CS253 Laboratory session 3

Part 1: Evaluating floating point expressions using the maths co-processor.

A floating point number, N, can be described in scientific notation in the following familiar way

$$N = M.R^E$$

where M is known as the significand (or coefficient or mantissa), M has the range of values

$$-1.0 < M < 1.0$$

E is the exponent and is an integer in the range

$$-\infty < E < \infty$$

R is called the radix and is equal to ten in the case of scientific notation,

$$R = 10$$

Thus a number such as 22/7 can be approximated by the following expression,

$$\frac{22}{7} = 3.14285 \times 10^0$$

The IEEE Short Real Format for floating point numbers is similar. These are called "floats" in most languages and can be used inside the maths co-processor such as the 8087. However there are some differences to consider when compared to scientific notation.

The following line of code found in the DATA segment of the listing of some machine code shows how four bytes are used to store a floating point number,

```
Address     Machine Code      Label Assembly Language (define data)
0000        40A00000          SX    dd    5.0
```

you can see from the above that the number 5.0 is stored as four bytes, starting at memory location 0000 in the data segment, this memory location is labelled SX. The four bytes contain 4x8=32 bits of information organised as follows,

```
40H = 0100,0000b
A0H = 1010,0000b
00H = 0000,0000b
00H = 0000,0000b
```

The number is coded in the 32 bits as follows,

```
0100,0000 - 1010,0000 - 0000,0000 - 0000,0000
```

the first bit contains 0 so the number is positive (1 means negative).

0100,0000 – 1010,0000 - 0000,0000 - 0000,0000

the next eight bits contain information about the exponent coded as follows,

1000,0001b = 129 decimal

The exponent is offset by -127 so that both negative and positive exponents can be coded.

So the exponent is $E = 129 - 127 = 2$ or you could say multiply significand by $2^2 = 4$

Finally the final 23 bits store the significand. The first binary digit for all numbers (except 0) is one so this bit is not stored and needs to be added back before working out the signifcand. Zero is a special case where all 32 bits are set to zero.

0100,0000 – 1010,0000 - 0000,0000 - 0000,0000

So add back the first bit

1010,0000 - 0000,0000 - 0000,0000

these bits are coded as $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \ldots$

or in decimal notation 0.5, 0.25, 0.125, 0.0625, 0.03125,…

So the signifcand is 1.010b or 1.25d

So putting it all together in decimal $1.25 \times 4 = 5$ or binary $1.01 \times 2^2 = 101 = 5$

In summary $N = S.M.R^E$

where		
	$(S = -1) or (S = +1)$	The sign, 1=-ve or 0=+ve, 1st bit.
	$0 < M < 1$	Signifcand (last 23 bits, add back bit=1 at start)
	$R = 2$	radix
	$-127 \le E \le 128$	exponent (bits 1 to 8)

Rebuild the MASM floating point code shown on the next page that evaluates the length of the hypotenuse given the length of the sides of the triangle are 3 and 4. Confirm the result by inspecting the 32 bit binary result is the same as that calculated above. You could use DOSBox to do this, but we have a VPL assignment that you can paste the code into and run it. Don't evaluate the code yet, VPL is expecting a different equation. The code is also available in for download as float.asm.

```
;Program to evaluate SQRT(SX^2+SY^2)
;By: Charles Markham
;Date: 22nd February 2016

;Note SQRT(25)=5 (Calculator expected value)
;Result: 0100,0000 - 1010,0000 - 0000,0000 - 0000,0000
;First bit 0 => Positive
;Bits 1-9    => =129, Subtract 127 => x2 to the power of 2
;Bits 10..   => [1],010,0000 = 1.01=1.01
;1.01*4=5.0 QED

        .model medium
        .8087                   ; Tell MASM co-processor is present
        .STACK 100h
        .DATA

        SX  dd  4.0             ; short real 4 bytes SX = 4.0
        SY  dd  5.0             ;
        HY  dd  0.0             ;
        cntrl   dw      03FFh   ; Control word for 8087
        stat    dw      0       ; Status after calculation

        .CODE                           ; Start of Program
        .STARTUP
        FINIT                           ; Set FPU to default state
        FLDCW           cntrl           ; Round even, Mask Interrupts
        FLD             SX              ; Push SX onto FP stack
        FMUL            ST,ST(0)        ; Multiply ST*ST result on ST
        FLD             SY              ; Push SY onto FP stack
        FMUL            ST,ST(0)        ; Multiply ST*ST
        FADD            ST,ST(1)        ; ADD top two numbers on stack
        FSQRT                                   ; Square root number on stack
        FSTSW   stat                    ; Load FPU status into [stat]
        mov             ax,stat         ; Copy [stat] into ax
        and al,0BFh                     ; Check all 6 status bits
        jnz pass                                ; If any bit set then jump
        FSTP HY                         ; Copy result from stack into HY


         mov    bx,OFFSET RESULT        ;bx=[RESULT+2]+256*[RESULT+3]
         inc    bx
         inc    bx
         mov    ax,[bx]
         mov    bx,ax

        mov     cx,16
back:   rol      bx,1                   ; Rotate bx
                jc      set             ; Check MSB first
        mov      dl,'0'                 ; If carry set dl='0'
                jmp     over
set:    mov      dl,'1'                 ; If carry not set dl='1'
over:   mov     ah,02h                  ; Print ASCII of dl
                int     021h
         loop   back                    ; Repeat 16 times

         mov    bx,OFFSET RESULT        ;bx=[RESULT+0]+256*[RESULT+1]
         mov    ax,[bx]
         mov    bx,ax

        mov     cx,16
back1:  rol     bx,1                    ; Rotate bx
        jc      set1                    ; Check MSB first
        mov      dl,'0'                 ; If carry set dl='0'
        jmp     over1
set1:   mov      dl,'1'                 ; If carry not set dl='1'
over1:  mov     ah,02h                  ; Print ASCII of dl
        int     021h
       loop     back1                   ; Repeat 16 times

        .EXIT
END
```

