CHAPTER 3

Machine-Level Representation of Programs

Computers execute *machine code*, sequences of bytes encoding the low-level operations that manipulate data, manage memory, read and write data on storage devices, and communicate over networks. A compiler generates machine code through a series of stages, based on the rules of the programming language, the instruction set of the target machine, and the conventions followed by the operating system. The GCC C compiler generates its output in the form of *assembly code*, a textual representation of the machine code giving the individual instructions in the program. GCC then invokes both an *assembler* and a *linker* to generate the executable machine code from the assembly code. In this chapter, we will take a close look at machine code and its human-readable representation as assembly code.

When programming in a high-level language such as C, and even more so in Java, we are shielded from the detailed, machine-level implementation of our program. In contrast, when writing programs in assembly code (as was done in the early days of computing) a programmer must specify the low-level instructions the program uses to carry out a computation. Most of the time, it is much more productive and reliable to work at the higher level of abstraction provided by a high-level language. The type checking provided by a compiler helps detect many program errors and makes sure we reference and manipulate data in consistent ways. With modern, optimizing compilers, the generated code is usually at least as efficient as what a skilled, assembly-language programmer would write by hand. Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.

So why should we spend our time learning machine code? Even though compilers do most of the work in generating assembly code, being able to read and understand it is an important skill for serious programmers. By invoking the compiler with appropriate command-line parameters, the compiler will generate a file showing its output in assembly-code form. By reading this code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter 5, programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the run-time behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package, as covered in Chapter 12, it is important to know what region of memory is used to hold the different program variables. This information is visible at the assembly-code level. As another example, many of the ways programs can be attacked, allowing worms and viruses to infest a system, involve nuances of the way programs store their run-time control information. Many attacks involve exploiting weaknesses in system programs to overwrite information and thereby take control of the system. Understanding how these vulnerabilities arise and how to guard against them requires a knowledge of the machine-level representation of programs. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by compilers.

In this chapter, we will learn the details of two particular assembly languages and see how C programs get compiled into these forms of machine code. Reading the assembly code generated by a compiler involves a different set of skills than writing assembly code by hand. We must understand the transformations typical compilers make in converting the constructs of C into machine code. Relative to the computations expressed in the C code, optimizing compilers can rearrange execution order, eliminate unneeded computations, replace slow operations with faster ones, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can often be a challenge—it's much like putting together a puzzle having a slightly different design than the picture on the box. It is a form of *reverse engineering*—trying to understand the process by which a system was created by studying the system and working backward. In this case, the system is a machine-generated assembly-language program, rather than something designed by a human. This simplifies the task of reverse engineering, because the generated code follows fairly regular patterns, and we can run experiments, having the compiler generate code for many different programs. In our presentation, we give many examples and provide a number of exercises illustrating different aspects of assembly language and compilers. This is a subject where mastering the details is a prerequisite to understanding the deeper and more fundamental concepts. Those who say "I understand the general principles, I don't want to bother learning the details" are deluding themselves. It is critical for you to spend time studying the examples, working through the exercises, and checking your solutions with those provided.

Our presentation is based on two related machine languages: Intel IA32, the dominant language of most computers today, and x86-64, its extension to run on 64-bit machines. Our focus starts with IA32. Intel processors have grown from primitive 16-bit processors in 1978 to the mainstream machines for today's desktop, laptop, and server computers. The architecture has grown correspondingly, with new features added and with the 16-bit architecture transformed to become IA32, supporting 32-bit data and addresses. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by GCC and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

Our technical presentation starts with a quick tour to show the relation between C, assembly code, and machine code. We then proceed to the details of IA32, starting with the representation and manipulation of data and the implementation of control. We see how control constructs in C, such as `if`, `while`, and `switch` statements, are implemented. We then cover the implementation of procedures, including how the program maintains a run-time stack to support the passing of data and control between procedures, as well as storage for local variables. Next, we consider how data structures such as arrays, structures, and unions are implemented at the machine level. With this background in machine-level programming, we can examine the problems of out of bounds memory references and the vulnerability of systems to buffer overflow attacks. We finish this part of the

presentation with some tips on using the GDB debugger for examining the run-time behavior of a machine-level program.

As we will discuss, the extension of IA32 to 64 bits, termed x86-64, was originally developed by Advanced Micro Devices (AMD), Intel's biggest competitor. Whereas a 32-bit machine can only make use of around 4 gigabytes (2^{32} bytes) of random-access memory, current 64-bit machines can use up to 256 terabytes (2^{48} bytes). The computer industry is currently in the midst of a transition from 32-bit to 64-bit machines. Most of the microprocessors in recent server and desktop machines, as well as in many laptops, support either 32-bit or 64-bit operation. However, most of the operating systems running on these machines support only 32-bit applications, and so the capabilities of the hardware are not fully utilized. As memory prices drop, and the desire to perform computations involving very large data sets increases, 64-bit machines and applications will become commonplace. It is therefore appropriate to take a close look at x86-64. We will see that in making the transition from 32 to 64 bits, the engineers at AMD also incorporated features that make the machines better targets for optimizing compilers and that improve system performance.

We provide Web Asides to cover material intended for dedicated machine-language enthusiasts. In one, we examine the code generated when code is compiled using higher degrees of optimization. Each successive version of the GCC compiler implements more sophisticated optimization algorithms, and these can radically transform a program to the point where it is difficult to understand the relation between the original source code and the generated machine-level program. Another Web Aside gives a brief presentation of ways to incorporate assembly code into C programs. For some applications, the programmer must drop down to assembly code to access low-level features of the machine. One approach is to write entire functions in assembly code and combine them with C functions during the linking stage. A second is to use GCC's support for embedding assembly code directly within C programs. We provide separate Web Asides for two different machine languages for floating-point code. The "x87" floating-point instructions have been available since the early days of Intel processors. This implementation of floating point is particularly arcane, and so we advise that only people determined to work with floating-point code on older machines attempt to study this section. The more recent "SSE" instructions were developed to support *multimedia applications*, but in their more recent versions (version 2 and later), and with more recent versions of GCC, SSE has become the preferred method for mapping floating point onto both IA32 and x86-64 machines.

3.1 A Historical Perspective

The Intel processor line, colloquially referred to as *x86*, has followed a long, evolutionary development. It started with one of the first single-chip, 16-bit microprocessors, where many compromises had to be made due to the limited capabilities of integrated circuit technology at the time. Since then, it has grown to take advantage of technology improvements as well as to satisfy the demands for higher performance and for supporting more advanced operating systems.

The list that follows shows some models of Intel processors and some of their key features, especially those affecting machine-level programming. We use the number of transistors required to implement the processors as an indication of how they have evolved in complexity (K denotes 1000, and M denotes 1,000,000).

8086: (1978, 29 K transistors). One of the first single-chip, 16-bit microprocessors. The 8088, a variant of the 8086 with an 8-bit external bus, formed the heart of the original IBM personal computers. IBM contracted with then-tiny Microsoft to develop the MS-DOS operating system. The original models came with 32,768 bytes of memory and two floppy drives (no hard drive). Architecturally, the machines were limited to a 655,360-byte address space—addresses were only 20 bits long (1,048,576 bytes addressable), and the operating system reserved 393,216 bytes for its own use. In 1980, Intel introduced the 8087 floating-point coprocessor (45 K transistors) to operate alongside an 8086 or 8088 processor, executing the floating-point instructions. The 8087 established the floating-point model for the x86 line, often referred to as "x87."

80286: (1982, 134 K transistors). Added more (and now obsolete) addressing modes. Formed the basis of the IBM PC-AT personal computer, the original platform for MS Windows.

i386: (1985, 275 K transistors). Expanded the architecture to 32 bits. Added the flat addressing model used by Linux and recent versions of the Windows family of operating system. This was the first machine in the series that could support a Unix operating system.

i486: (1989, 1.2 M transistors). Improved performance and integrated the floating-point unit onto the processor chip but did not significantly change the instruction set.

Pentium: (1993, 3.1 M transistors). Improved performance, but only added minor extensions to the instruction set.

PentiumPro: (1995, 5.5 M transistors). Introduced a radically new processor design, internally known as the *P6* microarchitecture. Added a class of "conditional move" instructions to the instruction set.

Pentium II: (1997, 7 M transistors). Continuation of the P6 microarchitecture.

Pentium III: (1999, 8.2 M transistors). Introduced SSE, a class of instructions for manipulating vectors of integer or floating-point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits. Later versions of this chip went up to 24 M transistors, due to the incorporation of the level-2 cache on chip.

Pentium 4: (2000, 42 M transistors). Extended SSE to SSE2, adding new data types (including double-precision floating point), along with 144 new instructions for these formats. With these extensions, compilers can use SSE instructions, rather than x87 instructions, to compile floating-point code. Introduced the *NetBurst* microarchitecture, which could operate at very high clock speeds, but at the cost of high power consumption.

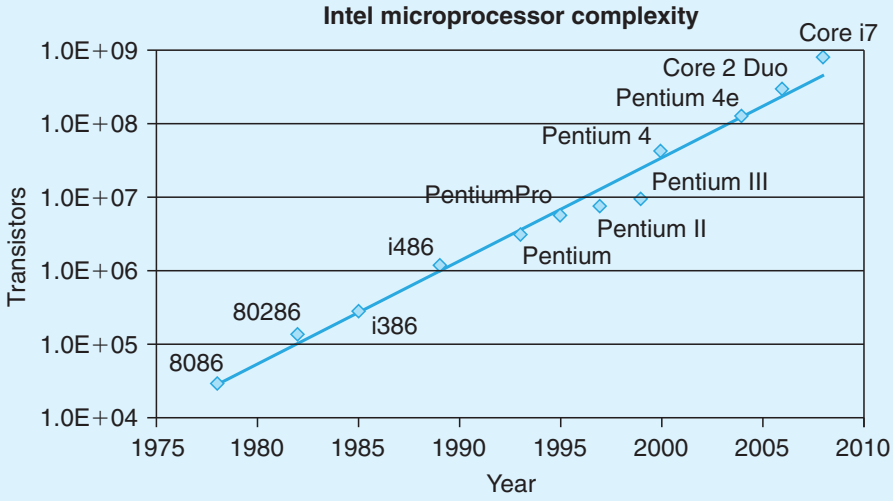
Pentium 4E: (2004, 125 M transistors). Added *hyperthreading*, a method to run two programs simultaneously on a single processor, as well as EM64T, Intel's implementation of a 64-bit extension to IA32 developed by Advanced Micro Devices (AMD), which we refer to as x86-64.

Core 2: (2006, 291 M transistors). Returned back to a microarchitecture similar to P6. First *multi-core* Intel microprocessor, where multiple processors are implemented on a single chip. Did not support hyperthreading.

Core i7: (2008, 781 M transistors). Incorporated both hyperthreading and multi-core, with the initial version supporting two executing programs on each core and up to four cores on each chip.

Each successive processor has been designed to be backward compatible—able to run code compiled for any earlier version. As we will see, there are many strange artifacts in the instruction set due to this evolutionary heritage. Intel has had several names for their processor line, including *IA32*, for "Intel Architecture 32-bit," and most recently *Intel64*, the 64-bit extension to IA32, which we will refer to as *x86-64*. We will refer to the overall line by the commonly used colloquial name "x86," reflecting the processor naming conventions up through the i486.

Aside Moore's law



If we plot the number of transistors in the different Intel processors versus the year of introduction, and use a logarithmic scale for the *y*-axis, we can see that the growth has been phenomenal. Fitting a line through the data, we see that the number of transistors increases at an annual rate of approximately 38%, meaning that the number of transistors doubles about every 26 months. This growth has been sustained over the multiple-decade history of x86 microprocessors.

In 1965, Gordon Moore, a founder of Intel Corporation, extrapolated from the chip technology of the day, in which they could fabricate circuits with around 64 transistors on a single chip, to predict that the number of transistors per chip would double every year for the next 10 years. This predication became known as *Moore's law*. As it turns out, his prediction was just a little bit optimistic, but also too short-sighted. Over more than 45 years, the semiconductor industry has been able to double transistor counts on average every 18 months.

Similar exponential growth rates have occurred for other aspects of computer technology—disk capacities, memory-chip capacities, and processor performance. These remarkable growth rates have been the major driving forces of the computer revolution.

Over the years, several companies have produced processors that are compatible with Intel processors, capable of running the exact same machine-level programs. Chief among these is Advanced Micro Devices (AMD). For years, AMD lagged just behind Intel in technology, forcing a marketing strategy where they produced processors that were less expensive although somewhat lower in performance. They became more competitive around 2002, being the first to break the 1-gigahertz clock-speed barrier for a commercially available microprocessor, and introducing x86-64, the widely adopted 64-bit extension to IA32. Although we will talk about Intel processors, our presentation holds just as well for the compatible processors produced by Intel's rivals.

Much of the complexity of x86 is not of concern to those interested in programs for the Linux operating system as generated by the GCC compiler. The memory model provided in the original 8086 and its extensions in the 80286 are obsolete. Instead, Linux uses what is referred to as *flat* addressing, where the entire memory space is viewed by the programmer as a large array of bytes.

As we can see in the list of developments, a number of formats and instructions have been added to x86 for manipulating vectors of small integers and floating-point numbers. These features were added to allow improved performance on multimedia applications, such as image processing, audio and video encoding and decoding, and three-dimensional computer graphics. In its default invocation for 32-bit execution, GCC assumes it is generating code for an i386, even though there are very few of these 1985-era microprocessors running any longer. Only by giving specific command-line options, or by compiling for 64-bit operation, will the compiler make use of the more recent extensions.

For the next part of our presentation, we will focus only on the IA32 instruction set. We will then look at the extension to 64 bits via x86-64 toward the end of the chapter.

3.2 Program Encodings

Suppose we write a C program as two files `p1.c` and `p2.c`. We can then compile this code on an IA32 machine using a Unix command line:

```
unix> gcc -O1 -o p p1.c p2.c
```

The command `gcc` indicates the GCC C compiler. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The command-line option `-O1` instructs the compiler to apply level-one optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code. As we will also see, invoking higher levels of optimization can generate code that is so heavily transformed that the relationship between the generated machine code and the original source code is difficult to understand. We will therefore use level-one optimization as a learning tool and then see what happens as we increase the level of optimization. In practice, level-two optimization (specified with the option `-O2`) is considered a better choice in terms of the resulting program performance.

The `gcc` command actually invokes a sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros, specified with `#define` declarations. Second, the *compiler* generates assembly-code versions of the two source files having names `p1.s` and `p2.s`. Next, the *assembler* converts the assembly code into binary *object-code* files `p1.o` and `p2.o`. Object code is one form of machine code—it contains binary representations of all of the instructions, but the addresses of global values are not yet filled in. Finally, the *linker* merges these two object-code files along with code implementing library functions (e.g., `printf`) and generates the final executable code file `p`. Executable code is the second form of machine code we will consider—it is the exact form of code that is executed by the processor. The relation between these different forms of machine code and the linking process is described in more detail in Chapter 7.

3.2.1 Machine-Level Code

As described in Section 1.9.2, computer systems employ several different forms of abstraction, hiding details of an implementation through the use of a simpler, abstract model. Two of these are especially important for machine-level programming. First, the format and behavior of a machine-level program is defined by the *instruction set architecture*, or "ISA," defining the processor state, the format of the instructions, and the effect each of these instructions will have on the state. Most ISAs, including IA32 and x86-64, describe the behavior of a program as if each instruction is executed in sequence, with one instruction completing before the next one begins. The processor hardware is far more elaborate, executing many instructions concurrently, but they employ safeguards to ensure that the overall behavior matches the sequential operation dictated by the ISA. Second, the memory addresses used by a machine-level program are virtual addresses, providing a memory model that appears to be a very large byte array. The actual implementation of the memory system involves a combination of multiple hardware memories and operating system software, as described in Chapter 9.

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model pro-

vided by C into the very elementary instructions that the processor executes. The assembly-code representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of machine code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

IA32 machine code differs greatly from the original C code. Parts of the processor state are visible that normally are hidden from the C programmer:

- The *program counter* (commonly referred to as the "PC," and called `%eip` in IA32) indicates the address in memory of the next instruction to be executed.
- The integer *register file* contains eight named locations storing 32-bit values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the local variables of a procedure, and the value to be returned by a function.
- The condition code registers hold status information about the most recently executed arithmetic or logical instruction. These are used to implement conditional changes in the control or data flow, such as is required to implement `if` and `while` statements.
- A set of floating-point registers store floating-point data.

Whereas C provides a model in which objects of different data types can be declared and allocated in memory, machine code views the memory as simply a large, byte-addressable array. Aggregate data types in C such as arrays and structures are represented in machine code as contiguous collections of bytes. Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.

The program memory contains the executable machine code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user (for example, by using the `malloc` library function). As mentioned earlier, the program memory is addressed using virtual addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the 32-bit addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

A single machine instruction performs only a very elementary operation. For example, it might add two numbers stored in registers, transfer data between memory and a register, or conditionally branch to a new instruction address. The compiler must generate sequences of such instructions to implement program constructs such as arithmetic expression evaluation, loops, or procedure calls and returns.

Aside The ever-changing forms of generated code

In our presentation, we will show the code generated by a particular version of GCC with particular settings of the command-line options. If you compile code on your own machine, chances are you will be using a different compiler or a different version of GCC and hence will generate different code. The open-source community supporting GCC keeps changing the code generator, attempting to generate more efficient code according to changing code guidelines provided by the microprocessor manufacturers.

Our goal in studying the examples shown in our presentation is to demonstrate how to examine assembly code and map it back to the constructs found in high-level programming languages. You will need to adapt these techniques to the style of code generated by your particular compiler.

3.2.2 Code Examples

Suppose we write a C code file `code.c` containing the following procedure definition:

```
1   int accum = 0;
2
3   int sum(int x, int y)
4   {
5       int t = x + y;
6       accum += t;
7       return t;
8   }
```

To see the assembly code generated by the C compiler, we can use the "`-S`" option on the command line:

```
unix> gcc -O1 -S code.c
```

This will cause GCC to run the compiler, generating an assembly file `code.s`, and go no further. (Normally it would then invoke the assembler to generate an object-code file.)

The assembly-code file contains various declarations including the set of lines:

```
sum:
  pushl   %ebp
  movl    %esp, %ebp
  movl    12(%ebp), %eax
  addl    8(%ebp), %eax
  addl    %eax, accum
  popl    %ebp
  ret
```

Each indented line in the above code corresponds to a single machine instruction. For example, the `pushl` instruction indicates that the contents of register `%ebp` should be pushed onto the program stack. All information about local variable names or data types has been stripped away. We still see a reference to the global

variable `accum`, since the compiler has not yet determined where in memory this variable will be stored.

If we use the '`-c`' command-line option, GCC will both compile and assemble the code:

```
unix> gcc -O1 -c code.c
```

This will generate an object-code file `code.o` that is in binary format and hence cannot be viewed directly. Embedded within the 800 bytes of the file `code.o` is a 17-byte sequence having hexadecimal representation

```
55 89 e5 8b 45 0c 03 45 08 01 05 00 00 00 00 5d c3
```

This is the object-code corresponding to the assembly instructions listed above. A key lesson to learn from this is that the program actually executed by the machine is simply a sequence of bytes encoding a series of instructions. The machine has very little information about the source code from which these instructions were generated.

Aside How do I find the byte representation of a program?

To generate these bytes, we used a *disassembler* (to be described shortly) to determine that the code for `sum` is 17 bytes long. Then we ran the GNU debugging tool GDB on file `code.o` and gave it the command

```
(gdb) x/17xb sum
```

telling it to examine (abbreviated 'x') 17 hex-formatted (also abbreviated 'x') bytes (abbreviated 'b'). You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 3.11.

To inspect the contents of machine-code files, a class of programs known as *disassemblers* can be invaluable. These programs generate a format similar to assembly code from the machine code. With Linux systems, the program OBJDUMP (for "object dump") can serve this role given the '`-d`' command-line flag:

```
unix> objdump -d code.o
```

The result is (where we have added line numbers on the left and annotations in italicized text) as follows:

```
     Disassembly of function sum in binary file code.o
1    00000000 <sum>:
     Offset   Bytes              Equivalent assembly language
2       0:    55                 push   %ebp
3       1:    89 e5              mov    %esp,%ebp
4       3:    8b 45 0c           mov    0xc(%ebp),%eax
5       6:    03 45 08           add    0x8(%ebp),%eax
6       9:    01 05 00 00 00 00  add    %eax,0x0
7       f:    5d                 pop    %ebp
8      10:    c3                 ret
```

On the left, we see the 17 hexadecimal byte values listed in the byte sequence earlier, partitioned into groups of 1 to 6 bytes each. Each of these groups is a single instruction, with the assembly-language equivalent shown on the right.

Several features about machine code and its disassembled representation are worth noting:

- IA32 instructions can range in length from 1 to 15 bytes. The instruction encoding is designed so that commonly used instructions and those with fewer operands require a smaller number of bytes than do less common ones or ones with more operands.
- The instruction format is designed in such a way that from a given starting position, there is a unique decoding of the bytes into machine instructions. For example, only the instruction `pushl %ebp` can start with byte value `55`.
- The disassembler determines the assembly code based purely on the byte sequences in the machine-code file. It does not require access to the source or assembly-code versions of the program.
- The disassembler uses a slightly different naming convention for the instructions than does the assembly code generated by GCC. In our example, it has omitted the suffix '`l`' from many of the instructions. These suffixes are size designators and can be omitted in most cases.

Generating the actual executable code requires running a linker on the set of object-code files, one of which must contain a function `main`. Suppose in file `main.c` we had the following function:

```
1  int main()
2  {
3      return sum(1, 3);
4  }
```

Then, we could generate an executable program `prog` as follows:

```
unix> gcc -O1 -o prog code.o main.c
```

The file `prog` has grown to 9,123 bytes, since it contains not just the code for our two procedures but also information used to start and terminate the program as well as to interact with the operating system. We can also disassemble the file `prog`:

```
unix> objdump -d prog
```

The disassembler will extract various code sequences, including the following:

```
    Disassembly of function sum in executable file prog
1   08048394 <sum>:
     Offset    Bytes                    Equivalent assembly language
2    8048394:  55                       push   %ebp
3    8048395:  89 e5                    mov    %esp,%ebp
4    8048397:  8b 45 0c                 mov    0xc(%ebp),%eax
```

```
5    804839a:  03 45 08                add    0x8(%ebp),%eax
6    804839d:  01 05 18 a0 04 08       add    %eax,0x804a018
7    80483a3:  5d                      pop    %ebp
8    80483a4:  c3                      ret
```

This code is almost identical to that generated by the disassembly of `code.c`. One important difference is that the addresses listed along the left are different—the linker has shifted the location of this code to a different range of addresses. A second difference is that the linker has determined the location for storing global variable `accum`. On line 6 of the disassembly for `code.o`, the address of `accum` was listed as `0`. In the disassembly of `prog`, the address has been set to `0x804a018`. This is shown in the assembly-code rendition of the instruction. It can also be seen in the last 4 bytes of the instruction, listed from least-significant to most as `18 a0 04 08`.

3.2.3 Notes on Formatting

The assembly code generated by GCC is difficult for a human to read. On one hand, it contains information with which we need not be concerned, while on the other hand, it does not provide any description of the program or how it works. For example, suppose the file `simple.c` contains the following code:

```
1   int simple(int *xp, int y)
2   {
3     int t = *xp + y;
4     *xp = t;
5     return t;
6   }
```

When GCC is run with flags '`-S`' and '`-O1`', it generates the following file for `simple.s`:

```
  .file   "simple.c"
  .text
.globl simple
  .type   simple, @function
simple:
  pushl   %ebp
  movl    %esp, %ebp
  movl    8(%ebp), %edx
  movl    12(%ebp), %eax
  addl    (%edx), %eax
  movl    %eax, (%edx)
  popl    %ebp
  ret
  .size   simple, .-simple
  .ident  "GCC: (Ubuntu 4.3.2-1ubuntu11) 4.3.2"
  .section        .note.GNU-stack,"",@progbits
```

All of the lines beginning with '.' are directives to guide the assembler and linker. We can generally ignore these. On the other hand, there are no explanatory remarks about what the instructions do or how they relate to the source code.

To provide a clearer presentation of assembly code, we will show it in a form that omits most of the directives, while including line numbers and explanatory annotations. For our example, an annotated version would appear as follows:

```
1   simple:
2     pushl   %ebp              Save frame pointer
3     movl    %esp, %ebp        Create new frame pointer
4     movl    8(%ebp), %edx     Retrieve xp
5     movl    12(%ebp), %eax    Retrieve y
6     addl    (%edx), %eax      Add *xp to get t
7     movl    %eax, (%edx)      Store t at xp
8     popl    %ebp              Restore frame pointer
9     ret                       Return
```

We typically show only the lines of code relevant to the point being discussed. Each line is numbered on the left for reference and annotated on the right by a brief description of the effect of the instruction and how it relates to the computations of the original C code. This is a stylized version of the way assembly-language programmers format their code.

Aside ATT versus Intel assembly-code formats

In our presentation, we show assembly code in ATT (named after "AT&T," the company that operated Bell Laboratories for many years) format, the default format for GCC, OBJDUMP, and the other tools we will consider. Other programming tools, including those from Microsoft as well as the documentation from Intel, show assembly code in *Intel* format. The two formats differ in a number of ways. As an example, GCC can generate code in Intel format for the `sum` function using the following command line:

```
unix> gcc -O1 -S -masm=intel code.c
```

This gives the following assembly code:

```
    Assembly code for simple in Intel format
1   simple:
2     push    ebp
3     mov     ebp, esp
4     mov     edx, DWORD PTR [ebp+8]
5     mov     eax, DWORD PTR [ebp+12]
6     add     eax, DWORD PTR [edx]
7     mov     DWORD PTR [edx], eax
8     pop     ebp
9     ret
```

We see that the Intel and ATT formats differ in the following ways:

- The Intel code omits the size designation suffixes. We see instruction `mov` instead of `movl`.
- The Intel code omits the '`%`' character in front of register names, using `esp` instead of `%esp`.
- The Intel code has a different way of describing locations in memory, for example '`DWORD PTR [ebp+8]`' rather than '`8(%ebp)`'.
- Instructions with multiple operands list them in the reverse order. This can be very confusing when switching between the two formats.

Although we will not be using Intel format in our presentation, you will encounter it in IA32 documentation from Intel and Windows documentation from Microsoft.

3.3 Data Formats

Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term "word" to refer to a 16-bit data type. Based on this, they refer to 32-bit quantities as "double words." They refer to 64-bit quantities as "quad words." Most instructions we will encounter operate on bytes or double words.

Figure 3.1 shows the IA32 representations used for the primitive data types of C. Most of the common data types are stored as double words. This includes both regular and long `int`'s, whether or not they are signed. In addition, all pointers (shown here as `char *`) are stored as 4-byte double words. Bytes are commonly used when manipulating string data. As we saw in Section 2.1, more recent extensions of the C language include the data type `long long`, which is represented using 8 bytes. IA32 does not support this data type in hardware. Instead, the compiler must generate sequences of instructions that operate on these data 32 bits

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`long long int`	—	—	4
`char *`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8
`long double`	Extended precision	`t`	10/12

Figure 3.1 **Sizes of C data types in IA32.** IA32 does not provide hardware support for 64-bit integer arithmetic. Compiling code with `long long` data requires generating sequences of operations to perform the arithmetic in 32-bit chunks.

at a time. Floating-point numbers come in three different forms: single-precision (4-byte) values, corresponding to C data type `float`; double-precision (8-byte) values, corresponding to C data type `double`; and extended-precision (10-byte) values. GCC uses the data type `long double` to refer to extended-precision floating-point values. It also stores them as 12-byte quantities to improve memory system performance, as will be discussed later. Using the `long double` data type (introduced in ISO C99) gives us access to the extended-precision capability of x86. For most other machines, this data type will be represented using the same 8-byte format of the ordinary `double` data type.

As the table indicates, most assembly-code instructions generated by GCC have a single-character suffix denoting the size of the operand. For example, the data movement instruction has three variants: `movb` (move byte), `movw` (move word), and `movl` (move double word). The suffix '`l`' is used for double words, since 32-bit quantities are considered to be "long words," a holdover from an era when 16-bit word sizes were standard. Note that the assembly code uses the suffix '`l`' to denote both a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating point involves an entirely different set of instructions and registers.

