

LAB MANUAL

Course: CSC321 Microprocessor and Assembly Language



Department of Computer Science

Assembly Language learning

- 1) Stage **J** (Journey inside-out the concept)
- 2) Stage **a₁** (Apply the learned)
- 3) Stage **V** (Verify the accuracy)
- 4) Stage **a₂** (Assess your work)

Table of Contents

Lab #	Topics Covered	Page #
Lab # 01	Data Representation: Conversions in Java	3
Lab # 02	Instruction set architecture (16 bits): Design and Working	10
Lab # 03	Debug Tool and Usage of software Interrupt (INT command)	17
Lab # 04	Assembly Language Introduction; MASM configuration; Usage of Dumpregs; Input and Output in Assembly using Call WriteTYPE and Call ReadTYPE	26
Lab # 05	Data Types in Assembly: BYTE, WORD and DWORD	36
Lab # 06	Data Transfer and Arithmetic Instructions; Flag affected by arithmetic; Data Related operators	42
Lab # 07	Array Processing (Direct, Indexed and Indirect); LOOP and Nested LOOP	50
Lab # 08	Introduction to Procedures; User defined Procedures; Procedures having flexibility	60
Lab # 09	Boolean Operations: AND, OR, NOT, XOR, TEST and CMP	72
Lab # 10	Flow Control and Conditional Jump Instructions: Signed Vs. Unsigned	82
Lab # 11	Implementation of HLL Control Structures	93
Lab # 12	Shift: Logical and Arithmetic; Signed and Unsigned Shift; Rotate Instructions (Left or Right and Others)	100
Lab # 13	Multiplication Instructions; MUL and IMUL; Applications of Multiplication	107
Lab # 14	Division Instructions; DIV and IDIV; Applications of Division	114
	Terminal Examination	

Statement Purpose:

This lab will give you practical implementation of conversions and arithmetic on unsigned and signed integers and real numbers among different radices of decimal, binary, and hexadecimals using programs in JAVA.

Activity Outcomes:

This lab teaches you the following topics:

- **Conversions between different radices.**
- **Representations of signed and unsigned integer numbers.**
- **Representation of floating point numbers.**
- **Arithmetic operations on numbers of all types.**

Instructor Note:

As pre-lab activity, read Chapter 1 section 1.3 from the book (Assembly Language for X86 processors, KIP R. IRVINE., 7th Edition (2015), Pearson), and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

Everything suited for processing with digital computers is represented as a sequence of 0s and 1s, whether it be numeric data, text, executable files, images, audio, or video. The meaning of a given sequence of bits within a computer depends on the context. In this section we describe how to represent integers in binary, decimal, and hexadecimal and how to convert between different representations. We also describe how to represent negative integers and floating-point numbers

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Hexadecimal to decimal and binary conversion, This Java program will convert Hexadecimal number to decimal, binary and Octal in Java programming language using JDK standard API methods. For beginner's hexadecimal is base 16 number, while decimal is base 10, Octal is base 8 and binary is base 2 numbers in Number systems. Binary only contains 0 and 1 bits, octal contains 0 to 7, decimal contains 0 to 9 and Hexadecimal contains 0-9, A, B, C, D, E, F where F represent 16. Thankfully Java library provides convenient method to convert any integer from one number system to another.

Solution:

Java API provides two methods which is used in converting number from one number system to other. One is Integer.parseInt() which is used to convert String to Integer in Java but also allows you to specify radix.

```
* Java program to convert Hexadecimal to binary, decimal and Octal in Java.  
* Hexadecimal is base 16, Decimal number is base 10, Octal is base 8  
* and Binary is base 2 number which has just two numbers 0 and 1.  
* @author  
*/
```

```
public class ConvertHexaToDecimal {  
    public static void main(String args[]) {  
        // Ask user to enter an Hexadecimal number in Console  
        System.out.println("Please enter Hexadecimal number : ");  
        Scanner scanner = new Scanner(System.in);  
  
        String hexadecimal = scanner.next();  
  
        //Converting Hexa decimal number to Decimal in Java  
        int decimal = Integer.parseInt(hexadecimal, 16);  
  
        System.out.println("Converted Decimal number is : " + decimal);  
    }  
}
```

```

//Converting hexa decimal number to binary in Java
String binary = Integer.toBinaryString(decimal);
System.out.printf("Hexadecimal to Binary conversion of %s is %s %n", hexadecimal,
binary );

// Converting Hex String to Octal in Java
String octal = Integer.toOctalString(decimal);
System.out.printf("Hexadecimal to Octal conversion of %s is %s %n", hexadecimal,
octal );
    }
}

```

Output:

Please enter Hexadecimal number :

A

Converted Decimal number is : 10

Hexadecimal to Binary conversion of A is 1010

Hexadecimal to Octal conversion of A is 12

Activity 2:

Example shows how to convert binary to decimal in JAVA.

Solution:

- **Create Java Main Class and named it Binary_Decimal.**
- **Copy the Code and test it by running the application**

```

import java.util.Scanner;
public class Binary_Decimal {
    Scanner scan;
    int num;
    void getVal() {
        System.out.println("Binary to Decimal");
        scan = new Scanner(System.in);
        System.out.println("\nEnter the number :");
        num = Integer.parseInt(scan.nextLine(), 2);
    }
    void convert() {
        String decimal = Integer.toString(num);
        System.out.println("Decimal Value is : " + decimal);
    }
}
class MainClass {
    public static void main(String args[]) {
        Binary_Decimalobj = new Binary_Decimal();
        obj.getVal();
        obj.convert();
    }
}

```

Activity 3:

The example below, how to convert from binary to Octal in JAVA.

```
import java.util.Scanner;
public class Binary_Octal {
    Scanner scan;
    int num;
    void getVal() {
        System.out.println("Binary to Octal");
        scan = new Scanner(System.in);
        System.out.println("\nEnter the number :");
        num = Integer.parseInt(scan.nextLine(), 2);
    }
    void convert() {
        String octal = Integer.toOctalString(num);
        System.out.println("Octal Value is : " + octal);
    }
}
class MainClass {
    public static void main(String args[]) {
        Binary_Octalobj = new Binary_Octal();
        obj.getVal();
        obj.convert();
    }
}
```

Activity 4:

Here it is demonstrated how to convert from binary to hexadecimal.

Solution:

```
import java.util.Scanner;
public class Binary_Hexa {
    Scanner scan;
    int num;
    void getVal() {
        System.out.println("Binary to Hexadecimal");
        scan = new Scanner(System.in);
        System.out.println("\nEnter the number :");
        num = Integer.parseInt(scan.nextLine(), 2);
    }
    void convert() {
        String hexa = Integer.toHexString(num);
        System.out.println("Hexadecimal Value is : " + hexa);
    }
}
```

```

class MainClass {
    public static void main(String args[]) {
        Binary_Hexaobj = new Binary_Hexa();
        obj.getVal();
        obj.convert();
    }
}

```

Activity 5:

Here your job is to write a function that takes an integer input and returns the number of 1's in its binary representation.

Solution:

```

public static intbitCount(int input) {
    int count = 0;
    for (int i = 0; i < 32; i++)
        count = count + (input >>> i & 1);
    return count;
}

```

```

public static intbitCount(int x) {
    if (x == 0) return 0;
    return (x & 1) + bitCount(x >>> 1);
}

```

This is how Integer.bitCount() is implemented by Java. See if you can figure out how it works.

```

public static intbitCount(int i) {
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f;
}

```

Stage V(verify)

Home Activities:

Activity 1:

Covert the following programs Pseudo-code in JAVA.Just use the algorithms given in activity.

1. Conversion from base 2 and base 16 numbers to Decimal

$$\begin{aligned}d_1 d_0 &= d_1 \times b^1 + d_0 \times b^0 \\&= (d_1 \times b) + d_0, \\d_2 d_1 d_0 &= d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 \\&= ((d_2 \times b) + d_1)b + d_0, \\d_3 d_2 d_1 d_0 &= d_3 \times b^3 + d_2 \times b^2 + d_1 \times b^1 + d_0 \times b^0 \\&= (((d_3 \times b) + d_2)b + d_1)b + d_0.\end{aligned}$$



```
Result = 0;  
for (i = n - 1 downto 0)  
    Result = (Result × b ) + di  
end for
```

2. Conversion from decimal number to base 2 and base 16

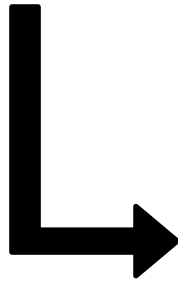
	Quotient	Remainder
167/2 =	83	1
83/2 =	41	1
41/2 =	20	1
20/2 =	10	0
10/2 =	5	0
5/2 =	2	1
2/2 =	1	0
1/2 =	0	1



```
Quotient = decimal number to be converted  
while (Quotient ≠ 0)  
    next most significant digit of result = Quotient MOD b  
    Quotient = Quotient DIV b  
end while
```

3. Conversion of ay Binary fraction to Decimal

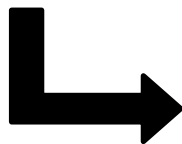
A fractional binary number $0.d_1 d_2 \dots d_{n-1} d_n$ with n bits



```
decimal_value = 0.0
for (i = n downto 1)
    decimal_value = (decimal_value +  $d_i$ )/b
end for
```

4. Conversion of decimal Fraction to Binary

0.78125×2	$=$	1.5625	\longrightarrow	1
0.5625×2	$=$	1.125	\longrightarrow	1
0.125×2	$=$	0.25	\longrightarrow	0
0.25×2	$=$	0.5	\longrightarrow	0
0.5×2	$=$	1.0	\longrightarrow	1
Terminate.				



```
value = fraction to be converted
digit_count = 0
repeat
    next digit of the result = integer (value  $\times$  b)
    value = fraction (value  $\times$  b)
    digit_count = digit_count + 1
until ((value = 0) OR (digit_count = F))
```

5. Decimal number to Floating Point Binary Representation

Step 1: *Convert the real number to binary.*

1a: Convert the integer part to binary

1b: Convert the fractional part to binary

1c: Put them together with a binary point.

Step 2: *Normalize the binary number.*

Move the binary point left or right until there is only a single 1 to the left of the binary point while adjusting the exponent appropriately. You should increase the exponent value by 1 if the binary point is moved to the left by one bit position; decrement by 1 if moving to the right.

Step 3: *Convert the exponent to excess or biased form.*

For short reals, use 127 as the bias;

For long reals, use 1023 as the bias.

Step 4: *Separate the three components.*

Separate mantissa, exponent, and sign
to store in the desired format.

3) Stage 2 (assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

Objective of this lab is to introduce students about 16-bit instruction set architecture. The way it is designed and work. Students will be guided about the way different instructions are formed. Instruction are composed of two main parts: opcode and operand. Here, our focus will be on key operations like mov, add, sub, and store. The operands are immediate, register or memory location. Here, we kept the opcode size of 4 bits so that each instruction format supports at max 16 operations. While other 12 bits are kept for the operands.

Activity Outcomes:

This lab teaches you the following topics:

- Students will know that the way 16-bit instruction set architecture is designed and work.
- Students will be guided about the way different instructions are formed.
- Students will get knowledge of the working of MOV, ADD, SUB and STORE commands in 16 Bit Architecture.

Instructor Note:

Instruction Set

Composed of two parts

- Opcode defines the number of operation supported by this instruction format and used for indexing that operation.
- Operands can be immediate value, memory address and register address.

In our ISA design, the length of instruction is 16 bits. Such that

- 4 bits for opcode
- 12 bits for operand(s)

Operations

- Our focus will be on three operations
 - LOADM
 - ADDR
 - STORER

OPCODE	Operation
0000	LOADI
0001	LOADM

0010	STOREI
0011	STORER
0100	ADDI
0101	ADDR
0110	SUBI
...	...
...	...
1111	...

Registers

- There are 16 registers hence 4 bits are required to address them
 - Registers are named as R0 to R15.

Register Address	Register
0000	R0
0001	R1
0010	R2
0011	R3
0100	R4
0101	R5
0110	R6
...	...
...	...
1111	R15

Memory

- There are 2^8 memory locations where data may reside. Hence 8 bits are required to address them
 - Memory locations are named as M0 to M255.

Memory Address	Memory
00000000	M0
00000001	M1
00000010	M2
00000011	M3
00000100	M4
00000101	M5
00000110	M6
...	...
...	...
11111111	M255

1) Stage J(Journey)

Introduction

LOADM Instruction:

- **Define:**
 - It loads an operand value from a memory location to a register.
- **Bit allocation:**
 - bits for Opcode
 - 4 bits for Register address
 - 8 bits for Memory address

LOADM	REGISTER ADDRESS	MEMORY ADDRESS
0	3	7 15

- **Working**
 - Consider an example:

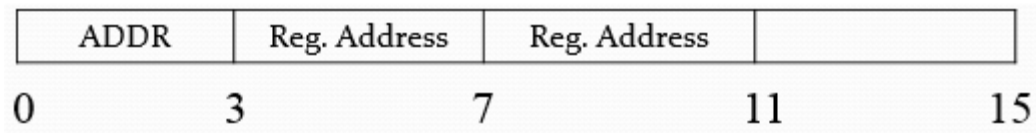
Assembly Code: **LOADM R4 M6**

Machine Code: **0001010000000110**

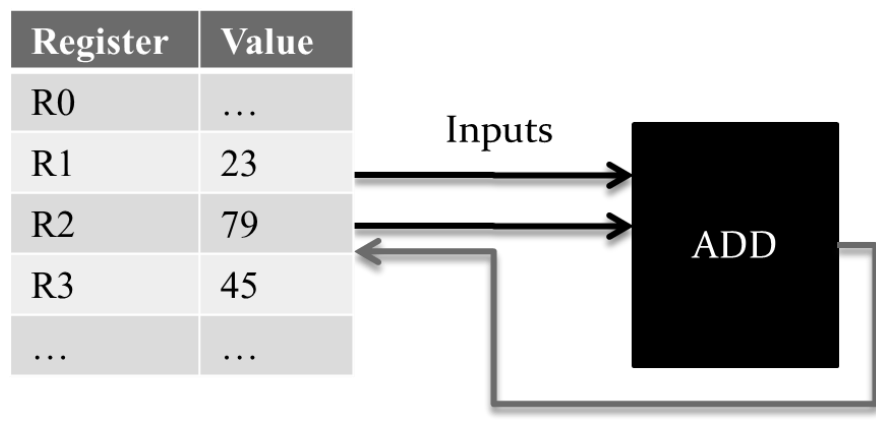
Register address	Value	Memory Address	Memory
...
R4	87	M6	87
...

ADDR Instruction:

- **Define:**
 - It adds register values and keeps the result in the destination register.
- **Bit allocation:**
 - 4 bits for Opcode
 - 4 bits for each Register address
 - 4 bits for Source and 4 bits for Destination



- **Working**



2) Stage a1 (apply)

Lab Activities:

Activity 1:

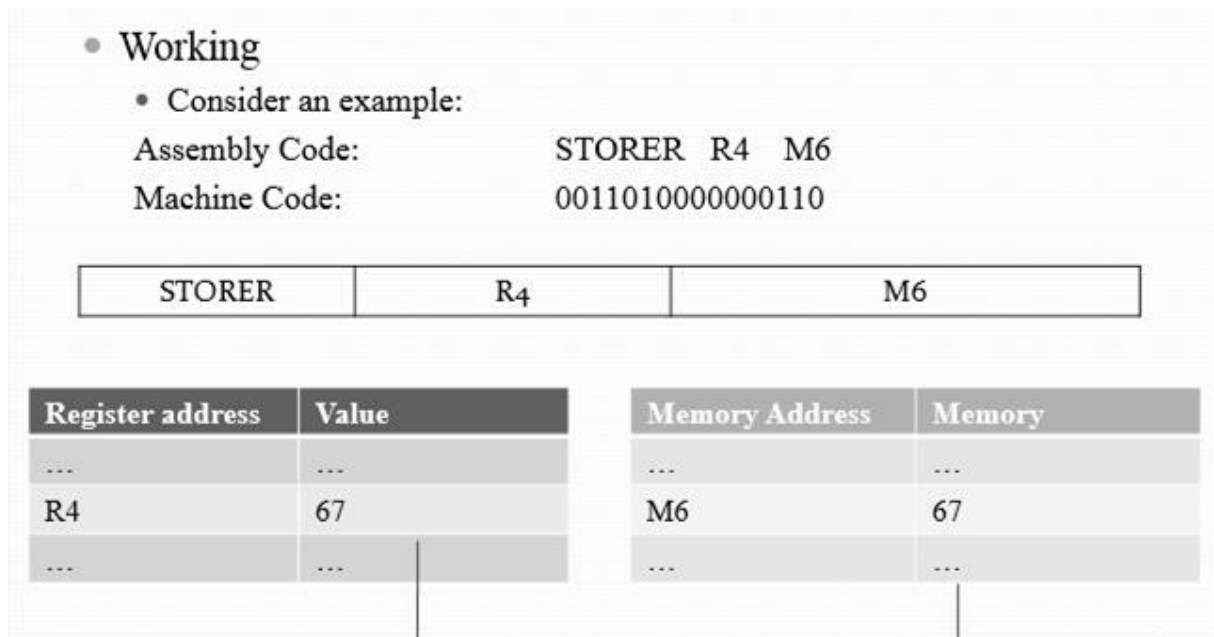
Consider the following instruction and provide its working based on the example discussed for LOADM.

