**Computer Organization and Assembly Language**

Table of Contents

[Syllabus 4](#_Toc103074502)

[Mapping of Course Learning Outcomes with Program Learning Outcomes 6](#_Toc103074503)

[Lab Marks Plan: 8](#_Toc103074504)

[Rubrics for Psychomotor and Affective domain 8](#_Toc103074505)

[Weekly plan: 1](#_Toc103074506)

[Lab Experiments 2](#_Toc103074507)

[*Lab 0 – Microsoft Assembler Installation and linker with VS* 3](#_Toc103074508)

[*Lab 1 - x86 Registers* 11](#_Toc103074516)

[Lab 2 – Introduction 15](#_Toc103074517)

[Lab 3 - Data Types 25](#_Toc103074529)

[Lab 4 - Arrays 30](#_Toc103074530)

[Lab 5 - Flags 36](#_Toc103074540)

[Lab 6 - Conditional Statements and Loops 46](#_Toc103074553)

[Lab 7- Procedures 56](#_Toc103074554)

[Lab 8 -Procedures with param 61](#_Toc103074555)

[Lab 9- Floating Point 63](#_Toc103074556)

[Lab 10 - File Handling Code 71](#_Toc103074557)

[Lab-11 Use of Loops and if Else (Open Ended lab) 75](#_Toc103074558)

[Lab-12 Open Ended Lab 76](#_Toc103074559)

# 

# Syllabus

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CMPE-223L Computer Organization and Assembly Language (Lab)** | | | | | | | | | | | | | | | | |
| **Course Designed By** | | **Department of Computer Engineering** | | | | | | | | | | | | | | |
| 1 | Program Learning Outcomes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | 8 | 9 | | 10 | 11 | | 12 |
| x |  |  |  | x |  |  | | x |  | |  | |  | |
| 2 | Mapping of Course Learning Outcomes vs Program Learning Outcomes | 2,3 |  |  |  | 1 |  |  | | 4 |  | |  | |  | |
| 3 | Domain and Level | Cognitive  Level 2 | | | | | | | | | | | | | | |
| 4 | Category | General (G) | | Basic Science (B) | | Engineering Sciences and Technical Arts (E) | | | | | | Professional Subjects (P) | | | | |
| -- | | -- | | x | | | | | | -- | | | | |
| 5 | Broad Area | Digital System Processing | | Circuits & Systems | | Electronics | | | Embedded Systems | | | Computing | | | | |
| -- | | -- | | -- | | | -- | | | x | | | | |

# Mapping of Course Learning Outcomes with Program Learning Outcomes

**Mapping of Course Learning Outcomes with Program Learning Outcomes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Course Learning Outcomes (CLO)** | **Program Learning Outcome (PLO)** | **Domain** | **Level** |
| 1. Use assembler, debugger to execute basic assembly language programs. | Modern Tool Usage (PLO5) | Cognitive | 2 |
| 1. Construct programs in assembly language using x86 32 and 64 bit instruction set. | Engineering Knowledge (PLO1) | Cognitive | 3 |
| 1. Analyse hardware control and communication using BIOS and DOS interrupts and system calls. | Engineering Knowledge (PLO1) | Cognitive | 4 |
| 1. Comply with plagiarism guidelines. | Ethics (PLO8) | Cognitive | 5 |

# 

# 

# Lab Marks Plan:

|  |  |  |
| --- | --- | --- |
| Sr. No. | Marks Split up | Maximum Marks (10) |
| 1 | Report Marks | 3 |
| 2 | Experiment Performance | 7 |

# Rubrics

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| CLOs | Description | PLOs/Domain/  Domain Level | Barely  Acceptable | Basic | Good | Very Good |
| Points |  |  | 0-2 | 3-5 | 6-8 | 9-10 |
| CLO1 | Use assembler, debugger to execute basic assembly language programs. | PLO5/Cogni/2 | Use breadboard and do not know the implementation of basic circuits. | Present the logic gate circuitry from Boolean expression only. | Implement the working hardware for basic logic gate and MSL. | Perform complete diagnosis of logic gates without errors. |
| CLO2 | Construct programs in assembly language using x86 32 and 64 bit instruction set. | PLO1/Cogni/3 | Demonstrate less or no knowledge of basic implementation of digital logic circuits. | Present digital logic circuits with no design and cost effective foundation. | Implement the cost-effective digital circuit with some design methodology | Deliver complete cost-effective multiple gate circuits patching with optimization and reusability of components |
| CLO3 | Analyse hardware control and communication using BIOS and DOS interrupts and system calls. | PLO1/Cogni/4 | Demonstrate none of the arithmetic and logic operations or its implementation using logic gates and MSL. | Present some arithmetic and logic operations using logic gates and MSL | Implement all required arithmetic and logic operations with some errors or less design Concepts | Perform correct arithmetic and logic operations supported with optimized design concepts. |
| CLO4 | Comply with plagiarism guidelines. | PLO8/Cogni/3 | Discuss the problem but no or barely solve the real-world application. | Display only proposal without implementation discussing solution to real world applications | Integrate digital logic devices for implementation of real world application with proposed methodology | Justify real world application with optimized design concepts and correct arithmetic and logic operation. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CMPE-223L** | **CMPE- Computer Organization and Assembly Language** | **L** | **T** | **P** | **C** |
|  | **Total Contact hours - 45** | **0** | **0** | **3** | **1** |
| **Prerequisite** |  |  |  |  |
| DLD-321L |  |  |  |  |
| **Purpose** |  |  | |  | |
| This Laboratory course will give a thorough knowledge about the basics of circuit analysis. | | | | | |
| **Course Learning Outcomes** | | | | | |
| CLO 1 | Use assembler, debugger to execute basic assembly language programs. | | | | |
| CLO 2 | Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | | |
| CLO 3 | Analyse hardware control and communication using BIOS and DOS interrupts and system calls. | | | | |
| CLO 4 | Comply with plagiarism guidelines. | | | | |

# Weekly plan:

|  |  |  |  |
| --- | --- | --- | --- |
| **Week No.** | **Lab Topics** | **CLO** | **PLO** |
| **1** | Lab 0 Microsoft Assembler Installation and linker with VS | CLO2 | PLO5 |
| **2** | Lab 1 – X86 Registers | CLO2 | PLO1 |
| **3** | Lab 2 – Introduction | CLO3 | PLO1 |
| **4** | Lab 3 – Data Types | CLO1 | PLO1 |
| **5** | Lab 4 – Arrays | CLO2 | PLO1 |
| **6** | **Viva** |  |  |
| **7** | Lab 5 – Flags | CLO1  CLO2 | PLO1 |
| **8** | Lab 6 – Conditional statements and loops | CLO1  CLO2 | PLO1  PLO5 |
| **9** | Lab 7 -Procedures | CLO1  CLO2 | PLO1  PLO5 |
| **10** | Lab 8 - Procedures with Param | CLO1  CLO2 | PLO1  PLO5 |
| **11** | **Project Proposals** |  |  |
| **12** | Lab 9 – Floating Point | CLO3 | PLO1  PLO5 |
| **13** | Lab-10 File Handling | CLO1  CLO2 | PLO1  PLO5 |
| **14** | Lab-11 Opend Ended Lab | CLO1  CLO2  CLO4 | PLO1  PLO5  PLO8 |
| **15** | Lab-12 Open Ended Lab | CLO4 | PLO1  PLO8 |

# Lab Experiments

# Lab 0 – Microsoft Assembler Installation and linker with VS

**Name: ……………………………. Registration No: …………………….**

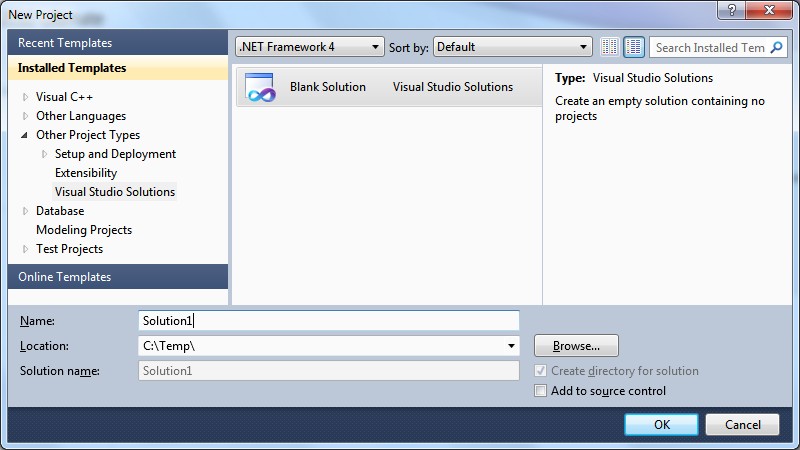
**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO2** | Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

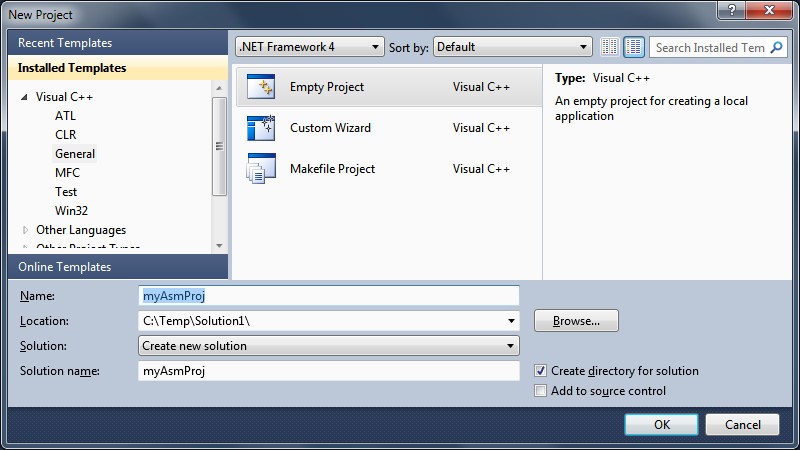
# Create an empty solution:

Use File | New | Project… Expand the ‘Other Project Type‘ tree, Select ‘Visual Studio Solutions‘, and create a new ‘Blank Solution‘ like Solution1

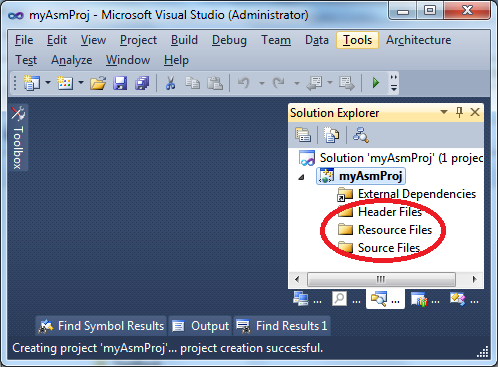


# Add an empty project:

Use File | Add | New Project… Expand the ‘Visual C++‘, ‘General‘ section and create a new ‘Empty Project‘ like myAsmProj

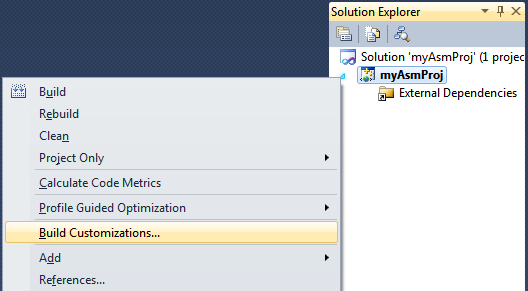


Simply delete three folders:

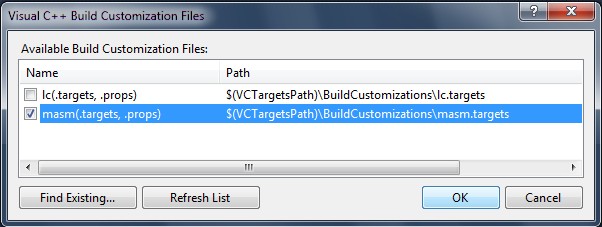


# Acquire the MASM options:

Right click on the Project in the Solution Explorer and select ‘Build Customizations…‘

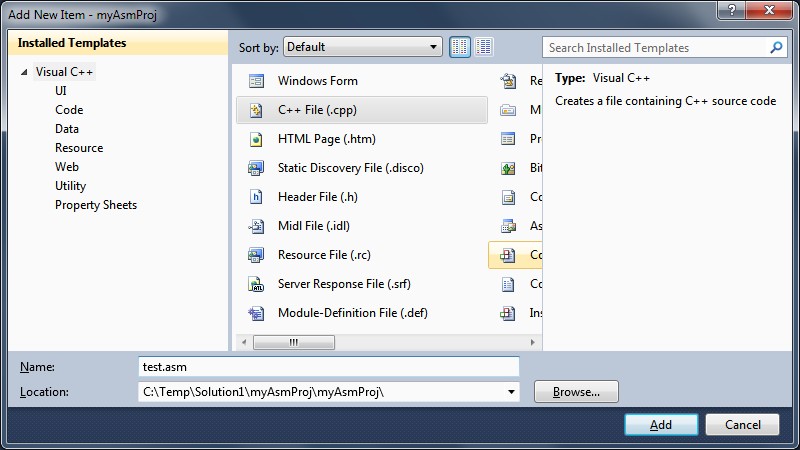


Tick the ‘*masm*‘ box and say OK.