3.4 Accessing Information

An IA32 central processing unit (CPU) contains a set of eight *registers* storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 3.2 diagrams the eight registers. Their names all begin with `%e`, but otherwise, they have peculiar names. With the original 8086, the registers were 16 bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first six registers can be considered general-purpose regis-

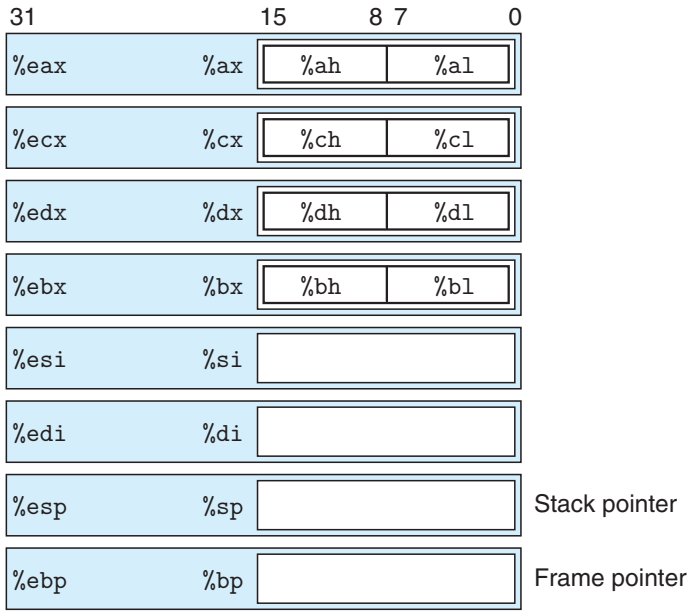


Figure 3.2
IA32 integer registers. All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The 2 low-order bytes of the first four registers can be accessed independently.

ters with no restrictions placed on their use. We said "for the most part," because some instructions use fixed registers as sources and/or destinations. In addition, within procedures there are different conventions for saving and restoring the first three registers (`%eax`, `%ecx`, and `%edx`) than for the next three (`%ebx`, `%edi`, and `%esi`). This will be discussed in Section 3.7. The final two registers (`%ebp` and `%esp`) contain pointers to important places in the program stack. They should only be altered according to the set of standard conventions for stack management.

As indicated in Figure 3.2, the low-order 2 bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining 3 bytes of the register do not change. Similarly, the low-order 16 bits of each register can be read or written by word operation instructions. This feature stems from IA32's evolutionary heritage as a 16-bit microprocessor and is also used when operating on integers with size designator `short`.

3.4.1 Operand Specifiers

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. IA32 supports a number of operand forms (see Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. In ATT-format assembly code, these are written with a '`$`' followed by an integer using standard C notation, for example, `$-577` or `$0x1F`. Any value that fits into a 32-bit word can be used, although the assembler will use 1- or 2-byte encodings

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

when possible. The second type, *register*, denotes the contents of one of the registers, either one of the eight 32-bit registers (e.g., `%eax`) for a double-word operation, one of the eight 16-bit registers (e.g., `%ax`) for a word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In Figure 3.3, we use the notation $\mathtt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $\mathtt{R}[\mathtt{E}_a]$, viewing the set of registers as an array R indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. Since we view the memory as a large array of bytes, we use the notation $\mathtt{M}_b[Addr]$ to denote a reference to the b-byte value stored in memory starting at address $Addr$. To simplify things, we will generally drop the subscript b.

As Figure 3.3 shows, there are many different *addressing modes* allowing different forms of memory references. The most general form is shown at the bottom of the table with syntax $Imm(\mathtt{E}_b,\mathtt{E}_i,s)$. Such a reference has four components: an immediate offset Imm, a base register $\mathtt{E}_b$, an index register $\mathtt{E}_i$, and a scale factor s, where s must be 1, 2, 4, or 8. The effective address is then computed as $Imm + \mathtt{R}[\mathtt{E}_b] + \mathtt{R}[\mathtt{E}_i] \cdot s$. This general form is often seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted. As we will see, the more complex addressing modes are useful when referencing array and structure elements.

Practice Problem 3.1

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the values for the indicated operands:

Operand	Value
%eax	_____
0x104	_____
$0x108	_____
(%eax)	_____
4(%eax)	_____
9(%eax,%edx)	_____
260(%ecx,%edx)	_____
0xFC(,%ecx,4)	_____
(%eax,%edx,4)	_____

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
MOVS	S, D	$D \leftarrow$ SignExtend(S)	Move with sign extension
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
MOVZ	S, D	$D \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	
`pushl`	S	R[%esp] $\leftarrow$ R[%esp] $-$ 4; M[R[%esp]] $\leftarrow S$	Push double word
`popl`	D	$D \leftarrow$ M[R[%esp]]; R[%esp] $\leftarrow$ R[%esp] $+$ 4	Pop double word

Figure 3.4 **Data movement instructions.**

3.4.2 Data Movement Instructions

Among the most heavily used instructions are those that copy data from one location to another. The generality of the operand notation allows a simple data movement instruction to perform what in many machines would require a number of instructions. Figure 3.4 lists the important data movement instructions. As can be seen, we group the many different instructions into *instruction classes*, where the instructions in a class perform the same operation, but with different operand sizes. For example, the MOV class consists of three instructions: `movb`, `movw`, and `movl`. All three of these instructions perform the same operation; they differ only in that they operate on data of size 1, 2, and 4 bytes, respectively.

The instructions in the MOV class copy their source values to their destinations. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination. Referring to Figure 3.2, the register operands for these instructions can be any of the eight 32-bit registers (`%eax`–`%ebp`) for `movl`, any of the eight 16-bit registers (`%ax`–`%bp`) for `movw`, and any of the single-byte register elements (`%ah`–`%bh`, `%al`–`%bl`) for `movb`. The following MOV instruction examples show the five

possible combinations of source and destination types. Recall that the source operand comes first and the destination second:

```
1      movl $0x4050,%eax          Immediate--Register, 4 bytes
2      movw  %bp,%sp              Register--Register,  2 bytes
3      movb (%edi,%ecx),%ah       Memory--Register,    1 byte
4      movb $-17,(%esp)           Immediate--Memory,   1 byte
5      movl %eax,-12(%ebp)        Register--Memory,    4 bytes
```

Both the MOVS and the MOVZ instruction classes serve to copy a smaller amount of source data to a larger data location, filling in the upper bits by either sign expansion (MOVS) or by zero expansion (MOVZ). With sign expansion, the upper bits of the destination are filled in with copies of the most significant bit of the source value. With zero expansion, the upper bits are filled with zeros. As can be seen, there are three instructions in each of these classes, covering all cases of 1- and 2-byte source sizes and 2- and 4-byte destination sizes (omitting the redundant combinations `movsww` and `movzww`, of course).

Aside Comparing byte movement instructions

Observe that the three byte-movement instructions `movb`, `movsbl`, and `movzbl` differ from each other in subtle ways. Here is an example:

```
       Assume initially that %dh = CD, %eax = 98765432
1       movb %dh,%al              %eax = 987654CD
2       movsbl %dh,%eax           %eax = FFFFFFCD
3       movzbl %dh,%eax           %eax = 000000CD
```

In these examples, all set the low-order byte of register `%eax` to the second byte of `%edx`. The `movb` instruction does not change the other 3 bytes. The `movsbl` instruction sets the other 3 bytes to either all ones or all zeros, depending on the high-order bit of the source byte. The `movzbl` instruction sets the other 3 bytes to all zeros in any case.

The final two data movement operations are used to push data onto and pop data from the program stack. As we will see, the stack plays a vital role in the handling of procedure calls. By way of background, a stack is a data structure where values can be added or deleted, but only according to a "last-in, first-out" discipline. We add data to a stack via a *push* operation and remove it via a *pop* operation, with the property that the value popped will always be the value that was most recently pushed and is still on the stack. A stack can be implemented as an array, where we always insert and remove elements from one end of the array. This end is called the *top* of the stack. With IA32, the program stack is stored in some region of memory. As illustrated in Figure 3.5, the stack grows downward such that the top element of the stack has the lowest address of all stack elements. (By convention, we draw stacks upside down, with the stack "top" shown at the bottom of the figure). The stack pointer `%esp` holds the address of the top stack element.

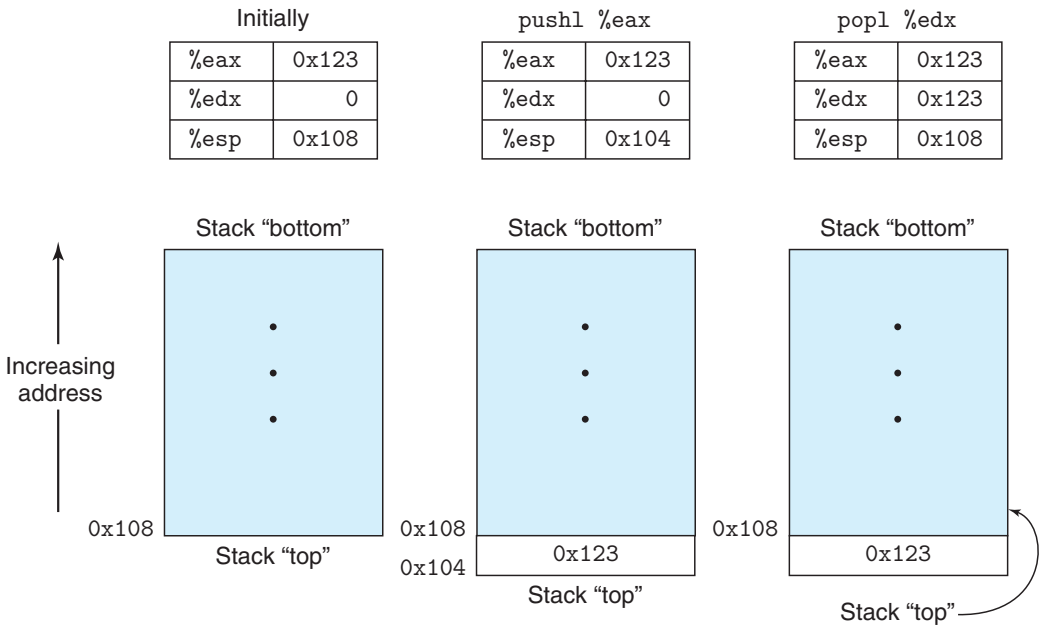


Figure 3.5 **Illustration of stack operation.** By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %esp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

The `pushl` instruction provides the ability to push data onto the stack, while the `popl` instruction pops it. Each of these instructions takes a single operand—the data source for pushing and the data destination for popping.

Pushing a double-word value onto the stack involves first decrementing the stack pointer by 4 and then writing the value at the new top of stack address. Therefore, the behavior of the instruction `pushl %ebp` is equivalent to that of the pair of instructions

```
subl $4,%esp              Decrement stack pointer
movl %ebp,(%esp)          Store %ebp on stack
```

except that the `pushl` instruction is encoded in the machine code as a single byte, whereas the pair of instructions shown above requires a total of 6 bytes. The first two columns in Figure 3.5 illustrate the effect of executing the instruction `pushl %eax` when `%esp` is `0x108` and `%eax` is `0x123`. First `%esp` is decremented by 4, giving `0x104`, and then `0x123` is stored at memory address `0x104`.

Popping a double word involves reading from the top of stack location and then incrementing the stack pointer by 4. Therefore, the instruction `popl %eax` is equivalent to the following pair of instructions:

```
movl (%esp),%eax          Read %eax from stack
addl $4,%esp              Increment stack pointer
```

The third column of Figure 3.5 illustrates the effect of executing the instruction `popl %edx` immediately after executing the `pushl`. Value `0x123` is read from

memory and written to register %edx. Register %esp is incremented back to 0x108. As shown in the figure, the value 0x123 remains at memory location 0x104 until it is overwritten (e.g., by another push operation). However, the stack top is always considered to be the address indicated by %esp. Any value stored beyond the stack top is considered invalid.

Since the stack is contained in the same memory as the program code and other forms of program data, programs can access arbitrary positions within the stack using the standard memory addressing methods. For example, assuming the topmost element of the stack is a double word, the instruction `movl 4(%esp),%edx` will copy the second double word from the stack to register %edx.

Practice Problem 3.2

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, `mov` can be rewritten as `movb`, `movw`, or `movl`.)

```
1      mov   %eax, (%esp)
2      mov   (%eax), %dx
3      mov   $0xFF, %bl
4      mov   (%esp,%edx,4), %dh
5      push  $0xFF
6      mov   %dx, (%eax)
7      pop   %edi
```

Practice Problem 3.3

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
1      movb $0xF, (%bl)
2      movl %ax, (%esp)
3      movw (%eax),4(%esp)
4      movb %ah,%sh
5      movl %eax,$0x123
6      movl %eax,%dx
7      movb %si, 8(%ebp)
```

3.4.3 Data Movement Example

As an example of code that uses data movement instructions, consider the data exchange routine shown in Figure 3.6, both as C code and as assembly code generated by GCC. We omit the portion of the assembly code that allocates space on the run-time stack on procedure entry and deallocates it prior to return. The details of this set-up and completion code will be covered when we discuss procedure linkage. The code we are left with is called the "body."

New to C? Some examples of pointers

Function `exchange` (Figure 3.6) provides a good illustration of the use of pointers in C. Argument `xp` is a pointer to an integer, while `y` is an integer itself. The statement

```
int x = *xp;
```

indicates that we should read the value stored in the location designated by `xp` and store it as a local variable named `x`. This read operation is known as pointer *dereferencing*. The C operator `*` performs pointer dereferencing.

The statement

```
*xp = y;
```

does the reverse—it writes the value of parameter `y` at the location designated by `xp`. This is also a form of pointer dereferencing (and hence the operator `*`), but it indicates a write operation since it is on the left-hand side of the assignment.

The following is an example of `exchange` in action:

```
int a = 4;
int b = exchange(&a, 3);
printf("a = %d, b = %d\n", a, b);
```

This code will print

```
a = 3, b = 4
```

The C operator `&` (called the "address of" operator) *creates* a pointer, in this case to the location holding local variable `a`. Function `exchange` then overwrote the value stored in `a` with 3 but returned 4 as the function value. Observe how by passing a pointer to `exchange`, it could modify data held at some remote location.

When the body of the procedure starts execution, procedure parameters `xp` and `y` are stored at offsets 8 and 12 relative to the address in register `%ebp`. Instructions 1 and 2 read parameter `xp` from memory and store it in register

(a) C code

```
1   int exchange(int *xp, int y)
2   {
3       int x = *xp;
4
5       *xp = y;
6       return x;
7   }
```

(b) Assembly code

```
    xp at %ebp+8, y at %ebp+12
1     movl    8(%ebp), %edx     Get xp
    By copying to %eax below, x becomes the return value
2     movl    (%edx), %eax      Get x at xp
3     movl    12(%ebp), %ecx    Get y
4     movl    %ecx, (%edx)      Store y at xp
```

Figure 3.6 **C and assembly code for exchange routine body.** The stack set-up and completion portions have been omitted.

%edx. Instruction 2 uses register %edx and reads x into register %eax, a direct implementation of the operation x = *xp in the C program. Later, register %eax will be used to return a value from this function, and so the return value will be x. Instruction 3 loads parameter y into register %ecx. Instruction 4 then writes this value to the memory location designated by xp in register %edx, a direct implementation of the operation *xp = y. This example illustrates how the MOV instructions can be used to read from memory to a register (instructions 1 to 3), and to write from a register to memory (instruction 4.)

Two features about this assembly code are worth noting. First, we see that what we call "pointers" in C are simply addresses. Dereferencing a pointer involves copying that pointer into a register, and then using this register in a memory reference. Second, local variables such as x are often kept in registers rather than stored in memory locations. Register access is much faster than memory access.

Practice Problem 3.4

Assume variables v and p declared with types

```
src_t v;
dest_t *p;
```

where src_t and dest_t are data types declared with typedef. We wish to use the appropriate data movement instruction to implement the operation

```
*p = (dest_t) v;
```

where v is stored in the appropriately named portion of register %eax (i.e., %eax, %ax, or %al), while pointer p is stored in register %edx.

For the following combinations of src_t and dest_t, write a line of assembly code that does the appropriate transfer. Recall that when performing a cast that involves both a size change and a change of "signedness" in C, the operation should change the signedness first (Section 2.2.6).

src_t	dest_t	Instruction
int	int	movl %eax, (%edx)
char	int	_____
char	unsigned	_____
unsigned char	int	_____
int	char	_____
unsigned	unsigned char	_____
unsigned	int	_____

Practice Problem 3.5

You are given the following information. A function with prototype

```
void decode1(int *xp, int *yp, int *zp);
```

is compiled into assembly code. The body of the code is as follows:

```
   xp at %ebp+8, yp at %ebp+12, zp at %ebp+16
1      movl    8(%ebp), %edi
2      movl    12(%ebp), %edx
3      movl    16(%ebp), %ecx
4      movl    (%edx), %ebx
5      movl    (%ecx), %esi
6      movl    (%edi), %eax
7      movl    %eax, (%edx)
8      movl    %ebx, (%ecx)
9      movl    %esi, (%edi)
```

Parameters `xp`, `yp`, and `zp` are stored at memory locations with offsets 8, 12, and 16, respectively, relative to the address in register `%ebp`.

Write C code for `decode1` that will have an effect equivalent to the assembly code above.

3.5 Arithmetic and Logical Operations

Figure 3.7 lists some of the integer and logic operations. Most of the operations are given as instruction classes, as they can have different variants with different operand sizes. (Only `leal` has no other size variants.) For example, the instruction class ADD consists of three addition instructions: `addb`, `addw`, and `addl`, adding bytes, words, and double words, respectively. Indeed, each of the instruction classes shown has instructions for operating on byte, word, and double-word data. The operations are divided into four groups: load effective address, unary, binary, and shifts. *Binary* operations have two operands, while *unary* operations have one operand. These operands are specified using the same notation as described in Section 3.4.

3.5.1 Load Effective Address

The *load effective address* instruction `leal` is actually a variant of the `movl` instruction. It has the form of an instruction that reads from memory to a register, but it does not reference memory at all. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 3.7 using the C address operator `&`S. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. Compilers often find clever uses of `leal` that have nothing to do with effective address computations. The destination operand must be a register.

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Figure 3.7 **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Practice Problem 3.6

Suppose register `%eax` holds value x and `%ecx` holds value y. Fill in the table below with formulas indicating the value that will be stored in register `%edx` for each of the given assembly code instructions:

Instruction	Result
`leal 6(%eax), %edx`	
`leal (%eax,%ecx), %edx`	
`leal (%eax,%ecx,4), %edx`	
`leal 7(%eax,%eax,8), %edx`	
`leal 0xA(,%ecx,4), %edx`	
`leal 9(%eax,%ecx,2), %edx`	

3.5.2 Unary and Binary Operations

Operations in the second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or

a memory location. For example, the instruction `incl (%esp)` causes the 4-byte element on the top of the stack to be incremented. This syntax is reminiscent of the C increment (++) and decrement (--) operators.

The third group consists of binary operations, where the second operand is used as both a source and a destination. This syntax is reminiscent of the C assignment operators, such as `x += y`. Observe, however, that the source operand is given first and the destination second. This looks peculiar for noncommutative operations. For example, the instruction `subl %eax,%edx` decrements register `%edx` by the value in `%eax`. (It helps to read the instruction as "Subtract `%eax` from `%edx`.") The first operand can be either an immediate value, a register, or a memory location. The second can be either a register or a memory location. As with the `movl` instruction, however, the two operands cannot both be memory locations.

Practice Problem 3.7

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
`addl %ecx,(%eax)`		
`subl %edx,4(%eax)`		
`imull $16,(%eax,%edx,4)`		
`incl 8(%eax)`		
`decl %ecx`		
`subl %edx,%eax`		

3.5.3 Shift Operations

The final group consists of shift operations, where the shift amount is given first, and the value to shift is given second. Both arithmetic and logical right shifts are possible. The shift amount is encoded as a single byte, since only shift amounts between 0 and 31 are possible (only the low-order 5 bits of the shift amount are considered). The shift amount is given either as an immediate or in the single-byte register element `%cl`. (These instructions are unusual in only allowing this specific register as operand.) As Figure 3.7 indicates, there are two names for the

left shift instruction: SAL and SHL. Both have the same effect, filling from the right with zeros. The right shift instructions differ in that SAR performs an arithmetic shift (fill with copies of the sign bit), whereas SHR performs a logical shift (fill with zeros). The destination operand of a shift operation can be either a register or a memory location. We denote the two different right shift operations in Figure 3.7 as $>>_A$ (arithmetic) and $>>_L$ (logical).

Practice Problem 3.8

Suppose we want to generate assembly code for the following C function:

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

The code that follows is a portion of the assembly code that performs the actual shifts and leaves the final value in register `%eax`. Two key instructions have been omitted. Parameters `x` and `n` are stored at memory locations with offsets 8 and 12, respectively, relative to the address in register `%ebp`.

```
1      movl    8(%ebp), %eax    Get x
2      _________               x <<= 2
3      movl    12(%ebp), %ecx   Get n
4      _________               x >>= n
```

Fill in the missing instructions, following the annotations on the right. The right shift should be performed arithmetically.

3.5.4 Discussion

We see that most of the instructions shown in Figure 3.7 can be used for either unsigned or two's-complement arithmetic. Only right shifting requires instructions that differentiate between signed versus unsigned data. This is one of the features that makes two's-complement arithmetic the preferred way to implement signed integer arithmetic.

Figure 3.8 shows an example of a function that performs arithmetic operations and its translation into assembly code. As before, we have omitted the stack set-up and completion portions. Function arguments `x`, `y`, and `z` are stored in memory at offsets 8, 12, and 16 relative to the address in register `%ebp`, respectively.

The assembly code instructions occur in a different order than in the C source code. Instructions 2 and 3 compute the expression `z*48` by a combination of `leal` and shift instructions. Line 5 computes the value of `x+y`. Line 6 computes the AND of `t1` and `0xFFFF`. The final multiply is computed by line 7. Since the destination of the multiply is register `%eax`, this will be the value returned by the function.

(a) C code

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = x+y;
6       int t2 = z*48;
7       int t3 = t1 & 0xFFFF;
8       int t4 = t2 * t3;
9       return t4;
10  }
```

(b) Assembly code

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
1     movl    16(%ebp), %eax        z
2     leal    (%eax,%eax,2), %eax   z*3
3     sall    $4, %eax              t2 = z*48
4     movl    12(%ebp), %edx        y
5     addl    8(%ebp), %edx         t1 = x+y
6     andl    $65535, %edx          t3 = t1&0xFFFF
7     imull   %edx, %eax            Return t4 = t2*t3
```

Figure 3.8 **C and assembly code for arithmetic routine body.** The stack set-up and completion portions have been omitted.

In the assembly code of Figure 3.8, the sequence of values in register %eax corresponds to program values z, 3*z, z*48, and t4 (as the return value). In general, compilers generate code that uses individual registers for multiple program values and moves program values among the registers.

Practice Problem 3.9

In the following variant of the function of Figure 3.8(a), the expressions have been replaced by blanks:

```
1   int arith(int x,
2             int y,
3             int z)
4   {
5       int t1 = _____;
6       int t2 = _____;
7       int t3 = _____;
8       int t4 = _____;
9       return t4;
10  }
```

The portion of the generated assembly code implementing these expressions is as follows:

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
1     movl    12(%ebp), %eax
2     xorl    8(%ebp), %eax
3     sarl    $3, %eax
4     notl    %eax
5     subl    16(%ebp), %eax
```

Based on this assembly code, fill in the missing portions of the C code.

Practice Problem 3.10

It is common to find assembly code lines of the form

```
xorl %edx,%edx
```

in code that was generated from C where no Exclusive-Or operations were present.

A. Explain the effect of this particular Exclusive-Or instruction and what useful operation it implements.

B. What would be the more straightforward way to express this operation in assembly code?

C. Compare the number of bytes to encode these two different implementations of the same operation.

3.5.5 Special Arithmetic Operations

Figure 3.9 describes instructions that support generating the full 64-bit product of two 32-bit numbers, as well as integer division.

The `imull` instruction, a member of the IMUL instruction class listed in Figure 3.7, is known as a "two-operand" multiply instruction. It generates a 32-bit product from two 32-bit operands, implementing the operations $*^{u}_{32}$ and $*^{t}_{32}$ described in Sections 2.3.4 and 2.3.5. Recall that when truncating the product to 32 bits, both unsigned multiply and two's-complement multiply have the same bit-level behavior. IA32 also provides two different "one-operand" multiply instructions to compute the full 64-bit product of two 32-bit values—one for unsigned (`mull`), and one for two's-complement (`imull`) multiplication. For both of these, one argument must be in register `%eax`, and the other is given as the instruction

Instruction		Effect	Description
`imull`	S	R[%edx]:R[%eax] $\leftarrow S \times$ R[%eax]	Signed full multiply
`mull`	S	R[%edx]:R[%eax] $\leftarrow S \times$ R[%eax]	Unsigned full multiply
`cltd`		R[%edx]:R[%eax] $\leftarrow$ SignExtend(R[%eax])	Convert to quad word
`idivl`	S	R[%edx] $\leftarrow$ R[%edx]:R[%eax] mod S; R[%eax] $\leftarrow$ R[%edx]:R[%eax] $\div S$	Signed divide
`divl`	S	R[%edx] $\leftarrow$ R[%edx]:R[%eax] mod S; R[%eax] $\leftarrow$ R[%edx]:R[%eax] $\div S$	Unsigned divide

Figure 3.9 **Special arithmetic operations.** These operations provide full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%edx` and `%eax` are viewed as forming a single 64-bit quad word.

source operand. The product is then stored in registers %edx (high-order 32 bits) and %eax (low-order 32 bits). Although the name imull is used for two distinct multiplication operations, the assembler can tell which one is intended by counting the number of operands.

As an example, suppose we have signed numbers x and y stored at positions 8 and 12 relative to %ebp, and we want to store their full 64-bit product as 8 bytes on top of the stack. The code would proceed as follows:

```
      x at %ebp+8, y at %ebp+12
1      movl    12(%ebp), %eax   Put y in %eax
2      imull   8(%ebp)          Multiply by x
3      movl    %eax, (%esp)     Store low-order 32 bits
4      movl    %edx, 4(%esp)    Store high-order 32 bits
```

Observe that the locations in which we store the two registers are correct for a little-endian machine—the high-order bits in register %edx are stored at offset 4 relative to the low-order bits in %eax. With the stack growing toward lower addresses, that means that the low-order bits are at the top of the stack.

Our earlier table of arithmetic operations (Figure 3.7) does not list any division or modulus operations. These operations are provided by the single-operand divide instructions similar to the single-operand multiply instructions. The signed division instruction idivl takes as dividend the 64-bit quantity in registers %edx (high-order 32 bits) and %eax (low-order 32 bits). The divisor is given as the instruction operand. The instruction stores the quotient in register %eax and the remainder in register %edx.

As an example, suppose we have signed numbers x and y stored at positions 8 and 12 relative to %ebp, and we want to store values x/y and x mod y on the stack. GCC generates the following code:

```
      x at %ebp+8, y at %ebp+12
1      movl    8(%ebp), %edx    Put x in %edx
2      movl    %edx, %eax       Copy x to %eax
3      sarl    $31, %edx        Sign extend x in %edx
4      idivl   12(%ebp)         Divide by y
5      movl    %eax, 4(%esp)    Store x / y
6      movl    %edx, (%esp)     Store x % y
```

The move instruction on line 1 and the arithmetic shift on line 3 have the combined effect of setting register %edx to either all zeros or all ones depending on the sign of x, while the move instruction on line 2 copies x into %eax. Thus, we have the combined registers %edx and %eax storing a 64-bit, sign-extended version of x. Following the idivl instruction, the quotient and remainder are copied to the top two stack locations (instructions 5 and 6).

A more conventional method of setting up the divisor makes use of the `cltd`[1] instruction. This instruction sign extends `%eax` into `%edx`. With this instruction, the code sequence shown above becomes

```
    x at %ebp+8, y at %ebp+12
1     movl    8(%ebp),%eax     Load x into %eax
2     cltd                     Sign extend into %edx
3     idivl   12(%ebp)         Divide by y
4     movl    %eax, 4(%esp)    Store x / y
5     movl    %edx, (%esp)     Store x % y
```

We can see that the first two instructions have the same overall effect as the first three instructions in our earlier code sequence. Different versions of GCC generate these two different ways of setting up the dividend for integer division.