STORER Instruction:

- **Define:**
 - It stores an operand value from a register to a Memory location.
- **Bit allocation:**
 - bits for Opcode
 - 4 bits for Register address
 - 8 bits for Memory address



Solution:



Activity 2:

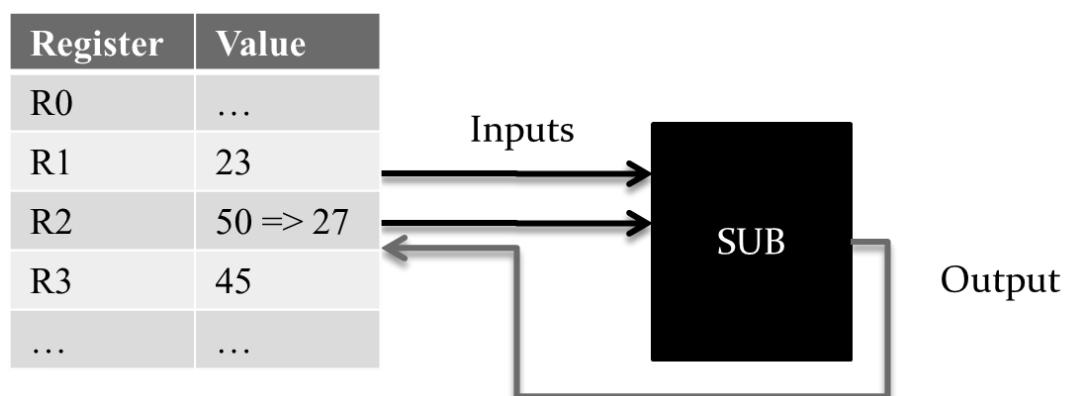
Consider the following instruction and provide its working based on the example discussed for ADDR.

SUBR Instruction:

- Define:
 - It subs register values and keeps the result in the destination register.
- Bit allocation:
 - 4 bits for Opcode
 - 4 bits for each Register address
 - 4 bits for Source and 4 bits for Destination



Solution:



3) Stage V(verify)

Home Activities:

1. Write a simulator for the discussed ISA design in JAVA language.

There should be an input function. It asks for the number of instructions of the program to be executed by simulator. Let suppose user enter 4.

It means there would be two load, one adds and one store instruction for this program.

Input pattern:

- **Step 1: Enter Instruction type (Load, Store or Add)**
- **Step 2: Ask for the operand input according to the Instruction.**

If user selected Load in step 1 then ask for

- **Operand # 1: Register Number**
- **Operand # 2: Memory Address**

Ask for the value of memory as well. (Store Memory in SD Array)

Input Test program (4 Instructions)

```
LOADM   R1    M8
          LOADM   R2    M7
          ADDR    R2    R1
          STORER   R2    M8
```

OR

```
1  1  8
1  2  7
5  2  1
3  2  8
```

Execution of the Instruction

- **Our CPU will use three stages for executing a single instruction. These are**
- **Fetch: CPU fetches the instruction from Instruction memory (Maintained in a 2D Array)**
- **Decode: Operation and operands are determined.**
- **Execute: Operation is performed on the operands.**


```

*****ISA Simulator*****
*****Cycle # 1*****
Fetch Instruction          Decode Instruction          Execute Instruction
1      1      8

Register: 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

#####
Memory:  10  20  30  40  50  60  70  80  90  30  50  60  70  80  90  10

```

```

*****Cycle # 2*****
Fetch Instruction          Decode Instruction          Execute Instruction
1      1      8
                        LOAD
                        Register R1
                        Memory M8

Register: 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

#####
Memory:  10  20  30  40  50  60  70  80  90  30  50  60  70  80  90  10

```

```

*****Cycle # 3*****
Fetch Instruction          Decode Instruction          Execute Instruction
1      1      8
                        LOAD
                        Register R1
                        Memory M8

Register: 0  90  0  0  0  0  0  0  0  0  0  0  0  0  0  0

#####
Memory:  10  20  30  40  50  60  70  80  90  30  50  60  70  80  90  10

```

4) Stage2(assess)

Demonstrate the home activity and Viva voce

Statement Purpose:

This lab will give you practical understanding of machine architecture (Von-Neumann and IA-32) with the help of DEBUG program. You will also learn how programs are run via Fetch-Decode-Execute cycle in Step by Step Mode. You will also understand the differences of little-endian and big-endian formats as well as how data is stored in main memory, registers, and stack and how to view, edit the data at runtime of programs. It is implementation of conversions and arithmetic on unsigned and signed integers and real numbers among different radices of decimal, binary, and hexadecimal using programs in JAVA.

Activity Outcomes:

This lab teaches you the following topics:

- **What is Von-Neumann architecture?**
- **IA-16 and IA-32 architecture from programmer's perspective.**
- **How Fetch-Decode-Execute cycle works?**
- **Differences between Little-Endian and Big-Endian formats.**
- **Viewing and editing contents of registers and main memory at run time.**

Instructor Note:

As pre-lab activity, read web material via googling on Introduction to Debug and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

Normally, there is no way to see the source code of a program while the program is running. This inability to "see under the covers" while the program is executing is a real handicap when you are debugging a program. The most primitive way of looking under the covers is to insert (depending on your programming language) print or display, or exhibit, or echo statements into your code, to display information about what is happening. But finding the location of a problem this way can be a slow, painful process. This is where a debugger comes in.

A debugger is a piece of software that enables you to run your program in debugging mode rather than in normal mode. Running a program in debugging mode allows you to look under the covers while your program is running. Specifically, a debugger enables you (1) to see the source code of each statement in your program as that statement executes, (2) to suspend or pause execution of the program at places of your choosing, (3) while the program is paused, to issue various commands in order to examine and change the internal state of the program, and (4) to resume (or continue) execution.

Debuggers come in two flavors (1) console-mode (or simply console) debuggers and (2) visual or graphical debuggers. Console debuggers are often a part of the language itself, or included in the language's standard libraries. The user interface to a console debugger is the keyboard and a console-mode window (Microsoft Windows users know this as a "DOS console"). When a program is executing under a console debugger, the lines of source code stream past the console window as they are executed. A typical debugger has many ways to specify the exact places in the program where you want execution to pause. When the debugger pauses, it displays a special debugger prompt that indicates that the debugger is waiting for keyboard input. The user types in commands that tell the debugger what to do next. Typical commands would be to display the value of certain program variables, or to continue execution of the program.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Typing various commands.

Type the following commands and see the behavior along with understanding of register dumps.

Debug, Q, R, A, T, G.

Solution:

- (i) Debug (enter the program)
- (ii) Q (Quit from program)
- (iii) R (To see contents of all registers)

- (iv) R <register> (To see specific register contents + option to change its value).
see the contents of AX register and try changing its values to 1, 21, 321, 4321, 54321, GF

Try to see value of AL (AX=AH+AL)

- (v) A (Assemble command)

To enter assembly language instructions into memory

A <starting address>

starting address is offset into Code Segment or you can specify Segment register.

Note: A 100 OR A CS: 100 are same

now type A 100 and Debug will prompt you to type assembly language program.

Type these instructions (pressing CR at end of each)

mov ax,1

mov bx,2

mov cx,3

add ax,bx

add ax,cx

INT 3

<CR> (Without typing an instruction)

Note: Debug assumes all numbers are in hex while most assemblers do not.

Also as you type instructions, Debug converts them to machine code

Also do not assemble beginning at address lower than 100 (first 256 bytes reserved for DOS and your program should not overwrite this area)

- (vi) U Command (un-assemble command)

To see machine code of your program. It takes machine code stored in memory and converts it back to assembly language.

U <starting address><ending address> or

U <starting address><L no of bytes>

U 100 10D or

U 100 LD (D bytes from start)

- (vii) G (Go command)

To execute instructions until breakpoint is reached.

At end register contents are displayed again and prompt is displayed.

G

G =100

G =100 109

now check the value of IP register, change it back to 100 and then use

G 109 (start address is not given so assume what is in IP, 109 is ending address here)

Activity 2:

Example shows how to understand the registers dumps and trace the program execution.

Solution:

- (i) Using R command.

First Line: General purpose registers, Index and pointer registers

Second Line: Segment registers, IP's value and FLAGS

Third Line: Location, Machine Code and Assembly Code of next instruction to be executed.

- (ii) Trace the program execution

T <=starting address><no of instructions>

Default starting=CS:IP

Default no of instructions=1

Trace will display register contents after each instruction.

T=100 5 (on your program) or

T 5 (keep in mind default for 1st fields)

Note that IP change value and points to next instructions address.

3rd line of registers show the Instruction and its machine code (which will be executed next)

Also notice how registers are changing values after each instruction esp AX register.

Some Errors

Try to write this program starting at 100 hex address

```
A 100
mov AL, FF3
mov AX, 23451
mov DS, 1200
mov SI, DH
mov AX, BH
mov AL, BX
<CR>
Q <CR>
```

Activity 3:

The example helps you understand unassembled command and physical address calculation.

- (i) Write a program starting at offset 100 as shown below

A 100

```

MOV AL,57
MOV DH,86
MOV DL, 72
MOV CX,DX
MOV BH,AL
MOV BL,9F
MOV AH,20
ADD AX,DX
ADD CX,BX
ADD AX, 1F35
<CR>

```

U 100 112 (Un-assemble)

(ii) Physical Address calculation

Note that program is starting at CS:IP (CS:0100).It is called logical address but remember address bus in 8088 is 20 bit. Physical address will be calculated as

Let CS=1235

IP=0100

Physical Address=12350+100=12450

In Hardware, 4-bit left shift of CS and then add IP

Activity 4:

Here it is demonstrated how to Examine and alter the memory contents and write dynamic programs that will change variable value every time the program is run.

Solution

(i) Memory Dump commands

F (Fill command)

E (Enter command)

D (Dump command)

D to see memory contents and F used to fill data in memory, used as

F <starting address><ending address><data> or

F <Starting address><L no of bytes><data>

Note: D (to see memory contents) syntax is also same

usually used to fill data segment (default offset in) with some initial values (zero say)

To fill another segment, mention segment with offset.

F 100 10F FF (Filling 16 bytes with FF)
F CS:100 1FF 20 (Filling 256 bytes/block of code segment with space)
F 100 L20 00 FF (Filling 20 hex (32) bytes alternately with 00 and FF)

Debug<cr>
F 100 14F 20
F 150 19F 00 (number after 14F is 150)
D 100 19F

F 104 10A FF
D 104 10A
D 100 10F
F CS:100 12F 20
D CS:100 12F

Now un-assemble your previous program and dump the memory and note how machine code bytes are stored in consecutive memory areas.

U 100 122
D CS:100 11F

Note: if byte value is not ASCII code, a dot (.) will be displayed.

E (to enter/alter any data values into memory portion)

F can fill same data while E is free

Syntax

E <address><data list> or

E <address>

Examples:

E 100 'AKBAR SALEEM'

D 100:10F

E 100 23 B5 03 4F (To enter numeric data)

To alter Data

E 106 (WILL show value at address 106, change it to other byte simply typing, you can press <CR> or <space>-you can go through each byte at a time by hitting space after each byte display or <-> instead of typing any value...see what each one shows)

Try change AKBAR'S first A with E

Now write a program whose data will be entered later on via E command as shown below

```
-A 100
mov AL, 00
add AL, [0200]
add AL, [0201]
add AL, [0202]
```

```

add AL, [0203]
add AL, [0204]
INT 3
<CR>
-E DS:0200 23 12 14 1F 2C
-D 0200 020F
-G 100 116

```

Activity 5:

Here you will learn how Little-endian Format of Data storage differs from Big Endian Format and how stack shrinks and expands in memory

Solution:

Low byte in register to Low byte address in memory, High byte in register to High byte address in memory

1. Move the value at location 6830 into BX register (note two values will be moved one at location 6830 and other at 6831 as BX is 16-bit)

- First Dump the 10 bytes from 6825 (D 6825 LA) and note the value at 6830 and 6831.
- Assemble a program at 100 (A 100).
- `mov BX, [6830] INT 3 <CR>`.
- Trace the program and check how value is stored in BX (Little Endean).

2. How stacks grow and shrinks in memory.

SP is attack pointer that points to start of the stack but start here is higher address in memory. So SP will be decremented each time when value will be pushed. Stack will grow toward lower memory address.

For POP operation, phenomena are reverse.

-A 100

```

mov AX, 24BF
mov DI, 85C0
mov DX, 5F90
mov SP, 1230
push AX
push DI
push DX
int 3

```


<cr>

-F 1225 1240 00 (Stack memory filled with all 00)

-D 1225 LF (Check)

-G =100 (Run program and see values of stack pointer)

-D 1225 1240 (CHECK STACK IS PUSHED OR NOT?)

3) StageV(verify)

Home Activities:

Activity 1:

Perform the following activities at your home PC.

- A. Create a DEBUG script file using any available editor or word processing software, for example, DOS Edit. Enter the lines below and save the file as "COVER.SCR" in the C:\SCRATCH subdirectory. If such a subdirectory does not exist, create it by issuing the command: "MD SCRATCH", then set the default path to it using the command: "CD C:\SCRATCH". You do not have to enter the comments. When executed, this program waits for a keystroke and then fills the screen with that character.

```
A 100
MOV BH,0      ;SET PAGE NO. FOR INT 10
MOV DX,0      ;ROW AND COL FOR VIDEO DISPLAY OF INT 10
MOV AH,2      ;DESIGNATE SERVICE 2 OF INT 10
INT 10        ;SET CURSOR TO 0,0 (TOP LEFT CORNER)
MOV AH,8      ;DESIGNATE SERVICE 8 OF INT 21
INT 21        ;KEYBD INPUT W/O ECHO, PLACE IN AL
MOV CX,7D0    ;LOAD NUM CHAR=2000D TO CX FOR WRITING ,INT 10
MOV AH,A      ;SERVICE A OF INT 10
INT 10        ;WRITE CHARACTER IN REG. AL CX TIMES
JMP 105       ;RETURN AND REPEAT- INFINITE LOOP
<blank line> <---- insert a blank line in the file
R CX
16
N C:\SCRATCH\COVER.COM
W
Q
<enter> <---- press the "Enter" key
```

- B. To create the executable DEBUG program, COVER.COM, enter "DEBUG <C:\SCRATCH\COVER.SCR" at the DOS prompt. At this point, the assembled code has been stored as COVER.COM in the C:\SCRATCH subdirectory. To see the assembly code, invoke DEBUG and use the N, L, and U commands in sequence (N COVER.COM, L, U). Try entering "U 100" and then "U 101" or "U 102". See anything different?
- C. To execute the program in DEBUG issue the command: G=100, or at the DOS prompt in the C:\SCRATCH subdirectory, enter: "COVER". To terminate the program in DEBUG or DOS, enter: Ctrl-C. Follow the comments and study the role of the registers in accomplishing the desired result. What happens if you change "MOV CX,7D0" to "MOV CX,3E8"? You can do this by changing just this one line of code in DEBUG:
- D. -a cs:xxxx<enter>
-cs:xxxx MOV CX,3E8 where xxxx is the offset of that line of code
- E. Using DEBUG, display the memory locations listed below, which describe the computer's configuration and help execute commands.

