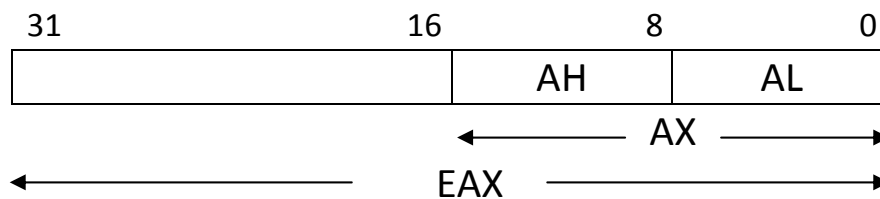


Assembly Language Lab 2

1. 32-bit Protected Mode (80386)

- In 386 CPU, registers become 32-bit wide except segment registers (selectors) remain 16-bits as they are. Two new 16-bit segment registers are also added, FS and GS. (Segment registers are now called *selectors* according to their new function in 80386 protected mode as explained next.)
- This extension made the single segment size up to 4GB (as the upper limit of 32-bit offset registers).



- So how a segment register stores the 32-bit address of memory? It can be solved like in real mode and store only the most significant 16 bits of the address, but doing this adds too hard limitation on the position of the segment in memory.
- Instead, segment register is interpreted differently in protected mode. It is interpreted as an *index* into a *descriptor table* than a register stores the starting address of the segment. Descriptor table is a table containing physical addresses of all segments beside some other information about these segments. It is stored in memory and its location is stored in a special register.
- In addition, protected mode uses a technique called *virtual memory*. The basic idea of virtual memory system is to only keep the data and code in physical memory that programs are currently using. Other data and code are stored temporarily on disk until they are needed again.
- Therefore, each segment is assigned an entry in a descriptor table. This entry has all the information that the system needs to know about that segment. This information includes: *is it currently in memory*; if in memory, *where is it (its physical address)*; *access permissions* (e.g. read-only, read-write). The index of the entry of the segment is the selector value that is stored in the segment register.
- To calculate the physical address from selector-offset pair:
 - 1- Get the segment physical address returned from the descriptor table,
 - 2- Add this address to the offset to get the physical address of the specified location.

- Furthermore, segments can be divided into smaller 4K-sized units called *pages*. The virtual memory system works with pages now instead of segments. This means that only parts of segment may be in memory at any one time.

2. The First Assembly Program

- Assembly language program is a series of statements which are either computer *instructions* or statements called *directives*.
- Each assembly instruction represents exactly one machine instruction. However, directives do not generate any machine code. Directives are just pseudo-instructions which instruct the assembler how to translate the program into machine code.
- The general form of an assembly instruction is:

[label:] mnemonic [operands] [;comment]

Label: allows the program to refer to a line of code by a name. (Actually, it is a name for the address of the machine code of this line.)

Mnemonic and operands: together perform the real work of the program. Mnemonic can be an instruction (like MOV, ADD) or directive (like BYTE, END).

Comment is any set of words preceded by a semi-colon.

- As we know, a program is divided into segments. There are directives that allow a programmer to define the starting and the ending of these segments. .DATA, .CODE, and .STACK directives are used to express the starting of data, code, and stack segments respectively. These directives tell the assembler that the next lines in the program belong to the specified segment. The segment ends by defining another segment directive or by the end of code file.
- *Data segment* is used to define program data. *Stack segment* is used to store data temporarily like saving registers values in the beginning of procedure call. *Code segment* contains the program instructions that do the required work.

3. Skeleton Program

The following code is a skeleton program that specifies the required statements to write an assembly program.

```
.386
.MODEL Flat, STDCALL
.STACK 4096
.DATA
    ;Your initialized data

.CODE
<label>
```

```

        ;Your code
    ret
END <label>

```

Explanation for the above skeleton program:

.386

An assembler directive, telling the assembler to use 80386 instructions set and disable later series instructions (like those of 486, Pentium...).

