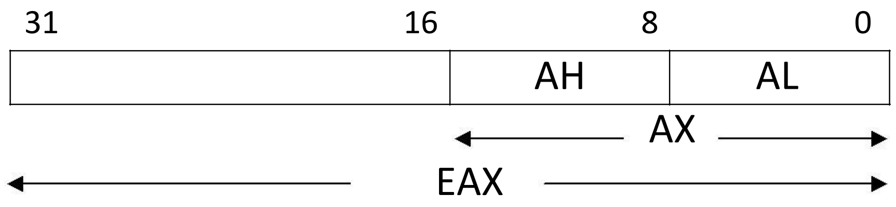
Assembly Language Lab 3

1. (Quiz) 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…).

memory_model, [language_type] **.MODEL**

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.

<u>The following two sections explain compilation procedures and assume:</u>

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

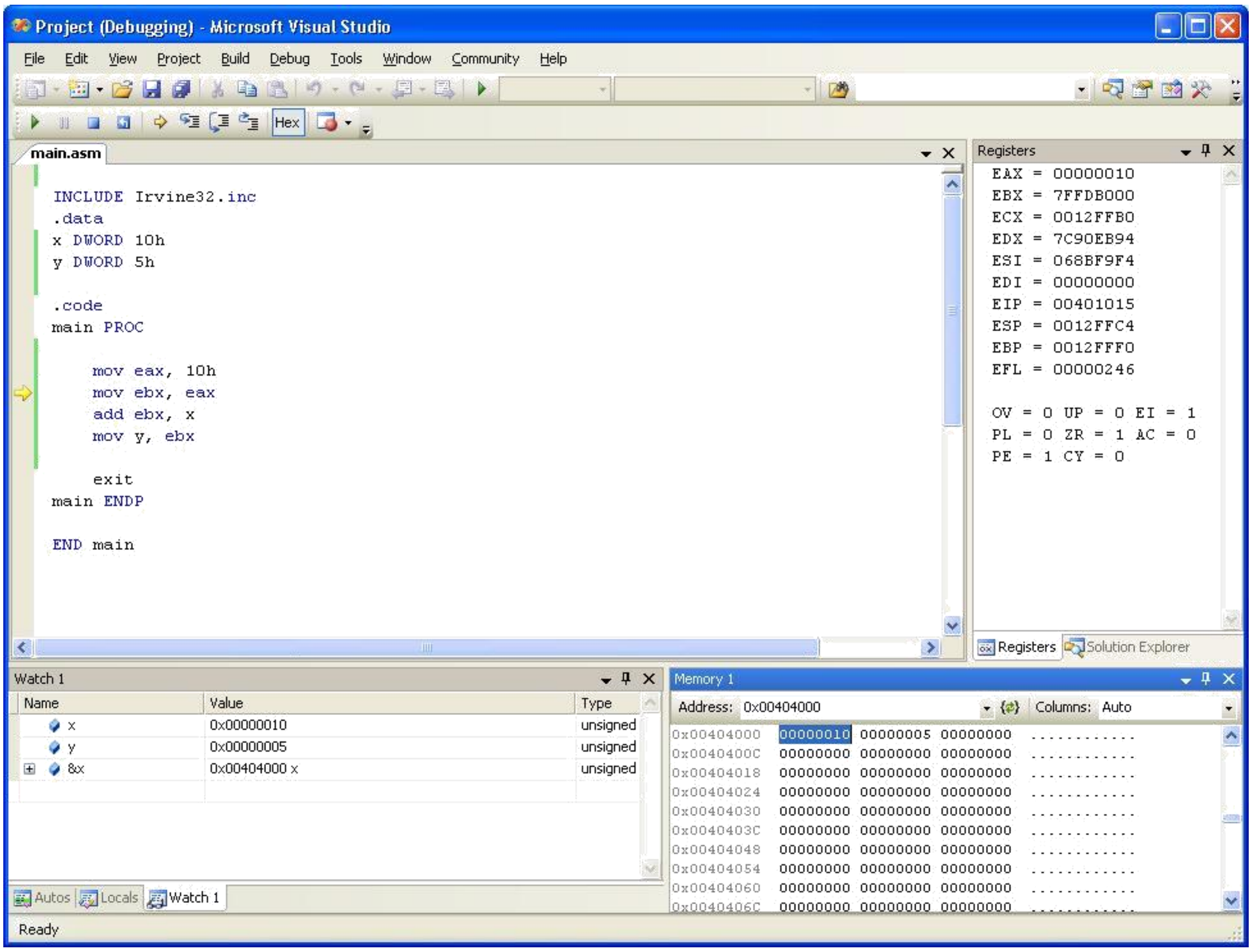
Select **Windows** from the Debug menu, and select **Watch 1**. A Watch window is like an electronic spreadsheet that displays the names and values of selected variables. As you step through a program, you can see variables in this window change value. Initially, the window is empty, but you can drag 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.


