CS231-01 Fall 2020
Homework Assignment #8 (Individual)
Due: Sunday, 12-13-2020, 11:59 PM Anywhere on Earth time (AoE)
Via Moodle

---

1 Assignment #8

1.1 Readings

- Read these sections in *The Art of Assembly*:
  - *Section 6.1, The Processor Status Register (Flags)*.
  - *Section 6.5, Program Control Flow Instructions*. Read the introduction.
  - *Section 6.5.3, The CMP Instruction*.
  - *Section 6.9.4, The Conditional Jump Instructions*. Ignore material about "manually extending the range of a jump" for 80286 and earlier. After the conditional jump instruction alias tables, skip over to the material which begins "The 80386 and later processors provide an extended form of the conditional jump…" and read to end.

- Optionally read these sections in *The Art of Assembly*:
  - *Section 6.9.1, Unconditional Jumps*. You will learn enough about how to use `jmp` in class. This section offers detailed information about how `jmp` instructions are encoded, but depends on knowledge of segment registers and addressing modes. Proper encoding of `jmp` instructions is not a concern for us because it is handled by the NASM assembler.
  - *Section 6.9.2, The CALL and RET Instructions*. You will learn enough about how to use `call` and `ret` in lecture. As with `jmp` this section offers detailed information about how they are encoded, but depends on knowledge of segment registers and addressing modes.
  - *Section 6.9.3, The INT, INTO, BOUND, and IRET Instructions*. This section offers some intuition regarding the `int` instructions that you have been using to make system calls (e.g. to print text to the console or exit the program).

- Optionally read *Dive into Systems* Chapter 5, Sections 5.3-5.10

- Read *Dive into Systems* Chapter 8, Sections 8.1-8.6. Carefully read the section in the introduction to this chapter on "x86 Syntax Branches" and see notes below. Ignore the parts about "conditional move" instructions, and you may skip section 8.5.2 "Tracing through Main".

1.1.1 Notes

The *Dive into Systems* readings for this assignment have "disassembly" figures that show the assembly instructions corresponding to compiled C code. Unfortunately the assembly code syntax that the book uses (AT&T/GNU Assembler) is different from the syntax used by the NASM assembler (Intel). For example, AT&T assembler syntax lists instruction operands in the opposite order from NASM, with the "source" operand first and "destination" second.

Although you may be able to intuit the AT&T assembly syntax, you are not required to learn it for this course. If you would like to see the C code disassembled into NASM-like instructions, you can execute the following commands. First, put your C code into a file, say `dis.c` and compile the file to an object file.

```
gcc -c dis.c
```

The `-c` flag instructs `gcc` not to run the linker to create an executable. This way, your file does not need to have a `main()` function. Then run `objdump` thusly:

```
objdump -M intel -d dis.o
```

The `-d` flag tells `objdump` to disassemble the C code, and the `-M intel` flag has it produce assembly in Intel syntax, which is almost identical to NASM's assembly syntax.

1.2 Problem 1: Character Decoder, Part 2

For this problem, you will write an assembly language program that is based on the program you wrote for Homework 7 Problem 3. Your program should behave as follows:

- It will first prompt the user for an integer, which we'll call N,
- It will then prompt the user to enter N integers,

- It will manipulate each integer the same way as the program you wrote for Homework 7 Problem 3,
- And, finally it will print the collection of N integers out as a string of characters.
- You may assume that N will never be larger than 64.
- You don't need to worry about whether the integers are signed or unsigned. `_getInput` takes care of converting the user input to 32-bit bit patterns

Examples:

```
./hw8_1
> 1
> 2824443240
geek
```

```
./hw8_1
> 4
> 3075841372
> 2894057575
> 2757336163
> 3810316394
Smith College
```

```
./hw8_1
> 1
> 3809860917
:-)
```

1.2.1 Testing

To facilitate developing your program we recommend that you automate testing of it. Instead of typing numbers on the keyboard yourself, you can create a data file with test inputs and redirect `stdin` to use your file as input to your program.

To do this, create a text file (with `emacs` or any other text editor) named, for example, `data.txt` . Make it so that `data.txt` contains the following numbers, and no extra blank lines, no extra spaces, and just one blank line at the end:

```
4
3075841372
2894057575
2757336163
3810316394
```

Run your program as follows:

```
./hw8_1 < data.txt
> > > > > Smith College
```

This way you don't have to type all these numbers yourself. Remember, programming is an exercise in controlled laziness!