.MODEL *memory_model*, [*language_type*]

It specifies the program memory model. Allowed models are [TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE, or FLAT]. *language_type* can be [C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL] and used to specify the compatibility with high level language in case there is an integration between the assembly code and another high level language code. The memory models differ in number of code and data segments and referencing type near or far (explained later). The following table shows the differences among memory models:

Memory Model	Default Code	Default Data	# code segments	# data segments
Tiny (DOS)	Near	Near	Single segment for both	
Small	Near	Near	1	1
Medium	Far	Near	multiple	1
Compact	Near	Far	1	multiple
Large	Far	Far	multiple	multiple
Huge	Far	Far	multiple	multiple
Flat (Windows)	Near	Near	Single segment for both	

Flat memory model and **stdcall** language type are often used with win32 assembly programs. We will use it with the most of programs we write during the course.

.STACK [*stack_size*]

A directive defines the stack segment and its size in bytes. The default size is 1024 byte. Nothing can be placed in the stack segment after its definition. .Stack directive just defines the size of the stack segment without allowing placing initial data in it.

.DATA

It defines the data segment where a programmer can place initialized data variables used in the program.

.CODE

It defines the code segment where assembly instructions are placed. The code segment or the main procedure in it must be ended by **ret** instruction (or anything similar that terminates the program execution) which returns control to the OS.

END <label>

End directive indicates the end of assembly code in this file. A label must be specified with END directive to tell the assembler where the program *entry point* exists (i.e. from which line the program should begin its execution).

4. Modified Skeleton Program

Text book we study offers a library which provides some basic operations required in many assembly programs. This library can be used by adding the following line in your program code:

```
INCLUDE Irvine32.inc
```

This line tells the assembler to include all defined functions in this file. It is like **#include** in C++ or **using** in C#. So we use the following modified skeleton program as a template for any program we write later on.

```
INCLUDE Irvine32.inc
.data
.code
main:      ;define a label main

    exit    ;terminate the execution of the program
END main   ;end the assembler work, and make the main as program
           ;entry point
```

Note that the following lines already exist in Irvine32.inc and so they are not written again in our program code.

```
.386
.MODEL Flat, STDCALL
.STACK 4096
```

Also, the code segment or the main procedure in it will be ended by **exit** which returns control to Windows after program terminate. It is an EQU for *exitprocess* API function and declared in Irvine32.inc.

5. Basic Instructions

MOV Instruction

The most basic instruction is the **MOV** instruction. It moves data from one location to another (like the assignment operator in a high-level language). It takes two operands:

mov dest, src

dest and *src* must have the same size and both should not be memory operands

Examples:

```
mov eax, 3    ; store 3 into EAX register (3 is immediate operand)
mov bx, ax    ; copy the value of AX into the BX register
```

ADD Instruction

The ADD instruction is used to add integers.

```
add eax, 4    ; eax = eax + 4
add al, ah    ; al = al + ah
```

6. The Assembler

Assembly code we study is compiled on Microsoft Macro Assembler (MASM). MASM is mainly a command line assembler which means a programmer writes assembly code in a text editor then uses command prompt to compile and link her/his code using MASM. Now, MASM 8 is integrated with Visual C++ 2005, so we can use VC++ 2005 to write, build and debug assembly programs. However, we can use this assembler solely to assemble our programs using command line too.

The following two sections explain compilation procedures and assume:

- 1- Visual Studio 2005 is installed in **C:\Program Files\Microsoft Visual Studio 8** directory and Visual C++ is included in its installation.
- 2- Files of Irvine library are placed in **C:\Irvine** directory

7. How to assemble/debug using command line

To assemble your assembly code using MASM command line:

- 1- Go to C:\Irvine folder and copy **asm32.bat** batch file to your windows directory (i.e. C:\windows).
- 2- Make a directory for storing your assembly code files and name it, let say **"MyAsm"**.
- 3- Write your assembly code in any text editor like *Notepad* then save it in "MyAsm" directory as, say **"main.asm"**. (don't forgot the file extension .asm).
- 4- Open command prompt (from Start menu > Run, type **"cmd"** and press enter)
- 5- Go to **MyAsm** directory in the command prompt by typing **"cd C:\MyAsm"**
- 6- Type **"asm32 main"** and press enter. (don't type the file extension .asm)
- 7- If source file is assembled ok, a message appears saying *"The executable file main.exe was produced"*. Otherwise, error messages are displayed. Fix error(s) in your code and repeat the previous step again.