0:417h-418h	These two locations describe the status of special keys, such as shift, and control
7 in the class	Note the effect of the various key presses (see fig. 6-6, and 6-7 in the class packet, the section titled: "More about Keyboards and Scan Codes").
0:41Ah	points to the keyboard buffer head location
0:41Ch	points to the keyboard buffer tail location
0:41Eh-43Ch	keyboard buffer: space for 15 key strokes beginning with 41Eh. Note the
	ASCII and scan codes in the buffer and relate them to
commands entered.	
0:449h	video mode
0:44Ch-44Dh	amount of display memory in bytes (remember "backwards" convention)

To see the operation of the keyboard buffer, in DEBUG issue the command: "d 0:41a" several times to fill the buffer with copies of this command. Using the ASCII display portion of DEBUG, identify the ASCII and scan codes for "d 0:41a" in the keyboard buffer. Remember key codes are stored in the buffer from low to high memory locations.

- F. To observe the effect of changing character attributes, use the DEBUG "Enter" command (E) to enter values directly into video RAM. You must first determine the appropriate addresses to use in video RAM. The starting address of video RAM depends on the video mode (check location 0:449 for video mode, which will be a text mode). Consult Fig. 4-10 in the class packet, the section titled: "More about Video Modes" for the starting segment address for the indicated mode. The offset address depends upon where you wish to place the characters on the screen.

In text mode, the complete screen contents are contained in memory between offset 0000h and 0F9Fh (0000d to 3999d). Pick a RAM location that maps to a row that you can easily see. A good screen location is in the last row, third column. Calculate the character memory offset value from the formula: $CO = ROW * A0 + COL * 2$. Remember rows are numbered (in hex) 0,1,...,18, and columns 0,1,2,...,4F. Using DEBUG, enter the desired ASCII and attribute bytes.

4) Stage 2 (assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

This lab will give you practical demonstration of how to setup Visual Studio so that assembly language programs would be written. You will identify each and every option relevant to Assembler. You will also write simple Assembly language programs and run them in debug mode. Also setting the environment to run 16-bit applications will be practices.

Activity Outcomes:

This lab teaches you the following topics:

- **Understand Assemblers.**
- **Making Basic Settings for Visual Studio 's Assembler**
- **Running the program in Visual Studio Debugging Mode**
- **Understanding breakpoints and memory dumps.**
- **16-bits applications.**

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Assembly Language for X86 processors, KIP R. IRVINE., 7th Edition (2015), Pearson), and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

In this lab, you will set up the MS visual Studio VC++ to check various commands, tools, and options available to help you write Assembly Language programs. Also you will start writing basic programs in Assembly Language and get understanding of various data types and operators along with basic data movement instructions.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

First, you must install Visual Studio 2010 and select the VC++ language option.

Note: In Lab it will be installed already.

VS 2010 and Visual C++ 2010 Express both include the current version of the Microsoft Assembler.

Verify that the Microsoft Assembler is installed by looking for the file ml.exe in the \vc\bin folder of your Visual Studio installation directory.

The latest copy of the book's link libraries and example programs have been copied to the c:\Irvine folder. Check them out.

Solution:

Check that the following files are copied into the c:\Irvine directory: You can also see Desktop copy as well.

Filename	Description
cmd.exe	Shortcut to the Windows command-line interpreter (named cmd.exe)
GraphWin.inc	Include file for writing Windows applications
Irvine16.inc	Include file used with the Irvine16 link library (16-bit applications)
Irvine16.lib	16-bit link function library used with this book
Irvine32.inc	Include file used with the Irvine32 link library (32-bit applications)
Link16.exe	16-bit linker
Irvine32.lib	32-bit link function library used with this book

User32.lib	Basic I/O link library
Macros.inc	Include file containing macros (explained in Chapter 10)
SmallWin.inc	Small-sized include file, used by Irvine32.inc
make16.bat	Batch file for building 16-bit applications
VirtualKeys.inc	Keyboard code definitions file, used by Irvine32.inc

A subdirectory named Examples will contain all the example programs shown in the book, as well as all the source code for the book's 16- and 32-bit link libraries.

Activity 2:

Example shows how to perform the following steps.

- (i) Add 'Start without Debugging' button in Debug menu.
- (ii) Set Tab size to 5 for all languages.
- (iii) Building the sample Assembly Language programs.

Solution:

Adding button:

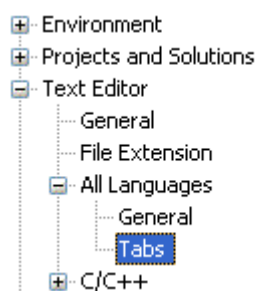
It's very useful to run programs without having to debug them. To do that, you will want to add a new command to the Debug menu: Start Without Debugging. Here's how to do it:

1. From the **Tools**, menu, select *Customize*.
2. Select the *Commands* tab.
3. Select **Menu** bar (**radiobutton**).
4. Click the **Add Command** button.
5. Select *Debug* from the Categories list.
6. Select **Start Without Debugging** in the right-hand list box.
7. Click the **OK** button.
8. Click the **Close** button.

In fact, you can use the same sequence to customize any of the menus and toolbars in Visual Studio.

Setting Tab size for all Languages:

Start Visual Studio, and select Options from the Tools menu. Select Text Editor, Select All Languages, and select Tabs:



Set the Tab Size and Indent Size to 5.

Building Sample Assembly Language Program

Visual Studio and Visual Studio Express require assembly language source files to belong to a project, which is a kind of container. A project holds configuration information such as the locations of the assembler, linker, and required libraries. A project has its own folder, and it holds the names and locations of all files belonging to it. We have created a sample project folder in the c:\Irvine\examples\ch03 directory, and its name is Project.

Do the following steps, in order:

- 1) **Start Visual Studio.**
- 2) **First you will open an existing Visual Studio project file. If you're using Visual Studio, select Open Project from the File menu. Or, if you're using Visual Studio Express, select Open, and select Project/Solution.**
- 3) **Navigate to the Desktop\Irvine\Examples\ch03 folder and open the file named Project.sln.**
- 4) **In the Solution Explorer window, you will see the word Project. This is the name of a Visual Studio project.**
- 5) **Click the small + sign and you will see assembly template file named main.asm Double click to open it. If main.asm is not there then you need to add it to the project. To do that, right-click on Project, select Add, select Existing Item, select main.asm, and click the Add button to close the dialog window. (You can use this sequence of commands in the future to add any asm file into a project.)**
- 6) **Next, you will open the main.asm file for editing. Double-click the file named main.asm to open it in the editing window. (Visual Studio users may see a popup dialog asking for the encoding method used in the asm file. just click the OK button to continue.)**

Main.asm is the file you will open every time because it is a template file and you will change it again and again for each program you will write. It will remain the same

Main.asm displays a message on screen (just to show you how assembly language looks like)

Tip: If the *Solution Explorer* window is not visible, select *Solution Explorer* from the *View* menu. Also, if you do not see *main.asm* in the Solution Explorer window, it might be hidden behind another window. To bring it to the front, click the *Solution Explorer* tab from the tabs shown along the bottom of the window.

You should see the following program (main.asm file) in the editor window it is an Assembly Language program you will change for each program you will write (so it acts as template file also)

What is in it will be clarified when you will start learning assembly from next class, here purpose is to show you a program and determines how to compile (For Assembly it is called Assemble/Build) it and Run it.

Note: Every program can be run in two ways. Run and Debug mode. You should be able to run it in both ways.

```

TITLE MASM Template                                     (main.asm)

; Description:
;
; Revision date:

INCLUDE Irvine32.inc

.data
myMessage BYTE "MASM program example",0dh,0ah,0

.code
main PROC
    call Clrscr

    mov edx,OFFSET myMessage
    call WriteString

    exit
main ENDP

END main

```

Later, we'll show you how to copy this program and use it as a starting point to write your own programs.

Activity 3:

The example shows how to build and run the program of Assembly. You will need to perform two steps

Build the Program

Now program is open in Editor, will build (assemble and link) the program. Select Build Project from the Build menu. In the Output window for Visual Studio at the bottom of the screen, you should see messages similar to the following, indicating the build progress:

```

1>----- Build started: Project: Project, Configuration: Debug Win32 -----
1>Assembling...
1>Assembling: .\main.asm
1>Linking...
1>Embedding manifest...
1>Build log was saved at "file://g:\masm\Project_sample\Debug\BuildLog.htm"
1>Project - 0 error(s), 0 warning(s)
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====

```

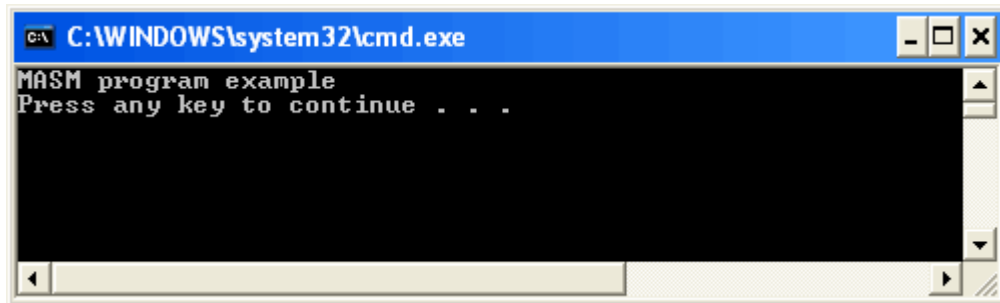
If you do not see these messages, the project has probably not been modified since it was last built. No problem--just select Rebuild Project from the Build menu.

Run the Program

Select Debug → Start without Debugging from the Debug menu.

Note: This button was added by you o start and now it will be used to run the program. If you do not see this button, Click Tools →Settings →Expert settings

The following console window should appear, although your window will be larger than the one shown here:



The "Press any key to continue..." message is automatically generated by Visual Studio.

Congratulations, you have just run your first Assembly Language program.

Press any key to close the Console window.

When you assembled and linked the project, a file named **Project.exe** was created inside the project's \Debug folder. This is the file that executes when you run the project. You can execute Project.exe by double-clicking its name inside Windows Explorer, but it will just flash on the screen and disappear. That is because Windows Explorer does not pause the display before closing the command window.

Activity 4:

Here it is demonstrated how to run the program in Debug mode and how to setup breakpoints and see Register and memory contents in Debug mode

Solution

There are two ways.

- 1) Just after the successful Build, Press F10 key from keyboard, an arrow will appear in front of the first line that will be run when you will press F10 button again. This process will continue and program will run line by line. At each line you should be able to see the results in
 - **Console window (if input/output was done)**
 - **Registers window(if results are in registers)**
 - **Memory window (if results are in variables)**
 - **Watch window (single point of contact to see variables/registers of your own choice)**

For our sample program, steps will be

- 1) **To begin stepping through your program in Debug mode, press the F10 key.**
- 2) **A yellow arrow will appear next to the first program statement (call Clrscr).The arrow indicates that the statement is next to be executed.**

- 3) **Press the F10 key (called Step Over) to execute the current statement. Continue pressing F10 until the program is about to execute the exit statement.**
- 4) **A small black window icon should appear on your Windows status bar. Open it and look at the contents of the Command window. You should see the words "MASM program example" in the window.**
- 5) **Press F10 one more time to end the program.**

2) Second way to run in Debug mode is by setting break point at any point inside the sample program. Then you will use the Visual C++ debugger to step through the program's execution one statement at a time. (coming next)

Registers

If you want to display the CPU registers any point when program is in Debug Mode , do the following:

Select Windows from the Debug menu. Select Registers from the drop-down list. The bottom window will display the register contents. Right click this window and check the item Flags to enable the display of conditional flags.

You can interrupt a debugging session at any time by selecting Stop Debugging from the Debug menu. You can do the same by clicking the blue square button on the toolbar. To remove a breakpoint from the program, click on the red dot so that it disappears.

Setting a Break-Point

If you set a breakpoint in a program, you can use the debugger to execute the program a full speed (more or less) until it reaches the breakpoint. At that point, the debugger drops into single-step mode.

- 1) **Click the mouse along the border to the left of the call WriteString statement. A large red dot should appear in the margin (it is breakpoint) or you can place the cursor to left and press F9, breakpoint will be inserted.F9is toggle key.**
- 2) **Select Start Debugging from the Debug menu. The program should run, and pause on the line with the breakpoint, showing the same Yellow arrow as before.**
- 3) **Press F10 until the program finishes (by method # 1)**

You can remove a breakpoint by clicking its red dot with the mouse. Take a few minutes to experiment with the Debug menu commands. Set more breakpoints and run the program again. For the time being, you can use the F11 key to step through the program in the same way the F10 key did.

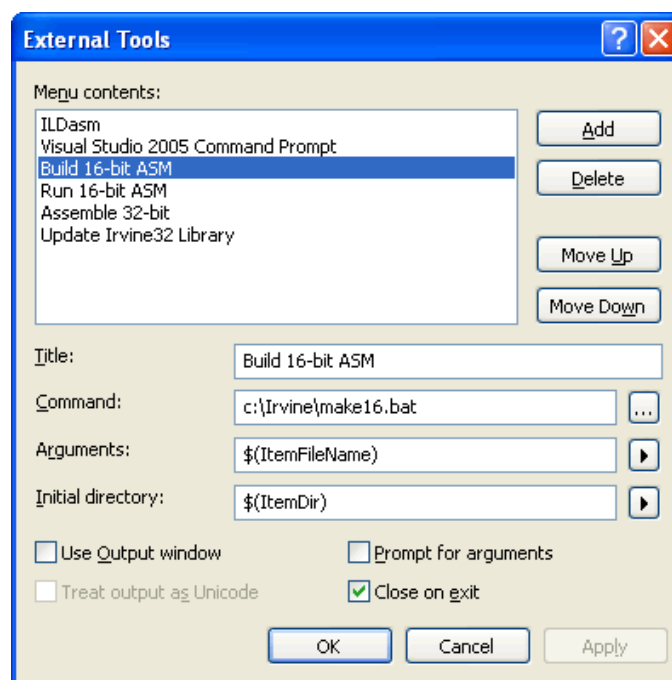
Activity 5:

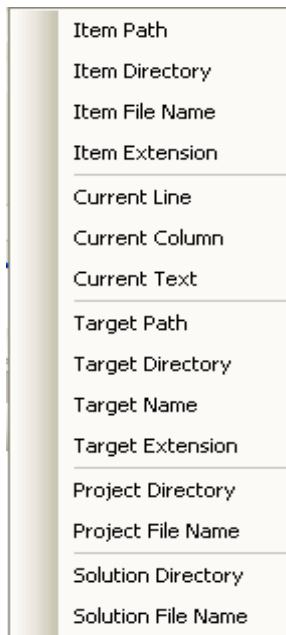
Here you will set the Visual Studio so that you would be able to build and run 16-bits applications.

Solution

1) Create and Build 16-Bit ASM command.

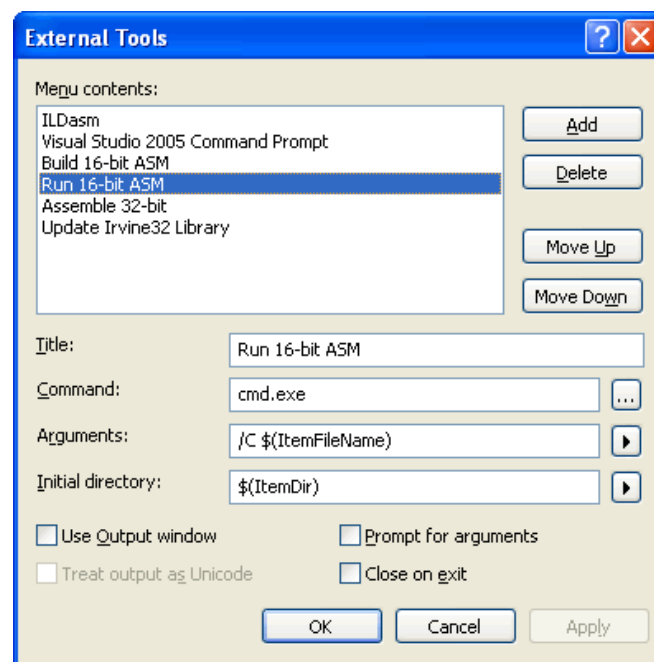
Click the Add button and fill in the Title, Command, Arguments, and Initial directory fields as shown in the screen snapshot. If you click the buttons with arrows on the right side of the Arguments and Initial directory fields, a convenient list appears. You can select an item without having to worry about spelling:





Click the Apply button to save the command and Create the Run 16-bit ASM Command.

