HOMEWORK 6

Directions: *This assignment is due in Canvas as a PDF by* ***11:59 p.m. on Sunday, November 16****.*
Scoring (5 problems, 30 total points): *16, 2, 4, 1, and 7 points, respectively*

QUESTION 1: ANALYSIS OF THE RUNTIME STACK

Suppose you want to compute the sum of the first n natural numbers ($0 + 1 + 2 + 3 + \cdots + n$). One way to do that is to compute $n(n + 1)/2$. A more awful way to compute it is to use a recursive function, like this:

```
uint32_t AddDownToZero(uint32_t n) {
     if (n == 0) {
          return 0;
     } else {
          return AddDownToZero(n-1) + n;
     }
}
```

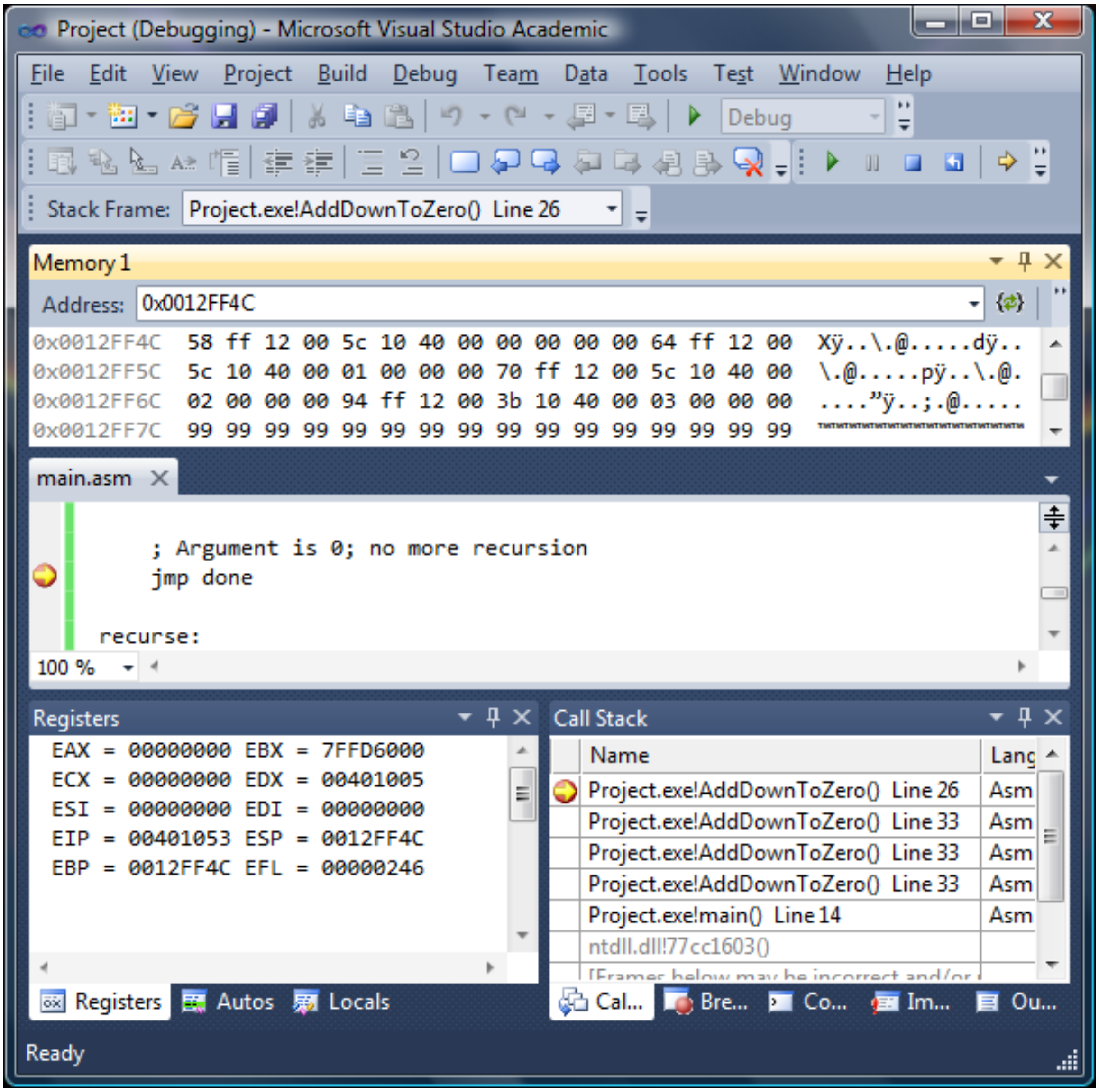
The program below is, essentially, a translation of the recursive procedure above. The main procedure pushes some garbage onto the stack, then invokes AddDownToZero(3) procedure to compute 0 + 1 + 2 + 3 = 6. (This program is available in Canvas as HW6-Stack.asm, if you want to step through it.)

```
INCLUDE Irvine32.inc

.code
main PROC
    push 99999999h           ; (Garbage)
    push 99999999h           ; (Garbage)
    push 99999999h           ; (Garbage)
    push 99999999h           ; (Garbage)

    ; Add the numbers 0 + 1 + 2 + 3 = 6
    push 3                   ; (Argument - Initial Call)
    call AddDownToZero       ; (Return Address - Initial Call)
    call WriteDec
    exit
main ENDP

AddDownToZero PROC
    enter 0,0                ; (Saved EBP)

    ; Load the argument into EAX and check if it's zero
    mov eax, DWORD PTR [ebp+8]
    cmp eax, 0
    jne recurse

    ; Argument is 0; no more recursion
    jmp done

recurse:
    ; Invoke AddDownToZero on the next smallest natural number
    dec eax
    push eax                 ; (Argument - Recursive Call)
    call AddDownToZero       ; (Return Address - Recursive Call)

    ; Add the argument value to the result of the recursive call
    add eax, DWORD PTR [ebp+8]

done:
    leave
    ret 4
AddDownToZero ENDP
END main
```