Project (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Community Help
Hex
main.asm
Registers
Solution Explorer
Watch 1
Memory 1
Autos Locals Watch 1
Ready

```
INCLUDE Irvine32.inc
.data
x DWORD 10h
y DWORD 5h

.code
main PROC

    mov eax, 10h
    mov ebx, eax
    add ebx, x
    mov y, ebx

    exit
main ENDP

END main
```

```
EAX = 00000010
EBX = 7FFDB000
ECX = 0012FFB0
EDX = 7C90EB94
ESI = 068BF9F4
EDI = 00000000
EIP = 00401015
ESP = 0012FFC4
EBP = 0012FFF0
EFL = 00000246

OV = 0 UP = 0 EI = 1
PL = 0 ZR = 1 AC = 0
PE = 1 CY = 0
```

Name	Value	Type
x	0x00000010	unsigned
y	0x00000005	unsigned
&x	0x00404000 x	unsigned

```
Address: 0x00404000                     Columns: Auto
0x00404000  00000010 00000005 00000000  ............
0x0040400C  00000000 00000000 00000000  ............
0x00404018  00000000 00000000 00000000  ............
0x00404024  00000000 00000000 00000000  ............
0x00404030  00000000 00000000 00000000  ............
0x0040403C  00000000 00000000 00000000  ............
0x00404048  00000000 00000000 00000000  ............
0x00404054  00000000 00000000 00000000  ............
0x00404060  00000000 00000000 00000000  ............
0x0040406C  00000000 00000000 00000000  ............
```

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.

10. Defining Data Items

The assembler provides a set of directives that permit allocating data items with various sizes. The general form of declaring variables is:

```
[name] Type initialization
```

Name: the variable name.
Type: is one of the data types that determine the number of bytes allocated for that variable. They can be one of the following:

Unsigned	Signed	Size (in bytes)
BYTE	SBYTE	1
WORD	SWORD	2
DWORD	SDWORD	4
QWORD		8
TBYTE		10

Initializer: it is the initial value that will be stored in this variable upon allocation. It should fit into the variable size. Also, it can be a single value or a comma-separated list of values and in this case the variable will refer to an array of these values. If there is no specific initial value for that variable, you just put a question mark '?'. Possible values (or expressions) that can be given in the initializer part are:

Initializer	Example	Remarks
Integer constant	2, 33d, 1CEh, 1010100b, -5	**d**: decimal, **h**: hexadecimal, **b**: binary Decimal is the default negative numbers are stored in 2's comp. form
String constant	'A', "F", 'Hello', "Hello"	It is often used with BYTE data type
Integer expression	3+4, 6*5-3	The result of the integer expression should fit the size of the variable

You can also define arrays by using this form:

```
[name] Type repeat-count Dup(initialization )
```

Repeat-Count specifies the number of items in the array and initializer is the initial value for each item. If the initializer is a single value then every item will be initialized by this value. However, if it is a list, then each item in the array will take the corresponding initial value from the list.

Examples:

```
ByteVal0   BYTE  ?                ;Un-initialized byte
ByteVal1   BYTE  48               ;Decimal value
ByteVal2   BYTE  30H              ;Hexdecimal value
ByteVal3   BYTE  01010110B        ;Binary value
ByteVal4   BYTE  1, 2, 3, 4       ;Define 4-items array of size byte
ByteVal5   BYTE  'Hello'          ;character string
ByteVal6   BYTE  'H','e','l','l','o'
                             ;list of characters and it's equivalent to
                             ;the previous character string 'Hello'
ByteVal7   BYTE  5 Dup(3)         ;Define 5-items array each item
                                  ;initialized by value = 3
ByteVal8   BYTE  3 Dup(1,2,3)     ;3-items array and initialized as 1, 2, 3
WordVal0   WORD  ?                ;uninitialized word
WordVal1   WORD  0FF30H           ;Hexdecimal value
WordVal2   WORD  65, 310          ;Define 2-words array
WordVal3   WORD  10 Dup(43)       ;Define 10-words array each item
                                  ;initialized by value = 43
DWordVal0  DWORD  ?               ;uninitialized double word
DWordVal1  DWORD 6F34A030H        ;Hexdecimal value
DWordVal2  DWORD 69065, 350, 65    ;Define 3 double words array
DWordVal3  DWORD 5 Dup(?)         ;Define 5 double Words array without
                                  ;initializing its items
```

Notes:

2. Intel processors store the value of multi-byte variable (e.g. WORD, DWORD…etc) in ***little Indian*** order. This means that the least significant byte is stored in the lowest address byte while the most significant byte is stored in the highest address byte. For example,

