

**MICRO PROCESSORS & MICRO CONTROLLER LAB**  
**MANUAL**  
**DEPARTEMENT**  
**OF**  
**ELECTRONICS & COMMUNICATION ENGINEERING**

**BY:**

**P.RAJESH** M.Tech

## **PROCEDURE OF MICRO PROCESSOR ASSEMBLER**

1. Copy downloaded assembler folder in installed OS drive
2. **CREATE A FOLDER IN ANY DRIVE (Eg: D: Drive )**
3. Open created folder and create a tex file document
4. Type programs in text files
5. Save text file with “.asm” extention (Eg: RSH.asm)
6. Text document is converted in to “asm file”.
7. Go to start menu
8. Open run and type “CMD” so that command will open
9. Type drive extension (eg: D: press enter) so drive is opened
10. If created folder is present in installed OS drive (eg: C: drive ) then in cmd type (eg: cd c:\ and enter)
11. Type cd <space> folder name enter
12. Type “ path=c:\assembler
13. masm
14. type filename.asm {eg: rsh.asm}
15. press enter three times
16. if any errors in programs it show else continue
17. type “link”
18. type filename.obj {eg: rsh.obj}
19. press enter three times
20. type “AFDEBUG”
21. press enter
22. type L<space>filename with out .asm
23. inputs will display in stacks
24. press f1 to get input in registers
25. note down outputs and stacks & flags

**Op-code:** A single instruction is called as an op-code that can be executed by the CPU. Here the 'MOV' instruction is called as an op-code.

**Operands:** A single piece data are called operands that can be operated by the op-code. Example, subtraction operation is performed by the operands that are subtracted by the operand.

**Syntax:** SUB b, c

### **8086 microprocessor assembly language programs**

#### **Write a Program For Read a Character From The Keyboard**

```
MOV ah, 1h          //keyboard input subprogram
INT 21h             // character input
// character is stored in al
MOV c, al           //copy character from alto c
```

**Eg:**

```
MOV AX,1000h // copy content 1000h into AX reg
MOV CX,AX    // copy content AX into CX reg
INT 21h
```

#### **Write a Program For Reading and Displaying a Character**

```
MOV ah, 1h          // keyboard input subprogram
INT 21h             //read character into al
MOV dl, al          //copy character to dl
MOV ah, 2h          //character output subprogram
INT 21h             // display character in dl
```

#### **Write a Program Using General Purpose Registers**

```
ORG 100h
MOV AL, VAR1        // check value of VAR1 by moving it to the 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 the AL.
RET
VAR1 DB 22h
END
```

#### **Write a Program For Displaying The String Using Library Functions**

```
include emu8086.inc //Macro declaration
ORG 100h
PRINT 'Hello World!'
```

```

GOTOXY 10, 5
PUTC 65      // 65 – is an ASCII code for ‘A’
PUTC ‘B’
RET          //return to the operating system.
END          //directive to stop the compiler.

```

### **Arithmetic and Logic Instructions**

The 8086 processes of arithmetic and logic unit has separated into three groups such as addition, division, and increment operation. Most Arithmetic and Logic Instructions affect the processor status register.

The assembly language programming 8086 mnemonics are in the form of op-code, such as MOV, MUL, JMP, and so on, which are used to perform the operations. Assembly language programming 8086 examples

#### **Addition**

```

ORG0000h
MOV DX, #07H  // move the value 7 to the register AX//
MOV AX, #09H  // move the value 9 to accumulator AX//
Add AX, 00H   // add CX value with R0 value and stores the result in AX//
END

```

#### **Multiplication**

```

ORG0000h
MOV DX, #04H  // move the value 4 to the register DX//
MOV AX, #08H  // move the value 8 to accumulator AX//
MUL AX, 06H   // Multiplied result is stored in the Accumulator AX //
END

```

#### **Subtraction**

```

ORG 0000h
MOV DX, #02H  // move the value 2 to register DX//
MOV AX, #08H  // move the value 8 to accumulator AX//
SUBB AX, 09H  // Result value is stored in the Accumulator A X//
END

```

#### **Division**

```

ORG 0000h
MOV DX, #08H  // move the value 3 to register DX//
MOV AX, #19H  // move the value 5 to accumulator AX//
DIV AX, 08H   // final value is stored in the Accumulator AX //
END

```

Therefore, this is all about Assembly Level Programming 8086, 8086 Processor Architecture simple example programs for 8086 processors, Arithmetic and Logic Instructions. Furthermore, any queries

regarding this article or electronics projects, you can contact us by commenting in the comment section below.

## Programming in Assembly Language

CS 272

Sam Houston State Univ.

Dr. Tim McGuire

### Memory Segmentation

- Memory segments are a direct consequence of using a 20 bit address in a 16 bit processor
- Memory is partitioned into 64K ( $2^{16}$ ) segments
- Each segment is identified by a 16-bit segment number ranging from **0000h-FFFFh**
- Within a segment, a memory location is specified by a 16-bit *offset* (the number of bytes from the beginning of the segment)
- The Segment:Offset address is a *logical address*

### Segment:Offset Addresses

- **A4FB:4872h** means offset **4872h** within segment **A4FBh**
- To get the physical address, the segment number is multiplied by 16 (shifted 4 bits to the left) and the offset is added
- **A4FB0h + 4872h = A9822h** (20 bit physical address)
- There is a lot of overlap between segments; a new segment begins every 16 bytes (addresses ending in 0h)
- We call these 16 bytes a *paragraph*
- Because segments may overlap, the segment:offset address is not unique

### 8086 Registers

- Information inside the microprocessor is stored in registers (fourteen 16-bit registers)
- *data registers* hold data for an operation
- *address registers* hold the address of an instruction or data
- The address registers are divided into *segment*, *pointer*, and *index* registers
- a *status register* (called FLAGS) keeps the current status of the processor

### Data Registers: AX, BX, CX, and DX

- Available to the programmer for general data manipulation
- Some operations require a particular register

- High and low bytes of data registers can be accessed separately, i.e., AX is divided into AH and AL
- AX (accumulator) is preferred for arithmetic, logic, and data transfer operations
- BX (base register) serves as an address register
- CX (count register) frequently serves as a loop counter
- DX (data register) is used in multiplication and division

### **Pointer and Index Registers: SP, BP, SI, DI**

- SP (*stack pointer*) points to the top of the processor's stack
- BP (*base pointer*) usually accesses data on the stack
- SI (*source index*) used to point to memory locations in the data segment
- DI (*destination index*) performs same functions as SI.
- DI and SI are often used for string operations

### **Segment Registers: CS, DS, SS, ES**

- CS (*code segment*) addresses the start of the program's machine code in memory
- DS (*data segment*) addresses the start of the program's data in memory
- SS (*stack segment*) addresses the start of the program's stack space in memory
- ES (*extra segment*) addresses an additional data segment, if necessary