Click the Add button again, and create a new command named Run 16-bit ASM:



Click the OK button to save the command and close the External Tools dialog.

3) Stage V(verify)

Home Activities: Perform the following at your home PC and write down the details of various options.

- 1) Assuming that our sample project is still open, select Project Properties from the Project menu. Expand the entry under Configuration Properties. Then expand the entry named Microsoft Macro Assembler. Write each option name and purpose you see here.
- 2) Click the entry named General under Microsoft Macro Assembler. Notice that the Include Paths option has been set to the c:\Irvine directory. This tells the assembler where to find files having a filename extension of ".inc". see the sample and describe details of each options listed here.
- 3) Next, select the Listing File entry, also in the Microsoft Macro Assembler group. Notice that the Assembled Code Listing File entry (shown below) has been assigned a macro name (starting with \$) that identifies the name of the source input file, with a file extension of .lst. So, if your program were named main.asm, the listing file would be named main.lst. See other options listed here and give details.
- 4) Find the Linker entry under Configuration Properties. Select the Input entry, and notice that two filenames have been added to the Additional Dependencies entry. The user32.lib file is a standard MS-Windows file. The irvine32.lib file is the link library file supplied with this book. There must be at least one space separating the file names:
 - Next, select Linker under Configuration Properties, and then select General. The Additional Library Directories option equals c:\Irvine, so the linker can find the Irvine32.lib library file. Also Select Linker under the Configuration Properties and select Debugging. Notice that the Generate DebugInfo option is set to Yes. Write down details of each option as well.

4) Stage2(assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

In this lab, student will know about the almost each and every data types assembly language support and their compatibility with high level programming languages.

Prior to the arrival of MASM, most assemblers provided very little capability for declaring and allocated complex data types. Generally, you could allocate bytes, words, and other primitive machine structures. You could also set aside a block of bytes. As high level languages improved their ability to declare and use abstract data types, assembly language fell farther and farther behind. Then MASM came along and changed all that[24]. Unfortunately, many long time assembly language programmers haven't bothered to learn the new MASM syntax for things like arrays, structures, and other data types. Likewise, many new assembly language programmers don't bother learning and using these data typing facilities because they're already overwhelmed by assembly language and want to minimize the number of things they've got to learn. This is really a shame because MASM data typing is one of the biggest improvements to assembly language since using mnemonics rather than binary opcodes for machine level programming.

Activity Outcomes:

This lab teaches you the following topics:

- **Understand data types in Assembly**
- **Storage considerations for each type**
- **Declaration of Integers, real, characters, strings, and arrays**
- **Various operators**
- **Data movement instructions**

Instructor Note:

As pre-lab activity, read Chapter 3 from the book (Assembly Language for X86 processors, KIP R. IRVINE., 7th Edition (2015), Pearson), and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

MASM defines intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type. The essential characteristic of each type is its size in bits: 8, 16, 32, 48, 64, and 80. Other characteristics (such as signed, pointer, or floating-point) are optional and are mainly for the benefit of programmers who want to be reminded about the type of data held in the variable. A variable declared as DWORD, for example, logically holds an unsigned 32-bit integer. In fact, it could hold a signed 32-bit integer, a 32-bit single precision real, or a 32-bit pointer. The assembler is not case sensitive, so a directive such as DWORD can be written as dword, Dword, dWord, and so on.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program that contains a definition of each data type listed in Table given below. Initialize each variable to a value that is consistent with its data type.

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer -D stands for double
SDWORD	32-bit signed integer. SD stands for signed
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Solution:

```
INCLUDE Irvine32.inc

.data

var1 BYTE 10h
var2 SBYTE -14
var3 WORD 2000h
var4 SWORD +2345
var5 DWORD 12345678h
var6 SDWORD -2342423
var7 FWORD 0
var8 QWORD 1234567812345678h
var9 TBYTE 1000000000123456789Ah
var10 REAL4 -1.25
var11 REAL8 3.2E+100
var12 REAL10 -6.223424E-2343


.code

main PROC

    exit

main ENDP

END main
```


Activity 2:

Write a program that defines symbolic constants for all of the days of the week. Create an array variable that uses the symbols as initializers.

Solution:

```
INCLUDE Irvine32.inc
```

```
Sunday    = 0
```

```
Monday    = 1
```

```
Tuesday   = 2
```

```
Wednesday = 3
```

```
Thursday  = 4
```

```
Friday    = 5
```

```
Saturday  = 6
```

```
.data
```

```
myDays BYTE Sunday, Monday, Tuesday, Wednesday,  
          Thursday, Friday, Saturday
```

```
.code
```

```
main PROC
```

```
    exit
```

```
main ENDP
```

```
END main
```

Activity 3:

Write a program that defines symbolic names for several string literals (characters between quotes). Use each symbolic name in a variable definition.

Solution:

```
INCLUDE Irvine32.inc
```

```
sym1 TEXTEQU <"System failure">
```

```
sym2 TEXTEQU <"Press any key to continue...">
```

```
sym3 TEXTEQU <"Insufficient user training">
```

```
sym4 TEXTEQU <"Please re-start the system">
```

```
.data
```

```
msg1 BYTE sym1
```

```
msg2 BYTE sym2
```

```
msg3 BYTE sym3
```

```
msg4 BYTE      sym4
```

```
.code
```

```
main PROC
```

```
    exit
```

```
main ENDP
```

```
END main
```

3) Stage v (verify)

Home Activities:

Home Task 1:

Define the following data in Assembly

```
Sort      BYTE      'y'          ; ASCII of y = 79H
value     WORD      25159          ; 25159D = 6247H
total     DWORD     542803535 ; 542803535D = 205A864FH
marks     DWORD     0, 0, 0, 0, 0, 0, 0, 0
value1    BYTE 'A'          ; character constant
value2    BYTE 0          ; smallest unsigned byte
value3    BYTE 255        ; largest unsigned byte
value4    SBYTE -128       ; smallest signed byte
value5    SBYTE +127       ; largest signed byte
value6    BYTE ?          ; uninitialized byte
```

Home Task 2:

Create the following menu by declaring a string in Assembly Language.

Checking Account

- 1) Create a new account.
- 2) Open an existing account
- 3) Credit the account
- 4) Debit the account
- 5) Exit

Home Task 3:

Create the following arrays in Assembly Language.

- 1) Array of ten integers.
- 2) Array of 100 characters.
- 3) Arrays of 4x3 integers
- 4) Array of 4x3x2 integers.

Home Task 4:

Implement each of the following declarations in assembly language:

- 1) `char initial;`
- 2) `char grade = 'B';`
- 3) `char x = 'P', y = 'Q';`
- 4) `int amount;`
- 5) `int count = 0;`
- 6) `int number = -396;`

4) Stage 2 (assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

In this lab, student will know about the almost each and every Data transfer instructions including memory to register and vice versa transfers, inter-register transfer, and memory to memory transfers. They will also be introduced instruction performing simple arithmetic operations.

Activity Outcomes:

This lab teaches you the following topics:

- **Understand how data is transferred inside the system during program run time.**
- **What transfer modes are allowed?**
- **Data transfer instructions.**
- **How arithmetic is performed inside CPU?**
- **How EFLAG register is affected by various arithmetic instructions?**
- **How to execute arithmetic instructions at Assembly programming level?**

Instructor Note:

As pre-lab activity, read Chapter 4 from the book (Assembly Language for X86 processors, KIP R. IRVINE., 7th Edition (2015), Pearson), and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

Assembly language consists of various types of instructions. Data transfer instructions are very important and these instructions moves data from registers to memory and vice versa. Data transfer instructions also allows data movements within the registers. These comes in various flavors like

MOV,MOVZX, MOVSX, XCHG etc.

MOV, a data transfer instruction, copies a source operand to a destination operand. The MOVZX instruction zero-extends a smaller operand into a larger one. The MOVSX instruction sign-extends a smaller operand into a larger register. The XCHG instruction exchanges the contents of two operands. At least one operand must be a register.

The following arithmetic instructions are important:

- **The INC instruction adds 1 to an operand.**
- **The DEC instruction subtracts 1 from an operand.**
- **The ADD instruction adds a source operand to a destination operand.**
- **The SUB instruction subtracts a source operand from a destination operand.**
- **The NEG instruction reverses the sign of an operand.**

When converting simple arithmetic expressions to assembly language, use standard operator precedence rules to select which expressions to evaluate first.

The following CPU status flags are affected by arithmetic operations:

- 1) The Sign flag is set when the outcome of an arithmetic operation is negative.
- 2) The Carry flag is set when the result of an unsigned arithmetic operation is too large for the destination operand.
- 3) The Parity flag indicates whether or not an even number of 1 bits occurs in the least significant byte of the destination operand immediately after an arithmetic or boolean instruction has executed.
- 4) The Auxiliary Carry flag is set when a carry or borrow occurs in bit position 3 of the destination operand.
- 5) The Zero flag is set when the outcome of an arithmetic operation is zero.
- 6) The Overflow flag is set when the result of a signed arithmetic operation is too large for the destination operand. In a byte operation, for example, the CPU detects overflow by exclusive-ORing the carry out of bit 6 with the carry out of bit 7.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program that uses addition and subtraction to set and clear the Zero and Sign flags. After each addition or subtraction instruction, insert the call DumpRegs statement to display the registers and flags. Using comments, explain how (and why) the Zero and Sign flags were affected by each instruction.

Solution:

```
INCLUDE Irvine32.inc

.data

.code

main PROC

    mov al,1                ; AL=1
    sub al,1                ; AL=0, SF=0, ZF=1
    call DumpRegs

    sub al,1                ; AL=-1, SF=1, ZF=0
    call DumpRegs

    add al,1                ; AL=0, SF=0, ZF=1
    call DumpRegs

    add al,1                ; AL=1, SF=0, ZF=0
    call DumpRegs

    exit
```

```
main ENDP
```

```
END main
```

Activity 2

:Write a program that uses addition and subtraction to set and clear the Overflow flag. After each addition or subtraction instruction, insert the call DumpRegs statement to display the registers and flags. Using comments, explain how (and why) the Overflow flag was affected by each instruction. Optional: include an ADD instruction that sets both the Carry and Overflow flags.

Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
.code
```

```
main PROC
```

```
    mov al,+127          ; AL=7Fh
    add al,1              ; AL=80h, OF=1
    call DumpRegs

    sub al,1              ; AL=7Fh, OF=1
    call DumpRegs

    sub al,1              ; AL=7Eh, OF=0
    call DumpRegs

    mov al,-128
    add al,-1
    call DumpRegs        ; AL=7Fh, OF=1
```


; Optional:

```
mov al,80h
add al,80h      ; AL=0, CF=1, OF=1
call DumpRegs
```

```
exit
```

```
main ENDP
```

```
END main
```

Activity 3

:Write instructions that use direct-offset addressing to move the four values in Uarray to the EAX, EBX, ECX, and EDX registers. When you follow this with a call DumpRegs statement, the following register values should display:

EAX=00001000 EBX=00002000 ECX=00003000 EDX=00004000

Next, write instructions that use direct-offset addressing to move the four values in Sarray to the EAX, EBX, ECX, and EDX registers. When you follow this with a call DumpRegs statement, the following register values should display:

EAX=FFFFFFFF EBX=FFFFFFFE ECX=FFFFFFFD EDX=FFFFFFFC

Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
Uarray WORD 1000h,2000h,3000h,4000h
```

```
Sarray SWORD -1,-2,-3,-4
```

```
.code
```

```
main PROC
```

```
; Move with zero extension:
```

```
movzx eax,Uarray
```

```
movzx ebx,Uarray+2
```

```
movzxecx,Uarray+4
movzxedx,Uarray+6
call    DumpRegs
```

; Move with sign extension:

```
movsxeax,Sarray
movsxebx,Sarray+2
movsxecx,Sarray+4
movsxedx,Sarray+6
call    DumpRegs
```

```
exit
```

```
main ENDP
```

```
END main
```

Activity 4:

Use a loop with indirect or indexed addressing to reverse the elements of an integer array in place. Do not copy the elements to any other array. Use the `sizeof`, `TYPE`, and `lengthof` operators to make the program as flexible as possible if the array size and type should be changed in the future. Optionally, you may display the modified array by calling the `DumpMem` method from the `Irvine32` library.

Solution

```
INCLUDE Irvine32.inc
```

```
.data
```

```
array DWORD 1,2,3,4,5,6,7,8,9
```

```
.code
```

```
main PROC
```

```

; point to the first and last array elements

mov     esi,0                                ; beginning of array
mov     edi,SIZEOF array - TYPE array        ; end of array
mov     ecx,LENGTHOF array / 2              ; loop (N / 2) times

```

```

; The loop swaps array elements from both ends, gradually
; moving towards the center element.

```

L1:

```

; exchange array[esi] with array[edi], using indexed addressing.

```

```

mov     edx,array[esi]                      ; temp = array[esi]
mov     eax,array[edi]                      ; array[esi] = array[edi]
mov     array[esi],eax
mov     array[edi],edx                      ; array[edi] = temp

add     esi,TYPE array                      ; first pointer moves forward
sub     edi,TYPE array                      ; second pointer backs up
loop    L1

```

```

; optional: display the array

```

```

mov     esi,OFFSET array
mov     ecx,LENGTHOF array
mov     ebx,TYPE array
call    DumpMem

```

```

exit

```

```

main ENDP

```

```

END main

```

3) Stage V(verify)

Home Activities:

Home Task 1:

Write a program that uses a loop to calculate the first seven values in the Fibonacci number sequence { 1,1,2,3,5,8,13 }. Place each value in the EAX register and display it with a call DumpRegs statement inside the loop.

Home Task 2:

Write a program that implements the following arithmetic expression:

$$EAX = -val2 + 7 - val3 + val1$$

In comments next to each instruction, write the hexadecimal value of EAX. Insert a call DumpRegs statement at the end of the program.

Home Task 3:

Write a program using the LOOP instruction with indirect addressing that copies a string from source to target, reversing the character order in the process.

If your program works correctly, you will see the following sequence of hexadecimal bytes on the screen when the program runs:

67 6E 69 72 74 73 20 65 63 72 75 6F 73 20 65 68

74 20 73 69 20 73 69 68 54

4) Stage 2(assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

In this lab, student will know about declaration of arrays and their processing element by element. Student will be introduced various modes to access array elements.

Other objective of this lab is to introduce students with instructions dealing with blocks of data whether it need to transferred or searched. Strings are also blocks of data as well as arrays so there would be efficient ways to work upon these data. Specific types of instructions exist which moves, search and scan blocks of data quickly as well as helping programmers to make code shorter in size, students will get an exposure to these set of instructions usually falls under the category of strings instructions.

