```
store_prod:
  movq    %rdx, %rax
  cqto
  movq    %rsi, %rcx
  sarq    $63, %rcx
  imulq   %rax, %rcx
  imulq   %rsi, %rdx
  addq    %rdx, %rcx
  mulq    %rsi
  addq    %rcx, %rdx
  movq    %rax, (%rdi)
  movq    %rdx, 8(%rdi)
  ret
```

This code uses three multiplications for the multiprecision arithmetic required to implement 128-bit arithmetic on a 64-bit machine. Describe the algorithm used to compute the product, and annotate the assembly code to show how it realizes your algorithm. *Hint:* When extending arguments of x and y to 128 bits, they can be rewritten as $x = 2^{64} \cdot x_h + x_l$ and $y = 2^{64} \cdot y_h + y_l$, where x_h, x_l, y_h, and y_l are 64-bit values. Similarly, the 128-bit product can be written as $p = 2^{64} \cdot p_h + p_l$, where p_h and p_l are 64-bit values. Show how the code computes the values of p_h and p_l in terms of x_h, x_l, y_h, and y_l.

3.60 ◆◆

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;
}
```

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.

A. Which registers hold program values `x`, `n`, `result`, and `mask`?

B. What are the initial values of `result` and `mask`?

C. What is the test condition for `mask`?

D. How does `mask` get updated?

E. How does `result` get updated?

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

3.61 ◆◆

In Section 3.6.6, we examined the following code as a candidate for the use of conditional data transfer:

```
long cread(long *xp) {
    return (xp ? *xp : 0);
}
```

We showed a trial implementation using a conditional move instruction but argued that it was not valid, since it could attempt to read from a null address.

Write a C function `cread_alt` that has the same behavior as `cread`, except that it can be compiled to use conditional data transfer. When compiled, the generated code should use a conditional move instruction rather than one of the jump instructions.

3.62 ◆◆

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 in Figure 3.52. 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.

3.63 ◆◆

This problem will give you a chance to reverse 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;
}
```

```
   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
```

Figure 3.52 Assembly code for Problem 3.62. This code implements the different branches of a `switch` statement.

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:

```
(gdb) 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.

```
    long switch_prob(long x, long n)
    x in %rdi, n in %rsi
0000000000400590 <switch_prob>:
  400590:  48 83 ee 3c              sub    $0x3c,%rsi
  400594:  48 83 fe 05              cmp    $0x5,%rsi
  400598:  77 29                    ja     4005c3 <switch_prob+0x33>
  40059a:  ff 24 f5 f8 06 40 00     jmpq   *0x4006f8(,%rsi,8)
  4005a1:  48 8d 04 fd 00 00 00     lea    0x0(,%rdi,8),%rax
  4005a8:  00
  4005a9:  c3                       retq
  4005aa:  48 89 f8                 mov    %rdi,%rax
  4005ad:  48 c1 f8 03              sar    $0x3,%rax
  4005b1:  c3                       retq
  4005b2:  48 89 f8                 mov    %rdi,%rax
  4005b5:  48 c1 e0 04              shl    $0x4,%rax
  4005b9:  48 29 f8                 sub    %rdi,%rax
  4005bc:  48 89 c7                 mov    %rax,%rdi
  4005bf:  48 0f af ff              imul   %rdi,%rdi
  4005c3:  48 8d 47 4b              lea    0x4b(%rdi),%rax
  4005c7:  c3                       retq
```

Figure 3.53 Disassembled code for Problem 3.63.

3.64 ◆◆◆

Consider the following source code, where R, S, and T are constants declared with `#define`:

```
long A[R][S][T];

long store_ele(long i, long j, long k, long *dest)
{
    *dest = A[i][j][k];
    return sizeof(A);
}
```

In compiling this program, GCC generates the following assembly code:

```
long store_ele(long i, long j, long k, long *dest)
i in %rdi, j in %rsi, k in %rdx, dest in %rcx
store_ele:
  leaq    (%rsi,%rsi,2), %rax
  leaq    (%rsi,%rax,4), %rax
  movq    %rdi, %rsi
  salq    $6, %rsi
  addq    %rsi, %rdi
  addq    %rax, %rdi
```