**LAB MANUAL**

***Computer Organization and Assembly Language***

***CSCS-203***

****

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE**

**FACULTY OF ENGINEERING & COMPUTER SCIENCES**

**NATIONAL UNIVERSITY OF MODERN LANGUAGES**

**H-9, ISLAMABAD**

**PAKISTAN**

**OBJECTIVES**

* Describe how the basic units of the Intel 8088 architecture work together to represent Integer Numbers, Floating Numbers and register representation inside the microprocessor.
* Implement assembly programs of intermediate complexity using the intel 8088 architecture.

**INTRODUCTION**

The purpose of this Lab is to teach the basics of Intel 80x86 assembly language. You will learn in this lab that assembly language (or simply” assembly”) is important because it is the principal link between the software world of high-level languages like C and Java and the hardware world of CPU design. Assembly language is the lowest-level; human-readable programming medium we can use to express complete application programs. Assembly language gives full access to the programmable features of the hardware, so a good understanding of it will provide valuable insight into the fundamentals of CPU design, the operation of the datapath, and program execution.

**SOFTWARE TOOLS**

* DOSBOX
* MASM
* Emu8086 simulator

**TABLE OF CONTENTS**

[**Preface** 02](#_Toc128393015)

[**Tools/ Technologies** 02](#_Toc128393016)

[**LAB 1: Introduction to Assembly Language** 04](#_Toc128393017)

[**LAB 2: Arithmetic Operations** 09](#_Toc128393018)

[**LAB 3: Program Flow Control Instructions** 16](#_Toc128393017)

[**LAB 4: String Operations** 28](#_Toc128393018)

[**LAB 5: Assembly Application Programs** 37](#_Toc128393017)

[**LAB 6: Logical Operations** 46](#_Toc128393018)

[**LAB 7: Shift and Rotate Operations** 4](#_Toc128393017)9

[**LAB 8: Defining and using Procedures** 55](#_Toc128393018)

[**LAB 9: Stack Operations** 61](#_Toc128393017)

[**LAB 10: File Handling in Assembly** 64](#_Toc128393018)

[**LAB 11: BIOS Level Programming** 69](#_Toc128393017)

[**LAB 12: Graphics in Assembly** 73](#_Toc128393018)

[**LAB 13: Interrupt Handling, Macros and Structures** 78](#_Toc128393017)

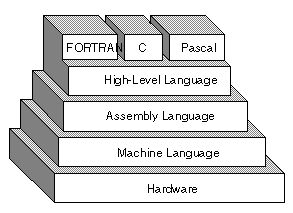
[**LAB 14: Input-Output and Parallel Port Operation** 82](#_Toc128393018)

# **LAB 1: Introduction to Assembly Language**

**Objectives:** To learn the basic commands, CPU registers and assembly language program structure.

**Introduction**

* Assemble language is the symbolic representation of machine code.
* Machine language is a Boolean language consisting of numbers that is specifically understood by a computer’s processor (the CPU).
* Of all languages, Assembly has the closest resemblance to the native machine language of the computer.
* For this reason, it is fast/speedy and it provides you direct access to the computer hardware.
* The only disadvantage is that assembly language programs are large and they would take too much time to write and maintain.
* An assembler is a program that converts source-code programs from assembly language into machine language. Call DOS/BIOS services.
* One Assembly language instruction corresponds to one machine language instruction.



**Theory**

**Central Processing Unit (CPU)**

The Central Processing Unit (CPU) is the physical device that executes instructions. The instructions that CPUs perform are generally very simple. Instructions may require the data they act on to be in special storage locations in the CPU itself called registers. The CPU can access data in registers much faster than data in memory. However, the number of registers in a CPU is limited, so the programmer must take care to keep only currently used data in registers.

**CPU Registers**

Registers are special work areas inside the CPU designed to be accessed at high speed. The registers are 16-bit long but you have the option of accessing the upper or lower halves of the four registers:

* Data Registers 16-bit: AX, BX, CX, and DX

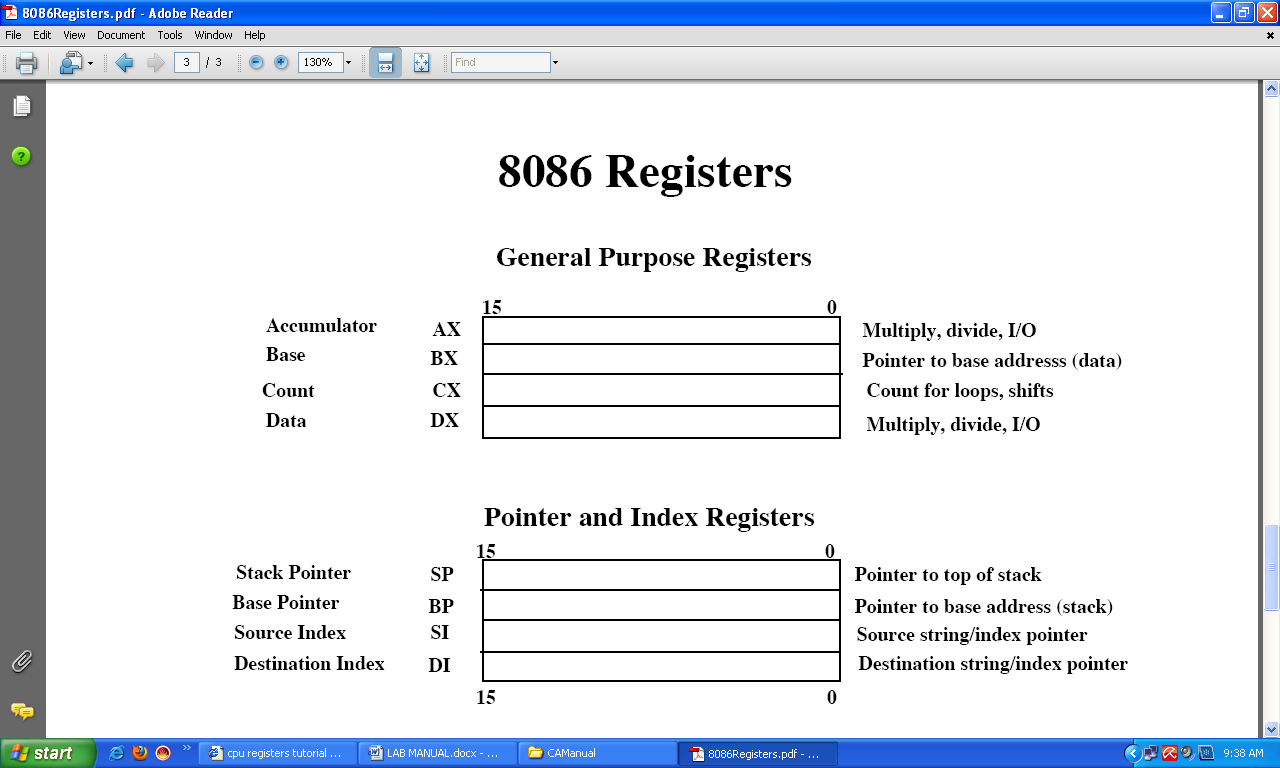
8-bit: AH, AL, BH, BL, CH, CL, DH, DL

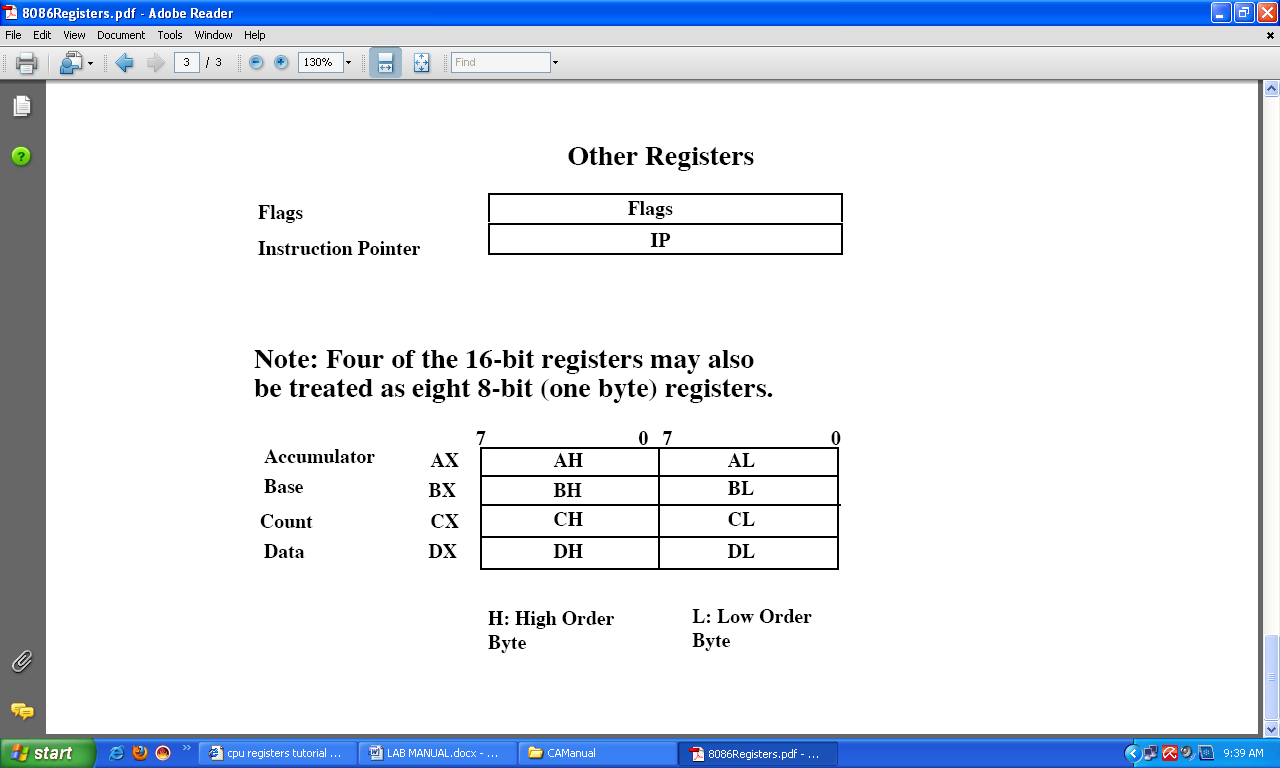
* Segment Registers CS, DS, SS, ES (Code, Data, Stack, Extra)
* Index Registers SI, DI, BP (Source, Destination, Base Pointer)
* Special Registers IP, SP (Instruction Pointer, Stack Pointer)
* Flag Registers Overflow, Direction, Interrupt, Trap, Sign, Zero, Auxiliary,

Carry, Parity

**Data Registers**

Four Registers named data registers or general purpose registers are used for arithmetic and data movement. Each register may be either addressed as 16-bit or 8-bit value. Bit positions are always numbered from right to left, starting with 0:





Each general purpose register has special attributes:

**AX** (Accumulator Register): AX is called the accumulator register because it is favored by the CPU for arithmetic operations.

**BX** (Base Register): BX register can also perform arithmetic and data movement, and it has special addressing abilities. It can hold a memory address that points to another variable.

**CX** (Counter Register): It acts as a counter for repeating or looping instructions. These instructions automatically repeat and decrement CX and quit when it is equal to 0.

**DX** (Data Register): DX is used for output purpose/display. It has a special role in multiply and divide operations. When multiplying for example DX holds the high 16-bits of the product.

