ICT 2256: COMPUTER ORGANIZATION AND MICROPROCESSOR SYSTEMS [3 credit]

Santhosha Rao & Raviraj Holla

Syllabus:

Microprocessor 8086 Architecture, Pin diagram, Modes of operation, Segmentation and memory addressing, Addressing modes, Assembler directives, Assembly language development tools, Instruction set, Stacks and subroutine, Macros and procedures, Assembly language programming, Interrupts, BIOS and DOS interrupts, Basic IO interfacing- 8255 Programmable Peripheral Interface, 8254 Programmable Interval Timer, 8259 Programmable Interrupt Counter, Computer Organization: Introduction, Execution Unit - Combinational shifter design, Adders, Arithmetic and Logic Unit design, Multiplication algorithms, Division algorithms., Control Unit- Introduction, Basic concepts, Hardwired and Micro programming approach, Memory Unit, Input & Output.

References:

- Hall D.V., *Microprocessors and Interfacing: Programming and Hardware (3e)*, Tata McGraw Hill, 2017, **ISBN-10:** 9781259006159
- Brey B.B., *The Intel Microprocessors: 8086 to Pentium Pro - Architecture, Programming and Interfacing (8e)*, Prentice Hall of India, 2012
- Udaykumar K, Umashankar B.S., *Advanced microprocessors and IBM –PC assembly language programming*, McGraw Hill Education, 2017.
- Rafiquzzaman M and Rajan C., *Modern computer Architecture*, Galgotia Publications Pvt. Ltd, 2012.

History of Microprocessors

http://www.computerhistory.org/exhibits/microprocessors/index.page

Year	name	Data size	memory size	#instructions	
1971	4004	4	4096 4-bit	45	first microprocessor
1973	8008	8	16K bytes	48	1st 8-bit µP
1973	8080	8	64K bytes		10 times faster than 8008
1973	MC6800	8	64K bytes		1st Motorola µP
1977	8085	8	64K bytes	246	Intel's most successful 8-bit general-purpose µP due to its low cost
	Z80	8			Zilog's most successful microprocessor
1978	8086	8,16	1M bytes	>20,000	1st 16-bit µP
1979	8088	8,16	1M bytes		prefetch instruction using cache
1981	IBM decided to use 8088 in its personal computer				
1983	80286	8,16	16M		
1986	80386	8,16,32	4G		
1989	80486	8,16,32	4G		
1993	Pentium	8,16,32	4G		
1995	Pentium Pro	64	64G		
1997	Pentium II	64	64G		
1999	Pentium III	?	?		
2000	Pentium 4	?	?		

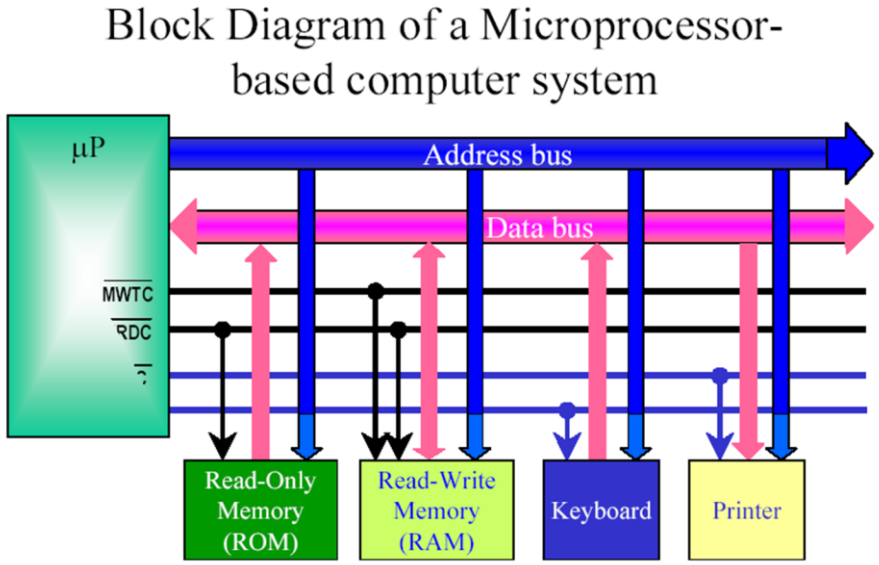
Basic elements of Microprocessor

Microprocessor – CPU. Executes instructions in microseconds

To perform an operation

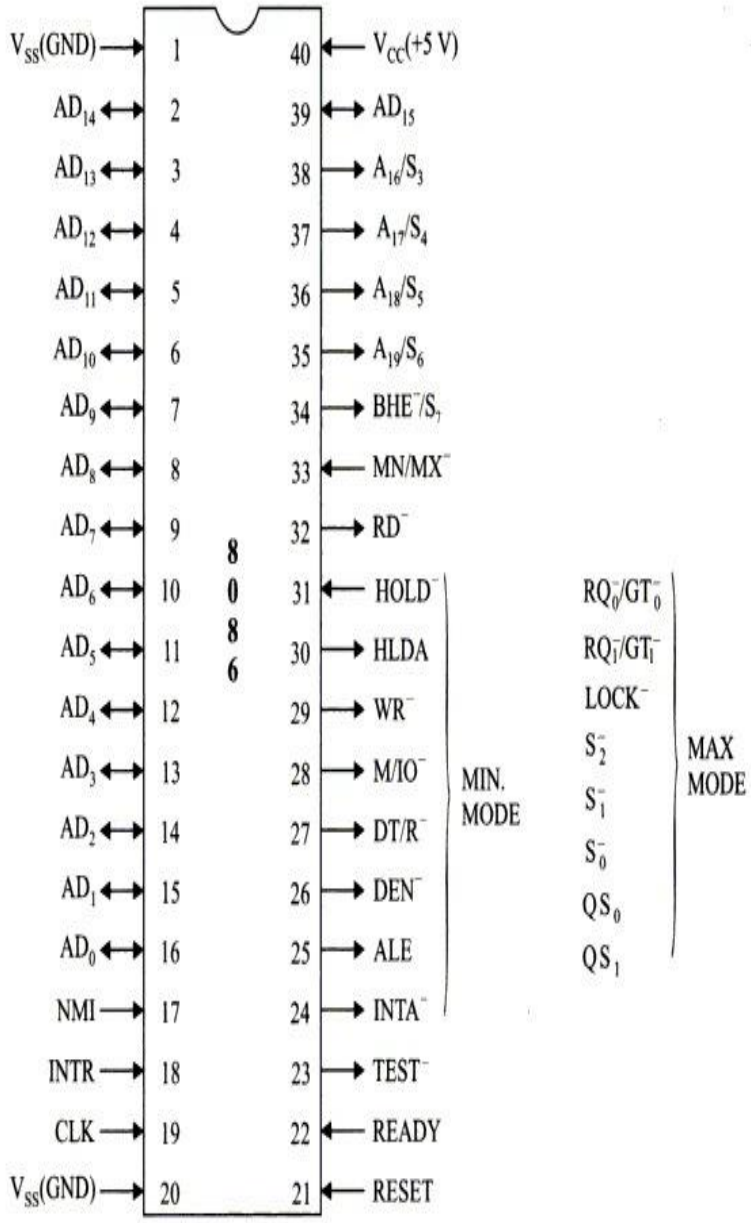
microprocessor requires:

- Registers
- Arithmetic and Logic Unit (ALU)
- Control Unit
- Instruction register
- Program counter
- Bus



Internal architecture of 8086

- It is a 16-bit µp.
- 8086 has a 20 bit address bus can access up to 2^{20} memory locations (1 MB).
- Word size is 16 bits and double word size is 4 bytes.
- It has multiplexed address and data bus AD0- AD15 and A16 – A19.
- It requires +5V power supply.
- A 40 pin dual in line package.
- Address ranges from 00000H to FFFFFH
- Memory is byte addressable - Every byte has a separate address.
- Works in Minimum & Maximum modes



Internal architecture of 8086

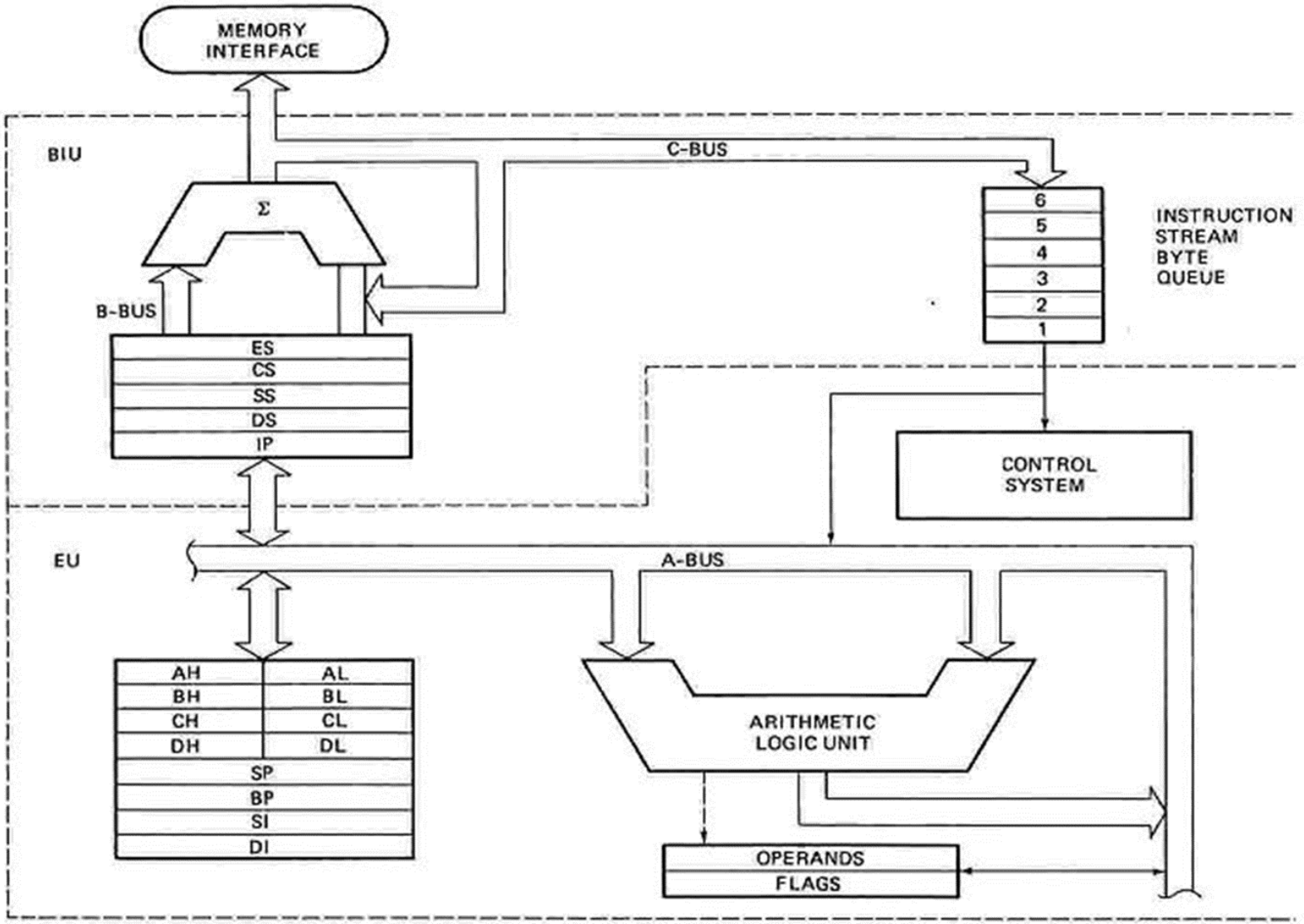
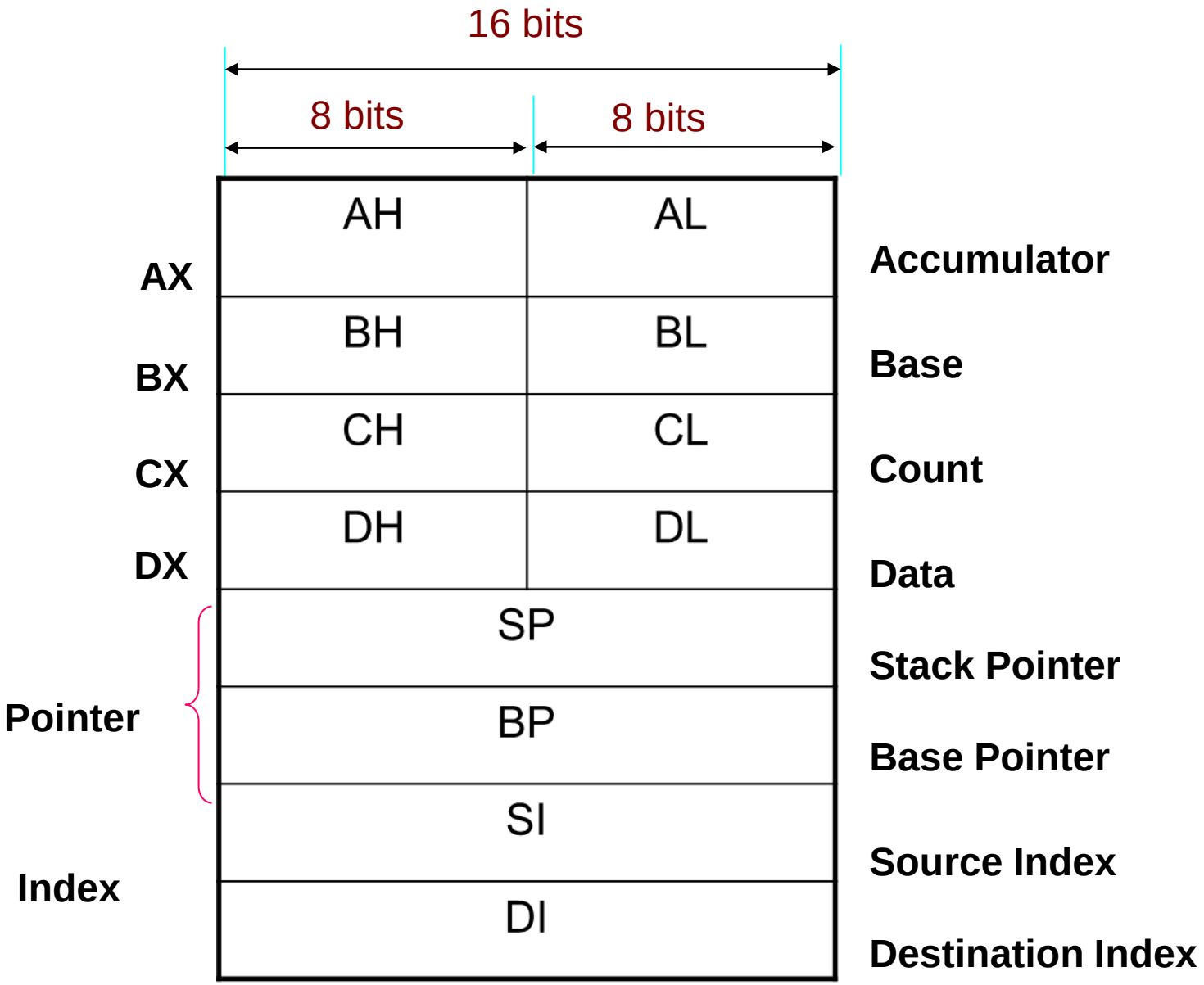


