CS231-01 Fall 2020
Homework Assignment #7 (Individual)
Due: Tuesday, 11-24-2020, 11:59 PM Anywhere on Earth time (AoE)
Via Moodle

---

1 Assignment #7

1.1 Readings

Part of the readings for this assignment are drawn from *The Art of Assembly*, an additional online text. Its advantage is that it has very detailed information about how each of the x86 assembly language instructions works. Unfortunately, however, it reads much more like a manual for experienced assembly programmers, mixing introductory material with advanced topics, and discussing more commonly used instructions alongside rarely used ones. For now ignore material in this text about condition code bits, "flags", "flag" registers, and segment registers. Also ignore other topics as recommended below.

- Read these sections in *The Art of Assembly*:
  - *Sections 1.3-1.8*. They cover logical operations and sign extensions.
  - *Section 6.5, Arithmetic Instructions*. Read introduction only.
  - *Section 6.5.1, The Addition Instructions: ADD, ADC, INC, XADD, AAA, and DAA*. Read intro only.
  - *Section 6.5.1.2, The ADD and ADC Instructions*. Ignore ADC.
  - *Section 6.5.1.2, The INC Instruction*
  - *Section 6.5.2, The Subtraction Instructions: SUB, SBB, DEC, AAS, and DAS*. Pay attention to SUB and DEC and skip the parts about SBB, AAS, and DAS.
  - *Section 6.5.6, The Multiplication Instructions: MUL, IMUL, and AAM*. Pay attention to MUL, but stop reading at "Imul (integer multiplication) operates on signed operands…" and ignore everything about IMUL and AAS.
  - *Section 6.5.7, The Division Instructions: DIV, IDIV, and AAD*. Pay attention to DIV, and ignore IDIV and AAD.
  - *Section 6.6.1, The Logical Instructions: AND, OR, XOR, and NOT*
  - *Section 6.9.6, The LOOP instruction*. Ignore what it says about `jcxz` / `jecxz` and `dec/jne` for now.
- Read *Dive into Systems* Chapter 4, Sections 4.5-4.9
- Optionally read *Dive into Systems* Chapter 5, Sections 5.1-5.2

1.2 Problem 1: Counting Sheep

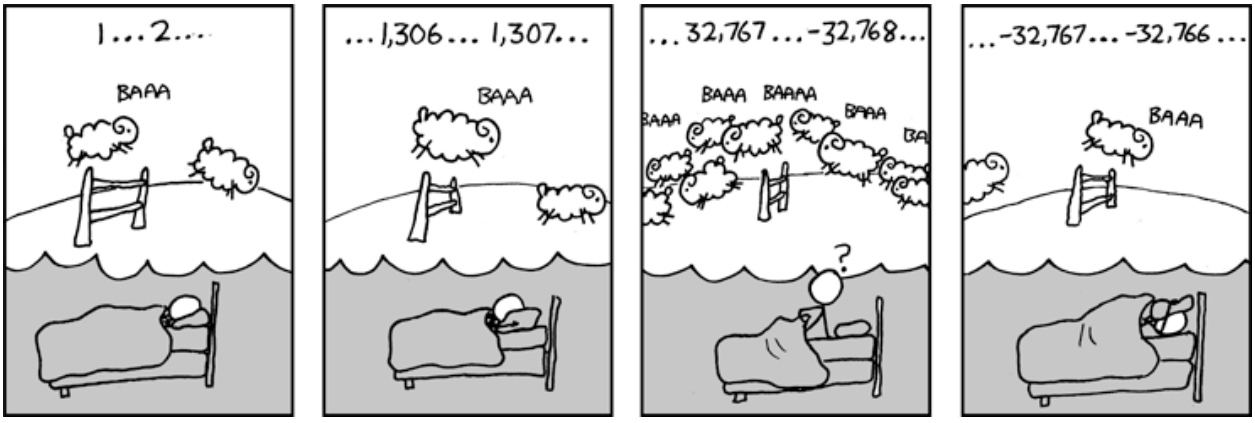

Figure 1: "Counting sheep cartoon" (from xkcd.com)

Above is a famous XKCD cartoon. If you're a computer scientist who understands 2's complement, you'll recognize that the programmer who's trying to fall asleep and is counting sheep, is actually using a 16-bit 2's complement integer to maintain their counter.