### **Instruction Pointer: IP**

- 8086 uses registers CS and IP to access instructions
- CS register contains the segment number of the next instruction and the IP contains the offset
- The IP is updated each time an instruction is executed so it will point to the next instruction
- The IP is not directly accessible to the user

### **The FLAGS register**

- Indicates the status of the microprocessor
- Two kinds of flag bits: *status flags* and *control flags*
- Status flags reflect the result of an instruction, e.g., when the result of an arithmetic operation is 0, ZF (*zero flag*) is set to 1 (true)
- Control flags enable or disable certain operations of the processor, e.g., if the IF (*interrupt flag*) is cleared (set to 0), inputs from the keyboard are ignored by the processor

### **Instructions Groups and Concepts**

- Data Transfer Instructions
- Arithmetic Instructions
- Logic Instructions
- Flow-control Instructions
- Processor Control Instructions
- String Instructions

### **Data Transfer Instructions**

- General instructions
  - **mov, pop, push, xchg, xlat/xlatb**
- Input/Output instructions
  - **in, out**
- Address instructions
  - **lds, lea, les**
- Flag instructions
  - **lahf, popf, pushf, sahf**

## General Instructions

- **mov** *destination, source*
- **pop** *destination*
- **push** *source*
- **xchg** *destination, source*
- **xlat(b)***table*
- Note that the destination comes first, just as in an assignment statement in C

## Examples

- **mov ax, [word1]**
  - "Move **word1** to **ax**"
  - Contents of register **ax** are replaced by the contents of the memory location **word1**
  - The brackets specify that the contents of **word1** are stored --  
**word1**==address, **[word1]**==contents
- **xchg ah, bl**
  - Swaps the contents of **ah** and **bl**
- **Illegal: mov [word1], [word2]**
  - can't have both operands be memory locations

## The Stack

- A data structure in which items are added and removed only from one end (the "top")
- A program must set aside a block of memory to hold the stack by declaring a stack segment

**stack 256**

- **SS** will contain the segment number of the stack segment -- **SP** will be initialized to **256 (100h)**
- The stack grows from higher memory addresses to lower ones

## PUSH and POP

- New words are added with **push**
- **push source**
  - **SP** is decreased by 2
  - a copy of the source contents is moved to **SS:SP**
- Items are removed with **pop**
- **pop destination**
  - Content of **SS:SP** is moved to the destination

- SP is increased by 2

### Stack example

```

push ax    ;Save ax and bx
push bx    ; on the stack
mov ax, -1  ;Assign test values
mov bx, -2
mov cx, 0
mov dx, 0
push ax    ;Push ax onto stack
push bx    ;Push bx onto stack
pop cx     ;Pop cx from stack
pop dx     ;Pop dx from stack
pop bx     ;Restore saved ax and bx
pop ax     ; values from stack

```

### Arithmetic Instructions

- Addition instructions
  - **aaa, adc, add, daa, inc**
- Subtraction instructions
  - **aas, cmp, das, dec, neg, sbb, sub**
- Multiplication instructions
  - **aam, imul, mul**
- Division instructions
  - **aad, cbw, cwd, div, idiv**

### Addition Instructions

- **aaa**
  - ASCII adjust for addition
- **adc *destination, source***
  - Add with carry
- **add *destination, source***
  - Add bytes or words
- **daa**
  - Decimal adjust for addition
- **inc *destination***
  - Increment

### ADD and INC

- ADD is used to add the contents of
  - two registers
  - a register and a memory location
  - a register and a constant
- INC is used to add 1 to the contents of a register or memory location



## Examples

- **add ax, [word1]**
  - "Add **word1** to **ax**"
  - Contents of register **ax** and memory location **word1** are added, and the sum is stored in **ax**
- **inc ah**
  - Adds one to the contents of **ah**
- **Illegal: add [word1], [word2]**
  - can't have both operands be memory locations

## Subtraction instructions

- **aas**
  - ASCII adjust for subtraction
- **cmp destination, source**
  - Compare
- **das**
  - Decimal adjust for subtraction
- **dec destination**
  - Decrement byte or word
- **neg destination**
  - Negate (two's complement)
- **sbb destination, source**
  - Subtract with borrow
- **sub destination, source**
  - Subtract

## Examples

- **sub ax, [word1]**
  - "Subtract **word1** from **ax**"
  - Contents of memory location **word1** is subtracted from the contents of register **ax**, and the sum is stored in **ax**
- **dec bx**
  - Subtracts one from the contents of **bx**
- **Illegal: sub [byte1], [byte2]**
  - can't have both operands be memory locations

## Multiplication instructions

- **aam**
  - ASCII adjust for multiply
- **imul source**
  - Integer (signed) multiply
- **mul source**
  - Unsigned multiply

## Byte and Word Multiplication

- If two bytes are multiplied, the result is a 16-bit word
- If two words are multiplied, the result is a 32-bit doubleword
- For the byte form, one number is contained in the source and the other is assumed to be in **al** -- the product will be in **ax**
- For the word form, one number is contained in the source and the other is assumed to be in **ax** -- the most significant 16 bits of the product will be in **dx** and the least significant 16 bits will be in **ax**

### Examples

- If **ax** contains **0002h** and **bx** contains **01FFh**

**mul bx**

**dx = 0000h ax = 03FEh**

- If **ax** contains **0001h** and **bx** contains **FFFFh**

**mul bx**

**dx = 0000h ax = FFFFh**

**imul bx**

**dx = FFFFh ax = FFFFh**

### Division instructions

- **aad**
  - ASCII adjust for divide
- **cbw**
  - convert byte to word
- **cwd**
  - convert word to doubleword
- **div *source***
  - unsigned divide
- **idiv *source***
  - integer (signed) divide

### Byte and Word Division

- When division is performed, there are two results, the quotient and the remainder
- These instructions divide 8 (or 16) bits into 16 (or 32) bits
- Quotient and remainder are same size as the divisor
- For the byte form, the 8 bit divisor is contained in the source and the dividend is assumed to be in **ax** -- the quotient will be in **al** and the remainder in **ah**
- For the word form, the 16 bit divisor is contained in the source and the dividend is assumed to be in **dx:ax** -- the quotient will be in **ax** and the remainder in **dx**

### Examples

- If **dx** = **0000h**, **ax** = **00005h**, and **bx** = **0002h**

**div bx**

**ax = 0002h dx = 0001h**

- If **dx** = **0000h**, **ax** = **0005h**, and **bx** = **FFFEh**

**div bx**

**ax = 0000h dx = 0005h**

**idiv bx**

**ax = FFFEH dx = 0001h**

### Divide Overflow

- It is possible that the quotient will be too big to fit in the specified destination (**al** or **ax**)
- This can happen if the divisor is much smaller than the dividend
- When this happens, the program terminates and the system displays the message "**Divide Overflow**"

