Assignment 4

Objectives:

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

Submission: Assignment 4 is a little unusual

- Doing Assigment 4 will help you to prepare for Midterm 1 even though Assigment 4 is due after our Midterm 1.
- On Friday, Oct. 15 by 4pm, submit the following 3 documents on CourSys:
    - **Assignment_4.pdf** which is to contain only your solution to Question 2. You do not have to submit your solution to Question 1. Do Question 1 as part of your studying for our Midterm 1.
    - **`main.c`** and **`calculator.s`**, i.e., your solution to Question 3.
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on the Monday after the due date) in order to provide feedback to the student.

Marking scheme:

- This assignment will be marked as follows:
    - Question 1 is not marked. Do Question 1 as part of your studying for our Midterm 1.
        - A solution to Question 1 will be posted along with this Assignment 4.
    - Questions 2 and 3 will be marked for correctness.
        - Solution to Question 2 and Question 3 will be posted on the Monday after the due date.
- The amount of marks for each question is indicated as part of the question.

---

1. [0 marks] Hand tracing assembly code - Do Question 1 as part of your studying for our Midterm 1.

Consider the following code:

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

C function `abs(…)`

```
    .globl  abs
abs:
    movl    %edi, %eax
    cmpl    $0, %eax
    jge     endif
    negl    %eax
endif:
    ret
```

`abs(…)` version 1 (based on a non-optimized gcc version)

```
    .globl  abs
abs:
    movl    %edi, %edx
    movl    %edi, %eax
    sarl    $31, %edx
    xorl    %edx, %eax
    subl    %edx, %eax
    ret
```

`abs(…)` version 2 (an optimized gcc version)

The left column contains the C function `abs(…)`, the middle column contains the assembly code version of the C function `abs(…)` we wrote in class (we shall call it "version 1") and the right column contains the assembly code version of `abs(…)` the gcc compiler may produce when it assembles the C function `abs(…)` using level 2 optimization ("`-O2`"). We shall call it "version 2".

Notice how gcc assembles `abs(…)` without branching, i.e., without affecting the execution flow (without using the jump instructions). We shall see in our next Unit (Chapter 4 of our textbook) that branching is rather unpredictable and may cause problem in the execution pipeline of the microprocessor. For this reason, the assembly code version (version 1) of `abs(…)` which branches may run more slowly.

In this question, you are asked to hand trace both versions of `abs(…)` using 2 test cases

- Test case 1: `x = 7`, expected result: `7`
- Test case 2: `x = -7`, expected result: `7`

and show the result of executing each instruction. In doing so, complete the tables below:

Note:

- The first table has been completed as an example. Feel free to use it as a model when you complete the other three tables.
- Remember that x – (-y) = x + y.

abs (…) version 1 **Test case 1: x = 7** **Expected result: 7**	**Result of executing instruction in the left column**
movl %edi, %eax	Copy content of %edi (x = 7) into %eax, i.e., %edi <- 00000000000000000000000000000111 <- 29 0's -> %eax <- 00000000000000000000000000000111 <- 29 0's ->
cmpl $0, %eax	3 outcomes: x – 0 > 0 x – 0 < 0 x – 0 = 0 Here since %eax contains 7, then the only possible (true) outcome is 7 – 0 > 0, i.e., when the microprocessor evaluates 7 – 0, it obtains the result 7. This result (being greater than 0: 7 > 0) sets the condition codes to "g" and therefore …
jge endif	… the jump instruction is executed
negl %eax	and this instruction is not executed
endif: ret	and this instruction is executed.

abs (…) version 1 **Test case 2) x = -7** **Expected result: 7**	**Result of executing instruction in the left column**
movl %edi, %eax	
cmpl $0, %eax	
jge endif	
negl %eax	
endif: ret	

abs (…) version 2 **Test case 1) x = 7** **Expected result: 7**	**Result of executing instruction in the left column**
`movl %edi, %edx`	
`movl %edi, %eax`	
`sarl $31, %edx`	
`xorl %edx, %eax`	
`subl %edx, %eax`	
`ret`	

abs (…) version 2 **Test case 2) x = - 7** **Expected result: 7**	**Result of executing instruction in the left column**
`movl %edi, %edx`	
`movl %edi, %eax`	
`sarl $31, %edx`	
`xorl %edx, %eax`	
`subl %edx, %eax`	
`ret`	

2. [8 marks] Translating assembly code into C code - Read the entire question before answering it!

Consider the following assembly code:

```
# long func(long x, int n)
# x in %rdi, n in %esi, result in %rax
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 related to the value in %cl*
cond:
     testq %rdx, %rdx    # Value in %rdx is >0, <0, =0 ?
     jne   loop          # jump if %rdx != 0
                         # fall thru to ret if %rdx = 0
     ret
```

* 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 C code 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 (note: 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 following questions helpful in figuring out the assembly code. Note that you do not have to submit the answers to the five questions below as part fo Assignemnt 4 as these answers will be reflected in the C function you are asked to complete and submit.

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?

3. [12 marks] Writing x86-64 assembly code

Download `Assn4_Q3_Files.zip`, in which you will find a `makefile`, `main.c` and an incomplete `calculator.s`. The latter contains four functions implementing 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`) and add its description in the indicated place in `calculator.s`.

Requirements:

- Your assembly program must follow the following standard:
  - Your code must be commented such that others (i.e., TA's) 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     # loop while j < N
      ```
    - 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 `mul` function.
  - Your code **must** compile (using gcc) and execute on the *target machine*.
  - Each of your code files (`main.c` and `calculator.s`) must contain a header comment block including the filename, the purpose/description of your program, your name and the date.
  - For all of the four functions, the register `%edi` will contain the argument `x` and the register `%esi` will contain the argument `y`. The register `%eax` will carry the return value.
  - You may use registers `%rax`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10` and `%r11` as temporary registers.
  - 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`.

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.