Now modify your code to evaluate the following expression, submit this code with comments at the start of the listing demonstrating the result obtained was what was expected.

$$A^3 + \sqrt{B}$$

where A=3 and B=9. Use run and evaluate to attempt to modify the program supplied to give the correct answer.

Hint: Replace SX by A and SY by B and then consider changing the stack operations to realise the equation requested. Evaluating a cube may require you to FLD A a second time.

Expected result: $A^3 + \sqrt{B} = 3^3 + \sqrt{9} = 27 + 3 = 30$

Expected output: 0,10000011,11100000….. ,132-127=5, 1.111x2^5=11110b=30d

To do: Submit this code using the VPL link on the sandbox page for evaluation

Part 2: So far we have created and compiled a number of assembly language programs using the Microsoft assembler (MASM). Writing a big program using MASM is probably impractical. However, you can embed assembly language in your C\C++ programs. This is something that you could do in practice so as to optimise a piece of code or make use of instructions that are not accessible via the standard libraries (such a MMX, multimedia extension).

This approach has the benefit of allowing you to write the input and output in C/C++ and use the assembly language for high speed calculation.

It should be said that in practice compilers are so good that it is very difficult to write better machine code than they can generate.

Launch Microsoft Visual Studio 2010 and create a new Windows 32 bit Console Application. Call the project *MASM_FP.sln*, other names will not work. On the solution explorer double click on the Sources folder and the open the file MASM_FP.cpp. Replace the boiler plate code provided by Microsoft with the code shown on the next page.

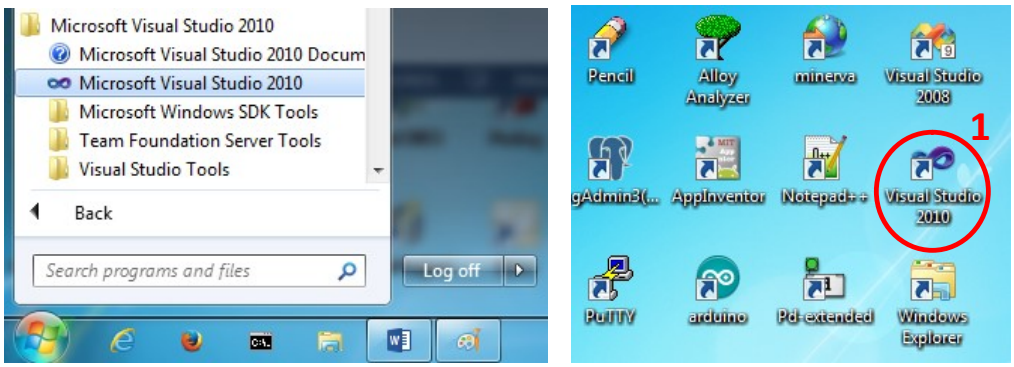


Figure 1: Launch VS2010

Create a New Project of type C++ (not c#) Windows 32 Console application called MASM_FP (short for Microsoft Assembler Floating Point). You could call it what you like.