FIGURE 8086 internal block diagram. (*Intel Corp.*)

Internal architecture of 8086

- 8086 has two blocks Bus Interfacing Unit (BIU) and Execution Unit (EU).
- BIU contains Instruction queue, Segment registers, Instruction pointer, Address adder.
- EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.
- The BIU handles all transactions of data and addresses on the buses for EU.
- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.
- EU – decodes instructions fetched by BIU, generates control signals, executes instructions.
- Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining.

Internal architecture of 8086



General purpose, pointer and index registers

Internal architecture of 8086- General purpose Registers

- Normally used for storing temporary results
- Each of the registers is 16 bits wide **(AX, BX, CX, DX)**
- Can be accessed as either 16 or 8 bits AX, AH, AL
- **AX**
  - Accumulator Register
  - Preferred register to use in arithmetic, logic and data transfer instructions
  - Must be used in multiplication and division operations
  - Must also be used in I/O operations
- **BX**
  - Base Register
  - Also serves as an address register

Internal architecture of 8086- General purpose Registers

- **CX**
  - Used as a loop counter
  - String operations
  - Used in shift and rotate operations
- **DX**
  - Used in multiplication and division
  - Also used in I/O operations

Register	Purpose
AX	Word multiply, word divide, word I /O
AL	Byte multiply, byte divide, byte I/O, decimal arithmetic
AH	Byte multiply, byte divide
BX	Store address information
CX	String operation, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O (Used to hold I/O address during I/O instructions. If the result is more than 16-bits, the lower order 16-bits are stored in accumulator and higher order 16-bits are stored in DX register)

Internal architecture of 8086- Pointer and Index Registers

- used to keep offset addresses.
- Used in various forms of memory addressing.

SP -Stack pointer

– Used as pointer to top of the stack

BP - Base Pointer

– Primarily used to access the data from the stack

– Can also be used to access data in other segments

Internal architecture of 8086- Pointer and Index Registers

- The index registers SI (Source Index) & DI (Destination Index) and BX are used as default pointers to access the Data segment

- **SI: Source Index register**

  – is required for some string operations

  – When string operations are performed, the SI register points to memory locations in the data segment

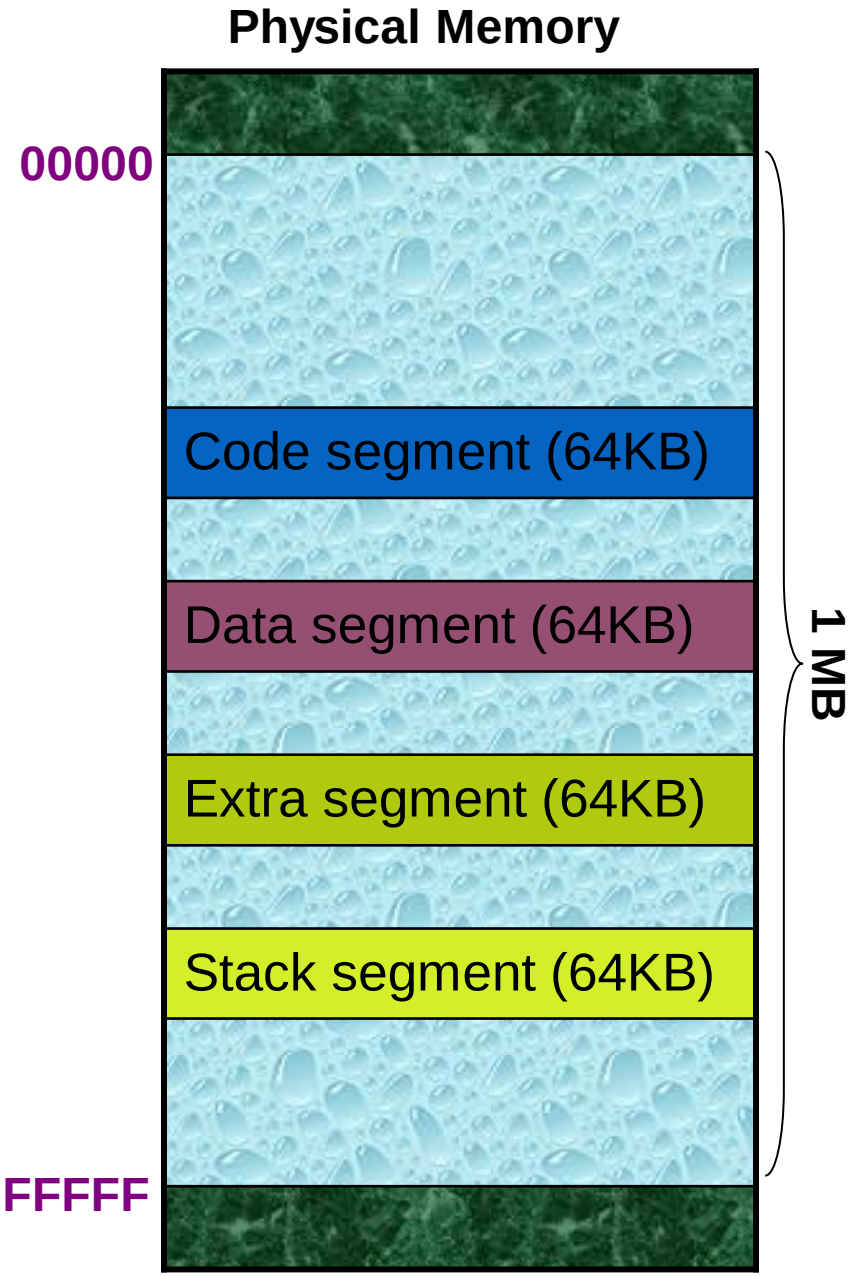
DI: Destination Index register

  – is also required for some string operations.

  – When string operations are performed, the DI register points to memory locations in the extra segment

8086 - Segmented memory

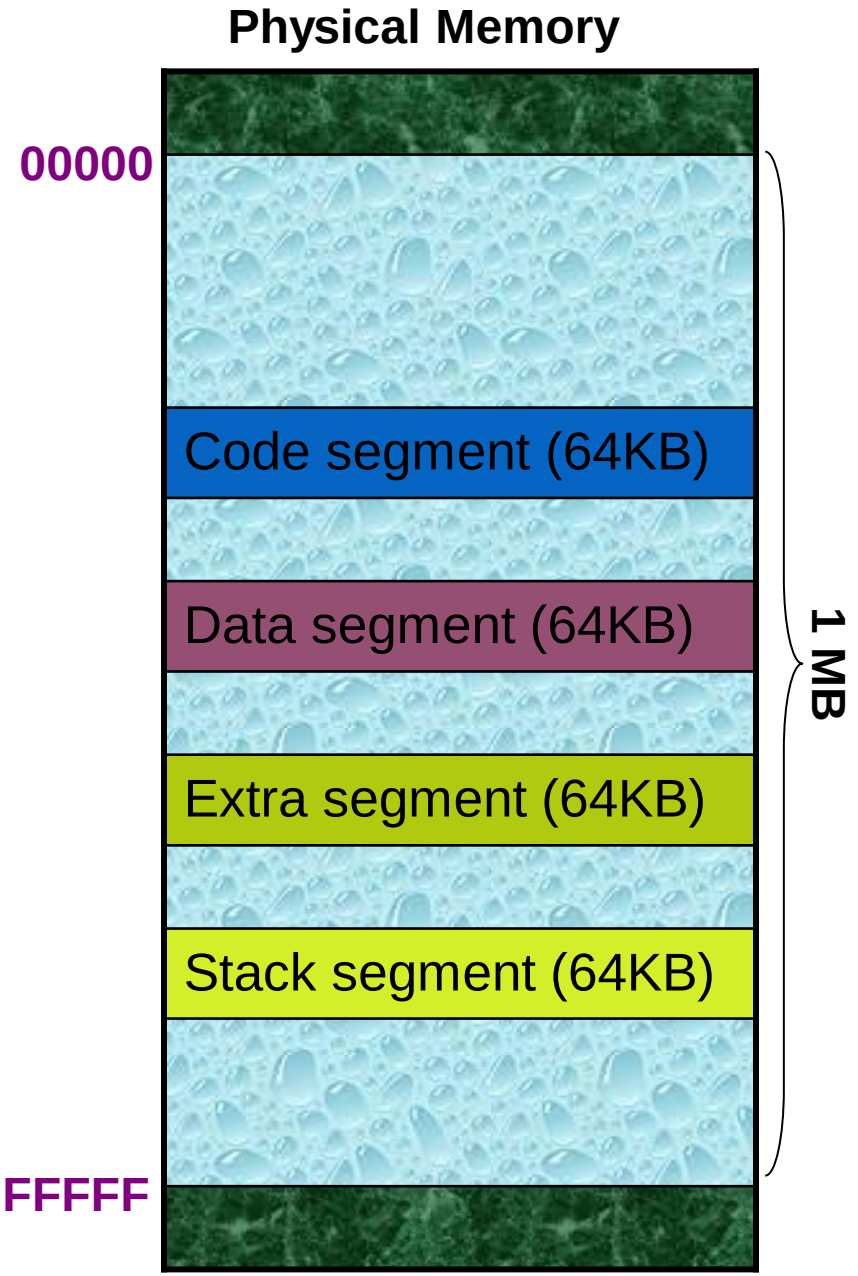
- The memory in an 8086 based system is organized as segmented memory.
- The CPU 8086 is able to address 1Mbyte of memory.
- The Complete physically available memory may be divided into a number of logical segments with max size of each segment – 64 KB



