Assignment 4 – Solution

Assignment 4 - Objectives:

- Hand tracing assembly code
- Translating x86-64 assembly code into C code
- Writing x86-64 assembly code

For each of our team-based assignments, you can either:

- Create a group with the same partner,
- Change your group (select a different partner), or
- Decide to work on you own.

Group of two (2):

- You can do Assignment 4 in a group of two (2) or you can work on your own.
- If you choose to work in a group of two (2):
  - Crowdmark will allow you to select your teammate (1). Both of you will only need to submit one assignment and when the TAs mark this one assignment, the marks will be distributed to both of you automatically (just like on CourSys).
  - I doubt Crowdmark will allow you to team up with a student from the other section, but try it and let me know if it works.
  - You can always work with a student from the other section, but both of you will need to submit your assignment separately, i.e., Crowdmark will not consider the both of you as a group.
  - You will need to form the same group of 2 on CourSys.
- If you choose to work on your own:
  - You will need to form a group of 1 on CourSys. (This is the way CourSys works.)

Requirements for this Assignment 4:

- Always show your work (as illustrated in lectures), if appropriate, and
- Make sure the pdf/jpeg/png documents you upload are of good quality, i.e., easy to read, therefore easy to mark! :) If the TA cannot read your work (and this has been the case in our past 3 assignments), you will get 0. :(

Marking scheme:

- This assignment will be marked as follows:
  - Questions 1, 2 and 3 will be marked for correctness.
- The amount of marks for each question is indicated as part of the question.
- A solution will be posted on Monday after the due date.

Deadline:

- Friday Feb. 10 at 23:59:59 on Crowdmark and CourSys.
  - Yes, it is the case that the deadline for our Assignment 4 coincides with our Midterm 1. This is meant this way. Why? Because doing Assignment 4 is part of our studying for Midterm 1.
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Enjoy!

Q1 (6 points)

Hand tracing assembly code

Consider the following assembly code:

```
abs(int):
  movl    %edi, %eax
  movl    %edi, %edx
  sarl    $31,  %edx
  xorl    %edx, %eax
  subl    %edx, %eax
  ret
```

which is what an older version of our **gcc** compiler produced, using the optimization level 1 (**-O1**), when it compiled the following **abs(int x)** function:

```
int abs(int x) {
  if ( x < 0 )
    x = -x;
  return x;
}
```

Notice how **gcc** assembles **abs(int x)** without branching, i.e., without affecting the execution flow (without using the jump instruction). We shall see in our next Unit that branching is rather unpredictable and may cause problem in the execution pipeline of the microprocessor.

Let's hand trace the above assembly code using the Test Case 1: x = 7 and expected result = 7:

movl %edi, %eax copies the content of %edi (x = 7) into %eax.

Once executed, here is the content of our registers:

```
%edi <- 00000000000000000000000000000111

        <-           29 0's          ->

%eax <- 00000000000000000000000000000111

        <-           29 0's          ->
```

movl %edi, %edx copies the content of %edi (x = 7) into %edx.

Once executed, here is the content of our registers:

```
%edi <- 00000000000000000000000000000111

        <-           29 0's          ->

%edx <- 00000000000000000000000000000111

        <-           29 0's          ->
```

`sarl $31, %edx` shifts the content of %edx right (arithmetic - fill in with sign bit -> 0) 31 times.

Once executed, here is the content of our register:

```
%edx <- 00000000000000000000000000000000

        <-            32 0's          ->

xorl %edx, %eax
%eax <- 00000000000000000000000000000111
^
%edx <- 00000000000000000000000000000000
%eax <- 00000000000000000000000000000111

subl %edx, %eax
%eax <- 00000000000000000000000000000111
-
%edx <- 00000000000000000000000000000000
%eax <- 00000000000000000000000000000111
ret
```

And the return value is `%eax = 00000000000000000000000000000111` = 7

Your turn now! Your task in this question is to hand trace the above assembly code using the Test Case 2: x = -7 and expected result = 7.