**How to Start an Assembly Program**

Essential code instruction for all programs start with the following commands:

.model small

.stack 100h

.data

.code

MY FUNCTION

Mov ah,2 ; display function

Mov dl,al ; character to display

Int 21h ; interrupt

Mov ah,4ch ; exit to DOS

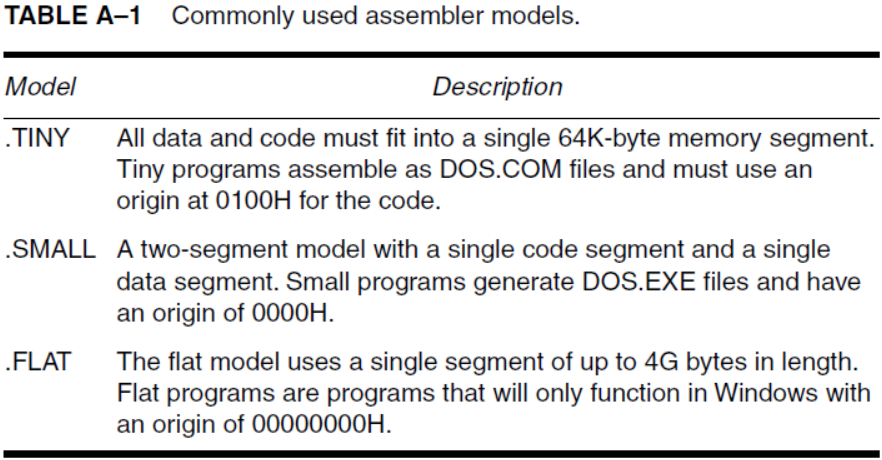
main proc

(your code is written inside this)

main endp

end main

**.model** determines the size of code and data a program can have e.g. **small**.

****

**.stack 100h** stores current contents of ax, bx registers into memory block.

**.data** corresponds to data segment, and it initializes global variables or strings. It is used for all the variable definitions.

**.code** corresponds to code segment and all instructions come in this.

**main proc** is the ‘procedure statement’ and all code is written below this.

**main endp** is the ‘end procedure’ statement.

**end main** is the last line of code.

**NOTE:** Any function can be defined between **main proc** and **main endp**.

The most basic instruction in assembly is the ‘mov’ instruction. It moves data from one location to another. It takes two operands: **mov** destination, source

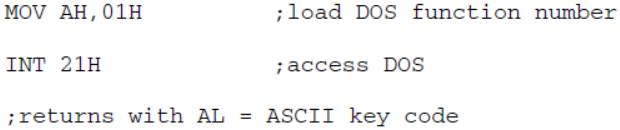
The data specified by **source is copied to destination**. Both operands may or may not be memory locations. Another important point to be kept in mind is that the source and destination operands must be same size.

* **IN/OUT Library**

**func # 1**  is for ‘input’ and **func # 2**  is for ‘output’.

Example: ah = func # (e.g. 1 or 2);

**INPUT EXAMPLE:** (function # 1 for single character input) [Read Operation]



{

e.g. mov ah,1

int 21h

**int** is for ‘interrupt’. ‘Int21h’ has the highest priority and it can invoke large no. of MS-DOS function calls.

**al** is used for single character input by the user (Default register byte).

**NOTE:** Before we call interrupt we assign ‘ah’ some value to read from.

**OUTPUT EXAMPLE:** (function # 2 for single character output/display)

mov ah,2

mov dl,'A'

int 21h

mov ah,2

mov dl,8h ;backspace

int 21h

mov ah,2

mov dl,'Z'

int 21h

e.g mov ah,2

{

mov dl, ‘A’

int 21h

**NOTE:** Default output byte is ‘dl’.

**;** is used to write comments to a line.

Example: mov ah,1; single character read in from ‘al’.

* **To Generate ENTER/New Line** following two commands are used:

0dh : (zero) carriage return (backspace).

🡨 Backspace

\_\_ New line

0ah : new row (line generation)

**4ch** returns control to DOS, is (used to exit). Example: mov ah,4ch.

## **Activity:**

* MASM must be installed and configured by using either of methods discussed.
* Must have tested a code on your installed assembler.
* Number system revision.
* Understanding of Assembly language and Assembler.
* Understanding of 8086 Data/Special purpose registers and Instructions which are discussed today.

# **LAB 2: Arithmetic Operations**

**Objectives:** To learn the basic arithmetic commands and their use in solving various problems.

**Arithmetic Operations**

**1.** **ADD**: A source operand is added to a destination operand and the sum is stored in the destination. Operands must be the same size.

Syntax: **Add** *destination, source*

Instruction Formats:

Add reg,reg Add reg,operand

Add mem,reg Add mem,operand

Add reg,mem

Example: Add al,20h ; al = al + 20h

Add al,bl ; al = al + bl

Add eax,[ebx] ; Add the 4 bytes in memory at the address contained in EBX into EAX

**2.** **Sub:** Subtracts the source operand from the destination operand.

Syntax: **Sub** *destination, source*

Instruction Formats:

Sub reg,reg Sub reg,operand

Sub mem,reg Sub mem,operand

Sub reg,mem

Example: Sub al,20h ; al = al – 20h

Sub al,bl ; al = al – bl

Sub eax,[ebx] ; Sub the 4 bytes in memory at the address contained in EBX into EAX

**3.** **Inc:** Increments or adds 1 to a register or memory operand.

Syntax: **inc** *destination*

Instruction Formats:

inc reg inc mem

Example: inc al ; al = al + 1

**4.** **Dec:** Decrements or subtracts 1 from an operand. Does not affect carry.

Syntax: **dec** *destination*

Instruction Formats:

dec reg dec mem

Example: dec al ; al = al – 1

**5.** **Neg:** It reverses the sign of a number by converting the number to its two’s complement and stores the result in the destination.

Syntax: **Neg** *destination*

Instruction Formats:

neg reg neg mem

Example: neg al ; al = -5 if it initially contained 5

**Code: (Add, Sub, Inc, Dec)**

.model small

.stack 100h

.data

.code

main proc

mov al,5

add al,2

;inc al

;sub al,1

;dec al

mov ah,2

mov dl,al

add dl,30h

int 21h

main endp

end main

**6.** **Mul:** Multiplies AL or AX by a *source* operand.

1. If the source is 8 bits, it is multiplied by AL and the product (16 bit) is stored in AX.
2. If the source is 16 bits, it is multiplied by AX and the product (32 bit) is stored in DX:AX.

Syntax: **Mul** *source* ; the destination is AX where AL is mul by source.

Instruction Formats:

mul reg mul mem

Example: mov al,2

mov bl,4

mul bl ; Ax = al\*bl = 8

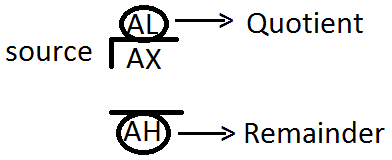
**7.** **Div:** Performs either 8 or 16-bit unsigned integer division.

1. If the divisor is 8 bits, the dividend is in AX, the quotient goes to AL, and the remainder to AH i.e. (AL = AX/operand & AH = remainder (modulus).
2. If the divisor is 16 bits, the dividend is DX:AX, the quotient is AX, and the remainder is DX.

Syntax: **Div** *source*

Instruction Formats:

div reg div mem



Example: mov ax,9

mov bl,2

div bl ; AL = 4, AH =1

**Code for Division:**

.model small

.stack 100h

.data

.code

main proc

mov ax,264

mov bl,8

div bl

;we want to display QUOTIENT digits, so divide quotient by 10

mov bh,10h ; to display values in separate digit we divide by 10

div bh ; e.g. 20/10 now al=2

mov bh,ah ;remainder moved to bh to avoid loss of contents

mov ah,2

mov dl,al ; display quotient 20/10 = 2 will display

add dl,30h ; ascii conversion

int 21h

mov ah,2

mov dl,bh ; now display the remainder contained in bh=0

add dl,30h ; ascii conversion

int 21h

main endp

end main

**Lab Code 01:** Perform the following arithmetic expression

Compute = -5 + (8 - 2)

**Hint:** Use neg, sub and add commands.

.model small

.stack 100h

.data

.code

main proc

mov al,5

neg al ; al = -5

mov bl,8

sub bl,2 ; (8-2)=6

add al,bl ; -5 + 6 =1

mov ah,2

mov dl,al

add dl,30h

int 21h

main endp

end main

**;Take Two numbers from user and add them; subtract them.**

.model small

.model small

.stack 100h

.data

.code

main proc

mov ah,1

int 21h

mov dl,al

;input second no

mov ah,1

int 21h

add dl,al

mov ah,2

sub dl,30h

int 21h

main endp

end main

.stack 100h

.data

.code

main proc

mov ah,1

int 21h

mov dl,al

;input second no

mov ah,1

int 21h

sub dl,al

mov ah,2

add dl,30h

int 21h

main endp

end main

**Lab Code 02:** To input a character at run time, convert its case (from lower to upper) and display it.

**Hint:** ASCII codes for ‘upper case’ and ‘lower case’ characters differ by 32d or 20h. (a=61h & A=41h).

.model small

.stack 100h

.data

.code

main proc

mov ah,1

int 21h

sub al,20h ; case converted

mov ah,2

;mov dl,0dh ; new line generation

;int 21h

;mov dl,0ah

;int 21h

mov dl,al

int 21h

mov ah,4ch ; terminate program and return to DOS

int 21h

main endp

end main

**Display all ASCII characters all 256**

**.**model small

**Calculate the table of 2.**

.model small

.code

main proc

mov cl,10

mov al,2 ; multiply for table of 2

mov bl,1 ; multiplier is one

label:

mul bl

mov al,2 ; restore value of 2

inc bl ; update multiplier

loop label

main endp

end main

.stack 100h

.data

.code

main proc

mov ah,2 ; display ch function

mov cx,256 ; no of ch to be displayed

mov dl,0; dl has ascii code of null ch

print:

int 21h ;display a character

inc dl ;increment ascii code

dec cx ; decrement counter

jnz print ;keep going till cx is not zero

mov ah,4ch

int 21h

main endp

end main

**Activity:**

**Code:** Print table of any number you input.

**Code:** Input any two numbers and Add, Subtract and Multiply and display.

# **LAB 3: Program Flow Control Instructions**

**Objectives:** To learn to change the sequence of execution of a program by using program flow control instructions.

**A. Flag Registers:**

The flag register is a special 16-bit register with individual bit positions assigned to show the status of CPU or the result of arithmetic operations. Each relevant bit position is given a name other positions are undefined:

A picture containing graphical user interface

Description automatically generated

**Note:** The Status of these flags can be used to make a decision to change the program flow control. The decision can be either TRUE of FALSE. Based on this decision the program executes/jumps to a certain line of code.

**B. Program Flow Control**

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

1. Unconditional Jumps
2. Conditional Jumps

**1. Unconditional Jumps:**

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

The basic syntax of JMP instruction:

Syntax: **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 is an example of JMP instruction:**

main proc

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:

main endp

**Another example:**

main proc

jmp label1 ;go to 'label1'

mov ah,2

mov dl,'k'

int 21h

label1:

mov ah,2

mov dl,'p'

int 21h

main endp

**2. Short Conditional Jumps**

Unlike JMP instruction that does an unconditional jump, there are instructions that do a conditional jump (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.