Unsigned division makes use of the `divl` instruction. Typically register `%edx` is set to 0 beforehand.

Practice Problem 3.11

Modify the assembly code shown for signed division so that it computes the unsigned quotient and remainder of numbers x and y and stores the results on the stack.

Practice Problem 3.12

Consider the following C function prototype, where `num_t` is a data type declared using `typedef`:

```
void store_prod(num_t *dest, unsigned x, num_t y) {
    *dest = x*y;
}
```

GCC generates the following assembly code implementing the body of the computation:

```
    dest at %ebp+8, x at %ebp+12, y at %ebp+16
1     movl    12(%ebp), %eax
2     movl    20(%ebp), %ecx
3     imull   %eax, %ecx
4     mull    16(%ebp)
5     leal    (%ecx,%edx), %edx
6     movl    8(%ebp), %ecx
7     movl    %eax, (%ecx)
8     movl    %edx, 4(%ecx)
```

1. This instruction is called `cdq` in the Intel documentation, one of the few cases where the ATT-format name for an instruction bears no relation to the Intel name.

Observe that this code requires two memory reads to fetch argument `y` (lines 2 and 4), two multiplies (lines 3 and 4), and two memory writes to store the result (lines 7 and 8).

A. What data type is `num_t`?

B. Describe the algorithm used to compute the product and argue that it is correct.

3.6 Control

So far, we have only considered the behavior of *straight-line* code, where instructions follow one another in sequence. Some constructs in C, such as conditionals, loops, and switches, require conditional execution, where the sequence of operations that gets performed depends on the outcomes of tests applied to the data. Machine code provides two basic low-level mechanisms for implementing conditional behavior: it tests data values and then either alters the control flow or the data flow based on the result of these tests.

Data-dependent control flow is the more general and more common approach for implementing conditional behavior, and so we will examine this first. Normally, both statements in C and instructions in machine code are executed *sequentially*, in the order they appear in the program. The execution order of a set of machine-code instructions can be altered with a *jump* instruction, indicating that control should pass to some other part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon this low-level mechanism to implement the control constructs of C.

In our presentation, we first cover the machine-level mechanisms and then show how the different control constructs of C are implemented with them. We then return to the use of conditional data transfer to implement data-dependent behavior.

3.6.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. The most useful condition codes are:

`CF`: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero Flag. The most recent operation yielded zero.

`SF`: Sign Flag. The most recent operation yielded a negative value.

`OF`: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Figure 3.10 **Comparison and test instructions.** These instructions set the condition codes without updating any other registers.

For example, suppose we used one of the ADD instructions to perform the equivalent of the C assignment `t=a+b`, where variables `a`, `b`, and `t` are integers. Then the condition codes would be set according to the following C expressions:

CF:	`(unsigned) t < (unsigned) a`	Unsigned overflow
ZF:	`(t == 0)`	Zero
SF:	`(t < 0)`	Negative
OF:	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

The `leal` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 3.7 cause the condition codes to be set. For the logical operations, such as XOR, the carry and overflow flags are set to 0. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to 0. For reasons that we will not delve into, the INC and DEC instructions set the overflow and zero flags, but they leave the carry flag unchanged.

In addition to the setting of condition codes by the instructions of Figure 3.7, there are two instruction classes (having 8, 16, and 32-bit forms) that set condition codes without altering any other registers; these are listed in Figure 3.10. The CMP instructions set the condition codes according to the differences of their two operands. They behave in the same way as the SUB instructions, except that they set the condition codes without updating their destinations. With ATT format, the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands. The TEST instructions behave in the same manner as the AND instructions, except that they set the condition codes without altering their destinations. Typically, the same operand is repeated (e.g., `testl %eax,%eax` to see whether `%eax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

Instruction		Synonym	Effect	Set condition
`sete`	*D*	`setz`	*D* ← ZF	Equal / zero
`setne`	*D*	`setnz`	*D* ← ~ZF	Not equal / not zero
`sets`	*D*		*D* ← SF	Negative
`setns`	*D*		*D* ← ~SF	Nonnegative
`setg`	*D*	`setnle`	*D* ← ~(SF ^ OF) & ~ZF	Greater (signed >)
`setge`	*D*	`setnl`	*D* ← ~(SF ^ OF)	Greater or equal (signed >=)
`setl`	*D*	`setnge`	*D* ← SF ^ OF	Less (signed <)
`setle`	*D*	`setng`	*D* ← (SF ^ OF) | ZF	Less or equal (signed <=)
`seta`	*D*	`setnbe`	*D* ← ~CF & ~ZF	Above (unsigned >)
`setae`	*D*	`setnb`	*D* ← ~CF	Above or equal (unsigned >=)
`setb`	*D*	`setnae`	*D* ← CF	Below (unsigned <)
`setbe`	*D*	`setna`	*D* ← CF | ZF	Below or equal (unsigned <=)

Figure 3.11 **The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," i.e., alternate names for the same machine instruction.

3.6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, there are three common ways of using the condition codes: (1) we can set a single byte to 0 or 1 depending on some combination of the condition codes, (2) we can conditionally jump to some other part of the program, or (3) we can conditionally transfer data. For the first case, the instructions described in Figure 3.11 set a single byte to 0 or to 1 depending on some combination of the condition codes. We refer to this entire class of instructions as the SET instructions; they differ from one another based on which combinations of condition codes they consider, as indicated by the different suffixes for the instruction names. It is important to recognize that the suffixes for these instructions denote different conditions and not different operand sizes. For example, instructions `setl` and `setb` denote "set less" and "set below," not "set long word" or "set byte."

A SET instruction has either one of the eight single-byte register elements (Figure 3.2) or a single-byte memory location as its destination, setting this byte to either 0 or 1. To generate a 32-bit result, we must also clear the high-order 24 bits. A typical instruction sequence to compute the C expression `a < b`, where `a` and `b` are both of type `int`, proceeds as follows:

```
    a is in %edx, b is in %eax
1     cmpl    %eax, %edx    Compare a:b
2     setl    %al           Set low order byte of %eax to 0 or 1
3     movzbl  %al, %eax     Set remaining bytes of %eax to 0
```

The `movzbl` instruction clears the high-order 3 bytes of `%eax`.

For some of the underlying machine instructions, there are multiple possible names, which we list as "synonyms." For example, both `setg` (for "set greater") and `setnle` (for "set not less or equal") refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic and logical operations set the condition codes, the descriptions of the different SET instructions apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t = a-b`. More specifically, let a, b, and t be the integers represented in two's-complement form by variables `a`, `b`, and `t`, respectively, and so $t = a -_w^t b$, where w depends on the sizes associated with `a` and `b`.

Consider the `sete`, or "set when equal" instruction. When $a = b$, we will have $t = 0$, and hence the zero flag indicates equality. Similarly, consider testing for signed comparison with the `setl`, or "set when less," instruction. When no overflow occurs (indicated by having `OF` set to 0), we will have $a < b$ when $a -_w^t b < 0$, indicated by having `SF` set to 1, and $a \geq b$ when $a -_w^t b \geq 0$, indicated by having `SF` set to 0. On the other hand, when overflow occurs, we will have $a < b$ when $a -_w^t b > 0$ (positive overflow) and $a > b$ when $a -_w^t b < 0$ (negative overflow). We cannot have overflow when $a = b$. Thus, when `OF` is set to 1, we will have $a < b$ if and only if `SF` is set to 0. Combining these cases, the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether $a < b$. The other signed comparison tests are based on other combinations of `SF ^ OF` and `ZF`.

For the testing of unsigned comparisons, we now let a and b be the integers represented in unsigned form by variables `a` and `b`. In performing the computation `t = a-b`, the carry flag will be set by the CMP instruction when the $a - b < 0$, and so the unsigned comparisons use combinations of the carry and zero flags.

It is important to note how machine code distinguishes between signed and unsigned values. Unlike in C, it does not associate a data type with each program value. Instead, it mostly uses the same instructions for the two cases, because many arithmetic operations have the same bit-level behavior for unsigned and two's-complement arithmetic. Some circumstances require different instructions to handle signed and unsigned operations, such as using different versions of right shifts, division and multiplication instructions, and different combinations of condition codes.

Practice Problem 3.13

The following C code

```
int comp(data_t a, data_t b) {
    return a COMP b;
}
```

shows a general comparison between arguments `a` and `b`, where we can set the data type of the arguments by declaring `data_t` with a `typedef` declaration, and we can set the comparison by defining `COMP` with a `#define` declaration.

Suppose `a` is in `%edx` and `b` is in `%eax`. For each of the following instruction sequences, determine which data types `data_t` and which comparisons `COMP` could

cause the compiler to generate this code. (There can be multiple correct answers; you should list them all.)

A.
```
cmpl    %eax, %edx
setl    %al
```

B.
```
cmpw    %ax, %dx
setge   %al
```

C.
```
cmpb    %al, %dl
setb    %al
```

D.
```
cmpl    %eax, %edx
setne   %al
```

Practice Problem 3.14

The following C code

```
int test(data_t a) {
    return a TEST 0;
}
```

shows a general comparison between argument `a` and 0, where we can set the data type of the argument by declaring `data_t` with a `typedef`, and the nature of the comparison by declaring `TEST` with a `#define` declaration. For each of the following instruction sequences, determine which data types `data_t` and which comparisons `TEST` could cause the compiler to generate this code. (There can be multiple correct answers; list all correct ones.)

A.
```
testl   %eax, %eax
setne   %al
```

B.
```
testw   %ax, %ax
sete    %al
```

C.
```
testb   %al, %al
setg    %al
```

D.
```
testw   %ax, %ax
seta    %al
```

3.6.3 Jump Instructions and Their Encodings

Under normal execution, instructions follow each other in the order they are listed. A *jump* instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated in

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja`	*Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae`	*Label*	`jnb`	`~CF`	Above or equal (unsigned >=)
`jb`	*Label*	`jnae`	`CF`	Below (unsigned <)
`jbe`	*Label*	`jna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.12 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

assembly code by a *label*. Consider the following (very contrived) assembly-code sequence:

```
1      movl $0,%eax              Set %eax to 0
2      jmp .L1                   Goto .L1
3      movl (%eax),%edx          Null pointer dereference
4    .L1:
5      popl %edx
```

The instruction `jmp .L1` will cause the program to skip over the `movl` instruction and instead resume execution with the `popl` instruction. In generating the object-code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

Figure 3.12 shows the different jump instructions. The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly by giving a label as the jump target, e.g., the label "`.L1`" in the code shown. Indirect jumps are written using '`*`' followed by an operand specifier using one of the formats described in Section 3.4.1. As examples, the instruction

```
jmp *%eax
```

uses the value in register `%eax` as the jump target, and the instruction

```
jmp *(%eax)
```

reads the jump target from memory, using the value in %eax as the read address.

The remaining jump instructions in the table are *conditional*—they either jump or continue executing at the next instruction in the code sequence, depending on some combination of the condition codes. The names of these instructions and the conditions under which they jump match those of the SET instructions (see Figure 3.11). As with the SET instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

Although we will not concern ourselves with the detailed format of machine code, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter 7. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are *PC relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using 1, 2, or 4 bytes. A second encoding method is to give an "absolute" address, using 4 bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

As an example of PC-relative addressing, the following fragment of assembly code was generated by compiling a file `silly.c`. It contains two jumps: the `jle` instruction on line 1 jumps forward to a higher address, while the `jg` instruction on line 8 jumps back to a lower one.

```
1     jle     .L2              if <=, goto dest2
2   .L5:                     dest1:
3     movl    %edx, %eax
4     sarl    %eax
5     subl    %eax, %edx
6     leal    (%edx,%edx,2), %edx
7     testl   %edx, %edx
8     jg      .L5              if >, goto dest1
9   .L2:                     dest2:
10    movl    %edx, %eax
```

The disassembled version of the ".o" format generated by the assembler is as follows:

```
1      8:   7e 0d                 jle    17 <silly+0x17>   Target = dest2
2      a:   89 d0                 mov    %edx,%eax       dest1:
3      c:   d1 f8                 sar    %eax
4      e:   29 c2                 sub    %eax,%edx
5     10:   8d 14 52              lea    (%edx,%edx,2),%edx
6     13:   85 d2                 test   %edx,%edx
7     15:   7f f3                 jg     a <silly+0xa>     Target = dest1
8     17:   89 d0                 mov    %edx,%eax       dest2:
```

In the annotations generated by the disassembler on the right, the jump targets are indicated as `0x17` for the jump instruction on line 1 and `0xa` for the jump instruction on line 7. Looking at the byte encodings of the instructions, however, we see that the target of the first jump instruction is encoded (in the second byte) as `0xd` (decimal 13). Adding this to `0xa` (decimal 10), the address of the following instruction, we get jump target address `0x17` (decimal 23), the address of the instruction on line 8.

Similarly, the target of the second jump instruction is encoded as `0xf3` (decimal −13) using a single-byte, two's-complement representation. Adding this to `0x17` (decimal 23), the address of the instruction on line 8, we get `0xa` (decimal 10), the address of the instruction on line 2.

As these examples illustrate, the value of the program counter when performing PC-relative addressing is the address of the instruction following the jump, not that of the jump itself. This convention dates back to early implementations, when the processor would update the program counter as its first step in executing an instruction.

The following shows the disassembled version of the program after linking:

```
1   804839c:  7e 0d                 jle    80483ab <silly+0x17>
2   804839e:  89 d0                 mov    %edx,%eax
3   80483a0:  d1 f8                 sar    %eax
4   80483a2:  29 c2                 sub    %eax,%edx
5   80483a4:  8d 14 52              lea    (%edx,%edx,2),%edx
6   80483a7:  85 d2                 test   %edx,%edx
7   80483a9:  7f f3                 jg     804839e <silly+0xa>
8   80483ab:  89 d0                 mov    %edx,%eax
```

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 1 and 7 remain unchanged. By using a PC-relative encoding of the jump targets, the instructions can be compactly encoded (requiring just 2 bytes), and the object code can be shifted to different positions in memory without alteration.

Practice Problem 3.15

In the following excerpts from a disassembled binary, some of the information has been replaced by `X`s. Answer the following questions about these instructions.

A. What is the target of the `je` instruction below? (You don't need to know anything about the `call` instruction here.)

```
804828f:       74 05                   je     XXXXXXX
8048291:       e8 1e 00 00 00          call   80482b4
```

B. What is the target of the `jb` instruction below?

```
8048357:       72 e7                   jb     XXXXXXX
8048359:       c6 05 10 a0 04 08 01    movb   $0x1,0x804a010
```

C. What is the address of the `mov` instruction?

```
XXXXXXX:      74 12                  je     8048391
XXXXXXX:      b8 00 00 00 00         mov    $0x0,%eax
```

D. In the code that follows, the jump target is encoded in PC-relative form as a 4-byte, two's-complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of IA32. What is the address of the jump target?

```
80482bf:      e9 e0 ff ff ff         jmp    XXXXXXX
80482c4:      90                     nop
```

E. Explain the relation between the annotation on the right and the byte coding on the left.

```
80482aa:      ff 25 fc 9f 04 08      jmp    *0x8049ffc
```

To implement the control constructs of C via conditional control transfer, the compiler must use the different types of jump instructions we have just seen. We will go through the most common constructs, starting from simple conditional branches, and then consider loops and `switch` statements.

3.6.4 Translating Conditional Branches

The most general way to translate conditional expressions and statements from C into machine code is to use combinations of conditional and unconditional jumps. (As an alternative, we will see in Section 3.6.6 that some conditionals can be implemented by conditional transfers of data rather than control.) For example, Figure 3.13(a) shows the C code for a function that computes the absolute value of the difference of two numbers.[2] GCC generates the assembly code shown as Figure 3.13(c). We have created a version in C, called `gotodiff` (Figure 3.13(b)), that more closely follows the control flow of this assembly code. It uses the `goto` statement in C, which is similar to the unconditional jump of assembly code. The statement `goto x_ge_y` on line 4 causes a jump to the label `x_ge_y` (since it occurs when $x \geq y$) on line 7, skipping the computation of `y-x` on line 5. If the test fails, the program computes the result as `y-x` and then transfers unconditionally to the end of the code. Using `goto` statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of assembly-code programs. We call this style of programming "goto code."

The assembly-code implementation first compares the two operands (line 3), setting the condition codes. If the comparison result indicates that x is greater

2. Actually, it can return a negative value if one of the subtractions overflows. Our interest here is to demonstrate machine code, not to implement robust code.

(a) Original C code

```
1   int absdiff(int x, int y) {
2       if (x < y)
3           return y - x;
4       else
5           return x - y;
6   }
```

(b) Equivalent goto version

```
1   int gotodiff(int x, int y) {
2       int result;
3       if (x >= y)
4           goto x_ge_y;
5       result =  y - x;
6       goto done;
7    x_ge_y:
8       result = x - y;
9    done:
10      return result;
11  }
```

(c) Generated assembly code

```
    x at %ebp+8, y at %ebp+12
1     movl    8(%ebp), %edx     Get x
2     movl    12(%ebp), %eax    Get y
3     cmpl    %eax, %edx        Compare x:y
4     jge     .L2               if >= goto x_ge_y
5     subl    %edx, %eax        Compute result = y-x
6     jmp     .L3               Goto done
7   .L2:                      x_ge_y:
8     subl    %eax, %edx        Compute result = x-y
9     movl    %edx, %eax        Set result as return value
10  .L3:                      done: Begin completion code
```

Figure 3.13 **Compilation of conditional statements.** C procedure `absdiff` (part (a)) contains an if-else statement. The generated assembly code is shown (part (c)), along with a C procedure `gotodiff` (part (b)) that mimics the control flow of the assembly code. The stack set-up and completion portions of the assembly code have been omitted.

than or equal to y, it then jumps to a block of code that computes `x-y` (line 8). Otherwise, it continues with the execution of code that computes `y-x` (line 5). In both cases, the computed result is stored in register `%eax`, and the program reaches line 10, at which point it executes the stack completion code (not shown).

The general form of an if-else statement in C is given by the template

```
if (test-expr)
    then-statement
else
    else-statement
```

where *test-expr* is an integer expression that evaluates either to 0 (interpreted as meaning "false") or to a nonzero value (interpreted as meaning "true"). Only one of the two branch statements (*then-statement* or *else-statement*) is executed.

For this general form, the assembly implementation typically adheres to the following form, where we use C syntax to describe the control flow:

```
    t = test-expr;
    if (!t)
        goto false;
    then-statement
    goto done;
false:
    else-statement
done:
```

That is, the compiler generates separate blocks of code for *then-statement* and *else-statement*. It inserts conditional and unconditional branches to make sure the correct block is executed.

Practice Problem 3.16

When given the C code

```
1   void cond(int a, int *p)
2   {
3       if (p && a > 0)
4           *p += a;
5   }
```

GCC generates the following assembly code for the body of the function:

```
    a %ebp+8, p at %ebp+12
1     movl    8(%ebp), %edx
2     movl    12(%ebp), %eax
3     testl   %eax, %eax
4     je      .L3
5     testl   %edx, %edx
6     jle     .L3
7     addl    %edx, (%eax)
8   .L3:
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.13(b). You might find it helpful to first annotate the assembly code as we have done in our examples.

B. Explain why the assembly code contains two conditional branches, even though the C code has only one `if` statement.

Practice Problem 3.17

An alternate rule for translating `if` statements into goto code is as follows:

```
    t = test-expr;
    if (t)
        goto true;
    else-statement
    goto done;
true:
    then-statement
done:
```

A. Rewrite the goto version of `absdiff` based on this alternate rule.

B. Can you think of any reasons for choosing one rule over the other?

Practice Problem 3.18

Starting with C code of the form

```
1   int test(int x, int y) {
2       int val = __________;
3       if (__________) {
4           if (__________)
5               val = __________;
6           else
7               val = __________;
8       } else if (__________)
9           val = __________;
10      return val;
11  }
```

GCC generates the following assembly code:

```
    x at %ebp+8, y at %ebp+12
1     movl    8(%ebp), %eax
2     movl    12(%ebp), %edx
3     cmpl    $-3, %eax
4     jge     .L2
5     cmpl    %edx, %eax
6     jle     .L3
7     imull   %edx, %eax
8     jmp     .L4
9   .L3:
10    leal    (%edx,%eax), %eax
11    jmp     .L4
12  .L2:
```

```
13      cmpl    $2, %eax
14      jg      .L5
15      xorl    %edx, %eax
16      jmp     .L4
17    .L5:
18      subl    %edx, %eax
19    .L4:
```

Fill in the missing expressions in the C code. To make the code fit into the C code template, you will need to undo some of the reordering of computations done by GCC.

3.6.5 Loops

C provides several looping constructs—namely, `do-while`, `while`, and `for`. No corresponding instructions exist in machine code. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Most compilers generate loop code based on the `do-while` form of a loop, even though this form is relatively uncommon in actual programs. Other loops are transformed into `do-while` form and then compiled into machine code. We will study the translation of loops as a progression, starting with `do-while` and then working toward ones with more complex implementations.

Do-While Loops

The general form of a `do-while` statement is as follows:

```
do
   body-statement
   while (test-expr);
```

The effect of the loop is to repeatedly execute *body-statement*, evaluate *test-expr*, and continue the loop if the evaluation result is nonzero. Observe that *body-statement* is executed at least once.

This general form can be translated into conditionals and `goto` statements as follows:

```
loop:
   body-statement
   t = test-expr;
   if (t)
      goto loop;
```

That is, on each iteration the program evaluates the body statement and then the test expression. If the test succeeds, we go back for another iteration.

As an example, Figure 3.14(a) shows an implementation of a routine to compute the factorial of its argument, written $n!$, with a `do-while` loop. This function only computes the proper value for $n > 0$.

(a) C code

```
1   int fact_do(int n)
2   {
3       int result = 1;
4       do {
5           result *= n;
6           n = n-1;
7       } while (n > 1);
8       return result;
9   }
```

(c) Corresponding assembly-language code

```
    Argument: n at %ebp+8
    Registers: n in %edx, result in %eax
1     movl    8(%ebp), %edx    Get n
2     movl    $1, %eax         Set result = 1
3   .L2:                     loop:
4     imull   %edx, %eax       Compute result *= n
5     subl    $1, %edx         Decrement n
6     cmpl    $1, %edx         Compare n:1
7     jg      .L2              If >, goto loop
    Return result
```

(b) Register usage

Register	Variable	Initially
%eax	result	1
%edx	n	*n*

Figure 3.14 **Code for do-while version of factorial program.** The C code, the generated assembly code, and a table of register usage is shown.

Practice Problem 3.19

A. What is the maximum value of n for which we can represent $n!$ with a 32-bit `int`?

B. What about for a 64-bit `long long int`?

The assembly code shown in Figure 3.14(c) shows a standard implementation of a `do-while` loop. Following the initialization of register `%edx` to hold n and `%eax` to hold `result`, the program begins looping. It first executes the *body* of the loop, consisting here of the updates to variables `result` and n (lines 4–5). It then tests whether $n > 1$, and, if so, it jumps back to the beginning of the loop. We see here that the conditional jump (line 7) is the key instruction in implementing a loop. It determines whether to continue iterating or to exit the loop.

Determining which registers are used for which program values can be challenging, especially with loop code. We have shown such a mapping in Figure 3.14. In this case, the mapping is fairly simple to determine: we can see n getting loaded into register `%edx` on line 1, getting decremented on line 5, and being tested on line 6. We therefore conclude that this register holds n.

We can see register `%eax` getting initialized to 1 (line 2), and being updated by multiplication on line 4. Furthermore, since `%eax` is used to return the function value, it is often chosen to hold program values that are returned. We therefore conclude that `%eax` corresponds to program value `result`.

Aside Reverse engineering loops

A key to understanding how the generated assembly code relates to the original source code is to find a mapping between program values and registers. This task was simple enough for the loop of Figure 3.14, but it can be much more challenging for more complex programs. The C compiler will often rearrange the computations, so that some variables in the C code have no counterpart in the machine code, and new values are introduced into the machine code that do not exist in the source code. Moreover, it will often try to minimize register usage by mapping multiple program values onto a single register.

The process we described for `fact_do` works as a general strategy for reverse engineering loops. Look at how registers are initialized before the loop, updated and tested within the loop, and used after the loop. Each of these provides a clue that can be combined to solve a puzzle. Be prepared for surprising transformations, some of which are clearly cases where the compiler was able to optimize the code, and others where it is hard to explain why the compiler chose that particular strategy. In our experience, GCC often makes transformations that provide no performance benefit and can even decrease code performance.

Practice Problem 3.20

For the C code

```
1   int dw_loop(int x, int y, int n) {
2       do {
3           x += n;
4           y *= n;
5           n--;
6       } while ((n > 0) && (y < n));
7       return x;
8   }
```

GCC generates the following assembly code:

```
    x at %ebp+8, y at %ebp+12, n at %ebp+16
1     movl    8(%ebp), %eax
2     movl    12(%ebp), %ecx
3     movl    16(%ebp), %edx
4   .L2:
5     addl    %edx, %eax
6     imull   %edx, %ecx
7     subl    $1, %edx
8     testl   %edx, %edx
9     jle     .L5
10    cmpl    %edx, %ecx
11    jl      .L2
12  .L5:
```

A. Make a table of register usage, similar to the one shown in Figure 3.14(b).

B. Identify *test-expr* and *body-statement* in the C code, and the corresponding lines in the assembly code.

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.14(b).

While Loops

The general form of a `while` statement is as follows:

```
while (test-expr)
   body-statement
```

It differs from `do-while` in that *test-expr* is evaluated and the loop is potentially terminated before the first execution of *body-statement*. There are a number of ways to translate a `while` loop into machine code. One common approach, also used by GCC, is to transform the code into a `do-while` loop by using a conditional branch to skip the first execution of the body if needed:

```
if (!test-expr)
   goto done;
do
   body-statement
   while (test-expr);
done:
```

This, in turn, can be transformed into goto code as

```
   t = test-expr;
   if (!t)
      goto done;
loop:
   body-statement
   t = test-expr;
   if (t)
      goto loop;
done:
```

Using this implementation strategy, the compiler can often optimize the initial test, for example determining that the test condition will always hold.

As an example, Figure 3.15 shows an implementation of the factorial function using a `while` loop (Figure 3.15(a)). This function correctly computes $0! = 1$. The adjacent function `fact_while_goto` (Figure 3.15(b)) is a C rendition of the assembly code generated by GCC. Comparing the code generated for `fact_while` (Figure 3.15) to that for `fact_do` (Figure 3.14), we see that they are nearly identical. The only difference is the initial test (line 3) and the jump around the loop (line 4). The compiler closely followed our template for converting a `while` loop to a `do-while` loop, and for translating this loop to goto code.

(a) C code

```
1   int fact_while(int n)
2   {
3       int result = 1;
4       while (n > 1) {
5           result *= n;
6           n = n-1;
7       }
8       return result;
9   }
```

(b) Equivalent goto version

```
1   int fact_while_goto(int n)
2   {
3       int result = 1;
4       if (n <= 1)
5           goto done;
6    loop:
7       result *= n;
8       n = n-1;
9       if (n > 1)
10          goto loop;
11   done:
12      return result;
13  }
```

(c) Corresponding assembly-language code

```
    Argument: n at %ebp+8
    Registers: n in %edx, result in %eax
1     movl    8(%ebp), %edx    Get n
2     movl    $1, %eax         Set result = 1
3     cmpl    $1, %edx         Compare n:1
4     jle     .L7              If <=, goto done
5   .L10:                    loop:
6     imull   %edx, %eax       Compute result *= n
7     subl    $1, %edx         Decrement n
8     cmpl    $1, %edx         Compare n:1
9     jg      .L10             If >, goto loop
10  .L7:                     done:
     Return result
```

Figure 3.15 **C and assembly code for `while` version of factorial.** The `fact_while_goto` function illustrates the operation of the assembly code version.

Practice Problem 3.21

For the C code

```
1   int loop_while(int a, int b)
2   {
3       int result = 1;
4       while (a < b) {
5           result *= (a+b);
6           a++;
7       }
8       return result;
9   }
```

GCC generates the following assembly code:

```
    a at %ebp+8, b at %ebp+12
 1    movl    8(%ebp), %ecx
 2    movl    12(%ebp), %ebx
 3    movl    $1, %eax
 4    cmpl    %ebx, %ecx
 5    jge     .L11
 6    leal    (%ebx,%ecx), %edx
 7    movl    $1, %eax
 8  .L12:
 9    imull   %edx, %eax
10    addl    $1, %ecx
11    addl    $1, %edx
12    cmpl    %ecx, %ebx
13    jg      .L12
14  .L11:
```

In generating this code, GCC makes an interesting transformation that, in effect, introduces a new program variable.

A. Register %edx is initialized on line 6 and updated within the loop on line 11. Consider this to be a new program variable. Describe how it relates to the variables in the C code.

B. Create a table of register usage for this function.

C. Annotate the assembly code to describe how it operates.

D. Write a goto version of the function (in C) that mimics how the assembly code program operates.

Practice Problem 3.22

A function, fun_a, has the following overall structure:

```
int fun_a(unsigned x) {
    int val = 0;
    while ( _________ ) {
        _________;
    }
    return _________;
}
```

The GCC C compiler generates the following assembly code:

```
    x at %ebp+8
 1    movl    8(%ebp), %edx
 2    movl    $0, %eax
 3    testl   %edx, %edx
```

```
4       je      .L7
5     .L10:
6       xorl    %edx, %eax
7       shrl    %edx            Shift right by 1
8       jne     .L10
9     .L7:
10      andl    $1, %eax
```

Reverse engineer the operation of this code and then do the following:

A. Use the assembly-code version to fill in the missing parts of the C code.

B. Describe in English what this function computes.

For Loops

The general form of a `for` loop is as follows:

```
for (init-expr; test-expr; update-expr)
    body-statement
```

The C language standard states (with one exception, highlighted in Problem 3.24) that the behavior of such a loop is identical to the following code, which uses a `while` loop:

```
init-expr;
while (test-expr) {
    body-statement
    update-expr;
}
```

The program first evaluates the initialization expression *init-expr*. It enters a loop where it first evaluates the test condition *test-expr*, exiting if the test fails, then executes the body of the loop *body-statement*, and finally evaluates the update expression *update-expr*.

The compiled form of this code is based on the transformation from `while` to `do-while` described previously, first giving a `do-while` form:

```
init-expr;
if (!test-expr)
    goto done;
do {
    body-statement
    update-expr;
} while (test-expr);
done:
```

This, in turn, can be transformed into goto code as

```
   init-expr;
   t = test-expr;
   if (!t)
      goto done;
loop:
   body-statement
   update-expr;
   t = test-expr;
   if (t)
      goto loop;
done:
```

As an example, consider a factorial function written with a `for` loop:

```
1  int fact_for(int n)
2  {
3      int i;
4      int result = 1;
5      for (i = 2; i <= n; i++)
6          result *= i;
7      return result;
8  }
```

As shown, the natural way of writing a factorial function with a `for` loop is to multiply factors from 2 up to n, and so this function is quite different from the code we showed using either a `while` or a `do-while` loop.

We can identify the different components of the `for` loop in this code as follows:

init-expr	`i = 2`
test-expr	`i <= n`
update-expr	`i++`
body-statement	`result *= i;`

Substituting these components into the template we have shown yields the following version in goto code:

```
1  int fact_for_goto(int n)
2  {
3      int i = 2;
4      int result = 1;
5      if (!(i <= n))
6          goto done;
7   loop:
8      result *= i;
9      i++;
```

```
10        if (i <= n)
11            goto loop;
12     done:
13        return result;
14    }
```

Indeed, a close examination of the assembly code produced by GCC closely follows this template:

```
    Argument: n at %ebp+8
    Registers: n in %ecx, i in %edx, result in %eax
 1     movl    8(%ebp), %ecx    Get n
 2     movl    $2, %edx         Set i to 2          (init)
 3     movl    $1, %eax         Set result to 1
 4     cmpl    $1, %ecx         Compare n:1         (!test)
 5     jle     .L14             If <=, goto done
 6    .L17:                    loop:
 7     imull   %edx, %eax       Compute result *= i (body)
 8     addl    $1, %edx         Increment i         (update)
 9     cmpl    %edx, %ecx       Compare n:i         (test)
10     jge     .L17             If >=, goto loop
11    .L14:                    done:
```

We see from this presentation that all three forms of loops in C—`do-while`, `while`, and `for`—can be translated by a single strategy, generating code that contains one or more conditional branches. Conditional transfer of control provides the basic mechanism for translating loops into machine code.

Practice Problem 3.23

A function `fun_b` has the following overall structure:

```
int fun_b(unsigned x) {
    int val = 0;
    int i;
    for ( _________ ; _________ ; _________ ) {
        _________
    }
    return val;
}
```

The GCC C compiler generates the following assembly code:

```
    x at %ebp+8
 1     movl    8(%ebp), %ebx
 2     movl    $0, %eax
 3     movl    $0, %ecx
 4    .L13:
```

```
 5      leal    (%eax,%eax), %edx
 6      movl    %ebx, %eax
 7      andl    $1, %eax
 8      orl     %edx, %eax
 9      shrl    %ebx              Shift right by 1
10      addl    $1, %ecx
11      cmpl    $32, %ecx
12      jne     .L13
```

Reverse engineer the operation of this code and then do the following:

A. Use the assembly-code version to fill in the missing parts of the C code.

B. Describe in English what this function computes.

Practice Problem 3.24

Executing a `continue` statement in C causes the program to jump to the end of the current loop iteration. The stated rule for translating a `for` loop into a `while` loop needs some refinement when dealing with `continue` statements. For example, consider the following code:

```
/* Example of for loop using a continue statement */
/* Sum even numbers between 0 and 9 */
int sum = 0;
int i;
for (i = 0; i < 10; i++) {
    if (i & 1)
        continue;
    sum += i;
}
```

A. What would we get if we naively applied our rule for translating the `for` loop into a `while` loop? What would be wrong with this code?

B. How could you replace the `continue` statement with a `goto` statement to ensure that the `while` loop correctly duplicates the behavior of the `for` loop?

3.6.6 Conditional Move Instructions

The conventional way to implement conditional operations is through a conditional transfer of *control*, where the program follows one execution path when a condition holds and another when it does not. This mechanism is simple and general, but it can be very inefficient on modern processors.

An alternate strategy is through a conditional transfer of *data*. This approach computes both outcomes of a conditional operation, and then selects one based on whether or not the condition holds. This strategy makes sense only in restricted cases, but it can then be implemented by a simple *conditional move* instruction that is better matched to the performance characteristics of modern processors.

We will examine this strategy and its implementation with more recent versions of IA32 processors.

Starting with the PentiumPro in 1995, recent generations of IA32 processors have had conditional move instructions that either do nothing or copy a value to a register, depending on the values of the condition codes. For years, these instructions have been largely unused. With its default settings, GCC did not generate code that used them, because that would prevent backward compatibility, even though almost all x86 processors manufactured by Intel and its competitors since 1997 have supported these instructions. More recently, for systems running on processors that are certain to support conditional moves, such as Intel-based Apple Macintosh computers (introduced in 2006) and the 64-bit versions of Linux and Windows, GCC will generate code using conditional moves. By giving special command-line parameters on other machines, we can indicate to GCC that the target machine supports conditional move instructions.

As an example, Figure 3.16(a) shows a variant form of the function `absdiff` we used in Figure 3.13 to illustrate conditional branching. This version uses a conditional *expression* rather than a conditional *statement* to illustrate the concepts behind conditional data transfers more clearly, but in fact GCC

(a) Original C code

```
1   int absdiff(int x, int y) {
2       return x < y ? y-x : x-y;
3   }
```

(b) Implementation using conditional assignment

```
1   int cmovdiff(int x, int y) {
2       int tval = y-x;
3       int rval = x-y;
4       int test = x < y;
5       /* Line below requires
6          single instruction: */
7       if (test) rval = tval;
8       return rval;
9   }
```

(c) Generated assembly code

```
    x at %ebp+8, y at %ebp+12
1     movl    8(%ebp), %ecx     Get x
2     movl    12(%ebp), %edx    Get y
3     movl    %edx, %ebx        Copy y
4     subl    %ecx, %ebx        Compute y-x
5     movl    %ecx, %eax        Copy x
6     subl    %edx, %eax        Compute x-y and set as return value
7     cmpl    %edx, %ecx        Compare x:y
8     cmovl   %ebx, %eax        If <, replace return value with y-x
```

Figure 3.16 **Compilation of conditional statements using conditional assignment.** C function `absdiff` (a) contains a conditional expression. The generated assembly code is shown (c), along with a C function `cmovdiff` (b) that mimics the operation of the assembly code. The stack set-up and completion portions of the assembly code have been omitted.

generates identical code for this version as it does for the version of Figure 3.13. If we compile this giving GCC the command-line option '`-march=i686`',[3] we generate the assembly code shown in Figure 3.16(c), having an approximate form shown by the C function `cmovdiff` shown in Figure 3.16(b). Studying the C version, we can see that it computes both `y-x` and `x-y`, naming these `tval` and `rval`, respectively. It then tests whether x is less than y, and if so, copies `tval` to `rval` before returning `rval`. The assembly code in Figure 3.16(c) follows the same logic. The key is that the single `cmovl` instruction (line 8) of the assembly code implements the conditional assignment (line 7) of `cmovdiff`. This instruction has the same syntax as a MOV instruction, except that it only performs the data movement if the specified condition holds. (The suffix '`l`' in `cmovl` stands for "less," not for "long.")

To understand why code based on conditional data transfers can outperform code based on conditional control transfers (as in Figure 3.13), we must understand something about how modern processors operate. As we will see in Chapters 4 and 5, processors achieve high performance through *pipelining*, where an instruction is processed via a sequence of stages, each performing one small portion of the required operations (e.g., fetching the instruction from memory, determining the instruction type, reading from memory, performing an arithmetic operation, writing to memory, and updating the program counter.) This approach achieves high performance by overlapping the steps of the successive instructions, such as fetching one instruction while performing the arithmetic operations for a previous instruction. To do this requires being able to determine the sequence of instructions to be executed well ahead of time in order to keep the pipeline full of instructions to be executed. When the machine encounters a conditional jump (referred to as a "branch"), it often cannot determine yet whether or not the jump will be followed. Processors employ sophisticated *branch prediction logic* to try to guess whether or not each jump instruction will be followed. As long as it can guess reliably (modern microprocessor designs try to achieve success rates on the order of 90%), the instruction pipeline will be kept full of instructions. Mispredicting a jump, on the other hand, requires that the processor discard much of the work it has already done on future instructions and then begin filling the pipeline with instructions starting at the correct location. As we will see, such a misprediction can incur a serious penalty, say, 20–40 clock cycles of wasted effort, causing a serious degradation of program performance.

As an example, we ran timings of the `absdiff` function on an Intel Core i7 processor using both methods of implementing the conditional operation. In a typical application, the outcome of the test `x < y` is highly unpredictable, and so even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time. In addition, the computations performed in each of the two code sequences require only a single clock cycle. As a consequence, the branch misprediction penalty dominates the performance of this function. For the IA32 code with conditional jumps, we found that the function requires around 13 clock

3. In GCC terminology, the Pentium should be considered model "586" and the PentiumPro should be considered model "686" of the x86 line.

cycles per call when the branching pattern is easily predictable, and around 35 clock cycles per call when the branching pattern is random. From this we can infer that the branch misprediction penalty is around 44 clock cycles. That means time required by the function ranges between around 13 and 57 cycles, depending on whether or not the branch is predicted correctly.

Aside How did you determine this penalty?

Assume the probability of misprediction is p, the time to execute the code without misprediction is T_{OK}, and the misprediction penalty is T_{MP}. Then the average time to execute the code as a function of p is $T_{avg}(p) = (1 - p)T_{OK} + p(T_{OK} + T_{MP}) = T_{OK} + pT_{MP}$. We are given T_{OK} and T_{ran}, the average time when $p = 0.5$, and we want to determine T_{MP}. Substituting into the equation, we get $T_{ran} = T_{avg}(0.5) = T_{OK} + 0.5T_{MP}$, and therefore $T_{MP} = 2(T_{ran} - T_{MP})$. So, for T_{OK} =13 and T_{ran} =35, we get T_{MP} =44.

On the other hand, the code compiled using conditional moves requires around 14 clock cycles regardless of the data being tested. The flow of control does not depend on data, and this makes it easier for the processor to keep its pipeline full.

Practice Problem 3.25

Running on a Pentium 4, our code required around 16 cycles when the branching pattern was highly predictable, and around 31 cycles when the pattern was random.

A. What is the approximate miss penalty?

B. How many cycles would the function require when the branch is mispredicted?

Figure 3.17 illustrates some of the conditional move instructions added to the IA32 instruction set with the introduction of the PentiumPro microprocessor and supported by most IA32 processors manufactured by Intel and its competitors since 1997. Each of these instructions has two operands: a source register or memory location S, and a destination register R. As with the different SET (Section 3.6.2) and jump instructions (Section 3.6.3), the outcome of these instructions depends on the values of the condition codes. The source value is read from either memory or the source register, but it is copied to the destination only if the specified condition holds.

For IA32, the source and destination values can be 16 or 32 bits long. Single-byte conditional moves are not supported. Unlike the unconditional instructions, where the operand length is explicitly encoded in the instruction name (e.g., `movw` and `movl`), the assembler can infer the operand length of a conditional move instruction from the name of the destination register, and so the same instruction name can be used for all operand lengths.

Unlike conditional jumps, the processor can execute conditional move instructions without having to predict the outcome of the test. The processor simply

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

Figure 3.17 **The conditional move instructions.** These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

reads the source value (possibly from memory), checks the condition code, and then either updates the destination register or keeps it the same. We will explore the implementation of conditional moves in Chapter 4.

To understand how conditional operations can be implemented via conditional data transfers, consider the following general form of conditional expression and assignment:

```
v = test-expr ? then-expr : else-expr;
```

With traditional IA32, the compiler generates code having a form shown by the following abstract code:

```
      if (!test-expr)
         goto false;
      v = true-expr;
      goto done;
   false:
      v = else-expr;
done:
```

This code contains two code sequences—one evaluating *then-expr* and one evaluating *else-expr*. A combination of conditional and unconditional jumps is used to ensure that just one of the sequences is evaluated.

For the code based on conditional move, both the *then-expr* and the *else-expr* are evaluated, with the final value chosen based on the evaluation *test-expr*. This can be described by the following abstract code:

```
vt = then-expr;
v  = else-expr;
t  = test-expr;
if (t) v = vt;
```

The final statement in this sequence is implemented with a conditional move—value `vt` is copied to `v` only if test condition `t` holds.

Not all conditional expressions can be compiled using conditional moves. Most significantly, the abstract code we have shown evaluates both *then-expr* and *else-expr* regardless of the test outcome. If one of those two expressions could possibly generate an error condition or a side effect, this could lead to invalid behavior. As an illustration, consider the following C function:

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

At first, this seems like a good candidate to compile using a conditional move to read the value designated by pointer `xp`, as shown in the following assembly code:

```
    Invalid implementation of function cread
    xp in register %edx
1     movl    $0, %eax          Set 0 as return value
2     testl   %edx, %edx        Test xp
3     cmovne  (%edx), %eax      if !0, dereference xp to get return value
```

This implementation is invalid, however, since the dereferencing of `xp` by the `cmovne` instruction (line 3) occurs even when the test fails, causing a null pointer dereferencing error. Instead, this code must be compiled using branching code.

A similar case holds when either of the two branches causes a side effect, as illustrated by the following function:

```
1   /* Global variable */
2   int lcount = 0;
3   int absdiff_se(int x, int y) {
4       return x < y ? (lcount++, y-x) : x-y;
5   }
```

This function increments global variable `lcount` as part of *then-expr*. Thus, branching code must be used to ensure this side effect only occurs when the test condition holds.

Using conditional moves also does not always improve code efficiency. For example, if either the *then-expr* or the *else-expr* evaluation requires a significant

computation, then this effort is wasted when the corresponding condition does not hold. Compilers must take into account the relative performance of wasted computation versus the potential for performance penalty due to branch misprediction. In truth, they do not really have enough information to make this decision reliably; for example, they do not know how well the branches will follow predictable patterns. Our experiments with GCC indicate that it only uses conditional moves when the two expressions can be computed very easily, for example, with single add instructions. In our experience, GCC uses conditional control transfers even in many cases where the cost of branch misprediction would exceed even more complex computations.

Overall, then, we see that conditional data transfers offer an alternative strategy to conditional control transfers for implementing conditional operations. They can only be used in restricted cases, but these cases are fairly common and provide a much better match to the operation of modern processors.

Practice Problem 3.26

In the following C function, we have left the definition of operation OP incomplete:

```
#define OP _________ /* Unknown operator */

int arith(int x) {
    return x OP 4;
}
```

When compiled, GCC generates the following assembly code:

```
   Register: x in %edx
1     leal    3(%edx), %eax
2     testl   %edx, %edx
3     cmovns  %edx, %eax
4     sarl    $2, %eax        Return value in %eax
```

A. What operation is OP?

B. Annotate the code to explain how it works.

Practice Problem 3.27

Starting with C code of the form

```
1   int test(int x, int y) {
2       int val = _________;
3       if (_________) {
4           if (_________)
5               val = _________;
6           else
7               val = _________;
```

```
8       } else if (_________)
9           val = _________;
10      return val;
11  }
```

GCC, with the command-line setting '`-march=i686`', generates the following assembly code:

```
    x at %ebp+8, y at %ebp+12
1     movl    8(%ebp), %ebx
2     movl    12(%ebp), %ecx
3     testl   %ecx, %ecx
4     jle     .L2
5     movl    %ebx, %edx
6     subl    %ecx, %edx
7     movl    %ecx, %eax
8     xorl    %ebx, %eax
9     cmpl    %ecx, %ebx
10    cmovl   %edx, %eax
11    jmp     .L4
12  .L2:
13    leal    0(,%ebx,4), %edx
14    leal    (%ecx,%ebx), %eax
15    cmpl    $-2, %ecx
16    cmovge  %edx, %eax
17  .L4:
```

Fill in the missing expressions in the C code.

3.6.7 Switch Statements

A `switch` statement provides a multi-way branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, they also allow an efficient implementation using a data structure called a *jump table*. A jump table is an array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i. The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction. The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases. GCC selects the method of translating a `switch` statement based on the number of cases and the sparsity of the case values. Jump tables are used when there are a number of cases (e.g., four or more) and they span a small range of values.

Figure 3.18(a) shows an example of a C `switch` statement. This example has a number of interesting features, including case labels that do not span a contiguous

(a) Switch statement

```
int switch_eg(int x, int n) {
    int result = x;

    switch (n) {

    case 100:
        result *= 13;
        break;

    case 102:
        result += 10;
        /* Fall through */

    case 103:
        result += 11;
        break;

    case 104:
    case 106:
        result *= result;
        break;

    default:
        result = 0;
    }

    return result;
}
```

(b) Translation into extended C

```
int switch_eg_impl(int x, int n) {
    /* Table of code pointers */
    static void *jt[7] = {
        &&loc_A, &&loc_def, &&loc_B,
        &&loc_C, &&loc_D, &&loc_def,
        &&loc_D
    };

    unsigned index = n - 100;
    int result;

    if (index > 6)
        goto loc_def;

    /* Multiway branch */
    goto *jt[index];

 loc_def:  /* Default case*/
    result = 0;
    goto done;

 loc_C:    /* Case 103 */
    result = x;
    goto rest;

 loc_A:    /* Case 100 */
    result = x * 13;
    goto done;

 loc_B:    /* Case 102 */
    result = x + 10;
    /* Fall through */

 rest:     /* Finish case 103 */
    result += 11;
    goto done;

 loc_D:    /* Cases 104, 106 */
    result = x * x;
    /* Fall through */

 done:
    return result;
}
```

Figure 3.18 **Switch statement example with translation into extended C.** The translation shows the structure of jump table `jt` and how it is accessed. Such tables are supported by GCC as an extension to the C language.

```
   x at %ebp+8, n at %ebp+12
1     movl    8(%ebp), %edx              Get x
2     movl    12(%ebp), %eax             Get n
      Set up jump table access
3     subl    $100, %eax                 Compute index = n-100
4     cmpl    $6, %eax                   Compare index:6
5     ja      .L2                        If >, goto loc_def
6     jmp     *.L7(,%eax,4)              Goto *jt[index]
      Default case
7   .L2:                                 loc_def:
8     movl    $0, %eax                     result = 0;
9     jmp     .L8                          Goto done
      Case 103
10  .L5:                                 loc_C:
11    movl    %edx, %eax                   result = x;
12    jmp     .L9                          Goto rest
      Case 100
13  .L3:                                 loc_A:
14    leal    (%edx,%edx,2), %eax          result = x*3;
15    leal    (%edx,%eax,4), %eax          result = x+4*result
16    jmp     .L8                          Goto done
      Case 102
17  .L4:                                 loc_B:
18    leal    10(%edx), %eax               result = x+10
      Fall through
19  .L9:                                 rest:
20    addl    $11, %eax                    result += 11;
21    jmp     .L8                          Goto done
      Cases 104, 106
22  .L6:                                 loc_D
23    movl    %edx, %eax                   result = x
24    imull   %edx, %eax                   result *= x
      Fall through
25  .L8:                                 done:
      Return result
```

Figure 3.19 **Assembly code for `switch` statement example in Figure 3.18.**

range (there are no labels for cases 101 and 105), cases with multiple labels (cases 104 and 106), and cases that *fall through* to other cases (case 102) because the code for the case does not end with a `break` statement.

Figure 3.19 shows the assembly code generated when compiling `switch_eg`. The behavior of this code is shown in C as the procedure `switch_eg_impl` in Figure 3.18(b). This code makes use of support provided by GCC for jump tables,

as an extension to the C language. The array `jt` contains seven entries, each of which is the address of a block of code. These locations are defined by labels in the code, and indicated in the entries in `jt` by code pointers, consisting of the labels prefixed by '`&&`.' (Recall that the operator `&` creates a pointer for a data value. In making this extension, the authors of GCC created a new operator `&&` to create a pointer for a code location.) We recommend that you study the C procedure `switch_eg_impl` and how it relates assembly code version.

Our original C code has cases for values 100, 102–104, and 106, but the switch variable `n` can be an arbitrary `int`. The compiler first shifts the range to between 0 and 6 by subtracting 100 from `n`, creating a new program variable that we call `index` in our C version. It further simplifies the branching possibilities by treating `index` as an *unsigned* value, making use of the fact that negative numbers in a two's-complement representation map to large positive numbers in an unsigned representation. It can therefore test whether `index` is outside of the range 0–6 by testing whether it is greater than 6. In the C and assembly code, there are five distinct locations to jump to, based on the value of `index`. These are: `loc_A` (identified in the assembly code as `.L3`), `loc_B` (`.L4`), `loc_C` (`.L5`), `loc_D` (`.L6`), and `loc_def` (`.L2`), where the latter is the destination for the default case. Each of these labels identifies a block of code implementing one of the case branches. In both the C and the assembly code, the program compares `index` to 6 and jumps to the code for the default case if it is greater.

The key step in executing a `switch` statement is to access a code location through the jump table. This occurs in line 16 in the C code, with a `goto` statement that references the jump table `jt`. This *computed goto* is supported by GCC as an extension to the C language. In our assembly-code version, a similar operation occurs on line 6, where the `jmp` instruction's operand is prefixed with '`*`', indicating an indirect jump, and the operand specifies a memory location indexed by register `%eax`, which holds the value of `index`. (We will see in Section 3.8 how array references are translated into machine code.)

Our C code declares the jump table as an array of seven elements, each of which is a pointer to a code location. These elements span values 0–6 of `index`, corresponding to values 100–106 of `n`. Observe the jump table handles duplicate cases by simply having the same code label (`loc_D`) for entries 4 and 6, and it handles missing cases by using the label for the default case (`loc_def`) as entries 1 and 5.

In the assembly code, the jump table is indicated by the following declarations, to which we have added comments:

```
1      .section         .rodata
2      .align 4         Align address to multiple of 4
3    .L7:
4      .long    .L3     Case 100: loc_A
5      .long    .L2     Case 101: loc_def
6      .long    .L4     Case 102: loc_B
7      .long    .L5     Case 103: loc_C
```

```
 8      .long   .L6      Case 104: loc_D
 9      .long   .L2      Case 105: loc_def
10      .long   .L6      Case 106: loc_D
```

These declarations state that within the segment of the object-code file called "`.rodata`" (for "Read-Only Data"), there should be a sequence of seven "long" (4-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly code labels (e.g., `.L3`). Label `.L7` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (line 6).

The different code blocks (C labels `loc_A` through `loc_D` and `loc_def`) implement the different branches of the `switch` statement. Most of them simply compute a value for `result` and then go to the end of the function. Similarly, the assembly-code blocks compute a value for register `%eax` and jump to the position indicated by label `.L8` at the end of the function. Only the code for case labels 102 and 103 do not follow this pattern, to account for the way that case 102 falls through to 103 in the original C code. This is handled in the assembly code and `switch_eg_impl` by having separate destinations for the two cases (`loc_C` and `loc_B` in C, `.L5` and `.L4` in assembly), where both of these blocks then converge on code that increments `result` by 11 (labeled `rest` in C and `.L9` in assembly).

Examining all of this code requires careful study, but the key point is to see that the use of a jump table allows a very efficient way to implement a multiway branch. In our case, the program could branch to five distinct locations with a single jump table reference. Even if we had a `switch` statement with hundreds of cases, they could be handled by a single jump table access.

Practice Problem 3.28

In the C function that follows, we have omitted the body of the `switch` statement. In the C code, the case labels did not span a contiguous range, and some cases had multiple labels.

```
int switch2(int x) {
    int result = 0;
    switch (x) {
        /* Body of switch statement omitted */
    }
    return result;
}
```

In compiling the function, GCC generates the assembly code that follows for the initial part of the procedure and for the jump table. Variable `x` is initially at offset 8 relative to register `%ebp`.

```
   x at %ebp+8
1     movl    8(%ebp), %eax
   Set up jump table access
2     addl    $2, %eax
3     cmpl    $6, %eax
4     ja      .L2
5     jmp     *.L8(,%eax,4)
```

```
   Jump table for switch2
1   .L8:
2     .long   .L3
3     .long   .L2
4     .long   .L4
5     .long   .L5
6     .long   .L6
7     .long   .L6
8     .long   .L7
```

Based on this information, answer the following questions:

A. What were the values of the case labels in the `switch` statement body?

B. What cases had multiple labels in the C code?

Practice Problem 3.29

For a C function `switcher` with the general structure

```
1   int switcher(int a, int b, int c)
2   {
3       int answer;
4       switch(a) {
5       case __________:       /* Case A */
6           c = __________;
7           /* Fall through */
8       case __________:       /* Case B */
9           answer = __________;
10          break;
11      case __________:       /* Case C */
12      case __________:       /* Case D */
13          answer = __________;
14          break;
15      case __________:       /* Case E */
16          answer = __________;
17          break;
18      default:
19          answer = __________;
20      }
21      return answer;
22  }
```

GCC generates the assembly code and jump table shown in Figure 3.20.

Fill in the missing parts of the C code. Except for the ordering of case labels `C` and `D`, there is only one way to fit the different cases into the template.