8086 - Segmented memory

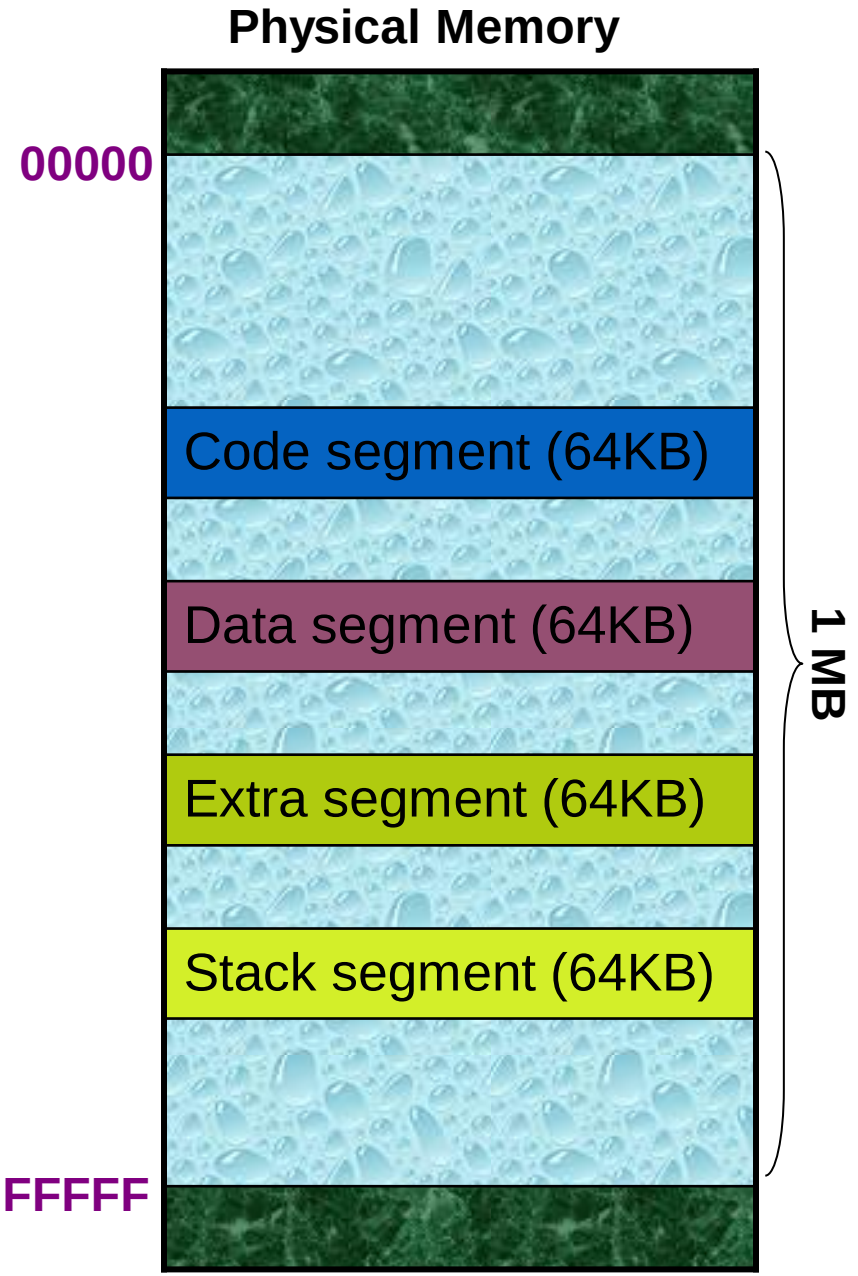
- A segment may be located any where in the memory
- Each of these segments can be used for a specific function.
- Code segment is used for storing the instructions.
- The stack segment is used as a stack and it is used to store the return addresses/data.
- The data and extra segments are used for storing data.

- **In the assembly language programming, more than one data/ code/ stack segments can be defined. But only one segment of each type can be accessed at any time.**
- **Segments may overlap.**

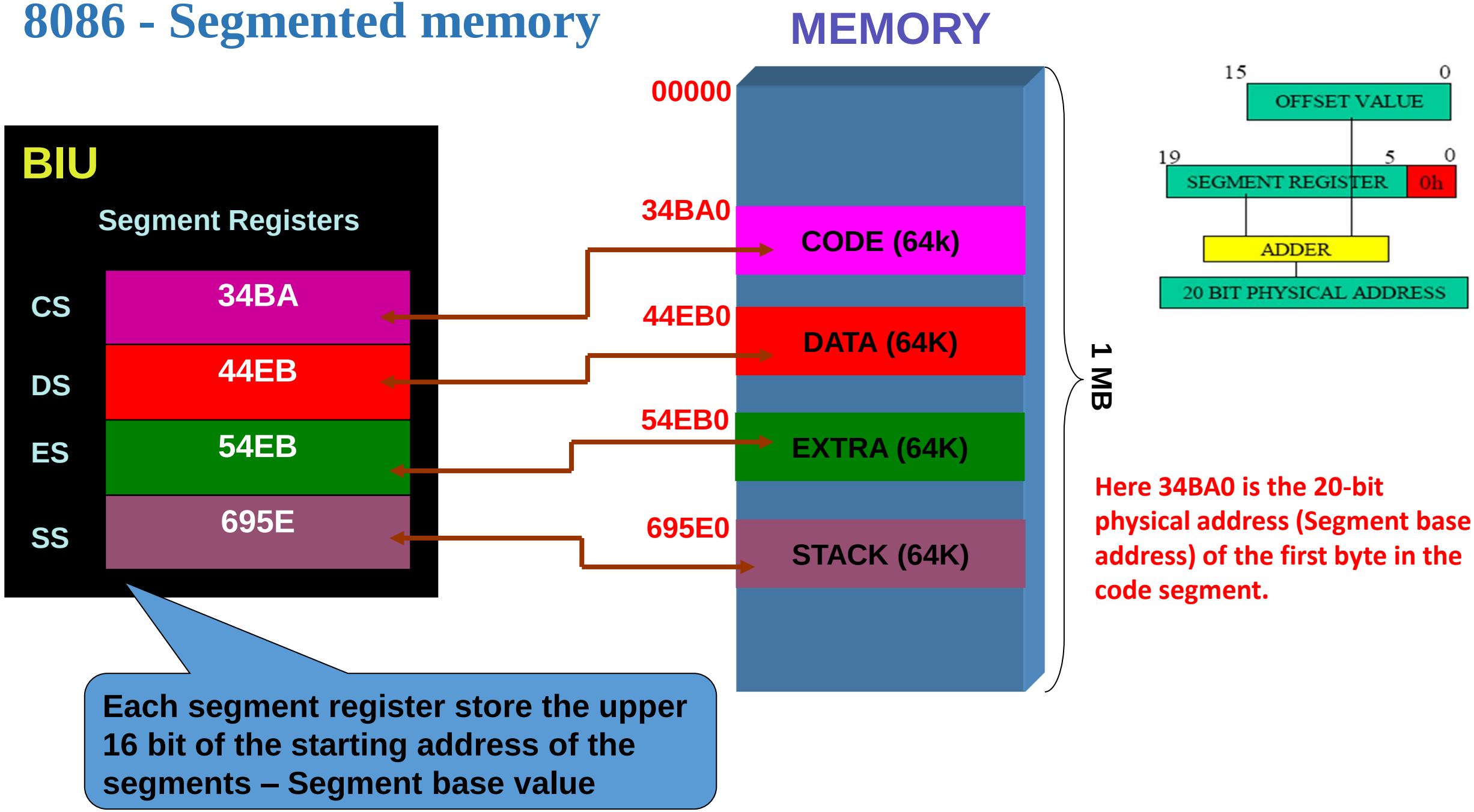


8086 - Segmented memory

- In 8086/88 the processors have 4 segments registers in BIU
  - **Code Segment register (CS)**
  - **Data Segment register (DS)**
  - **Extra Segment register (ES)**
  - **Stack Segment register(SS).**
- All are 16 bit registers.
- Each of the Segment registers store the upper 16 bit address of the starting address of the corresponding segments.

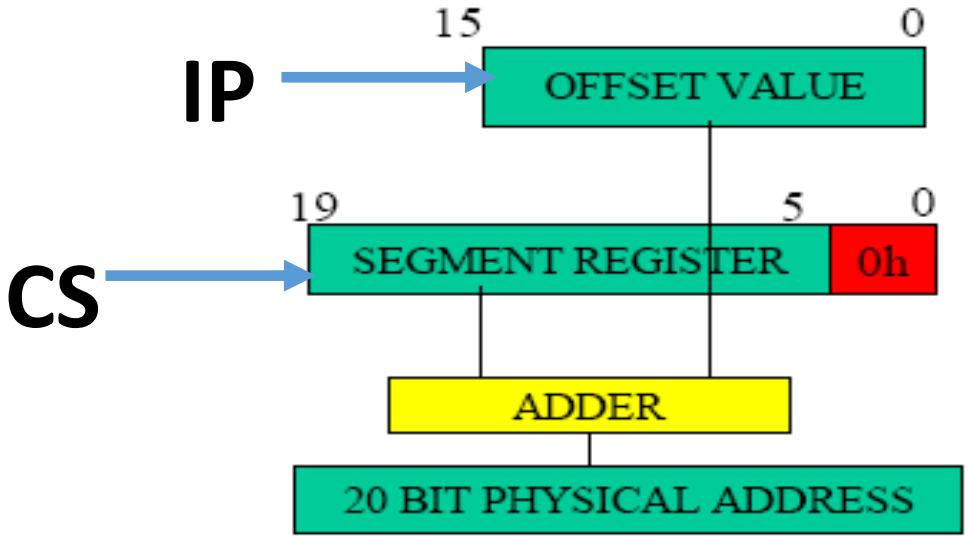


8086 - Segmented memory



8086 - Segmented memory

Generating 20-bit physical address for various bytes in code segment.



CS – holds segment base value for the code segment

IP – holds offset/displacement/Effective Address from the segment base address.

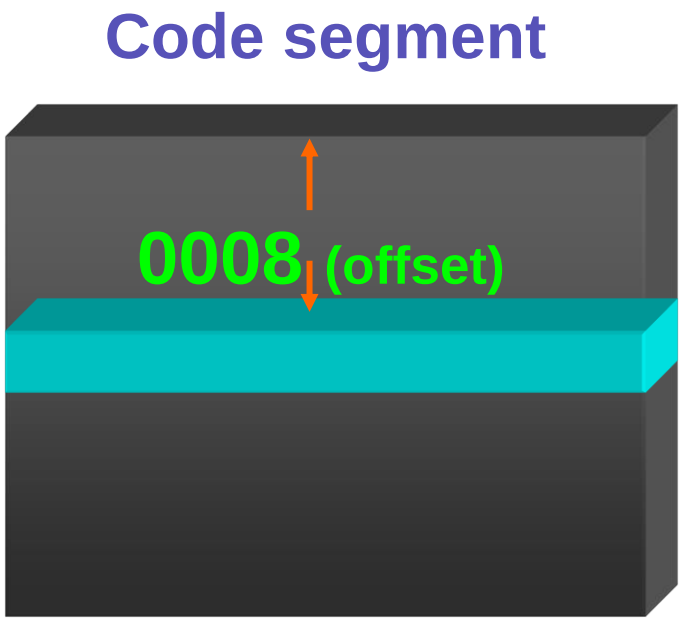
CS **1234** IP **0008**

1 2 3 4 0 (C S) +
0 0 0 8 (I P)

1 2 3 4 8 (physical address)

12340 H

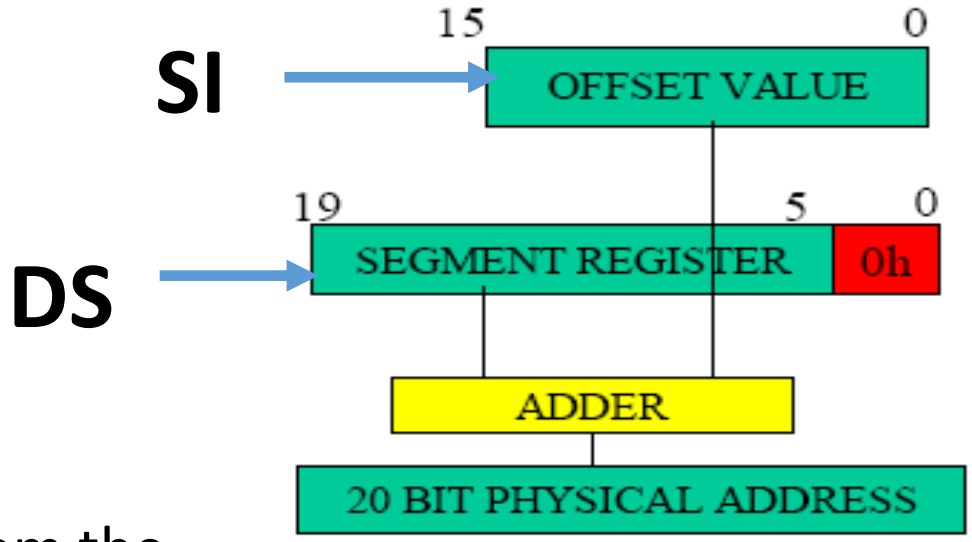
12348 H



Even physical address can be represented as CS:IP (1234:0008)

8086 - Segmented memory

Generating 20-bit physical address for various bytes in data segment.



