**" file is written into the Read Only Memory (ROM) chip, the system reads program from that chip, loads it in RAM module and runs the program. This principle is used for many modern devices such as micro-wave ovens and etc...

Traffic Lights



Usually to control the traffic lights an array (table) of values is used. In certain periods of time the value is read from the array and sent to a port. For example:

```
; controlling external device with 8086 microprocessor.
; realistic test for c:\emu8086\devices\Traffic_Lights.exe
```

```
#start=Traffic_Lights.exe#

name "traffic"


mov ax, all_red
out 4, ax


mov si, offset situation


next:
mov ax, [si]
out 4, ax

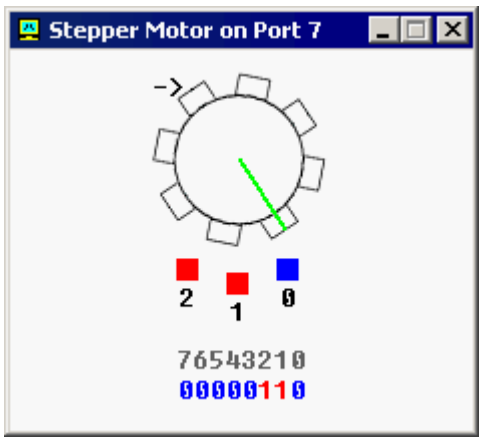
; wait 5 seconds (5 million microseconds)
mov     cx, 4Ch    ;    004C4B40h = 5,000,000
mov     dx, 4B40h
mov     ah, 86h
int     15h


add si, 2 ; next situation
cmp si, sit_end
jb  next
mov si, offset situation
jmp next


;                   FEDC_BA98_7654_3210
situation       dw      0000_0011_0000_1100b
s1              dw      0000_0110_1001_1010b
s2              dw      0000_1000_0110_0001b
s3              dw      0000_1000_0110_0001b
s4              dw      0000_0100_1101_0011b
sit_end = $


all_red         equ     0000_0010_0100_1001b
```

Stepper-Motor



The motor can be half stepped by turning on pair of magnets, followed by a single and so on.

The motor can be full stepped by turning on pair of magnets, followed by another pair of magnets and in the end followed by a single magnet and so on. The best way to make full step is to make two half steps.

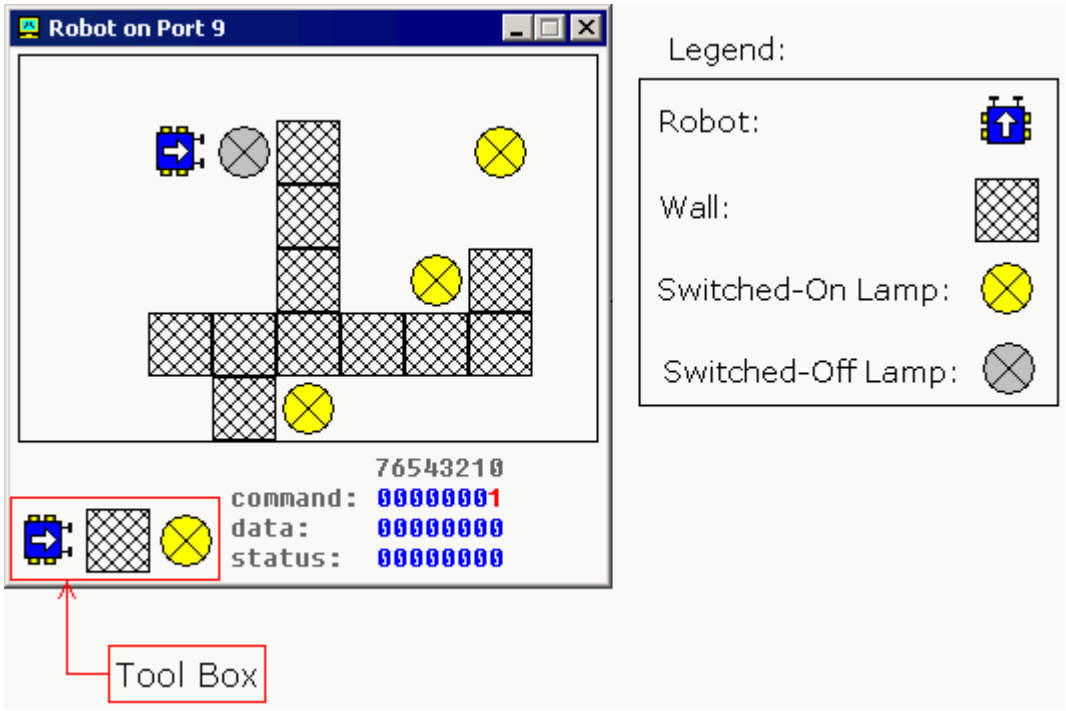
Half step is equal to **11.25** degrees.
Full step is equal to **22.5** degrees.

The motor can be turned both clock-wise and counter-clock-wise.

See **stepper_motor.asm** in c:\emu8086\examples\

See also **I/O ports** section of emu8086 reference.

Robot



Complete list of robot instruction set is given in **I/O ports** section of emu8086 reference.

To control the robot a complex algorithm should be used to achieve maximum efficiency. The simplest, yet very inefficient, is random moving algorithm, see **robot.asm** in c:\emu8086\examples\

It is also possible to use a data table (just like for Traffic Lights), this can be good if robot always works in the same surroundings.

Complete 8086 instruction set

Quick reference:

AAA	**CMPSB**	**JAE**	**JNBE**	**JPO**	**MOV**	**RCR**	**SCASB**
AAD	**CMPSW**	**JB**	**JNC**	**JS**	**MOVSB**	**REP**	**SCASW**
AAM	**CWD**	**JBE**	**JNE**	**JZ**	**MOVSW**	**REPE**	**SHL**
AAS	**DAA**	**JC**	**JNG**	**LAHF**	**MUL**	**REPNE**	**SHR**
ADC	**DAS**	**JCXZ**	**JNGE**	**LDS**	**NEG**	**REPNZ**	**STC**
ADD	**DEC**	**JE**	**JNL**	**LEA**	**NOP**	**REPZ**	**STD**
AND	**DIV**	**JG**	**JNLE**	**LES**	**NOT**	**RET**	**STI**
CALL	**HLT**	**JGE**	**JNO**	**LODSB**	**OR**	**RETF**	**STOSB**
CBW	**IDIV**	**JL**	**JNP**	**LODSW**	**OUT**	**ROL**	**STOSW**
CLC	**IMUL**	**JLE**	**JNS**	**LOOP**	**POP**	**ROR**	**SUB**
CLD	**IN**	**JMP**	**JNZ**	**LOOPE**	**POPA**	**SAHF**	**TEST**

CLI	**INC**	**JNA**	**JO**	**LOOPNE**	**POPF**	**SAL**	**XCHG**
CMC	**INT**	**JNAE**	**JP**	**LOOPNZ**	**PUSH**	**SAR**	**XLATB**
CMP	**INTO**	**JNB**	**JPE**	**LOOPZ**	**PUSHA**	**SBB**	**XOR**
	IRET				**PUSHF**		
	JA				**RCL**		

Operand types:

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

SREG: DS, ES, SS, and only as second operand: CS.

memory: [BX], [BX+SI+7], variable, etc...(see **Memory Access**).

immediate: 5, -24, 3Fh, 10001101b, etc...

Notes:

- When two operands are required for an instruction they are separated by comma. For example:

  REG, memory

- When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:

  ```
  AL, DL
  DX, AX
  m1 DB ?
  AL, m1
  m2 DW ?
  AX, m2
  ```

- Some instructions allow several operand combinations. For example:

  memory, immediate
  REG, immediate

  memory, REG
  REG, SREG

- Some examples contain macros, so it is advisable to use **Shift + F8** hot key to *Step Over* (to make macro code execute at maximum speed set **step delay** to zero), otherwise emulator will step through each instruction of a macro. Here is an example that uses PRINTN macro:
-
-
- ```
  include 'emu8086.inc'
  ```
  ```

