Name: ________________________ CSCI 237

Practice Problems: x86_64 ASM

1.) In a post-penny world, many stores will want to round their prices to the nearest nickel. The C function below (left) uses a switch statement to do just that. On the right is the (incomplete) assembly generated by compiling with flags `-Og -no-pie -fno-PIC`.

```c
long round_to_nickel(long cents) {
  switch (cents) {
    case 2: // round down by 2
    case 7: // to either 0 or 5
      cents -= 1;

    case 1: // round down by 1
    case 6: // to either 0 or 5
      cents -= 1;
      break;

    case 3: // round up by 2
    case 8: // to either 5 or 10
      cents += 1;

    case 4: // round up by 1
    case 9: // to either 5 or 10
      cents += 1;

    default: // no rounding needed
      break;
  }
  return cents;
}
```

```
round_to_nickel:
      cmpq  $9, %rdi
      ja    .L8
      jmp   *.L4(,%rdi,8)
.L4:
      .quad .L8
      .quad .L7
      .quad .L6
      .quad .L5
      .quad .L3
      .quad .L8
      .quad .L7
      .quad .L6
      .quad .L5
      .quad .L3
.L5:
      addq  $1, %rdi
.L3:
      leaq  1(%rdi), %rax
      ret
.L8:
      movq  %rdi, %rax
      ret
.L6:
      subq  $1, %rdi
.L7:
      leaq  -1(%rdi), %rax
      ret
```

Fill in the missing labels in the jump table to so that the assembly implementation is correct.

Is `jmp *.L4(,%rdi,8)` a direct or indirect jump?

It is indirect. The target address is in the jump table, and the * tells us to evaluate the address computation "`.L4(,%rdi,8)`" and then follow the address stored at that location.

2.) When the recursive C code on the left (the factorial function) is compiled with flags **-Og**, the (incomplete) assembly code on the right is generated by **gcc**:

```
long fact(long n) {
    if (n <= 1)
        return 1;

    return n * fact(n - 1);
}
```

```
fact:
     cmpq $1, %rdi
     jle  .L3
     pushq     %rbx
     movq %rdi, %rbx
     leaq -1(%rdi), %rdi
     call fact
     imulq     %rbx, %rax
     popq %rbx
     ret
.L3:
     movq $1, %rax
     ret
```

a.) What is the purpose of the instruction **pushq %rbx**?

Since %rbx is used to store the current value of n within the active function frame, push %rbx saves the old value of %rbx so it isn't overwritten.

b.) Complete the function by filling in the missing instruction.

c.) Within gdb, the command **disass fact** *produced the following output:*

```
0x401106 <fact+0>:     endbr64
0x40110a <fact+4>:     cmp    $0x1,%rdi
0x40110e <fact+8>:     jle    0x401123 <fact+29>
0x401110 <fact+10>:    push   %rbx
0x401111 <fact+11>:    mov    %rdi,%rbx
0x401114 <fact+14>:    lea    -0x1(%rdi),%rdi
0x401118 <fact+18>:    call   0x401106 <fact>
0x40111d <fact+23>:    imul   %rbx,%rax
0x401121 <fact+27>:    pop    %rbx
0x401122 <fact+28>:    ret
0x401123 <fact+29>:    mov    $0x1,%eax
0x401128 <fact+34>:    ret
```

Notes: using the standard C calling conventions, the first function argument is always stored in register %rdi, the second function argument in register %rsi, and the return value in register %rax.

Consider the function call **`factorial(4)`**.
Address `0x401123 <fact+29>` corresponds to the base case (if n <= 1). Fill in the register values and the stack diagram below with the appropriate values when the execution of factorial is about to return from the base case (as if gdb stopped at the breakpoint established using **`break *0x401128`**). Designate any values that are not known using a "?"

0x7fffffffe578	0x40113b	# ret address to caller
0x7fffffffe570	0	# initial value of %rbx
0x7fffffffe568	0x40111d	# ret addr for fact(3)
0x7fffffffe560	4	# pushed %rbx for fact(3)
0x7fffffffe558	0x40111d	# ret addr for fact(2)
0x7fffffffe550	3	# pushed %rbx for fact(2)
0x7fffffffe548	0x40111d	# ret addr for fact(1)
0x7fffffffe540	?	
0x7fffffffe538	?	
0x7fffffffe530	?	
0x7fffffffe528	?	

Registers:

%rdi	1
%rbx	2
%rax	1
%rsp	0x7fffffffe548

What instruction will be executed next?

`imul %rbx, %rax`

Notes: using the standard C calling conventions, the first function argument is always stored in register `%rdi`, the second function argument in register `%rsi`, and the return value in register `%rax`.

3.) Arrays: Consider the following C function that updates an array's contents:

```
void function(long *array, long i, long j)
{
   array[i+j] = array[i] + array[4] + 3;
}
```

The compiler has generated the following incomplete version of code for this function:

```
movq $3, %rcx                          # tmp = 3
addq ____32(%rdi)______________, %rcx  # tmp += array[4]
leaq ____(%rdi, %rsi, 8)_____, %r8     # %r8 = &array[i]
addq (%r8), %rcx                       # tmp += array[i]
addq %rsi, %rdx                        # calculate i+j
movq %rcx, (%rdi, %rdx, 8)______       # array[i+j]= tmp
ret
```

Fill in the incomplete portions indicated by empty lines to complete the code.

Notes: using the standard C calling conventions, the first function argument is always stored in register `%rdi`, the second function argument in register `%rsi`, and the return value in register `%rax`.