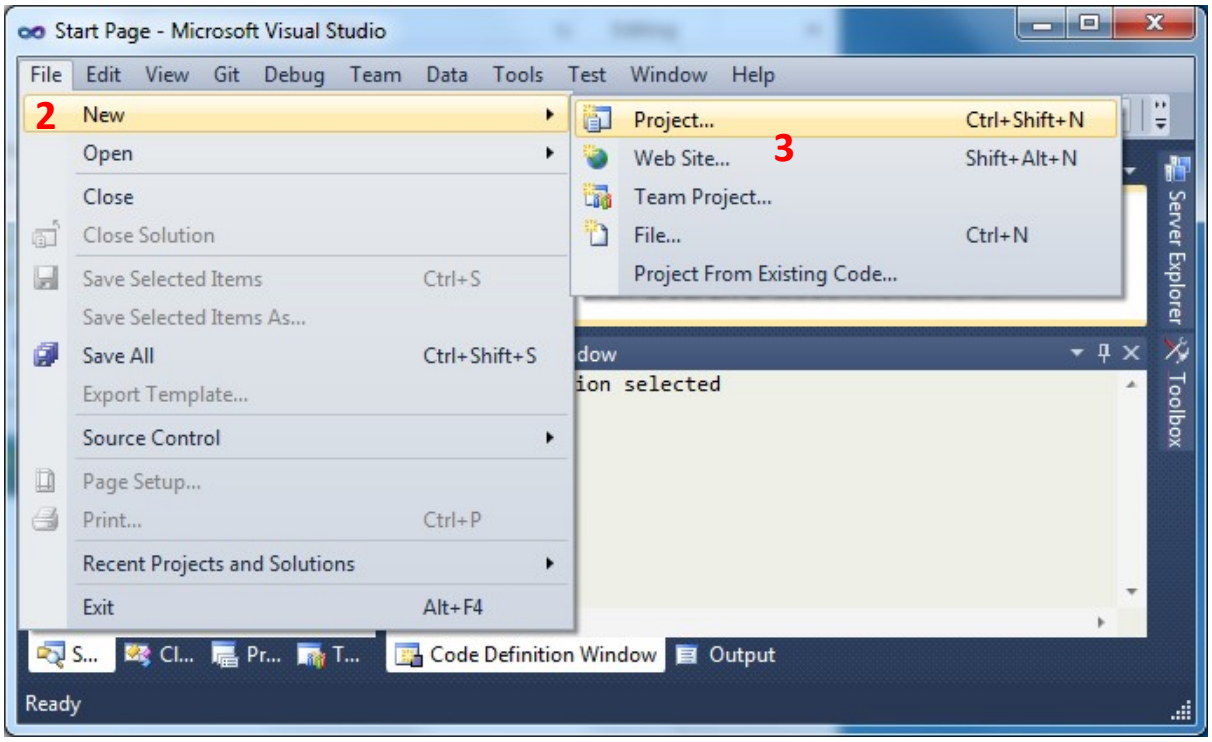


Figure 2: Create a new Windows 32 console application in C++.

The project could be saved locally in c:\temp or in x:\CS253 on your own disk drive