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Operands** | **Description** |
| **CMP** | REG, REG  REG, immediate  REG, memory memory, REG memory, immediate | Compare.  Algorithm: operand1 - operand2  result is not stored anywhere, only flags affected.  If  operand1 - operand2 > 0 ZF=0, Sign=0  operand1 - operand2 < 0 ZF=0, Sign=1 (**JS**)  operand1 - operand2 = 0 ZF=1 (**JZ**) **Example:**  MOV AL, 5  MOV BL, 5  CMP AL, BL ; ZF = 1 (so equal)  JZ label   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| **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'  main proc  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:     |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| **JG**  (same as **JA**) | Label | Short Jump if first operand is Greater than second operand (as set by CMP instruction). Signed.  Algorithm:  if (ZF = 0) and (SF = OF) then jump  **Example:**  include 'emu8086.inc'  main proc  MOV AL, 5  CMP AL, -5  JG label1  PRINT 'AL is not greater -5.'  JMP exit  label1:  PRINT 'AL is greater -5.'  exit:     |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| **JL**  (same as **JB**) | Label | Short Jump if first operand is Less than second operand (as set by CMP instruction). Signed.  Algorithm:  if SF = 1 then jump  **Example:**  include 'emu8086.inc'    MOV AL, 2  CMP AL, 5  JL label1  PRINT 'AL > 5'  JMP exit  label1:  PRINT 'AL < 5'  exit:     |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |

**More Conditional Jumps:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| JA | JAE | JB | JBE | JC | JL | JLE | JNA | JNAE |
| JNBE | JNC | JNG | JNGE | JNL | JNLE | JNO | JNP | JNS |
| JO | JP | JPE | JPO | JS | JNB | JNZ |  |  |

**Code-01:** Write a program to find if the number is Even or Odd? (**JE**)

**Hint:** Divide the number by 2, if the remainder is 0, its Even, else Odd.

include ‘emu8086.inc’

.model small

.code

main proc

mov ah,1 ;input the number to be checked

int 21h

mov bl,2 ; divisor/source moved to bl

div bl

cmp ah,0 ; remainder goes to AH, check it

**JE** even ; remainder zero is even

JNE odd ; remainder zero is odd

even:

print ‘Even No’

jmp exit

odd:

print ‘Odd No’

exit:

main endp

end main

**Code-02:** Count the entered characters before user presses space bar. (**JE**)

**Hint:** We use ‘cmp’ command and ‘increment’ counter if no space-bar.

.model small

.stack 100h

.data

.code

main proc

mov cl,0 ; set counter to zero

label1:

mov ah,1

int 21h

cmp al,' ' ; this is single space showing ASCII code for space bar

**JE** label2 ; if spacebar is found jump to label2

inc cl ; otherwise increment counter

jmp label1 ; again another input is taken

label2:

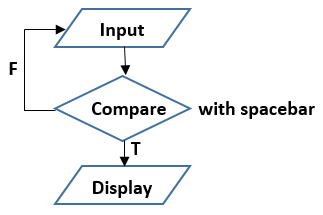
mov ah,2 ;display the count

mov dl,cl

**Program Flow Control:**

add dl,30h ; ASCII correction

int 21h

****

main endp

end main

**Code-03:** Compare two numbers and tell which one is bigger. (**JG** or **JA**)

**Hint:** Jump to a label if JG is TRUE and display the bigger number.

include ‘emu8086.inc’

.model small

.code

main proc

mov ah,1 ; input first number

int 21h

mov bl,al ; this number is saved to ‘bl’

mov ah,1 ; input second number

int 21h

CMP bl,al ; compare the first number with second

**JG** label1 ; if bl > al, goto label1

Print ‘Second no is bigger’

JMP exit

Label1:

Print ‘First no is bigger’

Exit:

Main endp

End main

**Code-04:** Write a code to find the smallest of the three numbers. (**JL** / **JB**)

**Hint:** First comparison: Value1 (V1) with Value2 (V2)

Second comparison: Any one of the smaller compared with (V3)

Include ‘emu8086.inc’

.model small

.data

v1 db ?

v2 db ?

v3 db ?

.code

main proc

mov ah,1 ; input first number v1

int 21h

mov v1,al

mov ah,1 ; input second number v2

int 21h

mov v2,al

mov ah,1 ; input third number v3

int 21h

mov v3,al

; finding the smallest number

Mov al,v1

Cmp al,v2 ; compare v1 with v2

**JL** label1 ; goto label1 if v1 < v2

Mov al,v2 ; else move v2 to AL

CMP al,v3 ; compare v2 with v3

**JL** label2 ; goto label2 if v2 < v3

Mov al,v3 ; else move v3 to AL (v3 = smallest)

JMP end

Label1:

CMP al,v3 ; compare v1 with v3

**JL** end ; goto end if v1 < v3 (v1 = smallest)

Mov al,v3 ; else (v3 = smallest)

Jmp end

Label2:

Mov al,v2 ; (v2 = smallest)

; display the smallest number

End:

Mov ah,2

Mov dl,al

Int 21h

Main endp

End main

**C. LOOPS:**

The loop instruction is the easiest way to repeat a block of statements a specific number of times. CX is automatically used as a counter and is decremented each time the loop repeats.

**Syntax** is: **loop** *destination*.

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Operands** | **Description** |
| **LOOP** | Label | Decrease CX, jump to label if CX not zero. If CX = 0, loop ends. Algorithm:   * CX = CX - 1 * if CX <> 0 then   + jump   else   * + no jump, continue   **Example:**  include 'emu8086.inc'  main proc  MOV CX, 5  label1:  PRINT 'loop!'  LOOP label1  Main endp   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |

**More Loops:**

|  |  |  |  |
| --- | --- | --- | --- |
| LOOPE | LOOPNE | LOOPNZ | LOOPZ |

**LOOPE:** Loop if Equal.

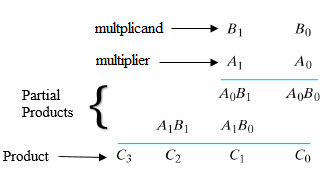
**LOOPNE:** Loop if Not Equal.

**LOOPZ:** Loop if Zero.

**LOOPNZ:** Loop if Not Zero.

**Code-05:** Write a program to multiply two positive numbers by a ‘repeated addition’ method. For example, to multiply 5**ⅹ**3, the program evaluates the product by adding 5 three times, or 5 + 5 + 5.

**Hint:** We move ‘multiplier’ to CX register to **loop** ‘Addition’ CX times.



.model small

.stack 100h

.data

.code

main proc

mov bl,5 ;multiplicand value moved to BL

mov cx,3 ;multiplier value moved to CX to loop CX times

mov al,0 ;initialize AL register to zero to perform addition

label1:

add al,bl ;BL will add to AL, CX times to produce multiplier result

**LOOP** label1

;Now we divie AL by 10 to display two digit No's separately

mov bh,10

div bh

mov bl,ah ;remainder moved to BL

mov ah,2

mov dl,al ;display quotient

add dl,30h

int 21h

mov ah,2

mov dl,bl ;display remainder

add dl,30h

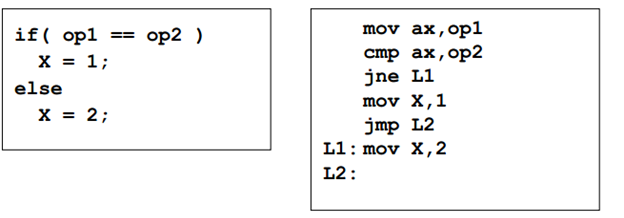
int 21h

main endp

end main

**Block-Structured IF Statements:**

* Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

****

**Compound Expression with AND:**

* When implementing the logical AND operator, consider that HLLs use short-circuit evaluation
* In the following example, if the first expression is false, the second expression is skipped:

**A picture containing logo

Description automatically generated**

**Compound Expression with OR :**

* When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
* In the following example, if the first expression is true, the second expression is skipped:

**Text

Description automatically generated with medium confidence**

**For Loop Instruction:**

* Cx used as Counter
* Automatically Decremented after every iteration
* Label is used to jump
* Ends when counter reaches zero
* Example:

**Mov cx,10 ;loop runs 10 times**

**Loop1:**

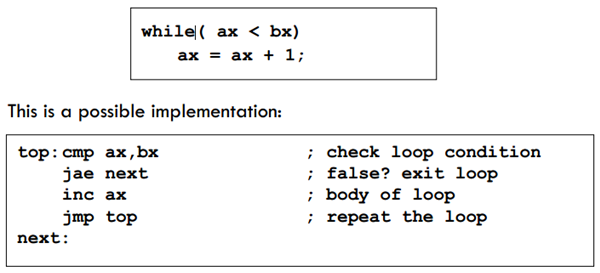
**…**

**Loop Loop1**

**WHILE Loops:**

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

**Example:**

****

## **Activity:**

* Write an Assembly program to print all the Odd numbers from 1 to 100 using loops and Compare instructions.
* Take two inputs from user and display the largest one.
* Take two inputs from user lets say x and y and implement the given c++ code.

if((x <= 4) && (y<=9))

{

For(int i=0; I < x; i++)

{

Cout<< x;

}

}

Else

{

Cout<<“0”;

}

# **LAB 4: String Operations**

**Objectives:** To learn to display, input a string, copy, search and reverse a string.

**1. Code 01: ‘Password protected application’, used to sign-in to a computer.**

Hints:

* Initialize a 4-digit password e.g. ‘NUML$’ in a db. (.data)
* Use function # 8, to hide password input on run time.
* Use function # 2, to display ‘\*’ or ‘.’ In place of password.
* Take a string at input and ‘cmp’ both strings location by location.
* Two ways of termination when a wrong password is entered.

i) Any key mismatch (terminate). ii) At the end of string input

Code:

.model small

.stack 100h

.data

prompt db 'Enter Password$'

correct db 'Correct Password$'

wrong db 'Wrong Password$'

password db 'Asim$'

.code

main proc

mov ax,@data

mov ds,ax

lea si,password

mov cx,4 ;limit cx=4 character password only

mov ah,9

LEA dx, prompt

int 21h

mov ah,2 ; new line

mov dl,0ah

int 21h

mov dl,0dh

int 21h

input:

mov ah,8 ;Function # 8 hides character input at run time

int 21h

cmp al,[si]

jne finish

mov ah,2

mov dl,'\*' ;display \* in case of password

int 21h

inc si

inc di

loop input

mov ah,2 ; new line code

mov dl,0dh

int 21h

mov dl,0ah

int 21h

mov ah,9

LEA dx, correct

int 21h

jmp terminate

finish:

mov ah,2

mov dl,’\*’

int 21h

mov dl,0dh ; new line code

int 21h

mov dl,0ah

int 21h

mov ah,9

LEA dx, wrong

int 21h

terminate:

mov ah,4ch

int 21h

main endp

end main

**2. Code 02: ‘Password Updated Code’, used to sign-in to a computer.**

.model small

.stack 100h

.data

prompt1 db 'Enter New Password$'

updated db 'Password Updated$'

password db 'aqib$'

.code

main proc

mov ax,@data

mov ds,ax

lea si,password

mov cx,4 ;limit cx=4 character password only

mov ah,9

mov dx,offset prompt1

int 21h

mov ah,2 ; new line

mov dl,0ah

int 21h

mov dl,0dh

int 21h

input:

mov ah,8 ;Function # 8 hides character input at run time

int 21h

mov [si],al

mov ah,2

mov dl,'\*' ;Display \* in case of password

int 21h

inc si

loop input

mov ah,2 ; new line code

mov dl,0dh

int 21h

mov dl,0ah

int 21h

mov ah,9

mov dx,offset updated

int 21h

mov ah,2 ; new line code

mov dl,0dh

int 21h

mov dl,0ah

int 21h

mov ah,9

mov dx,offset password

int 21h

main endp

end main

**3. Code 02: Count the ‘Capital characters’ in a defined string.**

Hints: (see flowchart below)

* E.g. the string ‘BDEFg$’
* ‘cmp’ to check if a letter falls in range of capital letters
* From A to Z (true) if ‘A = 41h to Z = 5Ah’.
* Use ‘Jge’ & ‘Jle’ if in range.