### Sign Extension of the Dividend

- Word division
  - The dividend is in **dx:ax** even if the actual dividend will fit in **ax**
  - For **div**, **dx** should be cleared
  - For **idiv**, **dx** should be made the sign extension of **ax** using **cwd**
- Byte division
  - The dividend is in **ax** even if the actual dividend will fit in **al**
  - For **div**, **ah** should be cleared
  - For **idiv**, **ah** should be made the sign extension of **al** using **cbw**

### Logic Instructions

- **and destination, source**
  - Logical AND
- **not destination**
  - Logical NOT (one's complement)
- **or destination, source**
  - Logical OR
- **test destination, source**
  - Test bits
- **xor destination, source**
  - Logical Exclusive OR
- The ability to manipulate bits is one of the advantages of assembly language
- One use of **and**, **or**, and **xor** is to selectively modify the bits in the destination using a bit pattern (*mask*)
- The **and** instruction can be used to clear specific destination bits

- The **or** instruction can be used to set specific destination bits
- The **xor** instruction can be used to complement specific destination bits

### Examples

- To clear the sign bit of **al** while leaving the other bits unchanged, use the **and** instruction with **01111111b = 7Fh** as the mask

**and al,7Fh**

- To set the most significant and least significant bits of **al** while preserving the other bits, use the **or** instruction with **10000001b = 81h** as the mask

**or al,81h**

- To change the sign bit of **dx**, use the **xor** instruction with a mask of **8000h**

**xor dx,8000h**

### The NOT instruction

- The **not** instruction performs the one's complement operation on the destination
- The format is
  - **not destination**
- To complement the bits in **ax**:
  - **not ax**
- To complement the bits in **WORD1**
  - **not [WORD1]**

### The TEST instruction

- The **test** instruction performs an **and** operation of the destination with the source but does not change the destination contents
- The purpose of the **test** instruction is to set the status flags (discussed later)

### Status Flags

<i>Bit</i>	<i>Name</i>	<i>Symbol</i>
<b>0</b>	<b>Carry flag</b>	<b>cf</b>
<b>2</b>	<b>Parity flag</b>	<b>pf</b>
<b>4</b>	<b>Auxiliary carry flag</b>	<b>af</b>
<b>6</b>	<b>Zero flag</b>	<b>zf</b>
<b>7</b>	<b>Sign flag</b>	<b>sf</b>
<b>11</b>	<b>Overflow flag</b>	<b>of</b>

### The Carry Flag (CF)

- CF = 1 if there is a carry out from the msb (most significant bit) on addition, or there is a borrow into the msb on subtraction

- CF = 0 otherwise
- CF is also affected by shift and rotate instructions

### The Parity Flag (PF)

- PF = 1 if the low byte of a result has an even number of one bits (even parity)
- PF = 0 otherwise (odd parity)

### The Auxiliary Carry Flag (AF)

- AF = 1 if there is a carry out from bit 3 on addition, or there is a borrow into the bit 3 on subtraction
- AF = 0 otherwise
- AF is used in binary-coded decimal (BCD) operations

### The Zero Flag (ZF)

- ZF = 1 for a zero result
- ZF = 0 for a non-zero result

### The Sign Flag (SF)

- SF = 1 if the msb of a result is 1; it means the result is negative if you are giving a signed interpretation
- SF = 0 if the msb is 0

### The Overflow Flag (OF)

- OF = 1 if signed overflow occurred
- OF = 0 otherwise

### Shift Instructions

- Shift and rotate instructions shift the bits in the destination operand by one or more positions either to the left or right
- The instructions have two formats:
  - *opcode destination, 1*
  - *opcode destination, cl*
- The first shifts by one position, the second shifts by *N* positions, where **cl** contains *N* (**cl** is the only register which can be used)

### Left Shift Instructions

- The SHL (shift left) instruction shifts the bits in the destination to the left.
- Zeros are shifted into the rightmost bit positions and the last bit shifted out goes into CF
- Effect on flags:
  - SF, PF, ZF reflect the result
  - AF is undefined

- CF = last bit shifted out
- OF = 1 if result changes sign on last shift

### SHL example

- **dh** contains 8Ah and **cl** contains 03h
- **dh = 10001010, cl = 00000011**
- after **shl dh,cl**
  - **dh = 01010000, cf = 0**

### The SAL instruction

- The **shl** instruction can be used to multiply an operand by powers of 2
- To emphasize the arithmetic nature of the operation, the opcode **sal** (*shift arithmetic left*) is used in instances where multiplication is intended
- **Both instructions generate the same machine code**

### Right Shift Instructions

- The SHR (shift right) instruction shifts the bits in the destination to the right.
- Zeros are shifted into the leftmost bit positions and the last bit shifted out goes into CF
- Effect on flags:
  - SF, PF, ZF reflect the result
  - AF is undefined
  - CF = last bit shifted out
  - OF = 1 if result changes on last shift

### SHR example

- **dh** contains 8Ah and **cl** contains 02h
- **dh = 10001010, cl = 00000010**
- after **shr dh,cl**
  - **dh = 001000010, cf = 1**

### The SAR instruction

- The **sar** (*shift arithmetic right*) instruction can be used to divide an operand by powers of 2
- **sar** operates like **shr**, except the msb retains its original value
- The effect on the flags is the same as for **shr**
- If unsigned division is desired, **shr** should be used instead of **sar**

### Rotate Instructions

- **Rotate Left**
  - The instruction **rol** (*rotate left*) shifts bits to the left
  - The msb is shifted into the rightmost bit
  - The **cf** also gets the the bit shifted out of the msb
- **Rotate Right**
  - **ror** (*rotate right*) rotates bits to the right

- the rightmost bit is shifted into the msb and also into the **cf**

### Rotate through Carry

- **Rotate through Carry Left**
  - The instruction **rcl** shifts bits to the left
  - The msb is shifted into **cf**
  - **cf** is shifted into the rightmost bit
- **Rotate through Carry Right**
  - **rcr** rotates bits to the right
  - The rightmost bit is shifted into **cf**
  - **cf** is shifted into the msb
- See SHIFT.ASM for an example

### Flow-Control Instructions

```
%TITLE "IBM Character Display -- XASCIL.ASM"
IDEAL
MODEL small
STACK 256
CODESEG
Start: mov ax, @data ; Initialize DS to address
      mov ds, ax    ; of data segment
      mov ah, 02h   ; display character function
      mov cx, 256   ; no. of chars to display
      mov dl, 0     ; dl has ASCII code of null char
Ploop: int 21h      ; display a character
      inc dl        ; increment ASCII code
      dec cx        ; decrement counter
      jnz Ploop     ; keep going if cx not zero
Exit:  mov ah, 04Ch ; DOS function: Exit program
      mov al, 0     ; Return exit code value
      int 21h       ; Call DOS. Terminate program
      END Start     ; End of program / entry point
```