```
    a at %ebp+8, b at %ebp+12, c at %ebp+16
1     movl    8(%ebp), %eax
2     cmpl    $7, %eax
3     ja      .L2
4     jmp     *.L7(,%eax,4)
5   .L2:
6     movl    12(%ebp), %eax
7     jmp     .L8
8   .L5:
9     movl    $4, %eax
10    jmp     .L8
11  .L6:
12    movl    12(%ebp), %eax
13    xorl    $15, %eax
14    movl    %eax, 16(%ebp)
15  .L3:
16    movl    16(%ebp), %eax
17    addl    $112, %eax
18    jmp     .L8
19  .L4:
20    movl    16(%ebp), %eax
21    addl    12(%ebp), %eax
22    sall    $2, %eax
23  .L8:
```

```
1   .L7:
2     .long   .L3
3     .long   .L2
4     .long   .L4
5     .long   .L2
6     .long   .L5
7     .long   .L6
8     .long   .L2
9     .long   .L4
```

Figure 3.20 **Assembly code and jump table for Problem 3.29.**

3.7 Procedures

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of a program to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

3.7.1 Stack Frame Structure

IA32 programs make use of the program stack to support procedure calls. The machine uses the stack to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a *stack frame*. Figure 3.21 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register `%ebp` serving as the *frame pointer*, and register `%esp`

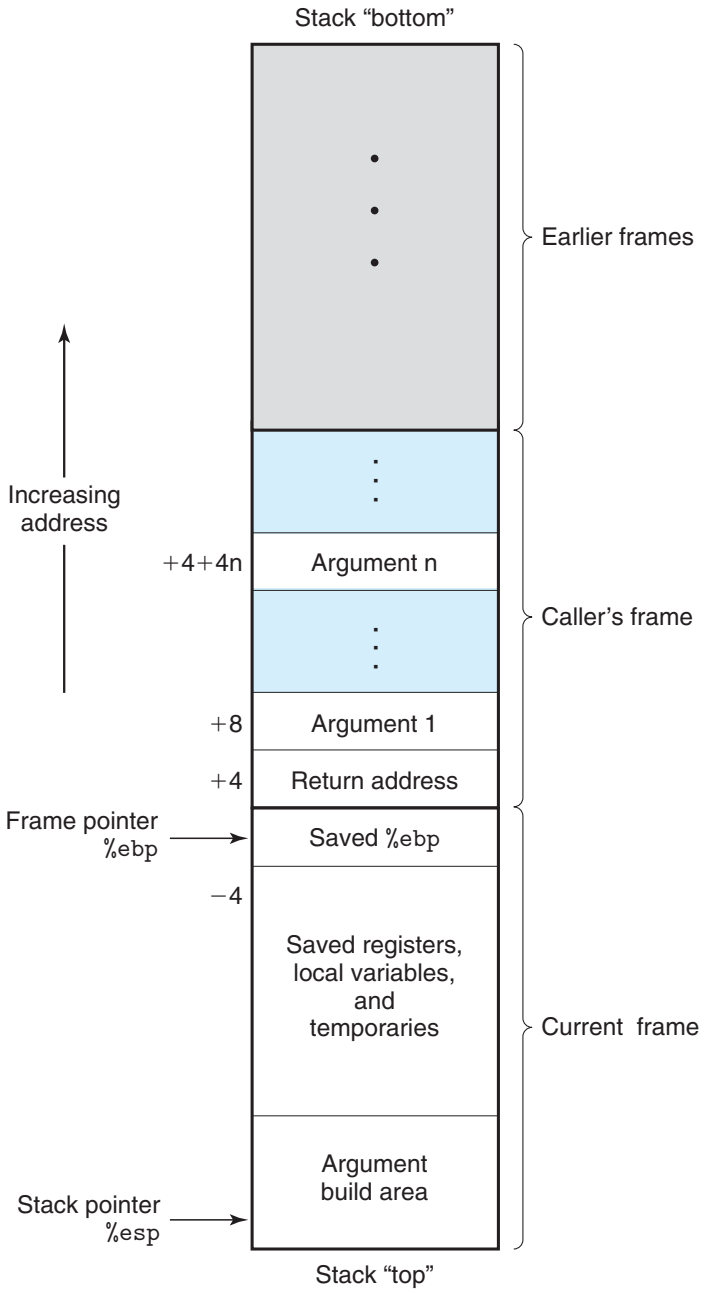


Figure 3.21
Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

serving as the *stack pointer*. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

Suppose procedure `P` (the *caller*) calls procedure `Q` (the *callee*). The arguments to `Q` are contained within the stack frame for `P`. In addition, when `P` calls `Q`, the *return address* within `P` where the program should resume execution when it returns from `Q` is pushed onto the stack, forming the end of `P`'s stack frame. The stack frame for `Q` starts with the saved value of the frame pointer (a copy of register `%ebp`), followed by copies of any other saved register values.

Procedure `Q` also uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:

- There are not enough registers to hold all of the local data.
- Some of the local variables are arrays or structures and hence must be accessed by array or structure references.
- The address operator '`&`' is applied to a local variable, and hence we must be able to generate an address for it.

In addition, `Q` uses the stack frame for storing arguments to any procedures it calls. As illustrated in Figure 3.21, within the called procedure, the first argument is positioned at offset 8 relative to `%ebp`, and the remaining arguments (assuming their data types require no more than 4 bytes) are stored in successive 4-byte blocks, so that argument i is at offset $4 + 4i$ relative to `%ebp`. Larger arguments (such as structures and larger numeric formats) require larger regions on the stack.

As described earlier, the stack grows toward lower addresses and the stack pointer `%esp` points to the top element of the stack. Data can be stored on and retrieved from the stack using the `pushl` and `popl` instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated by incrementing the stack pointer.

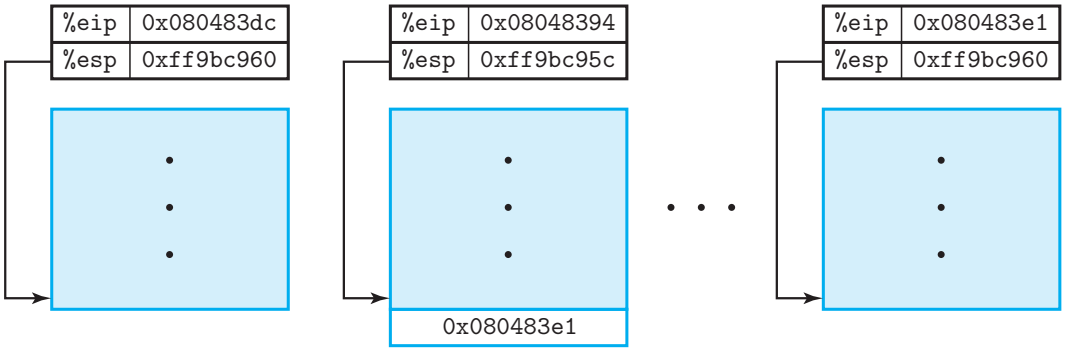
3.7.2 Transferring Control

The instructions supporting procedure calls and returns are shown in the following table:

Instruction		Description
`call`	*Label*	Procedure call
`call`	`*`*Operand*	Procedure call
`leave`		Prepare stack for return
`ret`		Return from call

The `call` instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can either be direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by a `*` followed by an operand specifier using one of the formats described in Section 3.4.1.

The effect of a `call` instruction is to push a return address on the stack and jump to the start of the called procedure. The return address is the address of the instruction immediately following the `call` in the program, so that execution will resume at this location when the called procedure returns. The `ret` instruction pops an address off the stack and jumps to this location. The proper use of this instruction is to have prepared the stack so that the stack pointer points to the place where the preceding `call` instruction stored its return address.



(a) Executing `call` (b) After `call` (c) After `ret`

Figure 3.22 **Illustration of `call` and `ret` functions.** The `call` instruction transfers control to the start of a function, while the `ret` instruction returns back to the instruction following the call.

Figure 3.22 illustrates the execution of the `call` and `ret` instructions for the `sum` and `main` functions introduced in Section 3.2.2. The following are excerpts of the disassembled code for the two functions:

```
    Beginning of function sum
1   08048394 <sum>:
2    8048394:  55                      push   %ebp
    . . .
    Return from function sum
3    80483a4:  c3                      ret
    . . .
    Call to sum from main
4    80483dc:  e8 b3 ff ff ff          call   8048394 <sum>
5    80483e1:  83 c4 14                add    $0x14,%esp
```

In this code, we can see that the `call` instruction with address `0x080483dc` in `main` calls function `sum`. This status is shown in Figure 3.22(a), with the indicated values for the stack pointer `%esp` and the program counter `%eip`. The effect of the `call` is to push the return address `0x080483e1` onto the stack and to jump to the first instruction in function `sum`, at address `0x08048394` (Figure 3.22(b)). The execution of function `sum` continues until it hits the `ret` instruction at address `0x080483a4`. This instruction pops the value `0x080483e1` from the stack and jumps to this address, resuming the execution of `main` just after the `call` instruction in `sum` (Figure 3.22(c)).

The `leave` instruction can be used to prepare the stack for returning. It is equivalent to the following code sequence:

```
1   movl %ebp, %esp  Set stack pointer to beginning of frame
2   popl %ebp        Restore saved %ebp and set stack ptr to end of caller's frame
```

Alternatively, this preparation can be performed by an explicit sequence of move and pop operations. Register %eax is used for returning the value from any function that returns an integer or pointer.

Practice Problem 3.30

The following code fragment occurs often in the compiled version of library routines:

```
1     call next
2   next:
3     popl %eax
```

A. To what value does register %eax get set?

B. Explain why there is no matching ret instruction to this call.

C. What useful purpose does this code fragment serve?

3.7.3 Register Usage Conventions

The set of program registers acts as a single resource shared by all of the procedures. Although only one procedure can be active at a given time, we must make sure that when one procedure (the *caller*) calls another (the *callee*), the callee does not overwrite some register value that the caller planned to use later. For this reason, IA32 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

By convention, registers %eax, %edx, and %ecx are classified as *caller-save* registers. When procedure Q is called by P, it can overwrite these registers without destroying any data required by P. On the other hand, registers %ebx, %esi, and %edi are classified as *callee-save* registers. This means that Q must save the values of any of these registers on the stack before overwriting them, and restore them before returning, because P (or some higher-level procedure) may need these values for its future computations. In addition, registers %ebp and %esp must be maintained according to the conventions described here.

As an example, consider the following code:

```
1   int P(int x)
2   {
3       int y = x*x;
4       int z = Q(y);
5       return y + z;
6   }
```

Procedure P computes y before calling Q, but it must also ensure that the value of y is available after Q returns. It can do this by one of two means:

- It can store the value of y in its own stack frame before calling Q; when Q returns, procedure P can then retrieve the value of y from the stack. In other words, P, the *caller*, saves the value.
- It can store the value of y in a callee-save register. If Q, or any procedure called by Q, wants to use this register, it must save the register value in its stack frame and restore the value before it returns (in other words, the *callee* saves the value). When Q returns to P, the value of y will be in the callee-save register, either because the register was never altered or because it was saved and restored.

Either convention can be made to work, as long as there is agreement as to which function is responsible for saving which value. IA32 follows both approaches, partitioning the registers into one set that is caller-save, and another set that is callee-save.

Practice Problem 3.31

The following code sequence occurs right near the beginning of the assembly code generated by GCC for a C procedure:

```
1     subl    $12, %esp
2     movl    %ebx, (%esp)
3     movl    %esi, 4(%esp)
4     movl    %edi, 8(%esp)
5     movl    8(%ebp), %ebx
6     movl    12(%ebp), %edi
7     movl    (%ebx), %esi
8     movl    (%edi), %eax
9     movl    16(%ebp), %edx
10    movl    (%edx), %ecx
```

We see that just three registers (%ebx, %esi, and %edi) are saved on the stack (lines 2–4). The program modifies these and three other registers (%eax, %ecx, and %edx). At the end of the procedure, the values of registers %edi, %esi, and %ebx are restored (not shown), while the other three are left in their modified states.

Explain this apparent inconsistency in the saving and restoring of register states.

3.7.4 Procedure Example

As an example, consider the C functions defined in Figure 3.23, where function caller includes a call to function swap_add. Figure 3.24 shows the stack frame structure both just before caller calls function swap_add and while swap_add

```
1   int swap_add(int *xp, int *yp)
2   {
3       int x = *xp;
4       int y = *yp;
5
6       *xp = y;
7       *yp = x;
8       return x + y;
9   }
10
11  int caller()
12  {
13      int arg1 = 534;
14      int arg2 = 1057;
15      int sum = swap_add(&arg1, &arg2);
16      int diff = arg1 - arg2;
17
18      return sum * diff;
19  }
```

Figure 3.23 **Example of procedure definition and call.**

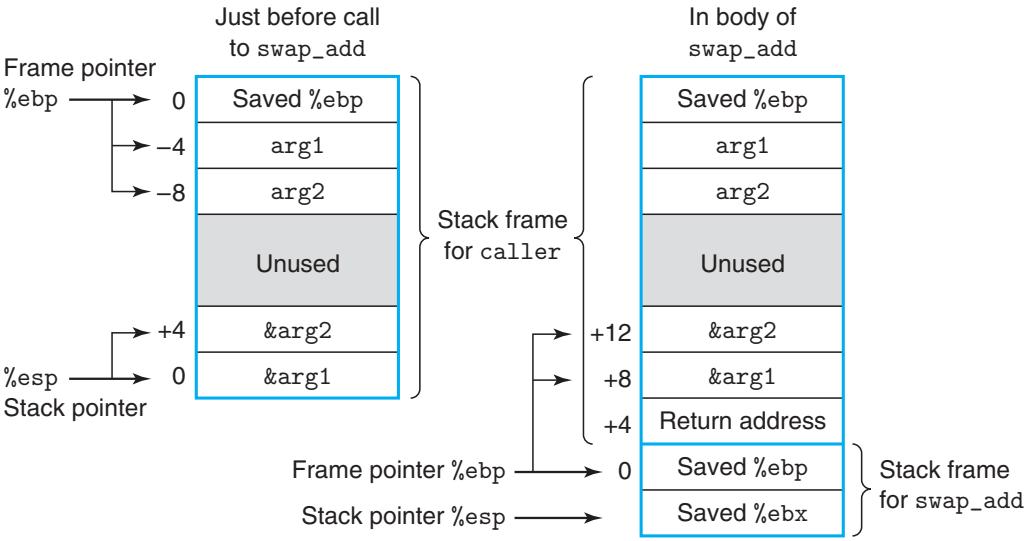


Figure 3.24 **Stack frames for `caller` and `swap_add`.** Procedure `swap_add` retrieves its arguments from the stack frame for `caller`.

is running. Some of the instructions access stack locations relative to the stack pointer %esp while others access locations relative to the base pointer %ebp. These offsets are identified by the lines shown relative to the two pointers.

New to C? Passing parameters to a function

Some languages, such as Pascal, provide two different ways to pass parameters to procedures—by *value*, where the caller provides the actual parameter value, and by *reference*, where the caller provides a pointer to the value. In C, all parameters are passed by value, but we can mimic the effect of a reference parameter by explicitly generating a pointer to a value and passing this pointer to a procedure. We can see this with the call by `caller` to `swap_add` (Figure 3.23). By passing pointers to `arg1` and `arg2`, `caller` provides a way for `swap_add` to modify these values.

One of the ways in which C++ extends C is the inclusion of reference parameters.

The stack frame for `caller` includes storage for local variables `arg1` and `arg2`, at positions −4 and −8 relative to the frame pointer. These variables must be stored on the stack, since the code must associate an address with them. The following assembly code from the compiled version of `caller` shows how it calls `swap_add`:

```
1   caller:
2     pushl   %ebp              Save old %ebp
3     movl    %esp, %ebp        Set %ebp as frame pointer
4     subl    $24, %esp         Allocate 24 bytes on stack
5     movl    $534, -4(%ebp)    Set arg1 to 534
6     movl    $1057, -8(%ebp)   Set arg2 to 1057
7     leal    -8(%ebp), %eax    Compute &arg2
8     movl    %eax, 4(%esp)     Store on stack
9     leal    -4(%ebp), %eax    Compute &arg1
10    movl    %eax, (%esp)      Store on stack
11    call    swap_add          Call the swap_add function
```

This code saves a copy of `%ebp` and sets `%ebp` to the beginning of the stack frame (lines 2–3). It then allocates 24 bytes on the stack by decrementing the stack pointer (recall that the stack grows toward lower addresses). It initializes `arg1` and `arg2` to 534 and 1057, respectively (lines 5–6), and computes the values of `&arg2` and `&arg1` and stores these on the stack to form the arguments to `swap_add` (lines 7–10). It stores these arguments relative to the stack pointer, at offsets 0 and +4 for later access by `swap_add`. It then calls `swap_add`. Of the 24 bytes allocated for the stack frame, 8 are used for the local variables, 8 are used for passing parameters to `swap_add`, and 8 are not used for anything.

Aside Why does GCC allocate space that never gets used?

We see that the code generated by GCC for `caller` allocates 24 bytes on the stack even though it only makes use of 16 of them. We will see many examples of this apparent wastefulness. GCC adheres to an x86 programming guideline that the total stack space used by the function should be a multiple of 16 bytes. Including the 4 bytes for the saved value of `%ebp` and the 4 bytes for the return address, `caller` uses a total of 32 bytes. The motivation for this convention is to ensure a proper *alignment* for accessing data. We will explain the reason for having alignment conventions and how they are implemented in Section 3.9.3.

The compiled code for `swap_add` has three parts: the "setup," where the stack frame is initialized; the "body," where the actual computation of the procedure is performed; and the "finish," where the stack state is restored and the procedure returns.

The following is the setup code for `swap_add`. Recall that before reaching this part of the code, the `call` instruction will have pushed the return address onto the stack.

```
1   swap_add:
2     pushl   %ebp              Save old %ebp
3     movl    %esp, %ebp        Set %ebp as frame pointer
4     pushl   %ebx              Save %ebx
```

Function `swap_add` requires register `%ebx` for temporary storage. Since this is a callee-save register, it pushes the old value onto the stack as part of the stack frame setup. At this point, the state of the stack is as shown on the right-hand side of Figure 3.24. Register `%ebp` has been shifted to serve as the frame pointer for `swap_add`.

The following is the body code for `swap_add`:

```
5     movl    8(%ebp), %edx     Get xp
6     movl    12(%ebp), %ecx    Get yp
7     movl    (%edx), %ebx      Get x
8     movl    (%ecx), %eax      Get y
9     movl    %eax, (%edx)      Store y at xp
10    movl    %ebx, (%ecx)      Store x at yp
11    addl    %ebx, %eax        Return value = x+y
```

This code retrieves its arguments from the stack frame for `caller`. Since the frame pointer has shifted, the locations of these arguments has shifted from positions +4 and 0 relative to the old value of `%esp` to positions +12 and +8 relative to new value of `%ebp`. The sum of variables x and y is stored in register `%eax` to be passed as the returned value.

The following is the finishing code for `swap_add`:

```
12    popl    %ebx              Restore %ebx
13    popl    %ebp              Restore %ebp
14    ret                       Return
```

This code restores the values of registers `%ebx` and `%ebp`, while also resetting the stack pointer so that it points to the stored return address, so that the `ret` instruction transfers control back to `caller`.

The following code in `caller` comes immediately after the instruction calling `swap_add`:

```
12    movl    -4(%ebp), %edx
13    subl    -8(%ebp), %edx
14    imull   %edx, %eax
15    leave
16    ret
```

This code retrieves the values of `arg1` and `arg2` from the stack in order to compute `diff`, and uses register `%eax` as the return value from `swap_add`. Observe the use of the `leave` instruction to reset both the stack and the frame pointer prior to return. We have seen in our code examples that the code generated by GCC sometimes uses a `leave` instruction to deallocate a stack frame, and sometimes it uses one or two `popl` instructions. Either approach is acceptable, and the guidelines from Intel and AMD as to which is preferable change over time.

We can see from this example that the compiler generates code to manage the stack structure according to a simple set of conventions. Arguments are passed to a function on the stack, where they can be retrieved using positive offsets (+8, +12, . . .) relative to `%ebp`. Space can be allocated on the stack either by using push instructions or by subtracting offsets from the stack pointer. Before returning, a function must restore the stack to its original condition by restoring any callee-saved registers and `%ebp`, and by resetting `%esp` so that it points to the return address. It is important for all procedures to follow a consistent set of conventions for setting up and restoring the stack in order for the program to execute properly.

Practice Problem 3.32

A C function `fun` has the following code body:

```
*p = d;
return x-c;
```

The IA32 code implementing this body is as follows:

```
1    movsbl  12(%ebp),%edx
2    movl    16(%ebp), %eax
3    movl    %edx, (%eax)
4    movswl  8(%ebp),%eax
5    movl    20(%ebp), %edx
6    subl    %eax, %edx
7    movl    %edx, %eax
```

Write a prototype for function `fun`, showing the types and ordering of the arguments `p`, `d`, `x`, and `c`.

Practice Problem 3.33

Given the C function

```
1  int proc(void)
2  {
3      int x,y;
4      scanf("%x %x", &y, &x);
5      return x-y;
6  }
```

GCC generates the following assembly code:

```
1   proc:
2     pushl   %ebp
3     movl    %esp, %ebp
4     subl    $40, %esp
5     leal    -4(%ebp), %eax
6     movl    %eax, 8(%esp)
7     leal    -8(%ebp), %eax
8     movl    %eax, 4(%esp)
9     movl    $.LC0, (%esp)    Pointer to string "%x %x"
10    call    scanf
      Diagram stack frame at this point
11    movl    -4(%ebp), %eax
12    subl    -8(%ebp), %eax
13    leave
14    ret
```

Assume that procedure `proc` starts executing with the following register values:

Register	Value
%esp	0x800040
%ebp	0x800060

Suppose `proc` calls `scanf` (line 10), and that `scanf` reads values `0x46` and `0x53` from the standard input. Assume that the string "`%x %x`" is stored at memory location `0x300070`.

A. What value does `%ebp` get set to on line 3?

B. What value does `%esp` get set to on line 4?

C. At what addresses are local variables `x` and `y` stored?

D. Draw a diagram of the stack frame for `proc` right after `scanf` returns. Include as much information as you can about the addresses and the contents of the stack frame elements.

E. Indicate the regions of the stack frame that are not used by `proc`.

3.7.5 Recursive Procedures

The stack and linkage conventions described in the previous section allow procedures to call themselves recursively. Since each call has its own private space on the stack, the local variables of the multiple outstanding calls do not interfere with one another. Furthermore, the stack discipline naturally provides the proper policy for allocating local storage when the procedure is called and deallocating it when it returns.

```
1   int rfact(int n)
2   {
3       int result;
4       if (n <= 1)
5           result = 1;
6       else
7           result = n * rfact(n-1);
8       return result;
9   }
```

Figure 3.25 **C code for recursive factorial program.**

Figure 3.25 shows the C code for a recursive factorial function. The assembly code generated by GCC is shown in Figure 3.26. Let us examine how the machine code will operate when called with argument n. The set-up code (lines 2–5) creates a stack frame containing the old version of %ebp, the saved value for callee-save register %ebx, and 4 bytes to hold the argument when it calls itself recursively, as illustrated in Figure 3.27. It uses register %ebx to save a copy of n (line 6). It sets the return value in register %eax to 1 (line 7) in anticipation of the case where $n \leq 1$, in which event it will jump to the completion code.

For the recursive case, it computes $n - 1$, stores it on the stack, and calls itself (lines 10–12). Upon completion of the code, we can assume (1) register %eax holds

```
   Argument: n at %ebp+8
   Registers: n in %ebx, result in %eax
1   rfact:
2     pushl   %ebp              Save old %ebp
3     movl    %esp, %ebp        Set %ebp as frame pointer
4     pushl   %ebx              Save callee save register %ebx
5     subl    $4, %esp          Allocate 4 bytes on stack
6     movl    8(%ebp), %ebx     Get n
7     movl    $1, %eax          result = 1
8     cmpl    $1, %ebx          Compare n:1
9     jle     .L53              If <=, goto done
10    leal    -1(%ebx), %eax    Compute n-1
11    movl    %eax, (%esp)      Store at top of stack
12    call    rfact             Call rfact(n-1)
13    imull   %ebx, %eax        Compute result = return value * n
14  .L53:                     done:
15    addl    $4, %esp          Deallocate 4 bytes from stack
16    popl    %ebx              Restore %ebx
17    popl    %ebp              Restore %ebp
18    ret                       Return result
```

Figure 3.26 **Assembly code for the recursive factorial program in Figure 3.25.**

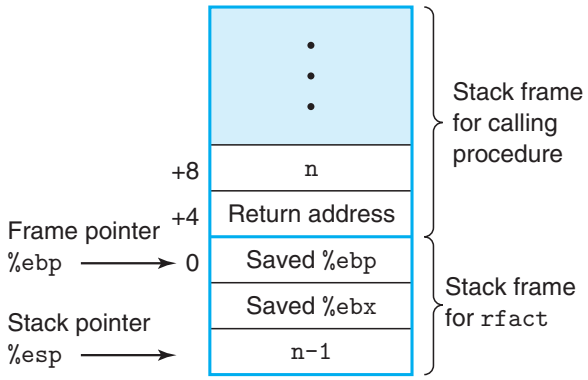


Figure 3.27
Stack frame for recursive factorial function. The state of the frame is shown just before the recursive call.

the value of $(n-1)!$ and (2) callee-save register %ebx holds the parameter n. It therefore multiplies these two quantities (line 13) to generate the return value of the function.

For both cases—the terminal condition and the recursive call—the code proceeds to the completion section (lines 15–17) to restore the stack and callee-saved register, and then it returns.

We can see that calling a function recursively proceeds just like any other function call. Our stack discipline provides a mechanism where each invocation of a function has its own private storage for state information (saved values of the return location, frame pointer, and callee-save registers). If need be, it can also provide storage for local variables. The stack discipline of allocation and deallocation naturally matches the call-return ordering of functions. This method of implementing function calls and returns even works for more complex patterns, including mutual recursion (for example, when procedure P calls Q, which in turn calls P).

Practice Problem 3.34

For a C function having the general structure

```
int rfun(unsigned x) {
    if ( __________ )
        return __________;
    unsigned nx = __________;
    int rv = rfun(nx);
    return __________;
}
```

GCC generates the following assembly code (with the setup and completion code omitted):

```
1     movl    8(%ebp), %ebx
2     movl    $0, %eax
3     testl   %ebx, %ebx
4     je      .L3
```

```
5      movl    %ebx, %eax
6      shrl    %eax              Shift right by 1
7      movl    %eax, (%esp)
8      call    rfun
9      movl    %ebx, %edx
10     andl    $1, %edx
11     leal    (%edx,%eax), %eax
12   .L3:
```

A. What value does `rfun` store in the callee-save register `%ebx`?

B. Fill in the missing expressions in the C code shown above.

C. Describe in English what function this code computes.

3.8 Array Allocation and Access

Arrays in C are one means of aggregating scalar data into larger data types. C uses a particularly simple implementation of arrays, and hence the translation into machine code is fairly straightforward. One unusual feature of C is that we can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in machine code.

Optimizing compilers are particularly good at simplifying the address computations used by array indexing. This can make the correspondence between the C code and its translation into machine code somewhat difficult to decipher.

3.8.1 Basic Principles

For data type T and integer constant N, the declaration

T `A[`N`];`

has two effects. First, it allocates a contiguous region of $L \cdot N$ bytes in memory, where L is the size (in bytes) of data type T. Let us denote the starting location as $x_{\mathtt{A}}$. Second, it introduces an identifier `A` that can be used as a pointer to the beginning of the array. The value of this pointer will be $x_{\mathtt{A}}$. The array elements can be accessed using an integer index ranging between 0 and $N-1$. Array element i will be stored at address $x_{\mathtt{A}} + L \cdot i$.

As examples, consider the following declarations:

```
char    A[12];
char   *B[8];
double  C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	4	32	x_B	$x_B + 4i$
C	8	48	x_C	$x_C + 8i$
D	4	20	x_D	$x_D + 4i$

Array `A` consists of 12 single-byte (`char`) elements. Array `C` consists of six double-precision floating-point values, each requiring 8 bytes. `B` and `D` are both arrays of pointers, and hence the array elements are 4 bytes each.

The memory referencing instructions of IA32 are designed to simplify array access. For example, suppose `E` is an array of `int`'s, and we wish to evaluate `E[i]`, where the address of `E` is stored in register `%edx` and i is stored in register `%ecx`. Then the instruction

```
movl (%edx,%ecx,4),%eax
```

will perform the address computation $x_E + 4i$, read that memory location, and copy the result to register `%eax`. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the common primitive data types.

