CSci Spring 2020 Section 010 Problem Set 2

Due in plain text or PDF format on Canvas at the beginning of lecture (3:35pm) on Wednesday, April 8th, 2020. We recommend that you type your solutions with a text editor or word processor and then convert them to PDF. Please label your assignment with your name, UMN email address, and the time of your lab section (12:10 pm, 1:25 pm, or 2:30 pm).

Problem 1

Background on division: This problem uses the `idivq` instruction for division that has not been covered in lecture. You can read more about it in Section 3.5.5 of the textbook, and we’ll describe the basics here. Here is a brief description of how 64-bit signed division works in x86-64:

- The `idivq` instruction can actually support a 128-bit dividend, with the high 64 bits in %rdx and the low 64 bits in %rax.
- A 64-bit dividend should be sign-extended to make the right 128-bit dividend.
- Start with the 64-bit dividend in %rax.
- Sign-extend from %rax to %rdx using the `cqto` instruction.
- The divisor can be placed in any 64-bit register.
- `idivq` takes one register as an argument which is the register that has the divisor in it.
- After `idivq`, the %rax register holds the quotient, while the %rdx register holds the remainder.

Fill in the blanks of the assembly code generated from the following C function and explain what the function does, what the parameters and variables are, and what conclusions can be made based on the return value of the function. Assume 64-bit operations and the first argument register in the assembly code contains `long n` at the start of the program. In answer, use the letters a through i to label the values you fill in the blanks. For the short answer part j, be precise.

```
int function_c(long n){
    if(n<=1){
        return 0;
    }
    long i=2;
    while(i<=n/2){
        if(n%i==0)
            return 0;
        i++;
    }
    return 1;
}
```

```
function_asm:

if(n<=1){
    cmpq $1, %rdi
    ___ .E2 a.
}  movq %rdi, %rax  
long i=2;
    cqto
    movq $2, %rsi
    __ idivq %rsi
    ___ %rax, %r11 b.
    ___ .L1:
    i++;  cmpq %r11, %rsi
    ___ .E1 c.
    movq %rdi, %rax
    cqto
    idivq %rsi
    ___ $0, %rdx d.
    je .E2
    addq ____, %rsi e.
    jmp .L1
.
    .E1:
    movq ____, %rax f.
    ___ g.
.
    .E2:
    ___ $0, %rax h.
    ___ i.

j. What does this function do?
```
Problem 2

Consider the table below, which shows the initial contents of some registers and memory locations:

<table>
<thead>
<tr>
<th>Registers</th>
<th>Values</th>
<th>Memory</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>16</td>
<td>0x3FF0</td>
<td>10</td>
</tr>
<tr>
<td>rdx</td>
<td>32</td>
<td>0x3FF8</td>
<td>100</td>
</tr>
<tr>
<td>rcx</td>
<td>2</td>
<td>0x4000</td>
<td>210</td>
</tr>
<tr>
<td>rbx</td>
<td>24</td>
<td>0x4008</td>
<td>24</td>
</tr>
</tbody>
</table>

a. Fill in Table 1 showing the results if the following machine code is run from the initial state (these instructions do not change the memory):

```assembly
movq $1, %rax
subq $24, %rdx
addq %rcx, %rax
shlq $3, %rdx
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Values</th>
<th>Memory</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>0x3FF0</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>rdx</td>
<td>0x3FF8</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>rcx</td>
<td>0x4000</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>rbx</td>
<td>0x4008</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

b. Fill in Table 2 showing the results if instead the following machine code is run from the initial state at the top:

```assembly
leaq (%rbx, %rcx, 4), %rax
movq %rdx, 8(%rax)
subq $8, %rbx
subq $10, (%rbx)
subq $16, %rax
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Values</th>
<th>Memory</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>0x3FF0</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>rdx</td>
<td>0x3FF8</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>rcx</td>
<td>0x4000</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>rbx</td>
<td>0x4008</td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>
Problem 3

This is the assembly associated with the function long function_A(long n):

function_A:
    movq  $-1, %rax
    movq  $0, %rcx
    movq  $3, %r10
    cmpq  %rcx, %rdi
    jl    .L5
    movq  $1, %rax
    movq  $1, %rdx
    jmp   .L3

.L4:
    imulq %r10, %rax
    addq  $1, %rdx

.L3:
    cmpq  %rdi, %rdx
    jle   .L4

.L5:
    ret

A. Write C code that corresponds to the assembly given above. Give the variables meaningful names, not the names of registers, including giving a more informative name for the parameter currently named n.

