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	**0x100**
0x108	**0xAB**
$0x108	**0x108**
(%rax)	**0xFF**
8(%rax)	**0xAB**
0xD(%rax, %rdx)	**0x13**
260(%rcx, %rdx)	**0xAB**
0xFC(,%rcx, 4)	**0xFF**
(%rax, %rdx, 8)	**0x11**

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)	**0x100**	**0x100**
subq %rdx, 8(%rax)	**0x108**	**0xA8**
imulq $16, (%rax, %rdx, 8)	**0x118**	**0x110**
incq 16(%rax)	**0x110**	**0x14**
decq %rcx	**%rcx**	**0x0**
subq %rdx, %rax	**%rax**	**0xFD**

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'

a	**x**	**x**	**x**	**b**	**b**	**b**	**b**	**c**	**c**	**c**	**c**	**c**	**c**	**c**	**c**	**d**	**x**	**x**	**x**	**x**	**x**	**x**	**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)` ___**24**_____

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]`?

Arrays are laid out in row-major order. To find the address of row 8, we need to find the size of each row. Each row has 7 columns and an int is 4 bytes, so each row is 7 * 4 = 28 bytes long. The offset of row 8 from the start of the array is 8 * 28 = 224 bytes. We want the third column at row 8, which is 3 * 12 = 12 bytes from the start of row 8, so array[8][3] is 224 + 12 = 236 bytes from the start of the array. To get the address we just need to add 0x4000 to 236 (decimal) so we first convert 236 to hex. 236 / 16 = 14 with a remainder of 12, so 236 decimal is 0xEC.

So the answer is 0x4000 + 0xEC = 0x40EC.

In general:

If `A[i][j]` is element of type T, which requires K bytes
Address of `A[i][j] = A + (i * C + j)* K`

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.

   unsigned char lower = c >> 4 & 0xf
   unsigned char upper = c << 4
   return upper | lower
}
```

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`	**`0x40055d`**
`%rsp`	**`0x7fffffffe328`**

Stack

Address	Value
`0x7fffffffe340`	**`0x0`**
`0x7fffffffe338`	**`0x4005b0`**
`0x7fffffffe330`	**`0x0`**
`0x7fffffffe328`	**`0x4005a2`**

`%rip` always holds the address of the next instruction to be executed. In this case it is the address of the beginning of the `swapNybble` function.

The semantics of the `call` statement is to push the return address onto the stack and jump to the address in the `call` statement. This will decrement the value of `%rsp` by 8.

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	**0x4005a2**
%rsp	**0x7fffffffe330**

Stack

Address	Value
0x7fffffffe340	**0x0**
0x7fffffffe338	**0x4005b0**
0x7fffffffe330	**0x0**
0x7fffffffe328	**0x4005a2**

The call to `swapNybble` must restore `%rsp` to the its value right before the call, so `%rsp` will be the same as before the function call. `%rip` always points to the next instruction to execute, in this case the `add` opcode. When we return from a function, `%rsp` is restored, but none of the memory contents are erased or changed on the stack immediately. They are left to be overwritten when a new function call sets up its stack frame, so the value of the stack is the same as in the previous diagram at this point in the code.

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,__%rcx_____

  400562:   b8 00 00 00 00          movq    $0x0,%rax

  400567:   eb 08                   jmp     __0x400571______

  400569:   48 03 0c c7             addq   (__%rdi____,%rax,8),%rcx

  40056d:   48 83 c0 01             addq   $0x1,%rax

  400571:   48 39 f0                cmpq   %rsi,%rax

  400574:   7c f3                   jl____ 400569 <sumArray+0xc>

  400576:   48 89 0a                movq   %rcx, __(%rdx)_________

  400579:   c3                      retq
```

To get the value for the first blank, note that there are two variables that are initialized to 0, `answer` and `i`. `%rax` is one of the variables (we know that from the second line of assembly code), let's look at where `%rax` is used in the rest of the code. Note that this function is `void` (it doesn't return anything) so `%rax` can be used to store the value of a local function variable. If we look at the following line: `addq $0x1,%rax` and compare it to the C code, we see the only time 1 is added to anything is `i++`, therefore `%rax` must hold the variable `i`. So the first line of assembly code must therefore be moving 0 into `answer`. To find out in which register `answer` is stored, let's look at where `answer` is used in the C code. The other place `answer` is used is in the line `answer += a[i]` and the only other `add` in the assembly is `addq (_____,%rax,8),%rcx`, so, `answer` must be held in `%rcx`.

To determine the address for the `jmp`, we need to look at the C code. The for loop initializes i = 0 and before the body of the loop is executed, the test `i < len` is performed. We can see the test in the assembly as `cmpq %rsi,%rax` (`%rax` is `i` and `%rsi` is the second argument to the function, `len`). So, we should jump to this line of assembly, which is at address `0x400571`. While we are looking at the test, we can see that the compare is `i:len`, so the next line must be `jl`, to match the test `i < len`.

The line `addq (____,%rax,8),%rcx`, corresponds to the line `answer += a[i]`. `%rax` is the index to the array with the scaling index of 8, the blank line must be the starting address of the array which is held in `%rdi` (the first argument of the function).

The last blank line `movq %rcx, _____`, corresponds to the line `*sum = answer`. sum is a pointer and is stored in `%rdx` (we know this because `sum` is the third argument to the function). We know from the above that `%rcx` is the `variable` answer. The line of C code above says to take the value in `answer` and store it in the memory location contained in the pointer `sum`. That is a memory reference, so the value `%rcx` is moved to `(%rdx)`.

Problem 6

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

```
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
```

```
long loopy(long n, long *a, long value) {

  long i;

  long x = 0;

  for(i = 0; i < n; i++) {

    if (a[i] > value) {

      x = x + 1;

    }

    value -= 1;

  }

  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:

```
.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
```

```
long func(long *a, long b) {

  int c = 0;
  int i = 0;

  while (a[i] != 0) {

    if (a[i] < b) {

      c = c + a[i];

    }

    i  = i + 1;

  }

  return c;

}
```

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).

It returns the sum of integers in an array that are less than the value b.