- 8- To test the program created type: `main.exe` in command prompt and press enter.

Example 1: the first program for assembling demo

```
INCLUDE Irvine32.inc
.data
.code
main:
    mov eax,10000h          ; EAX = 10000h
    add eax,40000h          ; EAX = 50000h
    call DumpRegs

    exit                    ;terminate the execution of the program
END main                   ;end the assembler work, and make the main as the program
                           ;entry point
```

`DumpRegs` is a function defined in `Irvine32` library and it shows the registers values and `call` instruction calls other procedures or functions.

To debug your program in command line:

- 1- Instead of using “asm32 main” to assemble your code, use “**asm32 /D main**”.
/D option allows Visual Studio 2005 to open your exe and to be ready for debugging.
- 2- To begin debugging, step into (or press F11) in your exe and your code will be opened in VS 2005 in debug mode.

8. How to assemble/debug using Visual Studio 2005

To assemble your code using Visual C++ 2005:

Go to C:\Irvine folder and copy **Project_Template** folder to your directory. **Project_Template** folder contains a C++ project that is configured to use Irvine library directly without any additional configuration.

Any time we want to make a new assembly project, we get a copy from `Project_Template` and edit our code in the new copy.

To debug your program in Visual C++ 2005:

Simply, step into (or press F11) in your code much like C# code debugging.

9. Using Visual Studio 2005 Debugger

In this section, we review VS 2005 Debugger capabilities. VS 2005 Debugger allows you to:

- Step through your program, viewing the source code
- Set breakpoints in your code
- View CPU registers and flags
- Watch the values of program variables
- View the runtime stack
- Display blocks of memory

Start the Debugger

Press the **F10** key to start the debugger. Your environment should appear as in the figure below, except that your window configuration may be different.