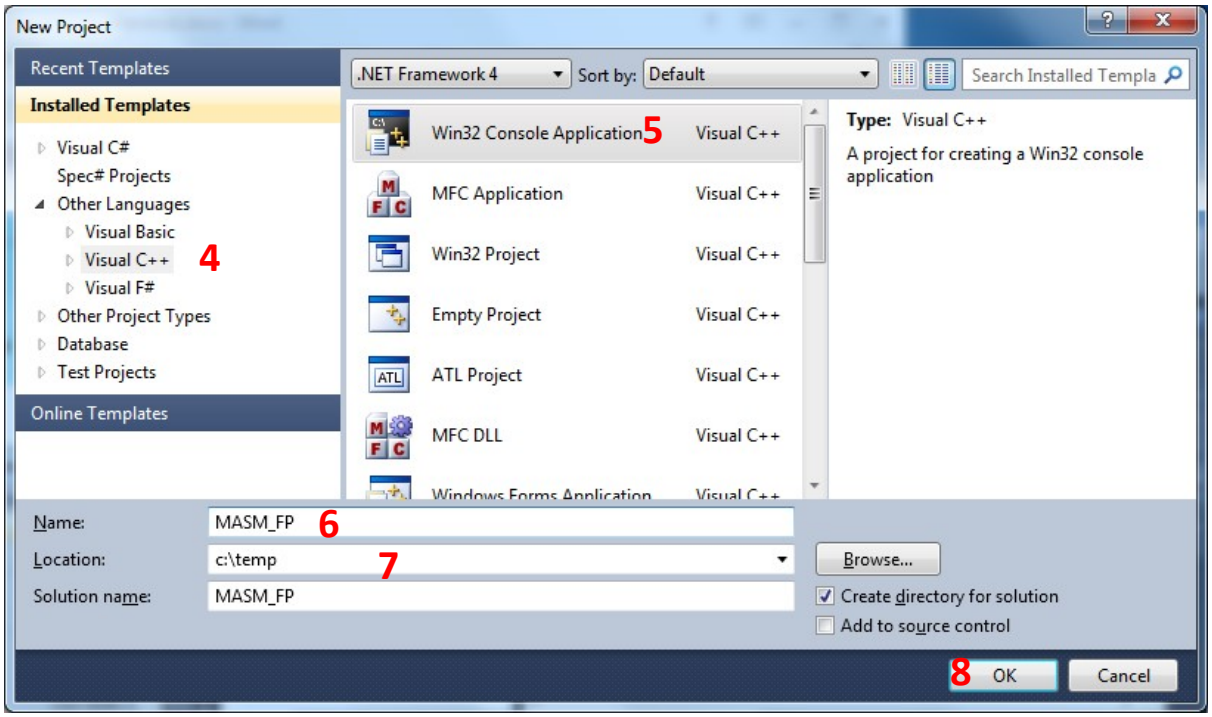


Figure 3: Create a new Windows 32 console application in C++.

Cut and paste the following MASM_FP code from the next page of this handout and use it to replace all the exisitng code contained in MASM_FP.cpp. You may need to click on the Solution Explorer in Visual Studio to find this file. If the IDE gets in a muddle then just choose Window – Reset Windows Layout from the tool bar. Set the configuration manager from "Debug" to "Release" and then compile and run the code by clicking on the green triangle "play" button.

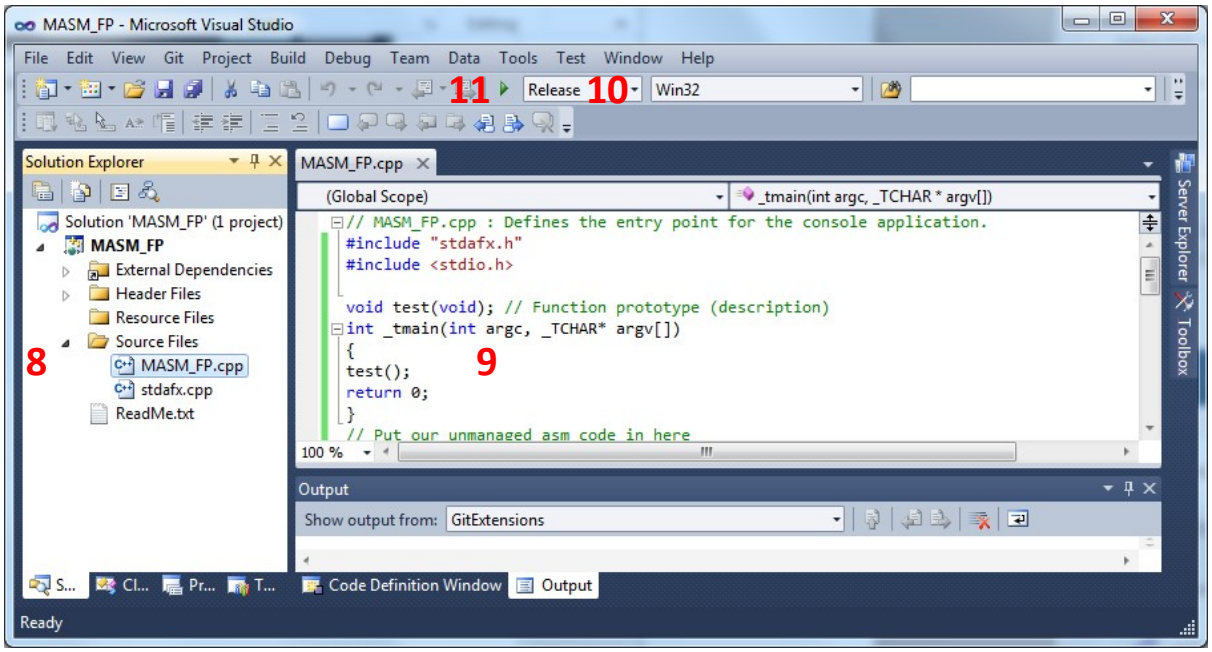


Figure 3: Entering the code, compile and run within the Visual Studio IDE.

All being well you should produce a run display that allows you to enter two numbers (short integers) and display the sum of the two. This makes use of in-line assembly language to do the addition.

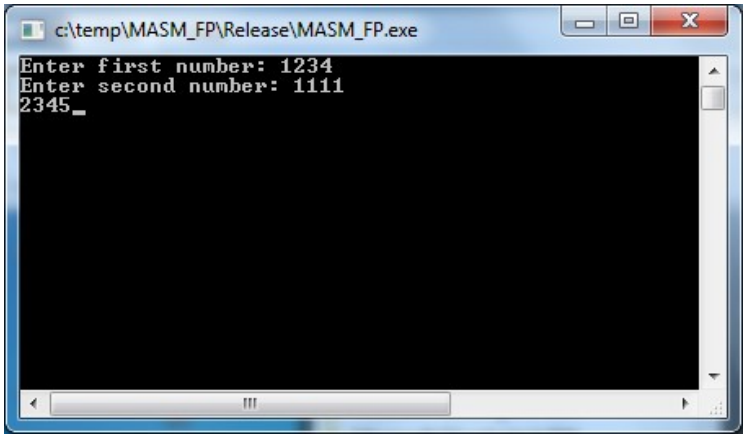


Figure 4: Addition using Assembly Language, Input and Output using C++.

