Name: _____ CSCI 237

Practice Problems: x86_64 ASM

1.) Conditionals: Consider the following C function returns the larger of its two char arguments:

```
char return_larger(char x, char y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

Using the `-Og` flag, the compiler generates the following assembly code:

```
    movl   %esi, %eax   # ret = ______
    cmpb   %sil, %dil   # calc __ - __, set cond codes
    jle    .L2          # jump if __________
    movl   %edi, %eax   # ret = ______
.L2:
    ret
```

a) Fill in the blanks using variables x, y, and appropriate logical operators to complete the comments associated with each line.

b) In the table below value, fill in the value of each condition code after executing the operation `cmpb %sil, %dil` for the given values of `x` and `y`:

x	***y***	***ZF***	***SF***	***OF***
2	*3*			
-115	*100*			
7	*7*			
3	*2*			
115	*-100*			

c) What expression involving `ZF`, `SF`, and `OF` evaluates to `1` when the jump is taken and `0` otherwise? In other words, under what conditions does `jle` trigger a jump to the target?

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`.

The function `return_larger2` implements the same logic as `return_larger`, but it uses a ternary operator.

```
char return_larger2(char x, char y)
{
   return x > y ? x : y;
}
```

Compiling this function using the -Og flag produces the following (incomplete) assembly:

```
        cmpb    %dil, %sil
        movl    %edi, %eax
        ______  %esi, %eax
        ret
```

d) Complete the assembly by filling in the missing instruction, and explain the logic of the completed assembly's implementation in words.

2.) ASM to C: We have provided assembly code (left) that corresponds to the partially complete C function `foo` (right). Please fill in the missing lines of the C code.

```
    cmpq    %rsi, %rdi
    je    .L4
    movq    %rdi, %rax
    andq    %rsi, %rax
    ret
.L4:
    movq    %rdi, %rax
    imulq   %rsi, %rax
    ret
```

```
long foo (long a, long b)
{
  long result;
  if (_____________) {

    _________________
  } else {

    _________________
  }
  return result;
}
```

3.) Loops: Consider the incomplete implementation of the C function `bar` on the left. Compiling the completed function with command-line option `-Og` produces the assembly code on the right.
First, use the box at the bottom to convert `bar` to an intermediate version (`bar_goto`) that uses `goto` statements. Then, fill in the missing lines of the C code for function `bar`.

```
long bar(long a, long b) {
  long result = _____;

  while (_______________) {

    ___________________;

    ___________________;
  }
  return result;
}
```

```
        movl    $1, %eax
        jmp     .L2
.L3:
        imulq   %rdi, %rax
        subq    $1, %rsi
.L2:
        testq   %rsi, %rsi
        jg      .L3
        ret
```

```
long bar_goto(long a, long b) {
  long result = __________;

  goto  ________;

loop:
    ________________________;

    ________________________;
test:

  if (_____________________)

        goto _____________;
  return result;
}
```

a) What does the function `bar` do, in words?

b) The following assembly code performs the same task as `bar`, but it is derived from C code that uses different control structures. Finish the C implementation of `bar2`. As above, you may wish to convert to a `goto` version as an intermediate step.

```
        movl    $0, %eax
        movl    $1, %edx
        jmp     .L2
.L3:
        imulq   %rdi, %rdx
        addl    $1, %eax
.L2:
        movslq  %eax, %rcx
        cmpq    %rsi, %rcx
        jl      .L3
        movq    %rdx, %rax
        ret
```

```
long bar2 (long a, long b)
{
  long result = _____;

    ____________________

    ____________________

    ____________________

  return result;
}
```

Register reference info:

8-byte	4-byte	2-byte	1-byte
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r8d	%r9w	%r9b
%r10	%r8d	%r10w	%r10b
%r11	%r8d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` are used to pass the first six integer or pointer parameters to called functions. Additional parameters (or large parameters such as structs passed by value) are passed on the stack.