### Conditional Jumps

- **jnz** is an example of a conditional jump
- Format is

*jxxx destination\_label*

- If the condition for the jump is true, the next instruction to be executed is the one at *destination\_label*.
- If the condition is false, the instruction immediately following the jump is done next
- For **jnz**, the condition is that the result of the previous operation is not zero

### Range of a Conditional Jump

- Table 4.6 (and Table 16.4) shows all the conditional jumps
- The *destination\_label* must precede the jump instruction by no more than 126 bytes, or follow it by no more than 127 bytes
- There are ways around this restriction (using the unconditional **jmp** instruction)

## The CMP Instruction

- The jump condition is often provided by the **cmp** (*compare*) instruction:

**cmp destination, source**

- **cmp** is just like **sub**, except that the destination is not changed -- only the flags are set
- Suppose **ax** = **7FFFh** and **bx** = **0001h**

```

    cmp ax, bx
    jg  below
    zf = 0 and sf = of = 0, so control transfers to label below

```

## Types of Conditional Jumps

- Signed Jumps:
  - **jg/jnle, jge/jnl, jl/jnge, jle/jng**
- Unsigned Jumps:
  - **ja/jnbe, jae/jnb, jb/jnae, jbe/jna**
- Single-Flag Jumps:
  - **je/jz, jne/jnz, jc, jnc, jo, jno, js, jns, jp/jpe, jnp/jpo**

## Signed versus Unsigned Jumps

- Each of the signed jumps has an analogous unsigned jump (e.g., the signed jump **jg** and the unsigned jump **ja**)
- Which jump to use depends on the context
- Using the wrong jump can lead to incorrect results
- When working with standard ASCII character, either signed or unsigned jumps are OK (msb is always 0)
- When working with the IBM extended ASCII codes, use unsigned jumps

## Conditional Jump Example

- Suppose **ax** and **bx** contained signed numbers. Write some code to put the biggest one in **cx**:

```

    mov cx,ax    ; put ax in cx
    cmp bx,cx    ; is bx bigger?
    jle NEXT     ; no, go on
    mov cx,bx    ; yes, put bx in cx
NEXT:

```

## The JMP Instruction



- **jmp** causes an unconditional jump
- **jmp destination**
- jmp can be used to get around the range restriction of a conditional jump
- e.g, (this example can be made shorter, *how?*)

```

TOP:          TOP:
; body of loop ; body of loop
; over 126 bytes      dec cx
dec cx          jnz BOTTOM
jnz TOP          jmp EXIT
mov ax, bx      BOTTOM:
                jmp TOP
EXIT:
                mov ax, bx

```

## Branching Structures

- IF-THEN
- IF-THEN-ELSE
- CASE
- AND conditions
- OR conditions

### IF-THEN structure

- Example -- to compute  $|ax|$ :

```

if ax < 0 then
    ax = -ax
endif

```

- Can be coded as:

```

; if ax < 0
    cmp ax, 0    ; ax < 0 ?
    jnl endif    ; no, exit
; then
    neg ax       ; yes, change sign
; endif

```

### IF-THEN-ELSE structure

- Example -- Suppose **al** and **bl** contain extended ASCII characters. Display the one that comes first in the character sequence:

```

if al <= bl then
    display the character in al
else

```

*display the character in bl*  
**endif**

- This example may be coded as:

```


    mov ah, 2    ; prepare for display
; if al <= bl
    cmp al, bl   ; al <= bl ?
    jnbe else    ; no, display bl
; then          ; al <= bl
    mov dl, al   ; move it to dl
    jmp display
else:           ; bl < al
    mov dl, bl
display:
    int 21h     ; display it
; endif

```

### The CASE structure

- Multi-way branch structure with following form:

```

case expression
    value1 : statement1
    value2 : statement2
    
    valuen : statementn
endcase

```

- Example -- If **ax** contains a negative number, put -1 in **bx**; if 0, put 0 in **bx**; if positive, put 1 in **bx**:

```

case ax
    < 0: put -1 in bx
    = 0: put 0 in bx
    > 0: put 1 in bx
endcase

```

- This example may be coded as:

```

; case ax
    cmp ax, 0    ; test ax
    jl neg       ; ax < 0
    je zero      ; ax = 0
    jg pos       ; ax > 0
neg:

```

```

    mov bx, -1
    jmp endcase
zero:
    xor bx,bx    ; put 0 in bx
    jmp endcase
pos:
    mov bx, 0
endcase:

```

- Only one **cmp** is needed, because jump instructions do not affect the flags

## AND conditions

- Example -- read a character and display it if it is uppercase:

```

    read a character into al
    if char >= 'A' and char <= 'Z' then
        display character
    endif
; read a character
    mov ah, 1    ;prepare to read
    int 21h      ;char in al
; if char >= 'A' and char <= 'Z'
    cmp al,'A'   ;char >= 'A'?
    jnge endif   ;no, exit
    cmp al,'Z'   ;char <= 'Z'?
    jnle endif   ;no, exit
;then display character
    mov dl,al    ;get char
    mov ah,2     ;prep for display
    int 21h      ;display char
endif:

```

## OR conditions

- Example -- read a character and display it if it is 'Y' or 'y':

```

    read a character into al
    if char = 'y' or char = 'Y' then
        display character
    endif
; read a character
    mov ah, 1    ;prepare to read
    int 21h      ;char in al
; if char = 'y' or char = 'Y'

```

```

    cmp al,'y'    ;char = 'y'?
    je then      ;yes, display it
    cmp al,'Y'    ;char = 'Y'?
    je then      ;yes, display it
    jmp endif     ;no, exit
then:
    mov ah,2      ;prep for display
    mov dl,al     ;move char
    int 21h       ;display char
endif:

```

## Looping Structures

- FOR loop
- WHILE loop
- REPEAT loop

### The FOR Loop

- The loop statements are repeated a known number of times (counter-controlled loop)

```

    for loop_count times do
        statements
    endfor

```

- The **loop** instruction implements a FOR loop:

```

    loop destination_label

```

- The counter for the loop is the register **cx** which is initialized to *loop\_count*
- The **loop** instruction causes **cx** to be decremented, and if **cx** 0, jump to *destination\_label*
- The destination label must precede the **loop** instruction by no more than 126 bytes
- A FOR loop can be implemented as follows:

```

    ;initialize cx to loop_count
TOP:
    ;body of the loop
    loop TOP

```

### FOR loop example

- a count-controlled loop to display a row of 80 stars

```

    mov cx,80     ; # of stars
    mov ah,2      ; disp char fnctn
    mov dl,'*'    ; char to display

```

**TOP:**

int 21h ; display a star  
loop TOP ; repeat 80 times

**FOR loop "gotcha"**

- The FOR loop implemented with the loop instruction always executes at least once
- If **cx** = 0 at the beginning, the loop will execute 65536 times!
- To prevent this, use a **jcxz** before the loop