Use the above hand tracing as a model for your answer, i.e., follow its format when showing the result of executing each instruction. Make sure you show the full content of both registers for the **mov*** instructions and make sure you show the full content of the modified register for the other instructions. Finally, show the value that is returned to the caller (or calling) function.

Remember that x – (-y) = x + y.

Optional - there is nothing for you to do in this part of Q1 except to read and enjoy it! :) In case you are curious, below is what our current version of `gcc` produces, using the optimization level 1 (`-O1`), when it compiles the above `abs(int x)` function:

```
abs(int):
  movl    %edi, %eax
  cltd
  xorl    %edx, %eax
  subl    %edx, %eax
  ret
```

As you can see, `gcc` now produces a shorter assembly program, replacing assembly instructions

```
  movl    %edi, %edx
  sarl    $31,  %edx
```

with `cltd`.

In plain English, this `cltd` instruction fills `%edx` with the bit sign (MSb) of the value stored in `%eax`. Therefore, if `%eax` contains 0x0000BEEF, i.e., a positive value, then `%edx` contains `0x00000000` once `cltd` has executed, and if `%eax` contains `0xBEEF0000`, i.e., a negative value, then `%edx` ends up containing `0xFFFFFFFF` once `cltd` has executed.

Using this instruction can be an efficient way of creating a bit mask.

abs (…) version 2 **Test case 2) x = - 7** **Expected result: 7**	**Result of executing instruction in the left column**
`movl %edi, %eax`	Copy content of %edi (x = -7) into %edx, i.e., %edi <- 11111111111111111111111111111001 &lt;- 29 1's -> %eax <- 11111111111111111111111111111001 &lt;- 29 1's ->
`movl %edi, %edx`	Copy content of %edi (x = 7) into %eax, i.e., %edi <- 11111111111111111111111111111001 &lt;- 29 1's -> %edx <- 11111111111111111111111111111001 &lt;- 29 1's ->
`sarl $31, %edx`	Shift right arithmetic (fill in with sign bit -> 1): %edx <- 11111111111111111111111111111111 &lt;- 32 1's ->
`xorl %edx, %eax`	%eax <- 11111111111111111111111111111001 ^ %edx <- 11111111111111111111111111111111 %eax <- 00000000000000000000000000000110
`subl %edx, %eax`	%eax <- 00000000000000000000000000000110 - %edx <- 11111111111111111111111111111111 %eax <- 00000000000000000000000000000111 same as adding two's complement of %edx %eax <- 00000000000000000000000000000110 + %edx <- 00000000000000000000000000000001 %eax <- 00000000000000000000000000000111
`ret`	and this instruction is executed.

And the return value is `%eax = 00000000000000000000000000000111 = 7`

- `sarl $31, %edx` -> creates a mask made of 32 sign bits
  - if x is positive then the mask is made of 32 0's
  - if x is negative then the mask is made of 32 1's
- `xorl %edx, %eax` -> then the mask is xor'ed with x
  - If the mask is made of 32 0's (i.e., x is positive) then xor produces x.

    - If the mask is made of 32 1's (i.e., x is negative) then xor produces the ones' complement of x, i.e., the bits of x are flipped. Note that this is not quite the negative version of x yet.
- `subl %edx, %eax` -> x – mask = x
    - If the mask is made of 32 0's (i.e., x is positive) then x remains x therefore no change because we are subtracting 0 from x.
    - If the mask is made of 32 1's (i.e., x was initially negative) then this instruction is subtracting -1 (i.e., adding 1 -> x –(-1)) to the ones' complement of x, hence x is now the two's complement of its initial negative value, i.e., x is now positive.

Q2 (8 points)

Translating x86-64 assembly code into C code

Read the entire question before answering it!