Program 1

```
// MASM_FP.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <stdio.h>

void test(void); // Function prototype (description)
int _tmain(int argc, _TCHAR* argv[])
{
	test();
	return 0;
}

// Put our unmanaged asm code in here
void test()
{
	unsigned short num1;
	unsigned short num2;
	unsigned short result;

	printf("Enter first number: ");
	scanf(" %hd",&num1);
	printf("Enter second number: ");
	scanf(" %hd",&num2);

	__asm
	{
		mov ax, num1 ; Direct addressing to access num1
		mov bx, num2 ; put value in num2 into bx
		add ax, bx ; ax=ax+bx
		mov result,ax; ; put value into result
	}

	printf("%hd",result); // Display result

	// Wait for enter to be pressed before terminating
	while(getchar()!=10); // Clear buffer of previous <ret>
	while(getchar()!=10); // Wait for a new <ret>
}
```

Part 3: Use the sample code on the next page to generate a suitable frame work for creating code to implement a floating point calculation using Assembly Language from within a C++ program. The program puts the contents of variables on the floating stack and then takes the value on the floating point stack and puts it back into the variable C.

When you run the code you should see that the value (2) has been passed in and then out of the floating units stack.

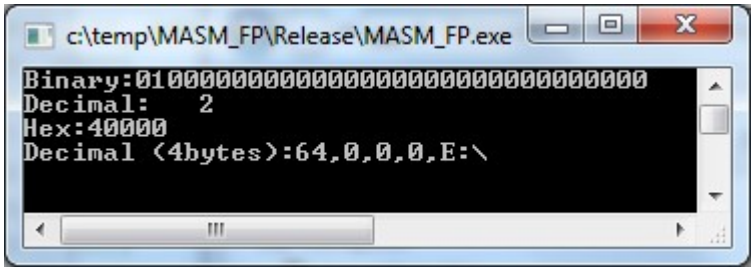


Figure 5: Sample code pushes to on FP stack, then pops the stack to return value of 2.

Modify this code so that it has the same functionality as part 1 of the laboratory handout. The code should evaluate the expression,

$$C = A^3 + \sqrt{B}$$

where A=3 and B=9.

Program 2

```
// MASM_FP.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <stdio.h>

void test(void); // Function prototype (description)
int _tmain(int argc, _TCHAR* argv[])
{
test();
return 0;
}

void test()
{
 float A=2,C=0;
 unsigned short cntrl=0x3FF,stat;
 __asm
 {
  FINIT
  FLDCW cntrl ; Round even, Mask Interrupts
  FLD A         ; Push SX onto FP stack

  FSTSW stat    ; Load FPU status into [stat]
  FSTP C             ; Copy result from stack into HY
 }

// Binary representation of the 4 bytes, (32 bits) coding HY
printf("Binary:");
unsigned char byt;
for(int x=3;x>=0;x--)
{
 byt=*((unsigned char *)&C+x);
 for(int y=128;y>0;y/=2)
 {
  if ((y&byt)==0) printf("0"); else printf("1");
 }
}

// Decimal format
printf("\nDecimal: %3.0f",C);

// Hex format
printf("\nHex:");
for(int x=3;x>=0;x--)
{
 byt=*((unsigned char *)&C+x);

 printf("%x",(unsigned int)byt);
}

// Decimal 4 byte format
printf("\nDecimal (4bytes):");
for(int x=3;x>=0;x--)
{
 byt=*((unsigned char *)&C+x);
 printf("%d,",(unsigned int)byt);
}
//
while(getchar()!=10);
while(getchar()!=10);
}
```

The VPL used to check your code normally works within the Linux operating system. To evaluate this Windows code, we have to run the executable in a Windows emulator called “Wine” (which has already been placed inside the Linux container used to evaluate the code within the VPL framework). All this has been done for you.

The approach does require your executable to a base64 (text version) of a zip file prior to submission.