Activity Outcomes:

This lab teaches you the following topics:

- **Understand how arrays are stored in memory?**
- **How little-endian order affects array storage in memory?**
- **How to access array elements in Assembly?**
- **Which addressing modes can be used to access array elements as one dimensional and as two dimensional?**
- **How strings can be stored and accessed via special instructions in Assembly.**
- **How row-major ordering of array storage differ from column major ordering of array storage?**

Instructor Note:

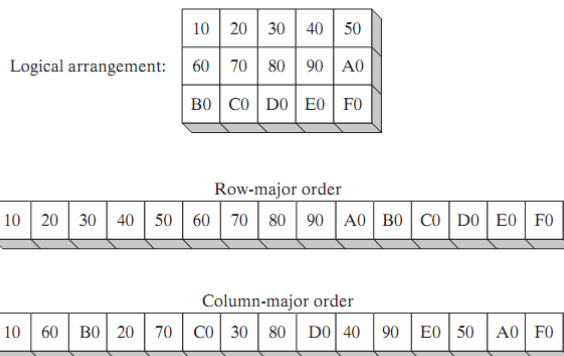
As pre-lab activity, read Chapter 4, 9 from the book (Assembly Language for X86 processors, KIP R. IRVINE., 7th Edition (2015), Pearson), and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

From an assembly language programmer's perspective, a two-dimensional array is a high-level abstraction of a one-dimensional array. High-level languages select one of two methods of arranging the rows and columns in memory: row-major order and column-major order. When row-major order (most common) is used, the first row appears at the beginning of the memory block. The last element in the first row is followed in memory by the first element of the second row. When column-major order is used, the elements in the first column appear at the beginning of the memory block. The last element in the first column is followed in memory by the first element of the second column.

If you implement a two-dimensional array in assembly language, you can choose either method. In this lab, we will use row-major order. If you write assembly language subroutines for a high-level language, you will follow the ordering specified in their documentation.



The x86 instruction set includes two operand types, base-index and base-index-displacement, both suited to array applications. We will examine both and show examples of how they can be used effectively.

String primitive instructions are unusual in that they require no register operands and are optimized for high-speed memory access. They are

- MOVS: Move string data
- CMPS: Compare strings
- SCAS: Scan string
- STOS: Store string data
- LODS: Load accumulator from string

Each string primitive instruction has a suffix of B, W, or D when manipulating bytes, words, and doublewords, respectively.

The repeat prefix REP repeats a string primitive instruction with automatic incrementing or decrementing of index registers. For example, when REPNE is used with SCASB, it scans memory bytes until a value in memory pointed to by EDI matches the contents of the AL register. The Direction flag determines whether the index register is incremented or decremented during each iteration of a string primitive instruction.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write an application that does the following: (1) fill a 32-bit array with 50 random integers; (2) Loop through the array, displaying each value, and count the number of negative values; (3) After the loop finishes, display the count.

Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
intArray SDWORD 50 DUP(?)
```

```
count DWORD 0
```

```
.code
```

```
main PROC
```

```
    call Randomize
```

```
    ; Fill the array with random values
```

```
    mov esi,OFFSET intArray    ; point to the array
```

```
    mov ecx,LENGTHOF intArray ; loop counter
```

```
L1:    call    Random32          ; EAX = random value
```

```
    call    WriteInt
```

```
    call    Crlf
```

```
    mov     [esi],eax
```

```
    add     esi,4
```

loop L1

; Search for negative values

mov esi,OFFSET intArray ; point to the array

mov ecx,LENGTHOF intArray ; loop counter

L2:

cmp dwordptr [esi],0 ; compare value to zero

jge L3 ; negative value?

inc count ; yes: add to count

L3:

add esi,4

loop L2

mov eax,count

call WriteDec

call Crlf

exit

main ENDP

END main

Activity 2:

Implement the following C++ code in assembly language, using the block-structured .IF and .WHILE directives. Assume that all variables are 32-bit signed integers:

```
int array[] = { 10,60,20,33,72,89,45,65,72,18};
```

```
int sample = 50;
```

```
int ArraySize = sizeof array / sizeof sample;
```

```
int index = 0;
```

```
int sum = 0;
```

```
while( index < ArraySize )
```

```
{
```

```
if( array[index] <= sample )
```

```
{
```

```
    sum += array[index];
```

```
}
```

```
    index++;
```

```
}
```

Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
array  DWORD 10,60,20,33,72,89,45,65,72,18
```

```
sample DWORD 50
```

```
arraySize = SIZEOF array / SIZEOF sample
```

```
index  DWORD 0
```

```
sum    DWORD 0
```

```
.code
```

```
main PROC
```

```

        mov esi,index                ; let ESI = index

.WHILE esi<arraySize                ; while index <arraySize

        mov eax, sample

        .IF array[esi*4] <= eax      ; must scale the index

        mov  eax,array[esi*4]        ; sum += array[index]
        add  sum, eax

        .ENDIF

        inc esi                      ; index++

.ENDW

; sum should equal 126

        mov eax, sum
        call WriteInt
        call Crlf

        exit

main ENDP
END main

```

Activity 3:

Write a program that defines symbolic names for several string literals (characters between quotes). Use each symbolic name in a variable definition.

Solution:

```
INCLUDE Irvine32.inc
```

```
sym1 TEXTEQU <"System failure">
```

```
sym2 TEXTEQU <"Press any key to continue...">
```

```
sym3 TEXTEQU <"Insufficient user training">
```

```
sym4 TEXTEQU <"Please re-start the system">
```

```
.data
```

```
msg1 BYTE sym1
```

```
msg2 BYTE sym2
```

```
msg3 BYTE sym3
```

```
msg4 BYTE      sym4
```

```
.code
```

```
main PROC
```

```
    exit
```

```
main ENDP
```

```
END main
```

Activity 4:

Write a program that defines and declare an array of 10 elements and then reverse the elements.

Solution

```
.386
.model flat, stdcall
.stack 4096

ExitProcess proto, dwExitCode: dword

.data
array DWORD 1, 5, 6, 8, 0Ah, 1Bh, 1Eh, 22h, 2Ah, 32h

.code
main PROC
    ; point to the first and last array elements
    mov     esi, 0                                ; beginning of
array
    mov     edi, sizeof array - type array        ; end of array
    mov     ecx, lengthof array / 2              ; loop (N / 2) times

    ; The loop swaps array elements from both ends, gradually
    ; moving towards the center element.

L1:
    ; exchange array[esi] with array[edi], using indexed addressing.

    mov     eax, array[esi]
    xchg     eax, array[edi]
    mov     array[esi], eax
```

```

add     esi,TYPE array           ; first pointer moves forward
sub     edi,TYPE array           ; second pointer backs up
loop    L1

```

; optional: display the array

```

mov     esi,OFFSET array
mov     ecx,LENGTHOF array
mov     ebx,TYPE array
call    DumpMem

```

```

call    ExitProcess,0

```

```

main endp

```

```

end main

```

Activity 5:

Using a loop and indexed addressing, write code that rotates the members of a 32-bit integer array forward one position. The value at the end of the array must wrap around to the first position. For example, the array.

[10,20,30,40] would be transformed into [40,10,20,30].

Solution:

```

ExitProcess proto

```

```

.data

```

```

array dword 10h,20h,30h,40h

```

```

arraySize = 4

```

```

.code

```

```

main proc

```

```

    mov rdi,3
    mov rsi,2

    mov eax,array[rdi*4]      ; save last value
    mov ecx,3

L1:
    mov edx,array[rsi*4]
    mov array[rdi*4],edx
    dec rsi
    dec rdi
    loop L1

    mov array[rdi*4],eax      ; store saved value in first position

    mov ecx,0                ; assign a process return code
    call ExitProcess          ; terminate the program

main endp
end

```

3) StageV(verify)

Home Activities:

Home Task 1:

Write a procedure named `calc_row_sum` that calculates the sum of a single row in a two-dimensional array of bytes, words, or doublewords. The procedure should have the following stack parameters: array offset, row size, array type, row index. It must return the sum in EAX. Use explicit stack parameters, not INVOKE or extended PROC. Write a program that tests your procedure with arrays of byte, word, and doubleword. Prompt the user for the row index, and display the sum of the selected row.

Home Task 2:

Google the Bubble Sort Algorithm implemented in any High level language and then implement it in Assembly.

Home Task 3:

Change the program in Home Task 2 such that add a variable to the BubbleSort that is set to 1 whenever a pair of values is exchanged within the inner loop. Use this variable to exit the sort before its normal completion if you discover that no exchanges took place during a complete pass through the array. (This variable is commonly known as an exchange flag.)

Home Task 4:

Searches backward through an array of integers for a matching value. Use special instructions to move, search, and copy a block of data/string from one memory location to another memory location

4) Stage 2 (assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

In this lab, student will know about the built-in functions and user defined functions that can be employed in programs to make development faster and make programs modular.

Activity Outcomes:

This lab teaches you the following topics:

- **Understand how to call built-in functions in Assembly?**
- **Understand how to create your own function in assembly**
- **Arguments passing methods.**
- **What are stack frames?**
- **How stack frame are created and destroyed?**
- **Recursive procedure and their working.**

Instructor Note:

As pre-lab activity, read Chapter 5 from the book (Assembly Language for X86 processors, KIP R. IRVINE., 7th Edition (2015), Pearson), and also as given by your theory instructor.

1) Stage J(Journey)

Introduction

A procedure is a unit of code designed to perform a particular sub-task of some main task. It is written out only once in some module, but can be used many times.

The advantages of using procedures are:

- Improves code readability because the code does not need to be laid out as one long sequence.
- Allows code to be reused because it can be written once and called from more than one place in the code.
- Allows tasks to be broken down into simpler components because procedures can be written for certain tasks (procedures can also call other procedures).
- Modular code facilitates modification.

A procedure is always enclosed within some code segment.

A procedure is defined as:

```
PROCEDURE_NAME PROC
```

```
.  
.   
.
```

```
PROCEDURE_NAME ENDP
```

where *PROCEDURE_NAME* is any valid identifier.

The PROC directive usually includes one of the operands NEAR or FAR. Example:

```
PROCEDURE_NAME PROC FAR
```

```
..
```

```
PROCEDURE_NAME ENDP
```

A NEAR procedure is defined in the same code segment from which it is called, and a FAR procedure is ordinarily defined in a separate code segment. Note if none of the operands NEAR or FAR follows the PROC directive, then the procedure is by default a NEAR procedure.

Built-In vs. User Defined

Students will be introduced to built-in procedures included in Irvine32 Library. They just need to call those procedures. Students will make their own procedures as well and call them at various places of their programs.

A procedure is invoked by a CALL instruction that can be direct or indirect. A direct procedure call has the format:

CALL PROCEDURE_NAME

In an indirect near procedure call, the operand for the CALL instruction is either a 16-bit general-purpose register or a memory word containing the offset address of the procedure.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program that displays the same string in four different colors, using a loop. Call the SetTextColor procedure from the book's link library. Any colors may be chosen, but you may find it easiest to change the foreground color.

Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
str1 BYTE "This line is displayed in color",0
```

```
.code
```

```
main PROC
```

```
    mov  eax, black + (white * 16)      ; black on white background
```

```
    mov  ecx,4                          ; loop counter
```

```
L1:    call  SetTextColor
```

```
    mov  edx, OFFSET str1
```

```

        call    WriteString
        call    Crlf
        add     eax,2                ; add 2 to foreground color
        loop   L1

        exit

main ENDP

END main

```

Activity 2:

Write a program that clears the screen, locates the cursor near the middle of the screen, prompts the user for two integers, adds the integers, and displays their sum.

Solution:

```

INCLUDE Irvine32.inc

.data
val1 SDWORD ?
val2 SDWORD ?
str1 BYTE "Enter first integer: ",0
str2 BYTE "Enter second integer: ",0
str3 BYTE "The sum is:          ",0

.code

main PROC

        call ClrScr

; Input the first integer
        mov  dh,10
        mov  dl,20
        call Gotoxy
        mov  edx, OFFSET str1

```

call WriteString

call ReadInt

mov val1,eax

; Input the second integer

mov dh,12

mov dl,20

call Gotoxy

mov edx, OFFSET str2

call WriteString

call ReadInt

add eax,val1

; Display the sum

mov dh,14

mov dl,20

call Gotoxy

mov edx, OFFSET str3

call WriteString

call WriteInt

call Crlf

call Crlf

exit

main ENDP

END main

Activity 3:

Create two procedures:

- 1) SetColor receives two BYTE parameters: forecolor and backcolor. It calls the SetTextColor procedure from the Irvine32 library.
- 2) WriteColorChar receives three byte parameters: char, forecolor, and backcolor. It displays a single character, using the color attributes specified in forecolor and backcolor. It calls the SetColor procedure, and it also calls WriteChar from the Irvine32 library. Both SetColor and WriteColorChar must contain declared parameters. Write a short test program that test both procedures. Be sure to create PROTO declarations for SetColor and WriteColorChar.

Solution:

```
INCLUDE Irvine32.inc
```

```
SetColor PROTO forecolor: BYTE, backcolor: BYTE
```

```
WriteColorChar PROTO char:BYTE,forecolor:BYTE, backcolor: BYTE
```

```
.data
```

```
.code
```

```
main PROC
```

```
    INVOKE WriteColorChar, 'A', white, blue
```

```
    INVOKE WriteColorChar, 'B', blue, white
```

```
    INVOKE WriteColorChar, 'C', green, black
```

```
    INVOKE WriteColorChar, 'D', yellow, gray
```

```
    INVOKE SetColor, lightGray, black
```

```
    call Crlf
```

```
    exit
```

```
main ENDP
```

```
WriteColorChar PROC USES eax,
```

```
char:BYTE,forecolor:BYTE, bgcolor: BYTE,
```

```
INVOKE SetColor, forecolor, bgcolor
```

```
mov al,char
```

```
call WriteChar
```

```
ret
```

```
WriteColorChar ENDP
```

```
SetColor PROC USES eax,
```

```
forecolor: BYTE, bgcolor: BYTE
```

```
movzx eax, bgcolor
```

```
shl eax,4
```

```
add al,forecolor
```

```
call SetTextColor
```

```
ret
```

```
SetColor ENDP
```

```
END main
```

Activity 4:

Write a wrapper procedure for the link library's DumpMem procedure, using stack parameters. The name can be slightly different, such as DumpMemory. The following is an example of how it should be called:

```
INVOKE DumpMemory,OFFSET array,LENGTHOF array,TYPE array
```

Write a test program that calls your procedure several times, using a variety of data types.

Solution

```
INCLUDE Irvine32.inc
```

```
DumpMemory PROTO, address: DWORD, units: DWORD, unitType:DWORD
```

```
.data
```

```
array DWORD 1,2,3,4,5,6,7,8,9,0Ah,0Bh,0Ch,0Dh,0Eh,0Fh
```

```
array1 WORD 0,9,8,7,6,5,4,3,2,1
```

```
array2 BYTE '1','2','3','4','5'
```

```
.code
```

```
main PROC
```

```
    INVOKE DumpMemory,OFFSETarray,LENGTHOFarray,TYPE array
```

```
    INVOKE DumpMemory, OFFSET array1,LENGTHOF array1,TYPE array1
```

```
    INVOKE DumpMemory, OFFSET array2,LENGTHOF array2,TYPE array2
```

```
    exit
```

```
main ENDP
```

```
;-----
```

```
DumpMemory PROC USES esi ebx ecx,
```

```
    address: DWORD,                ; starting address
```

```
    units: DWORD,                  ; number of units
```

```
    unitType:DWORD                 ; unit size
```

```
;
```

```
; Wrapper procedure for the link library's DumpMem procedure
```

```
; Receives: nothing
```

```
; Returns: nothing
```

;-----

```
mov    esi, address
mov    ecx, units
mov    ebx, unitType
call   DumpMem
ret
```

DumpMemory ENDP

END main

Activity 5:

Write a non-recursive version of the Factorial procedure that uses a loop. Write a short program that interactively tests your Factorial procedure. Let the user enter the value of n. If overflow occurs in your loop when calculating each factorial value, your program should display an error message. If no overflow occurs, display the calculated factorial.

Solution:

```
INCLUDE Irvine32.inc
```

```
.data
```

```
msgInput  BYTE "Enter the value of n to calculate "
```

```
          BYTE "the factorial (-1 to quit): ",0
```

```
msgOutput BYTE "The factorial is: ",0
```

```
.code
```

```
main PROC
```


L1:

```
mov  edx,OFFSET msgInput      ; message to display
call WriteString
call ReadInt                  ; get an integer from the user
call Crlf
cmp  eax, 0                   ; is n less than 0
jl   quit

call FactorialIterative       ; EAX = factorial(n)
jc   failed                   ; CF=1 if overflow occurred

push eax                      ; save the factorial
mov  edx,OFFSET msgOutput
call WriteString
pop   eax                     ; restore the factorial
call WriteDec                 ; and display it
```

failed:

```
call Crlf
call Crlf
loop L1

quit: exit
```

main ENDP

;-----

FactorialIterative PROC USES ecx edx

;

; Calculates a factorial non-recursively

```
; Receives: eax = value of n to calculate factorial
; Returns: if the result is valid, CF=0 and eax = calculated factorial
; If overflow occurred, CF=1 and EAX is unknown.
```

```
;-----
```

```
.data
```

```
factorialError BYTE "Error: Calculated value cannot "
                BYTE "fit into 32 bits",0
```

```
.code
```

```
.IF eax == 0 || eax == 1                ; special cases
    mov eax, 1                          ; factorial == 1
    jmp end_factorial                  ; quit procedure
.ENDIF
    mov ecx, eax                        ; ecx = counter
```

```
Factorial_loop:
```

```
    dec ecx                            ; ecx = n - 1
    mul ecx                            ; eax = n * (n - 1)
    jc error                           ; exit if overflow occurred
```

```
    cmp ecx, 1                        ; is counter > 1?
    ja Factorial_loop                 ; if true, keep looping
    jmp end_factorial
```

```
error:
```

```
    mov edx,OFFSET factorialError
    call WriteString
    stc                              ; CF=1 indicates error
```

```
end_factorial:
```

```
    ret
```

```
FactorialIterative ENDP
```

```
END main
```

3) Stage V(verify)

Home Activities:

Home Task 1:

Write a program that compares the runtime speeds of both the recursive Factorial procedure and the non-recursive Factorial procedure written for the preceding programming exercise. Use the GetMseconds procedure from the book's link library to measure and display the number of milliseconds required to call each Factorial procedure several thousand times in a row.

NOTE: A nested loop is used because the running time of a single loop is less than a millisecond

Home Task 2:

Write a recursive implementation of Euclid's algorithm for finding the greatest common divisor (GCD) of two integers. Note: we will only test this procedure with nonnegative integers.

Home Task 3:

Write a procedure named ShowParams that displays the address and hexadecimal value of the 32-bit parameters on the runtime stack of the procedure that called it. Input to the procedure will be a single integer that indicates the number of parameters to display.

Home Task 4:

Write a program that draws an 8 X 8 chess board, with alternating colored and white squares. Each square is 2 characters wide and 1-character high. You can use the SetTextColor and Gotoxy procedures from the Irvine32 library. Avoid the use of global variables, and use declared parameters in all procedures. Use short procedures that are focused on a single task.

Home Task 5:

Write a program that draws an 8 X 8 chess board, with alternating colored and white squares. Each square is 2 characters wide and 1-character high. You can use the SetTextColor and Gotoxy procedures from the Irvine32 library. Avoid the use of global variables, and use declared parameters in all procedures. Use short procedures that are focused on a single task.

Home Task 6:

Write a program that draws an 8 X 8 chess board, with alternating colored and white squares. Each square is 2 characters wide and 1 character high. You can use the SetTextColor and Gotoxy procedures from the Irvine32 library. Every 500 milliseconds, change the color of the colored squares and redisplay the board. Continue until you have shown the board 16 times, using all possible 4-bit background colors. (The white squares remain white throughout.)

4) Stage 2(assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

Objective of this lab is to introduce students to work with Boolean operators i.e. AND, OR, NOT and XOR and to know the impact of these operators on the destination operand. Further, TEST and CMP instructions are introduced which are the advance comparison instruction based on AND instruction. They will know various types of operation those can be performed on bits to make them set, clear, mask, unmask etc. They will know applications of bit manipulation as well.

Activity Outcomes:

This lab teaches you the following topics:

- **Students will know that programmer in assembly can efficiently manipulate bits to do some processing more effective like knowing the bit combination.**
- **They will know interesting property of bit level instruction to set, clear and toggle bits and they will be able to apply those in relevant applications.**

Instructor Note:

The TEST instruction performs an implied AND operation between each pair of matching bits in two operands and sets the Sign, Zero, and Parity flags based on the value assigned to the destination operand. The only difference between TEST and AND is that TEST does not modify the destination operand. The TEST instruction permits the same operand combinations as the AND instruction.

At the heart of any Boolean expression is some type of comparison. In Intel assembly language we use the CMP instruction to compare integers. The CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand.

- **Neither operand is modified:**
- **CMP destination, source**
- **CMP uses the same operand combinations as the AND instruction.**

When two unsigned operands are compared, the Zero and Carry flags indicate the following relations between operands:

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

When two signed operands are compared, the Sign, Zero, and Overflow flags indicate the following relations between operands:

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

1) Stage J(Journey)

Introduction

The AND, OR, XOR, NOT, and TEST instructions are called bitwise instructions because they work at the bit level. Each bit in a source operand is matched to a bit in the same position of the destination operand:

- The AND instruction produces 1 when both input bits are 1.
- The OR instruction produces 1 when at least one of the input bits is 1.
- The XOR instruction produces 1 only when the input bits are different.
- The NOT instruction reverses all bits in a destination operand.

Operation	Description
AND	Boolean AND operation between a source operand and a destination operand.
OR	Boolean OR operation between a source operand and a destination operand
XOR.	Boolean exclusive-OR operation between a source operand and a destination operand
NOT	Boolean NOT operation on a destination operand.

Encryption is a process that encodes data, and decryption is a process that decodes data. The XOR instruction can be used to perform simple encryption and decryption.

Here are the selected Boolean Operations that the students will be introduced in Lab via practical examples.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

How AND instruction is used to convert characters to uppercase?

Solution:

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

main PROC

 mov eax, 0

 mov al, 'a'

 call dumpregs

 call crlf

 call writebin

 call crlf

 call writechar

 call crlf

 and al, 11011111b

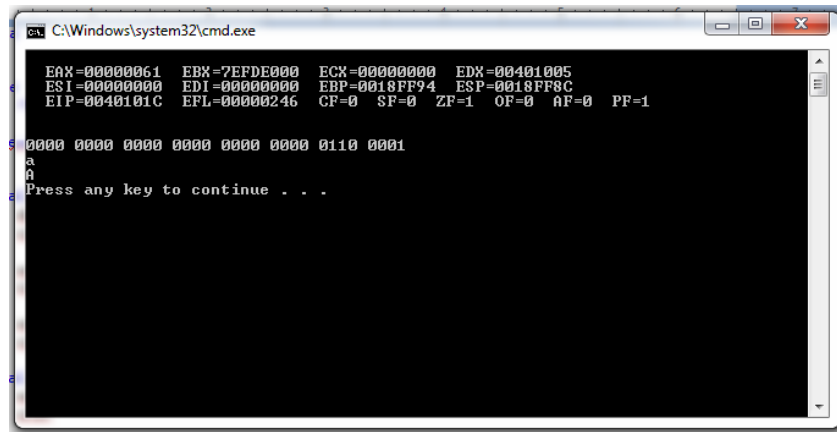
 call writechar

 call crlf

exit

main ENDP

END main



Activity 2:

How OR instruction is used to make ASCII code of input digits 0 to 9?

Solution:

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

main PROC

mov eax, 0

mov al, 1

call dumpregs

call crlf

call writebin

call crlf

call writechar

call crlf

or al, 00110000b

call writechar

```

        call crlf
        call dumpregs
    exit
main ENDP
END main

```

Activity 3:

TEST Instruction: No Flag affected as the output is nor zero neither using sign bit

Solution:

```

TITLE Lab Boolean Operators
; Author: Ashfaq Hussain Farooqi
INCLUDE Irvine32.inc

.data

.code

main PROC

    mov eax, 0
    mov ebx, 5
    add ebx, 6
    mov al, 00110110b
    call dumpregs
    test al, 10001100b
    call dumpregs

    exit
main ENDP
END main

```


Activity 4:

TEST Instruction: Flag affected as the output is not zero but setting sign bit

Solution:

```
TITLE Lab Boolean Operators
; Author: Ashfaq Hussain Farooqi
INCLUDE Irvine32.inc

.data
.code

main PROC
    mov eax, 0
    mov ebx, 5
    add ebx, 6
    mov al, 10101010b
    call dumpregs
    test al, 10010000b
    call dumpregs

    exit
main ENDP
END main
```

Activity 5:

TEST Instruction: Flag affected as the output is zero

Solution:

```
TITLE Lab Boolean Operators
; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data
```

```

.code
main PROC

    mov eax, 0

    mov ebx, 5

    add ebx, 6

    mov al, 10101010b

    call dumpregs

    test al, 01010000b

    call dumpregs

    exit

main ENDP

END main

```

Activity 6:

CMP Instruction (Unsigned): D==S; Zero Flag affected

Solution:

```

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

main PROC

    mov ebx, 5

    add ebx, 6

    call dumpregs

    mov eax, 5

    cmp eax, 5

    call dumpregs

    exit

main ENDP

```

END main

Activity 7:

CMP Instruction (Unsigned): $D > S$; No Flag affected

Solution:

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

main PROC

 mov ebx, 5

 add ebx, 6

 call dumpregs

 mov eax, 5

 cmp eax, 4

 call dumpregs

 exit

main ENDP

END main

Activity 8:

CMP Instruction (Unsigned): $D < S$; Carry Flag affected

Solution:

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

```

main PROC

    mov ebx, 5
    add ebx, 6
    call dumpregs
    mov eax, 5

cmp eax, 6
call dumpregs
    exit
main ENDP
END main

```

Activity 9:

CMP Instruction (Signed): D==S; Zero Flag affected

Solution:

Solution:

```

TITLE Lab Boolean Operators
; Author: Ashfaq Hussain Farooqi
INCLUDE Irvine32.inc

.data

.code

main PROC

    mov ebx, 5
    add ebx, 6
    call dumpregs
    mov eax, -10
    cmp eax, -10
    call dumpregs

    exit
main ENDP

```

END main

Activity 10:

CMP Instruction (Signed): D>S; SF = OF

Solution:

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

main PROC

 mov ebx, 5

 add ebx, 6

 call dumpregs

 mov eax, -10

 cmp eax, -15

 call dumpregs

exit

main ENDP

END main

Activity 11:

CMP Instruction (Signed): D < S; No Flag affected

Solution:

TITLE Lab Boolean Operators

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

main PROC

```

    mov ebx, 5
    add ebx, 6
    call dumpregs
    mov eax, -10
    cmp eax, -5
    call dumpregs

exit
main ENDP
END main

```

3) Stage v (verify)

Home Activities:

Activity 1:

Take a string from the user and display it in Capital letters.

Activity 2:

Take a string from the user and display it in Small letters.

Activity 3:

Take a string from the user called it a plain text. Convert it into the cipher text assuming key as 78h. display the cipher text. Again convert the cipher text into the original text. Note: You may use XOR command to cipher or encipher the text BYTE by BYTE.

4) Stage2(assess)

Demonstrate the home activities and Viva voce

Statement Purpose:

Objective of this lab is to introduce students the philosophy and ways to control transfer based on condition or unconditionally at various points in programs. They will make familiar with condition as well unconditional JUMP instructions based on any type of condition whatsoever.

Activity Outcomes:

This lab teaches you the following topics:

- **Students will know that transferring control based on conditions is very important to write real world applications as we have to do transfer our direction based on some decisions in real life.**
- **They will know that they can transfer control on various types of decision e.g. something is greater or smaller than other etc.**
- **They will know how processor make decisions and how to give him a program based on decisions.**

Instructor Note:

The circuits in the CPU can perform simple decision-making based on the current state of the processor. For the 8086/8088 processors, the processor state is implemented as nine bits, called flags, in the Flags register. Each decision made by the 8086/8088 CPU is based on the values of these flags.

The flags are classified as either status flags or control flags. There are 6 status flags: Carry flag (CF), Parity flag (PF), Auxiliary carry flag (AF), Zero flag (ZF), Sign flag (SF), and Overflow flag (OF). There are 3 control flags: Trap flag (TF), Interrupt flag (IF), and Direction flag (DF).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

The Flags register: Bits 8, 9, and 10 are the Control flags.

The Status flags reflect the result of some instructions executed by the processor. For example, when a subtraction operation results in a zero, the Zero flag (ZF) is set to 1 (true). The Control flags enable or disable certain operations of the processor. For example, if the Interrupt flag (IF) is cleared to 0, inputs from the keyboard are ignored by the processor.

Most 8086/8088 instructions can be classified into three categories:

- **Instructions that modify one or more Status flags. (Status flags modifying instructions).**
- **Instructions that modify one or more Control flags. (Control flags modifying instructions).**
- **Instructions that do not modify any flag.**

1) Stage J(Journey)

Introduction

FLOW CONTROL INSTRUCTIONS

- The JMP (Jump), CALL, RET, and IRET instructions transfer control unconditionally to another part of the program.
- The conditional jump instructions, except JCXZ and JECXZ, transfer control to another part of the program each depending on one or more Status flag settings. These instructions are of the form Jcondition, where condition is represented by one, two, or three letters.
- JCXZ and JECXZ transfer control to another part of the program if CX = 0 and ECX = 0 respectively.
- The LOOP instruction decrements CX and transfer control to the beginning of its loop if CX ≠ 0. If CX = 0 before the LOOP instruction, it is decremented to -1 at the end of the first iteration of the loop. This -1 is treated as the unsigned number 65535, thus the loop will iterate 65536 times.
- The conditional loop instructions LOOPE and LOOPZ, which are equivalent, decrement CX and transfer control to the beginning of their loops if CX > 0 and ZF = 1. If CX ≤ 0 before the loop, the loop is executed once.
- The conditional loop instructions LOOPNE and LOOPNZ, which are equivalent, decrement CX and transfer control to the beginning of their loops if CX > 0 and ZF = 0. If CX ≤ 0 before the loop, the loop is executed once.

THE JMP INSTRUCTION (Unconditional Jump)

The JMP instruction, whose syntax is:

JMP target

unconditionally transfers control to the target location. There are two major categories of JMP instructions:

- Intra-segment jump: A jump to a statement in the same code segment.
- Inter-segment or far jump: A jump to a statement in a different code segment.

Intra-segment jumps simply change the value in the IP register. Inter-segment jumps change both CS and IP.

Direct and Indirect jumps

A jump can either be direct or indirect. In a direct jump the address of the target is obtained from the instruction itself, i.e., the operand of the JMP instruction is a label. Example:

JMP L2

In an indirect jump the address of the target is obtained from a 16-bit or a 32-bit variable or a general-purpose register referenced by the JMP instruction.

CONDITIONAL JUMP INSTRUCTIONS

Conditional jumps are of the general form:

Jcondition StatementLabel

where condition is one, two, or three letters the StatementLabel must in the current code segment and should be within -128 to +127 bytes from the conditional jump instruction.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Jump based on Equality or Zero Flag: Take two numbers from the user and display whether number 2 is equal or not to number 1.

Solution:

Note: Replace JZ with JE and there will be no change in the output as both works similarly.

TITLE Lab Flow Control and Conditional Jump Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

msg1 BYTE "2nd Number is equal to 1st Number ",0

msg2 BYTE "2nd Number is not equal to 1st Number ",0

msg01 BYTE "Enter 1st Number",0

msg02 BYTE "Enter 2nd Number",0

.code

main PROC

mov edx, OFFSET msg01

call writestring

call crlf

call readint

```

mov ebx, eax
mov edx, OFFSET msg02
call writestring
call crlf
call readint
cmp eax, ebx
jz EQUAL

```

```

mov edx, OFFSET msg2
call writestring
call crlf
jmp LAST
EQUAL:
mov edx, OFFSET msg1
call writestring
call crlf
LAST:

```

```
exit
```

```
main ENDP
```

```
END main
```

Activity 2:

Jump based for Signed Number: Take two numbers from the user and display whether number 2 is greater or not from number 1.

Solution:

Note: Replace JG with JNBE and there will be no change in the output as both works similarly.

TITLE Lab Flow Control and Conditional Jump Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

```

msg1 BYTE "2nd Number is greater than 1st Number ",0
msg2 BYTE "2nd Number is less than 1st Number ",0
msg01 BYTE "Enter 1st Number",0
msg02 BYTE "Enter 2nd Number",0