Consider the following assembly code:

```
# long func(long x, int n)
func:
  movl  %esi, %ecx
  movl  $1,   %edx
  movl  $0,   %eax
  jmp   cond
loop:
  movq  %rdi, %r8
  andq  %rdx, %r8
  orq   %r8,  %rax
  salq  %cl,  %rdx     # shift left the value stored in %rdx by an amount
                       # dictated by the value stored in %cl - see Note below
cond:
  testq %rdx, %rdx
  jne   loop
  ret
```

Note: Section 3.5.3 of our textbook explains how a **shift** instruction works when it has the register `%cl` as one of its operands. Check it out!

The assembly code above was generated by compiling a C function that had the following overall form:

```
long func(long x, int n) {
  long result = _________;
  long mask;

  for (mask = _________ ;mask _________ ;mask = _________ )
    result |= ___________________________ ;
  return result;
}
```

Your task is to fill in the missing parts of the C function `func(...)` above to get a program equivalent (although it may not be exactly the same) to the generated assembly code displayed above it.

You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the C function variables.

You may also find the five questions below helpful in figuring out the assembly code.

Note that you do not have to submit the answers to these five questions as part of Assignment 4 as these answers will be reflected in the C function you are asked to complete and submit.

Here are these five questions:

a. Which registers hold program values x, n, `result` and `mask`?

b. What is the initial value of `result` and of `mask`?

c. What is the test condition for `mask`?

d. How is `mask` updated?

e. How is `result` updated?

a. Which registers hold program values **`x`** **`-> %rdi`**, **`n`** **`-> %esi`**, **`result`** **`-> %eax as well as %rax`**, and **`mask`** **`-> %rdx`**?
b. What is the initial value of **`result`** **`-> 0`** and of **`mask`** **`-> 1`**?
c. What is the test condition for **`mask`** **`-> loop when mask != 0`**?
d. How is **`mask`** updated **`-> by shifting its value 1 left by value represented by least significant (LS) 6 bits of %cl`**?
e. How is **`result`** updated **`-> |= (x & mask)`**?

Solution:

```
long func(long x, int n) {
     long result = 0;
     long mask;
     for ( mask = 1; mask != 0; mask = mask << n )
        result |= (x & mask);
     return result;
}
```

Solution: with matching x86-64 assembly instructions

```
long func(long x, int n) {
     long result = 0;  movl $0, %eax
     long mask;
          movl $1, %edx            salq %cl, %rdx
     for ( mask = 1; mask != 0; mask = mask << n )
               testq %rdx, %rdx
                 jne loop
        result |= (x & mask);  andq %rdx, %r8
   orq %r8, %rax
     return result;
}
```

Q3 (12 points)

Writing x86-64 assembly code

Download `Assn4-Q3-Files.zip` from our course web site (under Assignment 4), in which you will find a `makefile`, `main.c` and an incomplete `calculator.s`. The latter contains four functions implementing some arithmetic and logical operations in assembly code.

Your task is to complete the implementation of the three incomplete functions, namely, `plus`, `minus` and `mul`. In doing so, you must satisfy the requirements found in each of the functions of `calculator.s`. You must also satisfy the requirements below.

You will also need to figure out what the function `XX` does and once you have done so, you will need to change its name to something more descriptive in `main.c` and in `calculator.s` as well as adding its description in the indicated place in `calculator.s`.

Requirements:

- Your code must be commented such that others (i.e., TAs) can read your code and understand what each instruction does.
    - About comments:
        - Comment of Type 1: Here is an example of a useful comment:

          ```
          cmpl  %edx, %r8d        # if j (%r8d) < N (%edx), continue looping
          ```

        - Comment of Type 2: Here is an example of a **not so** useful comment:

          ```
          cmpl  %edx, %r8d        # compare %r8d to %edx
          ```

          Do you see the difference?
          Make sure you write comments of Type 1. :)

    - Also, describe the algorithm you used to perform the multiplication in a comment at the top of the `mul` function.
- Your code must compile using `gcc` and execute on our ***target machine***.