The code should evaluate the same expression as before,

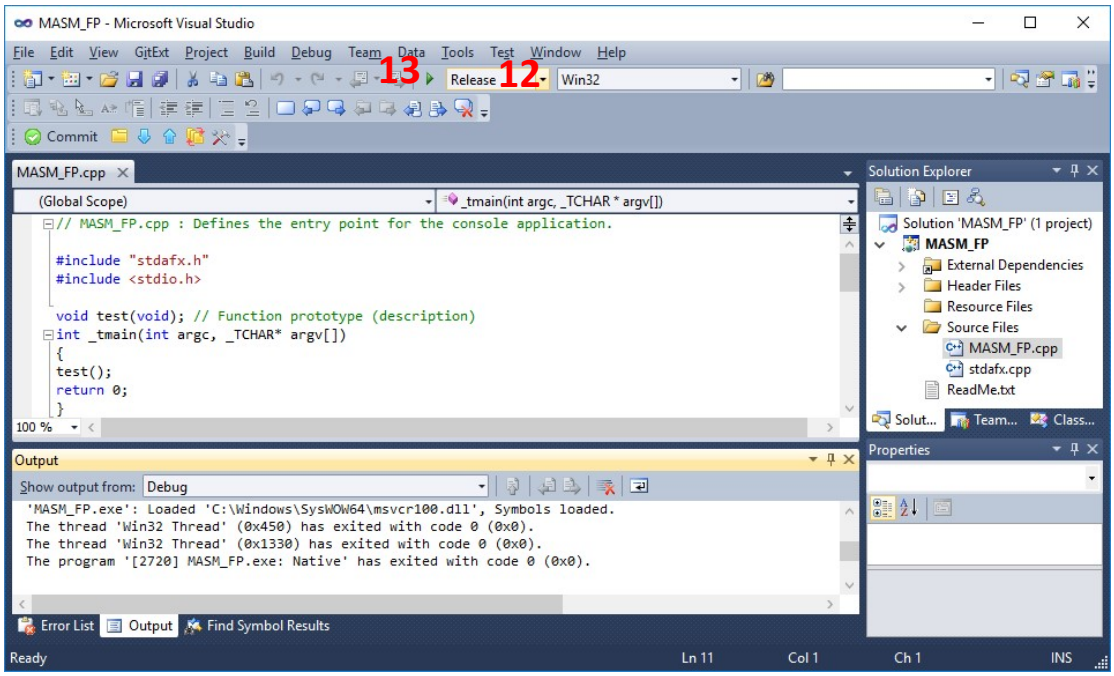
$$C = A^3 + \sqrt{B}$$

where A=3 and B=9.

The programs should only have one output line with the exact format specified, see template below.

```
#include "stdafx.h"
#include <stdio.h>

void test(void); // Function prototype (description)
int _tmain(int argc, _TCHAR* argv[])
{
test();
return 0;
}

void test()
{
 float A=3,B=9,C=0;
 unsigned short cntrl=0x3FF,stat;
 __asm
 {
  FINIT
  FLDCW cntrl ; Round even, Mask Interrupts
  FLD A         ; Push SX onto FP stack

  // Add code to evaluate A^3+SQRT(B) using MASM

  FSTSW stat    ; Load FPU status into [stat]
  FSTP C             ; Copy result from stack into HY
 }

// Decimal format
 printf("%3.3f",C);
}
```

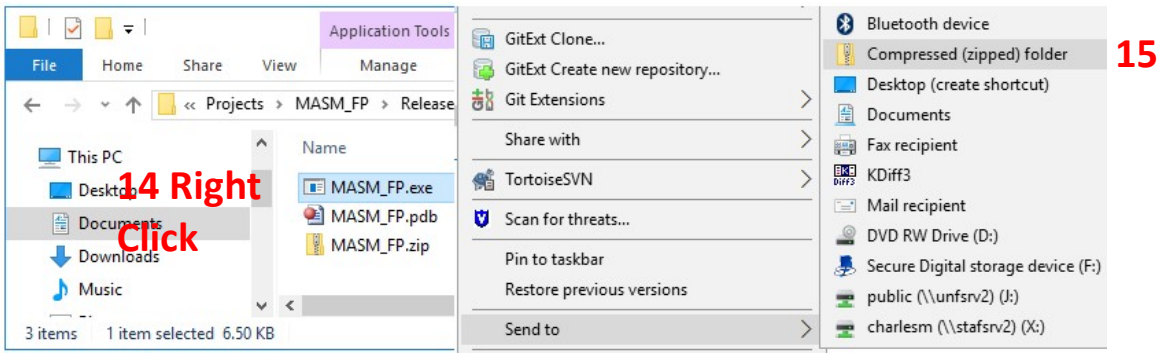
Compile and run this code in Release Mode rather than Debug Mode.



You may wish to add the following lines after the prinf() if you want to check what the program is doing at run time but they must be commented out and the program recompiled before you proceed.

```
printf("%3.3f",C);

 // while(getchar()!=10);
 // while(getchar()!=10);
}
```

Once you are happy that the program is complete. You need to zip the MASM_FP.exe found in the release folder of your project. Right click on the file and send it to zip.