**Flowchart for Code-2:**

**Text

Description automatically generated**

**True**

Code:

.model small

.stack 100h

.data

**msg** db 'BDEFg$'

.code

main proc

mov ax,@data

mov ds,ax

lea si,**msg**

mov cx,5 ;length of string except $

mov bl,0 ;set character count to zero

inrange: ;check if character falls in-ragne

mov al,[si]

inc si ;next character in string

cmp al,41h ;code for 'A'

jl inrange ;if character is < 41h

cmp al,5ah ;value for 'Z'

jg end ;end if value is greater than Z, no increment

inc bl ;both conditions Ok the increment count

end:

loop inrange

mov ah,2 ;new line

mov dl,bl

add dl,30h

int 21h

mov ah,4ch

int 21h

main endp

end main

## **Activity:**

**See ‘String Search’ Code in Lab-4.**

**4. Code 03: Search and Replace the characters in a string.**

Program Flow: Read and compare each character in string with Input value.

If the value to be changed is found in string. Ask for new value. Display it.

Code:

.model small

.data

string db 'house$' ;we want to replace 'u' with 'r'

msg1 db 'Enter value to be searched: $'

msg2 db 'Enter its new value: $'

msg3 db 'Updated Sring is: $'

.code

main proc

mov ax,@data

mov ds,ax

lea si,string

mov cx,5

mov ah,9

lea dx,msg1

int 21h

mov ah,1 ;enter value to be searched in a string

int 21h

label1:

cmp [si],al ;search for the entered value

je update ;if found go to label to update it

inc si ;otherwise increment SI and keep on searching

loop label1

update:

mov ah,9

lea dx,msg2

int 21h

mov ah,1 ;user will enter the new value to update it

int 21h

mov [si],al ;value updated

mov ah,9

lea dx,msg3

int 21h

mov ah,9

mov dx,offset string

int 21h

main endp

end main

1. **Code 05: Count the ‘Even’ numbers in an array.**

.model small

.data

array db 23h, 78h, 89h, 94h, 12h

.code

main proc

mov ax,@data

mov ds,ax

lea si,array

mov bl,2

mov ax,0

mov bh,0 ;counter

mov cx,5 ;elements

label:

mov al,[si]

div bl

inc si

cmp ah,0 ;remainder is checked

je label1

jmp lab

label1:

inc bh ;count even no

lab:

loop label

exit:

mov ah,2

mov dl,bh

add dl,30h

int 21h

main endp

end main

**Code 06: Count the ‘Even’ numbers in an entered string.**

Program Flow: User will enter digits till Enter key is pressed.

Each digit will be divided by 2 to find if remainder is zero. Display count.

Code:

.model small

.stack 100h

.code

main proc

mov cl,0 ;initialize even digits count to zero

mov bl,2 ;dividing a number by 2 reveals if its even or not

even:

mov ah,1 ;user will enter the digits

int 21h

cmp al,0dh ;enter key compare

je display

div bl

cmp ah,00h ;check remainder if it is zero, hence even number

je counter

jmp even

counter:

inc cl ;increment even digits counter value

jmp even

display:

mov ah,2 ; new line code

mov dl,0ah

int 21h

mov dl,0dh

int 21h

mov ah,2

mov dl,cl ;display the value of the count

add dl,30h

int 21h

main endp

end main

# **LAB 5: Assembly Application Programs**

**Objectives:** To learn to write the following application codes:

* + Password protected application, used to sign-in to a computer.
  + Count the capital characters in a defined string.
  + Search and Replace a character in a string.
  + Count the ‘Even’ numbers in an entered string.

. **String commands in Assembly**

* Int 21h has function # 9 to ‘display strings’. Example: mov ah,9; int 21h
* The string must ‘end’ with a $ character e.g. ‘Assembly$’
* DX has the ‘offset-address’ of string.
* The int 21h, function # 9, expects the ‘offset-address’ of string to be in the DX register. To get it we use Lea command.
* Lea (Load effective address) command, puts the source offset-address contained in DX into destination.
* Lea destination, source command adds ‘index-address + base address’ for complete address /offset of string.
* The hexadecimal bytes 0dh (carriage return) and 0ah (new line), are called end-of-line.

Example-1: Using pointer to access an array.

.model small

.data

arrayB db 20h, 30h

.code

Main proc

Mov ax,@data

Mov ds,ax

Lea si, arrayB

Mov al,[si]

Inc si

Mov al,[si] ; inc si ; mov al,40h ; mov [si],al

Main endp

End main

Example-2: Using pointer to access an array. Different data sizes.

.model small

.data

arrayA dw 1234h

db ‘A’

.code

Main proc

Mov ax,@data

Mov ds,ax

Lea si,arrayA

Mov ax,[si]

Add si,2

Mov bl,[si] ; A = 41h

Main endp

End main

1. String Display: To display a string at runtime.

.model small

.stack 100h

.data

msg db 'I am a student$'; Format: label db ‘values$’ (db = define byte)

.code

main proc

mov ax,@data ;to display string

mov ds,ax ;2 lines

lea si,msg

mov cx,14 ;14 characters to display

mov ah,2

l1:

mov dl,[si]

int 21h

inc si

loop l1 ;to display 14 characters via loop

main endp

end main

Code-02:

.model small

.stack 100h

.data ;data segment (DS)

message db ‘Hello World$’

.code

Main proc ;next two lines initialize data segment to starting address

Mov ax,@data ;these two lines are used to access variables under DS

Mov ds,ax ;bcoz we can’t move a ‘constant’ directly to DS

;two commands translate the name @Data into address & move to DS

Mov ah,9

Mov dx,offset message ; LEA DX,message

Int 21h

Mov ah,4ch ;terminate program and return control to DOS

Int 21h

Main endp

End main

2. String Input: To input a string at prompt, and display a string. If enter key pressed (0dh), then terminate program execution. (see Flowchart next)

.model small

.stack 100h

.data

a db '?' ;’?’symbol for variables that are not initialized

.code

main proc

mov ax,@data ;these two lines initialize data segment

mov ds,ax

lea si,a ; string 'a' stored in index location for SI pointer

l1:

mov ah,1

int 21h

cmp al,0dh ; compare with enter key

je l2 ; conditional jump to display stored string

mov [si],al ;[square brackets] return/store value at memory address.

inc si

jmp l1 ;unconditional jump for new character input

l2:

inc si

mov [si],'$' ; last character to add in string

mov ah,2

mov dl,0dh ;generate new line

int 21h

mov dl,0ah

int 21h

mov ah,9 ; display the string

mov dx,offset a ;function 9 expects offset address of string to be in DX

int 21h

mov ah,4ch ;return control to DOS

int 21h

main endp

end main

Flowchart for ‘String Input’:

**Input**

**Save**

**Compare**

**False**

**True**

**Display**

3. String Reversal: To reverse a string and display it in reverse order.

(See Flowchart)

.model small

.stack 100h

.data

array db 'abcde$'

.code

main proc

mov ax,@data ;these two lines initialize data segment

mov ds,ax

lea si,array ;to access each character in array

mov cx,0 ;initialize counter with zero

label1:

mov al,[si] ;square brackets return value and not address

cmp al,'$' ;if $ is not found then jump to label2

jne label2

jmp rev ;if $ found then jump to rev

label2: ;run this loop until $ sign achieved

inc cx ;increment loop value for each character

inc si ;to get next character

jmp label1 ;continue to run the loop till last character $

rev: ;$ sign has been found

dec si ;so as not to print last character of $

rev1: ;label for displaying in reverse

mov ah,2

mov dl,[si]

int 21h

dec si ;we are going backwards

loop rev1 ;runs the loop cx times

mov ah,4ch ;return control to DOS

int 21h

main endp

end main

Flowchart for ‘String Reversal’:

**False**

**True**

**Display**

**Rev**

**Cmp ‘$’**

4. String Copy: To copy a source string to destination string. (Flowchart)

Here we have used Si (Source) and Di (Destination) due to two strings.

DS has ‘base address’ of ‘string1’, so: DS+Si = source location

ES has ‘base address’ of ‘string 2’, so ES+Di = destination location

Movsb does two functions: Si++ and Di++ ,it copies + inc characters

Cld ; Clear Direction Flag, if DF = 0, pointers increment.

.model small

.stack 100h

.data

st1 db 'hello$' ;source string to be copied

st2 db '?' ;destination of copy

.code

main proc

mov ax,@data

mov ds,ax

mov es,ax

lea si,st1 ;these two lines copy 'si' to 'di' till $ sign

lea di,st2

cld ;clear direction flag so that string pointers auto increment

mov cx,6 ;6 character string

rep movsb ;move string byte using 'repeat' command

mov ah,9 ;string display

mov dx,offset st2 ; lea dx,st2

int 21h

mov ah,4ch

int 21h

main endp

end main

5. String Rev: To input a string at prompt, and display in reverse order.

.model small

.stack 100h

.data

array db '?'

.code

main proc

mov ax,@data ;these two lines initialize data segment

mov ds,ax

lea si,array ;to access each character in array

mov cx,0 ;initialize counter with zero

l1:

mov ah,1

int 21h

cmp al,0dh ; compare with enter key

je label1 ; conditional jump to display stored string

mov [si],al ;[square brackets] return/store value at memory address.

inc si

inc cx

jmp l1 ;unconditional jump for new character input

inc si

mov [si],'$'

label1:

mov ah,2 ;new line generation

mov dl,0dh

int 21h

mov dl,0ah

int 21h

dec si ;so as not to print last character of $

rev1: ;label for displaying in reverse

mov ah,2

mov dl,[si]

int 21h

dec si ;we are going backwards

loop rev1 ;runs the loop cx times

mov ah,4ch ;return control to DOS

int 21h

main endp

end main

**String Searching:** Based on character match. Good

.model small

.data

justice db 'justice for him$'

yes db 'Charcter Found$'

no db 'charcter not found$'

new db ‘Enter new character$’

.code

main proc

mov ax,@data

mov ds,ax

lea si,justice

mov cx,16

mov ah,1

int 21h

l1:

cmp [si],al

je l2

inc si

loop l1

jmp l3

l2:

mov ah,9

lea dx,yes

int 21h

mov ah,9

lea dx,new

int 21h

mov ah,1

int 21h

mov [si],al

jmp exit

l3:

mov ah,9

lea dx,no

int 21h

exit:

mov ah,9

lea dx,justice

int 21h

mov ah,4ch

int 21h

main endp

end main

**;Utilization of variables and printing string**

.model small

.stack 100h

.data

myVar1 db '1'

myVar2 db ?

myVar3 db "Hello World$"

.code

main proc

mov ax,@data ;this command is reponsible to link

;data memory to code

mov ds,ax ;linking variable so that they are

;directly available in the code

mov ah,2

mov dl,myVar1

int 21h

mov ah,2 ;

mov dl,0ah ;Print on new line

int 21h ;

mov dl,0dh ;Carage Return

int 21h ;

mov myVar2, 'T' ;Initialing the myVar2

mov ah,2

mov dl,myVar2

int 21h

mov ah,2

mov dl,0ah ;Print on new line

int 21h ;

mov dl,0dh ;Carage Return

int 21h ;

mov ah,9 ;Sub-routine for printing Strings

lea dx,myVar3 ;Load Effective Address so that we don't have to use offset

int 21h

mov ah,4ch

int 21h

main endp

end main

## **Activity :**

* Perform the multiplication on two numbers. Display Proper messages to user.
* Note: use strings to display messages
* Perform addition on two numbers. The result can either me of single digit or more.
* Note: use strings to display messages

