Problem 3.60 Solution:

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces students to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops.

A. We can see that `result` must be in register `%rax`, since this value gets returned as the final value. Parameter `x` is passed in `%rdi`. Parameter `n` is passed in `%esi` and then copied into `%ecx`. Register `%edx` is initialized to 1. We can infer that `mask` must be `%rdx`.

B. They are initialized to 0 and 1, respectively.

C. The condition for continuing the loop is that `mask` is nonzero.

D. The `salq` instruction updates `mask` to be `mask << n`.

E. Variable `result` is updated to be `result | (x&mask)`.

F. Here is the original code:

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

Problem 3.63 Solution:

This problem gives students practice analyzing disassembled code. The `switch` statement contains all the features one can imagine—cases with multiple labels, holes in the range of possible case values, and cases that fall through. The main trick is to use the jump table to identify the different entry points, and then analyze each block of code separately.

```
long switch_prob(long x, long n) {
    long result = x;
    switch(n) {
    case 60:
    case 62:
        result <<= 3;
        break;
    case 63:
        result >>= 3;
        break;
    case 64:
        result *= 15;
        /* Fall through */
    case 65:
        result *= result;
        /* Fall through */
    default:
        result += 75;
    }
    return result;
}
```