As we have discussed, the ATT format used by GCC is very different from the Intel format used in Intel documentation and by other compilers (including the Microsoft compilers).

Muchnick's book on compiler design [80] is considered the most comprehensive reference on code-optimization techniques. It covers many of the techniques we discuss here, such as register usage conventions.

Much has been written about the use of buffer overflow to attack systems over the Internet. Detailed analyses of the 1988 Internet worm have been published by Spafford [105] as well as by members of the team at MIT who helped stop its spread [35]. Since then a number of papers and projects have generated ways both to create and to prevent buffer overflow attacks. Seacord's book [97] provides a wealth of information about buffer overflow and other attacks on code generated by C compilers.

Homework Problems

3.58 ◆

For a function with prototype

```
long decode2(long x, long y, long z);
```

GCC generates the following assembly code:

```
1   decode2:
2     subq    %rdx, %rsi
3     imulq   %rsi, %rdi
4     movq    %rsi, %rax
5     salq    $63, %rax
6     sarq    $63, %rax
7     xorq    %rdi, %rax
8     ret
```

Parameters `x`, `y`, and `z` are passed in registers `%rdi`, `%rsi`, and `%rdx`. The code stores the return value in register `%rax`.

Write C code for `decode2` that will have an effect equivalent to the assembly code shown.

3.59 ◆◆

The following code computes the 128-bit product of two 64-bit signed values x and y and stores the result in memory:

```
1   typedef __int128 int128_t;
2
3   void store_prod(int128_t *dest, int64_t x, int64_t y) {
4       *dest = x * (int128_t) y;
5   }
```

GCC generates the following assembly code implementing the computation:

```
1   store_prod:
2     movq    %rdx, %rax
3     cqto
4     movq    %rsi, %rcx
5     sarq    $63, %rcx
6     imulq   %rax, %rcx
7     imulq   %rsi, %rdx
8     addq    %rdx, %rcx
9     mulq    %rsi
10    addq    %rcx, %rdx
11    movq    %rax, (%rdi)
12    movq    %rdx, 8(%rdi)
13    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
1   loop:
2     movl    %esi, %ecx
3     movl    $1, %edx
4     movl    $0, %eax
5     jmp     .L2
6   .L3:
7     movq    %rdi, %r8
8     andq    %rdx, %r8
9     orq     %r8, %rax
10    salq    %cl, %rdx
11  .L2:
12    testq   %rdx, %rdx
13    jne     .L3
14    rep; ret
```

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

```
1   long loop(long x, long n)
2   {
3       long result = _________;
4       long mask;
5       for (mask = _________; mask  _________ ; mask = _________ ) {
6           result |= _________ ;
7       }
8       return result;
9   }
```

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.

```
1   /* Enumerated type creates set of constants numbered 0 and upward */
2   typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;
3
4   long switch3(long *p1, long *p2, mode_t action)
5   {
6       long result = 0;
7       switch(action) {
8       case MODE_A:
9
10      case MODE_B:
11
12      case MODE_C:
13
14      case MODE_D:
15
16      case MODE_E:
17
18      default:
19
20      }
21      return result;
22  }
```

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:

```
1   long switch_prob(long x, long n) {
2       long result = x;
3       switch(n) {
4           /* Fill in code here */
5
6       }
7       return result;
8   }
```

```
    p1 in %rdi, p2 in %rsi, action in %edx
1   .L8:                            MODE_E
2     movl    $27, %eax
3     ret
4   .L3:                            MODE_A
5     movq    (%rsi), %rax
6     movq    (%rdi), %rdx
7     movq    %rdx, (%rsi)
8     ret
9   .L5:                            MODE_B
10    movq    (%rdi), %rax
11    addq    (%rsi), %rax
12    movq    %rax, (%rdi)
13    ret
14  .L6:                            MODE_C
15    movq    $59, (%rdi)
16    movq    (%rsi), %rax
17    ret
18  .L7:                            MODE_D
19    movq    (%rsi), %rax
20    movq    %rax, (%rdi)
21    movl    $27, %eax
22    ret
23  .L9:                            default
24    movl    $12, %eax
25    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.