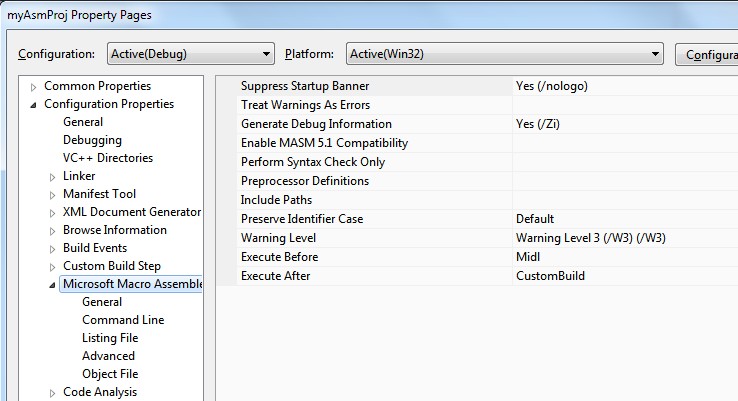
# Add a new source file:

Add a.asm file to the project by right clicking on the Project and selecting ‘Add | New Item…‘with ‘Text File ‘or ‘C++ File’. Enter a filename ending with .asm like test.asm



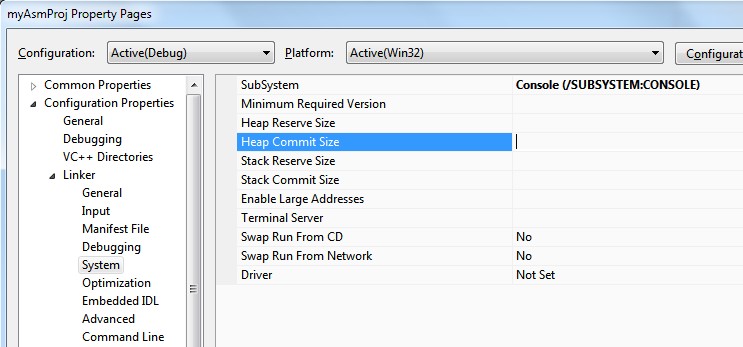
# Verify MASM :

Right click on the Project and select ‘*Properties*‘. You should see a dialog like this (Note the MASM item at the bottom of the tree). If you don’t then something went wrong

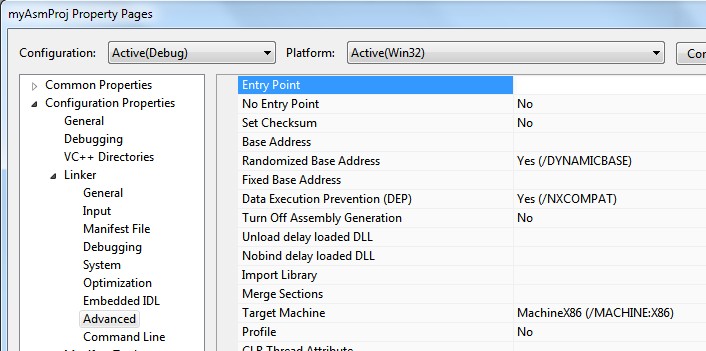


# Configure the linker:

In the above dialog, use Configuration Properties > Linker > System> SubSystem, set the SubSystem to Windows or Console like this

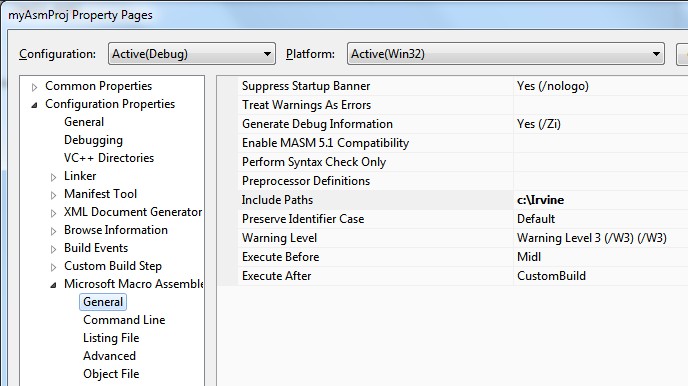


Don’t set Entry Point to the name of ‘main’ method (as per the END directive – see code). Make sure to go Configuration Properties > Linker > Advanced > Entry Point

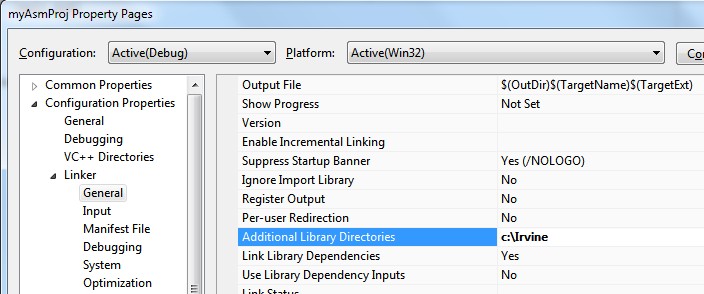


# Add Kip Irvine Support:

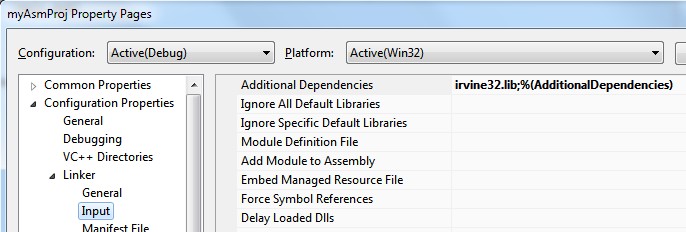
Set c:\Irvine in Include Path under the Microsoft Macro Assembler | General section



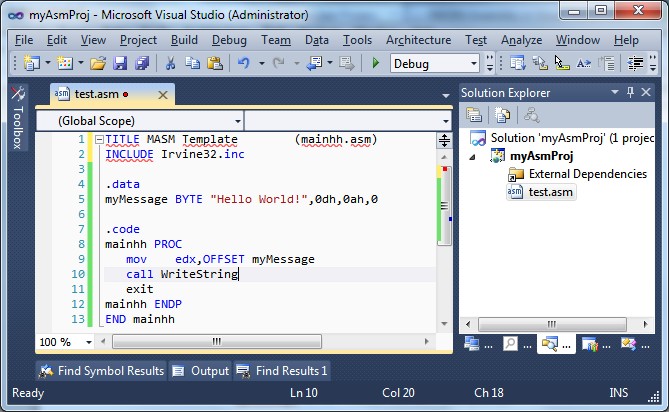
Set c:\Irvine in Additional Library Directories under the Link | General section



Set Irvine32.Lib in Additional Dependencies under the Link | Input section



1. **Try test code:** Copy the following to test.asm and run:



TITLE MASM Template (test.asm) INCLUDE Irvine32.inc

.data

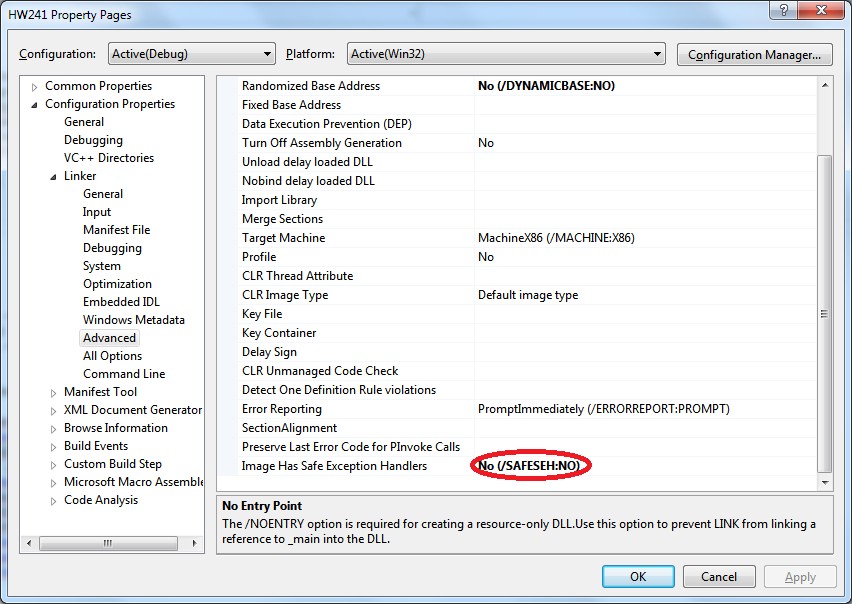
myMessage BYTE "Hello World!",0dh,0ah,0

.code mainhh PROC

mov edx,OFFSET myMessage call WriteString

exit mainhh ENDP END mainhh

1. **Upgrade Visual Studio 2010 project to Visual Studio 2012**: Opening the VS 2010 project in Visual Studio 2012, you must make



# Lab 1 - x86 Registers

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO2** | Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

The main tools to write programs in x86 assembly are the processor registers. The registers are like variables built in the processor. Using registers instead of memory to store values makes the process faster and cleaner. The problem with the x86 series of processors is that there are few registers to use. This section describes the main use of each register and ways to use them. That in note that the rules described here are more suggestions than strict rules.Here is a list of the available registers on the 386 and higher processors. This list shows the 32 bit registers. Most of the can be broken down to 16 or even 8 bits register.

**General registers**

EAX EBX ECX EDX

**Segment registers**

CS DS ES FS GS SS

**Index and pointers**

ESI EDI EBP EIP ESP

**Indicator**

EFLAGS

**General registers**   
as the title says, general register are the one we use most of the time Most of the instructions perform on these registers. They all can be broken down into 16 and 8 bit registers.

32 bits : EAX EBX ECX EDX

16 bits : AX BX CX DX

8 bits : AH AL BH BL CH CL DH DL

The "H" and "L" suffix on the 8 bit registers stand for high byte and low byte. With this out of the way, let's see their individual main use

**EAX,AX,AH,AL** : Called the Accumulator register.

It is used for I/O port access, arithmetic, interrupt calls,

etc...

**EBX,BX,BH,BL** : Called the Base register

It is used as a base pointer for memory access

Gets some interrupt return values

**ECX,CX,CH,CL** : Called the Counter register

It is used as a loop counter and for shifts

Gets some interrupt values

**EDX,DX,DH,DL** : Called the Data register

It is used for I/O port access, arithmetic, some interrupt

Calls.

**Segment registers**  
  
Segment registers hold the segment address of various items. They are only available in 16 values. They can only be set by a general register or special instructions. Some of them are critical for the good execution of the program and you might want to consider playing with them when you'll be ready for multi-segment programming.

**CS** : Holds the Code segment in which your program runs.

Changing its value might make the computer hang.

**DS**  : Holds the Data segment that your program accesses.

Changing its value might give erroneous data.

**ES,FS,GS** : These are extra segment registers available for

far pointer addressing like video memory and such.

**SS** : Holds the Stack segment your program uses.

Sometimes has the same value as DS.

Changing its value can give unpredictable results,

mostly data related.

**Indexes and pointers**   
  
Indexes and pointer and the offset part of and address. They have various uses but each register has a specific function. They sometime used with a segment register to point to far address (in a 1Mb range). The register with an "E" prefix can only be used in protected mode.

**ES:EDI EDI DI** : Destination index register

Used for string, memory array copying and setting and

for far pointer addressing with ES

**DS:ESI EDI SI** : Source index register

Used for string and memory array copying

**SS:EBP EBP BP** : Stack Base pointer register

Holds the base address of the stack

**SS:ESP ESP SP** : Stack pointer register

Holds the top address of the stack

**CS:EIP EIP IP** : Index Pointer

Holds the offset of the next instruction

It can only be read

**The EFLAGS register**  
  
The EFLAGS register hold the state of the processor. It is modified by many instructions and is used for comparing some parameters, conditional loops and conditional jumps. Each bit holds the state of specific parameter of the last instruction. Here is a listing:

Bit Label Description

---------------------------

0 CF Carry flag

2 PF Parity flag

4 AF Auxiliary carry flag

6 ZF Zero flag

7 SF Sign flag

8 TF Trap flag

9 IF Interrupt enable flag

10 DF Direction flag

11 OF Overflow flag

12-13 IOPL I/O Privilege level

14 NT Nested task flag

16 RF Resume flag

17 VM Virtual 8086 mode flag

18 AC Alignment check flag (486+)

19 VIF Virtual interrupt flag

20 VIP Virtual interrupt pending flag

21 ID ID flag

Those that are not listed are reserved by Intel.

