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:

```
decode2:
  subq    %rdx, %rsi
  imulq   %rsi, %rdi
  movq    %rsi, %rax
  salq    $63, %rax
  sarq    $63, %rax
  xorq    %rdi, %rax
  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:

```
typedef __int128 int128_t;

void store_prod(int128_t *dest, int64_t x, int64_t y) {
    *dest = x * (int128_t) y;
}
```

Gcc generates the following assembly code implementing the computation:

```
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
```

```
  addq    %rdi, %rdx
  movq    A(,%rdx,8), %rax
  movq    %rax, (%rcx)
  movl    $3640, %eax
  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`:

```
void transpose(long A[M][M]) {
    long i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < i; j++) {
            long t = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = t;
        }
}
```

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

```
.L6:
  movq    (%rdx), %rcx
  movq    (%rax), %rsi
  movq    %rsi, (%rdx)
  movq    %rcx, (%rax)
  addq    $8, %rdx
  addq    $120, %rax
  cmpq    %rdi, %rax
  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.

```
long sum_col(long n, long A[NR(n)][NC(n)], long j) {
    long i;
    long result = 0;
    for (i = 0; i < NR(n); i++)
        result += A[i][j];
    return result;
}
```

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

```
long sum_col(long n, long A[NR(n)][NC(n)], long j)
n in %rdi, A in %rsi, j in %rdx
sum_col:
  leaq    1(,%rdi,4), %r8
  leaq    (%rdi,%rdi,2), %rax
  movq    %rax, %rdi
  testq   %rax, %rax
  jle     .L4
  salq    $3, %r8
  leaq    (%rsi,%rdx,8), %rcx
  movl    $0, %eax
  movl    $0, %edx
.L3:
  addq    (%rcx), %rax
  addq    $1, %rdx
  addq    %r8, %rcx
  cmpq    %rdi, %rdx
  jne     .L3
  rep; ret
.L4:
  movl    $0, %eax
  ret
```

Use your reverse engineering skills to determine the definitions of NR and NC.

3.67 ◆◆

For this exercise, we will examine the code generated by GCC for functions that have structures as arguments and return values, and from this see how these language features are typically implemented.

The following C code has a function `process` having structures as argument and return values, and a function `eval` that calls `process`:

```
typedef struct {
    long a[2];
    long *p;
} strA;

```

```
typedef struct {
    long u[2];
    long q;
} strB;

strB process(strA s) {
    strB r;
    r.u[0] = s.a[1];
    r.u[1] = s.a[0];
    r.q =    *s.p;
    return r;
}

long eval(long x, long y, long z) {
    strA s;
    s.a[0] = x;
    s.a[1] = y;
    s.p = &z;
    strB r = process(s);
    return r.u[0] + r.u[1] + r.q;
}
```

GCC generates the following code for these two functions:

```
strB process(strA s)
process:
  movq    %rdi, %rax
  movq    24(%rsp), %rdx
  movq    (%rdx), %rdx
  movq    16(%rsp), %rcx
  movq    %rcx, (%rdi)
  movq    8(%rsp), %rcx
  movq    %rcx, 8(%rdi)
  movq    %rdx, 16(%rdi)
  ret
```

```
long eval(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
eval:
  subq    $104, %rsp
  movq    %rdx, 24(%rsp)
  leaq    24(%rsp), %rax
  movq    %rdi, (%rsp)
  movq    %rsi, 8(%rsp)
  movq    %rax, 16(%rsp)
  leaq    64(%rsp), %rdi
  call    process
```

```
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`:

```
typedef struct {
    int x[A][B]; /* Unknown constants A and B */
    long y;
} str1;

typedef struct {
    char array[B];
    int t;
    short s[A];
    long u;
} str2;

void setVal(str1 *p, str2 *q) {
    long v1 = q->t;
    long v2 = q->u;
    p->y = v1+v2;
}
```

Gcc generates the following code for `setVal`:

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

```
    movq    %rax, 184(%rdi)
    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:

```
typedef struct {
    int first;
    a_struct a[CNT];
    int last;
} b_struct;

void test(long i, b_struct *bp)
{
    int n = bp->first + bp->last;
    a_struct *ap = &bp->a[i];
    ap->x[ap->idx] = n;
}
```

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
0000000000000000 <test>:
   0:   8b 8e 20 01 00 00       mov    0x120(%rsi),%ecx
   6:   03 0e                   add    (%rsi),%ecx
   8:   48 8d 04 bf             lea    (%rdi,%rdi,4),%rax
   c:   48 8d 04 c6             lea    (%rsi,%rax,8),%rax
  10:   48 8b 50 08             mov    0x8(%rax),%rdx
  14:   48 63 c9                movslq %ecx,%rcx
  17:   48 89 4c d0 10          mov    %rcx,0x10(%rax,%rdx,8)
  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.

3.70 ◆◆◆

Consider the following union declaration:

```
union ele {
    struct {
        long *p;
        long y;
    } e1;
    struct {
        long x;
        union ele *next;
    } e2;
};
```

This declaration illustrates that structures can be embedded within unions.

The following function (with some expressions omitted) operates on a linked list having these unions as list elements:

```
void proc (union ele *up) {
    up->_________ = *(_________) - _________;
}
```

A. What are the offsets (in bytes) of the following fields:

```
e1.p       _______
e1.y       _______
e2.x       _______
e2.next    _______
```

B. How many total bytes does the structure require?

C. The compiler generates the following assembly code for `proc`:

```
void proc (union ele *up)
up in %rdi
proc:
  movq    8(%rdi), %rax
  movq    (%rax), %rdx
  movq    (%rdx), %rdx
  subq    8(%rax), %rdx
  movq    %rdx, (%rdi)
  ret
```

On the basis of this information, fill in the missing expressions in the code for `proc`. *Hint:* Some union references can have ambiguous interpretations. These ambiguities get resolved as you see where the references lead. There is only one answer that does not perform any casting and does not violate any type constraints.

3.71 ◆

Write a function `good_echo` that reads a line from standard input and writes it to standard output. Your implementation should work for an input line of arbitrary length. You may use the library function `fgets`, but you must make sure your function works correctly even when the input line requires more space than you have allocated for your buffer. Your code should also check for error conditions and return when one is encountered. Refer to the definitions of the standard I/O functions for documentation [45, 61].

3.72 ◆◆

Figure 3.54(a) shows the code for a function that is similar to function `vfunct` (Figure 3.43(a)). We used `vfunct` to illustrate the use of a frame pointer in managing variable-size stack frames. The new function `aframe` allocates space for local

(a) C code

```c
#include <alloca.h>

long aframe(long n, long idx, long *q)  {
    long i;
    long **p = alloca(n * sizeof(long *));
    p[0] = &i;
    for (i = 1; i < n; i++)
        p[i] = q;
    return *p[idx];
}
```

(b) Portions of generated assembly code

```
long aframe(long n, long idx, long *q)
n in %rdi, idx in %rsi, q in %rdx
aframe:
  pushq   %rbp
  movq    %rsp, %rbp
  subq    $16, %rsp              Allocate space for i (%rsp = s1)
  leaq    30(,%rdi,8), %rax
  andq    $-16, %rax
  subq    %rax, %rsp             Allocate space for array p (%rsp = s2)
  leaq    15(%rsp), %r8
  andq    $-16, %r8              Set %r8 to &p[0]
   .
   .
   .
```

Figure 3.54 Code for Problem 3.72. This function is similar to that of Figure 3.43.

array p by calling library function `alloca`. This function is similar to the more commonly used function `malloc`, except that it allocates space on the run-time stack. The space is automatically deallocated when the executing procedure returns.

Figure 3.54(b) shows the part of the assembly code that sets up the frame pointer and allocates space for local variables `i` and `p`. It is very similar to the corresponding code for `vframe`. Let us use the same notation as in Problem 3.49: The stack pointer is set to values s_1 at line 4 and s_2 at line 7. The start address of array `p` is set to value p at line 9. Extra space e_2 may arise between s_2 and p, and extra space e_1 may arise between the end of array `p` and s_1.

A. Explain, in mathematical terms, the logic in the computation of s_2.

B. Explain, in mathematical terms, the logic in the computation of p.

C. Find values of n and s_1 that lead to minimum and maximum values of e_1.

D. What alignment properties does this code guarantee for the values of s_2 and p?

3.73 ◆

Write a function in assembly code that matches the behavior of the function `find_range` in Figure 3.51. Your code should contain only one floating-point comparison instruction, and then it should use conditional branches to generate the correct result. Test your code on all 2^{32} possible argument values. Web Aside ASM:EASM on page 214 describes how to incorporate functions written in assembly code into C programs.

3.74 ◆◆

Write a function in assembly code that matches the behavior of the function `find_range` in Figure 3.51. Your code should contain only one floating-point comparison instruction, and then it should use conditional moves to generate the correct result. You might want to make use of the instruction `cmovp` (move if even parity). Test your code on all 2^{32} possible argument values. Web Aside ASM:EASM on page 214 describes how to incorporate functions written in assembly code into C programs.

3.75 ◆

ISO C99 includes extensions to support complex numbers. Any floating-point type can be modified with the keyword `complex`. Here are some sample functions that work with complex data and that call some of the associated library functions:

```
#include <complex.h>

double c_imag(double complex x) {
    return cimag(x);
}

double c_real(double complex x) {
    return creal(x);
}

```

```
double complex c_sub(double complex x, double complex y) {
    return x - y;
}
```

When compiled, GCC generates the following assembly code for these functions:

```
double c_imag(double complex x)
c_imag:
  movapd  %xmm1, %xmm0
  ret

double c_real(double complex x)
c_real:
  rep; ret

double complex c_sub(double complex x, double complex y)
c_sub:
  subsd   %xmm2, %xmm0
  subsd   %xmm3, %xmm1
  ret
```

Based on these examples, determine the following:

A. How are complex arguments passed to a function?

B. How are complex values returned from a function?

Solutions to Practice Problems

Solution to Problem 3.1 (page 218)

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
`%rax`	`0x100`	Register
`0x104`	`0xAB`	Absolute address
`$0x108`	`0x108`	Immediate
`(%rax)`	`0xFF`	Address `0x100`
`4(%rax)`	`0xAB`	Address `0x104`
`9(%rax,%rdx)`	`0x11`	Address `0x10C`
`260(%rcx,%rdx)`	`0x13`	Address `0x108`
`0xFC(,%rcx,4)`	`0xFF`	Address `0x100`
`(%rax,%rdx,4)`	`0x11`	Address `0x10C`

Solution to Problem 3.2 (page 221)

As we have seen, the assembly code generated by GCC includes suffixes on the instructions, while the disassembler does not. Being able to switch between these