# **LAB 6: Logical Operations**

**Objectives:** To learn the basic ‘logic commands’ and their use.

|  |  |  |
| --- | --- | --- |
| **Command** | **Addressing Format** | **Description** |

|  |  |  |
| --- | --- | --- |
| **AND** | REG, memory memory, REG REG, REG memory, immediate REG, immediate | **Logical AND between all bits of two operands.** Result is stored in operand1. **Example:**  mov al,'a' ;small to capital letter code  AND al,11011111b ;=223d, A=61h  mov ah,2  mov dl,al ;ASCII for 'A' is 41h  int 21h |
| **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. **Example:**  mov al,'A' ;capital to small letter code  OR al,00100000b ;=32d, a=41h  mov ah,2  mov dl,al ;ASCII for 'a' is 97d=61h  int 21h |
| **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. **Example:**  mov al,00000111b ;different bits give Logic-1  XOR al, 00000010b  mov ah,2  mov dl,al ;AL = 00000101b = 5d  add dl,30h  int 21h |

## 

## **Activity:**

**Code: Display a capital letters string into small letters.**

.model small

.data

msg db 'master$'

.code

main proc

mov ax,@data

mov ds,ax

lea si,msg

mov cx,6

label:

mov al,[si]

AND al,11011111b

inc si

mov ah,2

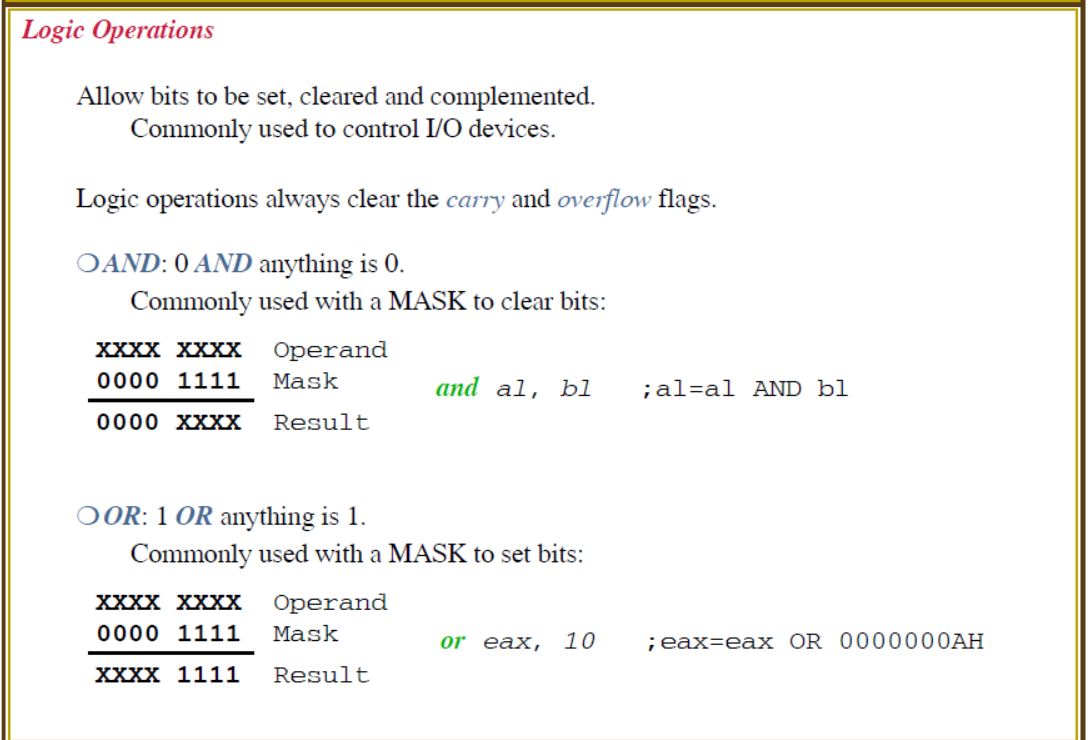
mov dl,al

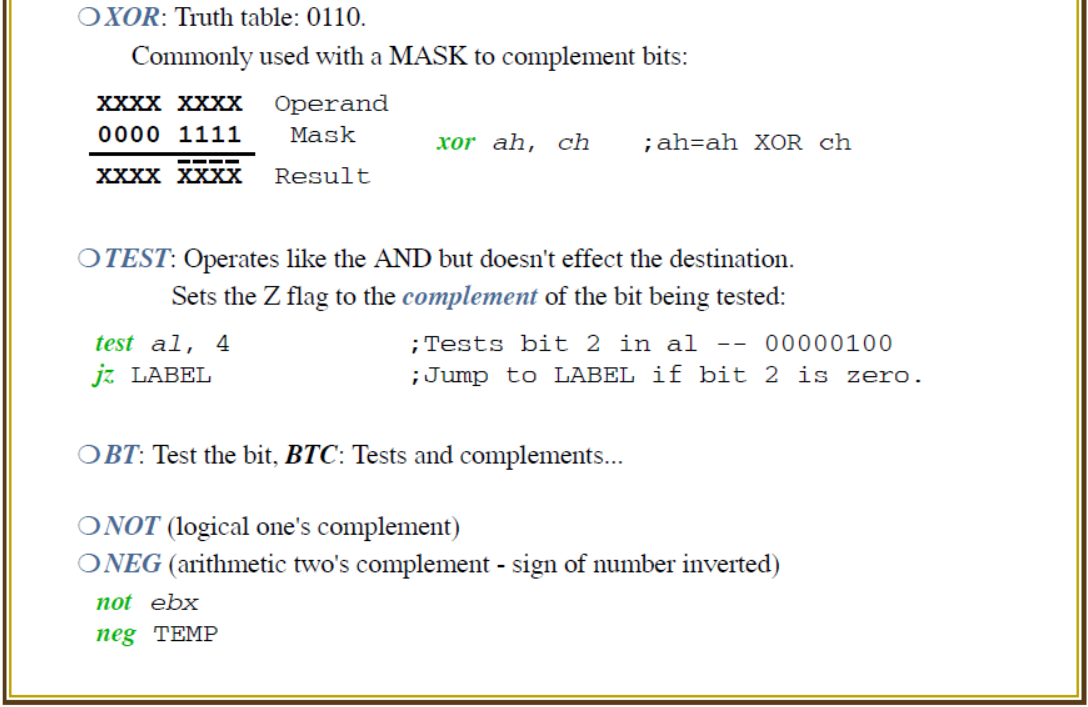
int 21h

loop label

main endp

end main

****

****

# **LAB 7: Shift and Rotate Operations**

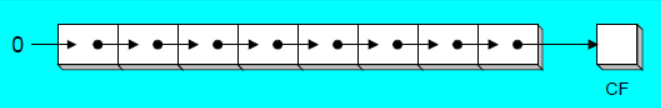
**Objectives:** To learn the basic ‘shift and rotate’ instructions and their use.

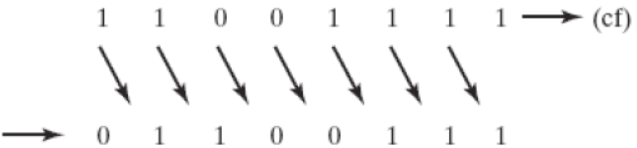
**Shift Instructions**

The 8086 can perform two types of Shift operations, the *logical shift* and the *arithmetic shift*. There are four shift operations (SHL, SRL, SAL, SAR).

**Logical Shift:** Fills the newly created bit position with 0 (zero):

**SHR**

****

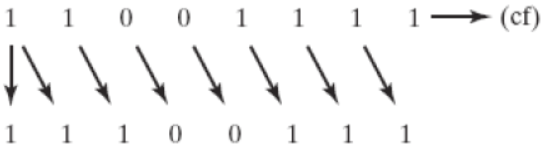
****

**Arithmetic Shift:** Right shift fills the newly created bit positions with a copy of the number’s sign bit (MSB):

**SAR**

Chart, box and whisker chart

Description automatically generated



‘Shifting Left’ by 1-bit **multiplies** a number by 2, and shifting left **n**-bits multiplies the operand by **2n**.

‘Shifting Right’ by 1-bit **divides** a number by 2, and shifting right by **n**-bits divides the operand by **2n**.

|  |  |  |
| --- | --- | --- |
| **Command** | **Meaning** | **Format** |
| **SAL** | Shift Arithmetic Left | SAL D, count |
| **SAR** | Shift Arithmetic Right | SAR D, count |
| **SHL** | Shift Logical Left | SHL D, count |
| **SHR** | Shift Logical Right | SHR D, count |

**Note:** Count can be an ‘immediate value’ or may be contained in ‘**CL**’ register.

If the source operand is specified as CL instead of a number, then the count in this register represents the bit positions the contents of the source operand are to be shifted. This permits the count to be defined under software control and allows a range of shifts from 1 to 255 bits.

|  |  |  |
| --- | --- | --- |
| **Command** | **Addressing Format** | **Description** |

|  |  |  |
| --- | --- | --- |
| **SHL** | REG, immediate REG, CL memory, immediate memory, CL | **Shift Left instruction performs a ‘Logical Left Shift’ on the destination operand, filling the lowest bit with 0(zero).**  Shifting Left 1-bit multiplies a number by 2.  **Example:**  mov al,2  **shl** al,2 **;** al = 2 \* 2**2** = 2 \* 4 = 8 ; shift left 2-bits.  **Note: SAL** (Shift Arith. Left) is identical to **SHL**. |
| **SHR** | REG, immediate REG, CL memory, immediate memory, CL | **It performs a ‘Logical Right Shift’ on the destination operand.** **The highest bit positon is filled with a 0 (zero).** See ‘SHR’ by 1-bit divides a number by 2. Fig  Shifting Right 1-bit divides a number by 2.  **Example:**  mov al,8  **shr** al,2 **;** al = 8 / 2**2**= 8 / 4 = 2 ; shift right 2-bits. |
| **SAR** | REG, immediate REG, CL memory, immediate memory, CL | **It performs a ‘Right Arithmetic Shift’ on the destination operand.** See ‘SAR’ Figure.  **Note:** An ‘Arithmetic Shift’ **preserves the no’s sign**.  **Example:**  mov al, -8  **sar** al,1 ; al = -4  mov cl,2  **sar** al,2 ; al = -1 |

**Multiplication and Division using Shift Instructions:**

The **SHL** instruction can be used to **multiply** an operand by 2**n** and the **SHR** instruction can be used to **divide** an operand by 2**n**.

The **MUL** and **DIV** instructions take much longer to execute than the ‘shift’ instructions. Therefore, when multiplying/dividing an operand by a small number, it is better to use ‘Shift’ instruction than to use the MUL/DIV instructions. For example, ‘MUL BL’ when BL=2, takes many more clock cycles than ‘SHL BL,1’.

**Code-01: Write a program to multiply AX=2 by 7 using only Shift and ADD instruction. You should not use the MUL instruction.**

**Hint:** Recall that shifting left n-bit multiplies the operand by 2**n**.

If the multiplier is not an absolute power of 2, then express the multiplier as a sum of terms which are absolute power of 2.

For example, to multiply AX by 7. (**7** = 4 + 2 + 1 = 2**2** + 2**1** + 1).

Answer = AX shifted left by 2 + AX shifted left by 1 + AX.

**Code:**

.model small

.stack 100h

.data

.code

Main proc

Mov AX,2 ; operand to be multiplied

Mov BX,AX

Mov CX, AX

SHL BX,2 ; AX\*4 = 2 \* 4 = 8

SHL CX,1 ; AX\*2 = 2 \* 2 = 4

Add BX,CX ; 4\*AX + 2\*AX = 6\*AX = 8 + 4 = 12

Add AX,BX ; 6\*AX + 1\*AX = **7**\*AX = 12 + 2 = 14

Main endp

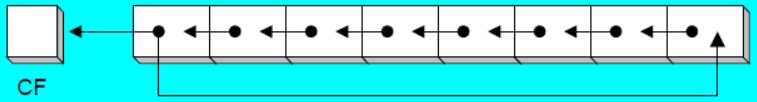
End main

**Code: To multiply AX by 36. Use the Hint 36 = 32 + 4. Also 26 = 16+8+2.**

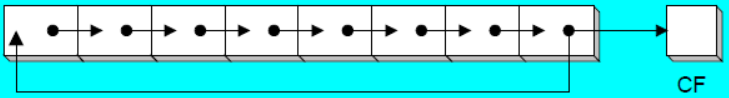
**Rotate Instructions**

The 8086 can perform two types of rotate instructions; the *rotate without carry* and the *rotate through carry*. There are four rotate operations (ROL, ROR, RCL and RCR). **Note:** In a ‘Rotate operation’, NO bits are lost.