**Undocumented registers**  
  
There are registers on the 80386 and higher processors that are not well documented by Intel. These are divided in control registers, debug registers, test registers and protected mode segmentation registers. As far as I know, the control registers, along with the segmentation registers, are used in protected mode programming, all of these registers are available on 80386 and higher processors except the test registers that have been removed on the Pentium. Control registers are CR0 to CR4, Debug registers are DR0 to DR7, test registers are TR3 to TR7 and the protected mode segmentation registers are GDTR (Global Descriptor Table Register), IDTR (Interrupt Descriptor Table Register), LDTR (Local DTR), and TR.

# Lab 2 – Introduction

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

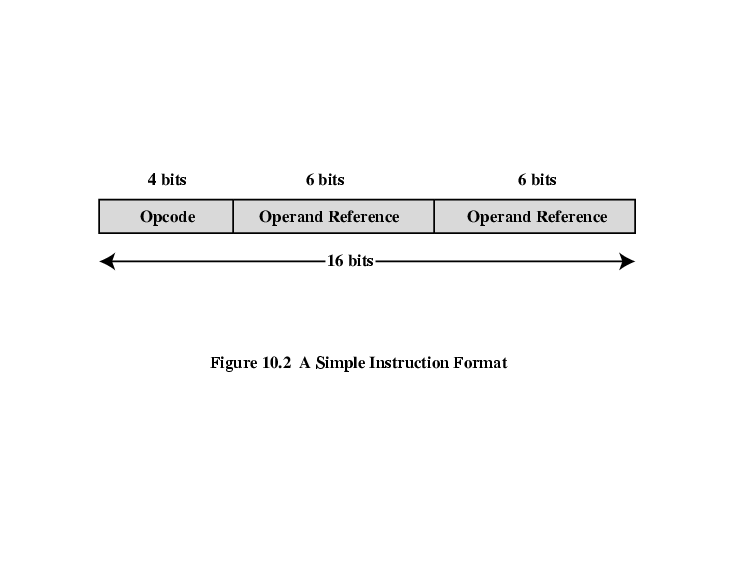
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO3** | Analyse hardware control and communication using BIOS and DOS interrupts and system calls. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Instructions:

Almost every instruction in assembly/low language has two parts. One is **opcode** the other is address **references.**

If we break down a 16-bit instruction, then we may get following components.



**Opcode**: This is a unique pattern which identify “what operation is to be done”.

**Operand references:** This is the unique address of a memory location “where the operation is needed to be done”. Operands can be immediate value, variable, memory address or register.

For example,

If add a,b is an instruction, then add is a nemonic which represent unique opcode of add operation, which is telling the CPU to add something. a,b are the operands whose values will be added in the result of the operation.

Depending upon the architecture, we may have a third reference where the result of the operation will be stored or just one reference. In case of one reference, accumulator register “ax” will be used implicitly to hold the temporary data during arithmetic operations.

Add operation can be executed as:

**One reference:**

mov a ;copy value of a into accumulator register “ax”.

add b ;add value of b into value of “ax” and keep in it.

sto c ;copy the value of accumulator register in “c variable”.

**Two References:**

add a,b ;add value of b into a and keep the result in a.

sto c,a ;copy the value of a into c.

**Three References:**

add c,a,b ;add values of a and b, and store the result in c.

# Register and their types:

Register are small but fastest memory locations located inside CPU. They are used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. There are various types of registers used for different purposes. Mostly used Registers named as AC or **Accumulator**, Data Register or DR, the AR or **Address Register**, **Program Counter** (PC), **Memory Data Register** (MDR).

Registers are grouped into several categories as follows:

* **Four general-purpose registers**, AX, BX, CX, and DX.
* **Four special-purpose registers**, SP, BP, SI, and DI.
* **Four segment registers**, CS, DS, ES, and SS.
* **The instruction pointer**, IP (sometimes referred to as the program counter).
* **The status flag register**, FLAGS.

## **General Purpose Registers:**

**32 bits registers:**

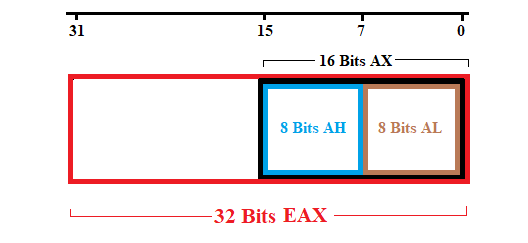
EAX, EBX, ECX, EDX are the 32, contains 4 bytes.

**16 bits registers:**

AX, BX, CX, DX are the 16 bit registers, contain lower two bytes of 32 bit registers.

**8 bit registers:**

AL, AH, BH, BL, CH, CL, DH, DL are the 8 bit register. The "H" and "L" suffix on the 8 bit registers stand for high byte and low byte of 16 bits registers.



**EAX,AX,AH,AL** :   
Called as the "accumulator''; some of the operations, such as MUL and DIV, require that one of the operands be in the accumulator. Some other operations, such as ADD and SUB, may be applied to any of the registers (that is, any of the eight general- and special-purpose registers) but are more efficient when working with the accumulator.

**EBX,BX,BH,BL** :  
Called as the "base'' register; it is the only general-purpose register which may be used for indirect addressing. For example, the instruction MOV [BX], AX causes the contents of AX to be stored in the memory location whose address is given in BX.

**ECX,CX,CH,CL** :  
Called as the "count'' register. The looping instructions (LOOP, LOOPE, and LOOPNE), the shift and rotate instructions (RCL, RCR, ROL, ROR, SHL, SHR, and SAR), and the string instructions (with the prefixes REP, REPE, and REPNE) all use the count register to determine how many times they will repeat.

**EDX,DX,DH,DL** :  
Called as the "data'' register; it is used together with AX for the word-size MUL and DIV operations, and it can also hold the port number for the IN and OUT instructions, but it is mostly available as a convenient place to store data, as are all of the other general-purpose registers.

# Special Purpose Register:

* SP is the stack pointer, indicating the current position of the top of the stack. You should generally never modify this directly, since the subroutine and interrupt call-and-return mechanisms depend on the contents of the stack.
* BP is the base pointer, which can be used for indirect addressing similar to BX.
* SI is the source index, used as a pointer to the current character being read in a string instruction (LODS, MOVS, or CMPS). It is also available as an offset to add to BX or BP when doing indirect addressing; for example, the instruction MOV [BX+SI], AX copies the contents of AX into the memory location whose address is the sum of the contents of BX and SI.
* DI is the destination index, used as a pointer to the current character being written or compared in a string instruction (MOVS, STOS, CMPS, or SCAS). It is also available as an offset, just like SI

In word is either 16-bit log memory storage unit.

Getting Started with Visual Studio:

# For Lab:

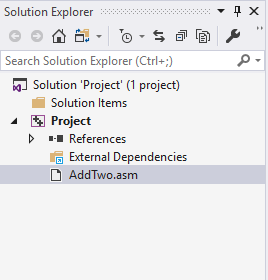
[Click here](http://kipirvine.com/asm/gettingStartedVS2015/Project32_VS2015.zip) and follow the link to download Visual Studio Project, extract it anywhere in computer.

# Opening a Project

Visual Studio requires 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.

Do the following steps, in order:

1. Start Visual Studio.
2. To begin, open our sample Visual Studio project file by selecting **File/Open/Project** from the Visual Studio menu.
3. Navigate to your working folder where you unzipped our project file, and select the file named **Project.sln**.
4. Once the project has been opened, you will see the project name in the Solution Explorer window. You should also see an assembly language source file in the project named AddTwo.asm. Double-click the file name to open it in the editor.



You should see the following program in the editor window:

|  |
| --- |
| **; AddTwo.asm - adds two 32-bit integers.**  **; Chapter 3 example**  **.386**  **.model flat,stdcall**  **.stack 4096**  **ExitProcess proto,dwExitCode:dword**  **.code**  **main proc**  **mov eax,5**  **add eax,6**  **invoke ExitProcess,0**  **main endp**  **end main** |

In the future, you can use this file as a starting point to create new programs by copying it and renaming the copy in the Solution Explorer window.

**Adding a File to a Project:** If you ever need to add an .asm file to an open project, do the following: (1) Right-click the project name in the Visual Studio window, select Add, select Existing Item. (2) In the *Add Existing Item*dialog window, browse to the location of the file you want to add, select the filename, and click the Add button to close the dialog window.

# Build the Program

Now you will build (assemble and link) the sample 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 ..\Project32\_VS2015\AddTwo.asm...**

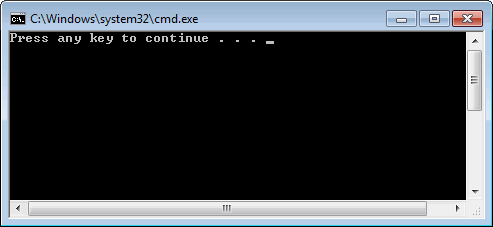
**1> Project.vcxproj -> ...\Project32\_VS2015\Debug\Project.exe**

**========== Rebuild All: 1 succeeded, 0 failed, 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 **Local Window Debugger**. 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.

Press any key to close the Console window.

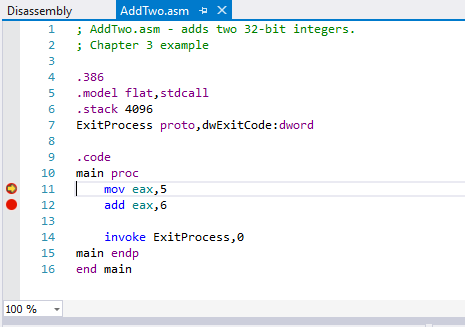
# Running the Sample Program in Debug Mode

#### **Setting a BreakPoint**

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.

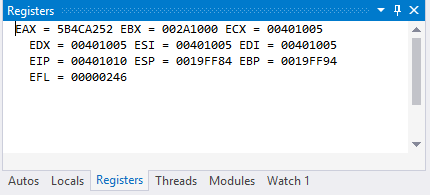
1. In our sample program, click the mouse along the border to the left of the **mov eax,5** statement. A large red dot should appear in the margin.
2. Select Local Window Debugger. The program should run, and pause on the line with the breakpoint, showing the Yellow arrow.
3. Press *Continue* until the program finishes.

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.



#### **Registers**

You can view current values of different registers while debugging a program. Press Alt+Ctrl+G to open register window while program is debugging.



#### **Copying a Source File**

One way to make a copy of an existing source code file is to use Windows Explorer to copy the file into your project directory. Then, right-click the project name in Solution Explorer, select Add, select Existing Item, and select the filename. Or you can copy and paste code from .asm file to any other text file to save the code using same project again and again.

# Structure of Assembly Program

Assembly programs have mainly two segments, one is data and other is code. Data part contains declared variables like words and strings etc. Code part contains code.

**Single Line Comments:** In assembly language, single line comments starts with semicolon ;

INCLUDE Irvine32.inc

.data

;Variables and Data

.code

main PROC ; Main Procedure/Function start

;Code

exit ; Exiting the Process

main ENDP ; Main Function/End

END main ; Marks the END of the end of Program, and main identify entry procedure.

This a snippet of code which does nothing at all. But it contains the structure of assembly program. We would write code inside our “main PROC” (Main Procedure) which is entry point of the program.

LAB 1

|  |  |  |  |
| --- | --- | --- | --- |
| Mnemonics | Syntax | Working | Example |
| mov | mov op1,op2 | Copy data from operand 2 to operand 1. | mov eax, 5  5 is being moved into eax register. |
| add | add op1, op2 | Add operand 2 into operand 1 and keep it stored in operand 1. | add eax, 6  integer 6 sums with eax and get store in eax. |
| mul | Mul op1 | It multiplies operand to eax and store result in edx:ex | Mul b It multiplies b with the content of eax and store result in edx:eax |
| call | call proc1 | Calling the procedure1 | Call writeint It will print the value present in eax register in console. |

(All arithmetic operations are performed in registers. Variable will only be used to store data. No processing can be done inside a variable.)

When two 32-bit integers are multiplied, output is 64-bit integer. In 32-bit architecture, 64-bit registers don’t exist. Hence to accommodate 64-bit integer, upper 32-bits are moved to edx and lower 32-bits are moved to eax.

***Procedures:***

These are procedures present in Kip Irvine library of assembly Language.

**writeint:**

It prints the value of eax register on console.

**readint:**

It generate an interrupt and waits for the user to enter something in the console and copy in the input to eax register.

**Variable Declaration:**

**Syntax: Example:**

name TYPE value num dword 7

Above, num is an variable of type DWORD, (dword has length 32-bit) and 7 is the value of num. If we don’t want to initialize variable we can use ? instead of 7 .

