Assignment 4 - Solution

Objectives:

- Range of C data types
- Hand tracing assembly code
- Translating assembly code into C code
- Writing assembly code

Submission: Assignment 4 is a little unusual

- Do Assignment 4 (all 4 questions) as part of your study for Midterm 1.
- You do not have to submit your solutions to Question 1 and to Question 2.
  - Solutions to Question 1 and Question 2 are posted.
- You do have to submit your solutions to Question 3 and Question 4.
  - Solutions to Question 3 and Question 4 will be posted after their due date.
- Submit your document called Assignment_4.pdf, which must include your answers to Question 3 and Question 4.
  - Add your full name and student number at the top of the first page of your document Assignment_4.pdf.
- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
- If you hand-write your answers (as opposed to using a computer application to write them): When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
- Submit your assignment electronically on CourSys.

Due:

- Submit your document called Assignment_4.pdf, which must include your answers to Question 3 and Question 4 on Thursday, Feb. 13 at 3pm
- Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.
Marking scheme:

This assignment will be marked as follows:

- Questions 1 and 2 are not marked (we do not need to submit them). They are to be done only as part of your study for Midterm 1.
- Questions 3 and 4 will be marked for correctness (we need to submit them). They are also to be done as part of your study for Midterm 1.

The amount of marks for each question is indicated as part of the question.

A solution to Question 3 and Question 4 will be posted after the due date.

1. [0 marks] For study purposes only - Range of C data types

Complete the following table:

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Range expressed in decimal numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>For example: [ -128\textsubscript{10} .. 127\textsubscript{10} ]</td>
</tr>
<tr>
<td>unsigned char</td>
<td>[0 .. 255]</td>
</tr>
<tr>
<td>short</td>
<td>[-32768 .. 32767]</td>
</tr>
<tr>
<td>unsigned short</td>
<td>[0 .. 65535]</td>
</tr>
<tr>
<td>int</td>
<td>[-2147483648 .. 2147483647]</td>
</tr>
</tbody>
</table>
unsigned int (or unsigned) [0 .. \(4.294967295\)]

long [-9.223372037 \( \times 10^{18} \) .. 9.223372037 \( \times 10^{18} \)]

unsigned long [0 .. \(1.844674407 \times 10^{19}\)]

2. [0 marks] For study purposes only - Hand tracing assembly code

Consider the following code:

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

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

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

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 produces 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 Chapter 4 that branching is rather unpredictable and may cause problem in the execution pipeline. 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 other words, complete the tables below:

**Note:**
- The first table has been completed as an example.
- Remember that \( x - (-y) = x + y \).

### abs(...) version 1  
**Test case 1: \( x = 7 \)  
**Expected result: 7**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl %edi, %eax</td>
<td>( \text{edi} \leftarrow 00000000000000000000000000111 )</td>
</tr>
</tbody>
</table>
| cmpl $0, %eax | Since it is false that \( x - 0 < 0 \) => \( 7 - 0 < 0 \)  
i.e.,  
\[
\begin{align*}
00000000000000000000000000111 \\
- 00000000000000000000000000000000 \\
> 00000000000000000000000000000000
\end{align*}
\] ... |
| jge endif | ... then jump instruction executed |
| negl %eax | not executed |
| endif: ret | executed |

### abs(...) version 1  
**Test case 2: \( x = -7 \)  
**Expected result: 7**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl %edi, %eax</td>
<td>( \text{edi} \leftarrow 11111111111111111111111111001 )</td>
</tr>
</tbody>
</table>
| cmpl $0, %eax | Since it is true that \( x - 0 < 0 \) => \( -7 - 0 < 0 \)  
i.e.,  
\[
\begin{align*}
11111111111111111111111111001 \\
- 00000000000000000000000000000000 \\
< 00000000000000000000000000000000
\end{align*}
\] ... |
### jge endif

... then jump not taken

### negl %eax

| eax <- 00000000000000000000000000000111 |
| --- | --- |
| <- 29 0’s | -> |

### endif: ret

executed

---

### abs (...) version 2

**Test case 1: x = 7**

**Expected result: 7**

<table>
<thead>
<tr>
<th>movl %edi, %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>edi &lt;- 00000000000000000000000000000111</td>
</tr>
<tr>
<td>&lt;- 29 0’s</td>
</tr>
<tr>
<td>edx &lt;- 00000000000000000000000000000111</td>
</tr>
<tr>
<td>&lt;- 29 0’s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>movl %edi, %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>edi &lt;- 00000000000000000000000000000111</td>
</tr>
<tr>
<td>&lt;- 29 0’s</td>
</tr>
<tr>
<td>eax &lt;- 00000000000000000000000000000111</td>
</tr>
<tr>
<td>&lt;- 29 0’s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sarl $31, %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>edx &lt;- 00000000000000000000000000000000</td>
</tr>
<tr>
<td>&lt;- 32 0’s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xorl %edx, %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax &lt;- 00000000000000000000000000000111</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td>edx &lt;- 00000000000000000000000000000000</td>
</tr>
<tr>
<td>eax &lt;- 00000000000000000000000000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>subl %edx, %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax &lt;- 00000000000000000000000000000000</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>edx &lt;- 00000000000000000000000000000000</td>
</tr>
<tr>
<td>ea &lt;- 00000000000000000000000000000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>executed</td>
</tr>
</tbody>
</table>