**jcxz SKIP**

**TOP:**

; body of loop  
loop TOP

**SKIP:**

**The WHILE Loop**

**while** *condition* **do**  
*statements*  
**endwhile**

- The condition is checked at the top of the loop
- The loop executes as long as the condition is true
- The loop executes 0 or more times

**WHILE example**

- Count the number of characters in an input line

```
count = 0
read char
while char <> carriage_return do
    increment count
    read char
endwhile
    mov dx,0 ;DX counts chars
    mov ah,1 ;read char fnctn
    int 21h ;read char into al
WHILE_:
    cmp al,0Dh ;ASCII CR?
    je ENDWHILE ;yes, exit
    inc dx ;not CR, inc count
    int 21h ;read another char
    jmp WHILE_ ;loop back
ENDWHILE:
```

- The label **WHILE\_** is used because **WHILE** is a reserved word

## The REPEAT Loop

**repeat**  
    *statements*  
**until condition**

- The condition is checked at the bottom of the loop
- The loop executes until the condition is true
- The loop executes 1 or more times

## REPEAT example

- read characters until a blank is read

```
repeat
    read character
until character is a blank
    mov ah,1    ;read char fnctn
REPEAT:
    int 21h    ;read char into al
;until
    cmp al,' ' ;a blank?
    jne REPEAT ;no, keep reading
```

- Using a **while** or a **repeat** is often a matter of personal preference. The **repeat** may be a little shorter because only one jump instruction is required, rather than two

## Digression: Displaying a String

- We've seen INT 21h, functions 1 and 2, to read and display a single character
- INT 21h, function 9 displays a character string
  - Input: **dx** = offset address of string
  - The string *must* end with a '\$' character -- The '\$' is not displayed

## The LEA Instruction

- INT 21h, function 9, expects the offset address of the string to be in **dx**
- To get it there, use the **lea** (*load effective address*) instruction

**lea destination,source**

- *destination* is a register, and *source* is a memory location
- For example, **lea dx, msg** puts the offset address of the variable **msg** into **dx**

## A digression from our digression -- program segment prefix (PSP)

- DOS prefaces each program it loads with a PSP
- The PSP contains information about the program, including any command line arguments

- The segment number of the PSP is loaded in ds, so ds does not contain the segment number of the DATASEG
- To correct this

```

mov ax,@data
mov ds,ax

```

- The assembler translates @data into a segment number
- Two instructions are necessary since a number cannot be moved directly into a segment register

So, back to printing a string...

```
%TITLE "Print String Program -- PRTSTR.ASM"
```

```
IDEAL
```

```
MODEL small
```

```
STACK 256
```

```
DATASEG
```

```
msg DB "Hello!$"
```

```
CODESEG
```

Start:

```

mov ax,@data    ;Initialize DS to address
mov ds,ax       ; of data segment
lea dx,[msg]    ;get message
mov ah,09h      ;display string function
int 21h         ;display message

```

Exit:

```

mov ah,4Ch      ;DOS function: Exit program
mov al,0        ;Return exit code value
int 21h         ;Call DOS. Terminate program
END Start      ;End of program / entry point

```

**MICROPROCESSOR LAB**  
**List Of Experiments**

**CYCLE-1:**

1. Addition of two 16-bit numbers using immediate addressing mode.
2. Subtraction of two 16-bit numbers using immediate addressing mode.
3. Addition of two 16-bit numbers using direct addressing mode.
4. Subtraction of two 16-bit numbers using direct addressing mode.
5. **Arithmetic Operation:**
  - a. Multiword addition
  - b. Multiword Subtraction
  - c. Multiplication of two 16-bit numbers
  - d. 32bit/16 division
6. **Signed operation:**
  - a. Multiplication
  - b. Division
7. **ASCII Arithmetic:**
  - a. AAA
  - b. AAS
  - c. AAM
  - d. AAD
  - e. DAA
  - f. DAS
8. **Logic Operations:**
  - a. Shift right
  - b. Shift left
  - c. Rotate Right without carry
  - d. Rotate left without carry
  - e. Rotate Right with carry
  - f. Rotate left with carry
  - g. Packed to unpacked
  - h. Unpacked to packed
  - i. BCD to ASCII
  - j. ASCII to BCD
9. **String Operation:**
  - a. String Comparison
  - b. Moving the block of string from one segment to another segment.
  - c. Sorting of string in ascending order
  - d. Sorting of string in descending order
  - e. Length of string
  - f. Reverse of string



### 1.1 ADDITION OF TWO 16 BITS NUMBERS SIGNED & UN SIGNED

ASSUME CS:CODE,DS:DATA

DATA SEGMENT

OPR1 DW 4269H

OPR2 DW 1000H

RES DW ?

DATA END

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

ADD AX,OPR2

MOV RES,AX

MOV AH,4CH (or) MOV AX,004CH

INT 21H

CODE ENDS

END START

END

RESULT: -

UNSIGNED:

INPUT: OPR1=4269H, OPR2= 1000H

OUTPUT: - 5269H

SIGNED: -

INPUT: - OPR1=9763H, OPR2= A973H

RES= 40D6H, CF=1

Or

**Mov Ax,7010H**

**Mov DS,AX**

**MOV AX,1000h (or ) MOV AX,[0001H]**

**ADD AX,2000H**

**MOV AX,004CH**

**HLT**

RESULT: -

AX=3000h

### 1.2. SUBTRACTION OF TWO 16 BITS NO:- SIGNED & UNSIGNED

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

OPR1 DW 4269H

OPR2 DW 1000H

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

SUB AX,OPR2

MOV RES,AX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

Or

**Mov Ax,7010H**

**Mov DS,AX**

**Mov AX,2200H**

**SUB AX,2000H**

**MOV AX,004CH**

**HLT**

RESULT: -

UNSIGNED:

INPUT: OPR1=4269H, OPR2= 1000H

OUTPUT: - 3269H

SIGNED: -

INPUT: - OPR1=9763H, OPR2= 8973H

RES= 0DF0H,

### 1.3. MULTIPLICATION OF TWO 16 BITS UNSIGNED

ASSUME CS:CODE,DS:DATA

DATA SEGMENT

OPR1 DW 2000H

OPR2 DW 4000H

RESLW DW ?

RESHW DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

MUL OPR2

MOV RESLW,AX

MOV RESHW,DX

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

(OR)

```
MOV AX,7000
MOV BX,2000H
MOV DS,AX
MOV AX,1000H
MUL BX
MOV [1000H],AX
MOV [1005H],DX
MOV AH,4CH
INT 21H
```

RESULT: -

UNSIGNED:

INPUT: OPR1=2000H, OPR2= 4000H

OUTPUT:- RESLW=0000H(AX)

RESHW=0800H(DX)

#### 1.4.MULTIPLICATION OF TWO 16 BITS

##### SIGNED NUMBERS

ASSUME CS:CODE,DS:DATA

DATA SEGMENT

OPR1 DW 7593H

OPR2 DW 6845H

RESLW DW ?