You will know about all type of variables in coming labs.

Example 1: Add Two Numbers

INCLUDE Irvine32.inc

.data

.code

main PROC

mov eax, 6 ; copy 6 into eax

add eax, 10 ; add 10 into eax and store result in eax

call writeint ; print value of eax register on colse

call readint ; Stoping Console from disappearning

exit

main ENDP

END main

Result: +16

+ sign shows that the result is signed.

Example 2: Multiply Two Numbers

INCLUDE Irvine32.inc

.data

b dword 40 ; declares 32-bit variable with value 40 of

.code

main PROC

mov eax, 12 ; Copy 12 to 32-bit register eax

mul b ; value of b is multiplied with value of eax and result is

; stored in eax.

call writeint ; print value of eax register on console

mov eax, edx ; move the value of edx to eax so that it can be printed

call writeint

call readint ; Stoping Console from disappearning

exit

main ENDP

END main

Result: +480+0

Here +480 is the value of eax register and +0 is the value of edx.

*Why edx is zero after multiply operation?*

Because, eax (32-bit reigster) can hold a unsigned integer value of 232 = 4294967295. So if the unsinged value of output after the multiplication is less than or equal to 4294967295, eax alone will be able to handle it and edx will remain zero.

Lab Tasks:

1. Write a program to solve this equation using only eax register (10 - 7) \* (5 + 6) \* 9.

(*Note:* *Multiple variables can be used).*

1. Write a program to solve this equation without using any variable (4 \* 5) - (3 + 7 \* 21).  
   (*Note: Multiple registers can be used*).

# Lab 3 - Data Types

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1** | Use assembler, debugger to execute basic assembly language programs. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Character ConstantsA character constant is a single character enclosed in single or double quotes. MASM stores the value in memory as the character’s binary ASCII code. Examples are  
'A'  
"d"

String ConstantsA string constant is a sequence of characters (including spaces) enclosed in single or double quotes:

'ABC'  
'X'  
"Good night, Gracie"  
'4096'

Embedded quotes are permitted when used in the manner shown by the following examples:  
  
"This isn't a test"  
'Say "Good night," Gracie'

**Reserved Words***Reserved words* have special meaning in MASM and can only be used in their correct context.  
There are different types of reserved words:  
**•** Instruction mnemonics, such as MOV, ADD, and MUL  
**•** Register names  
**•** Directives, which tell MASM how to assemble programs  
**•** Attributes, which provide size and usage information for variables and operands. Examples  
are BYTE and WORD  
**•** Operators, used in constant expressions  
**•** Predefined symbols, such as @data, which return constant integer values at assembly time

|  |  |
| --- | --- |
| **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 (can also be a Near pointer in real-address mode) |
| SWORD | 16-bit signed integer |
| DWORD | 32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double |
| SDWORD | 32-bit signed integer. SD stands for signed double |
| FWORD | 48-bit integer (Far pointer in protected mode) |
| 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 |

Defining BYTE and SBYTE DataThe BYTE (define byte) and SBYTE (define signed byte) directives allocate storage for one or more unsigned or signed values. Each initializer must fit into 8 bits of storage. For example,

|  |  |
| --- | --- |
| value1 BYTE 'A'  value2 BYTE 0  value3 BYTE 255  value4 SBYTE 128  value5 SBYTE +127 | ; character constant ; smallest unsigned byte ; largest unsigned byte ; smallest signed byte ; largest signed byte |

A question mark (?) initializer leaves the variable uninitialized, implying it will be assigned a value at runtime:

value6 BYTE ?

The optional name is a label marking the variable’s offset from the beginning of its enclosing segment. For example, if **value1** is located at offset 0000 in the data segment and consumes 1 byte of storage, **value2** is automatically located at offset 0001:

value1 BYTE 10h  
value2 BYTE 20h

The DB directive can also define an 8-bit variable, signed or unsigned:

|  |  |
| --- | --- |
| val1 DB 255  val2 DB -128 | ; unsigned byte ; signed byte |

***Defining Strings***To define a string of characters, enclose them in single or double quotation marks. The most common type of string ends with a null byte (containing 0). Called a *null-terminated* string, strings of this type are used in many programming languages:

greeting1 BYTE "Good afternoon",0  
greeting2 BYTE 'Good night',0

Each character uses a byte of storage. Strings are an exception to the rule that byte values must be separated by commas. Without that exception, **greeting1** would have to be defined as

greeting1 BYTE 'G','o','o','d' ....etc.

which would be exceedingly tedious. A string can be divided between multiple lines without  
having to supply a label for each line:

greeting1 BYTE "Welcome to the Encryption Demo program "  
BYTE "created by Kip Irvine.",0dh,0ah  
BYTE "If you wish to modify this program, please "  
BYTE "send me a copy.",0dh,0ah,0

The hexadecimal codes 0Dh and 0Ah are alternately called CR/LF (carriage-return line-feed) or *end-of-line characters*. When written to standard output, they move the cursor to the left column of the line following the current line.

The line continuation character (\) concatenates two source code lines into a single statement. It must be the last character on the line. The following statements are equivalent:

greeting1 BYTE "Welcome to the Encryption Demo program "  
and  
greeting1 \  
BYTE "Welcome to the Encryption Demo program "

**Defining WORD and SWORD Data**The WORD (define word) and SWORD (define signed word) directives create storage for one or  
more 16-bit integers:

|  |  |
| --- | --- |
| word1 WORD 65535  word2 SWORD -32768  word3 WORD ? | ; largest unsigned value ; smallest signed value ; uninitialized, unsigned |

The legacy DW directive can also be used:

|  |  |
| --- | --- |
| val1 DW 65535  val2 DW -32768 | ; unsigned ; signed |

**Defining DWORD and SDWORD Data**The DWORD (define doubleword) and SDWORD (define signed doubleword) directives allocate storage for one or more 32-bit integers:

|  |  |
| --- | --- |
| val1 DWORD 12345678h  val2 SDWORD 2147483648  val3 DWORD 20 DUP(?) | ; unsigned ; signed ; unsigned array |

The legacy DD directive can also be used:

|  |  |
| --- | --- |
| val1 DD 12345678h  val2 DD 2147483648 | ; unsigned ; signed |

The DWORD can be used to declare a variable that contains the 32-bit offset of another variable.  
Below, **pVal** contains the offset of **val3**:

pVal DWORD val3

**Defining QWORD Data**  
The QWORD (define quadword) directive allocates storage for 64-bit (8-byte) values:

quad1 QWORD 1234567812345678h

The legacy DQ directive can also be used:

quad1 DQ 1234567812345678h

Similarly, rest of the data types can be used.

Lab 2

1. **Division Operation:**

|  |  |  |  |
| --- | --- | --- | --- |
| Mnemonics | Syntax | Working | Example |
| div | div op1 | Divide value of eax register with op1 and quotient goes to eax and remainder goes to edx. | mov b, 5  mov eax, 10  div b  10 will be divided by 5 and quotient 2 will be stored in eax and remainder 0 in edx. |

**ReadInt:**

ReadInt is used to capture integer data entered by user into eax register.

Example: *Ask user to enter two number and divide first number with second and print quotient and remainder.*

INCLUDE Irvine32.inc

.data

a dword ?

b dword ?

.code

main PROC

call readint

mov a, eax

call readint

mov b, eax

mov eax, a

mov edx, 0

div b

call writeint

mov eax,edx

call writeint

call readint

exit

main ENDP

END main

1. **Procedure:**

**writestring:**

This is used to print string on console. To do that, offset of string is copied to edx folder. As a string is a consecutive array of character stored in consecutive memory locations (bytes), we first move offset of first byte to edx then the writestring will print all characters of strings.

Example: *Print Hello World*

INCLUDE Irvine32.inc

.data

string BYTE "Hello World",0

.code

main PROC

mov edx,offset string

call WriteString

call readint

exit

main ENDP

END main

# Lab 4 – Arrays

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

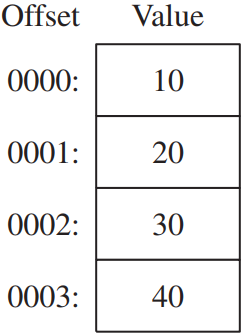
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO2** | Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Multiple Initializers (or Arrays)

If multiple initializers are used in the same data definition (making a list of data), its label refers only to the offset of the first initializer. Other members can be accessed by incrementing the offset of first member. In the following example, assume data is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003. It is just like array of bytes, all bytes are consecutive and accessible through a common mechanism.

**data BYTE 10,20,30,40**



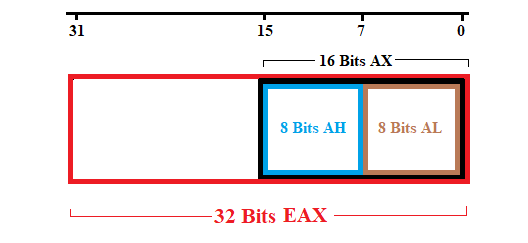
In Assembly language, to access integer 40 stored in 4th byte of the list, we will increment its label “**data**”by 4. I,e.

data+1 refers to offset 0001

data+2 refers to offset 0002

data+3 refers to offset 0003

data+4 refers to offset 0004

Recall Lab 1, first 8 bits of EAX register are called AL register. Byte variable can only be stored in AL register, because both have same size. To print a byte (a number) from the list “**data**”, offset of the respective byte will be moved to AL register. Before doing this, we will remove garbage value from EAX

Register by storing 0 in it. This is done because the incoming byte will only overwrite first 8 bits of EAX register. Rest of the 24 bits will still have garbage value. After byte has been moved to AL, **WriteInt** will print the value of EAX register as whole.

Example 1

INCLUDE Irvine32.inc

.data

data byte 10,20,30,40

.code

main PROC

;First Byte = 10

mov eax, 0

mov al,data

call WriteInt

;Second Byte =20

mov al,data+1

call WriteInt

;Third Byte = 30

mov al,data+2

call WriteInt

;Fourth Byte = 40

mov al,data+3

call WriteInt

call readint

exit

main ENDP

END main

**Output  
+10+20+30+40**

# Strings

Strings works just like arrays we discussed above. A string is an array of characters and each character is represented by a byte. Each character is represented by ASCII code (ranging from 0 to 255) which occupy a single byte. The only difference is that in array, integer values are separated by commas. While in string, characters are not separated by commas but encapsulated in quotations.

We already knew that by moving the first offset of string in EDX register and calling **WriteString** results into printing of string on console. But how can we access each letter of array? By doing the same as we did earlier for integers. But instead of characters, their respective ASCII codes will print on console. Jut replace **data BYTE 10,20,30,40** with **str BYTE “abcd”.** Console will print ASCII codes of a,b,c and d.

## WriteChar:The WriteChar procedure writes a single character to the console window. Pass the character (or its ASCII code) in AL. Sample call:

|  |  |
| --- | --- |
| **mov al,'A'** |  |
| **call WriteChar** | **; displays: "A"** |

Example 2

INCLUDE Irvine32.inc

.data

data byte “abcd”,0

.code

main PROC

;First Byte = a

mov eax, 0

mov al,data

call WriteInt

;Second Byte = b

mov al,data+1

call WriteInt

;Third Byte = c

mov al,data+2

call WriteChar

;Fourth Byte = d

mov al,data+3

call WriteChar

call readint

exit

main ENDP

END main

Output

+97+98cd

# DUP Operator The *DUP operator* allocates storage for multiple data items, using a constant expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data:

Syntax:

# name BYTE 20 DUP(0)

An array with size of 20 bytes, initialized with 1 byte of 0 , 20 times.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bytes** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# name BYTE 4 DUP(1,2,3,4)

An array with size of 16, initialized with 4 bytes of (1,2,3,4). 4 times.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bytes** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| Value | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |

# name BYTE 2 DUP(“STACK”)

An array with size of 10, initialized with 5 bytes of (“STACK”). 2 times. (As one character is one byte)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bytes** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Value | S | T | A | C | K | S | T | A | C | K |

# name BYTE 4 DUP(1,2)

An array with size of 8, initialized with 2 bytes of (1,2). 4 times.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bytes** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Value | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 |

In DUP allocation, length of the array or string is pre-defined. We can access these elements in the same way we accessed elements earlier in previous two examples.

# SizeOf

This procedure returns the size of an array or list.

Example 3

INCLUDE Irvine32.inc

.data

data byte 1,2,3,4

.code

main PROC

mov eax, sizeof data

call WriteInt

call readint

exit

main ENDP

END main

Output  
4

Strings are terminated with 0. This 0 is a part of string, so if we find the size of a string “abcd”, its size will be five instead of 4. Similary if we end our string with 0dh , 0ah, the size of string will be 6.