Practice Problem 3.35

Consider the following declarations:

```
short          S[7];
short         *T[3];
short        **U[6];
long double    V[8];
long double   *W[4];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element size	Total size	Start address	Element i
S	_____	_____	x_S	_____
T	_____	_____	x_T	_____
U	_____	_____	x_U	_____
V	_____	_____	x_V	_____
W	_____	_____	x_W	_____

3.8.2 Pointer Arithmetic

C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer. That is, if `p` is a pointer to data

of type T, and the value of `p` is x_p, then the expression `p+i` has value $x_p + L \cdot i$, where L is the size of data type T.

The unary operators `&` and `*` allow the generation and dereferencing of pointers. That is, for an expression *Expr* denoting some object, `&`*Expr* is a pointer giving the address of the object. For an expression *AExpr* denoting an address, `*`*AExpr* gives the value at that address. The expressions *Expr* and `*&`*Expr* are therefore equivalent. The array subscripting operation can be applied to both arrays and pointers. The array reference `A[i]` is identical to the expression `*(A+i)`. It computes the address of the *i*th array element and then accesses this memory location.

Expanding on our earlier example, suppose the starting address of integer array `E` and integer index i are stored in registers `%edx` and `%ecx`, respectively. The following are some expressions involving `E`. We also show an assembly-code implementation of each expression, with the result being stored in register `%eax`.

Expression	Type	Value	Assembly code
`E`	`int *`	x_E	`movl %edx,%eax`
`E[0]`	`int`	M[x_E]	`movl (%edx),%eax`
`E[i]`	`int`	M[$x_E + 4i$]	`movl (%edx,%ecx,4),%eax`
`&E[2]`	`int *`	$x_E + 8$	`leal 8(%edx),%eax`
`E+i-1`	`int *`	$x_E + 4i - 4$	`leal -4(%edx,%ecx,4),%eax`
`*(E+i-3)`	`int *`	M[$x_E + 4i - 12$]	`movl -12(%edx,%ecx,4),%eax`
`&E[i]-E`	`int`	i	`movl %ecx,%eax`

In these examples, the `leal` instruction is used to generate an address, while `movl` is used to reference memory (except in the first and last cases, where the former copies an address and the latter copies the index). The final example shows that one can compute the difference of two pointers within the same data structure, with the result divided by the size of the data type.

Practice Problem 3.36

Suppose the address of `short` integer array `S` and integer index i are stored in registers `%edx` and `%ecx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in register `%eax` if it is a pointer and register element `%ax` if it is a `short` integer.

Expression	Type	Value	Assembly code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

3.8.3 Nested Arrays

The general principles of array allocation and referencing hold even when we create arrays of arrays. For example, the declaration

```
int A[5][3];
```

is equivalent to the declaration

```
typedef int row3_t[3];
row3_t A[5];
```

Data type `row3_t` is defined to be an array of three integers. Array `A` contains five such elements, each requiring 12 bytes to store the three integers. The total array size is then $4 \cdot 5 \cdot 3 = 60$ bytes.

Array `A` can also be viewed as a two-dimensional array with five rows and three columns, referenced as `A[0][0]` through `A[4][2]`. The array elements are ordered in memory in "row major" order, meaning all elements of row 0, which can be written `A[0]`, followed by all elements of row 1 (`A[1]`), and so on.

Row	Element	Address
`A[0]`	`A[0][0]`	x_A
	`A[0][1]`	$x_A + 4$
	`A[0][2]`	$x_A + 8$
`A[1]`	`A[1][0]`	$x_A + 12$
	`A[1][1]`	$x_A + 16$
	`A[1][2]`	$x_A + 20$
`A[2]`	`A[2][0]`	$x_A + 24$
	`A[2][1]`	$x_A + 28$
	`A[2][2]`	$x_A + 32$
`A[3]`	`A[3][0]`	$x_A + 36$
	`A[3][1]`	$x_A + 40$
	`A[3][2]`	$x_A + 44$
`A[4]`	`A[4][0]`	$x_A + 48$
	`A[4][1]`	$x_A + 52$
	`A[4][2]`	$x_A + 56$

This ordering is a consequence of our nested declaration. Viewing `A` as an array of five elements, each of which is an array of three `int`'s, we first have `A[0]`, followed by `A[1]`, and so on.

To access elements of multidimensional arrays, the compiler generates code to compute the offset of the desired element and then uses one of the MOV instructions with the start of the array as the base address and the (possibly scaled) offset as an index. In general, for an array declared as

T `D[`*R*`][`*C*`];`

array element `D[i][j]` is at memory address

$$\&D[i][j] = x_D + L(C \cdot i + j), \tag{3.1}$$

where L is the size of data type T in bytes. As an example, consider the 5×3 integer array `A` defined earlier. Suppose x_A, i, and j are at offsets 8, 12, and 16 relative to `%ebp`, respectively. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
   A at %ebp+8, i at %ebp+12,j at %ebp+16
1     movl    12(%ebp), %eax          Get i
2     leal    (%eax,%eax,2), %eax     Compute 3*i
3     movl    16(%ebp), %edx          Get j
4     sall    $2, %edx                Compute j*4
5     addl    8(%ebp), %edx           Compute x_A + 4j
6     movl    (%edx,%eax,4), %eax     Read from M[x_A + 4j + 12i]
```

As can be seen, this code computes the element's address as $x_A + 4j + 12i = x_A + 4(3i + j)$ using a combination of shifting, adding, and scaling to avoid more costly multiplication instructions.

Practice Problem 3.37

Consider the following source code, where M and N are constants declared with `#define`:

```
1  int mat1[M][N];
2  int mat2[N][M];
3
4  int sum_element(int i, int j) {
5      return mat1[i][j] + mat2[j][i];
6  }
```

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

```
   i at %ebp+8, j at %ebp+12
1     movl    8(%ebp), %ecx
2     movl    12(%ebp), %edx
3     leal    0(,%ecx,8), %eax
4     subl    %ecx, %eax
5     addl    %edx, %eax
6     leal    (%edx,%edx,4), %edx
7     addl    %ecx, %edx
8     movl    mat1(,%eax,4), %eax
9     addl    mat2(,%edx,4), %eax
```

Use your reverse engineering skills to determine the values of M and N based on this assembly code.

3.8.4 Fixed-Size Arrays

The C compiler is able to make many optimizations for code operating on multi-dimensional arrays of fixed size. For example, suppose we declare data type `fix_matrix` to be 16×16 arrays of integers as follows:

```
1  #define N 16
2  typedef int fix_matrix[N][N];
```

(This example illustrates a good coding practice. Whenever a program uses some constant as an array dimension or buffer size, it is best to associate a name with it via a `#define` declaration, and then use this name consistently, rather than the numeric value. That way, if an occasion ever arises to change the value, it can be done by simply modifying the `#define` declaration.) The code in Figure 3.28(a) computes element i, k of the product of arrays `A` and `B`, according to the formula $\sum_{0 \le j < N} a_{i,j} \cdot b_{j,k}$. The C compiler generates code that we then recoded into C, shown as function `fix_prod_ele_opt` in Figure 3.28(b). This code contains a number of clever optimizations. It recognizes that the loop will access just the elements of row i of array `A`, and so it creates a local pointer variable, which we have named `Arow`, to provide direct access to row i of the array. `Arow` is initialized to `&A[i][0]`, and so array element `A[i][j]` can be accessed as `Arow[j]`. It also recognizes that the loop will access the elements of array `B` as `B[0][k]`, `B[1][k]`, . . . , `B[15][k]` in sequence. These elements occupy positions in memory starting with the address of array element `B[0][k]` and spaced 64 bytes apart. The program can therefore use a pointer variable `Bptr` to access these successive locations. In C, this pointer is shown as being incremented by N (16), although in fact the actual address is incremented by $4 \cdot 16 = 64$.

The following is the actual assembly code for the loop. We see that four variables are maintained in registers within the loop: `Arow`, `Bptr`, `j`, and `result`.

```
   Registers: Arow in %esi, Bptr in %ecx, j in %edx, result in %ebx
1   .L6:                                loop:
2     movl    (%ecx), %eax                Get *Bptr
3     imull   (%esi,%edx,4), %eax         Multiply by Arow[j]
4     addl    %eax, %ebx                  Add to result
5     addl    $1, %edx                    Increment j
6     addl    $64, %ecx                   Add 64 to Bptr
7     cmpl    $16, %edx                   Compare j:16
8     jne     .L6                         If !=, goto loop
```

As can be seen, register `%ecx` is incremented by 64 within the loop (line 6). Machine code considers every pointer to be a byte address, and so in compiling pointer arithmetic, it must scale every increment by the size of the underlying data type.

Practice Problem 3.38

The following C code sets the diagonal elements of one of our fixed-size arrays to `val`:

```
1   /* Set all diagonal elements to val */
2   void fix_set_diag(fix_matrix A, int val) {
3     int i;
4     for (i = 0; i < N; i++)
5       A[i][i] = val;
6   }
```

When compiled, GCC generates the following assembly code:

```
   A at %ebp+8, val at %ebp+12
1     movl    8(%ebp), %ecx
2     movl    12(%ebp), %edx
3     movl    $0, %eax
4   .L14:
5     movl    %edx, (%ecx,%eax)
6     addl    $68, %eax
7     cmpl    $1088, %eax
8     jne     .L14
```

Create a C-code program `fix_set_diag_opt` that uses optimizations similar to those in the assembly code, in the same style as the code in Figure 3.28(b). Use expressions involving the parameter N rather than integer constants, so that your code will work correctly if N is redefined.

3.8.5 Variable-Size Arrays

Historically, C only supported multidimensional arrays where the sizes (with the possible exception of the first dimension) could be determined at compile time. Programmers requiring variable-sized arrays had to allocate storage for these arrays using functions such as `malloc` or `calloc`, and had to explicitly encode the mapping of multidimensional arrays into single-dimension ones via row-major indexing, as expressed in Equation 3.1. ISO C99 introduced the capability to have array dimensions be expressions that are computed as the array is being allocated, and recent versions of GCC support most of the conventions for variable-sized arrays in ISO C99.

In the C version of variable-size arrays, we can declare an array `int A[`*expr1*`][`*expr2*`]`, either as a local variable or as an argument to a function, and then the dimensions of the array are determined by evaluating the expressions *expr1* and *expr2* at the time the declaration is encountered. So, for example, we can write a function to access element i, j of an $n \times n$ array as follows:

```
1   int var_ele(int n, int A[n][n], int i, int j) {
2       return A[i][j];
3   }
```

(a) Original C code

```
1   /* Compute i,k of fixed matrix product */
2   int fix_prod_ele (fix_matrix A, fix_matrix B,  int i, int k) {
3       int j;
4       int result = 0;
5
6       for (j = 0; j < N; j++)
7           result += A[i][j] * B[j][k];
8
9       return result;
10  }
```

(b) Optimized C code

```
1   /* Compute i,k of fixed matrix product */
2   int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k) {
3       int *Arow = &A[i][0];
4       int *Bptr = &B[0][k];
5       int result = 0;
6       int j;
7       for (j = 0; j != N; j++) {
8           result += Arow[j] * *Bptr;
9           Bptr += N;
10      }
11      return result;
12  }
```

Figure 3.28 **Original and optimized code to compute element i, k of matrix product for fixed-length arrays.** The compiler performs these optimizations automatically.

The parameter `n` must precede the parameter `A[n][n]`, so that the function can compute the array dimensions as the parameter is encountered.

GCC generates code for this referencing function as

```
   n at %ebp+8, A at %ebp+12, i at %ebp+16, j at %ebp+20
1     movl    8(%ebp), %eax            Get n
2     sall    $2, %eax                 Compute 4*n
3     movl    %eax, %edx               Copy 4*n
4     imull   16(%ebp), %edx           Compute 4*n*i
5     movl    20(%ebp), %eax           Get j
6     sall    $2, %eax                 Compute 4*j
7     addl    12(%ebp), %eax           Compute x_A + 4*j
8     movl    (%eax,%edx), %eax        Read from x_A + 4*(n*i + j)
```

As the annotations show, this code computes the address of element i, j as $x_A + 4(n \cdot i + j)$. The address computation is similar to that of the fixed-size array (page 236), except that (1) the positions of the arguments on the stack are shifted due to the addition of parameter `n`, and (2) a multiply instruction is used (line 4) to

```
1   /* Compute i,k of variable matrix product */
2   int var_prod_ele(int n, int A[n][n], int B[n][n], int i, int k) {
3       int j;
4       int result = 0;
5
6       for (j = 0; j < n; j++)
7           result += A[i][j] * B[j][k];
8
9       return result;
10  }
```

Figure 3.29 **Code to compute element i, k of matrix product for variable-sized arrays.** The compiler performs optimizations similar to those for fixed-size arrays.

compute $n \cdot i$, rather than an `leal` instruction to compute $3i$. We see therefore that referencing variable-size arrays requires only a slight generalization over fixed-size ones. The dynamic version must use a multiplication instruction to scale i by n, rather than a series of shifts and adds. In some processors, this multiplication can incur a significant performance penalty, but it is unavoidable in this case.

When variable-sized arrays are referenced within a loop, the compiler can often optimize the index computations by exploiting the regularity of the access patterns. For example, Figure 3.29 shows C code to compute element i, k of the product of two $n \times n$ arrays `A` and `B`. The compiler generates code similar to what we saw for fixed-size arrays. In fact, the code bears close resemblance to that of Figure 3.28(b), except that it scales `Bptr`, the pointer to element `B[j][k]`, by the variable value n rather than the fixed value N on each iteration.

The following is the assembly code for the loop of `var_prod_ele`:

```
    n stored at %ebp+8
    Registers: Arow in %esi, Bptr in %ecx, j in %edx,
        result in %ebx, %edi holds 4*n
1   .L30:                                   loop:
2     movl    (%ecx), %eax                    Get *Bptr
3     imull   (%esi,%edx,4), %eax             Multiply by Arow[j]
4     addl    %eax, %ebx                      Add to result
5     addl    $1, %edx                        Increment j
6     addl    %edi, %ecx                      Add 4*n to Bptr
7     cmpl    %edx, 8(%ebp)                   Compare n:j
8     jg      .L30                            If >, goto loop
```

We see that the program makes use of both a scaled value $4n$ (register `%edi`) for incrementing `Bptr` and the actual value of n stored at offset 8 from `%ebp` to check the loop bounds. The need for two values does not show up in the C code, due to the scaling of pointer arithmetic. The code retrieves the value of n from memory on each iteration to check for loop termination (line 7). This is an example of *register spilling*: there are not enough registers to hold all of the needed temporary data, and hence the compiler must keep some local variables in memory. In this case the compiler chose to spill n, because it is a "read-only" value—it does not change

value within the loop. IA32 must often spill loop values to memory, since the processor has so few registers. In general, reading from memory can be done more readily than writing to memory, and so spilling read-only variables is preferable. See Problem 3.61 regarding how to improve this code to avoid register spilling.

3.9 Heterogeneous Data Structures

C provides two mechanisms for creating data types by combining objects of different types: *structures*, declared using the keyword `struct`, aggregate multiple objects into a single unit; *unions*, declared using the keyword `union`, allow an object to be referenced using several different types.

3.9.1 Structures

The C `struct` declaration creates a data type that groups objects of possibly different types into a single object. The different components of a structure are referenced by names. The implementation of structures is similar to that of arrays in that all of the components of a structure are stored in a contiguous region of memory, and a pointer to a structure is the address of its first byte. The compiler maintains information about each structure type indicating the byte offset of each field. It generates references to structure elements using these offsets as displacements in memory referencing instructions.

New to C? Representing an object as a `struct`

The `struct` data type constructor is the closest thing C provides to the objects of C++ and Java. It allows the programmer to keep information about some entity in a single data structure, and reference that information with names.

For example, a graphics program might represent a rectangle as a structure:

```
struct rect {
    int llx;    /* X coordinate of lower-left corner */
    int lly;    /* Y coordinate of lower-left corner */
    int color;  /* Coding of color                   */
    int width;  /* Width (in pixels)                 */
    int height; /* Height (in pixels)                */
};
```

We could declare a variable `r` of type `struct rect` and set its field values as follows:

```
struct rect r;
r.llx = r.lly = 0;
r.color = 0xFF00FF;
r.width = 10;
r.height = 20;
```

where the expression `r.llx` selects field `llx` of structure `r`.

Alternatively, we can both declare the variable and initialize its fields with a single statement:

```
struct rect r = { 0, 0, 0xFF00FF, 10, 20 };
```

It is common to pass pointers to structures from one place to another rather than copying them. For example, the following function computes the area of a rectangle, where a pointer to the rectangle `struct` is passed to the function:

```
int area(struct rect *rp)
{
    return (*rp).width * (*rp).height;
}
```

The expression `(*rp).width` dereferences the pointer and selects the `width` field of the resulting structure. Parentheses are required, because the compiler would interpret the expression `*rp.width` as `*(rp.width)`, which is not valid. This combination of dereferencing and field selection is so common that C provides an alternative notation using `->`. That is, `rp->width` is equivalent to the expression `(*rp).width`. For example, we could write a function that rotates a rectangle counterclockwise by 90 degrees as

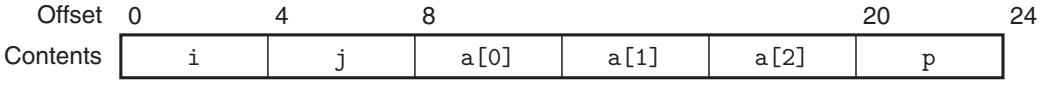
```
void rotate_left(struct rect *rp)
{
    /* Exchange width and height */
    int t      = rp->height;
    rp->height = rp->width;
    rp->width  = t;
    /* Shift to new lower-left corner */
    rp->llx   -= t;
}
```

The objects of C++ and Java are more elaborate than structures in C, in that they also associate a set of *methods* with an object that can be invoked to perform computation. In C, we would simply write these as ordinary functions, such as the functions `area` and `rotate_left` shown above.

As an example, consider the following structure declaration:

```
struct rec {
    int i;
    int j;
    int a[3];
    int *p;
};
```

This structure contains four fields: two 4-byte `int`'s, an array consisting of three 4-byte `int`'s, and a 4-byte integer pointer, giving a total of 24 bytes:



Observe that array `a` is embedded within the structure. The numbers along the top of the diagram give the byte offsets of the fields from the beginning of the structure.

To access the fields of a structure, the compiler generates code that adds the appropriate offset to the address of the structure. For example, suppose variable `r` of type `struct rec *` is in register `%edx`. Then the following code copies element `r->i` to element `r->j`:

```
1      movl    (%edx), %eax     Get r->i
2      movl    %eax, 4(%edx)    Store in r->j
```

Since the offset of field `i` is 0, the address of this field is simply the value of `r`. To store into field `j`, the code adds offset 4 to the address of `r`.

To generate a pointer to an object within a structure, we can simply add the field's offset to the structure address. For example, we can generate the pointer `&(r->a[1])` by adding offset $8 + 4 \cdot 1 = 12$. For pointer `r` in register `%eax` and integer variable `i` in register `%edx`, we can generate the pointer value `&(r->a[i])` with the single instruction

```
     Registers: r in %edx, i in %eax
1      leal    8(%edx,%eax,4), %eax     Set %eax to &r->a[i]
```

As a final example, the following code implements the statement

```
r->p = &r->a[r->i + r->j];
```

starting with `r` in register `%edx`:

```
1      movl    4(%edx), %eax            Get r->j
2      addl    (%edx), %eax             Add r->i
3      leal    8(%edx,%eax,4), %eax     Compute &r->a[r->i + r->j]
4      movl    %eax, 20(%edx)           Store in r->p
```

As these examples show, the selection of the different fields of a structure is handled completely at compile time. The machine code contains no information about the field declarations or the names of the fields.

Practice Problem 3.39

Consider the following structure declaration:

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

This declaration illustrates that one structure can be embedded within another, just as arrays can be embedded within structures, and arrays can be embedded within arrays.

The following procedure (with some expressions omitted) operates on this structure:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = _________;
    sp->p     = _________;
    sp->next  = _________;
}
```

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

```
    p: _______
  s.x: _______
  s.y: _______
 next: _______
```

B. How many total bytes does the structure require?

C. The compiler generates the following assembly code for the body of `sp_init`:

```
    sp at %ebp+8
1     movl    8(%ebp), %eax
2     movl    8(%eax), %edx
3     movl    %edx, 4(%eax)
4     leal    4(%eax), %edx
5     movl    %edx, (%eax)
6     movl    %eax, 12(%eax)
```

On the basis of this information, fill in the missing expressions in the code for `sp_init`.

3.9.2 Unions

Unions provide a way to circumvent the type system of C, allowing a single object to be referenced according to multiple types. The syntax of a union declaration is identical to that for structures, but its semantics are very different. Rather than having the different fields reference different blocks of memory, they all reference the same block.

Consider the following declarations:

```
struct S3 {
    char c;
    int i[2];
```

```
    double v;
};

union U3 {
    char c;
    int i[2];
    double v;
};
```

When compiled on an IA32 Linux machine, the offsets of the fields, as well as the total size of data types `S3` and `U3`, are as shown in the following table:

Type	c	i	v	Size
S3	0	4	12	20
U3	0	0	0	8

(We will see shortly why `i` has offset 4 in `S3` rather than 1, and we will discuss why the results would be different for a machine running Microsoft Windows.) For pointer `p` of type `union U3 *`, references `p->c`, `p->i[0]`, and `p->v` would all reference the beginning of the data structure. Observe also that the overall size of a union equals the maximum size of any of its fields.

Unions can be useful in several contexts. However, they can also lead to nasty bugs, since they bypass the safety provided by the C type system. One application is when we know in advance that the use of two different fields in a data structure will be mutually exclusive. Then, declaring these two fields as part of a union rather than a structure will reduce the total space allocated.

For example, suppose we want to implement a binary tree data structure where each leaf node has a `double` data value, while each internal node has pointers to two children, but no data. If we declare this as

```
struct NODE_S {
    struct NODE_S *left;
    struct NODE_S *right;
    double data;
};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as

```
union NODE_U {
    struct {
        union NODE_U *left;
        union NODE_U *right;
    } internal;
    double data;
};
```

then every node will require just 8 bytes. If `n` is a pointer to a node of type `union NODE *`, we would reference the data of a leaf node as `n->data`, and the children of an internal node as `n->internal.left` and `n->internal.right`.

With this encoding, however, there is no way to determine whether a given node is a leaf or an internal node. A common method is to introduce an enumerated type defining the different possible choices for the union, and then create a structure containing a tag field and the union:

```
typedef enum { N_LEAF, N_INTERNAL } nodetype_t;

struct NODE_T {
    nodetype_t type;
    union {
        struct {
            struct NODE_T *left;
            struct NODE_T *right;
        } internal;
        double data;
    } info;
};
```

This structure requires a total of 12 bytes: 4 for `type`, and either 4 each for `info.internal.left` and `info.internal.right`, or 8 for `info.data`. In this case, the savings gain of using a union is small relative to the awkwardness of the resulting code. For data structures with more fields, the savings can be more compelling.

Unions can also be used to access the bit patterns of different data types. For example, the following code returns the bit representation of a `float` as an `unsigned`:

```
1   unsigned float2bit(float f)
2   {
3       union {
4           float f;
5           unsigned u;
6       } temp;
7       temp.f = f;
8       return temp.u;
9   };
```

In this code, we store the argument in the union using one data type, and access it using another. Interestingly, the code generated for this procedure is identical to that for the following procedure:

```
1   unsigned copy(unsigned u)
2   {
3       return u;
4   }
```

The body of both procedures is just a single instruction:

```
1     movl    8(%ebp), %eax
```

This demonstrates the lack of type information in machine code. The argument will be at offset 8 relative to `%ebp` regardless of whether it is a `float` or an `unsigned`. The procedure simply copies its argument as the return value without modifying any bits.

When using unions to combine data types of different sizes, byte-ordering issues can become important. For example, suppose we write a procedure that will create an 8-byte `double` using the bit patterns given by two 4-byte `unsigned`'s:

```
1   double bit2double(unsigned word0, unsigned word1)
2   {
3       union {
4           double d;
5           unsigned u[2];
6       } temp;
7
8       temp.u[0] = word0;
9       temp.u[1] = word1;
10      return temp.d;
11  }
```

On a little-endian machine such as IA32, argument `word0` will become the low-order 4 bytes of `d`, while `word1` will become the high-order 4 bytes. On a big-endian machine, the role of the two arguments will be reversed.

Practice Problem 3.40

Suppose you are given the job of checking that a C compiler generates the proper code for structure and union access. You write the following structure declaration:

```
typedef union {
    struct {
        short  v;
        short  d;
        int    s;
    } t1;
    struct {
        int a[2];
        char  *p;
    } t2;
} u_type;
```

You write a series of functions of the form

```
void get(u_type *up, TYPE *dest) {
    *dest =  EXPR;
}
```

with different access expressions EXPR, and with destination data type TYPE set according to type associated with EXPR. You then examine the code generated when compiling the functions to see if they match your expectations.

Suppose in these functions that up and dest are loaded into registers %eax and %edx, respectively. Fill in the following table with data type TYPE and sequences of 1–3 instructions to compute the expression and store the result at dest. Try to use just registers %eax and %edx, using register %ecx when these do not suffice.

EXPR	TYPE	Code
up->t1.s	int	movl 4(%eax), %eax movl %eax, (%edx)
up->t1.v	_____	_____
&up->t1.d	_____	_____
up->t2.a	_____	_____
up->t2.a[up->t1.s]	_____	_____
*up->t2.p	_____	_____

3.9.3 Data Alignment

Many computer systems place restrictions on the allowable addresses for the primitive data types, requiring that the address for some type of object must be a multiple of some value K (typically 2, 4, or 8). Such *alignment restrictions* simplify the design of the hardware forming the interface between the processor and the memory system. For example, suppose a processor always fetches 8 bytes from memory with an address that must be a multiple of 8. If we can guarantee that any double will be aligned to have its address be a multiple of 8, then the value can be read or written with a single memory operation. Otherwise, we may need to

perform two memory accesses, since the object might be split across two 8-byte memory blocks.

The IA32 hardware will work correctly regardless of the alignment of data. However, Intel recommends that data be aligned to improve memory system performance. Linux follows an alignment policy where 2-byte data types (e.g., `short`) must have an address that is a multiple of 2, while any larger data types (e.g., `int`, `int *`, `float`, and `double`) must have an address that is a multiple of 4. Note that this requirement means that the least significant bit of the address of an object of type `short` must equal zero. Similarly, any object of type `int`, or any pointer, must be at an address having the low-order 2 bits equal to zero.

Aside A case of mandatory alignment

For most IA32 instructions, keeping data aligned improves efficiency, but it does not affect program behavior. On the other hand, some of the SSE instructions for implementing multimedia operations will not work correctly with unaligned data. These instructions operate on 16-byte blocks of data, and the instructions that transfer data between the SSE unit and memory require the memory addresses to be multiples of 16. Any attempt to access memory with an address that does not satisfy this alignment will lead to an *exception*, with the default behavior for the program to terminate.

This is the motivation behind the IA32 convention of making sure that every stack frame is a multiple of 16 bytes long (see the aside of page 226). The compiler can allocate storage within a stack frame in such a way that a block can be stored with a 16-byte alignment.

Aside Alignment with Microsoft Windows

Microsoft Windows imposes a stronger alignment requirement—any primitive object of K bytes, for $K = 2$, 4, or 8, must have an address that is a multiple of K. In particular, it requires that the address of a `double` or a `long long` be a multiple of 8. This requirement enhances the memory performance at the expense of some wasted space. The Linux convention, where 8-byte values are aligned on 4-byte boundaries was probably good for the i386, back when memory was scarce and memory interfaces were only 4 bytes wide. With modern processors, Microsoft's alignment is a better design decision. Data type `long double`, for which GCC generates IA32 code allocating 12 bytes (even though the actual data type requires only 10 bytes) has a 4-byte alignment requirement with both Windows and Linux.

Alignment is enforced by making sure that every data type is organized and allocated in such a way that every object within the type satisfies its alignment restrictions. The compiler places directives in the assembly code indicating the desired alignment for global data. For example, the assembly-code declaration of the jump table beginning on page 217 contains the following directive on line 2:

```
.align 4
```

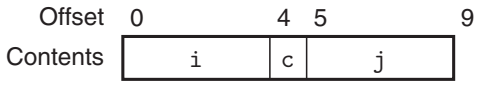
This ensures that the data following it (in this case the start of the jump table) will start with an address that is a multiple of 4. Since each table entry is 4 bytes long, the successive elements will obey the 4-byte alignment restriction.