```
8       addq    %rdi, %rdx
9       movq    A(,%rdx,8), %rax
10      movq    %rax, (%rcx)
11      movl    $3640, %eax
12      ret
```

A. Extend Equation 3.1 from two dimensions to three to provide a formula for the location of array element `A[i][j][k]`.

B. Use your reverse engineering skills to determine the values of R, S, and T based on the assembly code.

3.65 ◆

The following code transposes the elements of an $M \times M$ array, where M is a constant defined by `#define`:

```
1   void transpose(long A[M][M]) {
2       long i, j;
3       for (i = 0; i < M; i++)
4           for (j = 0; j < i; j++) {
5               long t = A[i][j];
6               A[i][j] = A[j][i];
7               A[j][i] = t;
8           }
9   }
```

When compiled with optimization level `-O1`, GCC generates the following code for the inner loop of the function:

```
1   .L6:
2     movq    (%rdx), %rcx
3     movq    (%rax), %rsi
4     movq    %rsi, (%rdx)
5     movq    %rcx, (%rax)
6     addq    $8, %rdx
7     addq    $120, %rax
8     cmpq    %rdi, %rax
9     jne     .L6
```

We can see that GCC has converted the array indexing to pointer code.

A. Which register holds a pointer to array element `A[i][j]`?

B. Which register holds a pointer to array element `A[j][i]`?

C. What is the value of M?

3.66 ◆

Consider the following source code, where `NR` and `NC` are macro expressions declared with `#define` that compute the dimensions of array `A` in terms of parameter n. This code computes the sum of the elements of column j of the array.

```
10    movq    72(%rsp), %rax
11    addq    64(%rsp), %rax
12    addq    80(%rsp), %rax
13    addq    $104, %rsp
14    ret
```

A. We can see on line 2 of function `eval` that it allocates 104 bytes on the stack. Diagram the stack frame for `eval`, showing the values that it stores on the stack prior to calling `process`.

B. What value does `eval` pass in its call to `process`?

C. How does the code for `process` access the elements of structure argument `s`?

D. How does the code for `process` set the fields of result structure `r`?

E. Complete your diagram of the stack frame for `eval`, showing how `eval` accesses the elements of structure `r` following the return from `process`.

F. What general principles can you discern about how structure values are passed as function arguments and how they are returned as function results?

3.68 ◆◆◆

In the following code, *A* and *B* are constants defined with `#define`:

```
1   typedef struct {
2       int x[A][B]; /* Unknown constants A and B */
3       long y;
4   } str1;
5
6   typedef struct {
7       char array[B];
8       int t;
9       short s[A];
10      long u;
11  } str2;
12
13  void setVal(str1 *p, str2 *q) {
14      long v1 = q->t;
15      long v2 = q->u;
16      p->y = v1+v2;
17  }
```

Gcc generates the following code for `setVal`:

```
    void setVal(str1 *p, str2 *q)
    p in %rdi, q in %rsi
1   setVal:
2     movslq  8(%rsi), %rax
3     addq    32(%rsi), %rax
```

```
4       movq    %rax, 184(%rdi)
5       ret
```

What are the values of A and B? (The solution is unique.)

3.69 ◆◆◆

You are charged with maintaining a large C program, and you come across the following code:

```
1   typedef struct {
2       int first;
3       a_struct a[CNT];
4       int last;
5   } b_struct;
6
7   void test(long i, b_struct *bp)
8   {
9       int n = bp->first + bp->last;
10      a_struct *ap = &bp->a[i];
11      ap->x[ap->idx] = n;
12  }
```

The declarations of the compile-time constant `CNT` and the structure `a_struct` are in a file for which you do not have the necessary access privilege. Fortunately, you have a copy of the `.o` version of code, which you are able to disassemble with the OBJDUMP program, yielding the following disassembly:

```
    void test(long i, b_struct *bp)
    i in %rdi, bp in %rsi
1   0000000000000000 <test>:
2      0:   8b 8e 20 01 00 00        mov    0x120(%rsi),%ecx
3      6:   03 0e                    add    (%rsi),%ecx
4      8:   48 8d 04 bf              lea    (%rdi,%rdi,4),%rax
5      c:   48 8d 04 c6              lea    (%rsi,%rax,8),%rax
6     10:   48 8b 50 08              mov    0x8(%rax),%rdx
7     14:   48 63 c9                 movslq %ecx,%rcx
8     17:   48 89 4c d0 10           mov    %rcx,0x10(%rax,%rdx,8)
9     1c:   c3                       retq
```

Using your reverse engineering skills, deduce the following:

A. The value of `CNT`.

B. A complete declaration of structure `a_struct`. Assume that the only fields in this structure are `idx` and `x`, and that both of these contain signed values.