DS – holds segment base value for the data segment

SI – holds offset/displacement/Effective Address from the segment base address. (Even DI, BX could be used)

23450 H

2 3 4 5 0 (DS) +
0 0 0 4 (SI)

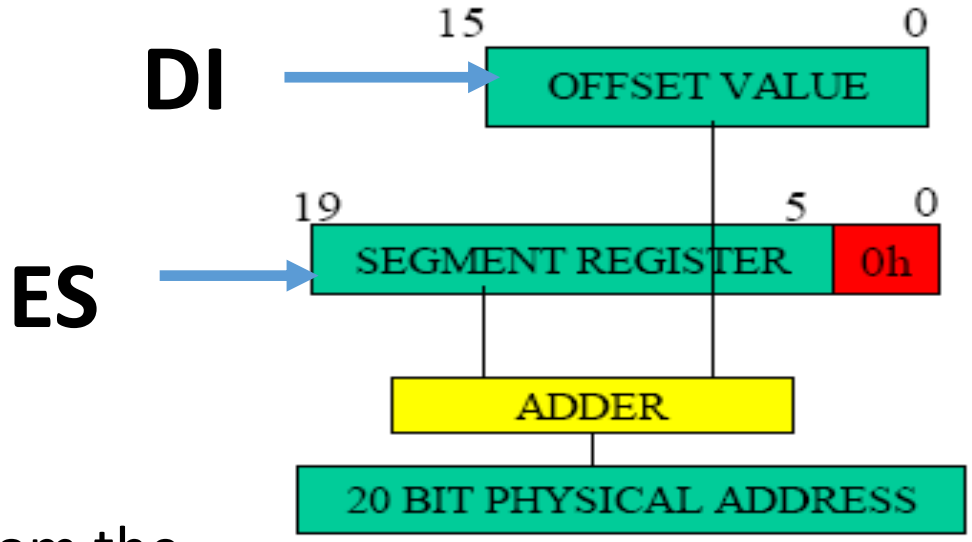
23454 H

2 3 4 5 4 (physical address)

Even physical address can be represented as DS:SI (2345:0004)

8086 - Segmented memory

Generating 20-bit physical address for various bytes in extra segment.



ES – holds segment base value for the extra segment

DI – holds offset/displacement/Effective Address from the segment base address.

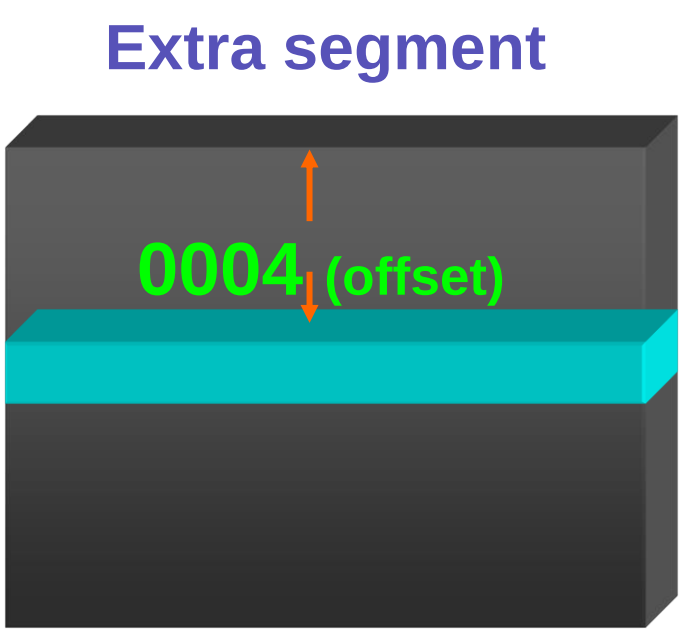
ES **2345** DI **0004**

23450 H

2 3 4 5 0 (ES) +
0 0 0 4 (SI)

23454 H

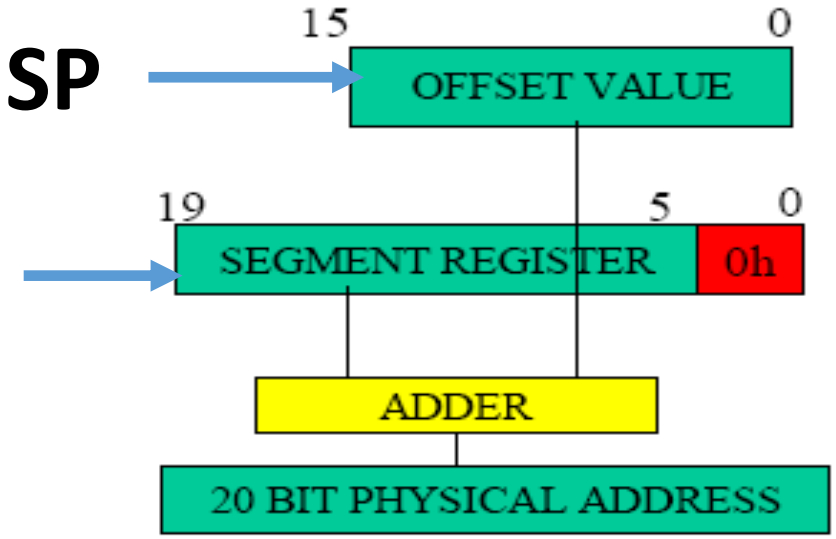
2 3 4 5 4 (physical address)



Even physical address can be represented as ES:DI (2345:0004)

8086 - Segmented memory

Generating 20-bit physical address for various bytes in stack segment.



SS – holds segment base value for the data segment

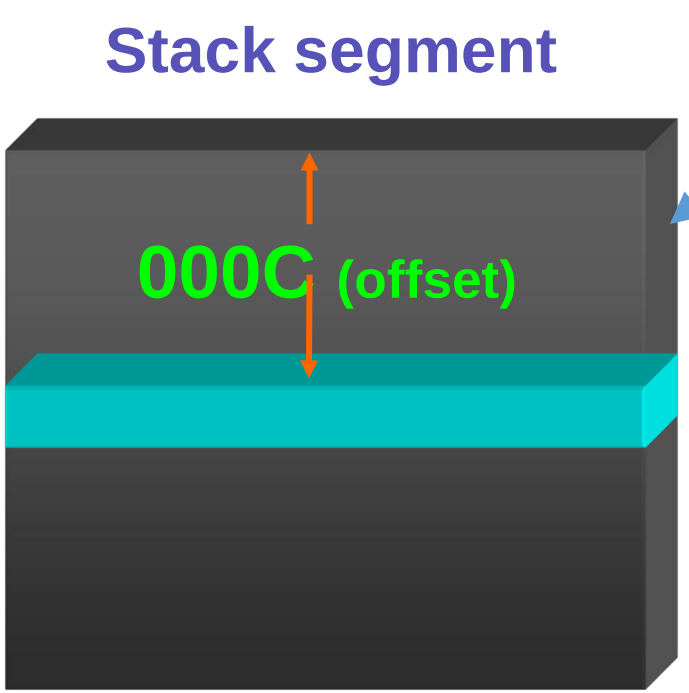
SP – holds offset/displacement/Effective Address of the TOS. **Intermediate elements can be accessed using BP.**

ES **2345** SP **000C**

2 3 4 5 0 (SS) +
0 0 0 C (SP)

2 3 4 5 C (physical address)

23450 H
23454 H
2345C H



0004 (offset of intermediate element)

BP – 0004H
SS:BP (2345:0004)-23454H

Even physical address of TOS can be represented as SS:SP (2345:000C)

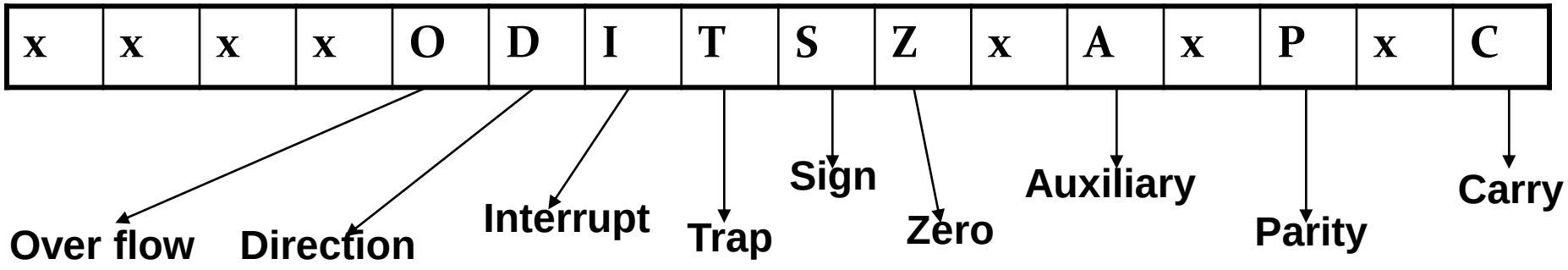
8086 - Segmented memory

Advantages of Segmentation

- Allows the memory capacity to be 1Mb although the actual addresses to be handled are of 16 bit size.
- Allows the placing of code, data and stack portions of the same program in different parts (segments) of the memory for data and code protection.
- Permits a program and/or its data to be put into different areas of memory each time program is executed, i.e. provision for relocation may be done .
- The segment registers are used to allow the instruction, data or stack portion of a program to be more than 64Kbytes long. The above can be achieved by using more than one code, data or stack segments.

8086 Internal architecture – Flag Register

- A flag is a flip flop which indicates some conditions produced by the execution of an instruction or controls certain operations of the EU .
- In 8086 The EU contains
  - a 16 bit flag register
  - 9 of the 16 are active flags and remaining 7 are undefined.
  - 6 flags indicate some conditions- status flags
  - 3 flags –control Flags



x - Unused

8086 Internal architecture – Flag Register

Flag	Purpose
Carry (C)	Holds the carry after addition or the borrow after subtraction. Also indicates some error conditions, as dictated by some programs and procedures .
Parity (P)	P=0;odd parity, P=1;even parity.
Auxiliary (A)	Holds the carry (half – carry) after addition or borrow after subtraction between bit positions 3 and 4 of the result (for example, in BCD addition or subtraction.)
Zero (Z)	Shows the result of the arithmetic or logic operation. Z=1; result is zero. Z=0; The result is 0
Sign (S)	Holds the sign of the result after an arithmetic/logic instruction execution. S=1; negative, S=0

8086 Instruction set (contd…)

Mnemonic	Meaning	Format	Operation	Flags affected
LEA	Load Effective Address	LEA Reg16,SRC	SRC → (Reg16)	None
LDS	Load Register And DS	LDS Reg16,SRC	(SRC) → (Reg16) (SRC+2) → (DS)	None
LES	Load Register and ES	LES Reg16,SRC	(SRC) → (Reg16) (SRC+2) → (DS)	None

8086 Instruction set (contd…)

LEA SI, SRC same as **MOV SI, offset SRC**

LDS SI, SRC

SI ← 0001H
DS ← 4569H

LES DI, SRC

DI ← 0001H
ES ← 4569H

Data segment

SRC	01
SRC+1	00
SRC+2	69
SRC+3	45

Packed BCD arithmetic

DAA Decimal Adjust after Addition

1. If the value of the low-order four bits (D_3-D_0) in the AL is greater than 9 or if AF is set, the instruction adds 6 (06) to the low-order four bits.
2. If the value of the high-order four bits (D_7-D_4) in the AL is greater than 9 or if carry flag is set, the instruction adds 6 (60) to the high-order four bits.

Examples :

```
1.                  ; AL = 0011 1001 = 39 BCD
                    ; CL = 0001 0010 = 12 BCD
Add AL, CL          ; AL = 0100 1011 = 4BH
DAA                 ; Add 0110 Because 1011 > 9
                    ; AL = 0101 0001 = 51 BCD
2.                  ; AL = 1001 0110 = 96 BCD
                    ; BL = 0000 0111 = 07 BCD
ADD AL, BL          ; AL = 1001 1101 = 9DH
DAA                 ; Add 0110 Because 1101 > 9
                    ; AL = 1010 0011 = A3H
                    ; 1010 > 9 so add 0110 0000
                    ; AL = 0000 0011 = 03 BCD, CF = 1. The result is 103.
```

The instruction updates the AF, CF, PF, and ZF. The OF is undefined after DAA instruction.

Note : only works for AL.