---

### abs (...) version 2

**Test case 2: x = -7**

**Expected result: 7**

<table>
<thead>
<tr>
<th>movl %edi, %edx</th>
</tr>
</thead>
<tbody>
<tr>
<td>edi &lt;- 11111111111111111111111111111001</td>
</tr>
<tr>
<td>&lt;- 29 1’s</td>
</tr>
<tr>
<td>edx &lt;- 11111111111111111111111111111001</td>
</tr>
<tr>
<td>&lt;- 29 1’s</td>
</tr>
</tbody>
</table>
What is happening in version 2 of \texttt{abs(...) is as follows:

- \texttt{sarl $31, \%edx} -> creates a mask made of 32 sign bits
  - if \texttt{x} is positive then the mask is made of 32 0’s
  - if \texttt{x} is negative then the mask is made of 32 1’s

- \texttt{xorl \%edx, \%eax} -> then the mask is xor’ed with \texttt{x}. This has the effect of complementing (flipping) the bits of \texttt{x}.
  - If the mask is made of 32 0’s (i.e., \texttt{x} is positive) then xor produces \texttt{x} – no change.
  - If the mask is made of 32 1’s (i.e., \texttt{x} is negative) then xor produces the complement of \texttt{x} (not quite the negative version of \texttt{x} yet).

- \texttt{subl \%edx, \%eax} -> \texttt{x – mask = x}
  - if the mask is made of 32 0’s (i.e., \texttt{x} is positive) then \texttt{x} remains \texttt{x} – no change because we are subtracting 0 from \texttt{x}
  - if the mask is made of 32 1’s (i.e., \texttt{x} was initially negative) then this instruction is subtracting -1 (i.e., adding 1 - \texttt{x} –(-1)) to the complement \texttt{x}, hence \texttt{x} is now the two’s complement of its initial value (a negative value). So now, \texttt{x} is the positive version of its initial negative value.
3. [10 marks] Also for study purposes - Translating assembly code into C code

Read the entire question before answering it!

Consider the following assembly code:

```assembly
# 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 rdx by content of cl
cond:
    testq %rdx, %rdx  # rdx <= rdx & rdx
    jne loop  # jump if not zero (when rdx & rdx != 0)
    ret
```

The preceding code was generated by compiling C code that had the following overall form:

```c
long func(long x, int n) {
    long result = 0;
    long mask;

    for (mask = 0x1 ; mask != 0 ; mask = mask << n )
        result |= (x & mask);
    return result;
}
```

Your task is to fill in the missing parts of the C function 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 figure out the assembly code (you do not have to provide an answer for them as their answer will be reflected in the C function you are asked to complete):
a. Which registers hold program values \( x, n, \text{result}, \) and \( \text{mask} \)?
b. What is the initial value of \( \text{result} \) and of \( \text{mask} \)?
c. What is the test condition for \( \text{mask} \)?
d. How is \( \text{mask} \) updated?
e. How is \( \text{result} \) updated?

4. [10 marks] Also for study purposes - Writing x86-64 assembly code

Download Assn4_Q4_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, \( \text{plus}, \text{minus} \) and \( \text{mul} \). 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 \( \text{lt} \) does 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. Also, describe the algorithm you used to perform the multiplication in a comment at the top of \( \text{mul} \) function.
  - Your code must compile (using gcc) and execute on the computers in CSIL (using the Linux platform).
  - Your program file 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 must not modify the values of any memory locations.

- You cannot modify the given makefile.

**Hint for the implementation of the \text{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.

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: Thursday Feb. 13
# Name: Your Name

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

# x in edi, y in esi
lt: # less than
  xorl %eax, %eax # eax < 0
  cmpl %esi, %edi # x - y < 0 ?
  setl %al # al <- 1 if x < y
  ret

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

minus: # performs integer subtraction
# Requirement:
# - you cannot use sub* instruction
  movl %esi, %eax
  negl %eax
  leal (%edi,%eax), %eax # eax <- x + (-y)
```
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? (i - y < 0?)
jge     endloop          # exit if i >= y
addl    %edi, %eax       # result (sum) += x
incl    %ecx              # i++
jmp     loop

endloop:
ret