```
    ORG 100h
    MOV AL, 1
    MOV BL, 2
    PRINTN 'Hello World!'   ; macro.
    MOV CL, 3
    PRINTN 'Welcome!'       ; macro.
RET
```

These marks are used to show the state of the flags:

1 - instruction sets this flag to **1**.
0 - instruction sets this flag to **0**.
r - flag value depends on result of the instruction.
? - flag value is undefined (maybe **1** or **0**).

Some instructions generate exactly the same machine code, so disassembler may have a problem decoding to your original code. This is especially important for Conditional Jump instructions (see "Program Flow Control" in Tutorials for more information).

Instructions in alphabetical order:

Instruction	Operands	Description

AAA	No operands	ASCII Adjust after Addition.
		Corrects result in AH and AL after addition when working with BCD values.
		
		It works according to the following Algorithm:
		
		if low nibble of AL > 9 or AF = 1 then:
		• AL = AL + 6
		• AH = AH + 1
		• AF = 1
		• CF = 1
		
		else
		• AF = 0
		• CF = 0

		 in both cases: clear the high nibble of AL. Example: MOV AX, 15 ; AH = 00, AL = 0Fh AAA ; AH = 01, AL = 05 RET C Z S O P A r ? ? ? ? r
AAD	No operands	 ASCII Adjust before Division. Prepares two BCD values for division. Algorithm: • AL = (AH * 10) + AL • AH = 0 Example: MOV AX, 0105h ; AH = 01, AL = 05 AAD ; AH = 00, AL = 0Fh (15) RET C Z S O P A ? r r ? r ?
AAM	No operands	 ASCII Adjust after Multiplication. Corrects the result of multiplication of two BCD values. Algorithm: • AH = AL / 10 • AL = remainder Example: MOV AL, 15 ; AL = 0Fh AAM ; AH = 01, AL = 05 RET C Z S O P A ? r r ? r ?
AAS	No operands	ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values. Algorithm: if low nibble of AL > 9 or AF = 1 then:

		• AL = AL - 6 • AH = AH - 1 • AF = 1 • CF = 1  else • AF = 0 • CF = 0 in both cases: clear the high nibble of AL. Example: MOV AX, 02FFh ; AH = 02, AL = 0FFh AAS ; AH = 01, AL = 09 RET C Z S O P A r ? ? ? ? r
ADC	REG, memory memory, REG REG, REG memory, immediate REG, immediate	 Add with Carry. Algorithm: operand1 = operand1 + operand2 + CF Example: STC ; set CF = 1 MOV AL, 5 ; AL = 5 ADC AL, 1 ; AL = 7 RET C Z S O P A r r r r r r

ADD	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		 Add.
		
		Algorithm:
		
		operand1 = operand1 + operand2
		
		Example:
		MOV AL, 5 ; AL = 5
		ADD AL, -3 ; AL = 2
		RET
		C Z S O P A
		r r r r r r

AND	REG, memory	Logical AND between all bits of two operands. Result is stored in

	memory, REG REG, REG memory, immediate REG, immediate	 operand1. These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Example: MOV AL, 'a' ; AL = 01100001b AND AL, 11011111b ; AL = 01000001b ('A') RET C Z S O P 0 r r 0 r

CALL	procedure
	name
	label
	4-byte address
		 Transfers control to procedure, return address is (IP) is pushed to stack. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset (this is a far call, so CS is also pushed to stack).
		
		Example:
		
		ORG 100h ; directive to make simple .com file.
		
		CALL p1
		
		ADD AX, 1
		
		RET ; return to OS.
		
		p1 PROC ; procedure declaration.
		MOV AX, 1234h
		RET ; return to caller.
		p1 ENDP
		C Z S O P A
		unchanged

CBW	No operands	Convert byte into word.
		
		Algorithm:
		
		if high bit of AL = 1 then:
		
		• AH = 255 (0FFh)
		
		else
		
		• AH = 0

		 Example:
		MOV AX, 0 ; AH = 0, AL = 0
		MOV AL, -5 ; AX = 000FBh (251)
		CBW ; AX = 0FFFBh (-5)
		RET
		C Z S O P A
		unchanged

CLC	No operands	 Clear Carry flag. Algorithm: CF = 0 C 0
CLD	No operands	 Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 0 D 0
CLI	No operands	 Clear Interrupt enable flag. This disables hardware interrupts. Algorithm: IF = 0 I 0
CMC	No operands	Complement Carry flag. Inverts value of CF. Algorithm: if CF = 1 then CF = 0 if CF = 0 then CF = 1 C r

CMP	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		 Compare.
		
		Algorithm:
		
		operand1 - operand2
		
		result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result.
		
		Example:
		MOV AL, 5
		MOV BL, 5
		CMP AL, BL ; AL = 5, ZF = 1 (so equal!)
		RET
		
		C Z S O P A
		r r r r r r

CMPSB	No operands	Compare bytes: ES:[DI] from DS:[SI].
		
		Algorithm:
		
		• DS:[SI] - ES:[DI]
		• set flags according to result: OF, SF, ZF, AF, PF, CF
		• if DF = 0 then
		  ○ SI = SI + 1
		  ○ DI = DI + 1
		
		else
		
		 
		  ○ SI = SI - 1
		  ○ DI = DI - 1
		
		Example:
		see **cmpsb.asm** in c:\emu8086\examples\.
		
		C Z S O P A
		r r r r r r

CMPSW	No operands	Compare words: ES:[DI] from DS:[SI].
		
		Algorithm:
		
		• DS:[SI] - ES:[DI]
		• set flags according to result: OF, SF, ZF, AF, PF, CF
		• if DF = 0 then
		  ○ SI = SI + 2
		  ○ DI = DI + 2

		 else ○ SI = SI - 2 ○ DI = DI - 2 Example: see **cmpsw.asm** in c:\emu8086\examples\. C Z S O P A r r r r r r

CWD	No operands	Convert Word to Double word.
		
		Algorithm:
		
		if high bit of AX = 1 then:
		
		• DX = 65535 (0FFFFh)
		
		else
		
		• DX = 0
		
		  Example:
		MOV DX, 0 ; DX = 0
		MOV AX, 0 ; AX = 0
		MOV AX, -5 ; DX AX = 00000h:0FFFBh
		CWD ; DX AX = 0FFFFh:0FFFBh
		RET
		C Z S O P A
		unchanged

DAA	No operands	Decimal adjust After Addition.
		Corrects the result of addition of two packed BCD values.
		
		Algorithm:
		
		if low nibble of AL > 9 or AF = 1 then:
		
		• AL = AL + 6
		• AF = 1
		
		if AL > 9Fh or CF = 1 then:
		
		• AL = AL + 60h
		• CF = 1
		
		Example:
		MOV AL, 0Fh ; AL = 0Fh (15)

		DAA ; AL = 15h RET C Z S O P A r r r r r r

DAS	No operands	Decimal adjust After Subtraction.
		Corrects the result of subtraction of two packed BCD values.
		
		Algorithm:
		
		if low nibble of AL > 9 or AF = 1 then:
		• AL = AL - 6
		• AF = 1
		
		if AL > 9Fh or CF = 1 then:
		• AL = AL - 60h
		• CF = 1
		
		Example:
		MOV AL, 0FFh ; AL = 0FFh (-1)
		DAS ; AL = 99h, CF = 1
		RET
		C Z S O P A
		r r r r r r

DEC	REG memory	Decrement. Algorithm: operand = operand - 1 Example: MOV AL, 255 ; AL = 0FFh (255 or -1) DEC AL ; AL = 0FEh (254 or -2) RET Z S O P A r r r r r CF - unchanged!
DIV	REG memory	Unsigned divide. Algorithm: when operand is a **byte**: AL = AX / operand AH = remainder (modulus) when operand is a **word**: AX = (DX AX) / operand

		 DX = remainder (modulus)
		Example:
		MOV AX, 203 ; AX = 00CBh
		MOV BL, 4
		DIV BL ; AL = 50 (32h), AH = 3
		RET
		C Z S O P A
		? ? ? ? ? ?

HLT	No operands	 Halt the System. Example: MOV AX, 5 HLT C Z S O P A unchanged

IDIV	REG
	memory
		 Signed divide.
		Algorithm:
		when operand is a **byte**:
		AL = AX / operand
		AH = remainder (modulus)
		when operand is a **word**:
		AX = (DX AX) / operand
		DX = remainder (modulus)
		Example:
		MOV AX, -203 ; AX = 0FF35h
		MOV BL, 4
		IDIV BL ; AL = -50 (0CEh), AH = -3 (0FDh)
		RET
		C Z S O P A
		? ? ? ? ? ?

IMUL	REG
	memory
		Signed multiply.
		Algorithm:
		when operand is a **byte**:
		AX = AL * operand.
		when operand is a **word**:
		(DX AX) = AX * operand.
		Example:
		MOV AL, -2
		MOV BL, -4
		IMUL BL ; AX = 8
		RET
		C Z S O P A
		r ? ? r ? ?

		 CF=OF=0 when result fits into operand of IMUL.
IN	AL, im.byte AL, DX AX, im.byte AX, DX	Input from port into **AL** or **AX**. Second operand is a port number. If required to access port number over 255 - **DX** register should be used.  Example: IN AX, 4 ; get status of traffic lights. IN AL, 7 ; get status of stepper-motor. C Z S O P A unchanged
INC	REG memory	 Increment. Algorithm: operand = operand + 1 Example: MOV AL, 4 INC AL ; AL = 5 RET Z S O P A r r r r r CF - unchanged!

INT	immediate
	byte
		
		Interrupt numbered by immediate byte (0..255).
		Algorithm:
		Push to stack:
		○ flags register
		○ CS
		○ IP
		• IF = 0
		• Transfer control to interrupt procedure
		Example:
		MOV AH, 0Eh ; teletype.
		MOV AL, 'A'
		INT 10h ; BIOS interrupt.
		RET
		C Z S O P A I
		unchanged 0

INTO	No operands	Interrupt 4 if Overflow flag is 1. Algorithm:

		 if OF = 1 then INT 4
		
		Example:
		; -5 - 127 = -132 (not in -128..127)
		; the result of SUB is wrong (124),
		; so OF = 1 is set:
		MOV AL, -5
		SUB AL, 127 ; AL = 7Ch (124)
		INTO ; process error.
		RET

IRET	No operands	 Interrupt Return. Algorithm: Pop from stack: ○ IP ○ CS ○ flags register C Z S O P A popped

JA	label	 Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if (CF = 0) and (ZF = 0) then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 250
		CMP AL, 5
		JA label1
		PRINT 'AL is not above 5'
		JMP exit
		label1:
		PRINT 'AL is above 5'
		exit:
		RET
		C Z S O P A
		unchanged

JAE	label	Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 0 then jump Example:

		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JAE label1
		PRINT 'AL is not above or equal to 5'
		JMP exit
		label1:
		PRINT 'AL is above or equal to 5'
		exit:
		RET
		C Z S O P A
		unchanged

JB	label	
		Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 1
		CMP AL, 5
		JB label1
		PRINT 'AL is not below 5'
		JMP exit
		label1:
		PRINT 'AL is below 5'
		exit:
		RET
		C Z S O P A
		unchanged

JBE	label	Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 or ZF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JBE label1
		PRINT 'AL is not below or equal to 5'
		JMP exit
		label1:
		PRINT 'AL is below or equal to 5'
		exit:

		 RET C Z S O P A unchanged

JC	label	 Short Jump if Carry flag is set to 1.
		
		Algorithm:
		
		if CF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 255
		ADD AL, 1
		JC label1
		PRINT 'no carry.'
		JMP exit
		label1:
		PRINT 'has carry.'
		exit:
		RET
		C Z S O P A
		unchanged

JCXZ	label	Short Jump if CX register is 0. Algorithm: if CX = 0 then jump Example: include 'emu8086.inc'  ORG 100h MOV CX, 0 JCXZ label1 PRINT 'CX is not zero.' JMP exit label1: PRINT 'CX is zero.' exit: RET C Z S O P A unchanged
JE	label	Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned. Algorithm: if ZF = 1 then jump Example:

		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JE label1
		PRINT 'AL is not equal to 5.'
		JMP exit
		label1:
		PRINT 'AL is equal to 5.'
		exit:
		RET
		C Z S O P A
		unchanged

JG	label	Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if (ZF = 0) and (SF = OF) then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 5
		 
		CMP AL, -5
		JG label1
		PRINT 'AL is not greater -5.'
		JMP exit
		label1:
		PRINT 'AL is greater -5.'
		exit:
		RET
		C Z S O P A
		unchanged

JGE	label	Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if SF = OF then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, -5
		JGE label1
		PRINT 'AL < -5'
		JMP exit
		label1:
		PRINT 'AL >= -5'
		exit:

		 RET C Z S O P A unchanged

JL	label	Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if SF <> OF then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, -2
		CMP AL, 5
		JL label1
		PRINT 'AL >= 5.'
		JMP exit
		  label1:
		PRINT 'AL < 5.'
		exit:
		RET
		C Z S O P A
		unchanged

JLE	label	 Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if SF <> OF or ZF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, -2
		CMP AL, 5
		JLE label1
		PRINT 'AL > 5.'
		JMP exit
		label1:
		PRINT 'AL <= 5.'
		exit:
		RET
		C Z S O P A
		unchanged

JMP	label 4-byte address	Unconditional Jump. Transfers control to another part of the program. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.

		Algorithm: always jump Example: include 'emu8086.inc' ORG 100h MOV AL, 5 JMP label1 ; jump over 2 lines! PRINT 'Not Jumped!'  MOV AL, 0 label1: PRINT 'Got Here!' RET C Z S O P A unchanged