Library routines that allocate memory, such as `malloc`, must be designed so that they return a pointer that satisfies the worst-case alignment restriction for the machine it is running on, typically 4 or 8. For code involving structures, the compiler may need to insert gaps in the field allocation to ensure that each structure element satisfies its alignment requirement. The structure then has some required alignment for its starting address.

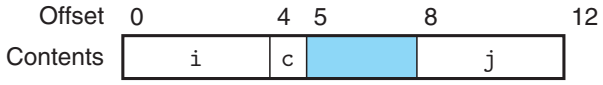
For example, consider the following structure declaration:

```
struct S1 {
    int  i;
    char c;
    int  j;
};
```

Suppose the compiler used the minimal 9-byte allocation, diagrammed as follows:



Then it would be impossible to satisfy the 4-byte alignment requirement for both fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap (shown here as shaded in blue) between fields `c` and `j`:



As a result, `j` has offset 8, and the overall structure size is 12 bytes. Furthermore, the compiler must ensure that any pointer `p` of type `struct S1*` satisfies a 4-byte alignment. Using our earlier notation, let pointer `p` have value x_p. Then x_p must be a multiple of 4. This guarantees that both `p->i` (address x_p) and `p->j` (address $x_p + 8$) will satisfy their 4-byte alignment requirements.

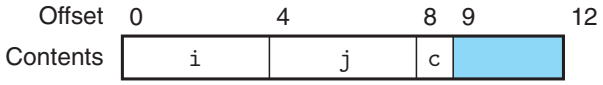
In addition, the compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement. For example, consider the following structure declaration:

```
struct S2 {
    int  i;
    int  j;
    char c;
};
```

If we pack this structure into 9 bytes, we can still satisfy the alignment requirements for fields `i` and `j` by making sure that the starting address of the structure satisfies a 4-byte alignment requirement. Consider, however, the following declaration:

```
struct S2 d[4];
```

With the 9-byte allocation, it is not possible to satisfy the alignment requirement for each element of `d`, because these elements will have addresses x_d, $x_d + 9$, $x_d + 18$, and $x_d + 27$. Instead, the compiler allocates 12 bytes for structure `S2`, with the final 3 bytes being wasted space:



That way the elements of `d` will have addresses x_d, $x_d + 12$, $x_d + 24$, and $x_d + 36$. As long as x_d is a multiple of 4, all of the alignment restrictions will be satisfied.

Practice Problem 3.41

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement under Linux/IA32.

A. `struct P1 { int i; char c; int j; char d; };`

B. `struct P2 { int i; char c; char d; int j; };`

C. `struct P3 { short w[3]; char c[3] };`

D. `struct P4 { short w[3]; char *c[3] };`

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

Practice Problem 3.42

For the structure declaration

```
struct {
    char     *a;
    short     b;
    double    c;
    char      d;
    float     e;
    char      f;
    long long g;
    void     *h;
} foo;
```

suppose it was compiled on a Windows machine, where each primitive data type of K bytes must have an offset that is a multiple of K.

A. What are the byte offsets of all the fields in the structure?

B. What is the total size of the structure?

C. Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size for the rearranged structure.

3.10 Putting It Together: Understanding Pointers

Pointers are a central feature of the C programming language. They serve as a uniform way to generate references to elements within different data structures. Pointers are a source of confusion for novice programmers, but the underlying concepts are fairly simple. Here we highlight some key principles of pointers and their mapping into machine code.

- *Every pointer has an associated type*. This type indicates what kind of object the pointer points to. Using the following pointer declarations as illustrations,

  ```
  int *ip;
  char **cpp;
  ```

  variable `ip` is a pointer to an object of type `int`, while `cpp` is a pointer to an object that itself is a pointer to an object of type `char`. In general, if the object has type T, then the pointer has type $*T$. The special `void *` type represents a generic pointer. For example, the `malloc` function returns a generic pointer, which is converted to a typed pointer via either an explicit cast or by the implicit casting of the assignment operation. Pointer types are not part of machine code; they are an abstraction provided by C to help programmers avoid addressing errors.
- *Every pointer has a value*. This value is an address of some object of the designated type. The special `NULL` (0) value indicates that the pointer does not point anywhere.
- *Pointers are created with the `&` operator*. This operator can be applied to any C expression that is categorized as an *lvalue*, meaning an expression that can appear on the left side of an assignment. Examples include variables and the elements of structures, unions, and arrays. We have seen that the machine-code realization of the `&` operator often uses the `leal` instruction to compute the expression value, since this instruction is designed to compute the address of a memory reference.
- *Pointers are dereferenced with the `*` operator*. The result is a value having the type associated with the pointer. Dereferencing is implemented by a memory reference, either storing to or retrieving from the specified address.
- *Arrays and pointers are closely related*. The name of an array can be referenced (but not updated) as if it were a pointer variable. Array referencing (e.g., `a[3]`) has the exact same effect as pointer arithmetic and dereferencing (e.g., `*(a+3)`). Both array referencing and pointer arithmetic require scaling the offsets by the object size. When we write an expression `p+i` for pointer `p` with value p, the resulting address is computed as $p + L \cdot i$, where L is the size of the data type associated with `p`.
- *Casting from one type of pointer to another changes its type but not its value.* One effect of casting is to change any scaling of pointer arithmetic. So for example, if `p` is a pointer of type `char *` having value p, then the expression

`(int *) p+7` computes $p + 28$, while `(int *) (p+7)` computes $p + 7$. (Recall that casting has higher precedence than addition.)

- *Pointers can also point to functions.* This provides a powerful capability for storing and passing references to code, which can be invoked in some other part of the program. For example, if we have a function defined by the prototype

```
int fun(int x, int *p);
```

then we can declare and assign a pointer `fp` to this function by the following code sequence:

```
(int) (*fp)(int, int *);
fp  = fun;
```

We can then invoke the function using this pointer:

```
int y = 1;
int result = fp(3, &y);
```

The value of a function pointer is the address of the first instruction in the machine-code representation of the function.

New to C? Function pointers

The syntax for declaring function pointers is especially difficult for novice programmers to understand. For a declaration such as

```
int (*f)(int*);
```

it helps to read it starting from the inside (starting with "`f`") and working outward. Thus, we see that `f` is a pointer, as indicated by "`(*f)`." It is a pointer to a function that has a single `int *` as an argument, as indicated by "`(*f)(int*)`". Finally, we see that it is a pointer to a function that takes an `int *` as an argument and returns `int`.

The parentheses around `*f` are required, because otherwise the declaration

```
int *f(int*);
```

would be read as

```
(int *) f(int*);
```

That is, it would be interpreted as a function prototype, declaring a function `f` that has an `int *` as its argument and returns an `int *`.

Kernighan & Ritchie [58, Sect. 5.12] present a helpful tutorial on reading C declarations.

3.11 Life in the Real World: Using the GDB Debugger

The GNU debugger GDB provides a number of useful features to support the run-time evaluation and analysis of machine-level programs. With the examples and exercises in this book, we attempt to infer the behavior of a program by just looking at the code. Using GDB, it becomes possible to study the behavior by watching the program in action, while having considerable control over its execution.

Figure 3.30 shows examples of some GDB commands that help when working with machine-level, IA32 programs. It is very helpful to first run OBJDUMP to get a disassembled version of the program. Our examples are based on running GDB on the file `prog`, described and disassembled on page 164. We start GDB with the following command line:

```
unix> gdb prog
```

The general scheme is to set breakpoints near points of interest in the program. These can be set to just after the entry of a function, or at a program address. When one of the breakpoints is hit during program execution, the program will halt and return control to the user. From a breakpoint, we can examine different registers and memory locations in various formats. We can also single-step the program, running just a few instructions at a time, or we can proceed to the next breakpoint.

As our examples suggest, GDB has an obscure command syntax, but the online help information (invoked within GDB with the `help` command) overcomes this shortcoming. Rather than using the command-line interface to GDB, many programmers prefer using DDD, an extension to GDB that provides a graphic user interface.

Web Aside ASM:OPT Machine code generated with higher levels of optimization

In our presentation, we have looked at machine code generated with level-one optimization (specified with the command-line option '`-O1`'). In practice, most heavily used programs are compiled with higher levels of optimization. For example, all of the GNU libraries and packages are compiled with level-two optimization, specified with the command-line option '`-O2`'.

Recent versions of GCC employ an extensive set of optimizations at level two, making the mapping between the source code and the generated code more difficult to discern. Here are some examples of the optimizations that can be found at level two:

- The control structures become more entangled. Most procedures have multiple return points, and the stack management code to set up and complete a function is intermixed with the code implementing the operations of the procedure.
- Procedure calls are often *inlined*, replacing them by the instructions implementing the procedures. This eliminates much of the overhead involved in calling and returning from a function, and it enables optimizations that are specific to individual function calls. On the other hand, if we try to set a breakpoint for a function in a debugger, we might never encounter a call to this function.

Command	Effect
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command line arguments here)
`kill`	Stop your program
Breakpoints	
`break sum`	Set breakpoint at entry to function `sum`
`break *0x8048394`	Set breakpoint at address `0x8048394`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas sum`	Disassemble function `sum`
`disas 0x8048397`	Disassemble function around address `0x8048397`
`disas 0x8048394 0x80483a4`	Disassemble code within specified address range
`print /x $eip`	Print program counter in hex
Examining data	
`print $eax`	Print contents of `%eax` in decimal
`print /x $eax`	Print contents of `%eax` in hex
`print /t $eax`	Print contents of `%eax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($ebp+8)`	Print contents of `%ebp` plus 8 in hex
`print *(int *) 0xfff076b0`	Print integer at address `0xfff076b0`
`print *(int *) ($ebp+8)`	Print integer at address `%ebp` + 8
`x/2w 0xfff076b0`	Examine two (4-byte) words starting at address `0xfff076b0`
`x/20b sum`	Examine first 20 bytes of function `sum`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.30 **Example GDB commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

- Recursion is often replaced by iteration. For example, the recursive factorial function `rfact` (Figure 3.25) is compiled into code very similar to that generated for the `while` loop implementation (Figure 3.15). Again, this can lead to some surprises when we try to monitor program execution with a debugger.

These optimizations can significantly improve program performance, but they make the mapping between source and machine code much more difficult to discern. This can make the programs more difficult to debug. Nonetheless, these higher level optimizations have now become standard, and so those who study programs at the machine level must become familiar with the possible optimizations they may encounter.

3.12 Out-of-Bounds Memory References and Buffer Overflow

We have seen that C does not perform any bounds checking for array references, and that local variables are stored on the stack along with state information such as saved register values and return addresses. This combination can lead to serious program errors, where the state stored on the stack gets corrupted by a write to an out-of-bounds array element. When the program then tries to reload the register or execute a `ret` instruction with this corrupted state, things can go seriously wrong.

A particularly common source of state corruption is known as *buffer overflow*. Typically some character array is allocated on the stack to hold a string, but the size of the string exceeds the space allocated for the array. This is demonstrated by the following program example:

```
1   /* Sample implementation of library function gets() */
2   char *gets(char *s)
3   {
4       int c;
5       char *dest = s;
6       int gotchar = 0; /* Has at least one character been read? */
7       while ((c = getchar()) != '\n' && c != EOF) {
8           *dest++ = c; /* No bounds checking! */
9           gotchar = 1;
10      }
11      *dest++ = '\0';  /* Terminate string */
12      if (c == EOF && !gotchar)
13          return NULL; /* End of file or error */
14      return s;
15  }
16
```

```
17  /* Read input line and write it back */
18  void echo()
19  {
20      char buf[8];  /* Way too small! */
21      gets(buf);
22      puts(buf);
23  }
```

The preceding code shows an implementation of the library function `gets` to demonstrate a serious problem with this function. It reads a line from the standard input, stopping when either a terminating newline character or some error condition is encountered. It copies this string to the location designated by argument `s`, and terminates the string with a null character. We show the use of `gets` in the function `echo`, which simply reads a line from standard input and echoes it back to standard output.

The problem with `gets` is that it has no way to determine whether sufficient space has been allocated to hold the entire string. In our `echo` example, we have purposely made the buffer very small—just eight characters long. Any string longer than seven characters will cause an out-of-bounds write.

Examining the assembly code generated by GCC for `echo` shows how the stack is organized.

```
1   echo:
2     pushl   %ebp              Save %ebp on stack
3     movl    %esp, %ebp
4     pushl   %ebx              Save %ebx
5     subl    $20, %esp         Allocate 20 bytes on stack
6     leal    -12(%ebp), %ebx   Compute buf as %ebp-12
7     movl    %ebx, (%esp)      Store buf at top of stack
8     call    gets              Call gets
9     movl    %ebx, (%esp)      Store buf at top of stack
10    call    puts              Call puts
11    addl    $20, %esp         Deallocate stack space
12    popl    %ebx              Restore %ebx
13    popl    %ebp              Restore %ebp
14    ret                       Return
```

We can see in this example that the program stores the contents of registers `%ebp` and `%ebx` on the stack, and then allocates an additional 20 bytes by subtracting 20 from the stack pointer (line 5). The location of character array `buf` is computed as 12 bytes below `%ebp` (line 6), just below the stored value of `%ebx`, as illustrated in Figure 3.31. As long as the user types at most seven characters, the string returned by `gets` (including the terminating null) will fit within the space allocated for `buf`. A longer string, however, will cause `gets` to overwrite some of the information

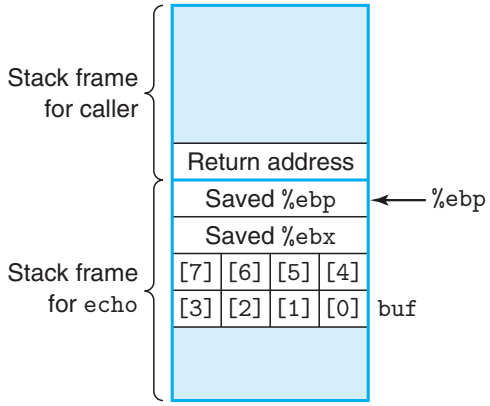


Figure 3.31
Stack organization for `echo` function. Character array `buf` is just below part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

stored on the stack. As the string gets longer, the following information will get corrupted:

Characters typed	Additional corrupted state
0–7	None
8–11	Saved value of %ebx
12–15	Saved value of %ebp
16–19	Return address
20+	Saved state in caller

As this table indicates, the corruption is cumulative—as the number of characters increases, more state gets corrupted. Depending on which portions of the state are affected, the program can misbehave in several different ways:

- If the stored value of %ebx is corrupted, then this register will not be restored properly in line 12, and so the caller will not be able to rely on the integrity of this register, even though it should be callee-saved.
- If the stored value of %ebp is corrupted, then this register will not be restored properly on line 13, and so the caller will not be able to reference its local variables or parameters properly.
- If the stored value of the return address is corrupted, then the `ret` instruction (line 14) will cause the program to jump to a totally unexpected location.

None of these behaviors would seem possible based on the C code. The impact of out-of-bounds writing to memory by functions such as `gets` can only be understood by studying the program at the machine-code level.

Our code for `echo` is simple but sloppy. A better version involves using the function `fgets`, which includes as an argument a count on the maximum number of bytes to read. Problem 3.68 asks you to write an echo function that can handle an input string of arbitrary length. In general, using `gets` or any function that can overflow storage is considered a bad programming practice. The C compiler even produces the following error message when compiling a file containing a call to `gets`: "The `gets` function is dangerous and should not be used." Unfortunately,

a number of commonly used library functions, including `strcpy`, `strcat`, and `sprintf`, have the property that they can generate a byte sequence without being given any indication of the size of the destination buffer [94]. Such conditions can lead to vulnerabilities to buffer overflow.

Practice Problem 3.43

Figure 3.32 shows a (low-quality) implementation of a function that reads a line from standard input, copies the string to newly allocated storage, and returns a pointer to the result.

Consider the following scenario. Procedure `getline` is called with the return address equal to `0x8048643`, register `%ebp` equal to `0xbffffc94`, register `%ebx` equal to `0x1`, register `%edi` is equal to `0x2`, and register `%esi` is equal to `0x3`. You type in the string " `0123456789012345678901 23`". The program terminates with

(a) C code

```
1   /* This is very low-quality code.
2      It is intended to illustrate bad programming practices.
3      See Problem 3.43. */
4   char *getline()
5   {
6       char buf[8];
7       char *result;
8       gets(buf);
9       result = malloc(strlen(buf));
10      strcpy(result, buf);
11      return result;
12  }
```

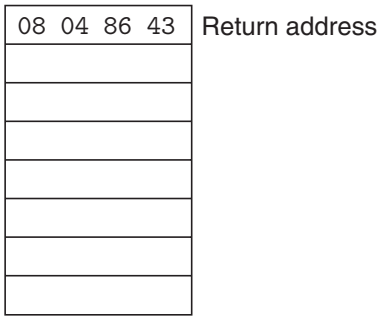
(b) Disassembly up through call to `gets`

```
1   080485c0 <getline>:
2    80485c0:  55                    push   %ebp
3    80485c1:  89 e5                 mov    %esp,%ebp
4    80485c3:  83 ec 28              sub    $0x28,%esp
5    80485c6:  89 5d f4              mov    %ebx,-0xc(%ebp)
6    80485c9:  89 75 f8              mov    %esi,-0x8(%ebp)
7    80485cc:  89 7d fc              mov    %edi,-0x4(%ebp)
     Diagram stack at this point
8    80485cf:  8d 75 ec              lea    -0x14(%ebp),%esi
9    80485d2:  89 34 24              mov    %esi,(%esp)
10   80485d5:  e8 a3 ff ff ff        call   804857d <gets>
     Modify diagram to show stack contents at this point
```

Figure 3.32 **C and disassembled code for Problem 3.43.**

a segmentation fault. You run GDB and determine that the error occurs during the execution of the `ret` instruction of `getline`.

A. Fill in the diagram that follows, indicating as much as you can about the stack just after executing the instruction at line 7 in the disassembly. Label the quantities stored on the stack (e.g., "Return address") on the right, and their hexadecimal values (if known) within the box. Each box represents 4 bytes. Indicate the position of `%ebp`.



B. Modify your diagram to show the effect of the call to `gets` (line 10).

C. To what address does the program attempt to return?

D. What register(s) have corrupted value(s) when `getline` returns?

E. Besides the potential for buffer overflow, what two other things are wrong with the code for `getline`?

A more pernicious use of buffer overflow is to get a program to perform a function that it would otherwise be unwilling to do. This is one of the most common methods to attack the security of a system over a computer network. Typically, the program is fed with a string that contains the byte encoding of some executable code, called the *exploit code*, plus some extra bytes that overwrite the return address with a pointer to the exploit code. The effect of executing the `ret` instruction is then to jump to the exploit code.

In one form of attack, the exploit code then uses a system call to start up a shell program, providing the attacker with a range of operating system functions. In another form, the exploit code performs some otherwise unauthorized task, repairs the damage to the stack, and then executes `ret` a second time, causing an (apparently) normal return to the caller.

As an example, the famous Internet worm of November 1988 used four different ways to gain access to many of the computers across the Internet. One was a buffer overflow attack on the finger daemon `fingerd`, which serves requests by the FINGER command. By invoking FINGER with an appropriate string, the worm could make the daemon at a remote site have a buffer overflow and execute code that gave the worm access to the remote system. Once the worm gained access to a system, it would replicate itself and consume virtually all of the machine's computing resources. As a consequence, hundreds of machines were effectively paralyzed until security experts could determine how to eliminate the worm. The author of

the worm was caught and prosecuted. He was sentenced to 3 years probation, 400 hours of community service, and a $10,500 fine. Even to this day, however, people continue to find security leaks in systems that leave them vulnerable to buffer overflow attacks. This highlights the need for careful programming. Any interface to the external environment should be made "bullet proof" so that no behavior by an external agent can cause the system to misbehave.

Aside Worms and viruses

Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [102], a *worm* is a program that can run by itself and can propagate a fully working version of itself to other machines. A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently. In the popular press, the term "virus" is used to refer to a variety of different strategies for spreading attacking code among systems, and so you will hear people saying "virus" for what more properly should be called a "worm."

3.12.1 Thwarting Buffer Overflow Attacks

Buffer overflow attacks have become so pervasive and have caused so many problems with computer systems that modern compilers and operating systems have implemented mechanisms to make it more difficult to mount these attacks and to limit the ways by which an intruder can seize control of a system via a buffer overflow attack. In this section, we will present ones that are provided by recent versions of GCC for Linux.

Stack Randomization

In order to insert exploit code into a system, the attacker needs to inject both the code as well as a pointer to this code as part of the attack string. Generating this pointer requires knowing the stack address where the string will be located. Historically, the stack addresses for a program were highly predictable. For all systems running the same combination of program and operating system version, the stack locations were fairly stable across many machines. So, for example, if an attacker could determine the stack addresses used by a common Web server, it could devise an attack that would work on many machines. Using infectious disease as an analogy, many systems were vulnerable to the exact same strain of a virus, a phenomenon often referred to as a *security monoculture* [93].

The idea of *stack randomization* is to make the position of the stack vary from one run of a program to another. Thus, even if many machines are running identical code, they would all be using different stack addresses. This is implemented by allocating a random amount of space between 0 and n bytes on the stack at the start of a program, for example, by using the allocation function `alloca`, which allocates space for a specified number of bytes on the stack. This allocated space is not used by the program, but it causes all subsequent stack locations to vary from one execution of a program to another. The allocation range n needs to be large enough to get sufficient variations in the stack addresses, yet small enough that it does not waste too much space in the program.

The following code shows a simple way to determine a "typical" stack address:

```
1  int main() {
2      int local;
3      printf("local at %p\n", &local);
4      return 0;
5  }
```

This code simply prints the address of a local variable in the `main` function. Running the code 10,000 times on a Linux machine in 32-bit mode, the addresses ranged from `0xff7fa7e0` to `0xffffd7e0`, a range of around 2^{23}. By comparison, running on an older Linux system, the same address occurred every time. Running in 64-bit mode on the newer machine, the addresses ranged from `0x7fff00241914` to `0x7ffffff98664`, a range of nearly 2^{32}.

Stack randomization has become standard practice in Linux systems. It is one of a larger class of techniques known as *address-space layout randomization*, or ASLR [95]. With ASLR, different parts of the program, including program code, library code, stack, global variables, and heap data, are loaded into different regions of memory each time a program is run. That means that a program running on one machine will have very different address mappings than the same program running on other machines. This can thwart some forms of attack.

Overall, however, a persistent attacker can overcome randomization by brute force, repeatedly attempting attacks with different addresses. A common trick is to include a long sequence of `nop` (pronounced "no op," short for "no operation") instructions before the actual exploit code. Executing this instruction has no effect, other than incrementing the program counter to the next instruction. As long as the attacker can guess an address somewhere within this sequence, the program will run through the sequence and then hit the exploit code. The common term for this sequence is a "nop sled" [94], expressing the idea that the program "slides" through the sequence. If we set up a 256-byte nop sled, then the randomization over $n = 2^{23}$ can be cracked by enumerating $2^{15} = 32{,}768$ starting addresses, which is entirely feasible for a determined attacker. For the 64-bit case, trying to enumerate $2^{24} = 16{,}777{,}216$ is a bit more daunting. We can see that stack randomization and other aspects of ASLR can increase the effort required to successfully attack a system, and therefore greatly reduce the rate at which a virus or worm can spread, but it cannot provide a complete safeguard.

Practice Problem 3.44

Running our stack-checking code 10,000 times on a system running Linux version 2.6.16, we obtained addresses ranging from a minimum of `0xffffb754` to a maximum of `0xffffd754`.

A. What is the approximate range of addresses?

B. If we attempted a buffer overrun with a 128-byte nop sled, how many attempts would it take to exhaustively test all starting addresses?

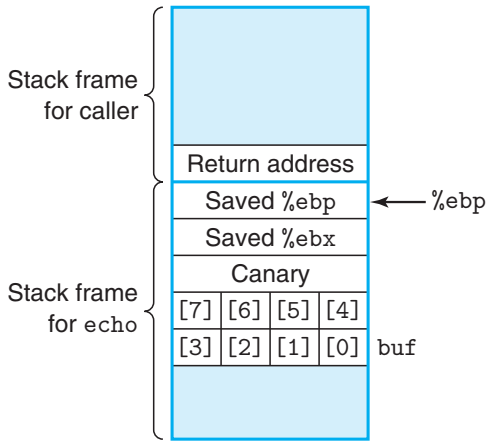


Figure 3.33
Stack organization for `echo` function with stack protector enabled. A special "canary" value is positioned between array `buf` and the saved state. The code checks the canary value to determine whether or not the stack state has been corrupted.

Stack Corruption Detection

A second line of defense is to be able to detect when a stack has been corrupted. We saw in the example of the `echo` function (Figure 3.31) that the corruption typically occurs when we overrun the bounds of a local buffer. In C, there is no reliable way to prevent writing beyond the bounds of an array. Instead, we can try to detect when such a write has occurred before any harmful effects can occur.

Recent versions of GCC incorporate a mechanism known as *stack protector* into the generated code to detect buffer overruns. The idea is to store a special *canary* value[4] in the stack frame between any local buffer and the rest of the stack state, as illustrated in Figure 3.33 [32, 94]. This canary value, also referred to as a *guard value*, is generated randomly each time the program is run, and so there is no easy way for an attacker to determine what it is. Before restoring the register state and returning from the function, the program checks if the canary has been altered by some operation of this function or one that it has called. If so, the program aborts with an error.

Recent versions of GCC try to determine whether a function is vulnerable to a stack overflow, and insert this type of overflow detection automatically. In fact, for our earlier demonstration of stack overflow, we had to give the command-line option "`-fno-stack-protector`" to prevent GCC from inserting this code. When we compile the function `echo` without this option, and hence with stack protector enabled, we get the following assembly code:

```
1   echo:
2     pushl   %ebp
3     movl    %esp, %ebp
4     pushl   %ebx
5     subl    $20, %esp
6     movl    %gs:20, %eax      Retrieve canary
7     movl    %eax, -8(%ebp)    Store on stack
```

4. The term "canary" refers to the historic use of these birds to detect the presence of dangerous gasses in coal mines.

```
 8     xorl    %eax, %eax          Zero out register
 9     leal    -16(%ebp), %ebx     Compute buf as %ebp-16
10     movl    %ebx, (%esp)        Store buf at top of stack
11     call    gets                Call gets
12     movl    %ebx, (%esp)        Store buf at top of stack
13     call    puts                Call puts
14     movl    -8(%ebp), %eax      Retrieve canary
15     xorl    %gs:20, %eax        Compare to stored value
16     je      .L19                If =, goto ok
17     call    __stack_chk_fail    Stack corrupted!
18   .L19:                       ok:
19     addl    $20, %esp           Normal return ...
20     popl    %ebx
21     popl    %ebp
22     ret
```

We see that this version of the function retrieves a value from memory (line 6) and stores it on the stack at offset −8 from `%ebp`. The instruction argument `%gs:20` is an indication that the canary value is read from memory using *segmented addressing*, an addressing mechanism that dates back to the 80286 and is seldom found in programs running on modern systems. By storing the canary in a special segment, it can be marked as "read only," so that an attacker cannot overwrite the stored canary value. Before restoring the register state and returning, the function compares the value stored at the stack location with the canary value (via the `xorl` instruction on line 15.) If the two are identical, the `xorl` instruction will yield 0, and the function will complete in the normal fashion. A nonzero value indicates that the canary on the stack has been modified, and so the code will call an error routine.

Stack protection does a good job of preventing a buffer overflow attack from corrupting state stored on the program stack. It incurs only a small performance penalty, especially because GCC only inserts it when there is a local buffer of type `char` in the function. Of course, there are other ways to corrupt the state of an executing program, but reducing the vulnerability of the stack thwarts many common attack strategies.

