CMPU 224 Quiz 2 Practice Problems

Problem 1
Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x108</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x110</td>
<td>0x13</td>
</tr>
<tr>
<td>0x118</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

Fill in the following table showing the values for the indicated operands:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>$0x108</td>
<td></td>
</tr>
<tr>
<td>(%rax)</td>
<td></td>
</tr>
<tr>
<td>8(%rax)</td>
<td></td>
</tr>
<tr>
<td>0xD(%rax, %rdx)</td>
<td></td>
</tr>
<tr>
<td>260(%rcx, %rdx)</td>
<td></td>
</tr>
<tr>
<td>0xFC(,%rcx, 4)</td>
<td></td>
</tr>
<tr>
<td>(%rax, %rdx, 8)</td>
<td></td>
</tr>
</tbody>
</table>

Fill in the following table showing the effects of the following instructions in terms of both the register or memory location that will be updated and the resulting value:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Destination</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq %rcx, (%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subq %rdx, 8(%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imulq $16, (%rax, %rdx, 8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>incq 16(%rax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decq %rcx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subq %rdx, %rax</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 2

Draw the memory layout of the following struct (starting at memory location 0 below) and give the total size of the structure (in other words, what does the `sizeof(struct question1)` return?).

```c
struct question1 {
    char a;
    int b;
    int *c;
    char d;
};
```

Structs are laid out in memory in the order that they appear in the struct. Padding may be added to ensure each member of the struct is aligned on a multiple of its size. The entire struct may be padded out such that an adjacent struct will start on a multiple of the largest field. I'll show the padding below with an 'x'

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
```

`sizeof(struct question1)`  ________________
Problem 3
The following array int array[25][7] is stored at the address: 0x4000. What is the address of the integer at array[8][3]?
Problem 4

Part A Write the following function, `swapNybble()`, which takes an unsigned char as an input and returns an unsigned char that has the lower 4-bits swapped with the upper 4 bits of the input. For example, `swapNybble(0xAB)` would return `0xBA`.

```c
unsigned char swapNybble(unsigned char c) {
    // Returns an unsigned char with the upper 4 bits swapped with
    // the lower 4 bits.
}
```

Part B Shown below is the assembly code for a function `test()`, which calls `swapNybble(0xAB)`. Right before the call to `swapNybble()`, %rip, %rsp, and the stack have the values shown below.

```assembly
0000000000400594 <test>:  
  400594:   48 83 ec 08             sub    $0x8,%rsp
  400598:   bf ab 00 00 00          mov    $0xab,%edi
    ->  40059d:   e8 bb ff ff ff          callq  40055d <swapNybble>
  4005a2:   48 83 c4 08             add    $0x8,%rsp
  4005a6:   c3                      retq
```

Right before call to `swapNybble()`

<table>
<thead>
<tr>
<th>%rip</th>
<th>0x40059d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7fffffff8330</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffff340</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffff338</td>
<td>0x4005b0</td>
</tr>
<tr>
<td>0x7fffffff330</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffff328</td>
<td>0x0</td>
</tr>
</tbody>
</table>
Below, show the values of `%rip`, `%rsp`, and the stack right after the `callq` opcode jumps to the start of `swapNybble()` but before any of the code in `swapNybble()` is executed.

<table>
<thead>
<tr>
<th>%rip</th>
<th>%rsp</th>
</tr>
</thead>
</table>

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde340</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde338</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde330</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde328</td>
<td></td>
</tr>
</tbody>
</table>

Show the values of `%rip`, `%rsp`, and the stack right after the `callq` has finished but before the `add` instruction has executed.

After the return from `swapNybble()` (Fill in the blanks below).

```
400594: 48 83 ec 08                 ; sub $0x8,%rsp
400598: bf ab 00 00 00             ; mov $0xab,%edi
40059d: e8 bb ff ff ff            ; callq 40055d <swapNybble>
```

<table>
<thead>
<tr>
<th>%rip</th>
<th>%rsp</th>
</tr>
</thead>
</table>

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde340</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde338</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde330</td>
<td></td>
</tr>
<tr>
<td>0x7fffffffde328</td>
<td></td>
</tr>
</tbody>
</table>
Problem 5
The following C program and its assembly language version are shown below. Fill in the missing blanks in the assembly language output.

```c
void sumArray(long *a, long len, long *sum){
    /* Sum an array with length len and store the answer in sum.
     * arguments:
     *     a -- an array of long integers to sum
     *     len -- the length of the array
     *     sum -- a pointer that holds the sum of the array
     */

    long i;
    long answer = 0;
    for (i = 0; i < len; i++) {
        answer += a[i];
    }
    *sum = answer;
}
```

```
000000000040055d <sumArray>:
40055d:   b9 00 00 00 00          movq    $0x0,_________
400562:   b8 00 00 00 00          movq    $0x0,%rax
400567:   eb 08                   jmp     _______________
400569:   48 03 0c c7             addq   (_______________,%rax,8),%rcx
40056d:   48 83 c0 01             addq   $0x1,%rax
400571:   48 39 f0                cmpq   %rsi,%rax
400574:   7c f3                   j_______ 400569 <sumArray+0xc>
400576:   48 89 0a                movq   %rcx, ______________
400579:   c3                      retq
```
Problem 6

A C function `loopy` and the assembly code it compiles to on a 64-bit Linux machine is shown below:

```assembly
loopy:
    movl $0, %eax
    movl $0, %ecx
    jmp .L2
.L4:
    cmpq %rdx, (%rsi,%rcx,8)
    jle .L3
    addq $1, %rax
.L3:
    subq $1, %rdx
    addq $1, %rcx
.L2:
    cmpq %rdi, %rcx
    jl .L4
    ret
```

```c
long loopy(long n, long *a, long value) {
    long i;
    long x = _____________;
    for(i = ______________; ______________; i++) {
        if (______________________) {
            x = _________________;
        }
    }
    ______________;
    ______________;
    return x;
}
```

Based on the assembly code, fill in the blanks in the C source code.

Notes:
- You may only use the C variable names `n`, `a`, `i`, `value`, and `x`, not register names.
- Use array notation (e.g., `a[]`) to show accesses or updates to elements of array `a`.
Problem 7
A C function `func` and the x86-64 assembly code it compiles to on Linux machine is shown below:

```
.long func(long *a, long b) {
    int c = 0;
    int i = 0;
    while (a[i]__________________) {
        if (a[i]__________________) {
            c = ________________;
        }
        i = ________________;
    }
    return ______________;
}
```

```
.func:
    movq  $0, %rcx
    movq  $0, %rax
    jmp   .L2
.L4:
    cmpq  %rsi, %rdx
    jge   .L3
    addq  %rdx, %rax
.L3:
    addq  $1, %rcx
.L2:
    movq  %rcx, %rdx
    movq  (%rdi,%rdx,8), %rdx
    testq %rdx, %rdx
    jne   .L4
    ret
```

Part A: Based on the assembly code, fill in the blanks in the C source code. Note, the only lines in the C code are what are shown above.

Notes:
- You may only use the C variable names `a`, `b`, `c`, `i`, numbers, and C expressions in the blanks.
- Use array notation to show accesses or updates to elements of array `a`.

Part B: Describe in one short sentence what this function does. (This is a hint that if your C code doesn't do something that is easy to explain in English, you probably want to take another look at what you came up with).