1.2.2 Submission

Submit your program using the Moodle link labeled "8.1 Character Decoder, Part 2" as a file named `hw8_1.asm` .

1.3 Problem 2: String Transformer

Write an assembly language program that prompts the user for a string of characters and prints it back to the console after having transformed all lower-case characters into their upper-case equivalents, and after having replaced all the periods with exclamation points.

You may use `_getString` from `231Lib` to read in the string. `_getString` will return the address of the string in `ecx` and the number of characters read in `edx` . As with problems in the previous assignments, the mapping between integers and text characters is dictated by a specification called ASCII (American Standard Code for Information Interchange). You can find an ASCII conversion table on Wikipedia or elsewhere via web search. For this assignment you may treat the ASCII text characters as signed, 2's complement 8-bit integers, and you can safely use the jump instructions from the "Signed Comparisons" table from *Art of Assembly* for comparing them.

1.3.1 Examples

```
cs231a-zz@gremlin ~ $ ./hw8_3
> hello
HELLO
```

```
cs231a-zz@gremlin ~ $ ./hw8_3
> 1234
1234
```

```
cs231a-zz@gremlin ~ $ ./hw8_3
> Mr. Owl ate my metal worm.
MR! OWL ATE MY METAL WORM!
```

```
cs231a-zz@gremlin ~ $ ./hw8_3
> abcdefg hijklmnop QRSTUV 112345667890!@#$%^&*()_+=-.. .    .
ABCDEFG HIJKLMNOP QRSTUV 112345667890!@#$%^&*()_+=-!! !    !
```

1.3.2 Submission

Submit your program using the Moodle link labeled "8.2 String Transformer" as a file named `hw8_2.asm` .

1.4 Problem 3: Functions in Assembly

For this problem you will write an assembly program that contains several functions, but no main program. Your program will be used as a library of functions, and it should not contain a `_start` label. Your functions will be linked with and tested by a main program provided by the autograder. The test program will call your functions with various parameters, making sure that they work well with different types of input.

Your program/library will contain four functions labeled `f1` , `f2` , `f3` and `f4` , which should behave as follows:

- `f1( s_addr, length )` . Receives the address of a string `s` as well as its length, both passed as 32-bit parameters on the stack. `f1` converts the string to uppercase, replacing all the alphabetic characters in the string with their capital letter equivalents. `f1` should not print the converted string to the console, instead the conversion is performed "in place" and `f1` modifies the original memory storage pointed to by the address `s` .

  Here is an example of how to the function will be called by the test program:

```
s1          db      "Hello world!"
s1Len       equ     $-s1

 ...

            mov     eax, s1
            push    eax
            mov     eax, s1Len
            push    eax
            call    f1

            mov    ecx, s1
            mov    edx, s1Len
            mov    eax, 4
            mov    ebx, 1
            int    0x80
```

  The code above will print "HELLO WORLD!"

  Note: your function should work well with empty strings, and not fall into infinite or close-to-infinite loops! An empty string has zero length.

- `f2(a, b, c)` : `f2` receives three 32-bit integer parameters in the stack and computes `4*(a + b) + 3*c` and returns the result in `eax` . Here is an example of how the function should be called:

```
a           dd      3
b           dd      5
c           dd      7
```

```
...
            push    dword[a]
            push    dword[b]
            push    dword[c]
            call    f2

; upon return from f2, eax should contain 4(3+5) + 3*7 = 32 + 21 = 53
```

- `f3( int_array, n )` : this function receives the address of an array of 32-bit integers, as well as the number of integers in the array. Both parameters are passed as 32-bit integers on the stack. `f3` returns in `eax` the number of negative integers in the array. `f3` should not modify any other registers.

  Here is an example of how to call the function `f3` :

```
array     dd       3, -5, 0, -1, 2, 10, 100, 4, 1
arrayLen  equ      ($-array)/4                    ; why divide by 4? ;)



 ...
            mov     eax, array
            push    eax
            mov     eax, arrayLen
            push    eax
            call    f3

            ; upon return from f3, eax contains 2,
            ; because there are 2 negative numbers in array.
```

  Your function should work well with empty arrays, and not fall into infinite or close-to-infinite loops! An empty array has zero length.