JNA	label	 Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 or ZF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 5
		JNA label1
		PRINT 'AL is above 5.'
		JMP exit
		label1:
		PRINT 'AL is not above 5.'
		exit:
		RET
		C Z S O P A
		unchanged

JNAE	label	Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 5
		JNAE label1
		PRINT 'AL >= 5.'
		JMP exit

		 label1: PRINT 'AL < 5.' exit: RET C Z S O P A unchanged

JNB	label	
		Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if CF = 0 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 7
		CMP AL, 5
		JNB label1
		PRINT 'AL < 5.'
		JMP exit
		label1:
		PRINT 'AL >= 5.'
		exit:
		RET
		C Z S O P A
		unchanged

JNBE	label	
		Short Jump if first operand is Not Below and Not Equal to second operand (as set by CMP instruction). Unsigned.
		
		Algorithm:
		
		if (CF = 0) and (ZF = 0) then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 7
		CMP AL, 5
		JNBE label1
		PRINT 'AL <= 5.'
		JMP exit
		label1:
		PRINT 'AL > 5.'
		exit:
		RET
		C Z S O P A
		unchanged

JNC	label	Short Jump if Carry flag is set to 0.

		 Algorithm:
		
		if CF = 0 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		ADD AL, 3
		JNC label1
		PRINT 'has carry.'
		JMP exit
		label1:
		PRINT 'no carry.'
		exit:
		RET
		C Z S O P A
		unchanged

JNE	label	 Short Jump if first operand is Not Equal to second operand (as set by CMP instruction). Signed/Unsigned.
		
		Algorithm:
		
		if ZF = 0 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 3
		JNE label1
		PRINT 'AL = 3.'
		JMP exit
		label1:
		PRINT 'Al <> 3.'
		exit:
		RET
		C Z S O P A
		unchanged

JNG	label	Short Jump if first operand is Not Greater then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if (ZF = 1) and (SF <> OF) then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 3
		JNG label1

		 PRINT 'AL > 3.' JMP exit label1: PRINT 'Al <= 3.' exit: RET C Z S O P A unchanged

JNGE	label	
		Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if SF <> OF then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, 3
		JNGE label1
		PRINT 'AL >= 3.'
		JMP exit
		label1:
		PRINT 'Al < 3.'
		exit:
		RET
		C Z S O P A
		unchanged

JNL	label	Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		 
		if SF = OF then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, -3
		JNL label1
		PRINT 'AL < -3.'
		JMP exit
		label1:
		PRINT 'Al >= -3.'
		exit:
		RET
		C Z S O P A
		unchanged

JNLE	label	 Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed.
		
		Algorithm:
		
		if (SF = OF) and (ZF = 0) then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 2
		CMP AL, -3
		JNLE label1
		PRINT 'AL <= -3.'
		JMP exit
		label1:
		PRINT 'Al > -3.'
		exit:
		RET
		C Z S O P A
		unchanged

JNO	label	Short Jump if Not Overflow.
		
		Algorithm:
		
		if OF = 0 then jump
		Example:
		; -5 - 2 = -7 (inside -128..127)
		 
		; the result of SUB is correct,
		; so OF = 0:
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, -5
		SUB AL, 2 ; AL = 0F9h (-7)
		JNO label1
		PRINT 'overflow!'
		JMP exit
		label1:
		PRINT 'no overflow.'
		exit:
		RET
		C Z S O P A
		unchanged

JNP	label	Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 0 then jump Example:

		include 'emu8086.inc' ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNP label1 PRINT 'parity even.' JMP exit label1: PRINT 'parity odd.' exit: RET C Z S O P A unchanged
JNS	label	Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if SF = 0 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNS label1 PRINT 'signed.' JMP exit label1: PRINT 'not signed.' exit: RET C Z S O P A unchanged
JNZ	label	Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if ZF = 0 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNZ label1 PRINT 'zero.' JMP exit label1: PRINT 'not zero.' exit:

		 RET C Z S O P A unchanged

JO	label	Short Jump if Overflow.
		
		Algorithm:
		
		if OF = 1 then jump
		Example:
		; -5 - 127 = -132 (not in -128..127)
		; the result of SUB is wrong (124),
		; so OF = 1 is set:
		
		include 'emu8086.inc'
		 
		org 100h
		MOV AL, -5
		SUB AL, 127 ; AL = 7Ch (124)
		JO label1
		PRINT 'no overflow.'
		JMP exit
		label1:
		PRINT 'overflow!'
		exit:
		RET
		C Z S O P A
		unchanged

JP	label	 Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 1 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 00000101b ; AL = 5 OR AL, 0 ; just set flags. JP label1 PRINT 'parity odd.' JMP exit label1: PRINT 'parity even.' exit: RET C Z S O P A unchanged

JPE	label	Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 1 then jump Example: include 'emu8086.inc'  ORG 100h MOV AL, 00000101b ; AL = 5 OR AL, 0 ; just set flags. JPE label1 PRINT 'parity odd.' JMP exit label1: PRINT 'parity even.' exit: RET C Z S O P A unchanged
JPO	label	 Short Jump if Parity Odd. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 0 then jump Example: include 'emu8086.inc' ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JPO label1 PRINT 'parity even.' JMP exit label1: PRINT 'parity odd.' exit: RET C Z S O P A unchanged
JS	label	Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if SF = 1 then jump Example: include 'emu8086.inc' ORG 100h

		MOV AL, 10000000b ; AL = -128 OR AL, 0 ; just set flags. JS label1 PRINT 'not signed.' JMP exit label1: PRINT 'signed.' exit: RET C Z S O P A unchanged

JZ	label	Short Jump if Zero (equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		if ZF = 1 then jump
		Example:
		include 'emu8086.inc'
		
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JZ label1
		PRINT 'AL is not equal to 5.'
		JMP exit
		label1:
		PRINT 'AL is equal to 5.'
		exit:
		RET
		C Z S O P A
		unchanged

LAHF	No operands	Load AH from 8 low bits of Flags register. Algorithm: AH = flags register AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. C Z S O P A unchanged
LDS	REG, memory	Load memory double word into word register and DS. Algorithm: • REG = first word

<table>
<tr><td></td><td></td><td> 
DS = second word
Example:

ORG 100h

LDS AX, m

RET

m DW 1234h
DW 5678h

END

AX is set to 1234h, DS is set to 5678h.

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td></tr>
<tr><td>LEA</td><td>REG, memory</td><td>  Load Effective Address.

Algorithm:

REG = address of memory (offset)
Example:

MOV BX, 35h
MOV DI, 12h
LEA SI, [BX+DI] ; SI = 35h + 12h = 47h
Note: The integrated 8086 assembler automatically replaces LEA with a more efficient MOV where possible. For example:

org 100h
LEA AX, m ; AX = offset of m
RET
m dw 1234h
END

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td></tr>
<tr><td>LES</td><td>REG, memory</td><td>Load memory double word into word register and ES.

Algorithm:

REG = first word</td></tr>
</table>

		 • ES = second word Example: ORG 100h LES AX, m RET m DW 1234h DW 5678h END AX is set to 1234h, ES is set to 5678h. C Z S O P A unchanged

LODSB	No operands	Load byte at DS:[SI] into AL. Update SI.
		
		Algorithm:
		
		• AL = DS:[SI]
		• if DF = 0 then
		○ SI = SI + 1
		
		else
		
		○ SI = SI - 1
		
		Example:
		
		  ORG 100h
		
		LEA SI, a1
		MOV CX, 5
		MOV AH, 0Eh
		
		m: LODSB
		INT 10h
		LOOP m
		
		RET
		
		a1 DB 'H', 'e', 'l', 'l', 'o'
		C Z S O P A
		unchanged

LODSW	No operands	Load word at DS:[SI] into AX. Update SI.

		 Algorithm:
		• AX = DS:[SI]
		• if DF = 0 then
		○ SI = SI + 2
		else
		○ SI = SI - 2
		Example:
		ORG 100h
		LEA SI, a1
		MOV CX, 5
		REP LODSW ; finally there will be 555h in AX.
		RET
		a1 dw 111h, 222h, 333h, 444h, 555h
		C Z S O P A
		unchanged

LOOP	label	Decrease CX, jump to label if CX not zero.
		Algorithm:
		 
		• CX = CX - 1
		• if CX <> 0 then
		○ jump
		else
		○ no jump, continue
		Example:
		include 'emu8086.inc'
		ORG 100h
		MOV CX, 5
		label1:
		PRINTN 'loop!'
		LOOP label1
		RET
		C Z S O P A
		unchanged

LOOPE	label	Decrease CX, jump to label if CX not zero and Equal (ZF = 1). Algorithm:

		
		• CX = CX - 1
		• if (CX <> 0) and (ZF = 1) then
		  ○ jump
		
		else
		
		  ○ no jump, continue
		
		Example:
		; Loop until result fits into AL alone,
		; or 5 times. The result will be over 255
		; on third loop (100+100+100),
		; so loop will exit.
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AX, 0
		MOV CX, 5
		label1:
		PUTC '*'
		ADD AX, 100
		CMP AH, 0
		LOOPE label1
		RET
		C Z S O P A
		unchanged

LOOPNE	label	Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0).
		
		Algorithm:
		
		• CX = CX - 1
		• if (CX <> 0) and (ZF = 0) then
		  ○ jump
		
		else
		
		  ○ no jump, continue
		
		Example:
		; Loop until '7' is found,
		; or 5 times.
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV SI, 0
		MOV CX, 5
		label1:
		PUTC '*'
		MOV AL, v1[SI]
		INC SI ; next byte (SI=SI+1).
		CMP AL, 7
		LOOPNE label1

		 RET v1 db 9, 8, 7, 6, 5 C Z S O P A unchanged

LOOPNZ	label	Decrease CX, jump to label if CX not zero and ZF = 0.
		
		Algorithm:
		
		• CX = CX - 1
		• if (CX <> 0) and (ZF = 0) then
		○ jump
		
		else
		 
		○ no jump, continue
		
		Example:
		; Loop until '7' is found,
		; or 5 times.
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV SI, 0
		MOV CX, 5
		label1:
		PUTC '*'
		MOV AL, v1[SI]
		INC SI ; next byte (SI=SI+1).
		CMP AL, 7
		LOOPNZ label1
		RET
		v1 db 9, 8, 7, 6, 5
		C Z S O P A
		unchanged

LOOPZ	label	Decrease CX, jump to label if CX not zero and ZF = 1. Algorithm: • CX = CX - 1 • if (CX <> 0) and (ZF = 1) then ○ jump else ○ no jump, continue Example: ; Loop until result fits into AL alone, ; or 5 times. The result will be over 255 ; on third loop (100+100+100),

		 ; so loop will exit.
		
		include 'emu8086.inc'
		
		ORG 100h
		MOV AX, 0
		MOV CX, 5
		label1:
		PUTC '*'
		ADD AX, 100
		CMP AH, 0
		LOOPZ label1
		RET
		C Z S O P A
		unchanged

MOV	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
	
	SREG, memory
	memory, SREG
	REG, SREG
	SREG, REG
		 Copy operand2 to operand1.
		
		The MOV instruction cannot:
		
		• set the value of the CS and IP registers.
		• copy value of one segment register to another segment register (should copy to general register first).
		• copy immediate value to segment register (should copy to general register first).
		
		Algorithm:
		
		operand1 = operand2
		Example:
		
		ORG 100h
		MOV AX, 0B800h ; set AX = B800h (VGA memory).
		MOV DS, AX ; copy value of AX to DS.
		MOV CL, 'A' ; CL = 41h (ASCII code).
		MOV CH, 01011111b ; CL = color attribute.
		MOV BX, 15Eh ; BX = position on screen.
		MOV [BX], CX ; w.[0B800h:015Eh] = CX.
		RET ; returns to operating system.
		C Z S O P A
		unchanged

MOVSB	No operands	Copy byte at DS:[SI] to ES:[DI]. Update SI and DI.
		
		Algorithm:
		
		• ES:[DI] = DS:[SI]
		• if DF = 0 then
		  ○ SI = SI + 1
		  ○ DI = DI + 1
		
		else

		○ SI = SI - 1 ○ DI = DI - 1  Example: ORG 100h CLD LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSB RET a1 DB 1,2,3,4,5 a2 DB 5 DUP(0) C Z S O P A unchanged

MOVSW	No operands	 Copy **word** at DS:[SI] to ES:[DI]. Update SI and DI.
		
		Algorithm:
		
		• ES:[DI] = DS:[SI]
		• if DF = 0 then
		○ SI = SI + 2
		○ DI = DI + 2
		
		else
		
		○ SI = SI - 2
		○ DI = DI - 2
		
		Example:
		
		ORG 100h
		
		CLD
		LEA SI, a1
		LEA DI, a2
		MOV CX, 5
		REP MOVSW
		
		RET
		
		a1 DW 1,2,3,4,5
		a2 DW 5 DUP(0)
		C Z S O P A
		unchanged

MUL	REG memory	Unsigned multiply. Algorithm:

		 when operand is a **byte**:
		AX = AL * operand.
		when operand is a **word**:
		(DX AX) = AX * operand.
		Example:
		MOV AL, 200 ; AL = 0C8h
		MOV BL, 4
		MUL BL ; AX = 0320h (800)
		RET