DAS Decimal Adjust after Subtraction

This instruction is used after subtracting two packed BCD numbers to make sure the result is correct packed BCD. Instruction works as follows :

1. If the value of the low-order four bits (D_3-D_0) in the AL is greater than 9 or if AF is set; the instruction subtracts 6 (06) from the low-order four bits.
2. If the value of the high-order four bits (D_7-D_4) in the AL is greater than 9 or if carry flag is set, the instruction subtracts 6 (60) from the high-order four bits.

```
1.                      ; AL = 0011 0010 = 32 BCD
                        ; CL = 0001 0111 = 17 BCD
SUB AL, CL              ; AL = 0001 1011 = 1BH
                        ; Subtract 0110 Because 1011 > 9
                        ; AL = 0001 0101 = 15 BCD
2.                      ; AL = 0010 0011 = 23 BCD
                        ; CL = 0101 1000 = 58 BCD
SUB AL, CL              ; AL = 1100 1011 = CBH CF = 1
                        ; Subtract 0110 (6) Because 1011 > 9
                        ; AL = 1100 0101 = C5H
                        ; Subtract 0110 0000 Because 1100 > 9
                        ; AL = 0110 0101 = 65 BCD  CF = 1,
                        ; CF = 1 means  borrow
                        ; is needed means number is negative (– 65).
```

The DAS instruction updates the AF, CF, PF, and ZF. The OF flag is undefined after DAS instruction.

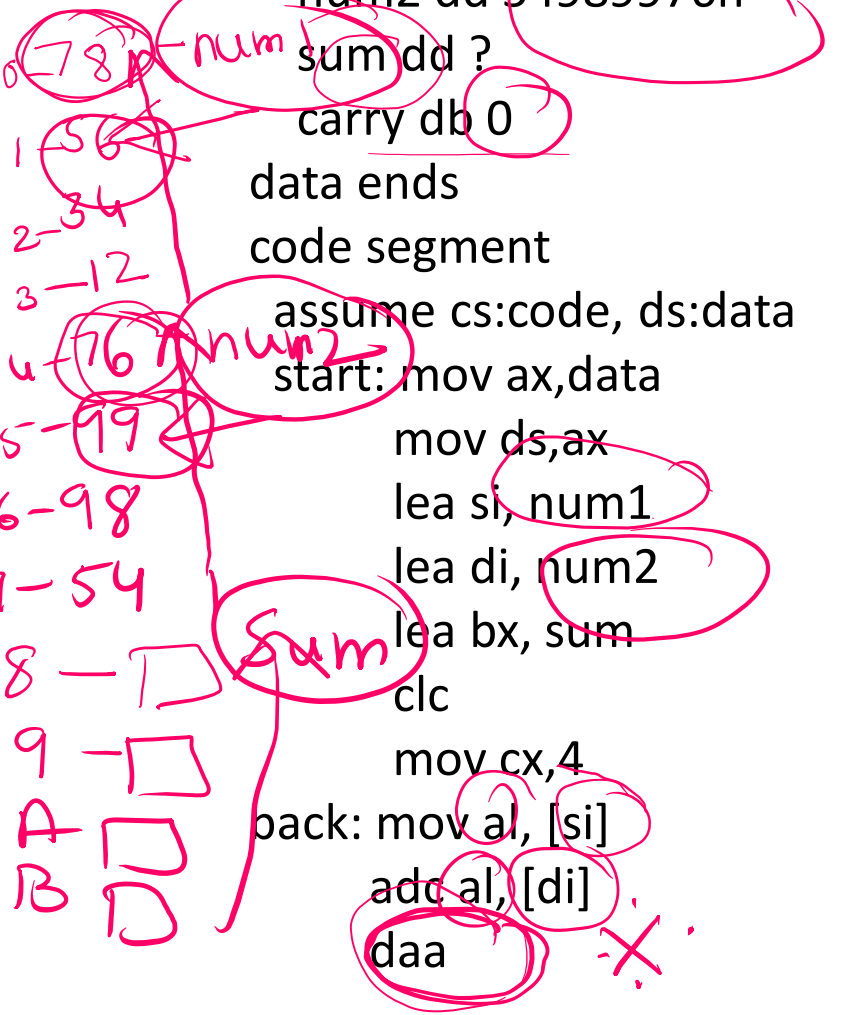
Note : DAS only works for AL

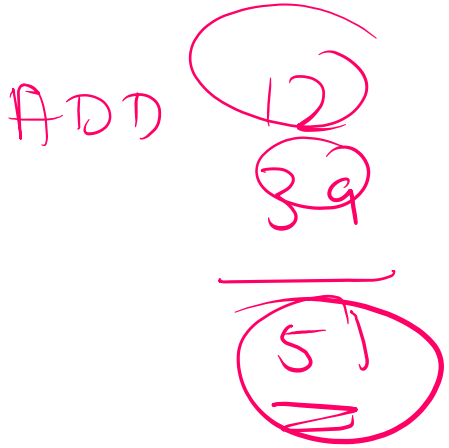
Add 2- 32 bit numbers

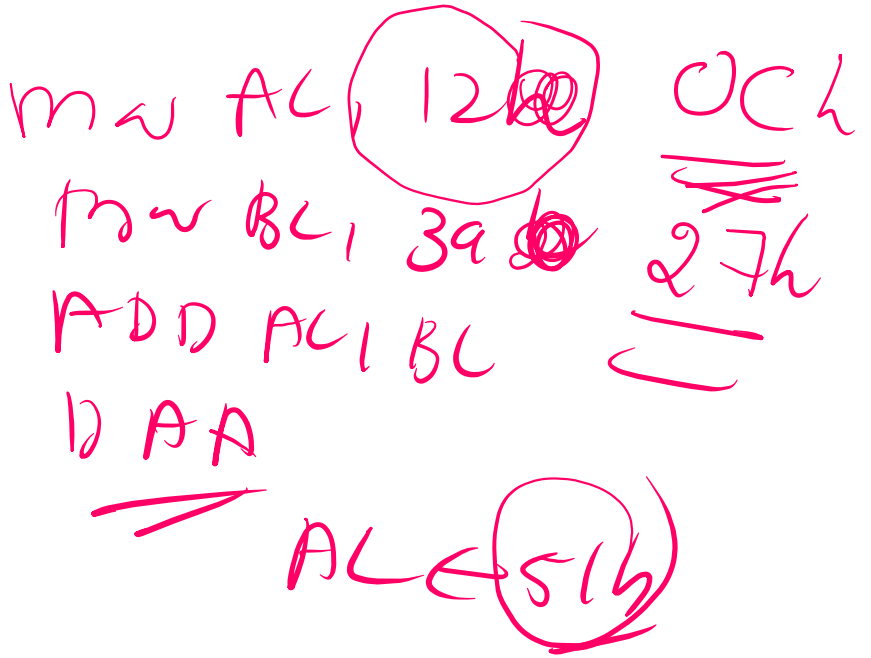
```
data segment
    num1 dd 12345A7Ch
    num2 dd 93455B91h
    sum dd ?
    carry db 0
data ends
code segment
  assume cs:code, ds:data
  start: mov ax,data
          mov ds,ax
          lea si, num1
          lea di, num2
          lea bx, sum
          clc
          mov cx,4
back: mov al, [si]
        adc al, [di]
        mov [bx], al
        inc si
        inc di
        inc bx
        loop back
      adc carry,0
      mov ah,4ch
      int 21h
code ends
end start
```

Add 2- 8 digit BCD numbers

```
data segment
   num1 dd 12345678h
   num2 dd 54989976h
   sum dd ?
   carry db 0
data ends
code segment
 assume cs:code, ds:data
 start: mov ax,data
        mov ds,ax
        lea si, num1
        lea di, num2
        lea bx, sum
        clc
        mov cx,4
back: mov al, [si]
      adc al, [di]
      daa
      mov [bx], al
      inc si
      inc di
      inc bx
      loop back
      adc carry,0
      mov ah,4ch
      int 21h
code ends
end start
```