.code
main PROC
    mov edx, OFFSET msg01
    call writestring
    call crlf
    call readint
    mov ebx, eax
    mov edx, OFFSET msg02
    call writestring
    call crlf
    call readint
    cmp eax, ebx
    jg Greater
    mov edx, OFFSET msg2
    call writestring
    call crlf
    jmp LAST
Greater:
    mov edx, OFFSET msg1
    call writestring

    call crlf
LAST:

    exit
main ENDP
END main

```

Activity 3:

Jump based on Unsigned Value: Take two numbers from the user and display whether number 2 is greater or not from number 1.

Solution:

Note: Replace JA with JNBE and there will be no change in the output as both works similarly.

TITLE Lab Flow Control and Conditional Jump Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

msg1 BYTE "2nd Number is greater than 1st Number ",0

msg2 BYTE "2nd Number is less than 1st Number ",0

msg01 BYTE "Enter 1st Number",0

msg02 BYTE "Enter 2nd Number",0

.code

main PROC

mov edx, OFFSET msg01

call writestring

call crlf

call readint

mov ebx, eax

mov edx, OFFSET msg02

call writestring

call crlf

call readint

cmp eax, ebx

ja Greater

mov edx, OFFSET msg2

call writestring

call crlf

```

        jmp LAST
Greater:
        mov edx, OFFSET msg1
        call writestring

        call crlf
LAST:

exit
main ENDP
END main

```

3) Stage v (verify)

Home Activities:

Activity 1:

Take 3 numbers will be entered by the user. Tell which number is the greatest. Write its Code.

Solution:

TITLE Lab Flow Control and Conditional Jump Instructions

; Author: Ashfaq Hussain Farooqi

include irvine32.inc

.data

num1 dword ?

num2 dword ?

num3 dword ?

res1 byte "Number 1 is greatest",0

res2 byte "Number 2 is greatest",0

res3 byte "Number 3 is greatest",0

.code

main proc

call readint

mov num1,eax

```

call readint
mov num2,eax
call readint
mov num3,eax

mov eax,num1
cmpeax,num2
jg con2
jmp num22
con2:
cmpeax,num3
jg result1
num22:
mov eax,num2
cmpeax,num1
jg con3
jmp result3
con3:
cmpeax,num3
jg result2
jmp result3
result1:      ;output
call crlf
mov edx,offset res1
call writestring
jmp endd
result2:      ;output
call crlf
mov edx,offset res2
call writestring

```

```

        jmp endd

result3:                ;output
        call crlf
        mov edx,offset res3
        call writestring
        jmp endd

endd:
        call crlf

exit
main endp
end main

```

Activity 2:

Take a number entered by the user. Tell whether the number is:

- **+ EVEN**
- **-EVEN**
- **+ODD**
- **-ODD**

Write its Code.

Solution:

TITLE Lab Flow Control and Conditional Jump Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

```

msg BYTE "The number is + even",0
msg1 BYTE "The number is + odd",0
msg2 BYTE "The number is - even",0
msg3 BYTE "The number is - odd",0
var DWORD ?

```

```

.code
main PROC
mov eax, 0
    call readint
    test ax, 8000h
    jz Positive
    test ax, 0001h
    jnz odd
    mov edx, offset msg2
    call writestring
    call crlf
    jmp last
odd:
    mov edx, offset msg3
    call writestring
    jmp last
positive:
    test ax, 0001h
    jz even1
    mov edx, offset msg1
    call writestring
    jmp last
even1:
mov edx, offset msg
call writestring
last:
call crlf
    exit
main ENDP
END main

```


4) Stage a2 (assess)

Assignment:

Problem: Find the greater value of the same indexes of two arrays (array1 and array2). Subtract the greater value from the smaller value. Store that value in the same index of array three as shown in the following table.

Array1	5	10	15	20	25
Array2	10	5	10	25	10
Array3 (Output)	5	5	5	5	15

Statement Purpose:

Objective of this lab is to introduce student's selection and iteration structures. Students will translate high level language selection and iteration constructs into Assembly to make their learning quicker.

Activity Outcomes:

This lab teaches you the following topics:

- **Students will know that they can write almost any type of program in assembly that is written in high level languages.**
- **After learning all control structures, now they will be using them in any type of application.**
- **They will be able to translate any High level language program almost into assembly up till this lab.**

Instructor Note:

Here we will introduce control structures; you would be familiar from introductory programming classes, these are the three types

- **Selection-----If Else etc**
- **Iteration-----While Do etc**
- **Sequence(by default)**

1) Stage J(Journey)

Introduction

IF-ENDIF Statement

Note: In what follows it is assumed that OP, OP1 and OP2 are signed operands such that in any comparison there is no assembly error, and statement' is the assembly language translation of statement.

IF(OP1 = OP2)THEN

 statement1

 statement2

ENDIF

can be translated to:

```
CMP OP1 , OP2
JE NEXT_LABEL
JMP END_IF
```

NEXT_LABEL:

```
statement1'
statement2'
```

END_IF:

however, if the condition is reversed a better solution is obtained:

```
CMP OP1 , OP2
JNE END_IF
```

```
statement1'
statement2'
```

END_IF:

2) Stage a1 (apply)

Lab Activities:

Activity 1: Convert in MASM generated code.

IF((AL > OP1) OR (AL >= OP2))THEN

Statement...

ENDIF

Solution:

```
CMP AL , OP1
```

```
JG L1
```

```
CMP AL , OP2
```

```
JGE L1
```

```
JMP L2
```

```
L1: statement'
```

```
L2:
```

however, a better solution is obtained by reversing each of the conditions and jumping to an END_IF label when any reversed condition is true:

```
CMP AL , OP1
```

```
JNG END_IF
```

```
CMP AL , OP2
```

```
JNGE END_IF
```

```
statement'
```

```
END_IF:
```

Activity 2: **Convert in MASM generated code.**

```
IF(OP1 <= OP2)THEN
```

```
    statement1
```

```
    statement2
```

```
ELSE
```

```
    statement3
```

```
ENDIF
```

Solution:

```
CMP OP1 , OP2
```

```
JLE L1
```

```
statement3'
```

```
JMP END_IF
```

```
L1:  statement1'
```

```
    statement2'
```

```
END_IF:
```

however, if the condition is reversed the following solution is obtained:

```
CMP OP1 , OP2
```

```
JNLE L1
```

```
statement1'
```

```
statement2'
```

```
JMP END_IF
```

```
L1:    statement3'
```

```
END_IF:
```

Activity 3:

Convert in MASM generated code.

```
WHILE(OP1 < OP2)DO
```

```
    statement1
```

```
    statement2
```

```
ENDWHILE
```

can be translated to:

```
START: CMP OP1 , OP2
```

```
JL  WHILE_BODY
```

```
JMP END_WHILE
```

```
WHILE_BODY:
```

```
statement1'
```

```
statement2'
```

```
JMP START
```

```
END_WHILE:
```

however, a better solution is obtained by reversing the condition and exiting the loop when the reversed condition becomes true:

```
START: CMP OP1 , OP2
```

```
JNL END_WHILE
```

```
statement1'
```

```
statement2'
```

```
JMP START
```

```
END_WHILE:
```

DO-WHILE (do, while loop condition is true)

```
DO
```

```
    statement1
```

```
    statement2
```

```
WHILE(OP1 < OP2)
```

can be translated to:

START: statement1'

statement2'

CMP OP1 , OP2

JL START

REPEAT- UNTIL LOOP (Repeat until the loop condition becomes true)

REPEAT

statement1

statement2

statement3

UNTIL(OP1 = OP2) OR (OP1 > OP3)

can be translated to:

REPEAT:

statement1'

statement2'

statement3'

CMP OP1 , OP2

JE END_REPEAT

CMP OP1 , OP3

JG END_REPEAT

JMP SREPEAT

END_REPEAT:

however, a better solution is obtained by reversing the last condition:

REPEAT:

statement1'

statement2'

statement3'

```

CMP OP1 , OP2
JE END_REPEAT
CMP OP1 , OP3
JNG SREPEAT
END_REPEAT:

```

3) Stage v (verify)

Home Activities:

Activity # 1:

Write MASM generate code for following:

```

for( i = 3 ; i<= 40 ; i++ )
{
    statement1 ;
    statement2 ;
    statement3 ;
}

```

Solution # 1:

```

MOV BL , 3

START: CMP BL , 40

JA END_FOR

statement1'

    statement2'

    statement3'

INC BL

JMP START

END_FOR:

```

Note: We assume that BL is not modified in any of the statements: statement1', statement2', and statement3'

Activity # 2:

Write MASM generate code for following:

```
switch(OP)
{
    case const1:  statement1 ;
                break ;
    case const2:  statement2 ;
                break ;
    . . .
    case constN:  statementN ;
                break ;
    default:      statementN+1 ;
}
```

Solution # 2:

```
CMP OP , const1
```

```
JE L1
```

```
CMP OP , const2
```

```
JE L2
```

```
. . .
```

```
CMP OP , constN
```

```
JE LN
```

```
statementN+1'
```

```
JMP END_SWITCH
```

```
L1:  statement1'
```

```
JMP END_SWITCH
```

```
L2:   statement2'
```

```
JMP END_SWITCH
```

```
. . .
```

```
LN:  statementN'
```

```
END_SWITCH:
```


Activity # 3:

Control structures are every important to write any program whether in high or low level programming language. Your home task is to study section 6.5 and 6.6 of the chapter 6 to get detailed exposure to control structures and try to implement Finite State Machines given in section 6.6 in Assembly using these control structure.

1) Stage a2 (assess)

Demonstrate Activity # 3 and Viva Voce

Statement Purpose:

Objective of this lab is to introduce students with some new types of instructions known as shift and rotate. Student must be able to shift and rotate any type of data at bit-level along with understanding their applications.

Activity Outcomes:

This lab teaches you the following topics:

- **Students will know that shifts and rotate instructions are very helpful in performing faster multiplication and divisions.**
- **They will also learn that how images are manipulated or distorted by shifting or rotating bits via various different ways (Shift Left, Shift Right, and Rotate Left/Right. Rotate through Carry Left/Right etc)**

Instructor Note:

Shift instructions are among the most characteristic of assembly language. To shift a number means to move its bits right or left. The SHL (shift left) instruction shifts each bit in a destination operand to the left, filling the lowest bit with 0. One of the best uses of SHL is for performing high-speed multiplication by powers of 2. Shifting any operand left by n bits multiplies the operand by 2^n . The SHR (shift right) instruction shifts each bit to the right, replacing the highest bit with a 0. Shifting any operand right by n bits divides the operand by 2^n .

SAL (shift arithmetic left) and SAR (shift arithmetic right) are shift instructions specifically designed for shifting signed numbers.

The ROL (rotate left) instruction shifts each bit to the left and copies the highest bit to both the Carry flag and the lowest bit position. The ROR (rotate right) instruction shifts each bit to the right and copies the lowest bit to both the Carry flag and the highest bit position.

The RCL (rotate carry left) instruction shifts each bit to the left and copies the highest bit into the Carry flag, which is first copied into the lowest bit of the result. The RCR (rotate carry right) instruction shifts each bit to the right and copies the lowest bit into the Carry flag. The Carry flag is copied into the highest bit of the result.

The SHLD (shift left double) and SHRD (shift right double) instructions, available on x86 processors, are particularly effective for shifting bits in large integers.

1) Stage J(Journey)

Introduction

Shift instructions are among the most characteristic of assembly language. To shift a number means to move its bits right or left. The SHL (shift left) instruction shifts each bit in a destination operand to the left, filling the lowest bit with 0. One of the best uses of SHL is for performing high-speed multiplication by powers of 2. Shifting any operand left by n bits multiplies the

operand by 2^n . The SHR (shift right) instruction shifts each bit to the right, replacing the highest bit with a 0. Shifting any operand right by n bits divides the operand by 2^n .

- Students will know that shifts and rotate instructions are very helpful in performing faster multiplication and divisions.
- They will also learn that how images are manipulated or distorted by shifting or rotating bits via various different ways (Shift Left, Shift Right, and Rotate Left/Right. Rotate through Carry Left/Right etc)

2) Stage a1 (apply)

Lab Activities:

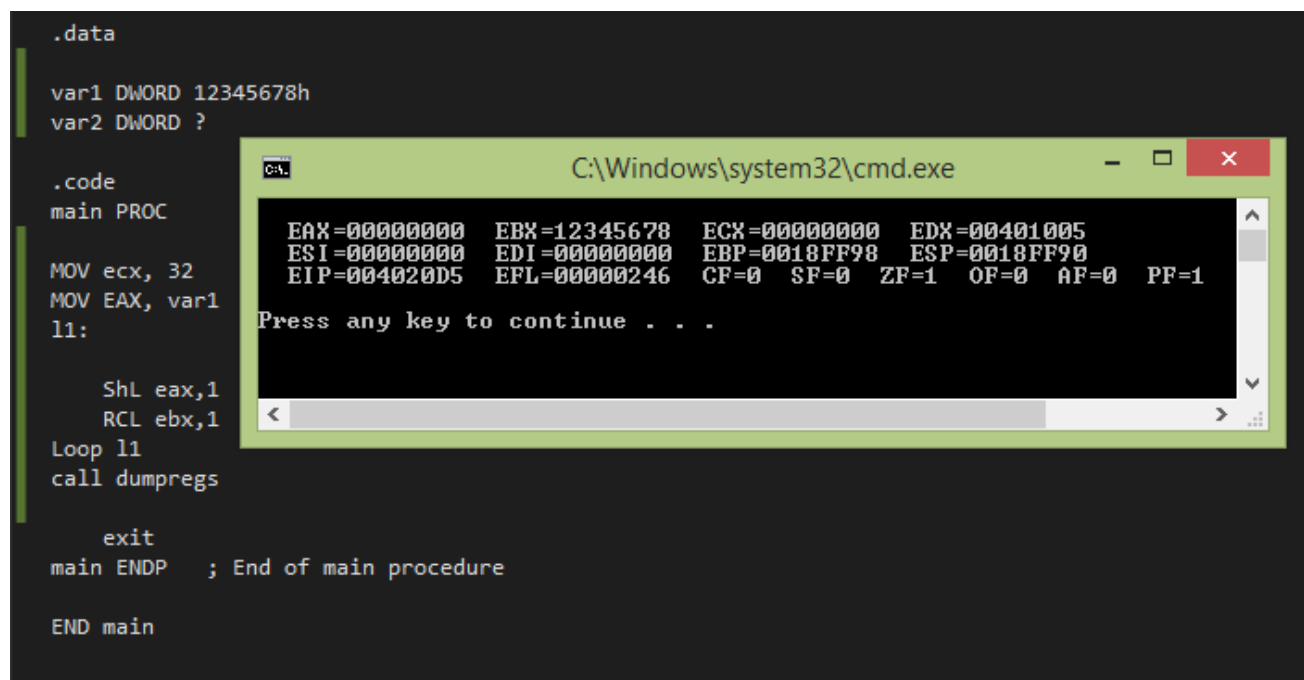
Activity 1:

Copy a number from one variable to another using shift.

Solution:

TITLE Lab Shift and Rotate Instructions and Applications

; Author: Ashfaq Hussain Farooqi



The image shows a screenshot of an assembly program and its execution in a debugger. The assembly code is as follows:

```
.data
var1 DWORD 12345678h
var2 DWORD ?

.code
main PROC
MOV ecx, 32
MOV EAX, var1
l1:
    Shl eax,1
    Rcl ebx,1
Loop l1
call dumpregs

    exit
main ENDP ; End of main procedure

END main
```

The debugger window shows the execution of the program. The register values are displayed as follows:

Register	Value
EAX	00000000
EBX	12345678
ECX	00000000
EDX	00401005
ESI	00000000
EDI	00000000
EBP	0018FF98
ESP	0018FF90
EIP	004020D5
EFL	00000246
CF	0
SF	0
ZF	1
OF	0
AF	0
PF	1

The debugger window also shows the instruction being executed: `Shl eax,1`. The instruction pointer (EIP) is 004020D5. The program is paused at the instruction `Shl eax,1`.

Activity 2:

Reversing a number using shift.

Solution:

TITLE Lab Shift and Rotate Instructions and Applications

; Author: Ashfaq Hussain Farooqi

```
.data

var1 DWORD 12345678h
```

```

        var2 DWORD ?
        count1 DWORD ?