		C Z S O P A
		r ? ? r ? ?
		CF=OF=0 when high section of the result is zero.

NEG	REG memory	 Negate. Makes operand negative (two's complement). Algorithm: • Invert all bits of the operand • Add 1 to inverted operand Example: MOV AL, 5 ; AL = 05h NEG AL ; AL = 0FBh (-5) NEG AL ; AL = 05h (5) RET C Z S O P A r r r r r r
NOP	No operands	 No Operation. Algorithm: • Do nothing Example: ; do nothing, 3 times: NOP NOP NOP RET C Z S O P A unchanged
NOT	REG memory	Invert each bit of the operand. Algorithm: • if bit is 1 turn it to 0. • if bit is 0 turn it to 1. Example:

		 MOV AL, 00011011b NOT AL ; AL = 11100100b RET C Z S O P A unchanged
OR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	 Logical OR between all bits of two operands. Result is stored in first operand. These rules apply: 1 OR 1 = 1 1 OR 0 = 1 0 OR 1 = 1 0 OR 0 = 0 Example: MOV AL, 'A' ; AL = 01000001b OR AL, 00100000b ; AL = 01100001b ('a') RET C Z S O P A 0 r r 0 r ?

OUT	im.byte, AL
	im.byte, AX
	DX, AL
	DX, AX
		
		Output from **AL** or **AX** to port.
		First operand is a port number. If required to access port number over 255 - **DX** register should be used.
		
		Example:
		MOV AX, 0FFFh ; Turn on all
		OUT 4, AX ; traffic lights.
		
		MOV AL, 100b ; Turn on the third
		OUT 7, AL ; magnet of the stepper-motor.
		C Z S O P A
		unchanged

POP	REG SREG memory	Get 16 bit value from the stack. Algorithm: • operand = SS:[SP] (top of the stack) • SP = SP + 2 Example: MOV AX, 1234h PUSH AX POP DX ; DX = 1234h RET C Z S O P A

		 unchanged
POPA	No operands	 Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack. SP value is ignored, it is Popped but not set to SP register). Note: this instruction works only on **80186** CPU and later! Algorithm: • POP DI • POP SI • POP BP • POP xx (SP value ignored) • POP BX • POP DX • POP CX • POP AX C Z S O P A unchanged
POPF	No operands	 Get flags register from the stack. Algorithm: • flags = SS:[SP] (top of the stack) • SP = SP + 2 C Z S O P A popped
PUSH	REG SREG memory immediate	Store 16 bit value in the stack. Note: **PUSH immediate** works only on 80186 CPU and later! Algorithm: • SP = SP - 2 • SS:[SP] (top of the stack) = operand Example: MOV AX, 1234h PUSH AX POP DX ; DX = 1234h RET C Z S O P A unchanged

		
PUSHA	No operands	 Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used. Note: this instruction works only on **80186** CPU and later! Algorithm: • PUSH AX • PUSH CX • PUSH DX • PUSH BX • PUSH SP • PUSH BP • PUSH SI • PUSH DI C Z S O P A unchanged
PUSHF	No operands	 Store flags register in the stack. Algorithm: • SP = SP - 2 • SS:[SP] (top of the stack) = flags C Z S O P A unchanged
RCL	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 left through Carry Flag. The number of rotates is set by operand2. When **immediate** is greater then 1, assembler generates several **RCL xx, 1** instructions because 8086 has machine code only for this instruction (the same principle works for all other shift/rotate instructions). Algorithm: shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position. Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCL AL, 1 ; AL = 00111001b, CF=0. RET C O

		 r r OF=0 if first operand keeps original sign.
RCR	memory, immediate REG, immediate memory, CL REG, CL	 Rotate operand1 right through Carry Flag. The number of rotates is set by operand2. Algorithm: shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position. Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCR AL, 1 ; AL = 10001110b, CF=0. RET C O r r OF=0 if first operand keeps original sign.

REP	chain instruction	 Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.
		
		Algorithm:
		
		check_cx:
		
		if CX <> 0 then
		
		• do following chain instruction
		• CX = CX - 1
		• go back to check_cx
		
		else
		
		• exit from REP cycle
		
		Z
		r

REPE	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 1 then: ○ go back to check_cx

		else ○ exit from REPE cycle else  • exit from REPE cycle Example: see **cmpsb.asm** in c:\emu8086\examples\. Z r

REPNE	chain instruction	
		Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times.
		Algorithm:
		check_cx:
		if CX <> 0 then
		• do following chain instruction
		• CX = CX - 1
		• if ZF = 0 then:
		○ go back to check_cx
		else
		○ exit from REPNE cycle
		else
		• exit from REPNE cycle
		Z
		r

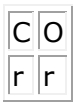
REPNZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 0 then:

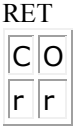
		 ○ go back to check_cx else ○ exit from REPNZ cycle else • exit from REPNZ cycle 

REPZ	chain instruction	 Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Zero), maximum CX times.
		Algorithm:
		check_cx:
		if CX <> 0 then
		• do following chain instruction
		• CX = CX - 1
		• if ZF = 1 then:
		○ go back to check_cx
		else
		○ exit from REPZ cycle
		else
		• exit from REPZ cycle
		 

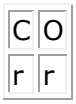
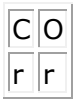
RET	No operands or even immediate	Return from near procedure. Algorithm: • Pop from stack: ○ IP • if immediate operand is present: SP = SP + operand Example: ORG 100h ; for COM file. CALL p1

		 ADD AX, 1 RET ; return to OS. p1 PROC ; procedure declaration. MOV AX, 1234h RET ; return to caller. p1 ENDP C Z S O P A unchanged
RETF	No operands or even immediate	 Return from Far procedure. Algorithm: • Pop from stack: ○ IP ○ CS • if immediate operand is present: SP = SP + operand C Z S O P A unchanged
ROL	memory, immediate REG, immediate memory, CL REG, CL	 Rotate operand1 left. The number of rotates is set by operand2. Algorithm: shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position. Example: MOV AL, 1Ch ; AL = 00011100b ROL AL, 1 ; AL = 00111000b, CF=0. RET C O r r OF=0 if first operand keeps original sign.
ROR	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 right. The number of rotates is set by operand2. Algorithm: shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position. Example: MOV AL, 1Ch ; AL = 00011100b ROR AL, 1 ; AL = 00001110b, CF=0. RET C O

		 OF=0 if first operand keeps original sign.
SAHF	No operands	 Store AH register into low 8 bits of Flags register. Algorithm: flags register = AH AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. 
SAL	memory, immediate REG, immediate memory, CL REG, CL	 Shift Arithmetic operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 0E0h ; AL = 11100000b SAL AL, 1 ; AL = 11000000b, CF=1. RET  OF=0 if first operand keeps original sign.

SAR	memory, immediate
	REG, immediate
	
	memory, CL
	REG, CL
		
		Shift Arithmetic operand1 Right. The number of shifts is set by operand2.
		
		Algorithm:
		
		• Shift all bits right, the bit that goes off is set to CF.
		• The sign bit that is inserted to the left-most position has the same value as before shift.
		
		Example:
		MOV AL, 0E0h ; AL = 11100000b
		SAR AL, 1 ; AL = 11110000b, CF=0.
		
		MOV BL, 4Ch ; BL = 01001100b
		SAR BL, 1 ; BL = 00100110b, CF=0.
		
		RET
		 
		OF=0 if first operand keeps original sign.

SBB	REG, memory memory, REG REG, REG memory, immediate REG, immediate	 Subtract with Borrow. Algorithm: operand1 = operand1 - operand2 - CF Example: STC MOV AL, 5 SBB AL, 3 ; AL = 5 - 3 - 1 = 1 RET C Z S O P A r r r r r r
SCASB	No operands	 Compare bytes: AL from ES:[DI]. Algorithm: • ES:[DI] - AL • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ DI = DI + 1 else ○ DI = DI - 1 C Z S O P A r r r r r r
SCASW	No operands	 Compare words: AX from ES:[DI]. Algorithm: • ES:[DI] - AX • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ DI = DI + 2 else ○ DI = DI - 2 C Z S O P A r r r r r r
SHL	memory,	Shift operand1 Left. The number of shifts is set by operand2.

	immediate REG, immediate memory, CL REG, CL	 Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 11100000b SHL AL, 1 ; AL = 11000000b, CF=1. RET  OF=0 if first operand keeps original sign.
SHR	memory, immediate REG, immediate memory, CL REG, CL	Shift operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • Zero bit is inserted to the left-most position.  Example: MOV AL, 00000111b SHR AL, 1 ; AL = 00000011b, CF=1. RET  OF=0 if first operand keeps original sign.
STC	No operands	 Set Carry flag. Algorithm: CF = 1 
STD	No operands	 Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 1 

STI	No operands	 Set Interrupt enable flag. This enables hardware interrupts. Algorithm: IF = 1 I 1

STOSB	No operands	Store byte in AL into ES:[DI]. Update DI.
		  Algorithm:
		• ES:[DI] = AL
		• if DF = 0 then
		  ○ DI = DI + 1
		else
		  ○ DI = DI - 1
		Example:
		ORG 100h
		
		LEA DI, a1
		MOV AL, 12h
		MOV CX, 5
		
		REP STOSB
		
		RET
		
		a1 DB 5 dup(0)
		C Z S O P A
		unchanged

STOSW	No operands	Store word in AX into ES:[DI]. Update DI.
		Algorithm:
		• ES:[DI] = AX
		• if DF = 0 then
		  ○ DI = DI + 2
		else
		  ○ DI = DI - 2
		Example:
		ORG 100h

		 LEA DI, a1
		MOV AX, 1234h
		MOV CX, 5
		
		REP STOSW
		
		RET
		
		a1 DW 5 dup(0)
		C Z S O P A
		unchanged

SUB	REG, memory
	memory, REG
	REG, REG
	memory, immediate
	REG, immediate
		 Subtract.
		
		Algorithm:
		
		operand1 = operand1 - operand2
		
		Example:
		MOV AL, 5
		SUB AL, 1 ; AL = 4
		
		RET
		C Z S O P A
		r r r r r r

TEST	REG, memory memory, REG REG, REG memory, immediate REG, immediate	 Logical AND between all bits of two operands for flags only. These flags are effected: **ZF, SF, PF.** Result is not stored anywhere. These rules apply: 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Example: MOV AL, 00000101b TEST AL, 1 ; ZF = 0. TEST AL, 10b ; ZF = 1. RET C Z S O P 0 r r 0 r
XCHG	REG, memory memory, REG REG, REG	Exchange values of two operands. Algorithm: operand1 < - > operand2

		
		Example:
		MOV AL, 5
		MOV AH, 2
		XCHG AL, AH ; AL = 2, AH = 5
		XCHG AL, AH ; AL = 5, AH = 2
		RET
		C Z S O P A
		unchanged

XLATB	No operands	 Translate byte from table. Copy value of memory byte at DS:[BX + unsigned AL] to AL register. Algorithm: AL = DS:[BX + unsigned AL] Example: ORG 100h LEA BX, dat MOV AL, 2 XLATB ; AL = 33h RET dat DB 11h, 22h, 33h, 44h, 55h C Z S O P A unchanged
XOR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	 Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand. These rules apply: 1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0 Example: MOV AL, 00000111b XOR AL, 00000010b ; AL = 00000101b RET C Z S O P A 0 r r 0 r ?

emu8086 Assembler - Frequently Asked Questions
The Microprocessor Emulator and 8086 Integrated Assembler

Please make sure you have the latest version of **EMU8086**
(if unsure click **help -> check for an update...** from the menu)
The solutions may not work in previous versions of the emulator/assembler.

General recommendation for Windows XP users:

- **1.** click Start.
- **2.** click Run.
- **3.** type "explorer"
- **4.** select from the menu "Tools" -> "Folder Options".
- **5.** click "View" tab.
- **6.** select "Show hidden files and folders".
- **7. un**check "Hide extensions for known file types".

To step forward press **F8** key, to run forward press **F9** or press and hold **F8**. To step backward press **F6** key, to run backward press and hold **F6**. The maximum number of steps-back can be set in **emu8086.ini**. For example:

MAXIMUM_STEPS_BACK=default ; by default it is set to 200 for a better performance.

or

MAXIMUM_STEPS_BACK=1000 ; this value should not be over 32767.

Question:

Why this code doesn't work?