RESHW DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

IMUL OPR2

MOV RESLW,AX

MOV RESHW,DX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

RESULT:

**CASE (1) :----TWO POSITIVE : INPUTS:**

OPR1: 7593H

OPR2 : 6845H

**OUTPUT:**

RESLW=689FH

RESHW=2FE3H

CASE(2): ----ONE POSITIVE NUMBER&

ONE NEGITIVE NUMBER:

**INPUTS:** OPR1 = 846DH  $\leftarrow$  2'S

COMPLEMENT IS (-7593H)

OPR2 = 6845H

**OUTPUTS:** RESLW= 9761H  $\leftarrow$  2'S

COMPLEMENT

RESHW= D01CH  $\leftarrow$  OF (-2FE3689FH)

CASE(3):-----TWO NEGITATIVE NUMBERS

**INPUTS:** OPR1 = 846DH  $\leftarrow$  2'S

COMPLEMENT IS (-7593H)

OPR2 = 97BBH

**OUTPUTS:** RESLW= 689FH  $\leftarrow$  2'S

COMPLEMENT

RESHW= 2FE3H  $\leftarrow$  OF (-2FE3689FH)

#### 1.5. DIVISION OF UN SIGNED NUMBERS

ASSUME CS: CODE, DS:DATA

DATA SEGMENT

OPR1 DW 2C58H

OPR2 DW 56H

RESQ DW ?

RESR DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

DIV OPR2

MOV RESQ,AX

MOV RESR,DX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

RESULT:

**CASE (1) :---** INPUTS: OPR1: 2C58H  
OPR2 : 56H

**OUTPUT:**

RESQ=H == 0084H  
RESR=H==0000H

### 1.6. DIVISION OF SIGNED NUMBERS

ASSUME CS: CODE, DS:DATA

DATA SEGMENT

OPR1 DW 2658H

OPR2 DW 0AAH

RESQ DW ?

RESR DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

IDIV OPR2

MOV RESQ,AX

MOV RESR,DX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

RESULT:

**CASE (1) :---** INPUTS: OPR1: 2658H  
OPR2 : AAH

**OUTPUT:**

RESQ == 0039H  
RESR == 007EH

CASE(2):----- ONE POSITIVE NUMBER &  
ONE NEGATIVE NUMBER

INPUT:-- OPR1 = D908H  $\leftarrow$  2'S COMPLETE  
OF (-26F8H)

OPR2 = 56H

OUTPUT :---- RESQ= 8CH (AL)  $\leftarrow$  2'S  
COMPLETE OF (-74H)

RESR= 00H (AH)

### 2.1. ASCII ADDITION

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

Char Db 8

Char1 Db 6

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AH,00H

MOV AL,CHAR

ADD AL,CHAR1

AAA

MOV RES,AX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

**RESULT:-**

**INPUT : CHAR=8**

**CHAR1=6**

**OUTPUT:= RES= 0104(AX)  $\leftarrow$  UNPACKED  
BCD OF 14**

### 2.2 ASCII SUBTRACTION

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

Char Db 9 NO NEED INVERTED

COMAS

Char1 Db 5

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

```

MOV AH,00H
MOV AL,CHAR
SUB AL,CHAR1
AAS
MOV RES,AX
MOV AH,4CH
*INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : CHAR=9**

**CHAR1=5**

**OUTPUT:= RES= 0004(AX)**

**CASE(II):- CHAR=5**

**CHAR1=9**

**RES=00FC(AX) ← 2'S COMPLEMENT(-4)**

### 2.3. ASCII MULTIPLICATION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
NUM1 Db 09H
NUM2 Db 05H
RES Dw ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AH,00H
MOV AL,NUM1
MUL NUM2
AAM
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : NUM1=09**

**NUM2=05**

**OUTPUT:= RES= 0405(AX) ← UN PACKED  
BCD OF 45**

### 2.4. ASCII DIVISION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
DIVIDEND DW 0607H
DIVISIOR DB 09H
RESQ DB ?
RESR DB ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,DIVIDEND
AAD
MOV CH,DIVISIOR
DIV CH
MOV RESQ,AL
MOV RESR,AH
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : DIVIDEND=0607H ←**

**UN PACKED BCD OF 67**

**DIVISIOR=09H**

**OUTPUT:= RESQ= 07(AL)**

**RESR=04(AH)**

### 3.1. LOGICAL AND OPERATION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 6493H
OPR2 DW 1936H
RES Dw ?
DATA ENDS
CODE SEGMENT
START:

```

```

MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
AND AX,OPR2
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : OPR1=6493H**

**OPR2=1936H**

**OUTPUT:= RES= 0012H**

### 3.2. LOGICAL OR OPERATION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 6493H
OPR2 DW 1936H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
OR AX,OPR2
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : OPR1=6493H**

**OPR2=1936H**

**OUTPUT:= RES= 7DB7H**

### 3.3. LOGICAL XOR OPERATION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT

```

```

OPR1 DW 6493H
OPR2 DW 1936H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
XOR AX,OPR2
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : OPR1=6493H**

**OPR2=1936H**

**OUTPUT:= RES= 7DA5H**

### 3.4. LOGICAL NOT OPERATION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 6493H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
NOT AX
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : OPR1=6493H**

**OUTPUT:= RES= 9B6CH**

## **2'S COMPLEMENT**

```
ASSUME CS: CODE, DS:DATA
DATA SEGMENT
NUM1 DW 1234H
RESULT DW ?
DATA ENDS
CODE SEGMENT
START:MOV AX,DATA
MOV DS,AX
MOV AX,NUM1
NOT AX
ADD AX,01H
MOV RESULT,AX
INT 21H
END START
CODE ENDS
```

### **4.1.SHIFT ARITHMETIC/LOGICAL LEFT OPERATION**

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 1639H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
SAL AX,01H-----→ (or) ←----- SHL
AX,01H
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

**RESULT:-**  
**INPUT : OPR1=1639H**  
**OUTPUT:= RES= 2C72H**

## **4.2. SHIFT LOGICAL RIGHT OPERATION**

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 8639H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
SHR AX,01H
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

**RESULT:-**  
**INPUT : OPR1=8639H**  
**OUTPUT:= RES= 431CH**

### **4.3. SHIFT ARITHMETIC RIGHT OPERATION**

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 8639H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
SAR AX,01H
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

**RESULT:-**

**INPUT : OPR1=8639H**

**OUTPUT:= RES= C31CH**

#### **4.4. ROTATE RIGHT WITH OUT CARRY**

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

OPR1 DW 1639H

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

ROR AX,01H

MOV RES,AX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

**RESULT:-**

**INPUT : OPR1=1639H**

**OUTPUT:= RES= 8B1CH**

#### **4.5. ROTATE RIGHT WITH CARRY**

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

OPR1 DW 1639H

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

ROR AX,01H

MOV RES,AX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

**RESULT:-**

**INPUT : OPR1=1639H**

**OUTPUT:= RES= 0B1CH**

#### **4.6. ROTATE LEFT WITH OUT CARRY**

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

OPR1 DW 8097H

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

ROL AX,01H

MOV RES,AX

MOV AH,4CH

INT 21H

CODE ENDS

END START

END

**RESULT:-**

**INPUT : OPR1=8097H**

**OUTPUT:= RES= 012FH**

#### **4.7. ROTATE LEFT WITH CARRY**

ASSUME CS: CODE,DS:DATA

DATA SEGMENT

OPR1 DW 8097H

RES DW ?

DATA ENDS

CODE SEGMENT

START:

MOV AX,DATA

MOV DS,AX

MOV AX,OPR1

RCL AX,01H

MOV RES,AX

MOV AH,4CH

