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>0x100</td>
</tr>
<tr>
<td>0x108</td>
<td>0xAB</td>
</tr>
<tr>
<td>$0x108</td>
<td>0x108</td>
</tr>
<tr>
<td>(%rax)</td>
<td>0xFF</td>
</tr>
<tr>
<td>8(%rax)</td>
<td>0xAB</td>
</tr>
<tr>
<td>0xD(%rax, %rdx)</td>
<td>0x13</td>
</tr>
<tr>
<td>260(%rcx, %rdx)</td>
<td>0xAB</td>
</tr>
<tr>
<td>0xFC(:,%rcx, 4)</td>
<td>0xFF</td>
</tr>
<tr>
<td>(%rax, %rdx, 8)</td>
<td>0x11</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>0x100</td>
<td>0x100</td>
</tr>
<tr>
<td>subq %rdx, 8(%rax)</td>
<td>0x108</td>
<td>0xA8</td>
</tr>
<tr>
<td>imulq $16, (%rax, %rdx, 8)</td>
<td>0x118</td>
<td>0x110</td>
</tr>
<tr>
<td>incq 16(%rax)</td>
<td>0x110</td>
<td>0x14</td>
</tr>
<tr>
<td>decq %rcx</td>
<td>%rcx</td>
<td>0x0</td>
</tr>
<tr>
<td>subq %rdx, %rax</td>
<td>%rax</td>
<td>0xFD</td>
</tr>
</tbody>
</table>
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 question2)` return?).

```c
struct question2 {
    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 0 1 2 3 1 1 1 1 1 1 1 7 1 1 1 0 1 2 3 4 2 5 6 7 8
```

`sizeof(struct question1)` ___24_____________

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, so the we first want to get the address of the row. There are 7 ints in a row, 4 bytes per row, and we want the address of row 8. \((8 \times 7 \times 4) = 224\) bytes. We want the address of the third element in the row, so the offset from the start of the array is 224 + \((3 \times 4) = 236\) decimal. Converting to hexadecimal, we get 236 decimal is 0xEC hexadecimal so the address of that element is 0x4000 + 0xEC = 0x40EC.

4. Suppose `%rax` has the value of 42, `%rcx` has the value 5, indicate the value that will be stored in register `%rdx` for each of the instructions:

```
leaq -5(%rax), %rdx          37
leaq 0x45, %rdx               69
leaq (%rax, %rcx), %rdx       47
leaq (%rax, %rcx, 2), %rdx    52
leaq 0x21(%rcx, %rcx, 1), %rdx 43
```
5. **Part A** Write the following function, `swapNybble()`, which takes an unsigned char as 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.

  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()`

<table>
<thead>
<tr>
<th>%rip</th>
<th>0x40059d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7fffffffbe330</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffbe340</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffbe338</td>
<td>0x4005b0</td>
</tr>
<tr>
<td>0x7fffffffbe330</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffbe328</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>0x40055d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7fffffffde328</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde340</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffde338</td>
<td>0x4005b0</td>
</tr>
<tr>
<td>0x7fffffffde330</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffde328</td>
<td>0x4005a2</td>
</tr>
</tbody>
</table>

 `%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 on to 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 return from `swapNybble()` (Fill in the blanks below).

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

<table>
<thead>
<tr>
<th>%rip</th>
<th>0x4005a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsp</td>
<td>0x7fffffffde330</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffffde340</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffde338</td>
<td>0x4005b0</td>
</tr>
<tr>
<td>0x7fffffffde330</td>
<td>0x0</td>
</tr>
<tr>
<td>0x7fffffffde328</td>
<td>0x4005a2</td>
</tr>
</tbody>
</table>
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.
6. 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;
}
```

```assembly
000000000040055d <sumArray>:
40055d:   b9 00 00 00 00          movq    $0x0,___
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                   j
        # l 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), so let's look at where `%rax` is used in the rest of the code. We see, `addq  $0x1,%rax`, and note the only time 1 is added to anything in the C code is `i++`, therefore `%rax` must hold the variable `i`. Therefore, the first line of assembly code must therefore be moving 0 into `answer`. To find out what register `answer` is stored in, 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 code 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 code *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).
7. A C function `loopy` and the assembly code it compiles to on a 64-bit Linux machine is shown below:

<table>
<thead>
<tr>
<th>assembly code</th>
<th>C source code</th>
</tr>
</thead>
</table>
| movl $0, %eax  
movl $0, %ecx  
jmp .L2 |
| movl $0, %ecx  
.L4:  
cmpq %rdx, (%rsi,%rcx,8)  
jie .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 to show accesses or updates to elements of array `a.`
8. A C function `func` and the x86-64 assembly code it compiles to on Linux machine is shown below:

```assembly
.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 a sums of integers in an array that are less than the value `b`.

Practice Problems in the book

The practice problems in sections 3.4 to 3.10 are also good ways to prepare for the quiz. I often draw inspiration from these problems for quiz questions, so it is likely you will see similar problems to these on the quiz.