```
org 100h

    myArray dw 2, 12, 8, 52, 108

    mov si, 0
    mov ax, myArray[si]

    ret
```

Solution:

There should be a jump over the variables/array declaration:

```
org 100h

jmp code

    myArray dw 2, 12, 8, 52, 108

code:  mov si, 0
    mov ax, myArray[si]
```

```
    ret
```

For the computer all bytes look the same, it cannot determine if it's an instruction or a variable. Here is an example of **MOV AL, 5** instruction that can be coded with simple variable declarations:

```
org 100h

byte1 db 176
byte2 db 5

ret
```

When you run this program in emulator you can see that bytes **176** and **5** are actually assembled into:

```
MOV AL, 5
```

This is very typical for Von Neumann Architecture to keep data and instructions in the same memory, It's even possible to write complete program by using only **DB** (define byte) directive.

```
org 100h

db 235  ; jump...
db 6    ;     6           -  six bytes forward (need to skip characters)
db 72   ; ascii code of 'H'
db 101  ; ascii code of 'e'
db 108  ; ascii code of 'l'
db 108  ; ascii code of 'l'
db 111  ; ascii code of 'o'
db 36   ; ascii code of '$'  - DOS function prints untill dollar.
db 186  ; mov DX,   ....     - DX is word = two bytes
db 2    ;           02     - little end
db 1    ;          01      - big end
db 180  ; mov AH, ....
db 9    ;          09
db 205  ; int ...
db 33   ;    21h           - 33 is 21h (hexadecimal)
db 195  ; ret              - stop the program.
```

8086 and all other Intel's microprocessors store the least significant byte at a lower address. **102h** is the address of '**H**' character = **org 100h** + **2 bytes** (jmp instruction). The above assembly code produces identical machine code to this little program:

```
org 100h

jmp code

    msg db 'Hello$'

code:  mov DX, offset msg
    mov AH, 9
    int 21h

    ret
```

If you open the produced "**.com**" file in any hex editor you can see hexadecimal values, every byte takes two hexadecimal digits, for example 235 = **EB**, etc... memory window of the emulator shows both hexadecimal and decimal values.

Problem:

The screen fonts are too small or too big?...

Solution:

The latest version of the emulator uses Terminal font by default and it is MSDOS/ASCII compatible.It is also possible to set the screen font to **Fixedsys** from the **options**. For other controls the font can be changed from c:\emu8086\emu8086.ini configuration file. It is well known that on some localized versions of Windows XP the Terminal font may be shown significantly smaller than in original English version. The latest version automatically changes default font to 12 unless it is set in emu8086.ini: **FIX_SMALL_FONTS=false**. The Fixedsys font is reported to be shown equally on all systems. It is reported that for small Terminal font D and 0 (zero) look very alike.

Starting from version 4.00-Beta-8 the integrated assembler of emu8086 can be used from command line. The switch is **/a** followed by a full path to assembly source code files. The assembler will assemble all files that are in source folder into **MyBuild** directory.

For example:

emu8086 /a c:\emu8086\examples

Note: any existing files in c:\emu8086\MyBuild\ subdirectory are to be overwritten. The assembler does not print out the status and error messages to console, instead it prints out everything to this file:
c:\emu8086\MyBuild_emu8086_log.txt

Do not run several instances of the assembler under the same path until <END> appears in the file. You may see if emu8086 is running by pressing the Ctrl+Alt+Del combination, or by just opening and reopening **_emu8086_log.txt** file in Notepad to see if the file is written completely. This can be checked automatically by another program (the file must be opened in shared mode).

The assembler does not save files with extensions **.com**, **.exe**, or **.bin**, instead it uses these extensions: **.com_**, **.exe_**, or **.bin_** (there is underline in the end). If you'd like to run the file for real just rename **.com_** to **.com** etc.

For batch rename just type:
ren *.com_ *.com

Theoretically it's possible to make a high level compiler that will use emu8086 as an assembler to generate the byte code. Maybe even C or C++ compiler. The example of a basic compiler program written in pure 8086 code may be available in the future.

To disable little status window in the lower right corner, set **SILENT_ASSEMBLER=true** in emu8086.ini

For the emulator physical drive **A:** is this file **c:\emu8086\FLOPPY_0** (for BIOS interrupts: INT 13h and boot).

For DOS interrupts (INT 21h) drive **A:** is emulated in this subdirectory:
c:\emu8086\vdrive\a

- assembler - the one who assembles, what ever in what ever, we generally refer to bytes and machine code.
- integrated - not disintegrated, i.e. all parts work together and supplement each other.
- compiler - the one who compiles bytes, it may use the assembler to make its job easier

and faster.

Question:

How do I print a result of a sum of two numbers?

Solution:

There are two general solutions to this task, small and **big**.

Short *"Macro Assembly"* solution:

```
; it is a much shorter solution, because procedures are hidden inside the include file.

include "emu8086.inc"

ORG    100h

MOV AX, 27
MOV BX, 77
ADD AX, BX

;now I will print the result which is in AX register

CALL PRINT_NUM

ret

DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS

end
```

For more information about macro definitions check out **tutorial 5**.

(source code of emu8086.inc is available - click **here** to study)
emu8086.inc is an open source library, you can dismantle, modify and use its procedures directly
in your code instead of using **include** directives and tricky macro definitions.
The procedure the prints the simple numeric value can take several hundreds of lines,
you can use this library as a short-cut; you can find actual assembly language code that does the printing
if you open **emu8086.inc** and search for DEFINE_PRINT_NUM and DEFINE_PRINT_NUM_UNS inside of it,
they look exactly as the first example, the advantage of macros is that many programs can use it keeping
their code relatively small.

Question:
How to calculate the number of elements in array?

Solution: The following code calculates the array size:

```
jmp start:
  array db 17,15,31,16,1,123, 71

  array_byte_size = $ - offset array

start:
  MOV AX, array_byte_size
```

$ is the location counter, it is used by the assembler to calculate locations of labels and variables.
note: this solution may not work in older versions of emu8086 integrated assembler, you can download an update **here**. the result is always in bytes. If you declare an array of words you need to divide the result by two, for example:

```
jmp start:
  array dw 12,23,31,15,19,431,17

  array_byte_size = $ - offset array

start:
  MOV AX, array_byte_size / 2
```

the remainder is always zero, because the number of bytes is even.

Question:

How can I do a far call, or is it not supported in the emulator?

```
mov bx,0h ;set es:bx to point to int 10h vector in ivt
mov es,bx
mov bx,40h
mov ah,0eh          ; set up int 10h params
mov al, 1           ; ASCII code of a funny face
pushf
call es:[bx]        ; do a far call to int10h vector ( wrong )
ret
end
```

Solution:

```
mov bx,0h           ; set es:bx to point to int 10h vector in ivt
mov es,bx
mov bx,40h
mov ah,0eh          ; set up int 10h params
mov al, 1           ; ASCII code of a funny face.
pushf
call far es:[bx]    ; do a far call to int10h vector
ret
end
```

Without **far** prefix the microprocessor sets a word value (2 bytes) that is pointed by es:[BX] to IP register; with **far** prefix microprocessor sets the word value that is pointed by es:[BX] to IP register, and the word at es:[BX+2] is set to CS register.

Question:

Is there another way to achieve the same result without using DD variables?

Solution:

DD definitions and far jumps are supported in the latest version, for example:

```
jmp far addr

addr dd 1235:5124h
```

If you are using earlier version of emu8086 you can use a workaround, because double words are really two 16 bit words, and words are really two bytes or 8 bits, it's possible to code without using any other variables but bytes. In other words, you can define two DW values to make a DD.

For example:

```
ddvar dw 0
      dw 0
```

Long jumps are supported in the latest version (**call far**). For previous versions of emu8086 there is another workaround:

This code is compiled perfectly by all versions:

```
jmp 1234h:4567h
```

and it is assembled into byte sequence:

```
 EA 67 45 34 12
```

It can be seen in memory window and in **emulator -> debug**.

Therefore, you can define in your code something similar to this code:

```
    db    0EAh              ; long jump instruction opcode.
oft dw    4567h               ; jump offset
sg  dw    1234h               ; jump segment
```

The above code is assembled into the same machine code, but allows you to modify the jump values easily and even replace them if required, for exampe:

```
mov cs:oft, 100h
```

when executed the above instruction modifies the upper code into:

```
jmp 1234h:100h
```

this is just a tiny example of self-modifying code, it's possible to do anything even without using DD (define double word) and segment overrides, in fact it is possible to use DB (define byte) only, because DW (define word) is just two DBs. it is important to remember that Intel architecture requires the little end of the number to be stored at the lower address, for example the value **1234h** is combined of two bytes and it is stored in the memory as **3412**.