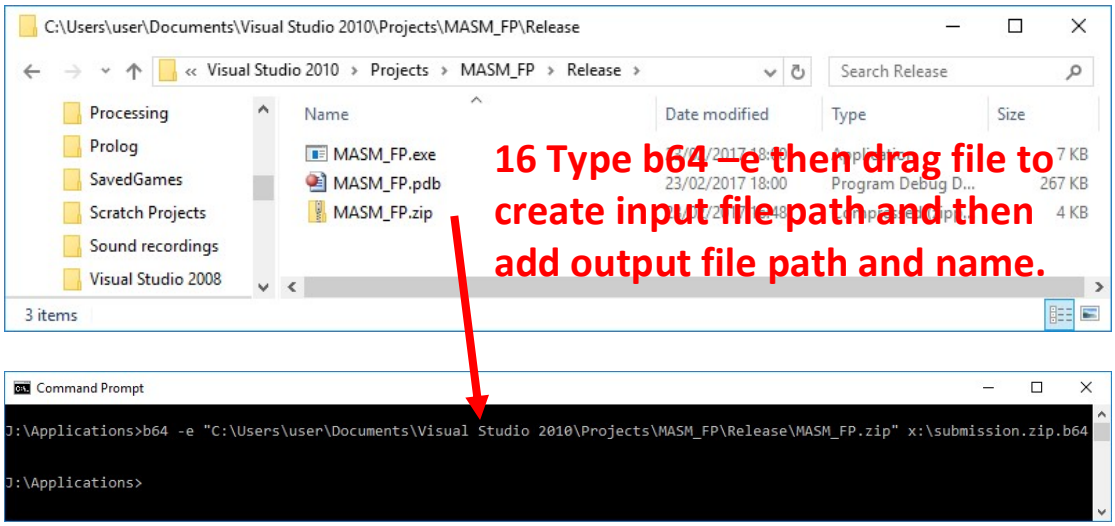


The zip file now needs to be base64 encoded (turned into text).

To do the is click on the command prompt and change the path to be j:\Applications, type J: and press enter then cd Applications and press enter.

Run the b64 command to convert your file to base 64. Type b64 –e on the command line and then drag the file (in Windows) to the command line and it will automatically type the path for you. Add the output file path and name which must be x:\submission.zip.b64 and then press enter.

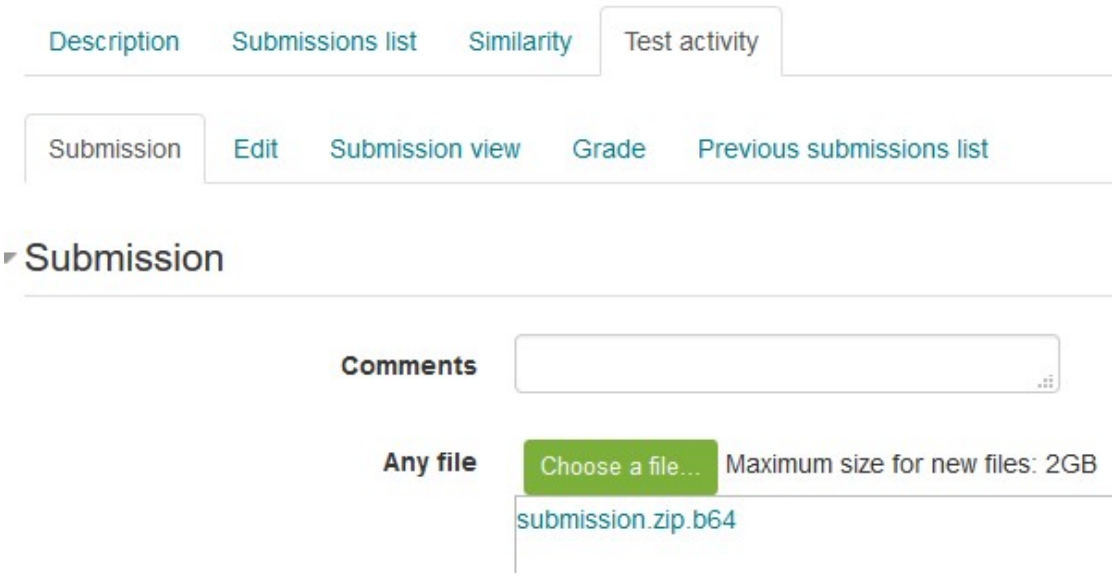
The line should look something like the following,