**Rotate without Carry:** Shifts each bit to the left or the right. The highest/lowest bit is copied into both the carry flag and the highest/lowest bit.

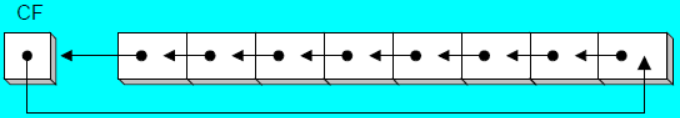
****

**ROL**

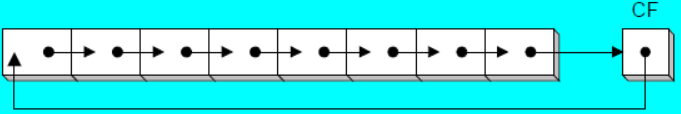
****

**ROR**

**Rotate with Carry:** Shifts each bit to the left or right. The carry flag bit (CF) is inserted into left/right bit position based on the respective operation.



**RCL**



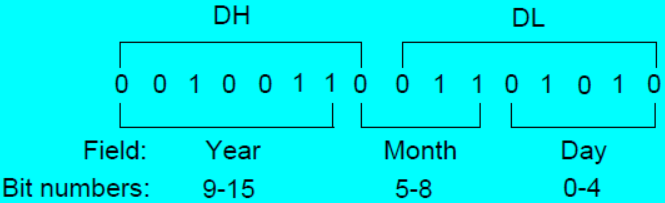
**RCR**

|  |  |  |
| --- | --- | --- |
| **ROL** | REG, immediate REG, CL memory, immediate memory, CL | ROL shifts each bit to the left. The highest bit (MSB) is copied into both the Carry Flag (CF) and into the lowest bit (LSB). No bits are lost. See ‘ROL’ Fig.  **Example:**  mov al, 11110000b  **rol** al,1 ; al = 1110000**1**, CF = 1. |
| **ROR** | REG, immediate REG, CL memory, immediate memory, CL | ROR shifts each bit to the right. The lowest bit (LSB) is copied into both the Carry Flag (CF) and into the highest bit (MSB). No bits are lost. See ‘ROR’ Fig. **Example:**  mov al, 11110000b  **ror** al,1 ; al = **0**1111000, CF = 0. |
| **RCL** | REG, immediate REG, CL memory, immediate memory, CL | RCL (Rotate Carry Left) shifts each bit to the left. Copies the Carry Flag (CF) to the Least Significant Bit (LSB). Copies the Most Significant Bit (MSB) to the Carry Flag (CF). See ‘RCL’ Figure.  **Example:**  **clc** ; clear carry flag, CF = 0  mov al, 88h ; CF,AL = 0 **1**0001000b  **rcl** al,1 ; CF,AL = **1** 0001000**0**b |
| **RCR** | REG, immediate REG, CL memory, immediate memory, CL | RCR (Rotate Carry Right) shifts each bit to the right. Copies the Carry Flag (CF) to the Most Significant Bit (MSB). Copies the Least Significant Bit (LSB) to the Carry Flag (CF). See ‘RCR’ Figure.  **Example:**  **stc** ; set carry flag, CF = 1  mov al, 10h ; CF,AL = **1** 0001000**0**b  **rcr** al,1 ; CF,AL = **0** **1**0001000b |

## 

## **Activity:**

**Code-02: The MS-DOS file Date field packs the ‘year, month, day’ into 16-bits. Isolate the ‘month’ field from the bit stream given below. [2Ah]**

****

**Code:** mov ax,dx ; make a copy of **DX = 0010\_0110\_0110\_1010b**

**Shr** ax,5 ; shift right 5 bits

And **al**,**0000**1111b ; clear bits 4 to 7

mov month,al ; save in month. **Note:** same to hours**:**min**:**sec. **[2Ch]**

# **LAB 8: Defining and Using Procedures**

**Objectives:** To learn how to make procedures and perform procedure calls.

**Procedures** also known as *subroutines* or *functions*is a named block of statements that ends in a ‘return’ statement. ‘Procedures’ allow large problems to be divided into smaller parts that can be called more than once. This will make the program more manageable. A procedure in Assembly is equivalent to a Java or C++ function. CPU uses the runtime **stack** to track the location of procedures. **Main proc** is the main procedure that ends with **main endp**.

**Defining a Procedure using PROC directive:**

A ‘Procedure’ is declared using the **PROC** and **ENDP** directives. A procedure must be assigned a name (a valid identifier). When you create a procedure, end it with a **RET** (return) instruction. **RET** forces the CPU to return to the location from where the procedure was called. Following is the example of a procedure named ‘Sample’.

**Text

Description automatically generated with medium confidence**

**Labels in Procedures:**

By default, **labels** for JMP and LOOP instructions are visible only within the procedure in which they are declared. For example the **label** named destination must be located in the same procedure as the JMP instruction: **JMP** destination

Because procedures have an automated way of returning and adjusting the runtime stack. So it is not a good idea to jump or loop outside the current procedure. Otherwise the runtime stack can easily become corrupted.

**What do Procedures take and return?**

Each procedure has a list of input parameters e.g. from registers or memory.

Each procedure returns a value stored in a register specified by the procedure.

A procedure may have a list of any special requirements called **pre-conditions**, that must be satisfied before the procedure is called e.g. passing values.

**Example:** Sum of three integers procedure.

Comments: The procedure named **SumOf** calculates the sum of three 16-bit integers. We will assume that relevant integers are assigned to AX, BX and CX before the procedure is called. The procedure returns the sum in AX:

**SumOf** PROC

Add ax, bx

Add ax,cx

Ret

**SumOf** ENDP

**CALL and RET Instructions:**

The **CALL** instruction calls a procedure and does the following:

* It pushes offset of next instruction (return address) on the stack.
* Copies the address of the called procedure into IP (instruction pointer).

The **RET** instruction returns from a procedure and:

* Pops top of the stack (return address) into IP.

**Passing Register Arguments to Procedures:**

A good procedure might be useful in many different programs. But it could only be used only once if it refers to specific variable names.

Passing *arguments (input parameters)*can make a procedure flexible, because parameter values can change at runtime.

Place all ‘parameters’ in a **main** accessible storage area, then call the procedure

Two common methods of ‘parameter passing’ are:

* General Purpose Registers (AX, BX, CX, DX).
* Stack (memory) method. (Push and Pop)

**Code-01:** Calling the **SumOf** three integers procedure.

.model small

.stack 100h

.data

**theSum** dw ?

.code

Main Proc

Mov ax,@data

Mov ds,ax

Mov ax,1000h ; argument

Mov bx,2000h ; argument

Mov cx,3000h ; argument

**Call SumOf** ; AX = AX + BX + CX

Mov theSum,ax

Mov ah,4ch

Int 21h

Main endp

**SumOf** PROC

Add ax, bx

Add ax,cx

Ret

SumOf ENDP

**Code-02:** A procedure to generate a new line.

.model small

.stack 100h

.data

.code

Main proc

Mov ah,2

Mov dl,’A’

Int 21h

**Call NewLine**

Mov ah,2

Mov dl,’B’

Int 21h

Mov ah,4ch ; terminate program

Int 21h

Main endp

**NewLine** proc

Mov ah,2

Mov dl,0dh

Int 21h

Mov dl,0ah

Int 21h

Ret

NewLine endp

**Code-03:** A procedure to display the two-digit product of two integers.

.model small

.stack 100h

.data

.code

Main proc

Mov al,8

Mov bl,2

Mul bl ; AX = AL \* BL = 8 \* 2 = 16

**Call Display**

Mov ah,4ch

Int 21h

Main endp

**Display** proc

Mov cl,10

Div cl

Mov ch,ah ; remainder moved to CH to avoid loss of contents

Mov ah,2

Mov dl,al ; display quotient first

Add dl,30h

Int 21h

Mov ah,2

Mov dl,ch ; display remainder second

Add dl,30h

Int 21h

Ret

Display endp

**Activity:** To do list.

* Write a Procedure which when called, prints time on Console. (Hint given below)

|  |  |  |
| --- | --- | --- |
| AH = **2Ch**; Get system time  AH = **2Ah**; Get system date | Get **System Time**:  Mov ah,2ch  Int 21h  CH = 0-23 hour; CL = 0-59 minutes; DH = 0-59 second DL = 1/100 seconds (0-99) | Get **System Date**:  Mov ah,2ah  Int 21h  DL = date; DH = month;  AL = day of week;  CX = current year |

* Write a Procedure which takes 5 numbers as argument. Calculate sum of numbers and return the sum. Prints the sum in main procedure. You have to use Stack for Passing and Retrieving Arguments.
* Write a procedure which displays a string in reverse order.

Hint: rev1:

Call display

loop rev1

**Code:**

.model small

.data

str db 'Master$'

.code

main proc

mov ax,@data

mov ds,ax

lea si,str

mov cx,9

read:

mov al,[si]

inc si

cmp al,'$'

je rev1

jne read

rev1:

call rev

loop rev1

mov ah,4ch

int 21h

**rev** proc

mov ah,2

mov dl,[si]

int 21h

dec si

ret

rev endp

# **LAB 9: Stack Operations**

**Objectives:** To Learn about runtime stack in Assembly.

Stack is an area of memory for keeping temporary data.

**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**).

**To use stack, stack segment must be defined in program.**

**.stack** size

If size is 100d, it reserves 100 bytes for stack.

If size is 100h, it reserves 256 bytes for stack.

Example: .stack 100h

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.

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.

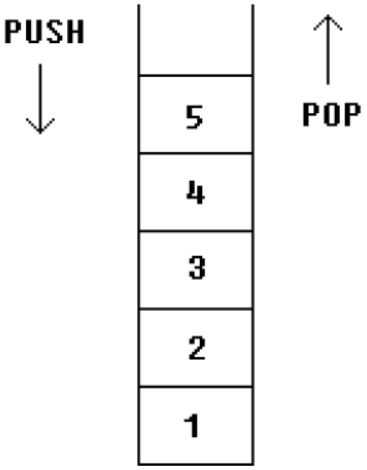
**Stack Segment (SS):**

* Specifies the segment address of Stack.
* Added to address during stack access.

**Stack Pointer (SP):**

* Stack Pointer Register (SP) points the top of Stack.
* Suppose the Stack size is set to 256bytes.
  + Stack Pointer will be Initialized with Value 256.
  + Depending on Architecture, PUSH and POP Operations will change Value of Stack Pointer.

|  |  |  |
| --- | --- | --- |
| **Pop** | REG  SREG  memory | **Get 16-bit value from the stack.**  **Algorithm:** Adds 2 to SP register.  l operand = SS:[SP] (top of the stack)  l SP = SP + 2  **Example:**  MOV AX, 1234h  PUSH AX  POP DX ; DX = 1234h  RET |
| **Push** | REG  SREG  memory  immediate | **Store 16-bit value in the stack.**  **Algorithm:** Subtracts 2 from SP register.  l SP = SP - 2  l SS:[SP] (top of the stack) = operand  **Example:**  MOV AX, 1234h  PUSH AX  POP DX ; DX = 1234h  RET |



**Example:** Push and Pop values from stack word sized and byte sized.

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.