Example 4

INCLUDE Irvine32.inc

.data

data byte "abcd",0

.code

main PROC

mov eax, sizeof data

call WriteInt

call readint

exit

main ENDP

END main

output  
+5

Example of DUP with SizeOf : Example 5

INCLUDE Irvine32.inc

.data

data byte 4 dup(1,2,3)

.code

main PROC

mov eax, sizeof data

call WriteInt

call readint

exit

main ENDP

END main

output

+12

# Lab 5 – Flags

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Overlapping Values:**

In previous lab, you have seen that MOV procedure require both operands to be of same size. That’s mean, we can’t copy a BYTE into AX or EAX but only in AL. When **oneByte** is moved to AL, it overwrites the existing value of AL but all values of other bytes remain same. When **oneDword** is moved to EAX, it overwrites AX. If 0 is moved to AX, all values of lower 16 bits of EAX will become 0. Try debugging and viewing register values with following program.

***“DumpRegs*** *print the current value of all registers and Flags.*

*Values printed by DumpRegs are in Hexdecimal, from now and on, we will always use hexadecimal values prefixed with ‘h’ to avoid consufing.”*

Example - OVERLAPPING

INCLUDE Irvine32.inc

.data

oneByte BYTE 78h

oneWord WORD 1234h

oneDword DWORD 12345678h

.code

main PROC

mov eax,0 ; EAX = 00000000h  
 call DumpRegs

mov al,oneByte ; EAX = 00000078h

call DumpRegs

mov ax,oneWord ; EAX = 00001234h

call DumpRegs

mov eax,oneDword ; EAX = 12345678h

call DumpRegs

mov ax,0 ; EAX = 12340000h

call DumpRegs

exit

main ENDP

END main

Copying Smaller Values to Larger OnesAlthough MOV cannot directly copy data from a smaller operand to a larger one, programmers can create workarounds. Suppose count (unsigned, 16 bits) must be moved to ECX (32 bits). We can set ECX to zero and move count to CX:

.data   
 count WORD 1  
 .code  
 mov ecx,0  
 mov cx,count  
  
What happens if we try the same approach with a signed integer equal to 16?  
  
 .data  
 signedVal SWORD -16 ; FFF0h (-16)  
 .code  
 mov ecx,0  
 mov cx,signedVal ; ECX = 0000FFF0h (+65,520)  
The value in ECX (65,520) is completely different from 16. On the other hand, if we had filled ECX first with FFFFFFFFh and then copied signedVal to CX, the final value would have been correct:  
  
 mov ecx,0FFFFFFFF  
 mov cx,signedVal ; ECX = FFFFFFF0h (-16)  
  
The effective result of this example was to use the highest bit of the source operand (1) to fill the upper 16 bits of the destination operand, ECX. This technique is called sign extension. Of course, we cannot always assume that the highest bit of the source is a 1. Fortunately, the engineers at Intel anticipated this problem when designing the Intel386 processor and introduced the MOVZX and MOVSX instructions to deal with both unsigned and signed integers.

# MOVZX Instruction

The MOVZX instruction (*move with zero-extend*) copies the contents of a source operand into a destination operand and zero-extends the value to 16 or 32 bits. This instruction is only used with unsigned integers. There are three variants:

**Format**

|  |  |
| --- | --- |
| MOVZX | *reg32,reg/mem8* |
| MOVZX | *reg32,reg/mem16* |
| MOVZX | *reg16,reg/mem8* |

|  |  |
| --- | --- |
| *\*reg/mem8 =* | 8-bit operand, which can be an 8-bit general register or memory byte |
| *\*reg/mem16 =* | 16-bit operand, which can be a 16-bit general register or memory word |

32 bit-operand can’t be copied to any destination using MOVZX. Because it is the job of MOV. In each of the three variants, the first operand (a register) is the destination and the second is the source. In following example, a BYTE (111) is moved in EAX. AL part of EAX will have value of that moved BYTE (111) and rest part of the EAX will automatically become zero.

Example - MOVZX

INCLUDE Irvine32.inc

.data

oneDword Dword 12345678h

oneByte Byte 11h

.code

main PROC

mov eax, oneDword ; EAX = 123456h

call DumpRegs

movzx eax, oneByte ; EAX = 000011h

call DumpRegs

call ReadInt

exit

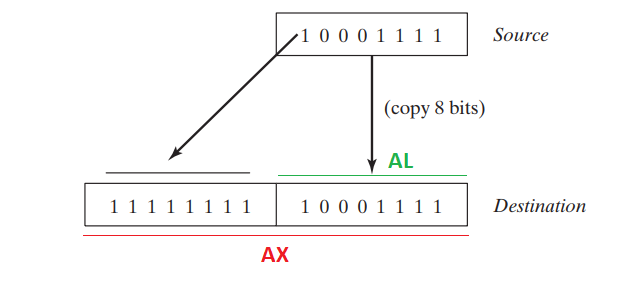
main ENDP

END main

# MOVSX Instruction

The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction just work like MOVZX, instead of copying 0, it copy (1 for negative 0 for positive) in all other bytes. In singed integers, Left most bit is reserved for sign. SBYTE (Signed-Byte) have left most bit of sign and rest 7 bits for storing data.   
So, SBYTE can store -128 to +127. While unsinged (BYTE) can store 0 to 255.

If we copy 10001111 from BYTE to AX (16-bit register). AL will contain 10001111 and the rest part of AX will have 1 in it.



Example - MOVSX

INCLUDE Irvine32.inc

.data

oneDword SDword 00101111100111001001001010111100b

oneByte SBYTE 10001001b

.code

main PROC

mov eax, oneDword ; EAX = 00101111100111001001001010111100

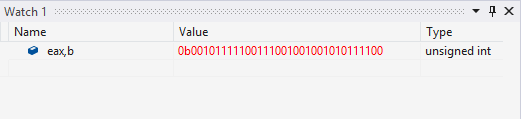
movsx eax, oneByte ; EAX = 11111111111111111111111110001001

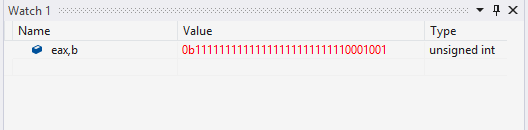
call ReadInt

exit

main ENDP

END main





# XCHG Instruction

The XCHG (exchange data) instruction exchanges the contents of two operands. There are three variants:

**Format**

XCHG *reg,reg*XCHG *reg,mem*XCHG *mem,reg*

The rules for operands in the XCHG instruction are the same as those for the MOV instruction except that XCHG does not accept immediate operands. In array sorting applications, XCHG provides a simple way to exchange two array elements. XCHG doesn’t work either if both operands are variables.

Example – Operand Exchange

INCLUDE Irvine32.inc

.data

a dword 1111h

b dword 2222h

.code

main PROC

Mov Eax,a

Mov Ebx,b

Call DumpRegs

Xchg Eax,Ebx

Call DumpRegs

Call ReadInt

Exit

main ENDP

END main

# INC and DEC Instructions

The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a single operand.

**Format:**

INC *reg/mem*DEC *reg/mem*

Following Example shows how a value can be incremented and decremented.

Example – Increment Decrement

INCLUDE Irvine32.inc

.data

a dword 1111h

.code

main PROC

Mov Eax,a

Call DumpRegs

Inc Eax

Call DumpRegs

Dec Eax

Call DumpRegs

Call ReadInt

Exit

main ENDP

END main

# Flags

# Flags Affected by Addition and Subtraction

When executing arithmetic instructions, we often want to know something about the result. Is it negative, positive, or zero? Is it too large or too small to fit into the destination operand? Answers to such questions can help us detect calculation errors that might otherwise cause erratic program behavior. We use the values of CPU status flags to check the outcome of arithmetic operations.

We also use status flag values to activate conditional branching instructions, the basic tools of program logic. Here’s a quick overview of the status flags.

**•** The **Carry flag** indicates unsigned integer overflow. For example, if an instruction has an 8-bit destination operand but the instruction generates a result larger than 11111111 binary, the

Carry flag is set.

**•** The **Overflow flag** indicates signed integer overflow. For example, if an instruction has a 16- bit destination operand but it generates a negative result smaller than -32,768 decimal, the

Overflow flag is set.

**•** The **Zero flag** indicates that an operation produced zero. For example, if an operand is subtracted from another of equal value, the Zero flag is set.

**•** The **Sign flag** indicates that an operation produced a negative result. If the most significant bit

(MSB) of the destination operand is set, the Sign flag is set.

**•** The **Parity flag** indicates whether or not an even number of 1 bit occurs in the least significant byte of the destination operand, immediately after an arithmetic or Boolean instruction has executed.

# Zero Flag

The Zero flag is set when the result of an arithmetic operation is zero. The following example show the state of Zero Flag when value of EAX become Zero after subtraction.

Example – Zero Flag

INCLUDE Irvine32.inc

.data

.code

main PROC

Mov Eax,0h

add Eax,10h

Call DumpRegs

Sub Eax, 10h

Call DumpRegs

Call ReadInt

Exit

main ENDP

END main

# Carry Flag

When adding two unsigned integers, the Carry is set to 1 if destination overflow. For example, If 255 is moved in AL and then try to add 1 in it, AL will overflow because it cannot store value greater than 255, CF will set to 1 to represent this overflow.

Example – Carry Flag

INCLUDE Irvine32.inc

.data

a BYTE 255

b BYTE 1

.code

main PROC

Call DumpRegs

Mov al,a

add al,b

Call DumpRegs

Call ReadInt

Exit

main ENDP

END main

On the other hand, if 1 is added to 255 in AX, the sum easily fits into 16 bits and the Carry flag is clear (set to 0).

# Sign Flag

The Sign flag is set when the result of a signed arithmetic operation is negative. In example, we tried to move 5 in EAX and then Subtracted 10 from it, causing a negative answer SF to become 1.

Example – Sign Flag

INCLUDE Irvine32.inc

.data

.code

main PROC

Call DumpRegs

Mov Eax, 5

Sub Eax, 10

Call DumpRegs

Call ReadInt

Exit

main ENDP

END main

# Overflow Flag

The Overflow flag is set when the result of a signed arithmetic operation overflows or underflows the destination operand. Adding 1 in +127 store in AL causes signed overflow.

.data

mov al,+127

add al,1 ;OF = 1

Subtracting 1 from -128 causes Signed underflow. -128 is smallest value that can be store in BYTE or AL.

.data

mov al,-128

sub al,1 ;OF = 1

Example – Overflow Flag (Signed Underflow)

INCLUDE Irvine32.inc

.data

.code

main PROC

Call DumpRegs

Mov AL, -128

Sub AL, 1

Call DumpRegs

Call ReadInt

Exit

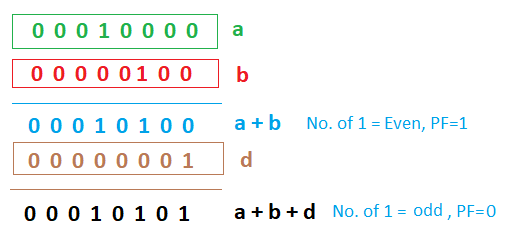
main ENDP

END main

# Parity Flag

The Parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bits, immediately after an arithmetic or Boolean instruction has executed. It get cleared when the result have odd number of 1 bits.

Consider following example, Result of adding two bytes have 2 bits set to one, which is even and the PF=1. If we add another byte in it, the resulting byte will have three 1 bits, making PF=0.



Check the code of this example on the next page.

Example – Parity Flags

INCLUDE Irvine32.inc

.data

a BYTE 010000b

b BYTE 000100b

d BYTE 000001b

.code

main PROC

Mov AL,a

Add AL,b

Call DumpRegs

Add AL,d

Call DumpRegs

Call ReadInt

Exit

main ENDP

END main

# Lab 6 - Conditional Statements and Loops

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Code Labels**

A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

target:

|  |  |
| --- | --- |
| mov  ... jmp | ax,bx  target |
|  |  |

A code label can share the same line with an instruction, or it can be on a line by itself:

|  |  |
| --- | --- |
| L1: mov  L2: | ax,bx |

Label names are created using the rules for identifiers discussed in Section 3.1.7. You can use the same code label more than once in a program if each label is unique within its enclosing procedure. (A procedure is like a function.)

Include Irvine32.inc

.data

msg1 byte "Hello",0

msg2 byte "World",0

msg3 byte "Exit",0

.code

main proc

mov edx, offset msg1

call writestring

jmp M3

M2:

mov edx, offset msg2

call writestring

jmp M4

M3:

mov edx, offset msg3

call writestring

jmp M2

M4:

call readint

exit

main endp

end main

Loop Instruction

Loop instruction uses ECX register as iterator. Loop instruction keeps on executing a specific label until the value of ECX register become zero. With each iteration, value of ECX get decremented automatically.