```

INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : OPR1=8097H**

**OUTPUT:= RES= 012EH**

### 5.1. MOVE BLOCK

```

ASSUME CS:CODE,DS:DATA,ES:EXTRA
DATA SEGMENT
STR DB 04H,0F9H,0BCH,98H,40H
COUNT EQU 05H
DATA ENDS
EXTRA SEGMENT
ORG 0010H
STR1 DB 05H DUP(?)
EXTRA ENDS
CODE SEGMENT
START:
mov ax,DATA
MOV DS,AX
MOV ES,AX
MOV SI,OFFSET STR
MOV DI,OFFSET STR1
MOV CL,COUNT
CLD
REP MOVSB
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT :**

**STR(DS:0000H)=04H,F9H,BCH,98H,40H**

**OUTPUT:= STR1(DS:0010H)=**

**04H,F9H,BCH,98H,40H**

### 5.2. REVERSE STRING

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT

```

```

STR DB 01H,02H,03H,04H
COUNT EQU 02H
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV Cx,COUNT
MOV SI,OFFSET STR
MOV DI,0003H
BACK: MoV AL,[SI]
XCHG [DI],AL
MOV [SI],AL
INC SI
DEC DI
DEC CL
JNZ BACK
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : STR(DS:0000H)=01H,02H,03H,04H**

**OUTPUT:= STR(DS:0000H)=**

**04H,03H,02H,01H**

### 5.3. LENGTH OF THE STRING

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
STR DB 01H,03H,08H,09H,05H,07H,02H
LENGTH DB ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AL,00H
MOV CL,00H
MOV SI,OFFSET STR
BACK: CMP AL,[SI]
JNC GO
INC CL

```



```

INC SI
JNZ BACK
GO:MOV LENGTH,CL
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT :**

**STR(DS:0000H)=01H,03H,08H,09H,05H,07H,02H**

**OUTPUT:= LENGTH=07H[CL]**

#### 5.4. STRING COMPARISION

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
STR DB 04H,05H,07H,08H
COUNT EQU 04H
ORG 0010H
STR1 DB 04H,06H,07H,09H
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV SI,OFFSET STR
MOV DI,OFFSET STR1
MOV CL,COUNT
CLD
REP CMPSB
MOV AH,4CH
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**INPUT : STR(DS:0000H)=04H,05H,07H,08H  
STR(DS:0000H)= 04H,06H,07H,09H**

**OUTPUT:= IF STR=STR1 THEN ZF=1  
IF STR ≠ STR1 THEN ZF=0**

#### 5.5. DOS/BIOS PROGRAMMING

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
MSG DB 0DH,0AH,"WELCOME TO MICRO  
PROCESSOR LAB", 0DB,0AH,"$"
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,09H
MOV DX,OFFSET MSG
INT 21H
CODE ENDS
END START
END

```

**RESULT:-**

**WELCOME TO MICRO  
PROCESSORS LAB**

#### 6.1. PACKED BCD TO UNPACKED BCD

```

ASSUME CS: CODE,DS:DATA
DATA SEGMENT
BCD DB 48H
UBCD DB ?
UBCD2 DB ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AL,BCD
MOV BL,AL
AND AL,0FH
MOV UBCD1,AL
MOV AL,BL
AND AL,0F0H
MOV CL,04H
ROR AL,CL
MOV UBCD2,AL
MOV AH,4CH
INT 21H

```

```
CODE ENDS
END START
END
```

RESULT:-

INPUT: 48

OUTPUT:- 0408

## 6.2. PACKED BCD TO ASCII

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
BCD DB 49H
ASCII1 DB ?
ASCII2 DB ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AL,BCD
MOV BL,AL
AND AL,0FH
OR AL,30H
MOV ASCII1,AL
MOV AL,BL
AND AL,0F0H
MOV CL,04H
ROR AL,CL
MOV ASCII2,AL
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

RESULT:-

INPUT: 49

OUTPUT:- 3439

## 7.1. ASCENDING ORDER

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
NUMS DW 5H,4H,3H,2H,1H
COUNT EQU 05H
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,0000H
MOV DL,COUNT-1
BACK1:MOV CL,DL
MOV SI,OFFSET NUMS
BACK: MOV AX,[SI]
CMP AX,[SI+2]
JC GO
XCHG [SI+2],AX
MOV [SI],AX
GO:INC SI
INC SI
LOOP BACK
DEC DL
JNZ BACK1
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

RESULT:-

INPUT: 5H,4H,3H,2H,1H

OUTPUT:- 1H,2H,3H,4H,5H

## 7.2. DESCENDING ORDER

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
NUMS DW 1H,2H,3H,4H,5H
COUNT EQU 05H
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,0000H
MOV DL,COUNT-1
BACK1:
MOV CL,DL
MOV SI,OFFSET NUMS
BACK: MOV AX,[SI]
CMP AX,[SI+2]
JNC GO
XCHG AX,[SI+2]
MOV [SI],AX
GO:
INC SI
INC SI
LOOP BACK
DEC DL
JNZ BACK1
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

RESULT:-

INPUT: 1H,2H,3H,4H,5H

OUTPUT:- 5H,4H,3H,2H,1H

## 8.1. MAXIMUM NUMBER

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
DLMS DW 0001H,0009H,0008H,0005H,0010H
COUNT EQU 05H
MAX DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV CX,COUNT-1
MOV SI,OFFSET DLMS
MOV AX,[SI]
BACK : CMP AX,[SI+2]
JNC GO
XCHG AX,[SI+2]
GO: INC SI
INC SI
LOOP BACK
MOV MAX,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

RESULT:-

INPUT: 0001H,0009H,0008H,0005H,0010H

OUTPUT:- STORED IN A&B LOCATION OF DS

## 8.2. MINIMUM NUMBER

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
DLMS DW
0007H,0009H,000FH,0008H,0005H,0006H
COUNT EQU 06H
MIN DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV CX,COUNT-1
MOV SI,OFFSET DLMS
MOV AX,[SI]
BACK : CMP AX,[SI+2]
JC GO
XCHG AX,[SI+2]
GO: INC SI
INC SI
LOOP BACK
MOV MIN,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

