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

\[
\text{long decode2(long } x, \text{ long } y, \text{ long } z)\; ;
\]

gcc generates the following assembly code:

\[
\begin{align*}
\text{decode2:} & \\
& \text{subq } \%rdx, \%rsi \\
& \text{imulq } \%rsi, \%rdi \\
& \text{movq } \%rsi, \%rax \\
& \text{salq } \%edx, \%rax \\
& \text{sarq } \%edx, \%rax \\
& \text{xorq } \%rdi, \%rax \\
& \text{ret}
\end{align*}
\]

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:

\[
\begin{align*}
\text{typedef } \_\_\text{int128 int128}_t; \\
\text{void store_prod(int128}_t \ast\text{dest, int64}_t \, x, \, \text{int64}_t \, y) \{ \\
& \ast\text{dest} = \text{dest} + \text{dest} \\
& \}
\end{align*}
\]

gcc generates the following assembly code implementing the computation:
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:

```assembly
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
.L3: 6  movq %rdi, %r8
  7  andq %r8, %r8
  8  orq %r8, %rax
  9  salq %cl, %rdx
.L2: 10  testq %rdx, %rdx
  11  jne .L3
  12  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.
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:

```c
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:

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

```c
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
0000000000000000 <test>:
2.  0: 8b 8e 20 01 00 00 mov  0x10(%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 00 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.
movq (%rsp), REG  \text{Read REG from stack}
addq $8, %rsp  \text{Increment stack pointer}

A. In light of analysis done in Practice Problem 4.8, does this code sequence correctly describe the behavior of the instruction popq %rsp? Explain.

B. How could you rewrite the code sequence so that it correctly describes both the cases where REG is %rsp as well as any other register?

4.47 \text{+++}
Your assignment will be to write a Y86-64 program to perform bubblesort. For reference, the following C function implements bubblesort using array referencing:

\begin{verbatim}
1 /* Bubble sort: Array version */
2 void bubble_a(long *data, long count) {
3     long i, last;
4     for (last = count-1; last > 0; last--) {
5         for (i = 0; i < last; i++) {
6             if (data[i+1] < data[i]) {
7                 /* Swap adjacent elements */
8                 long t = data[i+1];
9                 data[i+1] = data[i];
10                data[i] = t;
11               }
12         }
13     }
14 }
\end{verbatim}

A. Write and test a C version that references the array elements with pointers, rather than using array indexing.

B. Write and test a Y86-64 program consisting of the function and test code. You may find it useful to pattern your implementation after x86-64 code generated by compiling your C code. Although pointer comparisons are normally done using unsigned arithmetic, you can use signed arithmetic for this exercise.

4.48 \text{+++}
Modify the code you wrote for Problem 4.47 to implement the test and swap in the bubblesort function (lines 6–11) using no jumps and at most three conditional moves.

4.49 \text{+++}
Modify the code you wrote for Problem 4.47 to implement the test and swap in the bubblesort function (lines 6–11) using no jumps and just one conditional move.

4.50 \text{+++}
In Section 3.6.8, we saw that a common way to implement switch statements is to create a set of code blocks and then index those blocks using a jump table. Consider