LOOP label

**Example: Print “Hello” five times:**

Include Irvine32.inc

.data

msg1 byte "Hello",0

.code

main proc

mov ecx, 5

M1:

mov edx, offset msg1

call writestring

loop M1

call readint

exit

main endp

end main

**Nested Loops**

When dealing with nested loops, ECX Is used for both loops: inner and outer. Therefore, it is mandatory to store the status of outer loop before entering in inner and restore the status after exiting.

**Example:**

INCLUDE Irvine32.inc

.data

count Dword ?

prompt Byte 0dh,0ah,"Pakistan ",0

prompt1 Byte 0dh,0ah,"ZindaBd ",0

.code

main PROC

mov ecx, 5

Loop1:

mov edx, OFFSET prompt

call WriteString

mov count, ecx

mov ecx, 3

Loop2:

mov edx, OFFSET prompt1

call WriteString

LOOP Loop2

mov ecx, count

LOOP Loop1

call readInt

exit

main ENDP

END main

**PTR Operator**

You can use the PTR operator to override the declared size of an operand. This is only necessary when you’re trying to access the variable using a size attribute that’s different from the one used to declare the variable. Suppose, for example, that you would like to move the lower 16 bits of a doubleword variable named **myDouble** into AX. The assembler will not permit the following move because the operand sizes do not match:

.data  
myDouble DWORD 12345678h  
.code  
mov ax,myDouble ; error

But the WORD PTR operator makes it possible to move the low-order word (5678h) to AX:

mov ax,WORD PTR myDouble

Why wasn’t 1234h moved into AX? x86 processors use the *little endian* storage format in which the low-order byte is stored at the variable’s starting address.

**Example: Print an array of Five integers:**

Include Irvine32.inc

.data

arr byte 1,2,3,4,5

.code

main proc

mov ecx, lengthof arr

mov esi, offset arr

L1:

movzx eax,byte ptr [esi]

call writeint

inc esi

loop L1

call readint

exit

main endp

end main

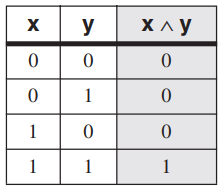
AND InstructionThe AND instruction performs a boolean (bitwise) AND operation between each pair of matching bits in two operands and places the result in the destination operand:

AND destination,source

The following operand combinations are permitted:

AND reg,regAND reg,memAND reg,immAND mem,regAND mem,imm

The operands can be 8, 16, or 32 bits, and they must be the same size. For each matching bit in the two operands, the following rule applies: If both bits equal 1, the result bit is 1; otherwise, it is 0.



The AND instruction lets you clear 1 or more bits in an operand without affecting other bits. The technique is called bit *masking*, much as you might use masking tape when painting a house to cover areas (such as windows) that should not be painted. Suppose, for example, that a control byte is about to be copied from the AL register to a hardware device. Further, we will assume that the device resets itself when bits 0 and 1 are cleared in the control byte. If we want to reset the device without modifying any other bits in AL, we can write the following:

**Example: 11111111b AND 11111100b**

Include Irvine32.inc

.data

.code

main proc

mov al, 11111111b

call writebin ; writing Bin

call crlf ; new line

AND al, 11111100b

call writebin ; writing Bin

exit

main endp

end main

**Result:**



***Flags***

The AND instruction always clears the Overflow and Carry flags.

**Converting Characters to Upper Case**The AND instruction provides an easy way to translate a letter from lowercase to uppercase. If we compare the ASCII codes of capital **A** and lowercase **a**, it becomes clear that only bit 5 is different:

0 1 **1** 0 0 0 0 1 = 61h ('a')  
0 1 **0** 0 0 0 0 1 = 41h ('A')

The rest of the alphabetic characters have the same relationship.

Include Irvine32.inc

.data

char byte "a"

.code

main proc

mov al, char

and al, 11011111b

call writechar

call readint

exit

main endp

end main

**Output: A**

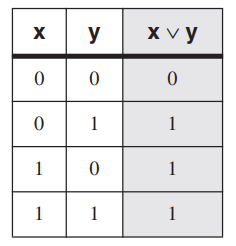
**OR Instruction**The OR instruction performs a boolean OR operation between each pair of matching bits in two operands and places the result in the destination operand:

OR *destination,source*

The OR instruction uses the same operand combinations as the AND instruction:

OR *reg,reg*OR *reg,mem*OR *reg,imm*OR *mem,reg*OR *mem,imm*

The operands can be 8, 16, or 32 bits, and they must be the same size. For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1.



The OR instruction is particularly useful when you need to set 1 or more bits in an operand without affecting any other bits.

**Example: 11001101 OR 11111100**

Include Irvine32.inc

.data

.code

main proc

mov al, 11001101b

call writebin ; writing Bin

call crlf ; new line

OR al, 11111100b

call writebin ; writing Bin

call readint

exit

main endp

end main

**Output**



***Flags***

The OR instruction always clears the Carry and Overflow flags.

**Converting Characters to lower Case**

In the previous example of converting a character from lower case to upper. By ORing 00100000 with the character will result it to become lower case.

**Sets**

AND instruction can be used as intersection of sets. Where sets are binary numbers. Similarly, OR instruction can be used as Union of sets.

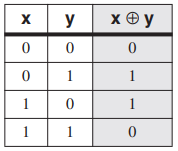
**XOR Instruction**

The XOR instruction performs a boolean exclusive-OR operation between each pair of matching bits in two operands and stores the result in the destination operand:

XOR *destination,source*

The XOR instruction uses the same operand combinations and sizes as the AND and OR instructions. For each matching bit in the two operands, the following applies: If both bits are the same (both 0 or both 1), the result is 0; otherwise, the result is 1. The following truth table describes the boolean expression

**x** ⊕ **y**:



***Flag***

The XOR instruction always clears the Overflow and Carry flags.

**NOT Instruction**The NOT instruction toggles (inverts) all bits in an operand. The result is called the *one’s complement*. The following operand types are permitted:

NOT *reg*NOT *mem*

For example, the one’s complement of F0h is 0Fh:

mov al,11110000b  
not al ; AL = 00001111b

***Flags***

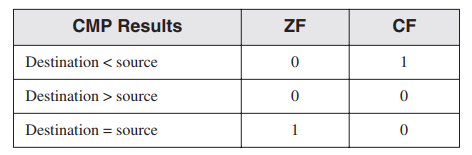
No flags are affected by the NOT instruction.

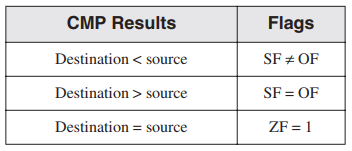
**CMP Instruction**

The CMP (compare) compares destination with source and sets some flags. Neither operand is modified:

CMP *destination,source*

Following registers are modified.



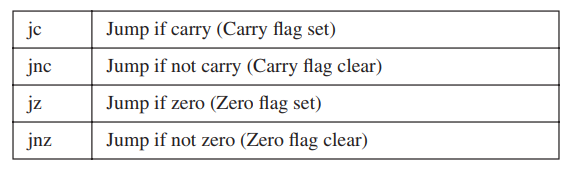


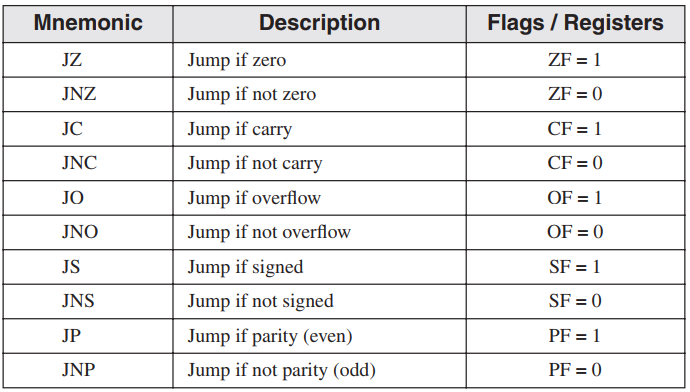
**Conditional Structures**There are no explicit high-level logic structures in the x86 instruction set, but you can implement them using a combination of comparisons and jumps. Two steps are involved in executing a conditional statement:

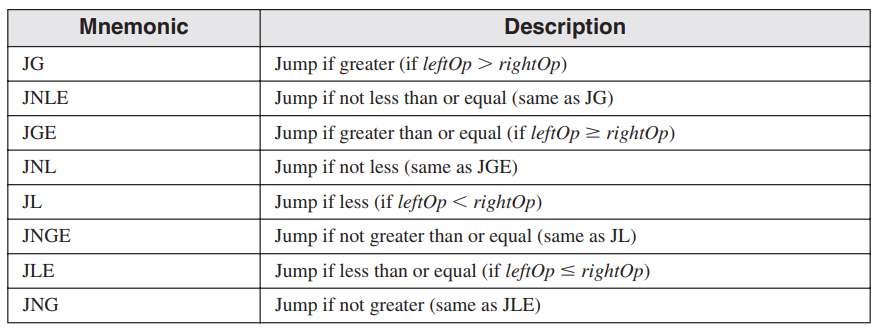
First, an operation such as CMP, AND, or SUB modifies the CPU status flags.

Second, a conditional jump instruction tests the flags and causes a branch to a new address.

J*cond destination*







**Example:**

Include Irvine32.inc

.data

char byte "c"

msg1 byte "Both are equal",0

msg2 byte "entered char is smaller",0

msg3 byte "entered char is bigger",0

.code

main proc

call readchar

cmp al, char

je L1

jl L2

jg L3

L1:

mov edx, offset msg1

call writestring

jmp \_exit

L2:

mov edx, offset msg2

call writestring

jmp \_exit

L3:

mov edx, offset msg3

call writestring

jmp \_exit

\_exit:

call readint

exit

main endp

end main

**Additional conditional Loops**

These loop statement act similar to ECX value, additionally they have following condition to meet in order to execute.

**LOOPZ loop if Zero flag = 0**

**LOOPE loop if Zero Flag = 1**

**LOOPNZ loop if Zero flag = 1**

**LOOPNE loop if Zero flag = 0**

# Lab 7- Procedures

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

CreateOutputFile: (Irvine32 only) The CreateOutputFile procedure creates a new disk file and opens it for writing. When you call the procedure, place the offset of a filename in EDX. When the procedure returns, EAX will contain a valid file handle (32-bit integer) if the file was created successfully. Otherwise, EAX equals INVALID\_HANDLE\_VALUE (a predefined constant).  
**Sample call:**

.data  
filename BYTE "newfile.txt",0  
.code  
mov edx,OFFSET filename  
call CreateOutputFile

OpenInputFile: (Irvine32 only) The OpenInputFile procedure opens an existing file for input. Pass it the offset of a filename in EDX. When it returns, if the file was opened successfully, EAX will contain a valid file handle. Otherwise, EAX will equal INVALID\_HANDLE\_VALUE (a predefined constant).  
**Sample call:**

.data  
filename BYTE "myfile.txt",0  
.code  
mov edx,OFFSET filename  
call OpenInputFile

**ReadFromFile** (*Irvine32 only)* The ReadFromFile procedure reads an input disk file into a memory buffer. When you call ReadFromFile, pass it an open file handle in EAX, the offset of a buffer in EDX, and the maximum number of bytes to read in ECX. When ReadFromFile returns, check the value of the Carry flag: If CF is clear, EAX contains a count of the number of bytes read from the file. But if CF is set, EAX contains a numeric system error code. You can call the WriteWindowsMsg procedure to get a text representation of the error. In the following example, as many as 5000 bytes are copied from the file into the buffer variable:

.data  
BUFFER\_SIZE = 5000  
buffer BYTE BUFFER\_SIZE DUP(?)  
bytesRead DWORD ?  
.code

|  |  |  |
| --- | --- | --- |
| mov  mov | edx,OFFSET buffer  ecx,BUFFER\_SIZE | ; points to buffer ; max bytes to read ; read the file |
| call ReadFromFile |  |  |

If the Carry flag were clear at this point, you could execute the following instruction:

mov bytesRead,eax ; count of bytes actually read

But if the Carry flag were set, you would call WriteWindowsMsg procedure, which displays a string that contains the error code and description of the most recent error generated by the  
application:

call WriteWindowsMsg

WriteToFile (Irvine32 only) The WriteToFile procedure writes the contents of a buffer to an output file. Pass it a valid file handle in EAX, the offset of the buffer in EDX, and the number of bytes to write in ECX. When the procedure returns, if EAX is greater than zero, it contains a count of the number of bytes written; otherwise, an error occurred. The following code calls WriteToFile:

BUFFER\_SIZE = 5000  
.data  
fileHandle DWORD ?  
buffer BYTE BUFFER\_SIZE DUP(?)  
.code  
mov eax,fileHandle  
mov edx,OFFSET buffer  
mov ecx,BUFFER\_SIZE  
call WriteToFile

**CloseFile** (*Irvine32 only)* The CloseFile procedure closes a file that was previously created or opened (see CreateOutputFile and OpenInputFile). The file is identified by a 32-bit integer *handle*, which is passed in EAX. If the file is closed successfully, the value returned in EAX will be nonzero. Sample call:

|  |  |
| --- | --- |
| mov  call | eax,fileHandle CloseFile |

WriteWindowsMsg (Irvine32 only) The WriteWindowsMsg procedure displays a string containing the most recent error generated by your application when executing a call to a system function. Sample call:

call WriteWindowsMsg  
  
The following is an example of a message string:  
Error 2: The system cannot find the file specified.

