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

```c
void function(long *array, long i, long j)
{
}
```

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

```assembly
movl $10, %rcx # tmp = 10
subq ____________ , %rcx # tmp -= array[4]
lea _____________, %r8 # %r8 = &array[j]
addq (%r8), %rcx # tmp += array[j]
subq $2, %rsi # calculate i-2
movq %rcx, _____________ # array[i-2]= tmp
ret
```

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

*Notes: using the standard C calling conventions, the first argument is always stored in register %rdi, the second in %rsi, and third in %rdx. On this system, a long is 8 bytes.*
1. Consider the assembly code (left) that corresponds to the partially complete C function `foo` (right). Please fill in the missing lines of the C code.

```
foo:
cmpq    %rsi, %rdi
jge    .L4
movq    %rdi, %rax
addq    %rsi, %rax
ret
.L4:
movq    %rdi, %rax
addq    %rdi, %rax
ret
```

```
long foo (long a, long b)
{
    long result;
    if ( !((a-b)>=0) ) {
        __result = a + b;__
    } else {
        __result = a+a;_____ 
    }
    return result;
}
```

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`. -0.5 if you get the negation wrong
3.) Consider the assembly code (left) that corresponds to the partially complete C function `foo` (right). Please fill in the missing lines of the C code.

<table>
<thead>
<tr>
<th>foo:</th>
<th>long foo (long a, long b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $0, %rax</td>
<td>long result = 0;</td>
</tr>
<tr>
<td>movl $0, %r8</td>
<td>for(long i=0; i&lt;b; i+=a)</td>
</tr>
<tr>
<td>jmp .L2</td>
<td>result += i*2;</td>
</tr>
<tr>
<td>.L1:movq %r8, %r9</td>
<td>}</td>
</tr>
<tr>
<td>salq $1, %r9</td>
<td>return result;</td>
</tr>
<tr>
<td>addq %r9, %rax</td>
<td>}</td>
</tr>
<tr>
<td>addq %rdi, %r8</td>
<td></td>
</tr>
<tr>
<td>.L2:</td>
<td></td>
</tr>
<tr>
<td>cmpq %r8, %rsi</td>
<td></td>
</tr>
<tr>
<td>jg .L1</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

*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.*
4.) Consider the complete assembly code (left) that corresponds to the complete C function `foo` (right). Please fill in the missing lines of the assembly code.

```assembly
foo:
    movl $0, %rax
    salq $1, %rdi
    movl $7, %esi
    callq
    ret
```

```c
long bar(long num1, long num2);

long foo (long a) {
    long result = 0;
    result = bar(a*2, 7);
    return result;
}
```

*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`.  

2. Describe 2 scenarios when it would not be a good idea to turn an if/else code block written in C into assembly code that uses a conditional move instruction instead of conditional jump instructions. For each, explain why it is not a good idea.

1.) Each block of code performs lots of computation. Since you execute both blocks of code, it would do a lot of extra work.

2.) If one block of code would be unsafe to execute if condition is not met, because program might crash.

3.) If blocks of code have side-effects (update variables) executing both blocks may result in erroneous updates of memory.