.code
main PROC
MOV ECX, 8
MOV EAX, var1
l1:
    mov count1, ecx
    mov ecx, 4
l2:
    Shr eax,1
    RCL ebx,1

    loop l2
    mov ecx, 4
l3:
    shr ebx,1
    rcl edx,1

    loop l3
    mov ecx, count1

Loop l1
call dumpregs

exit

main ENDP      ; End of main procedure

END main

```

Activity 3: **Multiply 2 binary numbers using shift.**

Solution:

TITLE Lab Shift and Rotate Instructions and Applications

; Author: Ashfaq Hussain Farooqi

.data

```

        Num1 DWORD 10010101b
        Num2 DWORD 00000100b
        Num3 DWORD 0d
        count1 DWORD ?
        msg1 BYTE "msg1",0dh,0ah,0
        msg2 BYTE "msg2",0dh,0ah,0

.code
main PROC
mov     ebx, num2
mov     ecx, 32
LL1:
        SHR ebx,1
        JC 11
        INC Num3
loop LL1
l2:
        ;MOV EDX, offset msg1
        ;Call WriteString
        JMP Endd
11:
        ;MOV EDX, offset msg2
        ;Call WriteString
        MOV EAX, Num3
MOV ecx, Num3
MOV EAX, Num1
LLL:
        SHL eax,1
Loop LLL
Call WriteInt
Endd:

```

CALL CRLF

exit

main ENDP ; End of main procedure

END main

3) Stage v (verify)

Home Activities:

Activity # 1:

Write a program that takes input from the user and checks if the number is positive/negative and even/odd using shift commands.

Solution:

TITLE Lab Shift and Rotate Instructions and Applications

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

msg1 BYTE "Enter number: ",0

msg2 BYTE "Number is negative!",0

msg3 BYTE "Number is positive!",0

msg4 BYTE "Number is odd!",0

msg5 BYTE "Number is even!",0

.code

main PROC

start:

MOV EDX,offset msg1

CALL writestring

CALL readint

MOV EBX,EAX

SHL EAX,1

JC mega

MOV EDX,offset msg3

CALL writestring

CALL crlf

JMP break

mega:

MOV EDX,offset msg2

CALL writestring

CALL crlf

break:

SHR EBX,1

MOV EAX,EBX

JC even1

MOV EDX,offset msg5

CALL writestring

CALL CRLF

JMP target

even1:

MOV EDX,offset msg4

CALL writestring

CALL crlf

target:

CALL crlf

JMP Start

exit

main ENDP

END main

Activity # 2:

Write a program that performs simple encryption by rotating each plaintext byte a varying number of positions in different directions. For example, in the following array that represents the encryption key, a negative value indicates a rotation to the left and a positive value indicates a rotation to the right. The integer in each position indicates the magnitude of the rotation:

key BYTE -2, 4, 1, 0, -3, 5, 2, -4, -4, 6

Your program should loop through a plaintext message and align the key to the first 10 bytes of the message. Rotate each plaintext byte by the amount indicated by its matching key array value. Then, align the key to the next 10 bytes of the message and repeat the process.

1) Stage a2 (assess)

Demonstrate Activity# 2 and Viva Voce

Statement Purpose:

Objective of this lab is to introduce students with Integer multiplication in x86 assembly language that can be performed as a 32-bit, 16-bit, or 8-bit operation. In many cases, it revolves around EAX or one of its subsets (AX, AL). The MUL and IMUL instructions perform unsigned and signed integer multiplication, respectively.

Activity Outcomes:

This lab teaches you the following topics:

- Students will know that MUL and IMUL instructions are very helpful in performing multiplication for unsigned and signed integers.
- They will also learn the way these two differs from each other and the way we can use the suitable one.
- IMUL comes with multiple operands so the students will be able to distinguish the flexibility provide by IMUL over MUL

Instructor Note:

MUL Instruction

- The MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.

The IMUL (signed multiply) instruction performs signed integer multiplication.

- Unlike the MUL instruction, IMUL preserves the sign of the product.
- It does this by sign extending the highest bit of the lower half of the product into the upper bits of the product.

1) Stage](Journey)

Introduction

MUL instruction formats are:

- MUL r/m8
- MUL r/m16
- MUL r/m32

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

The x86 instruction set supports three formats for the IMUL instruction:

- **One operand**
- **Two operands**
- **Three operands**

The one-operand formats store the product in AX, DX:AX, or EDX:EAX:

```
IMUL reg/mem8           ; AX = AL * reg/mem8
IMUL reg/mem16          ; DX:AX = AX * reg/mem16
IMUL reg/mem32          ; EDX:EAX = EAX * reg/mem32
```

The two-operand version of the IMUL instruction stores the product in the first operand, which must be a register.

- **The second operand (the multiplier) can be a register, memory operand, or immediate value**

```
IMUL reg16, reg/mem16
IMUL reg16, imm8
IMUL reg16, imm16
```

The three-operand formats store the product in the first operand.

- **The second operand can be a 16-bit register or memory operand, which is multiplied by the third operand, an 8- or 16-bit immediate value:**

```
IMUL reg16, reg/mem16, imm8
IMUL reg16, reg/mem16, imm16
```

2) Stage a1 (apply)

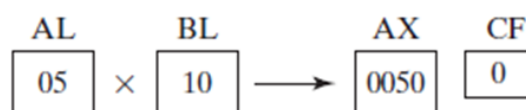
Lab Activities:

Activity 1:

Unsigned Multiplication: Multiply AL by BL, storing the product in AX.

Solution:

```
mov al, 5h
mov bl, 10h
mul bl                               ; AX = 0050h, CF = 0
```



The Carry flag is clear (CF = 0) because AH (the upper half of the product) equals zero

Activity 2:

Unsigned Multiplication: 100h * 2000h, using 16-bit operands.

Solution:

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

val1 WORD 2000h

val2 WORD 100h

.code

mov ax,val1

mul val2 ; DX:AX = 00200000h, CF=1

exit

main ENDP

END main

The Carry flag indicates whether or not the upper half of the product contains significant digits.

Activity 3:

Unsigned Multiplication: 12345h * 1000h, using 32-bit operands.

Solution:

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

Main proc

mov eax,12345h

mov ebx,1000h

mul ebx ; EDX:EAX = 0000000012345000h, CF=0

exit

main ENDP

END main

Activity 4:

Singed Multiplication; Multiply 48 by 4, producing +192 in AX.

Solution:

```
mov    al,48
mov    bl,4
imul   bl                ; AX = 00C0h, OF = 1
```

Although the product is correct, AH is not a sign extension of AL, so the Overflow flag is set.

Activity 5:

Singed Multiplication; Multiply -4 by 4, producing -16 in AX.

Solution:

```
mov    al,-4
mov    bl,4
imul   bl                ; AX = FFF0h, OF = 0
```

AH is a sign extension of AL so the Overflow flag is clear.

Activity 6:

Singed Multiplication; Multiply 48 by 4, producing 192 in DX:AX.

Solution:

```
mov    ax,48
mov    bx,4
imul   bx                ; DX:AX = 000000C0h, OF = 0
```

DX is a sign extension of AX, so the Overflow flag is clear

Activity 7:

Singed Multiplication; 4823424 * -423.

Solution:

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

```

.data

.code

Main proc

    mov eax,4823424

    mov ebx,-423

    imul ebx      ; EDX:EAX = FFFFFFFF86635D80h, OF=0

exit

main ENDP

END main

```

Activity 8: **Singed Multiplication; Give examples of two operands as much as possible.**

Solution:

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

```

.data

word1    SWORD          4

dword1    SDWORD        4

.code

mov      ax,    -16      ; AX = -16

mov      bx ,    2       ; BX = 2

mull     bx,     ax      ; BX = -32

imul     bx,     2       ; BX = -64

imul     bx,     word1   ; BX = -256

mov      eax,   -16      ; EAX = -16

mov      ebx,    2       ; EBX = 2

imul     ebx,   eax      ; EBX = -32

imul     ebx,    2       ; EBX = -64

imul     ebx,   dword1   ; EBX = -256

```

exit

main ENDP

END main

Activity 9:

Singed Multiplication; Give examples of three operands as much as possible.

Solution:

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

word1 SWORD 4

dword1 SDWORD 4

.code

imul bx, word1, -16 ; BX = -64

imul ebx, dword1, -16 ; EBX = -64

imul ebx, dword1, -2000000000 ; OF = 1

exit

main ENDP

END main

3) Stage v (verify)

Home Activities:

Activity 1:

Take two unsigned numbers from the user and display its product (8Bit, 16Bit, and 32Bit variables).

Activity 2:

Take two signed numbers from the user and display its product (8Bit, 16Bit, and 32Bit variables).

4) Stage a2 (assess)

Assignment:

Consider there are two arrays having six indexes. Populate them by taking inputs from the user. Now, populate the third array by multiplying the same index values. If the numbers are positive use MUL and if the numbers are signed use IMUL instruction.

Note: Consider numbers as Signed/Unsigned and are 32Bit values.

Statement Purpose:

Objective of this lab is to introduce students with Integer division in x86 assembly language that can be performed as a 32-bit, 16-bit, or 8-bit operation. In many cases, it revolves around EDI and EAX or its subsets (AX, AL). The DIV and IDIV instructions perform unsigned and signed integer division, respectively. The DIV instruction performs unsigned integer division, and IDIV performs signed integer division.

Activity Outcomes:

This lab teaches you the following topics:

- Students will know that DIV and IDIV instructions are very helpful in performing division for unsigned and signed integers.
- They will also learn the way these two differs from each other and the way we can use the suitable one.
- Students will be able to know that DIV and IDIV have the same operands so that major difference is the signed Unsigned inter division.
- Sign extension statements like CBW, CWD and CDQ are used and applied.
- Student will be able to divide 64bit number and will get information about the remainder and quotients as well for different size of variables.

Instructor Note:

DIV Instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers. A single operand is supplied (register or memory operand), which is assumed to be the divisor

The IDIV Instruction

The IDIV (signed divide) instruction performs signed integer division, using the same operands as DIV.

- Before executing 8-bit division, the dividend (AX) must be completely sign-extended.
- The remainder always has the same sign as the dividend.

1) Stage](Journey)

Introduction

Instruction formats for DIV:

- DIV reg/mem8
- DIV reg/mem16
- DIV reg/mem32

Dividend	Divisor	Quotient	Remainder
AX	<i>r/m8</i>	AL	AH
DX:AX	<i>r/m16</i>	AX	DX
EDX:EAX	<i>r/m32</i>	EAX	EDX

Instruction formats for IDIV are same as DIV:

- **IDIV reg/mem8**
- **IDIV reg/mem16**
- **IDIV reg/mem32**

Dividend	Divisor	Quotient	Remainder
AX	<i>r/m8</i>	AL	AH
DX:AX	<i>r/m16</i>	AX	DX
EDX:EAX	<i>r/m32</i>	EAX	EDX

To preserve the sign for 8bit numbers CBW instruction is used before IDIV while CWD for 16Bit and CDQ for 32Bit signed division.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

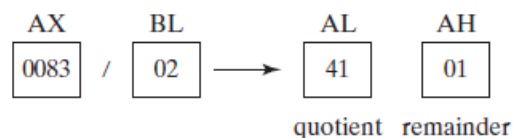
8-bit unsigned division (83h / 2), producing a quotient of 41h and a remainder of 1.

Solution:

```

mov ax,0083h           ; dividend
mov bl,2                ; divisor
div bl                  ; AL = 41h, AH = 01h

```



Activity 2:

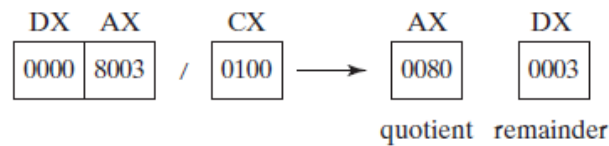
16-bit unsigned division (8003h / 100h), producing a quotient of 80h and a remainder of 3.

Solution:

```

mov dx,0                ; clear dividend, high
mov ax,8003h            ; dividend, low
mov cx,100h             ; divisor
div cx                  ; AX = 0080h, DX = 0003h

```



DX contains the high part of the dividend, so it must be cleared before the DIV instruction executes.

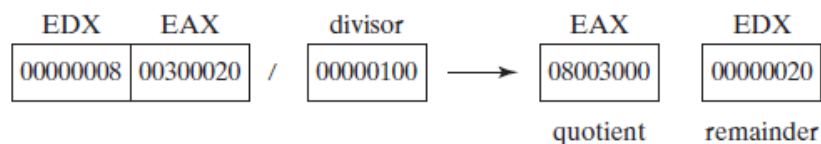
Activity 3: 32-bit unsigned division using a memory operand as the divisor.

Solution:

```

.data
dividend QWORD 0000000800300020h
divisor  DWORD 00000100h
.code
mov edx,DWORD PTR dividend + 4 ; high doubleword
mov eax,DWORD PTR dividend      ; low doubleword
div divisor                     ; EAX = 08003000h, EDX = 00000020h

```



Activity 4: 8 bit division. Divide -48 by 5.

Solution:

```

.data
byteVal SBYTE -48                ; D0 hexadecimal
.code
mov al,byteVal                   ; lower half of dividend
cbw                              ; extend AL into AH
mov bl,+5                        ; divisor
idiv bl                          ; AL = -9, AH = -3

```

Activity 5:

**16-bit division requires AX to be sign-extended into DX.
Divide -5000 by 256**

Solution:

```
.data
wordVal SWORD -5000
.code
mov ax,wordVal ; dividend, low
cwd ; extend AX into DX
mov bx,+256 ; divisor
idiv bx ; quotient AX = -19, rem DX = -136
```

Activity 6:

**32-bit division requires EAX to be sign-extended into EDX.
Divide 50,000 by -256**

Solution:

```
.data
dwordVal SDWORD +50000
.code
mov eax,dwordVal ; dividend, low
cdq ; extend EAX into EDX
mov ebx,-256 ; divisor
idiv ebx ; quotient EAX = -195, rem EDX = +80
```

3) Stage v (verify)

Home Activities:

Activity 1:

Solve this equation; var4 = (var1 + var2) * var3

Solution:

; Assume unsigned operands

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

Main proc

mov eax,var1

add eax,var2 ; EAX = var1 + var2

```

    mul var3      ; EAX = EAX * var3
    jc TooBig     ; check for carry
    mov var4,eax ; save product
    Jump next:
    TooBig:

```

```
exit
```

```
main ENDP
```

```
END main
```

Activity 2:

Solve this equation; $\text{var4} = (\text{var1} * 5) / (\text{var2} - 3)$;

Solution:

```
; Assume unsigned operands
```

```
TITLE Lab Multiplication and Division Instructions
```

```
; Author: Ashfaq Hussain Farooqi
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
.code
```

```
Main proc
```

```
mov eax,var1      ; left side
```

```
mov ebx,5
```

```
mul ebx    ; EDX:EAX = product
```

```
mov ebx,var2    ; right side
```

```
sub ebx,3
```

```
div ebx    ; final division
```

```
mov var4,eax
```

```
exit
```

```
main ENDP
```

```
END main
```

Activity 3:

Solve this equation; $\text{var4} = (\text{var1} * 5) / (\text{var2} - 3)$;

Solution:

Solve this equation; $\text{var4} = (\text{var1} * -5) / (-\text{var2} \% \text{var3})$;

Solution:

; Assume signed operands

TITLE Lab Multiplication and Division Instructions

; Author: Ashfaq Hussain Farooqi

INCLUDE Irvine32.inc

.data

.code

Main proc

mov eax,var2 ; begin right side

neg eax

cdq; sign-extend dividend

idiv var3 ; EDX = remainder

mov ebx,edx ; EBX = right side

mov eax,-5 ; begin left side

imul var1 ; EDX:EAX = left side

idiv ebx ; final division

mov var4,eax ; quotient

exit

main ENDP

END main

4) Stage a2 (assess)

Assignment:

Consider there are two arrays having six indexes. Populate them by taking inputs from the user. Now, populate the third array by dividing the same index values. If the numbers are positive use DIV and if the numbers are signed use IDIV instruction.

Note: Consider numbers as Signed/Unsigned and are 32Bit values.