- `f4( a, b, c )` : this function receives three 32-bit non-negative integers in the stack, and returns the smallest of the three in `eax` . `f4` should not modify any other registers.

  Here is an example of how to call the function `f4` .

```
x         dd        0x0fffffff
y         dd        5
z         dd        20

 ...

          push    dword[x]
          push    dword[y]
          push    dword[z]
          call    f4

          ; upon return from f4, eax contains 5,
          ; because 5 is the smallest non-negative int.

          push    dword[x]
          push    dword[x]        ; pushing x twice!
          push    dword[z]
          call    f4

          ; upon return from f4, eax contains 20.
          ; because 20 is the smallest of the 3 non-negative ints.
```

1.4.1 Skeleton Program

Here is a skeleton file for you to use:

```
;;; ------------------------------------------------------
;;; hw8_3.asm
;;; Your Name
;;; Add your documentation here
;;; ------------------------------------------------------

        global  f1, f2, f3, f4

f1:
        ret

f2:
```

```
        ret

f3:
        ret

f4:
        ret
```

1.4.2 FAQ

- 0 is a non-negative integer
- An empty string is a string with 0 characters
- An empty array is an array with 0 elements
- A program that takes more than 10 seconds to run will automatically get a 0/100 grade. Most tests applied to your program should not take longer than a fraction of a second.
- `f1` and `f2` may modify any of the registers without penalty. `f3` and `f4` should either leave all registers other than `eax` untouched or restore them to the values they had before the function was called. A program with `f3` or `f4` changing any of the registers aside from `eax` will automatically get a 0/100 grade.
- You may assume that all integers are signed, 2's complement integers and that you can safely use the jump instructions from the "Signed Comparisons" table from *Art of Assembly* for comparing them.

1.4.3 Making Your Functions Known to Other Programs

Your `hw8_3.asm` program should contain only the 4 functions described above. You may add other helper functions if it makes your programming task easier. For example, you may want to have a function that tests an integer and returns 1 or 0 depending on whether the integer is negative or not. There is no need for a `.data` segment or for a `_start` label in your program. However, you need to make your functions known to the real main program that will be provided in a different file. For this, you need to have this line (or several lines like it) at the top of your program:

```
        global f1, f2, f3, f4
```

1.4.4 Testing Your Code

Here is a simple program that you can use as a main program to test your functions. This is not a full-fledged program that will test all the features of your functions. Feel free to make this code more sophisticated to make sure your functions will work with special cases, such as an array with only negative numbers, or an array of identical values.

```
;;;-----------------------------------------------------------
;;; test0.asm
;;; tests the functions in hw8_3.asm by calling each one
;;; and printing its result.
;;; This is a skeleton test program!
;;;-----------------------------------------------------------
    section .data

tag1     db  "Testing f1", 10
tag2     db  "Testing f2", 10
tag3     db  "Testing f3", 10
tag4     db  "Testing f4", 10
tagLen   equ     $-tag4

;;; -------------------------------------------------------
;;; test data for f1
s1   db  "Hello there! 'az' AZ ~!@#<>?/ 1234578890 "
s1Len    equ     $-s1

;;; -------------------------------------------------------
;;; test data for f2
x    dd  2
y    dd  5
z    dd  10

;;; -------------------------------------------------------
;;; test data for f3 (number of negative ints)
Table1   dd  1, -21, 33, 303, 1001, 2001
Table1Len equ    ($-Table1)/4

;;; -------------------------------------------------------
;;; test data for f4 (find smallest)
x1   dd  10
```

```
x2   dd  20
x3   dd  15


    section .text

    extern  _printString, _println, _printInt, _printRegs
    extern  f1, f2, f3, f4

    global   _start
_start:
;;; ---------------------------------------------------
;;; test f1
testF1:
    mov  ecx, tag1
    mov  edx, tagLen
    call     _printString

    mov  eax, s1    ; pass s1 to f1
    push     eax
    mov  eax, s1Len
    push     eax
    call     f1
    mov  ecx, s1
    mov  edx, s1Len
    call     _printString
    call     _println

;;; ---------------------------------------------------
;;; test f2: computes  4*(a + b) + 3*c
;;; a = 2  b = 5  c = 10
;;; should return 4(2+5)+3.10 = 28 + 30 = 58
testF2:
    mov  ecx, tag2
    mov  edx, tagLen
    call     _printString

     push   dword[x]    ; pass x as a, y as b, z as c to f2
     push   dword[y]
```

```
     push   dword[z]
     call   f2
     call   _printInt
     call   _println


;;; ------------------------------------------------------
;;; test f3
testF3:
    mov  ecx, tag3
    mov  edx, tagLen
    call     _printString

    mov eax, Table1 ; pass Table and length to f3
    push    eax
    mov eax, Table1Len
    push    eax
    call    f3
    call    _printInt
    call    _println

;;; ------------------------------------------------------
;;; ; test f4
testF4:
    mov  ecx, tag4
    mov  edx, tagLen
    call     _printString

    push    dword[x1]
    push    dword[x2]
    push    dword[x3]

    call   f4

    call   _printInt
    call   _println

;;; ; exit
exit:
```

```
    mov     ebx, 0
    mov     eax, 1
    int     0x80
```