RESULT:-

INPUT:

0007H,0009H,000FH,0008H,0005H,0006H

OUTPUT:- 0005H IS IN C&D LOCATION

## 9.1. 2'S COMPLEMENT

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
OPR1 DW 45H
RES DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,OPR1
NEG OPR1
MOV RES,AX
MOV AH,4CH
INT 21H
CODE ENDS
END START
END
```

RESULT:-

INPUT: OPR1=0045H

OUTPUT:- FFBBH

## 9.2. AVERAGE OF TWO NUMBERS

```
ASSUME CS: CODE,DS:DATA
DATA SEGMENT
NO1 DB 0FH
NO2 DB 05H
AVG DW ?
DATA ENDS
CODE SEGMENT
START:
MOV AX,DATA
MOV DS,AX
MOV AX,00H
MOV AL,NO1
MOV AL,NO2
ADD AL,NO2
SAR AX,01H
MOV AVG,AX
```

```

INT 21H
CODE ENDS
END START
END

```

RESULT:-

INPUT: NO1=0FH,, NO2=05H

OUTPUT:- 0AH IS IN ACCUMULATOR REGISTER

## ADDITIONAL PROGRAMS

### STRING OPERATIONS

#### Left shift operation

```

mov si,7800H
mov cl,[si]
mov ax,si
add ax,cx
mov si,ax
mov al,[si]
mov [si],00h
dec cl
back : dec si
mov bl,[si]
mov [si],al
mov al,bl
dec cl
jnz back
hlt

```

#### RIGHT shift operation

```

mov si,7800h
mov cl,[si]
inc si
mov al,[si]
mov [si],00h
dec ci
back : inc si
mov bl,[si]
mov [si],al

```

```

mov al,bl
dec cl
jnz back
hlt

```

### COUNTING OF OCCURRENCE OF A LETTER IN A GIVEN STRING

```

Data Segment
STR DB 'AIMTOBECOMEAASTRONUT'
A DB 0H
MSG1 DB 10,13,'COUNT OF A IS:$'
DATA ENDS
DISPLAY MACRO MSG
MOV AH,9
LEA DX,MSG
INT 21H
ENDM
CODE SEGMENT
ASSUME CS:CODE,DS:DATA
START:
MOV AX,DATA
MOV DS,AX
LEA SI,STR
MOV CX,10
CHECK:
MOV AL,[SI]
CMP AL,'A'
JNE N1
INC A
N1:
CMP AL,'A'
JNE N2
INC A
N2:INC SI
LOOP CHECK
MOV AL,A
DISPLAY MSG1
MOV DL,A
ADD DL,30H
MOV AH,2
INT 21H
MOV AH,4CH
INT 21H

```

CODE ENDS  
END START

RESULT:  
Count A is 2

### Reversing a String

```
Data segment
N1 db 'communication'
Len equ ($-n1)
N2 db len dup(00)
Data ends
Code segment
Assume cs:code,ds:data
Start:
mov ax,data
mov ds,ax
mov bx,offset n1
mov si,bx
mov di,offset n2
add di,len
cld
mov cx,len
a1:mov al,[si]
mov [di],al
inc si
dec di
loop a1
mov [di],'$'
mov dx,offset n1
mov ah,09
int 21h
mov dx,offset n2+1
mov ah,09
int 21h
hlt
code ends
end start
```

Result :  
Communication

### Display a Given String

```
Prog:
.model small
.stack 100h
.data
String1 db 'Electronics and Communication
Engineering $'
.code
Main proc
Mov AX,@data
Mov DS,AX
MOV AH,09H
MOV DX,OFFSET STRING1
INT 21H
MOV AH,4CH
INT 21H
MAIN ENDP
END MAIN
```

RESULT :  
Electronics and Communication Engineering

### FACTORIAL NUMBER

```
MOV AX,06H
MOV CX,AX
AHEAD:DEC CX
JNZ COPY
PROCEED:MUL CX
JNZ AHEAD
COPY:MOV DX,AX
HLT
```

RESULT:

AX=0006 BX=0000 CX=0005 DX=0006  
SP=FFFE BP=0000 SI=0000 DI=0000  
DS=0100 ES=0100 SS=0100 CS=0100  
IP=000E NV UP EI PL NZ NA PE NC  
0100:000E F4 HLT

## SUM OF NUMBERS OF AN ARRAY

```
MOV SI,7800H
MOV CX,[SI]
INC SI
INC SI
MOV AX,000H
LOOP: ADD AX,[SI]
INC SI
INC SI
DEC CI
JNZ LOOP
MOV [SI],AX
INT 21H
```

```
AX=0000 BX=0000 CX=00A0 DX=0000
SP=FFFE BP=0000 SI=78C4 DI=0000
DS=0100 ES=0100 SS=0100 CS=0100
IP=000E NV UP EI PL NZ NA PO NC
0100:000E FEC9 DEC CL
```

## All logic Instruction in one program

```
MOV AX,7000H
MOV DS,AX
```

```
MOV AX,0200H
MOV BX,0800H
```

```
AND AX,BX
MOV [1000H],AX
```

ENDS

```
MOV AX,0200H
MOV BX,0200H
OR AX,BX
MOV [5000H],AX
ENDS
```

```
MOV AX,0200H
```

```
MOV BX,0400H
XOR AX,BX
ENDS
```

```
MOV AX,1234H
NOT AX
ENDS
```

```
MOV AX,1234H
NOT AX
ADD AX,01H
MOV [5300H],AX
ENDS
```

## Shift Instructions in One program

```
MOV AX,7010H
MOV DS,AX
```

```
MOV AX,8639H
SHR AX,01H
MOV [1000H],AX
ENDS
```

```
MOV AX,8639H
SAR AX,01H
MOV [1005H],AX
ENDS
```

```
MOV AX,0900H
SHL AX,02H
MOV [2000H],AX
ENDS
```

```
MOV AX,0900H
SAL AX,02H
MOV [2005H],AX
ENDS
```

```
MOV AX,8000H
ROR AX,01H
MOV AX,0900H
ROL AX,01H
MOV AX,8000H
```

```
RCR AX,01H
MOV AX,9000H
RCL AX,02H
ENDS
HLT
```

### **Arithmetic Instructions in one Program**

```
MOV AX,7015H
```

```
MOV DS,AX
```

```
MOV AX,4000H
MOV BX,2000H
```

```
MUL BX
MOV [1000H],AX
MOV [1005H],DX
MOV AX,004CH
```

```
ends
```

```
MOV AX,2000H
ADD AX,BX
MOV [1006H],AX
ends
```

```
MOV AX,2000H
sub AX,BX
MOV [2006H],AX
ends
```

```
mov ax,1000h
div bx
mov [1008h],ax
mov [2000h],dx
ends
```