**MsgBox** *(Irvine32 only)* The MsgBox procedure displays a graphical popup message box with an optional caption. (This works when the program is running in a console window.) Pass it the offset of a string in EDX, which will appear in the inside the box. Optionally, pass the offset of a string for the box’s title in EBX. To leave the title blank, set EBX to zero. Sample call:

.data  
caption db "Dialog Title", 0  
HelloMsg BYTE "This is a pop-up message box.", 0dh,0ah  
BYTE "Click OK to continue...", 0  
.code  
mov ebx,OFFSET caption  
mov edsx,OFFSET HelloMsg  
call MsgBox

Sample output:

**MsgBoxAsk** *(Irvine32 only)* The MsgBoxAsk procedure displays a graphical popup message box with Yes and No buttons. (This works when the program is running in a console window.) Pass it the offset of a question string in EDX, which will appear in the inside the box. Optionally, pass the offset of a string for the box’s title in EBX. To leave the title blank, set EBX to zero. MsgBoxAsk returns an integer in EAX that tells you which button was selected by the user. The value will be one of two predefined Windows constants: IDYES (equal to 6) or IDNO (equal to 7). Sample call:

.data  
caption BYTE "Survey Completed",0  
question BYTE "Thank you for completing the survey."  
BYTE 0dh,0ah  
BYTE "Would you like to receive the results?",0  
.code  
mov ebx,OFFSET caption  
mov edx,OFFSET question  
call MsgBoxAsk  
;(check return value in EAX)

**ReadString***:* The ReadString procedure reads a string from the keyboard, stopping when the user presses the Enter key. Pass the offset of a buffer in EDX and set ECX to the maximum number of characters the user can enter, plus 1 (to save space for the terminating null byte). The procedure returns the count of the number of characters typed by the user in EAX. Sample call:

.data

|  |  |
| --- | --- |
| buffer BYTE 21 DUP(0)  byteCount DWORD ?  .code | ; input buffer ; holds counter |
| mov  mov  call  mov | edx,OFFSET buffer  ecx,SIZEOF buffer  ReadString  byteCount,eax | ; point to the buffer ; specify max characters ; input the string ; number of characters |

ReadString automatically inserts a null terminator in memory at the end of the string. The following is a hexadecimal and ASCII dump of the first 8 bytes of **buffer** after the user has entered the string “ABCDEFG”: The variable **byteCount** equals 7.

**Random32:**The Random32 procedure generates and returns a 32-bit random integer in EAX. When called repeatedly, Random32 generates a simulated random sequence. The numbers are created using a simple function having an input called a *seed*. The function uses the seed in a formula that generates the random value. Subsequent random values are generated using each previously generated random value as their seeds. The following code snippet shows a sample call to Random32:

.data  
randVal DWORD ?  
.code

|  |  |
| --- | --- |
| call  mov | Random32 randVal,eax |

Random32 is also available in the Irvine16 library, returning its value in EAX. *Randomize* The Randomize procedure initializes the starting seed value of the Random32 and RandomRange procedures. The seed equals the time of day, accurate to 1/100 of a second. Each time you run a program that calls Random32 and RandomRange, the generated sequence of random numbers will be unique. You need only to call Randomize once at the beginning of a program. The following example produces 10 random integers:

|  |  |
| --- | --- |
| call  mov  L1: call | Randomize ecx,10 Random32 |
|  | ; use or display random value in EAX here... |
| loop | L1 |

**Randomize**The Randomize procedure initializes the starting seed value of the Random32 and RandomRange procedures. The seed equals the time of day, accurate to 1/100 of a second. Each time you run a program that calls Random32 and RandomRange, the generated sequence of random numbers will be unique. You need only to call Randomize once at the beginning of a program. The following example produces 10 random integers:

|  |  |
| --- | --- |
| call  mov  L1: call | Randomize ecx,10 Random32 |
|  | ; use or display random value in EAX here... |
| loop | L1 |

**RandomRange**The RandomRange procedure produces a random integer within the range of 0 to *n*  1, where *n* is an input parameter passed in the EAX register. The random integer is returned in EAX. The following example generates a single random integer between 0 and 4999 and places it in a variable named *randVal*.

.data  
randVal DWORD ?  
.code

|  |  |
| --- | --- |
| mov | eax,5000 |
| call RandomRange |  |
| mov | randVal,eax |

**Custom Functions/Procedures:**

Format:

<name> PROC

; code  
 ret ; return

<name> ENDP

# Lab 8 -Procedures with param

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Invoke**:

This directive works like call but it can pass parameters to the procedure. It pushes the parameters on stack and call the procedure.

ADDR OperatorThe ADDR operator can be used to pass a pointer argument when calling a procedure using INVOKE. The following INVOKE statement, for example, passes the address of myArray to the FillArray procedure:

INVOKE FillArray, Param1, Param2, ADDR myArray

**PROTO Directive**The PROTO directive creates a prototype for an existing procedure. A *prototype* declares a procedure’s name and parameter list. It allows you to call a procedure before defining it and to verify that the number and types of arguments match the procedure definition. (The C and C++ languages use function prototypes to validate function calls at compile time.)

**Uses Reg**

**Example – 1 : Add two Number**

TITLE MASM Template (main.asm)

INCLUDE Irvine32.inc

.data

var1 Dword 5

var2 Dword 10

buffer dword 0

.code

addTwo proto, val1:DWORD, Val2:Dword

main PROC

invoke addTwo, var1,var2

;mov buffer, eax

call writeint

exit

main ENDP

addTwo proc uses ecx, val1:DWORD, Val2:Dword

mov eax, val1

add eax, val2

mov ebx, eax

ret

addTwo ENDP

END main

**Example – 2 : Sum of Array**

TITLE MASM Template (main.asm)

INCLUDE Irvine32.inc

.data

arr dword 5,5,5,5,5

arrLen dword ?

.code

sumArray proto, arr: ptr dword, arSize:dword

main PROC

mov eax, lengthof arr

mov arrLen, eax

invoke sumArray, Addr arr, arrlen

call writeint

exit

main ENDP

sumArray proc, ar: ptr dword, arLen:dword

mov esi,ar ; address of the array

mov ecx,arLen ; size of the array

mov eax,0 ; set the sum to zero

cmp ecx,0 ; length = zero?

je L2

L1: add eax,[esi] ; add each integer to sum

add esi,4 ; point to next integer

loop L1 ; repeat for array size

L2: ret

sumArray ENDP

END main

# Lab 9- Floating Point

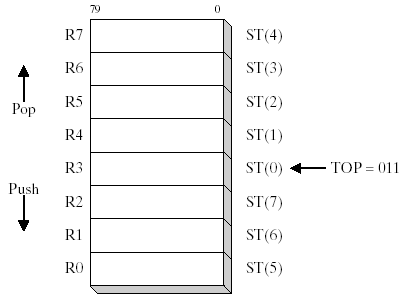
**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO3** | Analyse hardware control and communication using BIOS and DOS interrupts and system calls. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

Floating Point:

There are 8 separate registers for floating point data, named R0 to R7. Three-bit field named TOP in the FPU status word identifies the register number that is currently the top of stack. FPU stack can be viewed by calling **ShowFPUStack.**

****

Data Types:

There are 3 date types dedicated for representing floating point units.

REAL4 32 Bits, 1 for sign, 8 for exponent, 23 bits for fractional part.  
REAL8 64 Bits, 1 for sign, 11 for exponent, 52 bits for fractional part.  
REAL10 80 Bits, 1 for sign, 16 for exponent, 63 bits for fractional part.

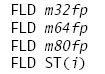
****

FPU Instruction Set:

Operands

* + zero, one, or two
  + no immediate operands
  + no general-purpose registers (EAX, EBX, ...)
  + integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations
  + if an instruction has two operands, one must be a FPU register

**FLD and FILD:**FLD loads a floating point on the top of the stack. FILD convert the integer into floating point value and load it on the stack.  
Allowed format**:**

****

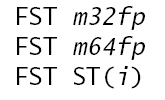
**Example:**

****

**Loading Constants**:

The following instructions load specialized constants on the stack. They have no operands

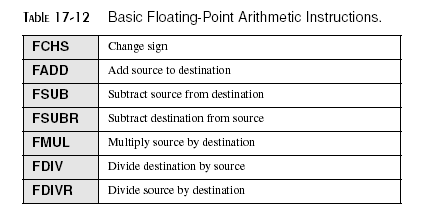
* The **FLD1** instruction pushes 1.0 onto the register stack.
* The **FLDL2T** instruction pushes log2 10 onto the register stack.
* The **FLDL2E** instruction pushes log2 e onto the register stack.
* The **FLDPI** instruction pushes onto the register stack.
* The **FLDLG2** instruction pushes log10 2 onto the register stack.
* The **FLDLN2** instruction pushes loge 2 onto the register stack.
* The **FLDZ** (load zero) instruction pushes 0.0 on the FPU stack.

**FST:**It copies floating point from the top of stack to memory.  
Format:  
  


**FSTP:**  
It copies and pop the top element of FPU stack.

**FBSTP:**

It converts the top element of the stack into TBYTE (10 byte Integer) and then copies it to TBYTE memory and pop.



**ReadFloat:**Get values from console and write it on the top of the stack.

**WriteFloat:**

Print the top value of stack.

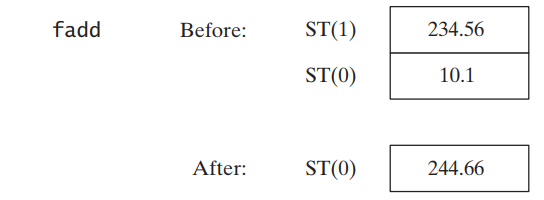
**FCHS and FABS**The FCHS (change sign) instruction reverses the sign of the floating-point value in ST(0). The FABS (absolute value) instruction clears the sign of the number in ST(0) to create its absolute value. Neither instruction has operands:  
  
FCHS  
FABS

**FADD, FADDP, FIADD**The FADD (add) instruction has the following formats, where *m32fp* is a REAL4 memory operand, *m64fp* is a REAL8 operand, and *i* is a register number:

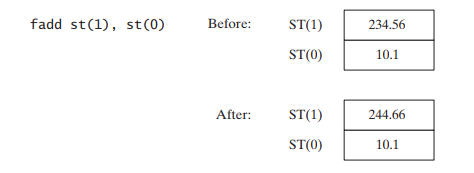
FADD   
 FADD *m32fp* FADD *m64fp*

|  |  |
| --- | --- |
| FADD  FADD | ST(0), ST(*i*) ST(*i*), ST(0) |

***No Operands:***If no operands are used with FADD, ST(0) is added to ST(1). The result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the result on the top of the stack. The following figure demonstrates FADD, assuming that the stack already contains two values:



***Register Operands:***Starting with the same stack contents, the following illustration demonstrates adding ST(0) to ST(1):



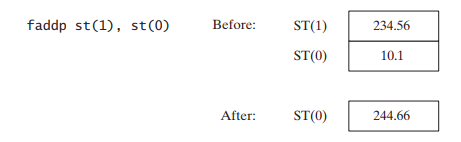
***Memory Operand:***When used with a memory operand, FADD adds the operand to ST(0). Here are examples:

|  |  |
| --- | --- |
| fadd mySingle | ; ST(0) += mySingle ; ST(0) += [esi] |
| fadd | REAL8 PTR[esi] |

**FADDP:**The FADDP (add with pop) instruction pops ST(0) from the stack after performing the addition operation. MASM supports the following format:

FADDP ST(*i*),ST(0)

The following figure shows how FADDP works:



**FIADD:**The FIADD (add integer) instruction converts the source operand to double extended precision floating point format before adding the operand to ST(0). It has the following syntax:

FIADD *m16int*FIADD *m32int*

**Example:**

.data  
myInteger DWORD 1  
.code  
fiadd myInteger ; ST(0) += myInteger

**FSUB, FSUBP, FISUB**The FSUB instruction subtracts a source operand from a destination operand, storing the difference in the destination operand. The destination is always an FPU register, and the source can be either an FPU register or memory. It accepts the same operands as FADD:

|  |  |
| --- | --- |
| FSUB | *m32fp* |
| FSUB  FSUB  FSUB | *m64fp* ST(0), ST(*i*) ST(*i*), ST(0) |

FSUB’s operation is similar to that of FADD, except that it subtracts rather than adds. For example, the no-operand form of FSUB subtracts ST(0) from ST(1). The result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the result on the top of the stack. FSUB with a memory operand subtracts the memory operand from ST(0) and does not pop the stack.