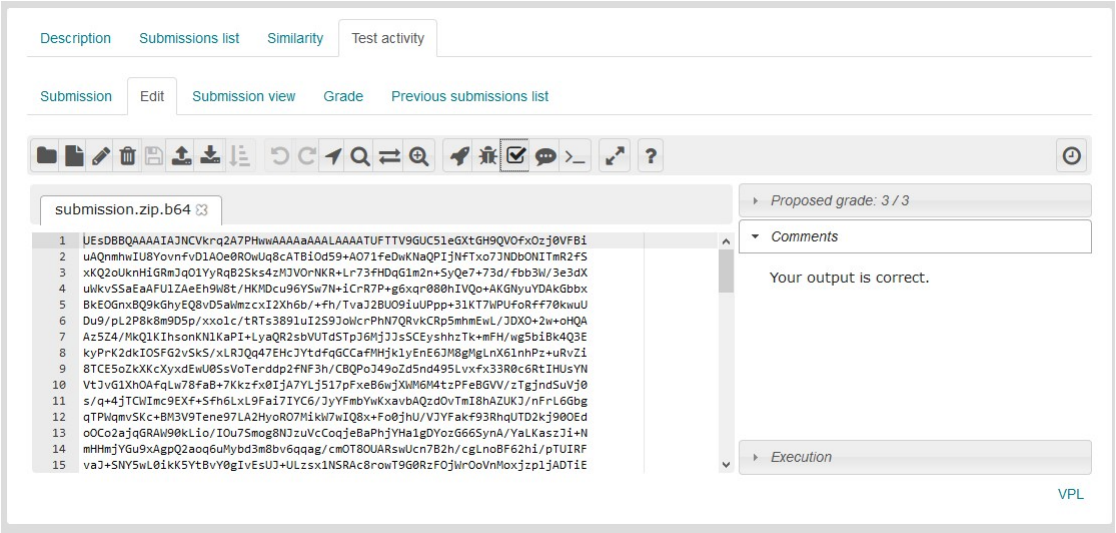
Go to moodle and select the VPL assignment,

Part3: VPL VS2010 C++ Executable upload (evaluation of R=A^3+SQRT(B)) [3 marks]

Select Test Activity and then Select Submission and then drag and drop your file in the submission box (it is in the Windows X:\ folder).



Finally run the evaluation script by going to edit and clicking the tick box.



To do: Ensure that the above process has been graded as above.

Part 4: The aim of the next section is to show how it possible using assembly language to directly access CPU instructions that are not directly available when you use a higher level language.

MMX was a technology added to Pentium processors to allow them to do simultaneous operations on a group of data using a single instruction. This is a form of parallel processing known as SIMD (single instruction multiple data), that can run on a single processor.

Note: Pipelining (see last week's quiz) is also another form of parallel processing that can be done using a single processor computer.

The code copies an array of bytes `NUM1={0,1,2,3,4,5,6,7}` and `mmx_word NUM2={1,1,1,1,1,1,1,1}` into two internal registers of the mmx processor called mm0 and mm1.

Note these are in fact the same as ST(0) and ST(1) used by the floating point processor (so you can't do both MMX and Floating Point at the same time).

When you run the code the instruction, `paddb mm0,mm1` , does all eight additions using a single operation. This is at least eight times faster than doing the same calculation one addition at a time.

Look at the run time code and note the significance of the run time display.

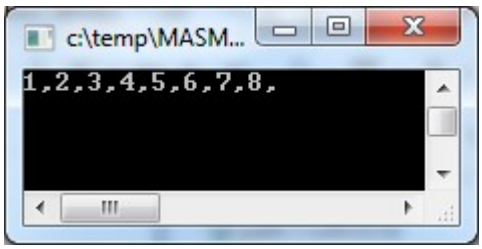


Figure 4: Addition using Assembly Language, Input and Output using C++.

To Do: Complete the General Quiz

Program 5 MMX

```
// MASM_FP.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <stdio.h>

void test(void); // Function prototype (description)
int _tmain(int argc, _TCHAR* argv[])
{
test();
return 0;
}

// Put our unmanged asm code in here
void test()
{

union mmx_word{
              unsigned char byte[8];
              unsigned __int64 value;
            };

mmx_word NUM1={0,1,2,3,4,5,6,7};
mmx_word NUM2={1,1,1,1,1,1,1,1};

 __asm
 {
   movq  mm0,NUM1
   movq  mm1,NUM2
   paddb mm0,mm1  // Add 8 bytes simultaneously
   movq  NUM1,mm0
 }

 for(int i=0;i<8;i++) printf("%d,",(unsigned int)NUM1.byte[i]);

 // Floating point value
 //printf("%I64u",(unsigned __int64)NUM1.byte[0]);


 printf("\n");

 // Wait for enter to be pressed before terminating
 while(getchar()!=10);  // Clear buffer of previous <ret>
 while(getchar()!=10);  // Pait for a new <ret>
}

// other way of declaring NUM1, NUM2, no need to copy...
//NUM1.byte[0] = 0;
//...
```