Now, suppose you do the following:

1. Set a breakpoint on the `jmp done` line. (When the breakpoint is hit, the procedure argument is 0, so no more recursive calls will be made.)
2. Run this program in the debugger.
3. When the breakpoint is hit, open a Memory window on the memory address currently in ESP.

At this point, the Memory, Registers, and Call Stack windows will look something like this.



In the screenshot above, the Memory window shows the 64 bytes of data starting at memory address 0012FF4Ch, which is the value in the ESP register. This corresponds to 64 ÷ 4 = 16 values that have been pushed onto the stack. As the Call Stack window shows, the `main` procedure called `AddDownToZero`, and it made three additional recursive calls.

▸ **List the 16 values on the runtime stack, starting with the entry on the top of the stack. For each entry, list (1) the value, in hexadecimal, and (2) which of the following that entry corresponds to:**

- One of the "garbage" values pushed at the start of main.
- The argument for the initial call.
- The return address for the initial call.
- The saved value of EBP (when the stack frame was created).
- The argument for a recursive call.
- The return address for a recursive call.

Question 2: Analysis of the Runtime Stack

In the previous question, the main procedure invoked AddDownToZero(3). When the breakpoint was hit, 64 bytes had been pushed onto the stack: 16 bytes from the four "garbage" values pushed at the start of main, plus 48 bytes due to the invocations of AddDownToZero.

If the main procedure had invoked AddDownToZero(0) instead, then upon hitting the breakpoint, only 28 bytes would have been pushed onto the stack: 16 bytes of "garbage," plus 12 bytes due to calls.

▸ **Instead of invoking AddDownToZero(3) from main, suppose you change the invocation to AddDownToZero(*i*), for some arbitrary nonnegative integer *i*. How many bytes will be pushed onto the stack when the breakpoint is it? Give your answer as a formula in terms of *i*.**

Question 3: Analysis of Memory Operands/Data Declarations

Consider the following program.

```
INCLUDE Irvine32.inc

.data
start WORD  0BEEFh
      WORD  (What goes here?  Fill in the blank.)
      DWORD (What goes here?  Fill in the blank.)
      BYTE  (What goes here?  Fill in the blank.)
      BYTE  (What goes here?  Fill in the blank.)
STOP = $

.code
main PROC
    lea esi, start
a:  movzx eax, WORD PTR [esi]
    call WriteHex
    call Crlf
    add esi, SIZEOF WORD
    cmp esi, STOP
    jne a
    exit
main ENDP
END main
```