Assume that, instead, the numbers printed on the cartoon had been:

```
1...
2...
1,306...
1,307...
262,143...
-262,144...
-262,143...
-262,142...
```

What is the size, in bits, of the 2's complement integers used by our programmer now?

1.2.1 Submission

Submit your answer using the Moodle link labeled "7.1 Counting Sheep".

1.3 Problem 2: Bash Lab #4

For Problem 2, complete Bash Lab #4, a self-paced Linux/Bash lab, described in a separate PDF on Moodle. You will need to complete the "7.2 Bash Lab #4 Quiz" on Moodle before the deadline for this lab to count for your Assignment #7 grade (this is described in the separate PDF).

1.4 Problem 3: Character Decoder

1.4.1 Preparation

Recreate this program in your gremlin account (or you may get it by executing `getcopy hw7_3_prep.asm`)

```
;;; hw7_3_prep.asm
;;;
;;; To assemble, link, and run:
;;;     nasm -f elf  hw7_3_prep.asm
;;;     ld -melf_i386 -o hw7_3_prep hw7_3_prep.o 231Lib.o
;;;     ./hw7_3_prep
;;;
    extern  _printString
    extern  _println
    extern  _getInput

        section .data
prompt  db      "> "
x       dd      0

        section .text
        global  _start
_start:
;;; display prompt
    mov eax, 4
    mov ebx, 1
    mov ecx, prompt
    mov edx, 2
    int 0x80
```

```
;;; get 32-bit integer, as a decimal
    call    _getInput

;;; now int entered by user is in eax.  Save in x
    mov dword[x], eax

;;; display x as a string of 4 ascii chars
    mov eax, 4
    mov ebx, 1
    mov ecx, x      ; address of "string"
    mov edx, 4      ; number of chars: 1 int = 4 bytes
    int 0x80

    call    _println    ; print a line-feed

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

Assemble, and link as indicated in the header. Run it as follows:

```
cs231a-zz@gremlin ~ $ ./hw7_3_prep
> 1146441548
LOUD
cs231a-zz@gremlin ~ $
```

1.4.2 "Endianness"

Note the output of the program: "LOUD". When you enter 1146441548 at the prompt, the program takes this integer and stores it into the 32-bit, 4-byte space (allocated with `dd`) in RAM/main memory associated with the label `x` in your program. 1146441548 in hexadecimal is `0x44554F4C`. So the contents of memory near the address represented by the label `x` will be the following:

Address	Contents
`x`	`0x4C`
`x+1`	`0x4F`
`x+2`	`0x55`
`x+3`	`0x44`
…	…

Notice that since each hexadecimal digit is four bits, a byte is represented by a consecutive pair of hexadecimal digits, and that the four bytes, as pairs of hex digits, appear in "reverse" in comparison to the way we usually read a number from left to right. `0x4C` , which is the least significant, rightmost byte in the hex representation of the number, appears as the first byte starting from the `x` address, followed by `0x4F` , etc. In *Dive into Systems* Section 4.7, you'll read about byte order and "endianness", and how x86 is a "little-endian" architecture, meaning that the CPU stores integers in RAM from the least significant byte ("little end") to the most significant byte in consecutive addresses.

Let us now look at `x` as a string of characters. NASM will not care if we treat a group of 4 bytes as an 32-bit 4-byte value (analogous to a Java/C `int`) in one part of the program, and as a string (analogous to sequence of 1-byte `char` s) in another part of the program.

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. `0x4C` is the ASCII code for 'L', `0x4F` is the ASCII code for 'O', `0x55` is the ASCII code for 'U', and `0x44` is the ASCII code for 'D'. So, we can think of the sequence of four bytes pointed to by `x` and subsequent addresses as this:

Address	Contents
`x`	`'L'`
`x+1`	`'O'`
`x+2`	`'U'`
`x+3`	`'D'`
…	…

So when the program prints the 4-byte string starting at `x` , we get the word "LOUD" printed to the console. If you want to try it yourself, here's a way to find integers corresponding to 4-character strings:

- Pick a 4-character word. Say, "GOOD"
- Find the hexadecimal codes for G, O, and D: `0x47`, `0x4F`, and `0x44`.
- List the hexadecimal codes "backwards": `0x444F4F47`
- Use an online converter to figure out what the decimal version of `0X444F4F47` is. You will get 1146048327
- Enter this number into your `hw7_3_prep` program and verify that you get `"GOOD"` as the output.

1.4.3 A More Complex Decoder in Assembly

Your assignment is to write an assembly program inspired by the `hw7_3_prep.asm` program, this one named `hw7_3.asm` that can be used to decipher a secret message entered as an integer. The secret message will be printed and will contain 4 characters.

Your program will read a 32-bit integer from the user, decipher it (explained shortly), and print a string of 4 characters corresponding to the 4 bytes stored in the 32-bit integer. We assume that the integer will be unsigned for this assignment.

Here is an example:

```
./hw7_3
> 2789908291
Love
```

Here is how your program should decipher the 32-bit integer. As with `hw7_3_prep.asm`, we assume that the integer is stored in RAM at label `x`, defined as a `dd`:

```
x       dd      0
```

As with `hw7_3_prep.asm`, we get `x` by using the `_getInput` function of the `231Lib` library:

```
        call    _getInput
        mov     dword[x], eax
```

Your program will "decrypt" the 4 bytes at Addresses `x`, `x+1`, `x+2`, and `x+3` as follows:

- The byte at Address `x` will have its "lower" 4 bits "flipped", `0` 's changed to `1` 's and `1` 's changed to `0` 's. "Lower" 4 bits refers to the least significant, rightmost 4 bits. For example, if the byte contains `0x1A` , or `00011010` binary, the lower 4 bits are the four on the right, `1010` . Flipping these lower 4 bits results in `0101` , resulting in `00010101` or `0x15` for the full eight bits.
- The byte at Address `x+1` will have the "upper" or top 4 bits flipped. "Upper" 4 bits refers to the most significant, leftmost 4 bits. For example, if the byte contains `0x12` , or `00010010` in binary, the upper 4 bits are the four on the right, `0001` . Flipping the upper 4 bits results in `1110` , resulting in `11100010` , or `0xE2` for the full 8 bits.
- The byte at Address `x+2` will have the middle 4 bits flipped. For example, if the byte contains `0xF3` , or `11110011` binary, the middle 4 bits are the `1100` . Flipping these results in `0011` for those four bits, and `11001111` or `0xCF` for the full 8 bits.
- The byte at Address `x+3` will have its top, leftmost 2 bits and its bottom, rightmost 2 bits flipped. For example, if the byte contains `0x15` , or `00010101` binary, the leftmost `00` will be flipped to `11` , and the rightmost `01` will be flipped to `10` resulting in `11010110` , or `0xD6` for the full 8 bits.

Once the 4 bytes are "decrypted" as just explained, your program will simply print `x` as if it were a string of 4 ASCII characters. just as it is done in `hw7_3_prep.asm` . Your program should also print a line-feed character at the end of the string. In other words, your program will print 5 chars, the 4 characters/bytes contained in `x` , plus a line-feed.

Another example:

```
./hw7_3
> 2941493068
Cool
```

1.4.4 Implementation Details

- As with `hw7_3_prep.asm` , your program should be linked with the `231Lib` library.
- As in the `hw7_3_prep.asm` program, use the `_getInput` function of the `231Lib` library to get the integer
- Your program must output a line-feed at the end of the 4-character message, or Moodle may not give your program full credit.

1.4.5 Tips

Make sure your program does not output invisible ASCII characters. A good way to catch them is to pass the output of your program through the `cat -v` command. Here is an example where I modified my program to make it output additional bytes in memory:

```
./hw7_3 | cat -v
> 2941493068
Cool
^@^@^@^@^@^@^@^@^@
```

The `^@` characters do not normally show up when the program is run without piping the output through `cat -v` .

Other numbers that should translate to recognizable 4-letter English words:

```
    3108409701
    3109389662
    2824443240
    2790760293
    2722795845
    2823722341
    2890309485
```

If you need to debug your code, and need to print the contents of the registers, at any point, just call `_printRegs` , as follows:

- first, add this line at the top of your program:

  ```
  extern  _printRegs
  ```

- In your code segment, when you want to print the registers, just add this instruction:

  ```
  call    _printRegs
  ```

  and all the registers will be printed in decimal and hexadecimal:

```
eax 65766F4C 1702260556 1702260556
ebx 00000001 1 1
ecx 080495BD 134518205 134518205
edx 00000004 4 4
edi 00000000 0 0
esi 00000000 0 0
```

(We'll learn about `edi` and `esi` later)

1.4.6 Submission

Submit your program using the Moodle link labeled "7.3 Character Decoder" as a file named `hw7_3.asm` .

1.5 Problem #4 `Mul`

Write a program similar in its format to the ones you wrote for the previous assignment that prompts the user for 3 integers, `a` , `b` , and `c` , and computes the expression

```
ans =  4*(a^2 + b^2) + c
```

where the caret symbol, `^` , represents the exponentiation/power operator. Do not try to find a power instruction. There isn't one. Instead, multiply the operand by itself using the `mul` instruction.

1.5.1 Notes

- Please use `hw6_4_skel.asm` from Assignment 6 Problem 4 as a "skeleton" file for this assignment (Otherwise you may get errors from the autograder that your data section is not correct)
- Your program should output the text `ans =` followed by the value of `ans` , followed by a line-feed.
- Your program will need to treat `a` , `b` , `c` , and `ans` the same way Java treats `int` s, as 32-bit values.
- You may assume that `ans` and any product produced by a `mul` instruction are always small enough to fit in a 32-bit register.

1.5.2 Examples

```
cs231a-zz@gremlin ~ $ ./hw7_4
> 1
> 2
> 3
ans = 23

cs231a-zz@gremlin ~ $ ./hw7_4
> 10
> 20
> 30
ans = 2030

cs231a-zz@gremlin ~ $ ./hw7_4
> 100
> 200
> 300
ans = 200300
```

1.5.3 Submission

Submit your program on Moodle using the link labeled "7.4 The Mul Instruction" as a file named `hw7_4.asm` .

1.6 Problem 5: Modified Fibonacci

Write a program in assembly that computes a modified version of the Fibonacci series. The series is defined recursively as follows:

```
f(1) = 1
f(2) = 1
f(3) = 1
f(n) = f(n-1) + f(n-2) + f(n-3)       if n > 3
```

where each term is a 32-bit integer. Your program only needs to be able to correctly calculate values of the series for n > 3 (it's OK if your program crashes or loops infinitely on other inputs).

Feel free to name your program file `hw7_5.asm` and make it prompt the user for an index (integer number), and output the modified Fibonacci term corresponding to that index.

Examples:

```
cs231a-zz@gremlin ~ $ ./hw7_5
> 4
3
cs231a-zz@gremlin ~ $ ./hw7_5
> 5
5
cs231a-zz@gremlin ~ $ ./hw7_5
> 6
9
cs231a-zz@gremlin ~ $ ./hw7_5
> 10
105
cs231a-zz@gremlin ~ $ ./hw7_5
> 18
13745
```

You may assume that the user will always input a positive integer greater than 3 and less than 1000. For this problem your program should not output `ans =` , or any similar text, just the value.

1.6.1 Tips:

As you are working on this program, you may run into a situation where you run your executable and it gets into an infinite loop and doesn't stop on its own. In this situation, pressing Control-C will terminate the running process and return you to the `bash` prompt.

1.6.2 Submission

Submit your program on Moodle using the link labeled "7.5 Modified Fibonacci" as a file named `hw7_5.asm` .