- Add a header comment block to your `calculator.s`, which must include the filename, the purpose/description of your program, your name and the date.
- Remember that the **function call protocol** of `x86-64` says that the register `%edi` contains the parameter (or argument) x, the register `%esi` contains the parameter (or argument) y and that the register `%eax` carries the return value. Make sure the code of all your four functions abides to this protocol.
- You may use registers `%rax`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10` and `%r11` in your code.
- You must **not** modify the values of registers `%rbx`, `%rbp`, `%rsp`, `%r12`, `%r13`, `%r14` and `%r15`. We shall soon see why.
- You cannot modify the given `makefile`.

Finally, submit your modified `main.c` and your completed `calculator.s` on **CourSys** and write the word **Done!** in the answer space below so that Crowdmark does not think you haven't completed this question of Assignment 4. Do not upload your files in the answer space below.

Hint for the implementation of the `mul` function:
Long ago, computers were restricted in their arithmetic prowess and were only able to perform additions and subtractions. Multiplications and divisions needed to be implemented by the programmer using the arithmetic operations available.

```
/*
 * Filename: main.c
 *
 * Purpose: Assignment 4 Question 3
 *          Test driver of calculator.s
 *
 * Date: 2023
 *
 * Name: AL
 *
*/

#include <stdlib.h>  // atoi()
#include <stdio.h>   // printf()

int lt(int x, int y);   // Make sure you change the name of
                        // this function - see calculator.s
```

```
int plus(int x, int y);
int minus(int x, int y);
int mul(int x, int y);


int main(int argc, char *argv[]) {
   int x = 0;
   int y = 0;
   int result = 0;

   if (argc == 3) {
      x = atoi(argv[1]);
      y = atoi(argv[2]);

      result = lt(x, y); // Make sure you change the name of
                         // this function - see calculator.s

      // Make sure you change ??? to the appropriate symbol
      printf("%d < %d -> %d\n", x, y, result);

      result = plus(x, y);
      printf("%d + %d = %d\n", x, y, result);

      result = minus(x, y);
      printf("%d - %d = %d\n", x, y, result);

      result = mul(x, y);
      printf("%d * %d = %d\n", x, y, result);

   } else {
     printf("Must supply 2 integers arguments.\n");
     return 1;
   }

   return 0;
}
```

Possible Solution:

```
# filename: calculator.s
# Description: Contains arithmetic and logical functions:
#              lt (less than function),
#              plus (without the use of "add"),
#              minus (without the use of "sub"),
#              mul (without the use of "imul" or the like)
# Date:
# Name: Your Name

   .globl    lt
   .globl    plus
   .globl    minus
   .globl    mul

# x in edi, y in esi

lt: # less than
    xorl  %eax, %eax          # 0 -> %eax
    cmpl  %esi, %edi          # x < y ?
    setl  %al                 # if so then 1 (true) -> %al
   ret

plus:  # performs integer addition
# Requirement:
# - you cannot use add* instruction
# - you cannot use a loop
    leal  (%edi,%esi), %eax   # x + y -> %eax
    ret


minus: # performs integer subtraction
# Requirement:
# - you cannot use sub* instruction
# - you cannot use a loop
    movl  %esi, %eax
    negl  %eax
    leal  (%edi,%eax), %eax   # x + (- y) -> %eax
    ret


mul: # performs unsigned integer multiplication
# Requirements:
# - you cannot use imul* instruction
#    (or any kind of instruction that multiplies such as mul)
# - you must use a loop

# algorithm:
#     set sum to 0
#     iterate y times:
#        sum += x
```

```
    xorl  %eax, %eax          # result (sum) = 0
    xorl  %ecx, %ecx          # i = 0
loop:
    cmpl  %esi, %ecx          # i < y ?
    jge   endloop             # if i >= y then terminate
    addl  %edi, %eax          # result (sum) += x
    incl  %ecx                # i++
    jmp   loop                # iterate (loop)
endloop:
    ret
```