   ```
   Val1       DWORD      12345678h
   ```

   If the offset address of Val1 is 100, for example, then, the following will be the placement of values in that variable:

Offset	**Value**
100:	78h
101:	56h
102:	34h
103:	12h

2. When specifying a hexadecimal value and begins with a letter (A-F), you have to put a zero in the begging of this value. If you do not put this zero, the assembler will be confused if this is a hexadecimal value or normal label. For example,
   ```
   Hexval: WORD E123h ;Error: assembler thinks E123h is a label
   Hexval: WORD 0E123h ;correct: as labels never starts with number
   ```

Symbolic Constants

The equ directive can be used to define symbols. Symbols are *named constants* that can be used in the assembly program. The format of defining symbolic constants is:

```
symbol equ value
```

Symbol is considered as a substitution for its value and it cannot be modified by any instruction within the program or redefined by another directive and so it is different from memory variables.

Example:

```
PI_VALUE equ 3.14
```

11. Addressing Modes

As we know, most assembly instructions have operands. The instruction operands can reside whether in a register or in memory. There are a number of methods that allow accessing operands. These methods are called *addressing*. Common addressing modes in Intel assembly language are:

1. **Register Addressing**: using register name like MOV AX, DX
2. **Immediate Addressing (*source operand only*)** by specifying the operand value in the instruction itself like MOV EAX, 23
3. **Direct memory Addressing:** using the variable name like MOV AX, mem_var
4. **Direct Offset Addressing** like MOV CL, byteArr[2], MOV CL, [byteArr+2] or MOV CL, byteArr+2. This example copies the 2nd byte after byteArr to CL. This is much like accessing items in an array but here the number used expresses on a number of bytes not an index of items. For example, to get the 2nd item in a DWORD array, we use dwordArr[4] (not dwordArr[1]). That is because the 1st item is dwordArr[0], the 2nd item is dwordArr[4], 3rd item is dwordArr[8] and so on.
5. **Indirect memory Addressing** uses register value as the *operand address (not the operand value)*. Base registers (EBX, EBP), index registers (ESI, EDI) and other general registers (EAX, ECX, EDX) can be used to reference to memory items by their addresses. Register value is considered to be the offset address

within a segment. All registers are associated with *data segment* except EBP associated with *stack segment*. This mode has variations such as:

- MOV BX, [ESI]
- **Base Displacement Addressing** like MOV ECX, [EBX+3]
- **Base-Index Addressing** like MOV DX, [EBX+ESI]
- **Base-Index with Displacement Addressing** like MOV AX, [EBX+ESI+4]

Example 1: Addressing Modes

Debug the following program and notice changes in the registers and memory variables.

```
   INCLUDE Irvine32.inc
   .data
   Xval     BYTE  50h
   Yval     BYTE  ?

   Zval     WORD  10h, 20h, 30h, 40h

   .code
   main:

   MOV      AX, 1034H        ;Immediate addressing mode (Ax = 1034H)

   MOV      DX, AX           ;register  addressing mode (DX = 1034H)

   MOV      CL, Xval         ;Direct-memory addressing mode (CL = 50H)
;Note that number in brackets or that added to zval expresses bytes.
;Since zval is word -sized, to get the third item, we have to multiply by
;2. If it is double word-sized we should multiply by 4.
   MOV      CX, Zval[4]      ;direct offset addressing mode (CX = 30h)
   MOV      CX, [Zval+4]     ;direct offset also (exactly like Zval[4])

   MOV      CX, Zval+4       ;direct offset also (exactly like Zval[4])

;OFFSET reserved word retrieves the address of the next variable.
   MOV      EBX, offset Xval       ;put the offset address of Xval in EBX
   MOV      ESI, offset Zval       ;put the offset address of Zval in ESI

   MOV      EDI, offset Yval       ;put the offset address of Yval in EDI

   MOV      CL, [EBX]        ;indirect memory addressing mode.

                             ;It is equivalent to MOV CL, Xval (CL = 50h)

   MOV      BYTE PTR [EDI], 25h    ;indirect memory addressing mode.
                                   ;BYTE PTR tells the memory location
                                   ;pointed by EDI is of size BYTE. (Yval =

                                   ;25h)

   ;MOV     [EDI], 20    ;ERROR: must specify what the size the [EDI] refers.
                         ;the assembler cannot guess if 20 will be stored in

                         ;byte, word, double word or even more.
```

```
MOV     AX, [ESI+0] ;index with displacement addressing
MOV     BX, [ESI+2] ;These 4 instructions copy the 4 items of Zval
MOV     CX, [ESI+4] ;array to ax, bx, cx, dx registers respectively.
MOV     DX, [ESI+6]

exit
END main
```