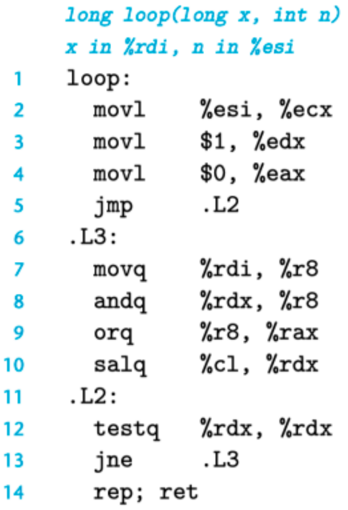
1. Consider the following assembly code:



```
long loop(long x, int n)
x in %rdi, n in %esi
loop:
  movl    %esi, %ecx
  movl    $1, %edx
  movl    $0, %eax
  jmp     .L2
.L3:
  movq    %rdi, %r8
  andq    %rdx, %r8
  orq     %r8, %rax
  salq    %cl, %rdx
.L2:
  testq   %rdx, %rdx
  jne     .L3
  rep; ret
```

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

```
long loop(long x, long n) {
        long result = ____
        long mask;
        for (mask =____; mask ____; mask ____) {

                result |=____;
        }
        return result;
}
```

a. Which registers hold program values **x**, **n**, **result**, and **mask**?

%rax = result
%rdi = x
%esi = n
%rdx = mask

b. What are the initial values of **result** and **mask**?

result = 0
mask = 1

c. What is the test condition for **mask**?

If it is not equal/not zero

d. How does **mask** get updated?

After each iteration of the loop the value of mask shifts to the left by 1

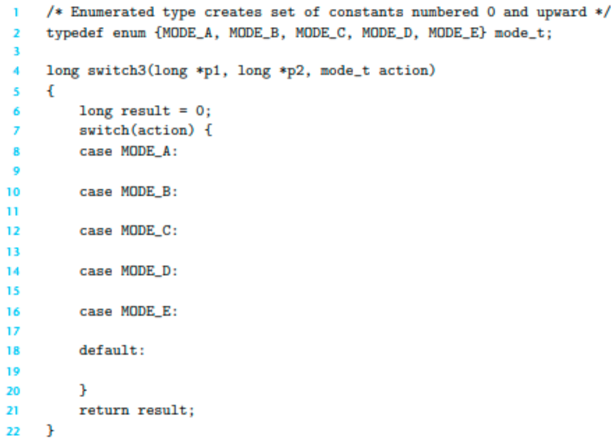
e. How does **result** get updated?

After each iteration of the loop result gets updated using the | function with the result of x & mask

f. Fill in all the missing parts of the C code.

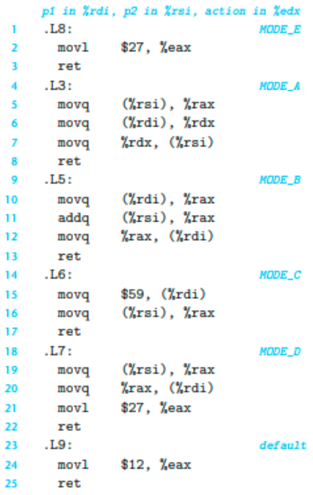
```
long loop(long x, long n) {
        long result = 0;
        long mask;
        for (mask = 1; mask <= n; mask <<=1) {
                result |= (x & mask);
        }
        return result;
}
```

2. The code that follows shows an example of branching on an enumerated type value in a switch statement. Recall that enumerated types in C are simply a way to introduce a set of names having associated integer values. By default, the values assigned to the names count from zero upward. In our code, the actions associated with the different case labels have been omitted.



```
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

long switch3(long *p1, long *p2, mode_t action)
{
    long result = 0;
    switch(action) {
    case MODE_A:

    case MODE_B:

    case MODE_C:

    case MODE_D:

    case MODE_E:

    default:

    }
    return result;
}
```

The part of the generated assembly code implementing the different actions is shown below. The annotations indicate the argument locations, the register values, and the case labels for the different jump destinations.
Fill in the missing parts of the C code. It contained one case that fell through to another—try to reconstruct this.



```
   p1 in %rdi, p2 in %rsi, action in %edx
   .L8:                                MODE_E
     movl    $27, %eax
     ret
   .L3:                                MODE_A
     movq    (%rsi), %rax
     movq    (%rdi), %rdx
     movq    %rdx, (%rsi)
     ret
   .L5:                                MODE_B
     movq    (%rdi), %rax
     addq    (%rsi), %rax
     movq    %rax, (%rdi)
     ret
   .L6:                                MODE_C
     movq    $59, (%rdi)
     movq    (%rsi), %rax
     ret
   .L7:                                MODE_D
     movq    (%rsi), %rax
     movq    %rax, (%rdi)
     movl    $27, %eax
     ret
   .L9:                                default
     movl    $12, %eax
     ret
```

```
long switch3(long *p1, long *p2, mode_t action){
        long result = 0;
        switch(action) {
                case MODE_A:
                        result = *p2;
                        action = *p1;
                        *p2 = action;
                        break;
                case MODE_B:
                        result = *p1 + *p2;
                        *p1 = result;
                        break;
                case MODE_C:
                        *p1 = 59;
                        result = *p2;
                        break;
        case MODE_D:
            result = *p2;
            *p1 = result;
            result = 27;
            break;
        case MODE_E:
            result = 27;
            break;
        default:
            result = 12;
            break;
    }
    return result;
}
```