Assemble, link, and run as follows:

```
231a-zz@gremlin:~$ nasm -f elf test0.asm
231a-zz@gremlin:~$ nasm -f elf hw8_3.asm
231a-zz@gremlin:~$ nasm -f elf 231Lib.asm
231a-zz@gremlin:~$ ld -melf_i386  -o test0 test0.o hw8_3.o 231Lib.o
231a-zz@gremlin:~$ ./test0
Testing f1
HELLO THERE! 'AZ' AZ ~!@#<>?/ 1234578890
Testing f2
58
Testing f3
1
Testing f4
10
```

Note: the simple test program above does not check whether `f3` or `f4` modify the `ebx`, `ecx`, `edx`, `esi`, or `edi` registers, nor does it test how your functions react to empty strings or arrays. But the test program that the Moodle autograder uses will!

1.4.5 Submission

Submit your program using the Moodle link labeled "8.3 Functions in Assembly" as a file named `hw8_3.asm`.

1.5 Problem 4: Recursion in Assembly

1.5.1 Your Assignment

Write a recursive function `fib3( n )` that computes the modified "sum-of-previous-three" Fibonacci terms, defined as follows:

```
fib3(1) = 1
fib3(2) = 1
fib3(3) = 1
fib3(4) = 3
fib3(5) = 5
...
fib3( n ) = fib3( n-1 )+fib3( n-2 )+fib3( n-3 )
fib3( n ) = 0 if n <= 0
```

Store your function in an assembly file called `hw8_4.asm` that does must not contain a `_start` label, similarly to the way you wrote the 4 functions (`f1` , `f2` , `f3` and `f4`) of Problem 3. Your file will be linked with a separate test program.

1.5.2 Requirements

- Your function `fib3` must be recursive. No credit will be given to a non-recursive function. You may receive 100% for a non-recursive function from the autograder, but all submissions will be checked later by hand.
- Your `fib3()` function should return its result in `eax` .
- Your function should expect the parameter `n` to be a signed 32-bit integer.
- Your program cannot use an array to help compute the `fib3` terms.

Here is an example of how your function will be called from the test program:

```
        extern fib3, _printDec

        mov    eax, 10          ; will compute fib3(10)
        push   eax              ; pass param to fib3 in stack
        call   fib3             ; will return the result in eax
        call   _printDec        ; will print the result
```

1.5.3 Submission

Submit your program using the Moodle link labeled “8.4 Recursion in Assembly” as a file named `hw8_4.asm` .

1.6 Problem 5: Bash Lab #5

For Problem 5, complete Bash Lab #5, a self-paced Linux/Bash lab, described in a separate PDF on Moodle. This lab has no associated Moodle quiz to submit when you have completed it, but you will be tested on the material in your final exam.

1.7 Problem 6 (Optional): Bash Lab #6

For Problem 6, complete Bash Lab #6, a self-paced Linux/Bash lab, described in a separate PDF on Moodle. This lab has no associated Moodle quiz to submit when you have completed it. This lab is optional; You will not be tested on the material in your final exam, however, you will find the things you learn in it useful in the future whenever you work with `bash` or Linux.