**Example:** Another use of stack is for exchanging the values.

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.

## **Activity:**

* Initialize an Array of 5 Indexes.
  + Declare main procedure
  + Initializer list having 0,2,4,8,10 on respective indexes
  + Push values of Array in a Stack
  + Pop & Print each value on Console.
  + Beware to deal with n-digit number while printing.
* Write a Procedure which when called, prints time on Console. (Review Lab # 02, Slide # 25)
* Write a Procedure which takes 5 numbers as argument. Calculate sum of numbers and return the sum. Prints the sum in main procedure. You have to use Stack for Passing and Retrieving Arguments.

# **LAB 10: File Handling in Assembly**

**Objectives:** To learn how to deal with files in following ways:

* + Opening File
  + Reading File
  + Detecting Next line in File
  + Counting Characters in File
  + Writing to File
  + Appending File
  + Closing File

**File Opening:**

* In EMU8086, create a file “myfile.txt” with some text in directory “C:\emu8086\MyBuild”
* Under data directive, define a string which contains file name and append 0 at end, the last character to write to a file must be a zero.
  + file db “myfile.txt”,0
* Define a buffer in which file contents will be stored after reading, in data segm.
  + buffer db 5000 dup("$")

**Loading File Handler:**

* First step is to Load File handler
  + File handler acts as a pointer to file

mov dx, offset file ; Load address of String “file”

mov al, 0 ; Open file (read-only)

mov ah, 3dh ; Load File Handler and store in ax, function number

int 21h

Reading file:

mov bx,ax ; Move file Handler to bx

mov dx, offset buffer ; Load address of string in which file contents will be stored after reading

mov ah, 3fh ; Interrupt to read file, function number

int 21h ; Read file and store the contents in string whose address is stored in dx

mov ah,9 ; to display the contents of the file

int 21h

**Printing Contents of File:**

* All the contents of file are stored in string “Buffer”.
* Now print this String using int21h function 09h interrupt.

**Detecting Next Line in File:**

* Contents of file is stored in buffer string.
* Next line will be indicated by “0DH” and “0AH”.
* 46h, 41h, 53h, 54h, 0Dh, 0Ah

**Writing to file:**

* First step is to Load File handler
* msg db “Happy$”
  + mov dx, offset file ; Load address of String “file”
  + mov al, 2 ; Open file (read/write)
  + mov ah, 3dh ; Load File Handler and store in ax
  + int 21h

mov cx, 5 ; Number of bytes to write

mov bx,ax ; Move file Handler to bx

mov dx, offset msg ; Load offset of string which is to be written to file

mov ah, 40h ; Write to file, function number

int 21h

**Appending File:**

* Move file pointer to end of file before writing to file

mov cx,0

mov ah, 42h ; Move file pointer

mov al, 02h ; End of File

int 21h

**Closing file:**

mov ah, 3eh ; close the file, function number

int 21h

## **Activity:**

* Write assembly program, which opens a file and print its contents on CONSOLE and closes the file.
* Write Assembly program which writes a String to a file. Define String in program.
* Write Assembly program which appends a String to a file.
* Write Assembly program which takes 5 numbers input from user and write to file.

**Check this file handling code:**

.model small

.stack 100h

.data

file2 db "file.txt",0

buffer db 5000 dup("$")

msg db "Hello World","$"

count db 0

.code

mov ax, @data

mov ds, ax

mov dx, offset file2

mov al, 2

mov ah, 3dh ; Open File Handler

int 21h

mov cx,10

mov ah,40h

int 21h

mov bx,ax ; Move file Handler to bxs

mov dx, offset buffer

mov ah, 3fh

int 21h ; Read file and store the contents in address stored in dx

mov dx, offset buffer

mov ah,09

int 21h

mov ah, 4ch

int 21h

end

**Duplicate code for Reading a file.**

.model small

.stack 100h

.data

file db "input.txt",0

buffer db 5000 dup("$")

.code

mov ax,@data

mov ds,ax

;open file in read only mode

;-----------------------------

mov dx,offset file ;load address of strig

mov al,0 ;open in read only mode

mov ah,3dh ;load handler in ax

int 21h

;-----------------------------

mov bx,ax ;load handler in bx

;-----------------------------

mov dx,offset buffer ;load address of string to store

mov ah,3fh ;function to read file

int 21h

;-----------------------------

mov dx,offset buffer ;load address of string

mov ah,09h ;write string on console

int 21h

;-----------------------------

mov ah,4ch

int 21h

end

**Duplicate Code for writing to a file.**

.model small

.stack 100h

.data

file db "output.txt",0

msg db "Fast Nuces$"

.code

mov ax,@data

mov ds,ax

;open file in read and write mode

;-----------------------------

mov dx,offset file ;load address of strig

mov al,2 ;open in read and write mode

mov ah,3dh ;load handler in ax

int 21h

;-----------------------------

mov bx,ax ;load handler in bx

;-----------------------------

mov cx,5 ;how many bytes to write

mov dx,offset buffer ;load address of string to write

mov ah,40h ;function to write file

int 21h

;-----------------------------

;mov dx,offset msg ;load address of string

;mov ah,09h ;write string on console

;int 21h

;-----------------------------

mov ah,4ch

int 21h

end

# **LAB 11: BIOS Level Programming**

**Objectives:** To Learn about Keyboard and Mouse control at BIOS level

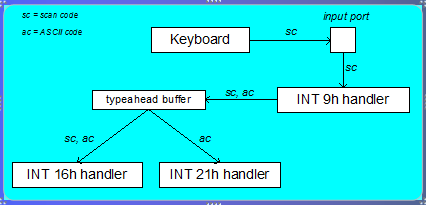
* BIOS interrupts
* Keyboard Input with INT 16h
* Scan code and ASCII code

**PC-BIOS:**

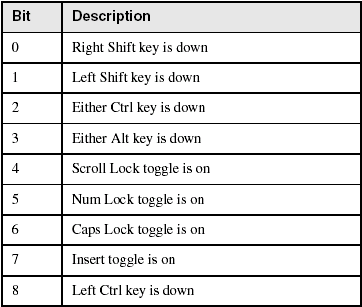
* The BIOS (Basic Input-Output System) provides low-level hardware drivers for the operating system.
  + accessible to 16-bit applications
  + written in assembly language, of course
  + source code published by IBM in early 1980's
* Advantages over MS-DOS:
  + permits graphics and color programming
  + faster I/O speeds
  + read mouse, serial port, parallel port
  + low-level disk access

**Keyboard Input with INT 16h:**

* How the Keyboard Works
* INT 16h Functions
* Keystroke sends a scan code to the keyboard serial input port
* Interrupt triggered: INT 9h service routine executes
* Scan code and ASCII code inserted into keyboard typeahead buffer



**Keyboard Flags:**

Table

Description automatically generated

**INT 16h Functions:**

* Provide low-level access to the keyboard, more so than MS-DOS.
* Input-output cannot be redirected at the command prompt.
* Function number is always in the AH register
* Important functions:
  + set typematic rate
  + push key into buffer
  + wait for key
  + check keyboard buffer
  + get keyboard flags

**Function 10h: Wait for Key:**

If a key is waiting in the buffer, the function returns it immediately. If no key is waiting, the program pauses (blocks), waiting for user input.

* .data
* scanCode BYTE ?
* ASCIICode BYTE ?
* .code
* mov ah,10h
* int 16h
* mov scanCode,ah
* mov ASCIICode,al

Mouse handling in Assembly.

* MS-DOS functions for reading the mouse
* INT 33h functions
* Mouse Tracking Program Example

**Reset Mouse and Get Status:**

* INT 33h, AX = 0
* Example:
  + mov ax,0
  + int 33h
  + cmp ax,0
  + je MouseNotAvailable
  + mov numberOfButtons,bx

**Show/Hide Mouse:**

* INT 33h, AX = 1 (show), AX = 2 (hide)
* Example:
  + mov ax,1 ; show
  + int 33h
  + mov ax,2 ; hide
  + int 33h

**Get Mouse Position & Status:**

* INT 33h, AX = 4
* Example:
  + mov ax,4
  + mov cx,200 ; X-position
  + mov dx,100 ; Y-position
  + int 33h

**Get Button Press Information:**

* INT 33h, AX = 5
* Example:
  + mov ax,5
  + mov bx,0 ; button ID
  + int 33h
  + test ax,1 ; left button down?
  + jz skip ; no - skip
  + mov X\_coord,cx ; yes: save coordinates
  + mov Y\_coord,dx

**Other Mouse Functions:**

* AX = 6: Get Button Release Information
* AX = 7: Set Horizontal Limits
* AX = 8: Set Vertical Limits

## **Activity:**

* Get the mouse button press information, whether the mouse click was left, right or middle.
* Write a key logger using BIOS level interrupts, which will keep the record of all the keys pressed on keyboard.
* Note: Use Scan and ASCII codes for the activity.

# **LAB 12: Graphics in Assembly**

**Objectives:** To Learn to use Graphics in Assembly.

* Video interrupt
* Interrupt functions
* Video modes
* Key board interrupt

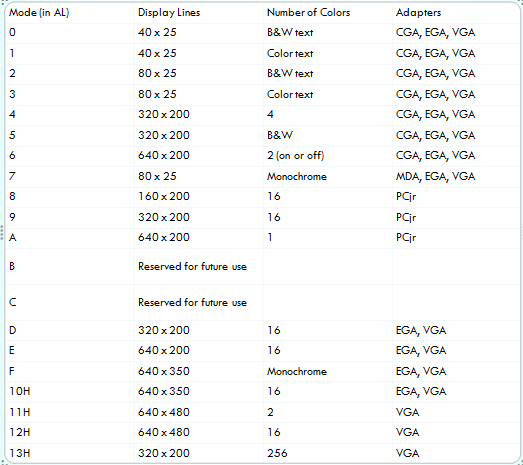
**Video Programming:**

* Basic background
* Controlling the color
* Int 10h video functions

**Int 10h:**

* Video Functions Interrupt
* Preserves only BX, CX,DX and segment registers.
* Other registers preserve onto stack before calling 10h.
* Ah = video Function Number.
* Al = Function Mode
* Mov ah,0 ;set video mode
* Mov al,3 ;choose mode 3
* Int 10h

**Screen Mode Numbers:**

****

**Getting Pixels AH=0CH:**

* Display a pixel…
  + - MOV AH, 0Ch
    - MOV AL, Color Number
    - MOV CX, X-axis ; CX = 10
    - MOV DX, Y-axis ; DX = 20
    - INT 10H
  + Example
    - MOV Al,00 ;Black Color
    - MOV Al,01 ;Blue Color
* MOV Al,03 ;Green Color

**Access to pixel:**

* Get/spot a pixel on screen by specify the row and column number.
* CX having Column Number
* DX having Row Number
* AL Color Number
* MOV AH, 0CH having function number for put a pixel on screen
  + Example
    - MOV CX, 10
    - MOV dx, 10
    - MOV al, 01 ;Blue
    - MOV ah, 0CH
    - INT 10H

**Clear Screen (CLS):**

* Put them in in procedur and call whenever you want to clear the screen.
* Example
  + - myfunction proc
    - MOV AL,03
    - MOV AH,0
    - INT 10H
    - ret
    - myfunction endp

Now call this procedure by “CALL myfunction”