B. Explain in a sentence or two what this function does.
Problem 4

(Based on the textbook problem 2.87.)

We’ve defined a new floating point standard, called UMN-20, which follows the basic rules of IEEE floating point, but contains 20 bits. This format has 1 sign bit, 6 exponent bits \( (k=6) \), and 13 fraction bits \( (n=13) \). The exponent bias is \( 2^{6-1} - 1 = 31 \).

A. Fill in the table that follows for each of the numbers given, with the following instructions for each column:

- **Hex**: the five hexadecimal digits describing the encoded form.
- **M**: the value of the significand. This should be a number of the form \( x \) or \( x/y \) where \( x \) is an integer and \( y \) is an integral power of 2. Examples include 1, 67/64, and 3/2.
- **E**: the integer value of the exponent.
- **V**: the numeric value represented. Use the notation \( \pm x \) or \( \pm x \times 2^z \), where \( x \) and \( z \) are integers.
- **D**: the (possibly approximate) decimal numeric value. Include at least 3 non-zero fraction digits.

You need not fill in entries marked –.

**Example**: to represent the number \( 3/4 \) we would have \( s=0, M=3/2, \) and \( E=-1 \). Our number would therefore have an exponent field of \( 011110_2 \) (decimal value of \( -1 + 31 = 30 \)) and a significand field of \( 1000000000000_2 \), giving a hex representation \( 3D000 \). The numeric value is 0.75.

<table>
<thead>
<tr>
<th>Description</th>
<th>Hex</th>
<th>M</th>
<th>E</th>
<th>V</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 3/4 )</td>
<td>3D000</td>
<td>3/2</td>
<td>-1</td>
<td>( 3 \times 2^{-2} )</td>
<td>0.75</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest value &lt; -2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest positive normalized</td>
<td>12340</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number with hex 12340</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. Floating point numbers in general, and in this case specifically the UMN-20 format, support addition and subtraction, but this operation is not necessarily associative. In real numbers, \( (2^{31} + 2^{31}) - 2^{31} = 2^{31} + (2^{31} - 2^{31}) \), but suppose we did the same operations with UMN-20 floating point. Fill in the Value parts of the following table with how those results would be represented in UMN-20, briefly describing how you get each result.

<table>
<thead>
<tr>
<th>Computation</th>
<th>Value</th>
<th>Computation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1 = 2^{31} + 2^{31} )</td>
<td></td>
<td>( R_1 = 2^{31} )</td>
<td></td>
</tr>
<tr>
<td>( L_2 = 2^{31} )</td>
<td></td>
<td>( R_2 = 2^{31} - 2^{31} )</td>
<td></td>
</tr>
<tr>
<td>( L = L_1 - L_2 )</td>
<td></td>
<td>( R = R_1 + R_2 )</td>
<td></td>
</tr>
</tbody>
</table>

Do \( L \) and \( R \) have the same value?
Problem 5

The assembly for the following function was produced with GCC.

```
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp
movq %rdi, -24(%rbp)
movl $0, -8(%rbp)
.L6:
cmpl $25, -8(%rbp)
jg .L7
movl $26, -4(%rbp)
.L5:
cmpl $25, -4(%rbp)
jle .L4
call rand
movl %eax, -4(%rbp)
andl $31, -4(%rbp)
jmp .L5
.L4:
movq -24(%rbp), %rdx
movl -4(%rbp), %ecx
movl -8(%rbp), %eax
movl %ecx, %esi
movl %eax, %edi
call swap
addl $1, -8(%rbp)
jmp .L6
.L7:
nop
leave
ret
```

Fill in the blanks for the C code, which was compiled to obtain this function. Assume that the function `rand` (part of the standard library) returns a random non-negative `int`. The function `swap(x, y, arr)` switches the position of two entries at indexes `x` and `y` in the array `arr`. You may find it helpful to make a table showing which stack locations are used to hold various local variables.

```c
void create_shuffle(char *table){
    for (int i = ___; i < ___; i++){
        int j = ___;
        while (j >= ___){
            j = ___;
            j = ___ & ___;
        }
        swap(___ , ___ , table);
    }
}
```