CMPU 224 Quiz 2 Practice Problems

Problem 1

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

Address	Value
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

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

Operand	Value
%rax	
0x108	
$0x108	
(%rax)	
8(%rax)	
0xD(%rax, %rdx)	
260(%rcx, %rdx)	
0xFC(,%rcx, 4)	
(%rax, %rdx, 8)	

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:

Instruction	Destination	Value
addq %rcx, (%rax)		
subq %rdx, 8(%rax)		
imulq $16, (%rax, %rdx, 8)		
incq 16(%rax)		
decq %rcx		
subq %rdx, %rax		

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?).

```
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

`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`.

```
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.

```
  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()`

%rip	0x40059d
%rsp	0x7fffffffe330

Stack

Address	Value
0x7fffffffe340	0x0
0x7fffffffe338	0x4005b0
0x7fffffffe330	0x0
0x7fffffffe328	0x0

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.

%rip	
%rsp	

Stack

Address	Value
0x7fffffffe340	
0x7fffffffe338	
0x7fffffffe330	
0x7fffffffe328	

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).

```
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
```

%rip	
%rsp	

Stack

Address	Value
0x7fffffffe340	
0x7fffffffe338	
0x7fffffffe330	
0x7fffffffe328	

Problem 5

The following C program and its assembly language version are shown below. Fill in the missing blanks in the assembly language output.

```
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:

<pre>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</pre>
	<pre>long loopy(long n, long *a, long value) {
	
	  long i;
	
	  long x = _____________;
	
	  for(i = ______________; ______________; i++) {
	
	    if (______________________) {
	
	      x = _________________;
	
	    }
	
	    ____________________;
	
	  }
	
	  return x;
	
	}</pre>

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:

<pre>.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</pre>
	<pre>long func(long *a, long b) {
	
	  int c = 0;
	  int i = 0;
	
	  while (a[i]__________________) {
	
	    if (a[i]_______________) {
	
	      c = _________________;
	
	    }
	
	    i = ________________;
	
	  }
	
	  return ______________;
	
	}</pre>

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).