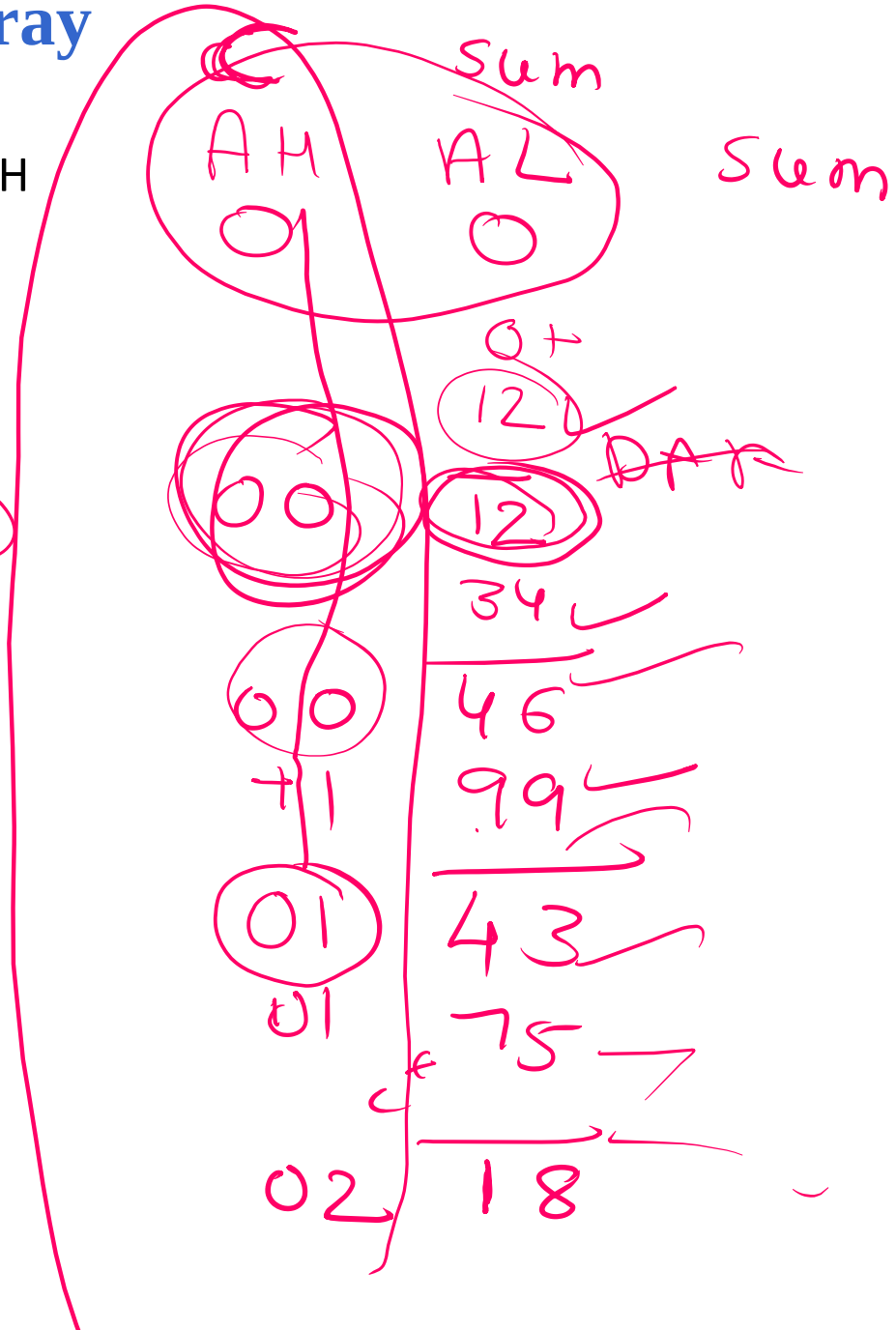




Sum of 10- 2 digit BCD numbers of an array

```
DATA SEGMENT
ARRAY1 DB 12H, 34H, 99H, 75H, 12H, 76H, 68H, 91H, 12H, 96H
COUNT DW $-ARRAY1
SUM DW ?
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:DATA
START: MOV AX,DATA
       MOV DS,AX
       LEA SI, ARRAY1
       MOV AX,0
       CLC
       MOV CX,COUNT
BACK: ADC AL,[SI]
       DAA
       XCHG AH,AL
       ADC AL,0
       DAA
       XCHG AH,AL
       INC SI
       LOOP BACK

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



Control flow and JUMP instructions

JMP <label> ; unconditionally branch to label Intra segment/Inter segment

Conditional Jump instructions

J<*condition_code*> <label> ; conditionally branch to a label

(range of –128 to +127 bytes).

Jump based on a single flag

JZ/JE ;Jump if zero flag set to 1 (Jump if result is zero)
JNZ/JNE ;Jump if Not Zero (Z flag = 0 i.e. result is nonzero)
JS ;Jump if Sign flag set to 1 (result is negative)
JNS ;Jump if Not Sign (result is positive)
JC ;Jump if Carry flag set to 1
JNC ;Jump if No Carry
JP ;Jump if Parity flag set to 1 (Parity is even)
JNP ;Jump if No Parity (Parity is odd)
JO ;Jump if Overflow flag set to 1
JNO ;Jump if No Overflow

Conditional JUMPS based on multiple flags

Unsigned

JB – Jump if Below (**JNA** – Jump if Not Above) (CF==1) && (ZF==0)
JBE – Jump if Below or Equal (**JNA** – Jump if Not Above) (CF==1) || (ZF==1)
JNB – Jump if Below (**JAE** – Jump if Above or Equal) (CF==0) || (ZF==1)
JNBE – Jump if NOT Below or Equal(**JA** – Jump if Above) (CF==0) && (ZF==0)

Signed

JL – Jump if Less (**JNGE** – Jump if Not Greater or Equal) (SF != OF) && (ZF==0)
JLE – Jump if Less or Equal (**JNG** – Jump if Not Greater) (SF != OF) || (ZF==1)
JNL – Jump if Not Less (**JGE** – Jump if Greater or Equal) (SF == OF) || (ZF==1)
JNLE – Jump if Not Less or Equal (**JG** – Jump if Greater) (SF == OF) && (ZF==0)

Usually used after Compare instruction

CMP DST, SRC

(DST)-(SRC) - Result is no where stored, flags are affected similar to SUB instruction

Conditional JUMPS based on more than one flag

Unsigned

JB – Jump if Below (**JNAE** – Jump if Not Above or equal) (CF==1) && (ZF==0)
JBE – Jump if Below or Equal (**JNA** – Jump if Not Above) (CF==1) || (ZF==1)
JNB – Jump if Below (**JAE** – Jump if Above or Equal) (CF==0) || (ZF==1)
JNBE – Jump if NOT Below or Equal(**JA** – Jump if Above) (CF==0) && (ZF==0)

Signed

JL – Jump if Less (**JNGE** – Jump if Not Greater or Equal) (SF != OF) && (ZF==0)
JLE – Jump if Less or Equal (**JNG** – Jump if Not Greater) (SF != OF) || (ZF==1)
JNL – Jump if Not Less (**JGE** – Jump if Greater or Equal) (SF == OF) || (ZF==1)
JNLE – Jump if Not Less or Equal (**JG** – Jump if Greater) (SF == OF) && (ZF==0)

Logical Instructions

Mnemonic	Meaning	Format	Operation	Flags Affected
AND	Logical AND	AND DST,SRC	(DST) = (DST) & (SRC)	OF, SF, ZF, PF, CF
OR	Logical OR	OR DST,SRC	(DST) = (DST) | (SRC)	AF undefined
XOR	Logical Exclusive OR	XOR DST,SRC	(DST) = (DST) $\oplus$ (SRC)	
NOT	LOGICAL NOT	NOT DST	(DST) = 1's comp of (DST)	None

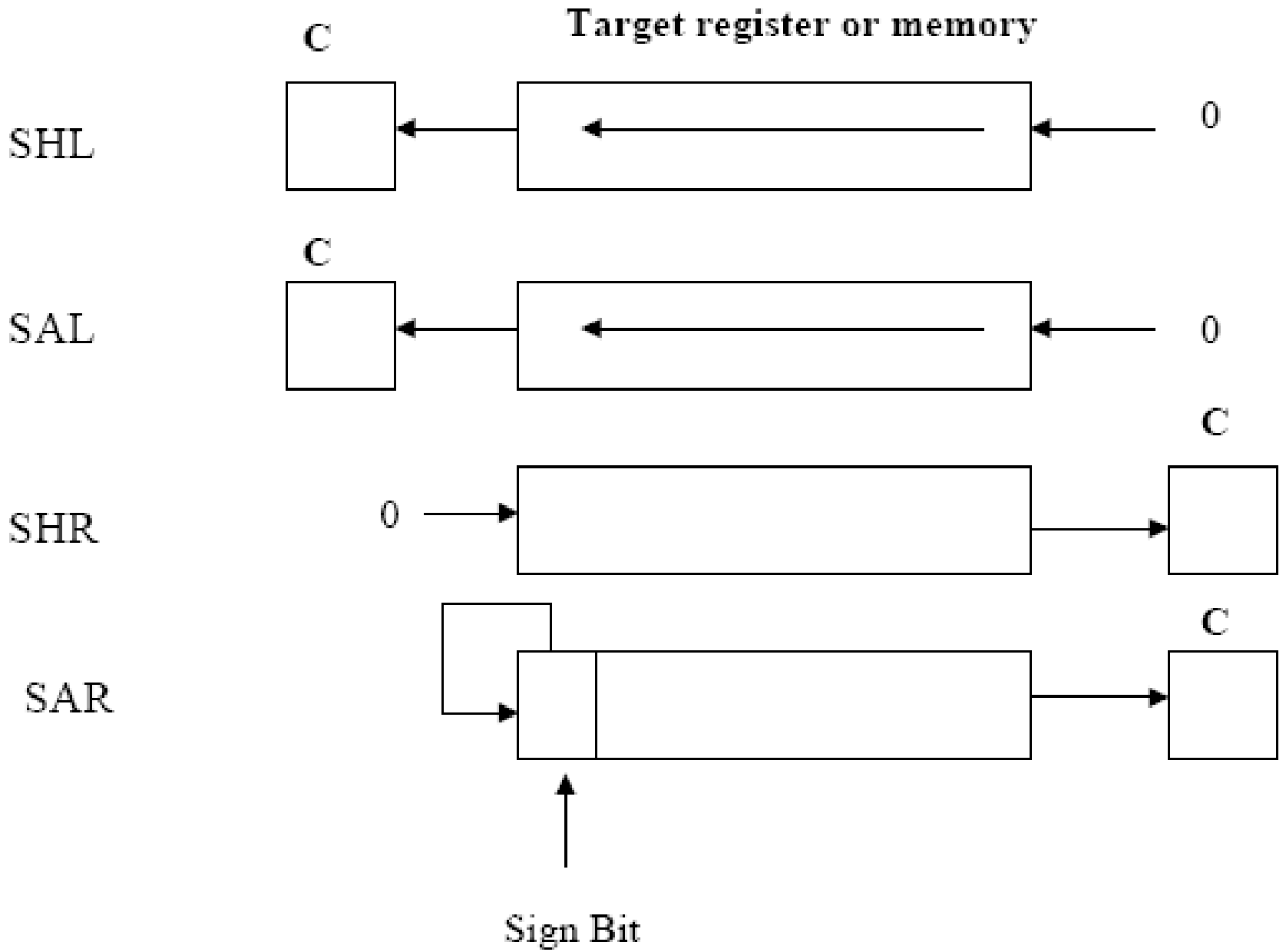
Destination	Source
Register Register Memory Register Memory	Register Memory Register Immediate Immediate

Destination
Register Memory

Shift Instructions

Mnemo -nic	Meaning	Format	Operation	Flags Affected
SAL/SHL	Shift arithmetic Left/shift Logical left	SAL/SHL DST, Count (Count is CL if not 1, else 1)	Shift the (DST) left by the number of bit positions equal to count and fill the vacated bits positions on the right with zeros	CF,PF,SF, ZF
SHR	Shift logical right	SHR DST, Count	Shift the (DST) right by the number of bit positions equal to count and fill the vacated bits positions on the left with zeros	
SAR	Shift arithmetic right	SAR DST, Count	Shift the (DST) right by the number of bit positions equal to count and fill the vacated bits positions on the left with the original most significant bit	

Shift Instructions

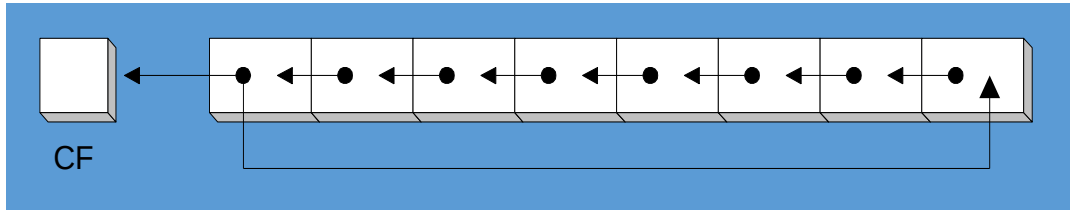


Rotate Instructions

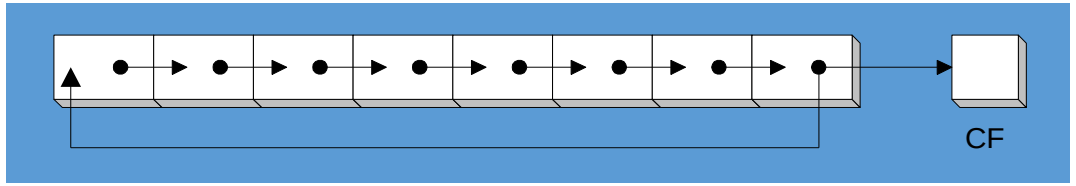
Mnem -onic	Meaning	Format	Operation	Flags Affected
ROL	Rotate Left	ROL DST,Count	Rotate the (DST) left by the number of bit positions equal to Count. Each bit shifted out from the left most bit goes back into the rightmost bit position.	CF, PF, SF. ZF
ROR	Rotate Right	ROR DST,Count	Rotate the (DST) right by the number of bit positions equal to Count. Each bit shifted out from the rightmost bit goes back into the leftmost bit position.	
RCL	Rotate Left through Carry	RCL DST,Count	Same as ROL except carry is attached to (DST) for rotation.	
RCR	Rotate right through Carry	RCR DST,Count	Same as ROR except carry is attached to (DST) for rotation.	

Rotate Instructions

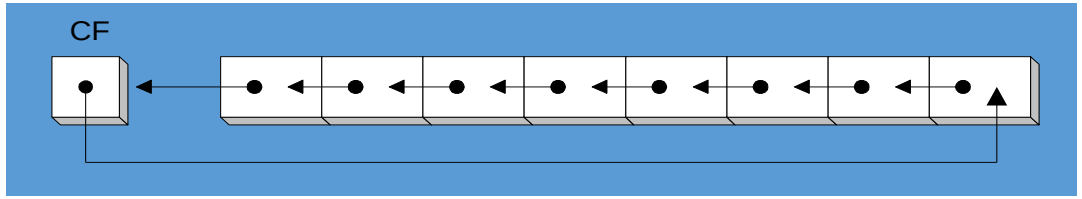
ROL



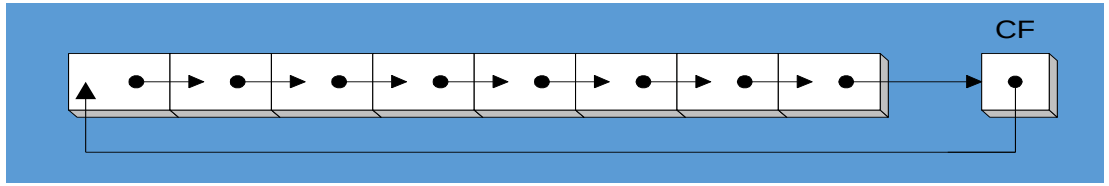
ROR



RCL



RCR

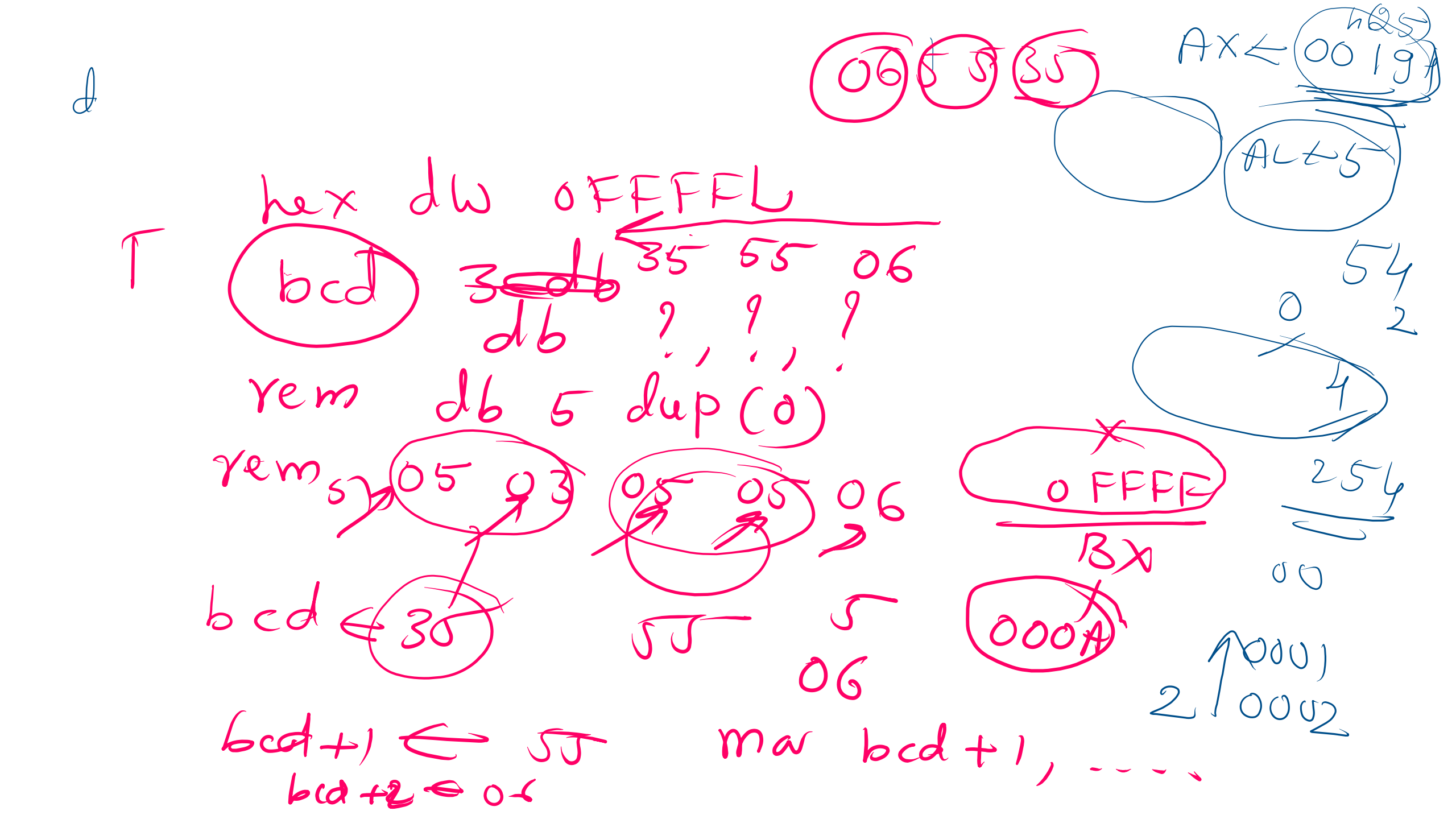


Flag manipulation Instructions

MNEM-ONIC	MEANING	OPERATION	Flags Affected
CLC	Clear Carry Flag	(CF) ← 0	CF
STC	Set Carry Flag	(CF) ← 1	CF
CMC	Complement Carry Flag	(CF) ← $(CF)^{I}$	CF
CLD	Clear Direction Flag	(DF) ← 0 SI & DI will be auto incremented while string instructions are executed.	DF
STD	Set Direction Flag	(DF) ← 1 SI & DI will be auto decremented while string instructions are executed.	DF
CLI	Clear Interrupt Flag	(IF) ← 0	IF
STI	Set Interrupt Flag	(IF) ← 1	IF

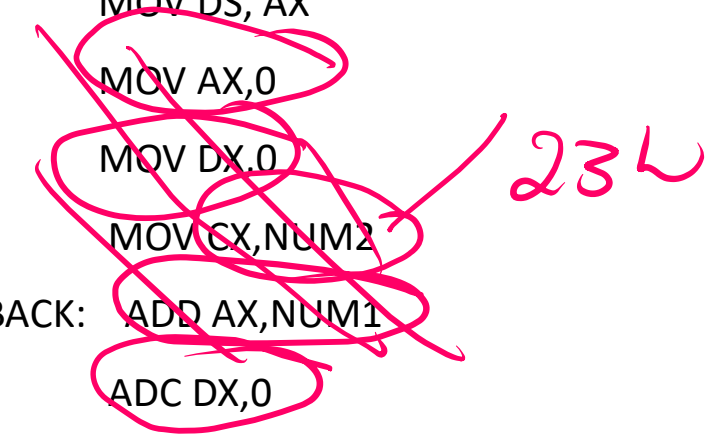
Program to convert 2-digit BCD to HEX

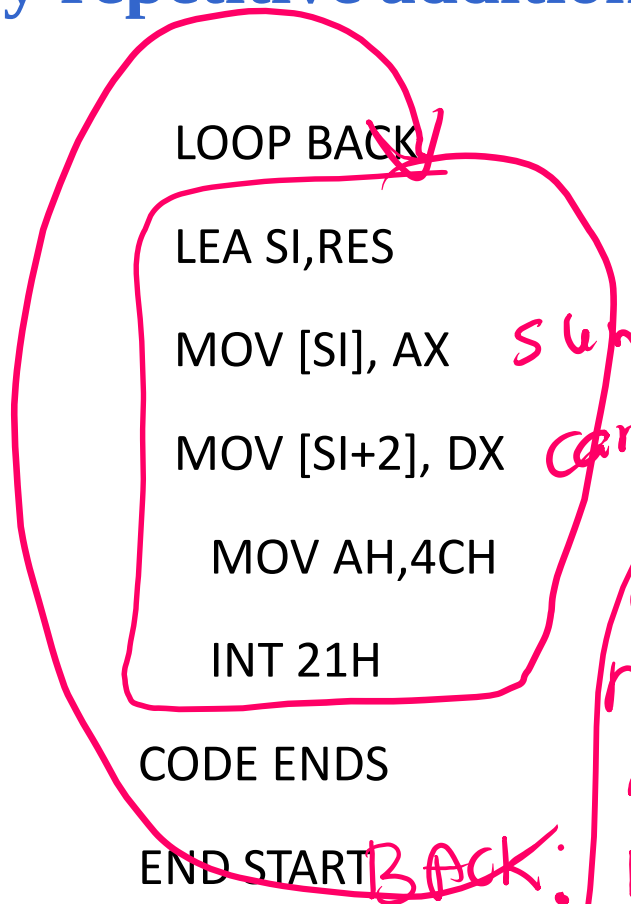
```
DATA SEGMENT
BCD DB 63H
HEX DB ?
DATA ENDS
CODE SEGMENT
ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA
MOV DS, AX
MOV AL, BCD
MOV BL, 0AH
MOV CL, 04H
SHR AL, CL
MUL BL
MOV CL, BCD
AND CL, 0FH
ADD AL, CL
MOV HEX, AL
MOV AH, 4CH
INT 21H
CODE ENDS
```

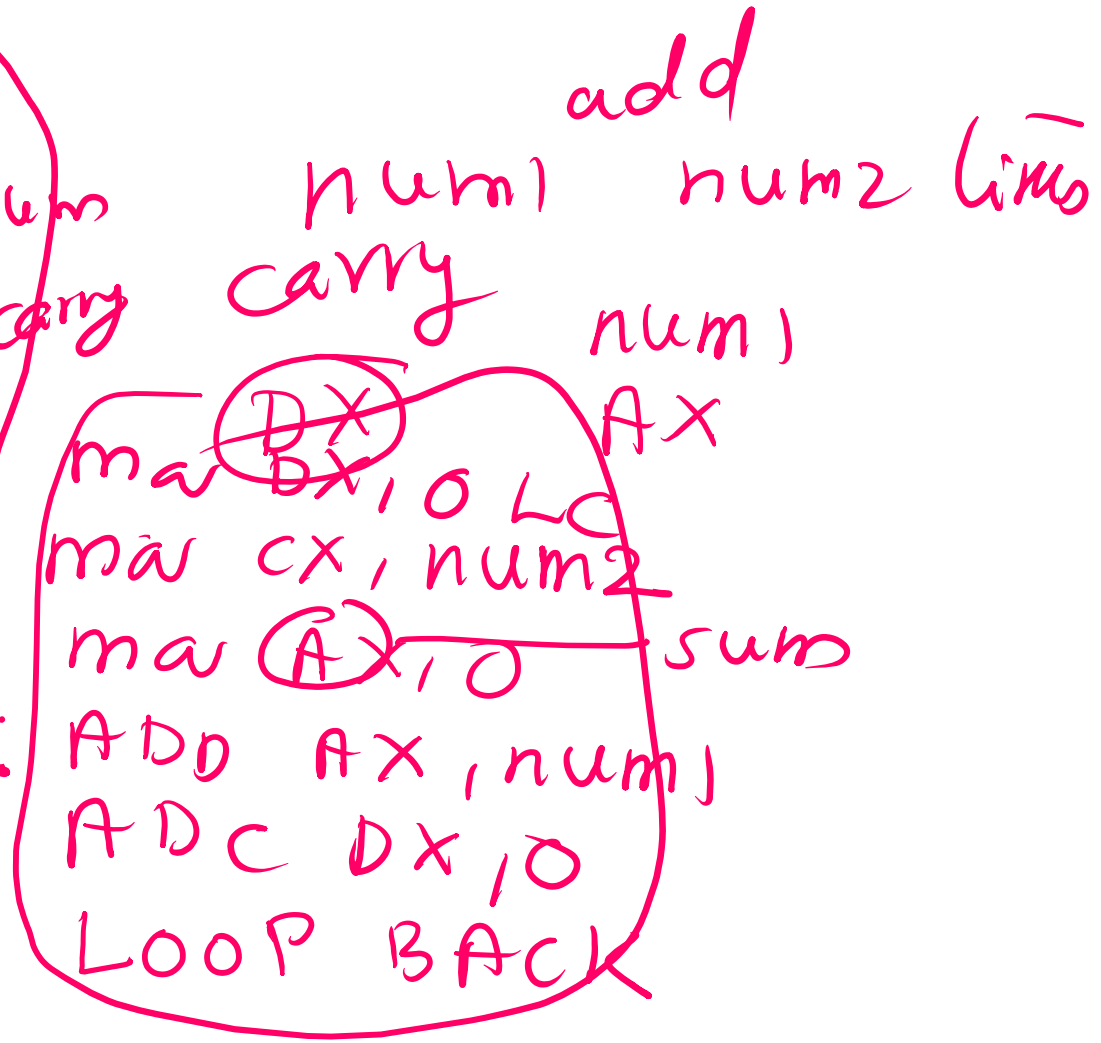

d
hex dw 0FFFFL
bcd
db
rem db 5 dup(0)
rem
bcd+1 ← 55
bcd+2 ← 06
mov bcd+1, ...
AX ← 0019
AL ← 5
54
0
2
4
254
00
0FFFF
BX
0004
0001
20002

Write an assembly language program to perform multiplication of 16 bit unsigned numbers by repetitive addition.