```
org 100h

mov ax, 0
mov es, ax

mov ax, es:[40h]
mov word_offset, ax

mov ax, es:[40h+2]
mov word_segment, ax
```

```
mov ah,0eh ; set up parameters for int 10h
mov al,1  ; ASCII code of a funny face.

; do same things as int does
pushf
push cs
mov bx, rr
push bx

opcode      db 0EAh   ; jmp word_segment:word_offset
word_offset  dw ?
word_segment dw ?

rr:
mov ax, 1  ; return here

ret

end
```

Question:

It would be very useful to have the option of invoking a DOS shell at the build directory from the compile finished dialogue.

Solution:

The latest version of emu8086 has external button that allows to launch command prompt or debug.exe with preloaded executable and it also allows to run executables in real environment. for previous versions of emu8086 you can download Microsoft utility called **command prompt here**, after the compilation click **browse...**, to open **C:\emu8086\MyBuild** folder in file manager, then **right-click** this folder and select "**open command prompt here**" from the pop-up menu.

Question:

Is it possible to set a break point?

Answer:

Yes, it's possible to click the instruction line and click **Set break point** from **Debug** menu of the emulator.
It is also possible to keep a log similar to **debug** program, if you click **View -> Keep Debug Log**.
The break point is set to currently selected address (segment:offset).
The emulator will stop running when the physical address of CS:IP registers is equivalent to break point address (note: several effective address may represent the same physical address, for example **0700:114A = 0714:000A)**
Another way to set a break point is to click **debug -> stop on condition** and set value of IP register. The easiest way to get IP values is from the listing under LOC column. To get listing click **debug -> listing**
In addition it's possible to the emulator to stop on codition **AX = 1234h** and to put the follwoing lines in several places of your code:
MOV AX, 1234h
MOV AX, 0

Question:

I am aware that 8086 is limited to 32,767 for positive and to -32,768 for negative. I am aware that this is the 16-bit processor, that was used in earlier computer systems, but even in 8-bit Atari 2600 score counts in many games went into the 100,000s, way beyond 32,000.

Solution:

Here is the example that calculates and displays the sum of two 100-bit values (30 digits).
32 bits can store values up to: 4,294,967,296 because **2^32 = 4294967296** (this is only 10 decimal digits).
100 bits can hold up to 31 decimal digits because **2^100 = 1267650600228229401496703205376**
(31 decimal digits = 100 binary digits = 100 bits)

```
; this example shows how to add huge unpacked BCD numbers (BCD is binary coded decimal).

; this allows to over come the 16 bit and even 32 bit limitation.
; because 32 digit decimal value holds over 100 bits!
; the number of digits in num1 and num2 can be easily increased.

ORG     100h

; skip data:
JMP     code

; the number of digits in numbers:
; it's important to reserve 0 as most significant digit, to avoid overflow.
; so if you need to operate with 250 digit values, you need to declare len = 251
len     EQU     32

; every decimal digit is stored in a separate byte.

; first number is: 42345461236123451234535179521
num1    DB      0,0,4,2,3,4,5,4,6,1,2,3,6,1,2,3,4,5,1,2,3,4,4,5,3,5,1,7,9,5,2,1
; second number is: 71237884777198112351313788249 8
num2    DB      0,0,7,1,2,3,7,8,8,4,7,7,7,1,9,8,1,1,2,3,5,1,3,1,3,7,8,8,2,4,9,8

; we will calculate this:

; sum = num1 + num2

; 42345461236123451234535179521 + 71237884777198112351313788249 8 =
;             = 113583346013321563585767306 2019

sum     DB      len dup(0) ; declare array to keep the result.

; you may check the result on paper, or click Start , then Run, then type "calc" and hit enter key.

code:   nop ; the entry point.

; digit pointer:
XOR     BX, BX

; setup the loop:
MOV     CX, len
MOV         BX, len-1  ; point to lest significant digit.

next_digit:
```

```
        ; add digits:
        MOV     AL, num1[BX]
        ADC     AL, num2[BX]

        ; this is a very useful instruction that
        ; adjusts the value of addition
        ; to be string compatible
        AAA

        ; AAA stands for ASCII ADD ADJUST.
        ; --- algorithm behind AAA ---
        ; if low nibble of AL > 9 or AF = 1 then:
                ;     AL = AL + 6
                ;     AH = AH + 1
                ;     AF = 1
                ;     CF = 1
                ; else
                ;     AF = 0
        ;     CF = 0
        ;
        ; in both cases: clear the high nibble of AL.
        ; --- end of AAA logic ---

        ; store result:
        MOV     sum[BX], AL

        ; point to next digit:
        DEC     BX

        LOOP    next_digit

; include carry in result (if any):
ADC     sum[BX], 0


; print out the result:
MOV     CX, len

; start printing from most significant digit:
MOV     BX, 0

print_d:
        MOV     AL, sum[BX]
        ; convert to ASCII char:
        OR      AL, 30h

        MOV     AH, 0Eh
        INT     10h

        INC     BX

        LOOP    print_d

RET

END
```

With some more diligence it's possible to make a program that inputs 200 digit values and prints out their sum.

Question:

I'm making an interrupt counter; for that I am using 1 phototransister and sdk-86 board at college. I am not having this kit at home so I have a problem to see the output.
here is issue.: when light on phototransister is on and off pulse is generated, this pulse comes just like the harwared iterrupt. my program must to count these pulses continuously; for that I am using 8255kit and SDK-86kit at college, but at home I don't have this equempent at home. Am I able to emulate the output of real system? Perchanps, I have to develope 8255 device as an externel device in emu8086; but how can I prog this device in vb? I am using ports: 30h, 31h, 32h, and 33h. I dont know vb...

Answer:

You don't have to know vb, but you have to know any real programming language apart from html/javascript. the programming language must allow the programmer to have complete control over the file input/output operations, then you can just open the file **c:\emu8086.io** in shared mode and read values from it like from a real i/o port. byte at offset 30h corresponds to port 30h, word at offset 33h corresponds to port 33h. the operating system automatically caches files that are accessed frequently, this makes the interaction between the emulator and a virtual device just a little bit slower than direct memory-to-memory to communication. in fact, you can create 8255 device in 16 bit or even in 32 bit assembly language.

Note: the latest version supports hardware interrupts: **c:\emu8086.hw**, setting a none-zero value to any byte in that file triggers a hardware interrupt. the emulator must be running or step button must be pressed to process the hardware interrupt. For example:

```
idle:
        nop
jmp idle
```

Question:

I want to know about memory models and segmentation and memory considerations in embedded systems.

Answer:

You may find these links helpful:

- **A feel for things**.
- **Advanced Embedded X86 Programming: Protection and Segmentation**.
- **Embedded X86 Programming: Protected Mode**.
- **Micro Minis**.
- **RISCy Business**.
- **In search of a common API for connected devices**.
- **Taming the x86 beast**.
- **Intel 8086 Family Architecture**.

Question:

What physical address corresponds to DS:103Fh if DS=94D0h

Answer:

94D0h * 10h + 103Fh = 95D3Fh
and it's equivalent to effective address: 95D3h:000Fh

it's possible to use emu8086 integrated calculator to make these calculations (set *show result* to hex).

note: 10h = 16

Question:

I would like to print out the assembly language program as well as the corresponding machine language code. How can I do so ?

Solution:

It is not possible to print out the source code directly from emu8086, but you may click **file** -> **export to HTML...** and print it from the browser or even upload it to the server preserving true code colors and allowing others just to copy & paste it.

The corresponding machine code can be opened and then printed out by clicking **view -> listing** right after the successful assembling/compilation or from the emulator's menu.

Question:

Can we use breakpoint int 03h with emu 8086?

Answer:

It is possible to overwrite the default stub address for int 03h in interrupt vector table with a custom function. And it is possible to insert **CC** byte to substitute the first byte of any instruction, however the easiest way to set a break point is to click an instruction and then click **debug -> set break point** from the menu.

Editor hints:

- To repeat a successful text search press F3 key.
- To cut a line press **Ctrl + Y** simultaneously.
- Free positioning of the text cursor can be turned off from the options by checking **confine caret to text**.

65535 and **-1** are the same 16 bit values in binary representation: 1111111111111111b as **254** and **-2** have the same binary code too: 11111110b

Question:

It is good that emu8086 supports virtual devices for emulating the io commands. But how does the IO work for real? (Or: How do I get the Address of a device e.g. the serial port)

Answer:

It is practically the same. The device conrolling is very simple. You may try searching for "**PC PhD: Inside PC Interfacing**". The only problem is the price. It's good if you can afford to buy real devices or a CPU workbench and experiment with the real things. However, for academic and educational purpoces, the emulator is much cheaper and easier to use, plus you cannot nor burn nor shortcut it. Using emu8086 technology anyone can make free additional hardware devices. Free hardware easy - in any programming language.

Question:

How do I set the output screen to 40*25, so I dont have to resize it everytime it runs.

Answer:

```
mov ax, 0
int 10h
```

It's possible to change the colours by clicking the "options" button. The latest version uses yellow color to select lines of bytes when the instruction in disassembled list is clicked, it shows exactly how many bytes the instruction takes. The yellow background is no longer recommended to avoid the confusion.

Instead of showing the offset the emulator shows the physical address now. You can easily calculate the offset even without the calculator, because the loading segment is always 0700 (unless it's a custom .bin file), so if physical address is 07100 then the offset is 100 and the segment is 700.

The file system emulation is still undergoing heavy checks, there are a few new but undocumented interrupts. INT 21h/4Eh and INT 21h/4Fh. These should allow to get the directory file list.

Question:
What is **org 100h** ?

Answer:

First of all, it's a directive which instructs the assembler to build a simple **.com** file. unlike instructions, this directive is not converted into any machine code. **com** files are compatible with DOS and they can run in Windows command prompt, and it's the most tiny executable format that is available.

Literally this directive sets the **location counter** to 256 (100h). Location counter is represented in source code as dollar. This is an example of how location counter value can be accessed: **MOV AX, $** the execution of this instruction will make **AX** contain the address of instruction that put this address in it.... but usually, it's not something to worry about, just remember that **org 100h** must be the first line if you want to make a tiny single segment executable file. note: dollar inside "$" or '$' is not a location counter, but an ASCII character.

Location counter has nothing to do with string terminating "$" that is historically used by MS-DOS print functions.

Question:

What is **org 7c00h** ?

Answer:

It is very similar to **org 100h**. This directive instructs the assembler to add **7C00h** to all addresses of all variables that are declared in your program. It operates exactly the same way as **ORG 100h** directive, but instead of adding **100h** (or 256 bytes) it adds **7C00h**.

For example if you write this code:

```
mov ax, var1
```
and the address of **var1** is **10h**

without **ORG 100h** directive assembler produces the following machine code:

```
mov ax, [10h]
```
however with **ORG 100h** directive assembler automatically updates the machine code to:

```
mov ax, [10h+100h]
```
and it is equivalent to this code:

```
mov ax, [110h]
```
org 7C00h directive must be used because the computer loads boot records into the memory at address **0000:7C00**.

If program is not using variable names and only operates directly with numeric address values (such as [**2001h**] or [**0000:1232h**]... etc, and not **var1**, **var2**...) and it does not use any labels then there is no practical use for **ORG** directive. generally it's much more convenient to use names for specific memory locations (variables), for these cases **ORG** directive can save a lot of time for the programmer, by calculating the correct offset automatically.

Notes:

- **ORG** directive does not load the program into specific memory area.
- Misuse of **ORG** directive can cause your program not to operate correctly.
- The area where the boot module of the operating system is loaded is defined on hardware level by the computer system/BIOS manufacture.
- When **.com** files are loaded by DOS/prompt, they are loaded at any available segment, but offset is always 100h (for example 12C9:0100).

Question:
Where is a numeric Table of Opcodes?

Answer:

A list of all 8086 CPU compatible instructions is published here (without numeric opcodes). Only those instructions that appear both in Pentium ® manuals and in this reference may be used for 8086 microprocessor. For a complete set of opcodes and encoding tables please check

out:

The Greatest Resources

IA-32 Intel ® Architecture Software Developer Manuals

- **Basic Architecture:**
- Instruction Set Summary, 16-bit Processors and Segmentation (1978), System Programming Guide
- **http://download.intel.com/design/Pentium4/manuals/25366517.pdf**
- **System Programming Guide:**
- 8086 Emulation, Real-Address Mode:
- **http://download.intel.com/design/Pentium4/manuals/25366817.pdf**
- **Instruction Set Reference:**
- Only 16 bit instructions may run on the original 8086 microprocessor.
- Part 1, Instruction Format, Instructions from A to M:
- **http://download.intel.com/design/Pentium4/manuals/25366617.pdf**
- Part 2, Instructions from N to Z, Opcode Map, Instruction Formats and Encodings:
- **http://download.intel.com/design/Pentium4/manuals/25366717.pdf**

AMD64 ® Architecture Programmer Manuals

- **Application Programming:**
- Overview of the AMD64 Architecture:
- Memory Model and Memory Organization, Registers, Instruction Summary:
- **http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf**
- **System Programming:**
- Figures, Tables, x86 and AMD64 Operating Modes, Memory Model:
- **http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf**

- **General-Purpose Instructions and System Instructions:**
- Only 16 bit instructions are compatible with the original 8086 CPU.
- Instruction Byte Order, General-Purpose Instruction Reference, Opcode and Operand Encodings,
- **http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf**
-

Notes about I/O port emulation for c:\emu8086.io

It is not recommended to use two neighbouring 16 bit ports, for example port 0 and port 1.
Every port has a byte length (8 bit), two byte port (16 bit word) is emulated using 2 bytes or 2 byte ports.
When the emulator outputs the second word it overwrites the high byte of the first word.

```
; For example:
MOV AL, 34h
OUT 25, AL
MOV AL, 12h
OUT 26, AL
; is equvalent to:
MOV AX, 1234h
OUT 25, AX
```

Question:

```
; I am trying to compile the following:

org 256

mov dx, bugi

ret

bugi db 55

; The variable _bugi_ is a byte while DX is
; a word but the integrated assembler does not complain. Why?
```

Answer:

To make the integrated assembler to generate more errors you may set:

STRICT_SYNTAX=true

in this file:

C:\emu8086\emu8086.ini

By default it is set to false to enable coding a little bit faster without the necessity to use "**byte ptr**" and "**word ptr**" in places where it is clear without these long constructions (i.e. when one of the operands is a register).

Note: the settings in emu8086.ini do not apply to fasm (flat assembler). To use fasm add **#fasm#** or any valid **format** directive (valid for emu8086 version 4.00-Beta-15 and above)

For differences between the integrated assembler (MASM/TASM compatible) and FASM see fasm_compatibility.asm
FASM does not require the **offset** directive. By default all textual labels are offsets (even if defined with DB/DW)
To specify a variable **[ ]** must be put around it.

To avoid conflicts between 8086 integrated assembler and fasm, it is recommended to place this directive on top of all files that are designed for flat assembler:
#fasm#

Question:

I've installed emu8086 on several computers in one of my electronics labs. Everything seems to work correctly when I am logged onto any of the PC's but, when any of the students log on, the virtual device programs controlled by the example ASM programs do not respond. ex; using LED_display_test.ASM.

The lab is set up with Windows XP machines on a domain. I have admin privileges but the students do not. I tried setting the security setting of **C:\emu8086** so all users have full privileges but it did not help. Are there other folders that are in play when the program is running?

Solution:

In order for virtual devices to work correctly, it is required to set READ/WRITE privileges for these files that are created by the emulator in the root folder of the drive C:
C:\emu8086.io
c:\emu8086.hw

These files are are used to communicate between the virtual devices and the emulator, and it should be allowed for programs that run under students' login to create, read and write to and from these files freely.

To see simulated memory - click emulator's "aux" button and then select "memory" from the popup menu.

1_sample.asm

```
name "hi-world"

; this example prints out  "hello world!"
; by writing directly to video memory.
; in vga memory: first byte is ascii character, byte that follows is character attribute.
; if you change the second byte, you can change the color of
; the character even after it is printed.
; character attribute is 8 bit value,
; high 4 bits set background color and low 4 bits set foreground color.

; hex    bin       color
;
; 0      0000      black
; 1      0001      blue
; 2      0010      green
; 3      0011      cyan
; 4      0100      red
; 5      0101      magenta
; 6      0110      brown
; 7      0111      light gray
; 8      1000      dark gray
; 9      1001      light blue
; a      1010      light green
; b      1011      light cyan
; c      1100      light red
; d      1101      light magenta
; e      1110      yellow
; f      1111      white



org 100h

; set video mode
mov ax, 3     ; text mode 80x25, 16 colors, 8 pages (ah=0, al=3)
int 10h       ; do it!
```

```
; cancel blinking and enable all 16 colors:
mov ax, 1003h
mov bx, 0
int 10h


; set segment register:
mov     ax, 0b800h
mov     ds, ax

; print "hello world"
; first byte is ascii code, second byte is color code.

mov [02h], 'h'

mov [04h], 'e'

mov [06h], 'l'

mov [08h], 'l'

mov [0ah], 'o'

mov [0ch], ','

mov [0eh], 'w'

mov [10h], 'o'

mov [12h], 'r'

mov [14h], 'l'

mov [16h], 'd'

mov [18h], '!'




; color all characters:
mov cx, 12  ; number of characters.
mov di, 03h ; start from byte after 'h'

c:  mov [di], 11101100b   ; light red(1100) on yellow(1110)
    add di, 2 ; skip over next ascii code in vga memory.
    loop c

; wait for any key press:
mov ah, 0
int 16h
```

```
name "add-sub"

org 100h

mov al, 5       ; bin=00000101b
mov bl, 10      ; hex=0ah or bin=00001010b

; 5 + 10 = 15 (decimal) or hex=0fh or bin=00001111b
add bl, al

; 15 - 1 = 14 (decimal) or hex=0eh or bin=00001110b
sub bl, 1

; print result in binary:
mov cx, 8
print: mov ah, 2   ; print function.
       mov dl, '0'
       test bl, 10000000b  ; test first bit.
       jz zero
       mov dl, '1'
zero:  int 21h
       shl bl, 1
loop print