Suppose the output of the above program is:

```
0000BEEF
0000F00D
00003344
00001122
0000B2A1
```

‣ **If the program produces this output, what are the missing data declarations in the code above?**

Question 4: Analysis of Memory Operands/Data Declarations

‣ **Suppose `lea esi, start` puts the value 00405000h into ESI. What is the value of the symbolic constant STOP?**

Question 5: Analysis of Compiler-Generated Code

Consider the following (simple) C++ program.

```
int main() {
        float value = 5.25f;
        value = value + 1.0f;
        return 0;
}
```

Visual C++ compiles the above program into the following assembly language code.

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.40219.01

    TITLE   C:\Users\joverbey\3350-fa14\Code\HW6-Disasm\HW6-Disasm\main.cpp
    .686P
    .XMM
    include listing.inc
    .model  flat

INCLUDELIB OLDNAMES

EXTRN   @__security_check_cookie@4:PROC
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@40a80000
PUBLIC  _main
EXTRN   __fltused:DWORD
;   COMDAT __real@3ff0000000000000
; File c:\users\joverbey\3350-fa14\code\hw6-disasm\hw6-disasm\main.cpp
CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1
CONST   ENDS
;   COMDAT __real@40a80000
CONST   SEGMENT
__real@40a80000 DD 040a80000r       ; 5.25
; Function compile flags: /Odtp
CONST   ENDS
;   COMDAT _main
_TEXT   SEGMENT
_value$ = -4                        ; size = 4
_main   PROC                        ; COMDAT
```

```
; 1    : int main() {

    push    ebp
    mov ebp, esp
    push    ecx

; 2    :   float value = 5.25f;

    fld DWORD PTR __real@40a80000
    fstp    DWORD PTR _value$[ebp]

; 3    :   value = value + 1.0f;

    fld DWORD PTR _value$[ebp]
    fadd    QWORD PTR __real@3ff0000000000000
    fstp    DWORD PTR _value$[ebp]

; 4    :   return 0;

    xor eax, eax

; 5    : }

    mov esp, ebp
    pop ebp
    ret 0
_main   ENDP
_TEXT   ENDS
END
```

You can mostly ignore the code in gray. You should be able to understand the code in black. Note:

- The DQ is an older form of the QWORD directive. It is used to declare 64-byte data.
- The DD directive is an older form of the DWORD directive. It is used to declare 32-byte data.

Answer the following subquestions:

(a) The compiler generates a DWORD declaration called `__real@40a80000` and sets it equal to hexadecimal 40A80000, which represent a single-precision floating point number (presumably 5.25, according to the compiler's comment). Write out the 32 bits. Indicate which bits correspond to the sign bit, biased exponent, and significant (fractional part). Then, plug them into the appropriate formula to show that they do, in fact, represent the number 5.25.

(b) The `main` procedure contains a local variable named `value`. Visual C++ has decided to store this value at [ebp–4].[1] However, in the prologue, Visual C++ did **not** use `sub esp, 4` to reserve four bytes of space to store it! Is this a compiler bug? If not, explain how it is reserving space to store this local variable.

(c) The last statement of the C++ code is `return 0`. How does the generated code guarantee that the value 0 will always be returned?

(d) The statement "value = value + 1.0f" from the C++ code is translated into a sequence of three instructions: fld, fadd, and fstp. Briefly explain this translation; give a convincing argument that this does, in fact, add 1.0 to the value variable. Describe what memory accesses are performed, and what values are pushed onto/popped from the FPU stack to perform this computation. (Note: The generated code uses a one-operand version of the fadd instruction, which we did not discuss in lecture, although it is described in your textbook.)

---

[1] Note that it defined a symbolic constant named `_value$` equal to –4. It uses a slightly different syntax than we did for memory operands: `_value$[ebp]` is equivalent to `[ebp-4]`.