LAB 12

Computer Organization & Assembly Language (COAL) | EL 2003

Student Name: _____

Roll No.: _____

Section: _____

Marks Awarded _____

Prepared by: Talha Shahid

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES (NUCES), KARACHI

Lab Session 11 | HIGH LEVEL LANGUAGE INTERFACE

Objectives:

- General Conventions, `.model` Directive.
- Implementing Inline Assembly Code and calling C and C++ Functions.

Section 01 | General Conventions, `.model` Directive

MASM uses the `.model` directive to determine several important characteristics of a program: its memory model type, procedure naming scheme, and parameter passing convention. The last two are particularly important when assembly language is called by programs written in other programming languages. The syntax of the `.model` directive is

```
.model memorymodel [,modeloptions]
```

`MemoryModel` The `memorymodel` field can be one of the models described in the following **Table**. All of the modes, with the exception of `flat`, are used when programming in `16-bit` real-address mode.

	Model	Description
1	`Tiny`	A single segment, containing both code and data. This model is used by programs having a .com extension in their filenames
2	`Small`	One code segment and one data segment. All code and data are near, by default.
3	`Medium`	Multiple code segments and a single data segment.
4	`Compact`	One code segment and multiple data segments.
5	`Large`	Multiple code and data segments.
6	`Huge`	Same as the large model, except that individual data items may be larger than a single segment.
7	`Flat`	Protected mode. Uses 32-bit offsets for code and data. All data and code (including system resources) are in a single 32-bit segment.

Protected Mode programs use the `flat` memory model, in which offsets are 32 bits, and the `code` and `data` can be as large as **4 GByte**. The **Irvine32.inc** file, for example, contains the following `.model` directive:

```
.model flat, STDCALL
```

STDCALL

The `STDCALL` language specifier causes subroutine arguments to be pushed on the stack in **reverse order (last to first)**. Suppose we write the following function call in a high-level language:

```
AddTwo( 5, 6 );
```

The following assembly language code is equivalent:

```
push 6
push 5
call AddTwo
```

Section 02 | Implementing Inline Assembly Code and calling C and C++ Functions

`__asm` Directive in Microsoft Visual C++

Inline assembly code is assembly language source code that is inserted directly into high level language programs. Most C and C++ compilers support this feature.

In Visual C++, the `__asm` directive can be placed at the beginning of a single statement, or it can mark the beginning of a block of assembly language statements (called an asm block).

Syntax:

```
__asm statement
__asm{
          statement-1
          statement-2
          ...
          statement-n

     }
```

Steps to Implement Inline Assembly Code inside C++ Code

- Create a Win32 Console program in Visual C++. Create a C++, `.cpp` file and insert a `main` function that calls the function from assembly into C++.
- In the same folder as the C++ file, create an assembly language module with `.asm` extension. It should contain a procedure which is to be called in the C++ code, declared with the C calling convention. (don't create another `.asm` file if one is already created)
- Build and run the project. If you modify the `.asm` file, rebuild the project before running it again.

Once your program has been set up properly, you can add code to the `.asm` file that calls C/ C++ language functions.

Example: Assembly Code

```
.model flat, C ;Use the flat memory model. Use C calling conventions
.code
	clear PROC
		xor eax, eax
		xor ebx, ebx
	ret
	clear ENDP
END
```

Example: C++ code

```
#include "iostream"
using namespace std;
extern "C" void clear();
int main()
{
	clear();
	unsigned long int a, b, c;
	cout << "Enter first number: "; cin >> a;
	cout << "Enter secoond number: "; cin >> b;
	__asm
	{
		mov eax, a
		mov ebx, b
		add eax, ebx
		mov c, eax
	}
	cout << "The addition is: " << c;
	return 0;
}void displayMultiplication(int value, int number, int counter) {
	cout << number << " x " << counter << " = " << value << endl;
```

```
}
```

Using Irvine32 Library along with C++ Code

Example: Assembly Code

```
Include Irvine32.inc

askForInteger PROTO C
displayMultiplication PROTO C, value:DWORD, number:DWORD, counter:DWORD
tableLength = 10
.data
	intVal DWORD ?
.code

DisplayTable PROC C
	INVOKE askForInteger
	mov intVal, eax
	mov ecx, tableLength
	mov ebx, 1
	L1:
		push ecx
		imul eax, ebx
		INVOKE displayMultiplication, eax, intVal, ebx
		mov eax, intVal
		inc ebx
		pop ecx
		loop L1
		ret
DisplayTable ENDP
END
```

Example: C++ code

```
#include "iostream"
using namespace std;
// extern "C" instruct the compiler to use C calling conventions
extern "C" {
	void DisplayTable();

	int askForInteger();
	void displayMultiplication(int value, int number, int counter);
}
int main()
{
	DisplayTable();
	return 0;
}
int askForInteger() {
```

```
	int n;
	cout << "Enter a positive integer: ";
	cin >> n;
	return n;
}
void displayMultiplication(int value, int number, int counter) {
	cout << number << " x " << counter << " = " << value << endl;
}
```