; print binary suffix:
mov dl, 'b'
int 21h

; wait for any key press:
mov ah, 0
int 16h

ret
```

string.

Object code = 1111 011Z, Z is donot care bit. It may be 1 or 0

(ii) REPE (or REPZ). Code = F3. Repeat if CX is not and ZF = 1

REPE and REPZ are two mnemonics for the same instruction. REPE or REPZ is a prefix instruction which is used before the string instruction CMPS or SCAS. It repeats the string instruction when CX is not zero and zero flag ZF = 1. Repetition stops when CX is decremented to 0 or ZF = 0.

(iii) REPNE (or REPNZ). Repeat if CX is not zero and ZF = 0. Code = F2

REPNE and REPNZ are two mnemonics for the same instruction. REPNE or REPNZ is a prefix instruction which is often used before the string instruction SCAS. It repeats the string instruction when CX is not zero and ZF = 0. Repetition stops when CX is decremented to 0 or ZF = 1.

11.6 EXAMPLES OF 8086 ASSEMBLY LANGUAGE PROGRAMS

11.6.1 Largest Number In a Data Array

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	B8,00,00		MOV	AX, 0000	; initial value for comparison
0104	BE,00,02		MOV	SI, 0200	; memory address in SI
0107	8B,0C		MOV	CX , [SI]	; count in CX
0109	46	BACK	INC	SI	; increment SI
010A	46		INC	SI	; increment SI
010B	3B,04		CMP	AX , [SI]	; compare previous largest number with next number
010D	73,02		JAE	GO	; jump if number in AX is greater i.e. CF = 0
010F	8B, 04		MOV	AX,[SI]	; save next larger number in AX
0111	E2, F6	GO	LOOP	BACK	; jump to BACK until CX becomes zero
0113	A3, 51,02		MOV	[0251], AX	; store largest number in memory
0116	CC		INT 3		; interrupt program

The given data array is : 8341, 7258, 4639, 8453, 9630. These are hexadecimal numbers.

Since there are five 16-bit numbers in a given data array, the count is 0005. Two consecutive memory locations store a 16-bit number i.e. two bytes of a 16-bit number. The data fed in the memory for the above program are as given below.

Data	Result
0200 – 05H	0251 – 30H
0201 – 00H	0252 – 96H
0202 – 41H	
0203 – 83H	
0204 – 58H	
0205 – 72H	
0206 – 39H	
0207 – 46H	
0208 – 53H	
0209 – 84H	
020A – 30H	
020B – 96H	

A 16-bit number is stored in two consecutive memory locations. The result is 9630H being the largest number in the given data array. It is stored in the memory locations 0251H and 0252H. At 010D effective memory address, 73H is the opcode of JAE instruction. 02 is displacement at the effective memory location 010E. The program execution jumps from the effective memory location 010E to 0111H, if the carry flag CF = 0. Therefore, the displacement is 02. There is LOOP instruction at the effective memory location 0111H. The program execution jumps from the effective memory location 0112H to 0109H. In this case the displacement is ten memory locations back *i.e.* the displacement is – 0A. The 2's complement of 0A is F6 which has been given at the effective memory address 0112H.

11.6.2 Smallest Number In a Data Array

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	B8,FF,FF		MOV	AX, FFFF	; Initial value for comparison
0104	BE,00,02		MOV	SI, 0200	; Memory address in SI
0107	8B,0C		MOV	CX , [SI]	; Count in CX
0109	46	BACK	INC	SI	; Increment SI
010A	46		INC	SI	; Increment SI
010B	3B,04		CMP	AX, [SI]	; Compare previous smallest with next number
010D	72,02		JB	GO	; Jump if number in AX is smaller i.e. CF=1
010F	8B,04		MOV	AX , [SI]	; Save next smaller
0111	E2,F6	GO	LOOP	BACK	; Jump to BACK until CX becomes zero
0113	A3,51,02		MOV	[0251] , AX	; Store smallest number in memory
0116	CC		INT 3		; Interrupt program

The given data array is 8341, 7258, 4639, 8453, 9630. These are hexadecimal numbers.

As there are five 16-bit numbers in the given data array, the count is 0005. Two consecutive memory locations store a 16-bit number *i.e.* two bytes of a 16-bit number. The data fed in the memory for the above program are as given below:

Data	Result
0200 - 05 H	0251 - 39 H
0201 - 00 H	0252 - 46 H
0202 - 41 H	
0203 - 83 H	

0204 - 58 H
0205 - 72 H
0206 - 39 H
0207 - 46 H
0208 - 53 H
0209 - 84 H
020A - 30 H
020B - 96 H

A 16-Bit number is stored in two consecutive memory locations. The result is 4639H being the smallest number in the given data array. It is stored in the memory locations 0251H and 0252H. At 010D effective memory location, 72H is the opcode of the JB instruction. 02 is the displacement at the effective memory location 010E. The program execution jumps from the effective memory location 010E to 0111H, if the array flag CF=1. Therefore, the displacement is 02. There is LOOP instruction at the effective memory location 0111H. The program execution jumps from the effective memory location 0112H back to 0109H. In this case the displacement is ten memory locations back *i.e.* the displacement is – 0A. The 2's complement of 0A is F6 which has been shown at the effective memory location 0112H.

11.6.3 Largest 8-bit Number in an 8-Bit Data Array

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	B0,00		MOV	AL, 00	; Initial value for comparison
0103	BE,00,02		MOV	SI, 0200	; Memory address in SI
0106	8B,0C		MOV	CX, [SI]	; Count in CX
0108	46		INC	SI	; Increment SI
0109	46	BACK	INC	SI	; Increment SI
010A	3A,04		CMP	AL, [SI]	; Compare previous largest number with next number
010C	73,02		JAE	GO	; Jump if number in AL is larger i.e. CF=0
010E	8A,04		MOV	AL, [SI]	; Save new larger
0110	E2,F7	GO	LOOP	BACK	; Jump to BACK until CX becomes zero
0112	A2,51,02		MOV	[0251], AL	; Store largest number in memory
0115	CC		INT3		; Interrupt program

At the memory location 0111H, there is a backward displacement of 9 i.e. – 9. Its 2's complement is F7. At 010D memory location 02 is forward displacement.

The given data array is : 54, 83, 59, A2, 86, B9. These numbers are hexadecimal numbers.

Data	Result
0200 - 06 Count	0251 - B9 H
0201 - 00	
0202 - 54 H	
0203 - 83 H	
0204 - 59 H	
0205 - A2 H	
0206 - 86 H	
0207 - B9 H	

11.6.4 Smallest 8-Bit Number in an 8-Bit Data Array

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	B0,FF		MOV	AL , FF	; Ini. value for comparison
0103	BE,00,02		MOV	SI , 0200	; Memory address in SI
0106	8B,0C		MOV	CX, [SI]	; Count in CX
0108	46		INC	SI	; Increment SI
0109	46	BACK	INC	SI	; Increment SI
010A	3A,04		CMP	AL, [SI]	; Compare previous smallest number with next number
010C	72,02		JB	GO	; Jump if number in AL is smaller i.e. CF=1
010E	8A,04		MOV	AL, [SI]	; Save new smaller
0110	E2,F7	GO	LOOP	BACK	; Jump to BACK until CX becomes zero
0112	A2,51,02		MOV	[0251] , AL	; Store smallest number in memory
0115	CC		INT 3		; Interrupt program

At effective memory address 0111H there is a backward displacement of 9 *i.e.* displacement is –9. Its 2's complement is F7. At 010D effective memory address 02 is forward displacement.

The given data array is : 54, 83, 59, A2, 86, B9. These numbers are hexadecimal numbers.

Data	**Result**
0200 - 06 Count	0251 - 54 H
0201 - 00 H	
0202 - 54 H	
0203 - 83 H	
0204 - 59 H	
0205 - A2 H	
0206 - 86 H	
0207 - B9 H	

11.6.5 Sum of a Series of 16-Bit Numbers ; Sum : 16-Bit

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	B8,00,00		MOV	AX, 0000	; initial sum 0000
0104	BE,01,02		MOV	SI, 0201	; memory address in SI
0107	8B,0C		MOV	CX, [SI]	; count in CX
0109	46	BACK	INC	SI	; increment SI
010A	46		INC	SI	; increment SI
010B	03,04		ADD	AX , [SI]	; add next number to previous sum
010D	E2,FA		LOOP	BACK	; jump to BACK until CX, becomes zero
010F	A3,01,03		MOV	[0301], AX	; store sum in memory
0112	CC		INT 3		; interrupt program

Example: Add the following Series of numbers : 5893, 3B61 and 439A.

Data	**Result**
0201 - 03 count	0301 - 8E H
0202 - 00	0302 - D7 H

0203 - 93 H
0204 - 58H
0205 - 61 H
0206 - 3B H
0207 - 9A H
0208 - 43 H

At the memory location 010E, there is backward displacement of 6 *i.e.* –6. Its 2's complement is FA.

11.6.6 Sum of a Series of 16-Bit Numbers; Sum : 32-Bit

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	B8,00,00		MOV	AX , 0000	; initial sum 0000
0104	BB,00,00		MOV	BX , 0000	; MSBs of sum 0000
0107	BE,01,02		MOV	SI , 0201 H	; memory address in SI
010A	8B,0C		MOV	CX, [SI]	; count in CX
010C	46	BACK	INC	SI	; increment SI
010D	46		INC	SI	; increment SI
010E	03,04		ADD	AX, [SI]	; add next number to previous sum
0110	73,01		JAE	GO	; jump to GO, if CF is zero
0112	43		INC	BX	; add carry to MSBs of sum
0113	E2,F7	GO	LOOP	BACK	; jump to BACK until CX = 00
0115	A3,01,04		MOV	[0401], AX	; store LSBs of sum in memory
0118	89,1E,03,04		MOV	[0403], BX	; store MSBs of sum in memory
011C	CC		INT 3		; interrupt program

Example: Add the following series of numbers : 94B5, 69A8 and E142.

Data	**Result**
0201 - 03 count	0401 - 60 H
0202 - 00 H	0402 - 1E H
0203 - B5 H	0403 - 02 H
0204 - 94 H	0404 - 00 H
0205 - 69 H	
0206 - A8 H	
0207 - 42 H	
0208 - E1 H	

At the memory location 0114H, there is a backward displacement of 9, *i.e.* – 9. Its 2's complement is F7. At the memory location 0111H, there is a forward displacement of 1.

11.6.7 Block Move or Relocate (Byte Move)

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	FC			CLD	; clear direction flag DF
0102	BE,00,02		MOV	SI, 0200	; source address in SI
0105	BF,02,03		MOV	DI, 0302	; destination address in DI
0108	8B,0C		MOV	CX, [SI]	; count in CX
010A	46		INC	SI	; increment SI
010B	46		INC	SI	; increment SI
010C	A4	BACK	MOVSB		; move byte
010D	E2,FD		LOOP	BACK	; jump to BACK until CX becomes zero
010F	CC		INT		; interrupt program

Data	Result
0200 - 05 count	0302 - 94H
0201 - 00 H	0303 - 86H
0202 - 94 H	0304 - 25H
0203 - 86 H	0305 - 78H
0204 - 25 H	0306 - 69H
0205 - 78 H	
0206 - 69 H	

The instruction MOVSB moves byte from the memory location addressed by SI to the memory location addressed by DI. After the execution of MOVSB instruction SI and DI are automatically incremented by one, if the direction flag DF is zero. The DF flag has already been cleared in the beginning of the program. At 010E effective memory address FD is the 2's complement of backward displacement which is –3.

11.6.8 Block Move or Relocate (Word Move)

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	FC		CLD		; clear direction flag DF
0102	BE,00,02		MOV	SI,0200 H	; source address in SI
0105	BF,02,03		MOV	DI, 0302 H	; destination addressin DI
0108	8B,0C		MOV	CX, [SI]	; count in CX
010A	46		INC	SI	; increment SI
010B	46		INC	SI	; increment SI
010C	A5	BACK	MOVSW		; move word
010D	E2, FD		LOOP	BACK	; jump to BACK until CX becomes zero
010F	CC		INT 3		; interrupt program

Data	Result
0200 - 03 count	0302 - 54 H
0201 - 00 H	0303 - 86 H
0202 - 54 H	0304 - 94 H
0203 - 86 H	0305 - 45 H
0204 - 94 H	0306 - 38 H
0205 - 45 H	0307 - 63 H
0206 - 38 H	
0207 - 63 H	

The instruction MOVSW move word (two bytes at a time) from the memory locations addressed by SI to the memory locations addressed by DI. After the execution of MOVSW instruction SI and DI are incremented by two, if the direction flag DF is zero. For example, if SI contains 0202 H, the MOVSW instruction moves the contents of 0202 H and 0203 H to the memory locations 0302 H and 0303 H, if DI contains 0302 H. The DF flag has already been cleared in the beginning of the program. At the effective memory address 010E, there is a backward displacement of –3, whose 2's complement is FD.

11.6.9 Block Move (Byte Move) Using REP Instruction

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	BE,00,02		MOV	SI , 0200 H	; source address in SI
0104	BF,02,03		MOV	DI, 0302 H	; destination address in DI
0107	8B,0C		MOV	CX, [SI]	; count in CX
0109	46		INC	SI	; increment SI

010A	46		INC	SI	; increment SI
010B	F2		REP		; repeat next string instruction until CX becomes zero
010C	A4		MOVSB		; move bytes
010D	CC		INT 3		; interrupt program

Data

0200 - 05 count
0201 - 00 H
0202 - 94 H
0203 - 86 H
0204 - 25 H
0205 - 78 H
0206 - 69 H

Result

0302 - 94 H
0303 - 86 H
0304 - 25 H
0305 - 78 H
0306 - 69 H

The REP instruction is used as a prefix instruction before string instructions like MOVSB etc. It decrements CX register and repeats the string instruction until the content of CX register becomes zero. For example,

REP
MOVSB

11.6.10 Block Move (Word Move) Using REP Instruction

Effective Address	*Mnemonic Codes*	*Label*	*Mnemonics*	*Operands*	*Comments*
0101	BE,00,02		MOV	SI, 0200 H	; source address in SI
0104	BF,02,03		MOV	DI, 0302 H	; destination address in DI
0107	8B,0C		MOV	CX, [SI]	; count in CX
0109	46		INC	SI	; increment SI
010A	46		INC	SI	; increment SI
010B	F2		REP		; repeat next string instruction until CX becomes zero
010C	A5		MOVSW		; move words
010D	CC		INT 3		; interrupt program

Data

0200 - 03 count
0201 - 00
0202 - 54 H
0203 - 86 H
0204 - 94 H
0205 - 45 H
0206 - 38 H
0207 - 63 H

Result

0302 - 54 H
0303 - 86 H
0304 - 94 H
0305 - 45 H
0306 - 38 H
0307 - 63 H

The REP instruction is used as a prefix instruction before string instructions like MOVSW etc. It decrements CX register and repeats the string instruction until the contents of register CX becomes zero.

For example,

REP
MOVSW