Answers:

Q1:
a. We're performing 64 – 64. The result, stored in %ecx, is = 0. Hence, the ZF flag is set to 1.
When the operands are interpreted as signed integers (64 and -64), the arithmetic operation does not overflow. Hence, the OF flag is set to 0.
When operands are interpreted as unsigned integers (64 and 4294967232), the arithmetic operation overflows. Hence, the CF flag is set to 1.
When the result is interpreted as a signed integer, the result is non-negative. Hence, the SF flag is set to 0.

b. This is 42 – 64. The result is non zero. Hence the ZF flag is set to 0.
When the operands are interpreted as signed integers (42 and -64), the arithmetic operation does not overflow. Hence the OF flag is set to 0.
When the operands are interpreted as unsigned integers (42 and 4294967232), the arithmetic operation does not overflow. Hence, the CF flag is set to 0.
When the result is interpreted as a signed integer, the result is negative. (We know this since the MSB of the result from the addl instruction is 1). Hence, the SF flag is set to 1.

c.
The result is nonzero, and so the ZF flag is set to 0.
When the operands are interpreted as signed integers, the arithmetic operation overflows. Hence the OF flag is set to 1.
When the operands are interpreted as unsigned integers, the arithmetic operation does not overflow. Hence, the CF flag is set to 0.
When the result is interpreted as a signed integer, it is negative. Hence the SF flag is set to 1.

Q2:

Conditional jump instruction	**Jump condition**
`je`	`ZF`
`jne`	`~ZF`
`js`	`SF`
`jns`	`~SF`
`jg`	`~(SF ^ OF) & ~ZF`
`jge`	`~(SF ^ OF)`
`jl`	`SF ^ OF`
`jle`	`(SF ^ OF) | ZF`
`ja`	`~CF & ~ZF`

`jae`	`~CF`
`jb`	`CF`
`jbe`	`CF | ZF`

je and jne just use ZF in their conditional jumps
js and jns just use SF
jg, jge, jl, and jle use both OF and SF, since using cmp may result in an overflow, which would mean the SF should be interpreted differently.
jg and jle also use the ZF.
ja, jae, jb, and jbe use CF, since using cmp will result in a carry if the subtraction results in what "would be" a negative number.
ja and jbe also use the ZF.

Note that the condition for jg is the negation of the condition for jle, since > is the negation of ≤. (And similarly with jl and jge, ja and jbe, and jae and jb.)

Q3.
The "jl .L7" causes the actual looping, which can be seen since you are going back up several instructions in the function.
The "`addl $1, %ecx`" instruction causes the "`cmpl %esi, %ecx`" to eventually allow us to break out of the loop. Note that the condition flags are *not* altered by the subsequent leal and movl instructions before the jl.

Q4.
The entire stack will be very large, containing ~50,000 stackframes for the fibonacci function. We'll see the consequences of the stack getting too large in lecture, but you may already be familiar with the infamous "segmentation fault." Hacker tip: do not write code like this during your Google interview!

Q5.
This particular implementation of fibonacci is tail-recursive. Once a particular call to fibonacci_helper is completed, nothing about that call has to be kept around on the stack, since it is just returning the value of a subsequent call to fibonacci. If the compiler is smart, it will just reuse the same stack frame for the 99,999th call to fibonacci_helper as it used for the first, by just replacing the original arguments that the function took with the updated values. So we generally *don't* have to worry about this version of fibonacci causing problems for large inputs.

Q6.
a. We can see that result must be in register `%edi`, since this value gets copied to `%eax` at the end of the function as the return value (line 13). We can see that `%esi` and `%ebx` get loaded with the values of `x` and `n` (lines 1 and 2), leaving `%edx` as the one holding variable mask (line 4.)

b. Register `%edi` (result) is initialized to −1 and `%edx` (mask) to 1.

c. The condition for continuing the loop (line 12) is that mask is nonzero.

d. The shift instruction on line 10 updates mask to be mask << n. Note %cl is the lower 8 bits of %ecx.

e. Lines 6–8 update result to be result ^ (x&mask).

f. Here is the original code:

```
1 int loop(int x, int n)
2 {
3     int result = -1;
4     int mask;
5     for (mask = 0x1; mask != 0; mask = mask << n) {
6         result ^= (x & mask);
7     }
8     return result;
9 }
```

Bonus answer:
What if the function doesn't *know* how many arguments it actually takes; e.g. a variable-argument function like printf? The compiled version of printf relies on the first argument to determine how arguments have been passed to it (e.g. if the first arg were "%d %d %s", it'd expect 3 additional arguments). If that format string were some unknown number of bytes away from %ebp, we'd never be able to figure out what are actually arguments to printf.

Q7:
a: Registers `%edi`, `%esi`, and `%ebx` are callee-saved registers. These registers must be saved on the stack by the callee and restored before returning, since the calling function expects them to be the same as when the callee was called.

b: `%eax`, `%ecx`, and `%edx` are caller-saved registers. This means the callee may use and overwrite these registers without destroying any data required by the caller.

c: `16(%ebp)` refers to the 3rd argument passed to this function. `24(%ebp)` refers to the 4th argument passed to this function. Lines 4 and 5 result in the multiplication of the 3rd and 4th arguments passed to this function. (We know that the first four arguments (there may be more) are all 4-bytes long, since we use "addl", "subl", etc. when dealing with them)

d. Subtracting values from %esp creates space between %ebp and %esp where we could store local variables or place the arguments to a function that we will call.

Q8:
a. We started with `%esp` = `0x800040`. Line 2 decrements it by 4, thereby resulting in `0x80003C` in the `%esp` register. Hence, this is the new value in `%ebp`.

b. We can see how the two `leal` instructions compute the arguments to pass to `scanf`. Since arguments are pushed in **reverse** order, we can see that `x` is at offset -4 relative to `%ebp` and y is at offset -8. The addresses are therefore `0x800038` and `0x800034`.

c. Starting with the original value of `0x800040`, line 2 decremented the stack pointed by 4. Line 4 decremented it by 24, and line 5 decremented it by 4. The three pushes decremented it by 12, resulting in an overall change of 44. Thus, after line 10 `%esp = 0x800014`.

d.

Memory Addresses	**Stack Contents**

0x80003C (%ebp)	0x800060
0x800038	0x53
0x800034	0x46
0x800030	
0x80002C	
0x800028	
0x800024	
0x800020	
0x80001C	0x800038
0x800018	0x800034
0x800014 (%esp)	0x300070

The call instruction will push “12” onto the stack before jumping to scanf, since that is the return address where we should start executing once scanf returns.

Note: byte addresses 0x800020 to 0x800033 are unused by `proc`. These “wasted” spaces are allocated to improve cache performance. We'll see how and why later this semester.