Watch Window

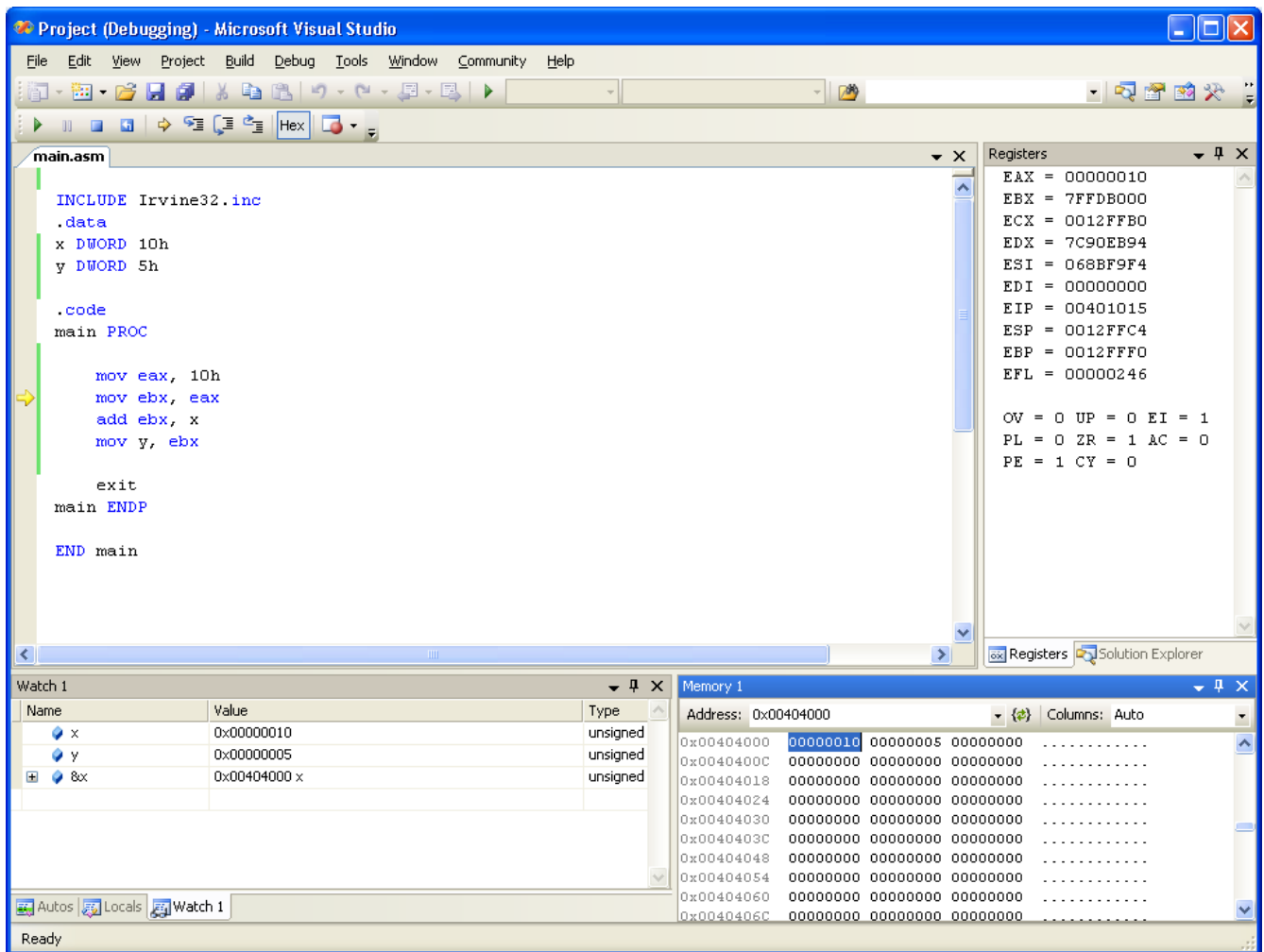
Select **Windows** from the Debug menu, and select **Watch 1**. A Watch window is like an electronic spreadsheet that displays the names and values of selected variables. As you step through a program, you can see variables in this window change value. Initially, the window is empty, but you can drag any program variable into the window with your mouse or even by typing its name in the watch window directly.

Drag (or Type) the **x**, **y**, and **&x** variables into the Watch window and note their current values. The values are initially displayed in decimal, so select hexadecimal format by right-clicking on the watch window and selecting **Hexadecimal Display** from the popup menu, as shown in the figure. Note that the **&x** refers to the address of the **x** variable.

Memory Window

Select **Windows** from the **Debug** menu, and select **Memory**. The Memory window displays a raw dump of memory in either hexadecimal or decimal. It is particularly useful when working with array variables. Since we don't have any arrays in the program, let's display the value of **x**. Next to the Address label, type: **&x (or 0x00404000)**

The Memory window displays a series of individual memory bytes, beginning at the address of **x**. Right-click on the window, and select **4-byte Integer** to display memory in double word chunks, as shown in the figure below. Along the left side of the memory window is shown the address of the first value in each line. Also, at the right side, it displays the text interpretation of the memory content in ANSI code.



Register Window

Select **Windows** from the Debug menu, and select **Register**. The Register window displays the contents of the CPU registers. The flag values are not shown by default, but you can add them in by right-clicking and selecting **Flags**.

Step Into (F11)/ Step Over (F10)

Another way to step through a program is to use the **Trace** (F11) command. It steps down into procedure calls. In contrast, the F10 key just executes procedure calls without tracing into the procedure code.