**FSUBP***:* The FSUBP (subtract with pop) instruction pops ST(0) from the stack after performing the subtraction. MASM supports the following format

FSUBP ST(*i*),ST(0)

*FISUB* The FISUB (subtract integer) instruction converts the source operand to double extended precision floating point format before subtracting the operand from ST(0):  
FISUB *m16int*FISUB *m32int*

**FMUL, FMULP, FIMUL**The FMUL instruction multiplies a source operand by a destination operand, storing the product in the destination operand. The destination is always an FPU register, and the source can be a register or memory operand. It uses the same syntax as FADD and FSUB:

FMUL  
FMUL *m32fp*FMUL *m64fp*

|  |  |
| --- | --- |
| FMUL  FMUL | ST(0), ST(*i*) ST(*i*), ST(0) |

**FMUL:** operation is similar to that of FADD, except it multiplies rather than adds. For example, the no-operand form of FMUL multiplies ST(0) by ST(1). The product is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the product on the top of the stack. Similarly, FMUL with a memory operand multiplies ST(0) by the memory operand:

fmul mySingle ; ST(0) \*= mySingle

**FMULP:**The FMULP (multiply with pop) instruction pops ST(0) from the stack after performing the multiplication. MASM supports the following format:

FMULP ST(*i*),ST(0)

**FIMUL:** is identical to FIADD, except that it multiplies rather than adds:

FIMUL *m16int*FIMUL *m32int*

**FDIV, FDIVP, FIDIV**The FDIV instruction divides a destination operand by a source operand, storing the dividend in the destination operand. The destination is always a register, and the source operand can be either a register or memory. It has the same syntax as FADD and FSUB:

FDIV  
 FDIV *m32fp* FDIV *m64fp*

|  |  |  |
| --- | --- | --- |
| FDIV  FDIV |  | ST(0), ST(*i*) ST(*i*), ST(0) |

**FDIV’s:** operation is similar to that of FADD, except that it divides rather than adds. For example, the no-operand form of FDIV divides ST(1) by ST(0). ST(0) is popped from the stack, leaving the dividend on the top of the stack. FDIV with a memory operand divides ST(0) by the memory operand. The following code divides **dblOne** by **dblTwo** and stores the quotient in **dblQuot**:

.data  
dblOne REAL8 1234.56  
dblTwo REAL8 10.0  
dblQuot REAL8 ?  
.code

|  |  |  |
| --- | --- | --- |
| fld  fdiv  fstp | dblOne  dblTwo  dblQuot | ; load into ST(0) ; divide ST(0) by dblTwo ; store ST(0) to dblQuot |

If the source operand is zero, a divide-by-zero exception is generated. A number of special cases apply when operands equal to positive or negative infinity, zero, and NaN are divided. For details, see the Intel Instruction Set Reference manual.

**FIDIV**: The FIDIV instruction converts an integer source operand to double extended-precision  
floating-point format before dividing it into ST(0). Syntax:

|  |  |
| --- | --- |
| FIDIV | m16int |
| FIDIV | m32int |

DDIVP: It just work like FDIV except it pops the stack once.

**Comparing Floating-Point Values**Floating-point values cannot be compared using the CMP instruction—the latter uses integer subtraction to perform comparisons. Instead, the FCOM instruction must be used. After executing FCOM, special steps must be taken before using conditional jump instructions (JA, JB, JE, etc.) in logical IF statements. Since all floating-point values are implicitly signed, FCOM performs a signed comparison.

**FCOM, FCOMP, FCOMPP**   
The FCOM (compare floating-point values) instruction compares ST(0) to its source operand. The source can be a memory operand or FPU register. Syntax:

|  |  |
| --- | --- |
| Instruction | Description |
| FCOM | Compare ST(0) to ST(1) |
| FCOM m32fp | Compare ST(0) to m32fp |
| FCOM m64fp | Compare ST(0) to m64fp |
| FCOM ST(i) | Compare ST(0) to ST(i) |

The **FCOMP** instruction carries out the same operations with the same types of operands, and ends by popping ST(0) from the stack. The **FCOMPP** instruction is the same as that of FCOMP, except it pops the stack one more time.

|  |  |
| --- | --- |
| **Condition** | **Conditional Jump to Use** |
| ST(0) > SRC | JA |
| ST(0) < SRC | JB |
| ST(0) = SRC | JE |

# Lab 10 - File Handling Code

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Recommended Readings:**

Kip Irvine, 6th Edition, 11.1.6 & 11.1.7 topic. This chapter also contains some interesting functions of controlling cursor, console color, position etc. Do read the topics before jumping into the code.

**Create File / Open File / Closing File**

TITLE Creating a File

INCLUDE Irvine32.inc

BUFFER\_SIZE = 501

.data

msg1 byte "File Opened Successfully",0

msg2 byte "File Don't Exist",0

msg3 byte "File closed!",0

filename byte "Test.txt",0

filehandle dword ?

.code

main PROC

INVOKE CreateFile,

ADDR filename, ; ptr to filename

GENERIC\_READ, ; mode = Can read

DO\_NOT\_SHARE, ; share mode

NULL, ; ptr to security attributes

OPEN\_EXISTING, ; open an existing file

FILE\_ATTRIBUTE\_NORMAL, ; normal file attribute

0 ; not used

mov filehandle, eax ; Copy handle to variable

cmp eax, INVALID\_HANDLE\_VALUE ; checking if the handle is valid or not

je L1

jne L2

L1:

mov edx,offset msg2 ; Print if the handle is not valid

call WriteString

jmp L3

L2:

call crlf

mov edx,offset msg1 ; Print if the handle is valid

call WriteString

jmp L3

L3:

invoke CloseHandle, filehandle ; Calling CloseHandle function

; to close the file by passing filehandle

cmp eax, 0 ; eax becomes zero if file is not closed

jne L4

je L5

L4:

call crlf

mov edx, offset msg3 ; print file close message

call writestring

L5:

exit

main ENDP

END main

**Write File:**

You have to open file has GENERIC\_WRITE.

TITLE Creating a File

INCLUDE Irvine32.inc

BUFFER\_SIZE = 501

.data

msg1 byte "HelloHumans",0,0ah,0dh

msg2 byte "Hello Martians",0

filename byte "Test.txt",0

filehandle dword ?

bytesWritten dword 1 dup(0)

.code

main PROC

INVOKE CreateFile,

ADDR filename, ; ptr to filename

GENERIC\_WRITE, ; mode = Can read

DO\_NOT\_SHARE, ; share mode

NULL, ; ptr to security attributes

OPEN\_ALWAYS, ; open an existing file

FILE\_ATTRIBUTE\_NORMAL, ; normal file attribute

0 ; not used

mov filehandle, eax ; Copy handle to variable

INVOKE WriteFile,

filehandle, ; file handle

addr msg2, ; msg to write

sizeof msg2, ; size of bytes to write

addr bytesWritten, ; num bytes written

0

call writeint

invoke CloseHandle,

filehandle

exit

main ENDP

END main

**Read File**

You have to open file has GENERIC\_READ.

TITLE Creating a File

INCLUDE Irvine32.inc

BUFFER\_SIZE = 501

.data

msg1 byte 20 dup(0)

filename byte "Test.txt",0

filehandle dword ?

bytesRead dword 1 dup(0)

.code

main PROC

INVOKE CreateFile,

ADDR filename, ; ptr to filename

GENERIC\_READ, ; mode = Can read

DO\_NOT\_SHARE, ; share mode

NULL, ; ptr to security attributes

OPEN\_ALWAYS, ; open an existing file

FILE\_ATTRIBUTE\_NORMAL, ; normal file attribute

0 ; not used

mov filehandle, eax ; Copy handle to variable

invoke ReadFile,

filehandle, ; file handle

addr msg1, ; where to read

20, ; num bytes to read

addr bytesRead, ; bytes actually read

0

invoke CloseHandle,

filehandle

mov edx, offset msg1 ; Write String

call WriteString

exit

main ENDP

END main

Above codes read and write files form beginning, we can set file pointer to the end of the file to append the file instead of re-writing from start. Similarly, we can read specific line instead of reading whole documents.

**File Pointer:**

**Directly using below code will set the pointer to the end of the file, so we can append it from end.**

INVOKE SetFilePointer,

fileHandle, ; file handle

0, ; distance low

0, ; distance high

FILE\_END ; move method

**Following code moves the pointer on 10th position from start of the beginning.**

INVOKE SetFilePointer,

fileHandle, ; file handle

10, ; distance low

0, ; distance high

FILE\_BEGIN ; move method

# Lab-11 Use of Loops and if Else (Open Ended lab)

**Open Ended Lab (OEL):**

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2**  **CLO4** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set.  Comply with plagiarism guidelines. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

An open-ended lab is where students are given the freedom to develop their own experiments, instead of merely following the already set guidelines from a lab manual or elsewhere.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **CLOs Covered** | **Taxonomy Domain/Level** | **PLOs** |
| **1** | Use assembler, debugger to execute basic assembly language programs. | Cognitive/2 | Modern tool |
| **2** | Construct programs in assembly language using x86 32 and 64 bit instruction set. | Cognitive/2 | Engg Knowledge |
| **3** | Comply with plagiarism guidelines | Affective/3 | Ethics |

**Open Ended Lab Description:**

**Note: Upload zip folder containing .asm file and exe file of your program. Zip folder name should be your roll number.**

**Question 01: [Marks: 2\*5=10]**

**Write an assembly code that prints the steric \* in the following formats:**

1. **B.**

|  |  |
| --- | --- |
| **\***  **\*\***  **\*\*\***  **\*\*\*\***  **\*\*\*\*\***  **\*\*\*\*\*\*** | **\*\*\*\*\*\***  **\*\*\*\*\***  **\*\*\*\***  **\*\*\***  **\*\***  **\*** |

# Lab-12 Open Ended Lab

**Name: ……………………………. Registration No: …………………….**

**Date: ……………………………. Grade and Signature: ………………………**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLO1**  **CLO2**  **CLO4** | Use assembler, debugger to execute basic assembly language programs.  Construct programs in assembly language using x86 32 and 64 bit instruction set.  Comply with plagiarism guidelines. | | | |
| **Psychomotor/Affective** | **Level1 (1)** | **Level2 (2-3)** | **Level3 (4-5)** | **Level4 (6-7)** |
|  |  |  |  |
| **Report Marks (3)** |  | | **Total marks (10)** |  |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

A *listing file* contains a copy of the program’s source code, with line numbers, the numeric address of each instruction, the machine code bytes of each instruction (in hexadecimal), and a symbol table. The symbol table contains the names of all program identifiers, segments, and related information. Advanced programmers sometimes use the listing file to get detailed information about the program. Figure 1 shows a partial listing file for the *AddTwo* program. Let’s examine it in more detail. Lines 1–7 contain no executable code, so they are copied directly from the source file without changes. Line 9 shows that the beginning of the code segment is located at address 00000000 (in a 32-bit program, addresses display as 8 hexadecimal digits). This address is relative to the beginning of the program’s memory footprint, but it will be converted into an absolute memory address when the program is loaded into memory. When that happens, the program might start at an address such as 00040000h. Lines 10 and 11 also show the same starting address of 00000000, because the first executable statement is the MOV instruction on line 11. Notice on line 11 that several hexadecimal bytes appear between the address and the source code. These bytes (B8 00000005) represent the machine code instruction (B8), and the constant 32-bit value (00000005) that is assigned to EAX.

Similarly, other interpretation can be analyzing from this file thus whenever you want to make sure the assembler is generating the correct machine code bytes based on your program, the listing file is your best resource.

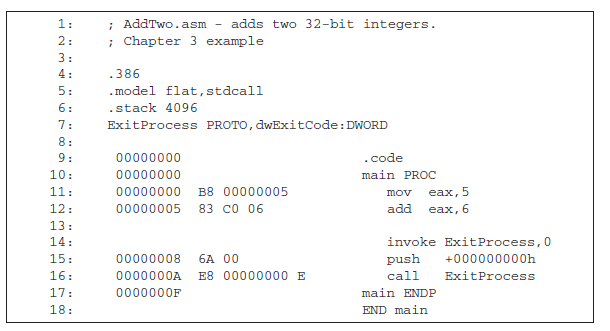


Figure 1

**Expected Outcome:**

Write a program in visual studio that calculates the expression, using registers: A = (A + B) - (C + D). Assign integer values to the EAX, EBX, ECX, and EDX registers. Generate lst file and analyze the content based on above arguments