**3.60**

Consider the following assembly code:

*long loop (long x, int n)*

*x in %rdi, n in %esi*

1. loop:
2. movl %esi, %ecx
   1. *move %esi into %ecx – that is, the fourth argument (mask) = second argument(n)*
   2. *both of these are the lower 32 bits*
3. movl $1, %edx
   1. *move 1 to %edx – that is, the lower 32 bits of 3rd argument (result)*
4. movl $0, %eax
   1. *eax is the lower 32 bits of rax, which is the return value, which we set to zero*
5. jump .L2 *//go to .L2*
6. .L3
7. movq %rdi, %r8
   1. *// r8 = rdi 5th argument (mask) = 1st argument (x)*
8. andq %rdx, %r8
   1. *//compare rdx and r8. (rdx&r8e )store Set up a flag.*
   2. *this is third argument(result) and 5th argument (mask)*
9. orq %r8, %rax *//rax = rax|r8 (rax sets up a return value)*

salq %cl, %rdx *//leftshift rdx by %cl*

1. .L2
2. testq %rdx, %rdx
   1. *see if rdx – the third argument (result) is 0*
3. jne .L3 *//if rdx is not 0, then jump*
4. rep; ret

The preceding code was generated by compiling C code that had the following overall form:

long loop(long x, int n)

{

long result = 1;

long mask;

for (mask = n; mask!=x; mask << n &0xFF) {

result |=x;

}

return result;

}

long loop(long x, int n)

{

long result =1;

long mask;

for (mask = n; mask!=0; mask << n &0xFF) {

result |= (x&result);

}

return result;

}

long loop(long x, int n)

{

long result =0;

long mask;

for (mask = 1; mask!=0; mask = mask<< (n &0xFF)) {

result |=(x&mask);

}

return result;

}

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register %rax. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

1. Which registers hold program values x, n, result, and mask?
   1. %r8 and %rdi hold x, %esi and %ecx hold n, %rax holds result, %rdx holds mask
2. What are the initial values of result and mask?
   1. result = 0, mask = 1
3. What is the test condition for mask?
   1. testq %rdx, %rdx
4. How does mask get updated?
   1. At the end of every loop, mask gets left-shifted by the last byte of n.
5. How does result get updated?
   1. In the loop, result becomes equal to itself OR the result of x AND mask.

**3.63**

This problem will give you a chance'to reverge engineer a switch statement from

disassembled machine code. In the following procedure, the body of'the switch statement has been omitted:

long switch\_prob (long x, long n){

long result = x;

switch (n){

*//Fill in code here*

}

return result;

}

*long switch\_prob (long x, int n)*

*x in %rdi, n in %rsi*

1. 0000000000400590 <switch\_prob>;
2. 400590: 48 83 ee 3c sub $0x3c, %rsi
3. 400594: 48 83 fe 05 cmp $0x5, %rsi
4. 400598: 77 29 ja 4005c3 <switch\_prob+0x33>
5. 40059a: ff 24 f5 f8 06 40 00 jmpq \*0x4006f8(, %rsi, 8)
6. 4005a1: 48 8d 04 fd 00 00 00 lea 0x0(,%rdi,8),%rax
7. 4005a8: 00
8. 4005a9: c3 retq
9. 4005aa: 48 89 f8 mov %rdi,%rax
10. 4005ad: 48 89 f8 03 sar $0x3, %rax
11. 4005b1: c3 48 c1 e0 04 retq
12. 4005b2: 48 89 f8 mov %rdi, %rax
13. 4005b5: 48 c1 e0 04 shl $0x4, %rax
14. 4005b9: 48 29 f8 sub %rdi, %rax
15. 4005bc: 48 89 c7 mov %rax , %rdi
16. 4005bf: 48 0f af ff imul %rdi, %rdi
17. 4005c3: 48 8d 47 4b lea 0x4b(%rdi), %rax
18. 4005c7: c3 retq

Figure 3.53 shows the disassembled machine code for the procedure. The jump table resides in a different area of memory. We can see from the indirect jump on line 5 that the jump table begins at address 0x4006f8. Using the GDB debugger, we can examine the six 8-byte words of memory comprising the jump table with the command x/6gx 0x4006f8. GDB prints the following:

(gbd) *x/6gx 0x4006f8*

0x4006f8: 0x00000000004005a1 0x00000000004005c3

0x400708: 0x00000000004005a1 0x00000000004005aa

0x400718: 0x00000000004005b2 0x00000000004005bf

Fill in the body of the switch statement with C code that will have the same behavior as the machine code.