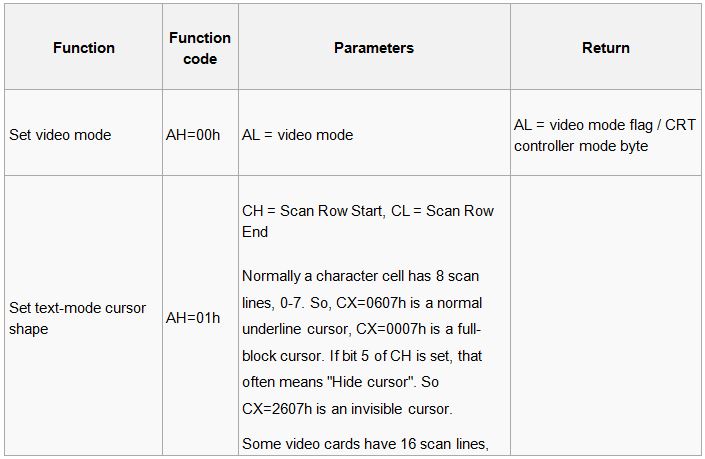
**Cursor Position (Goto XY) AH=02:**

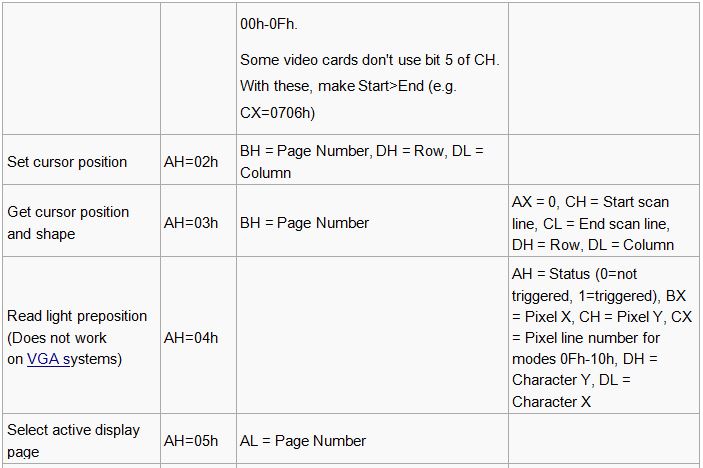
* Set cursor position using function number AH=02H.
* BH having page number (0,1,2,…)
* DX having Row and Column numbers (DH=ROW, DL=Column).
* Example
  + - MOV AH,02H
    - MOV DH, 10 ;Row Number
    - MOV DL, 20 ;Column Number
    - INT 10H

**Change Background Color AH=06:**

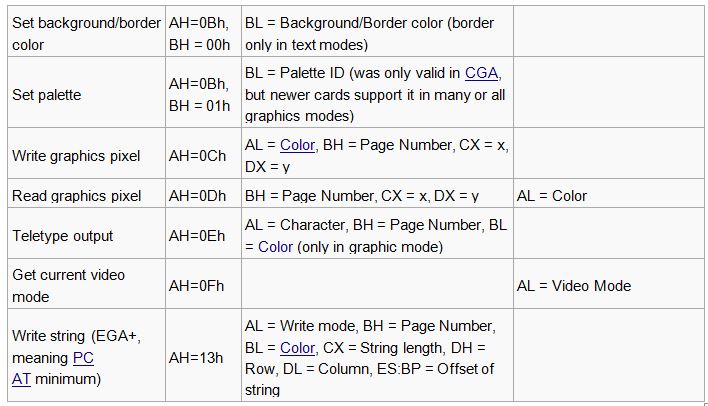
* Change background color using 06H funtion number of INT 10H
* MOV AH,06H
* AL having Scrol line
* BH having background and foreground color
* CX having (CH= upper row number, CL left coloumn number)
* DX having (DH= upper row number, DL left coloumn number)
  + Example
    - MOV AH, 06h
    - MOV AL, 0
    - MOV CX, 0
    - MOV DH, 80
    - MOV DL, 80
    - MOV BH, 14h
    - INT 10h

**Function Numbers With Parameters:**

****

****

****

****

## **Activity:**

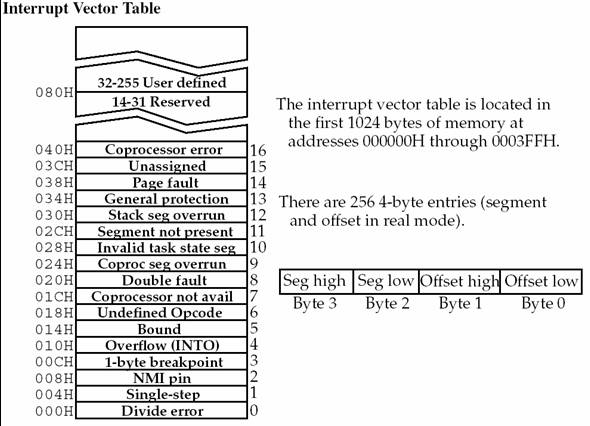
* Draw a border of any ASCII character in the center of the screen and write your name in this border using (MOV AH,02H),(MOV AH,09H) of BIOS interrupt (INT 10H).
* Draw a Square full of pixels by using (MOV AH,0CH)
* Draw a Rectangular by using ASCII character . (MOV AH,02H),(MOV AH,09H) of BIOS interrupt (INT 10H).

# **LAB 13: Interrupt Handling, Macros and Structure**

**Objectives:**

* Interrupt Handling
  + Interrupt Vector Table
  + Exceptions, Traps and Interrupts
  + Divide by Zero Exception
  + Overflow Exception
* Macros

**Interrupt Vector Table:**



**Replacing Existing Handler with user defined Handler:**

* int 21h/25h
  + ds:dx contains address of new function
  + ‘al’ register should contains Interrupt Number for which Handler/Function is to be changed.
* Example
  + mov dx, seg myfunc
  + mov ds, dx
  + mov dx, offset myfunc
  + mov al, 00h; Int 0h is triggered for Divde Zero Exception
  + mov ah, 25h
  + int 21h

**Structures**:

* Defining
  + Defined using ***struc*** and ***ends*** directives
  + Inside the structure, you define fields using the same syntax as for ordinary variables.
* Basic Syntax
  + name STRUCT
  + *data declaration*
  + nameENDS

**Example:**

* Defining a Structure Employee information with different fields as below

Diagram

Description automatically generated

**Declaring Structure Variables:**

* Under Data Segment
* .data
  + worker Employee <>

**Field Referencing:**

* To Access Structure Variable Fields, Under Code Segment
* .code
  + mov dx, worker.years
  + mov worker.salaryhistory,20000

**Macros**:

* Macros are just like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions.
* Macro Definition
  + name MACRO [parameters,...]
  + <instructions>
  + ENDM

**Using Macros:**

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

**Passing Arguments to Macro:**

* 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

Example:

* Unlike procedures, macros should be defined above the code that uses it.
* For Example
  + .code
  + mymacro macro p1,p2,p3
  + mov ax,p1
  + mov bx,p2
  + mov cx,p3
  + endm
  + main proc
  + mymacro 1,2,3
  + mov ah,4ch
  + int 21h
  + main endp
  + end

**Defining Macros in Separate file:**

* To define Macros in Separate file;
  + Open your assembler Source Directory
    - C:\masm611\include\
    - C:\emu8086\inc\
  + Create a File named “mymacros.inc”
  + Write your Macro in this file and Save. Make sure your file have extension .inc
  + Include this file in your source program (\*.asm), by writing below line on top of your code
    - include mymacros.asm
  + Compile your Code.

Graphical user interface, text, application, Word

Description automatically generatedGraphical user interface, text, application, Word

Description automatically generated

## **Activity:**

* Write an interrupt which is when called displays time on the screen.

# **LAB 14: Input – Output, Parallel Port Operation**

**Objectives:**

* Getting introduced to parallel port, introduction to pin configuration of the port
* Learning how to address parallel port of computer through assembly and how to write on parallel port and how to read data from any external source
* This lab is being designed to make the students enable to interface the microprocessor to external world and making them enable to control externally interfaced devices by microprocessor

**IN/OUT Instruction:**

|  |  |  |
| --- | --- | --- |
| **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. |
| **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. |

**REQUIPMENT:-**

* SOFTWARE:

1. Turbo assembler (TASM) or Microsoft Assembler(MASM)

* HARDWARE:

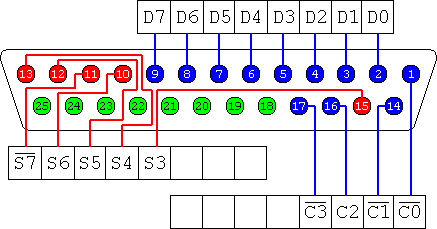
1. Computer with parallel port
2. Parallel cable
3. 8 LEDs
4. 8x1KΩ Resistances

**INTRODUCTION TO PARALLEL PORT:**

PC parallel pot is 25 pin D shaped female connector usually used for connecting printer and other devices. The pins are classified into following categories:

1. 8 output pins accessed via the DATA PORT
2. 5 input pins one inverted accessed via status port
3. 4 output pins (3 inverted) accessed via control port
4. Other 8 pins are grounded

This is the basic description of the parallel port which specifies the basic pin configuration of the parallel port which gives an idea about how to use the parallel port. You can see the pin configuration in the figure below:



The output of the parallel port is normally TTL level. The currents the pins can sink is different for different ports which can vary from 12mA to 20 mA.

**THE OUT COMMAND:**

The OUT command is used to write the data on parallel port. This command has the following syntax:

OUT dx, ax

The dx should contain the address of parallel port data register. The address of data pins of parallel port is **378h,** so you should movethis address todx. The above command sends the contents of AX to the address contained in DX. So you should use this pair of commands to write data on parallel port.

Mov dx, 378h

Out dx, ax

**ROUTINE TO GENERATE A DELAY:**

When generating different patterns you need to have a specific time delay in between different patterns. To generate a small delay you can write the function:

delay proc

mov dx,15000

l1:

dec dx

mov bx,0ffffh

l2:

dec bx

cmp bx,0

jne l2

cmp dx,0

jne l1

ret

delay endp

**THE CALL AND RET STATEMENT:**

The return command here is used to transfer back the control back to the calling procedure. Every procedure (except main procedure should have a RET statement at some place (usually the last statement). To invoke a procedure CALL instruction is used. The syntax is

CALL NAME

Executing CALL statement causes the following to happen:

1. The return address to the calling program is saved on the stack. This is the offset address of the next instruction after call statement.
2. IP gets the offset address of first instruction of the procedure which transfers the control to the procedure
3. To return the control back RET statement is used

**ROTATE INSTRUCTION:**

The ROL (Rotate left) and ROR (Rotate right) shift the bits to left and right respectively.

You can see in figure how does ROR works.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Syntax is

ROR destination,cl

The value of cl would determine how much time the ROR would be executed.

**PROCEDURE:-**

**Writing on parallel port:**

Before starting the procedure here is a sample code to blink all the LEDs.

.model small

.code

main proc

mov cx,5

start1:

mov ax,0ffffh

mov dx,0378h

out dx,ax

call delay

mov ax,0h

mov dx,0378h

out dx,ax

call delay

jmp start1

mov ah,4ch

int 21h

main endp

delay proc

mov dx,15000

l1:

dec dx

mov bx,0ffffh

l2:

dec bx

cmp bx,0

jne l2

cmp dx,0

jne l1

ret

delay endp

end main

* Connect the positive ends of LEDs to data pins (pin 2 to 9) and negative ends to resistances
* Ground the other end of resistances i.e. connect to any other pin (from pin 18 to 25)
* Assemble the code and verify the pattern you sent on the port

## **Activity:**

1. Turn on all the LEDs
2. Create an LED pattern by turning on LEDs towards left
3. Create an LED pattern by turning on LEDs towards right
4. Create an LED pattern by just blinking odd LEDs
5. Write the codes in specified blank space

Insert a proper delay between the patterns. Don’t connect more than one LED with a single pin of parallel port, because it can sink only a current up to 20mA.