```
DATA SEGMENT
 NUM1 DW 1223H
 NUM2 DW 23H
 RES DW 2 DUP(?)
DATA ENDS
CODE SEGMENT
 ASSUME CS:CODE, DS:DATA
 START: MOV AX,DATA
        MOV DS, AX
        MOV AX,0
        MOV DX,0
        MOV CX,NUM2
BACK:   ADD AX,NUM1
        ADC DX,0
        LOOP BACK
        LEA SI,RES
        MOV [SI], AX
        MOV [SI+2], DX
          MOV AH,4CH
          INT 21H
CODE ENDS
END START
```







Write an ALP to find the GCD of two unsigned bytes stored in memory.

```
data segment
   num dw 12, 8
   res dw ?
data ends
stack segment
   dw 100 dup(?)
   tos label word
stack ends
code segment
assume cs:code, ds:data, ss:stack
start: mov ax,data
          mov ds,ax
          mov ax, stack
          mov ss,ax
          lea sp, tos
          mov ax, num
          mov bx, num+1
          call gcd
          mov res, ax
          mov ah, 4ch
          int 21h
          gcd proc
          cmp ax, bx
          je exit
          Jb bga
          sub ax, bx
          jmp gcd
          bga: sub bx, ax
          jmp gcd
          exit: ret
          gcd endp
code ends
end start
```

Write an ALP to find the LCM of two unsigned bytes stored in memory.

```
data segment
  num db 25,15
  lcm dw ?
data ends
stack segment
  dw 100 dup(?)
  tos label word
stack ends
code segment
Assume cs:code, ds:data, ss:stack
start: mov ax,data
          mov ds,ax
          mov ax, stack
          mov ss,ax
           lea sp, tos
          mov ah,0
          mov al,num
          mov bl,num+1
back:  push ax
          div bl
          cmp ah,0
          jz down
          pop ax
          add al,num
          adc ah,0
          jmp back
down:pop lcm
mov ah,4ch
        int 21h
code ends
end start
```

Write an ALP to sort an array in ascending order –selection sort

```
DATA SEGMENT
ARRAY DB 10 DUP(?)
LEN EQU 10
DATA ENDS

CODE SEGMENT
 ASSUME CS:CODE,DS:DATA
 START: MOV AX,DATA
MOV DS,AX
MOV CX,LEN-1 ; No. of passes
LEA SI,ARRAY
UP1: MOV BX,CX ; No. of comparisons in a pass
MOV DI, SI
UP: MOV AL,[SI]
CMP AL, [DI+1]
 JBE SKIP
XCHG AL, [DI+1]
MOV [SI],AL
SKIP: INC DI
DEC BX
JNZ UP
INC SI
LOOP UP1
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

Write an ALP to sort an array in ascending order –Bubble sort

```
DATA SEGMENT
ARRAY DB 10 DUP(?)
LEN EQU 10
DATA ENDS

CODE SEGMENT
 ASSUME CS:CODE,DS:DATA
 START: MOV AX,DATA
        MOV DS,AX
        MOV CX,LEN-1 ; No. of passes
        LEA SI,ARRAY
 UP1:MOV BX, CX
     LEA SI,ARRAY
UP: MOV AL,[SI]
     CMP AL,[SI+1]
      JBE SKIP
      XCHG AL,[SI+1]
      MOV [SI],AL
 SKIP: INC SI
      DEC BX
      JNZ UP
      LOOP UP1
       MOV AH,4CH
      INT 21H
      CODE ENDS
      END START
```

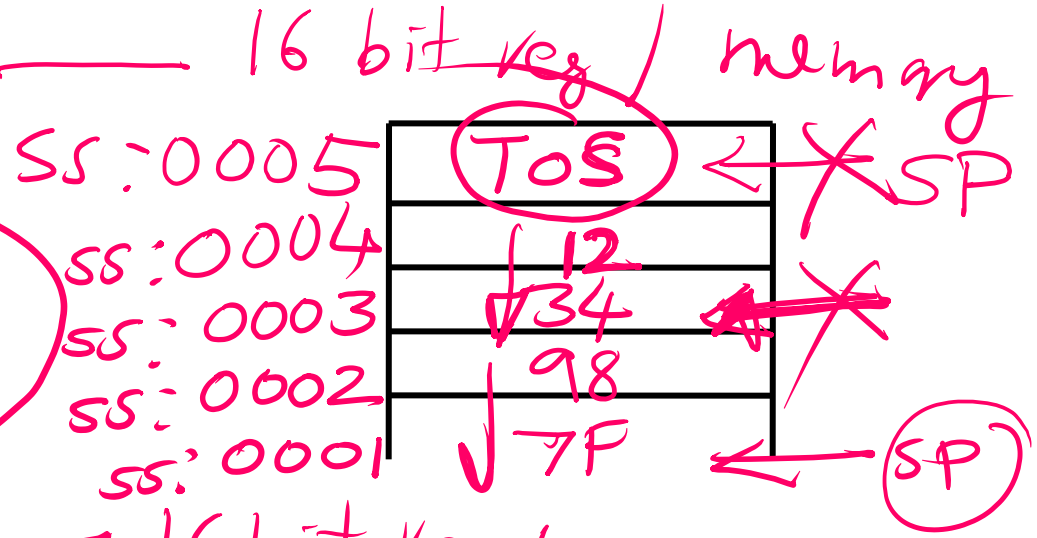
PUSH and POP instructions

PUSH the data to stack

PUSH src — 16 bit reg / memory

PUSH AX

SP = SP-2;
(SS:SP) ← (src)



PUSH AX

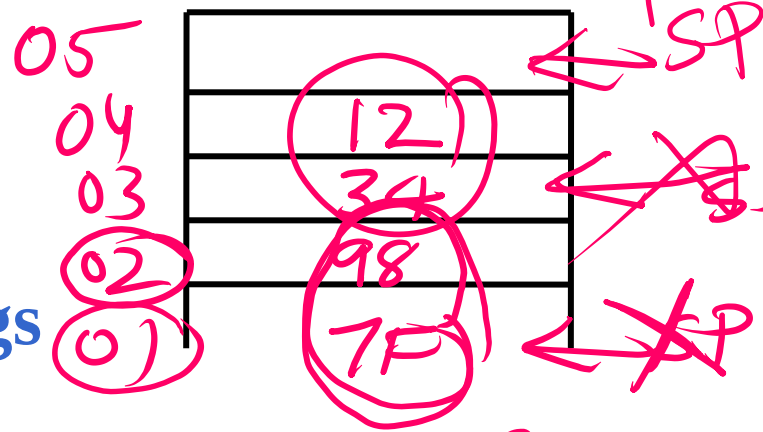
PUSH [BX]

PUSH [0000H]

POP dst — 16 bit reg / memory

POP the data from stack

POP AX
(dst) ← (SS:SP);
SP = SP + 2;



mov AX, 1234h

PUSH AX

mov BX, 987Fh

PUSH BX

POP CX CX ← 987F

POP DX DX ← 1234

PUSHF ; Push flags

POPF ; POP Flags

CALL and RET instructions

CALL <function_name>

Pushes the IP to the TOS;

RET

Pops the TOS to IP;

CALL bcd_hex

----

bcd_hex PROC <NEAR/FAR>

<function body>

RET

bcd_hex ENDP

Write an ALP to find the factorial of unsigned byte using recursion

```
stack segment
  stk dw 100 dup(?)
  tos label word
stack ends

data segment
  num db 07
  res dw ?
data ends

code segment
Assume cs:code, ds:data,
ss:stack
start: mov ax, data
       mov ds,ax
       mov ax,stack
       mov ss,ax
       lea sp,tos
       mov al,num
       mov ah,0
       call fact
       mov ah,4ch
       int 21h
fact proc
       cmp ax,01h
       je exit
       push ax
       dec ax
       call fact
       pop ax
       mul res
       mov res,ax
       ret
exit: mov res,01
       ret
fact endp
code ends
end start
```

Instructions

LAHF – Load AH with lower byte of FLAG register

SAHF – Store AH in lower byte of flag register

JCXZ – Jump to label if CX == 0

LOOPZ/LOOPE – Decrement CX; Jump to label if (CX != 0 & Z ==1)

LOOPNZ/LOOPNE - Decrement CX; Jump to label if (CX != 0 & Z ==0)

XLAT – Table translation (Operation performed - MOV AL, [BX+AL])

MACRO

- A macro is a group of instructions that perform one task, just as a procedure performs one task.
- The difference is that a procedure is accessed via a CALL instruction, whereas a macro, and all the instructions defined in the macro, is inserted in the program at the point of usage.
- Creating a macro is very similar to creating a new opcode, which is actually a sequence of instructions, in this case, that can be used in the program.
- You type the name of the macro and any parameters associated with it, and the
- assembler then inserts them into the program.
- Macro sequences execute faster than procedures because there is no CALL or RET instruction to execute.

MACRO

```
MOVE MACRO A,B
PUSH AX
MOV AX,B
MOV A,AX
POP AX
ENDM
DATA SEGMENT
VAR1 DB 5
VAR2 DB ?
VAR3 DB ?
DATA ENDS
```

```
CODE SEGMENT
ASSUME CS:CODE, DS:DATA
START : MOV AX, DATA
        MOV DS, AX
        MOVE VAR2, VAR1
        MOVE  VAR3, VAR2
CODE ENDS
END START
```

MACRO

```
PUSH_ALL MACRO
PUSH AX
PUSH BX
PUSH CX
PUSH DX
ENDM
```

```
SWAPIF_AGB A, B
LOCAL DOWN
MOV AL, A
CMP AL, B
JBE DOWN
XCHG AL, B
MOV A, AL
DOWN : NOP
ENDM
```

DOS SERVICES

DOS contains many functions that can be accessed by outside programs. These functions are invoked using assembly language instruction INT 21H.

Character input with echo:

MOV AH, 01H

INT 21H

This instruction reads a character from the standard input device (keyboard) and echoes it to the standard output device (screen). AL register contains the ASCII value of the character input.

Display character:

MOV AH, 02H

INT 21H

This instruction displays a character at the standard output device. DL register contains the ASCII value of the character to be displayed

DOS SERVICES

Read/Display a character:

Read :

MOV AH, 06H
MOV DL, OFFH
INT 21H

This instruction reads a character from the standard input device (keyboard) without waiting for a keypress. AL register contains the ASCII value and Z flag is RESET if key if pressed. Else AL contains ZERO and Z flag is SET.

Display:

MOV AH, 06H
MOV DL, 00-FEH
INT 21H

DOS SERVICES

Buffered input (Read a string)

```
MOV AH 0AH
LEA DX, MAXLEN
INT 21H
```

```
MAXLEN DB 10
ACTLEN DB ?
STR DB 10 DUP(?)
```

Can read up to 10- characters including ENTER key. ACTLEN is actual number of characters entered excluding enter key. String is terminated with enter key.

DOS SERVICES – Display a string

Character input without echo:
MOV AH, 08H
INT 21H
This instruction reads a character from the keyboard without echoing it on the screen or waits until character is available. AL register contains the character input.
Display character string:
MOV AH, 09H
INT 21H
This instruction displays a string of characters on the screen. The string must be terminated with the character '$", which is not displayed. DS: DX register pair contains segment: offset of the string.

DOS SERVICES – Read a character and display on the next line

```
CODE SEGMENT
ASSUME CS: CODE
START:
MOV AH, 01H ; Read char
INT 21H

PUSH AX

MOV DL, 0DH; CR
MOV AH, 02H
INT 21H
MOV DL, 0AH ; LF – Cursor to beginning of next line
MOV AH, 02
INT 21H

POP AX
MOV DL, AL ; Display the char
MOV AH,02H
INT 21H

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

String Instructions

You must ensure SI and DI are offsets into DS and ES respectively.

Direction Flag (0 = Up, 1 = Down)
CLD - Increment addresses (left to right)
STD - Decrement addresses (right to left)

String Instructions

Mnemo-Nic	meaning	format	Operation	
MOVS	Move string DS:SI →ES:DI	MOVSB /MOVS W	(ES:DI) ← (DS:SI) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	
CMPS	Compare string DS:SI →ES:DI	CMPSB / CMPS W	Set flags as per (DS:SI) - (ES:DI) (SI) ← (SI) ± 1 or 2 (DI) ← (DI) ± 1 or 2	

String Instructions

Mnemo-Nic	meaning	format	Operation
SCAS	Scan string	SCASB/ SCASW	Set flags as per (AL or AX) - (ES:DI) (DI) ← (DI) ± 1 or 2
LODS	Load string	LODSB/ LODSW	(AL or AX) ← (DS:SI) (SI) ← (SI) ± 1 or 2
STOS	Store string	STOSB/ STOSW	(ES:DI) ← (AL or AX) (DI) ← (DI) ± 1 or 2

String Instructions

Prefix		Meaning
REP		Repeat while not end of string Execute the string instr. Dec CX. Repeat if CX ≠ 0.
REPE/REPZ		Repeat if (CX ≠ 0 and ZF = 1)
REPNE/REPNZ		Repeat if (CX ≠ 0 and ZF = 0)

Copy string

```
DATA SEGMENT
STR1 DB ‘MIT Manipal'
LEN EQU $-STR1
STR2 DB 20 DUP(0)
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA,ES:DATA
 START: MOV AX,DATA
MOV DS,AX
MOV ES,AX
LEA SI,STR1
LEA DI,STR2
MOV CX,LEN
CLD
REP MOVSB
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

Reverse string

```
DATA SEGMENT
STR1 DB ‘ICTDEPT'
LEN EQU $-STR1
STR2 DB 20 DUP(0)
DATA ENDS
CODE SEGMENT
ASSUME CS:CODE,DS:DATA,ES:DATA
 START: MOV AX,DATA
MOV DS,AX
MOV ES,AX
LEA SI,STR1
LEA DI,STR2+LEN-1
MOV CX,LEN
UP: CLD
LODSB
STD
STOSB
LOOP UP
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

Check for Palindrome

```
DATA SEGMENT
 STR1 DB ‘MADAM'
LEN EQU $-STR1
STR2 DB 20 DUP(0)
MES1 DB 10,13,'WORD IS PALINDROME$'
MES2 DB 10,13,'WORD IS NOT PALINDROME$'
DATA ENDS
CODE SEGMENT
 ASSUME CS:CODE,DS:DATA,ES:DATA
 START: MOV AX,DATA
MOV DS,AX
MOV ES,AX
LEA SI,STR1
LEA DI,STR1+LEN-1
MOV CX,LEN
SHR CX,1 ; LEN/2
UP: CLD
LODSB
STD
SCASB
LOOPZ UP
JNZ NOTPALIN
LEA DX,MES1
MOV AH,09H
INT 21H
JMP EXIT
 NOTPALIN: LEA DX,MES2
MOV AH,09H
INT 21H
EXIT: MOV AH,4CH
INT 21H
CODE ENDS
END START
```