Practice Problem 3.45

The function `intlen`, along with the functions `len` and `iptoa`, provides a very convoluted way of computing the number of decimal digits required to represent an integer. We will use this as a way to study some aspects of the GCC stack protector facility.

```
int len(char *s) {
    return strlen(s);
}

void iptoa(char *s, int *p)
```

```
{
    int val = *p;
    sprintf(s, "%d", val);
}

int intlen(int x) {
    int v;
    char buf[12];
    v = x;
    iptoa(buf, &v);
    return len(buf);
}
```

The following show portions of the code for `intlen`, compiled both with and without stack protector:

Without protector

```
1     subl    $36, %esp
2     movl    8(%ebp), %eax
3     movl    %eax, -8(%ebp)
4     leal    -8(%ebp), %eax
5     movl    %eax, 4(%esp)
6     leal    -20(%ebp), %ebx
7     movl    %ebx, (%esp)
8     call    iptoa
```

With protector

```
1     subl    $52, %esp
2     movl    %gs:20, %eax
3     movl    %eax, -8(%ebp)
4     xorl    %eax, %eax
5     movl    8(%ebp), %eax
6     movl    %eax, -24(%ebp)
7     leal    -24(%ebp), %eax
8     movl    %eax, 4(%esp)
9     leal    -20(%ebp), %ebx
10    movl    %ebx, (%esp)
11    call    iptoa
```

A. For both versions: What are the positions in the stack frame for `buf`, `v`, and (when present) the canary value?

B. How would the rearranged ordering of the local variables in the protected code provide greater security against a buffer overrun attack?

Limiting Executable Code Regions

A final step is to eliminate the ability of an attacker to insert executable code into a system. One method is to limit which memory regions hold executable code. In typical programs, only the portion of memory holding the code generated by the compiler need be executable. The other portions can be restricted to allow just reading and writing. As we will see in Chapter 9, the virtual memory space is logically divided into *pages*, typically with 2048 or 4096 bytes per page. The hardware supports different forms of *memory protection*, indicating the forms of access allowed by both user programs and by the operating system kernel. Many systems allow control over three forms of access: read (reading data from memory), write (storing data into memory), and execute (treating the memory contents as machine-level code). Historically, the x86 architecture merged the read and execute access controls into a single 1-bit flag, so that any page marked as readable was also executable. The stack had to be kept both readable and writable, and therefore the bytes on the stack were also executable. Various schemes were implemented to be able to limit some pages to being readable but not executable, but these generally introduced significant inefficiencies.

More recently, AMD introduced an "NX" (for "no-execute") bit into the memory protection for its 64-bit processors, separating the read and execute access modes, and Intel followed suit. With this feature, the stack can be marked as being readable and writable, but not executable, and the checking of whether a page is executable is performed in hardware, with no penalty in efficiency.

Some types of programs require the ability to dynamically generate and execute code. For example, "just-in-time" compilation techniques dynamically generate code for programs written in interpreted languages, such as Java, to improve execution performance. Whether or not we can restrict the executable code to just that part generated by the compiler in creating the original program depends on the language and the operating system.

The techniques we have outlined—randomization, stack protection, and limiting which portions of memory can hold executable code—are three of the most common mechanisms used to minimize the vulnerability of programs to buffer overflow attacks. They all have the properties that they require no special effort on the part of the programmer and incur very little or no performance penalty. Each separately reduces the level of vulnerability, and in combination they become even more effective. Unfortunately, there are still ways to attack computers [81, 94], and so worms and viruses continue to compromise the integrity of many machines.

Web Aside ASM:EASM Combining assembly code with C programs

Although a C compiler does a good job of converting the computations we express in a program into machine code, there are some features of a machine that cannot be accessed by a C program. For example, IA32 machines have a condition code `PF` (for "parity flag") that is set to 1 when there is an even number of ones in the low-order 8 bits of the computed result. Computing this information in C

requires at least seven shifting, masking, and exclusive-or operations (see Problem 2.65). It is ironic that the hardware performs this computation as part of every arithmetic or logical operation, but there is no way for a C program to determine the value of the PF condition code.

There are two ways to incorporate assembly code into C programs. First, we can write an entire function as a separate assembly-code file and let the assembler and linker combine this with code we have written in C. Second, we can use the *inline assembly* feature of GCC, where brief sections of assembly code can be incorporated into a C program using the `asm` directive. This approach has the advantage that it minimizes the amount of machine-specific code.

Of course, including assembly code in a C program makes the code specific to a particular class of machines (such as IA32), and so it should only be used when the desired feature can only be accessed in this way.

3.13 x86-64: Extending IA32 to 64 Bits

Intel's IA32 instruction set architecture (ISA) has been the dominant instruction format for the world's computers for many years. IA32 has been the platform of choice for most Windows, Linux, and, since 2006, even Macintosh computers. The IA32 format used today was, for the most part, defined in 1985 with the introduction of the i386 microprocessor, extending the 16-bit instruction set defined by the original 8086 to 32 bits. Even though subsequent processor generations have introduced new instruction types and formats, many compilers, including GCC, have avoided using these features in the interest of maintaining backward compatibility. For example, we saw in Section 3.6.6 that the conditional move instructions, introduced by Intel in 1995, can yield significant efficiency improvements over more traditional conditional branches, yet in most configurations GCC will not generate these instructions.

A shift is underway to a 64-bit version of the Intel instruction set. Originally developed by Advanced Micro Devices (AMD) and named *x86-64*, it is now supported by most processors from AMD (who now call it *AMD64*) and by Intel, who refer to it as *Intel64*. Most people still refer to it as "x86-64," and we follow this convention. (Some vendors have shortened this to simply "x64".) Newer versions of Linux and Windows support this extension, although systems still run only 32-bit versions of these operating systems. In extending GCC to support x86-64, the developers saw an opportunity to also make use of some of the instruction-set features that had been added in more recent generations of IA32 processors.

This combination of new hardware and revised compiler makes x86-64 code substantially different in form and in performance than IA32 code. In creating the 64-bit extension, the AMD engineers adopted some of the features found in reduced instruction set computers (RISC) [49] that made them the favored targets for optimizing compilers. For example, there are now 16 general-purpose registers, rather than the performance-limiting 8 of the original 8086. The developers of GCC were able to exploit these features, as well as those of more recent generations of the IA32 architecture, to obtain substantial performance improvements. For example, procedure parameters are now passed via registers rather than on the stack, greatly reducing the number of memory read and write operations.

This section serves as a supplement to our description of IA32, describing the extensions in both the hardware and the software support to accommodate x86-64. We assume readers are already familiar with IA32. We start with a brief history of how AMD and Intel arrived at x86-64, followed by a summary of the main features that distinguish x86-64 code from IA32 code, and then work our way through the individual features.

3.13.1 History and Motivation for x86-64

Over the many years since introduction of the i386 in 1985, the capabilities of microprocessors have changed dramatically. In 1985, a fully configured high-end desktop computer, such as the Sun-3 workstation sold by Sun Microsystems, had at most 8 megabytes of random-access memory (RAM) and 100 megabytes of disk storage. It used a Motorola 68020 microprocessor (Intel microprocessors of that era did not have the necessary features and performance for high-end machines) with a 12.5-megahertz clock and ran around 4 million instructions per second. Nowadays, a typical high-end desktop system has 4 gigabytes of RAM (512× increase), 1 terabyte of disk storage (10,000× increase), and a nearly 4-gigahertz clock, running around 5 billion instructions per second (1250× increase). Microprocessor-based systems have become pervasive. Even today's supercomputers are based on harnessing the power of many microprocessors computing in parallel. Given these large quantitative improvements, it is remarkable that the world's computing base mostly runs code that is binary compatible with machines that existed back in 1985 (except that they did not have nearly enough memory to handle today's operating systems and applications).

The 32-bit word size of the IA32 has become a major limitation in growing the capacity of microprocessors. Most significantly, the word size of a machine defines the range of virtual addresses that programs can use, giving a 4-gigabyte virtual address space in the case of 32 bits. It is now feasible to buy more than this amount of RAM for a machine, but the system cannot make effective use of it. For applications that involve manipulating large data sets, such as scientific computing, databases, and data mining, the 32-bit word size makes life difficult for programmers. They must write code using *out-of-core* algorithms,[5] where the data reside on disk and are explicitly read into memory for processing.

Further progress in computing technology requires shifting to a larger word size. Following the tradition of growing word sizes by doubling, the next logical step is 64 bits. In fact, 64-bit machines have been available for some time. Digital Equipment Corporation introduced its Alpha processor in 1992, and it became a popular choice for high-end computing. Sun Microsystems introduced a 64-bit version of its SPARC architecture in 1995. At the time, however, Intel was not a serious contender for high-end computers, and so the company was under less pressure to switch to 64 bits.

5. The physical memory of a machine is often referred to as *core memory,* dating to an era when each bit of a random-access memory was implemented with a magnetized ferrite core.

Intel's first foray into 64-bit computers were the Itanium processors, based on a totally new instruction set, known as "IA64." Unlike Intel's historic strategy of maintaining backward compatibility as it introduced each new generation of microprocessor, IA64 is based on a radically new approach jointly developed with Hewlett-Packard. Its *Very Large Instruction Word* (VLIW) format packs multiple instructions into bundles, allowing higher degrees of parallel execution. Implementing IA64 proved to be very difficult, and so the first Itanium chips did not appear until 2001, and these did not achieve the expected level of performance on real applications. Although the performance of Itanium-based systems has improved, they have not captured a significant share of the computer market. Itanium machines can execute IA32 code in a compatibility mode, but not with very good performance. Most users have preferred to make do with less expensive, and often faster, IA32-based systems.

Meanwhile, Intel's archrival, Advanced Micro Devices (AMD), saw an opportunity to exploit Intel's misstep with IA64. For years, AMD had lagged just behind Intel in technology, and so they were relegated to competing with Intel on the basis of price. Typically, Intel would introduce a new microprocessor at a price premium. AMD would come along 6 to 12 months later and have to undercut Intel significantly to get any sales—a strategy that worked but yielded very low profits. In 2003, AMD introduced a 64-bit microprocessor based on its "x86-64" instruction set. As the name implies, x86-64 is an evolution of the Intel instruction set to 64 bits. It maintains full backward compatibility with IA32, but it adds new data formats, as well as other features that enable higher capacity and higher performance. With x86-64, AMD captured some of the high-end market that had historically belonged to Intel. AMD's recent generations of processors have indeed proved very successful as high-performance machines. Most recently, AMD has renamed this instruction set *AMD64*, but "x86-64" persists as a favored name.

Intel realized that its strategy of a complete shift from IA32 to IA64 was not working, and so began supporting their own variant of x86-64 in 2004 with processors in the Pentium 4 Xeon line. Since they had already used the name "IA64" to refer to Itanium, they then faced a difficulty in finding their own name for this 64-bit extension. In the end, they decided to describe x86-64 as an enhancement to IA32, and so they referred to it as *IA32-EM64T*, for "Enhanced Memory 64-bit Technology." In late 2006, they adopted the name *Intel64*.

On the compiler side, the developers of GCC steadfastly maintained binary compatibility with the i386, even as useful features were being added to the IA32 instruction set, including conditional moves and a more modern set of floating-point instructions. These features would only be used when code was compiled with special settings of command-line options. Switching to x86-64 as a target provided an opportunity for GCC to give up backward compatibility and instead exploit these newer features even with standard command-line options.

In this text, we use "IA32" to refer to the combination of hardware and GCC code found in traditional 32-bit versions of Linux running on Intel-based machines. We use "x86-64" to refer to the hardware and code combination running on the newer 64-bit machines from AMD and Intel. In the worlds of Linux and GCC, these two platforms are referred to as "i386" and "x86_64," respectively.

3.13.2 An Overview of x86-64

The combination of the new hardware supplied by Intel and AMD, and the new versions of GCC targeting these machines makes x86-64 code substantially different from that generated for IA32 machines. The main features include:

- Pointers and long integers are 64 bits long. Integer arithmetic operations support 8, 16, 32, and 64-bit data types.
- The set of general-purpose registers is expanded from 8 to 16.
- Much of the program state is held in registers rather than on the stack. Integer and pointer procedure arguments (up to 6) are passed via registers. Some procedures do not need to access the stack at all.
- Conditional operations are implemented using conditional move instructions when possible, yielding better performance than traditional branching code.
- Floating-point operations are implemented using the register-oriented instruction set introduced with SSE version 2, rather than the stack-based approach supported by IA32.

Data Types

Figure 3.34 shows the sizes of different C data types for x86-64, and compares them to the sizes for IA32 (rightmost column). We see that pointers (shown here as data type `char *`) require 8 bytes rather than 4. These are referred to as *quad words* by Intel, since they are 4 times longer than the nominal 16-bit "word." In principle, this gives programs the ability to access 2^{64} bytes, or 16 *exabytes*, of memory (around 18.4×10^{18} bytes). That seems like an astonishing amount of memory, but keep in mind that 4 gigabytes seemed like an extremely large amount of memory when the first 32-bit machines appeared in the late 1970s. In practice, most machines do not really support the full address range—the current

C declaration	Intel data type	Assembly code suffix	x86-64 size (bytes)	IA32 Size
`char`	Byte	`b`	1	1
`short`	Word	`w`	2	2
`int`	Double word	`l`	4	4
`long int`	Quad word	`q`	8	4
`long long int`	Quad word	`q`	8	8
`char *`	Quad word	`q`	8	4
`float`	Single precision	`s`	4	4
`double`	Double precision	`d`	8	8
`long double`	Extended precision	`t`	10/16	10/12

Figure 3.34 **Sizes of standard data types with x86-64.** These are compared to the sizes for IA32. Both long integers and pointers require 8 bytes, as compared to 4 for IA32.

generations of AMD and Intel x86-64 machines support 256 terabytes (2^{48} bytes) of virtual memory—but allocating a full 64 bits for pointers is a good idea for long-term compatibility.

We also see that the prefix "`long`" changes integers to 64 bits, allowing a considerably larger range of values. In fact, data type `long` becomes identical to `long long`. Moreover, the hardware provides registers that can hold 64-bit integers and instructions that can operate on these quad words.

As with IA32, the `long` prefix also changes a floating-point `double` to use the 80-bit format supported by IA32 (Section 2.4.6). These are stored in memory with an allocation of 16 bytes for x86-64, compared to 12 bytes for IA32. This improves the performance of memory read and write operations, which typically fetch 8 or 16 bytes at a time. Whether 12 or 16 bytes are allocated, only the low-order 10 bytes are actually used. Moreover, the `long double` data type is only supported by an older class of floating-point instructions that have some idiosyncratic properties (see Web Aside DATA:IA32-FP), while both the `float` and `double` data types are supported by the more recent SSE instructions. The `long double` data type should only be used by programs requiring the additional precision and range the extended-precision format provides over the double-precision format.

Practice Problem 3.46

As shown in Figure 6.17(b), the cost of DRAM, the memory technology used to implement the main memories of microprocessors, has dropped from around $8,000 per megabyte in 1980 to around $0.06 in 2010, roughly a factor of 1.48 every year, or around 51 every 10 years. Let us assume these trends will continue indefinitely (which may not be realistic), and that our budget for a machine's memory is around $1,000, so that we would have configured a machine with 128 kilobytes in 1980 and with 16.3 gigabytes in 2010.

A. Estimate when our $1,000 budget would pay for 256 terabytes of memory.

B. Estimate when our $1,000 budget would pay for 16 exabytes of memory.

C. How much earlier would these transition points occur if we raised our DRAM budget to $10,000?

Assembly-Code Example

In Section 3.2.3, we presented the IA32 assembly code generated by GCC for a function `simple`. Below is the C code for `simple_l`, similar to `simple`, except that it uses long integers:

```
long int simple_l(long int *xp, long int y)
{
    long int t = *xp + y;
    *xp = t;
    return t;
}
```

When GCC is run on an x86-64 Linux machine with the command line

```
unix> gcc -O1 -S -m32 code.c
```

it generates code that is compatible with any IA32 machine (we annotate the code to highlight which instructions read (R) data from memory and which instructions write (W) data to memory):

```
   IA32 implementation of function simple_l.
   xp at %ebp+8, y at %ebp+12
1   simple_l:
2     pushl   %ebp              Save frame pointer       (W)
3     movl    %esp, %ebp        Create new frame pointer
4     movl    8(%ebp), %edx     Retrieve xp              (R)
5     movl    12(%ebp), %eax    Retrieve yp              (R)
6     addl    (%edx), %eax      Add *xp to get t         (R)
7     movl    %eax, (%edx)      Store t at xp            (W)
8     popl    %ebp              Restore frame pointer    (R)
9     ret                       Return                   (R)
```

When we instruct GCC to generate x86-64 code

```
unix> gcc -O1 -S -m64 code.c
```

(on most machines, the flag `-m64` is not required), we get very different code:

```
   x86-64 version of function simple_l.
   xp in %rdi, y in %rsi
1   simple_l:
2     movq    %rsi, %rax        Copy y
3     addq    (%rdi), %rax      Add *xp to get t  (R)
4     movq    %rax, (%rdi)      Store t at xp     (W)
5     ret                       Return            (R)
```

Some of the key differences include:

- Instead of `movl` and `addl` instructions, we see `movq` and `addq`. The pointers and variables declared as long integers are now 64 bits (quad words) rather than 32 bits (long words).
- We see the 64-bit versions of registers (e.g., `%rsi` and `%rdi`, rather than `%esi` and `%edi`). The procedure returns a value by storing it in register `%rax`.
- No stack frame gets generated in the x86-64 version. This eliminates the instructions that set up (lines 2–3) and remove (line 8) the stack frame in the IA32 code.
- Arguments `xp` and `y` are passed in registers (`%rdi` and `%rsi`, respectively) rather than on the stack. This eliminates the need to fetch the arguments from memory.

The net effect of these changes is that the IA32 code consists of eight instructions making seven memory references (five reads, two writes), while the x86-64 code consists of four instructions making three memory references (two reads, one write). The relative performance of the two versions depends greatly on the hardware on which they are executed. Running on an Intel Pentium 4E, one of the first Intel machines to support x86-64, we found that the IA32 version requires around 18 clock cycles per call to `simple_l`, while the x86-64 version requires only 12. This 50% performance improvement on the same machine with the same C code is quite striking. On a newer Intel Core i7 processor, we found that both versions required around 12 clock cycles, indicating no performance improvement. On other machines we have tried, the performance difference lies somewhere between these two extremes. In general, x86-64 code is more compact, requires fewer memory accesses, and runs more efficiently than the corresponding IA32 code.

3.13.3 Accessing Information

Figure 3.35 shows the set of general-purpose registers under x86-64. Compared to the registers for IA32 (Figure 3.2), we see a number of differences:

- The number of registers has been doubled to 16.
- All registers are 64 bits long. The 64-bit extensions of the IA32 registers are named `%rax`, `%rcx`, `%rdx`, `%rbx`, `%rsi`, `%rdi`, `%rsp`, and `%rbp`. The new registers are named `%r8`–`%r15`.
- The low-order 32 bits of each register can be accessed directly. This gives us the familiar registers from IA32: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, and `%ebp`, as well as eight new 32-bit registers: `%r8d`–`%r15d`.
- The low-order 16 bits of each register can be accessed directly, as is the case for IA32. The *word-size* versions of the new registers are named `%r8w`–`%r15w`.
- The low-order 8 bits of each register can be accessed directly. This is true in IA32 only for the first four registers (`%al`, `%cl`, `%dl`, `%bl`). The byte-size versions of the other IA32 registers are named `%sil`, `%dil`, `%spl`, and `%bpl`. The byte-size versions of the new registers are named `%r8b`–`%r15b`.
- For backward compatibility, the second byte of registers `%rax`, `%rcx`, `%rdx`, and `%rbx` can be directly accessed by instructions having single-byte operands.

As with IA32, most of the registers can be used interchangeably, but there are some special cases. Register `%rsp` has special status, in that it holds a pointer to the top stack element. Unlike in IA32, however, there is no frame pointer register; register `%rbp` is available for use as a general-purpose register. Particular conventions are used for passing procedure arguments via registers and for how registers are to be saved and restored during procedure calls, as is discussed in Section 3.13.4. In addition, some arithmetic instructions make special use of registers `%rax` and `%rdx`.

For the most part, the operand specifiers of x86-64 are just the same as those in IA32 (see Figure 3.3), except that the base and index register identifiers must

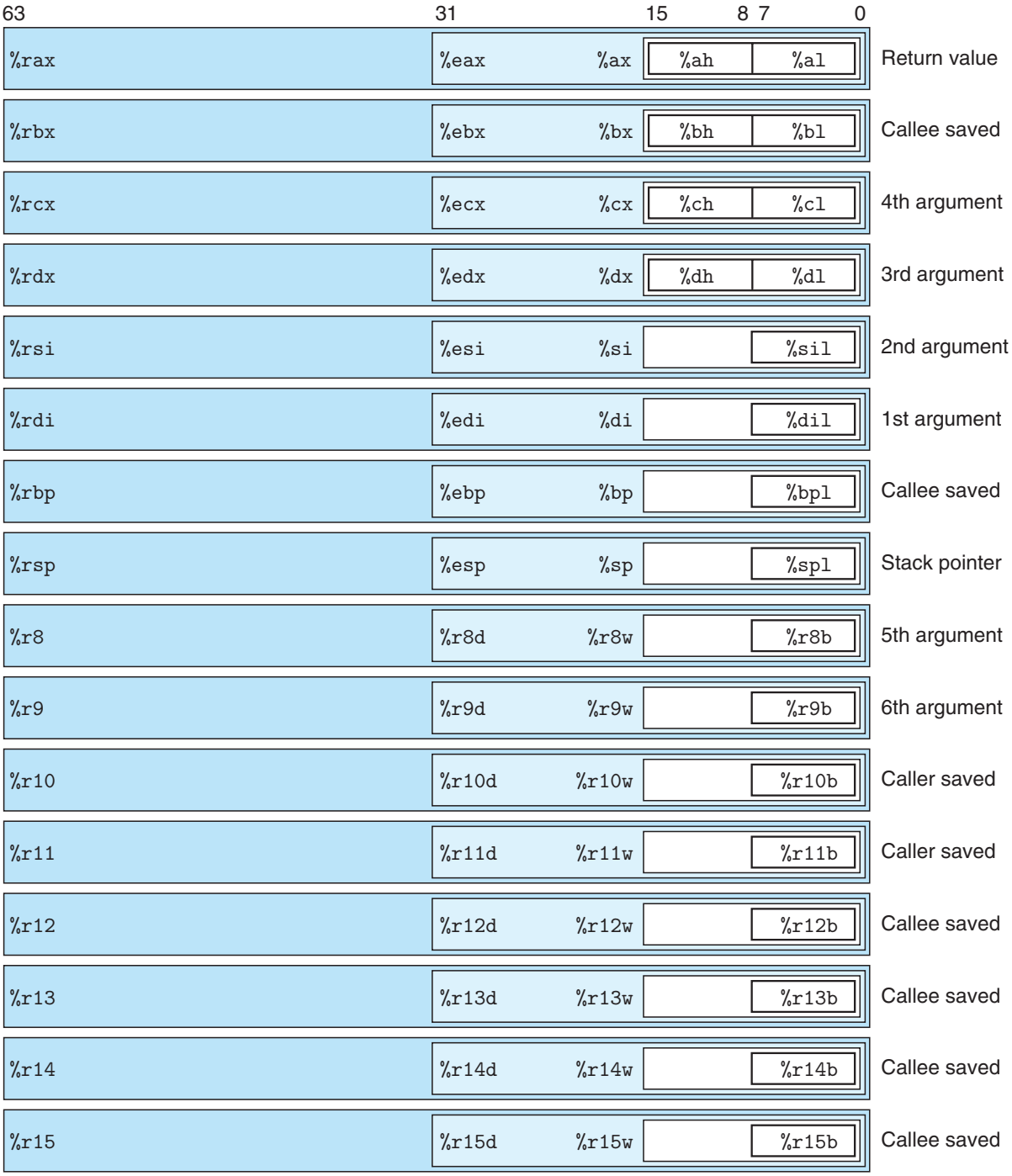


Figure 3.35 **Integer registers.** The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).

use the 'r' version of a register (e.g., %rax) rather than the 'e' version. In addition to the IA32 addressing forms, some forms of *PC-relative* operand addressing are supported. With IA32, this form of addressing is only supported for jump and other control transfer instructions (see Section 3.6.3). This mode is provided to compensate for the fact that the offsets (shown in Figure 3.3 as *Imm*) are only 32 bits long. By viewing this field as a 32-bit two's-complement number, instructions can access data within a window of around $\pm 2.15 \times 10^9$ relative to the program counter. With x86-64, the program counter is named %rip.

As an example of PC-relative data addressing, consider the following procedure, which calls the function simple_l examined earlier:

```
long int gval1 = 567;
long int gval2 = 763;

long int call_simple_l()
{
    long int z = simple_l(&gval1, 12L);
    return z + gval2;
}
```

This code references global variables gval1 and gval2. When this function is compiled, assembled, and linked, we get the following executable code (as generated by the disassembler objdump):

```
1  0000000000400541 <call_simple_l>:
2    400541:  be 0c 00 00 00        mov    $0xc,%esi            Load 12 as 2nd argument
3    400546:  bf 20 10 60 00        mov    $0x601020,%edi       Load &gval1 as 1st argument
4    40054b:  e8 c3 ff ff ff        callq  400513 <simple_l>    Call simple_l
5    400550:  48 03 05 d1 0a 20 00  add    0x200ad1(%rip),%rax  Add gval2 to result
6    400557:  c3                    retq                        Return
```

The instruction on line 3 stores the address of global variable gval1 in register %rdi. It does this by copying the constant value 0x601020 into register %edi. The upper 32 bits of %rdi are automatically set to zero. The instruction on line 5 retrieves the value of gval2 and adds it to the value returned by the call to simple_l. Here we see PC-relative addressing—the immediate value 0x200ad1 is added to the address of the following instruction to get 0x200ad1 + 0x400557 = 0x601028.

Figure 3.36 documents some of the data movement instructions available with x86-64 beyond those found in IA32 (see Figure 3.4). Some instructions require the destination to be a register, indicated by *R*. Others can have either a register or a memory location as destination, indicated by *D*. Most of these instructions fall within a class of instructions seen with IA32. The movabsq instruction, on the other hand, has no counterpart in IA32. This instruction can copy a full 64-bit immediate value to its destination register. When the movq instruction has an immediate value as its source operand, it is limited to a 32-bit value, which is sign-extended to 64 bits.

Instruction		Effect	Description
`movabsq`	I, R	$R \leftarrow I$	Move absolute quad word
MOV	S, D	$D \leftarrow S$	Move
`movq`		Move quad word	
MOVS	S, D	$D \leftarrow$ SignExtend(S)	Move with sign extension
`movsbq`		Move sign-extended byte to quad word	
`movswq`		Move sign-extended word to quad word	
`movslq`		Move sign-extended double word to quad word	
MOVZ	S, D	$D \leftarrow$ ZeroExtend(S)	Move with zero extension
`movzbq`		Move zero-extended byte to quad word	
`movzwq`		Move zero-extended word to quad word	
`pushq`	S	R[%rsp] $\leftarrow$ R[%rsp] $-$ 8; M[R[%rsp]] $\leftarrow S$	Push quad word
`popq`	D	$D \leftarrow$ M[R[%rsp]]; R[%rsp] $\leftarrow$ R[%rsp] $+$ 8	Pop quad word

Figure 3.36 **Data movement instructions.** These supplement the movement instructions of IA32 (Figure 3.4). The `movabsq` instruction only allows immediate data (shown as I) as the source value. Others allow immediate data, a register, or memory (shown as S). Some instructions require the destination to be a register (shown as R), while others allow both register and memory destinations (shown as D).

Moving from a smaller data size to a larger one can involve either sign extension (MOVS) or zero extension (MOVZ). Perhaps unexpectedly, instructions that move or generate 32-bit register values also set the upper 32 bits of the register to zero. Consequently there is no need for an instruction `movzlq`. Similarly, the instruction `movzbq` has the exact same behavior as `movzbl` when the destination is a register—both set the upper 56 bits of the destination register to zero. This is in contrast to instructions that generate 8- or 16-bit values, such as `movb`; these instructions do not alter the other bits in the register. The new stack instructions `pushq` and `popq` allow pushing and popping of 64-bit values.

Practice Problem 3.47

The following C function converts an argument of type `src_t` to a return value of type `dst_t`, where these two types are defined using `typedef`:

```
dest_t cvt(src_t x)
{
    dest_t y = (dest_t) x;
    return y;
}
```

Assume argument `x` is in the appropriately named portion of register `%rdi` (i.e., `%rdi`, `%edi`, `%di`, or `%dil`), and that some form of data movement instruction is to be used to perform the type conversion and to copy the value to the appropriately named portion of register `%rax`. Fill in the following table indicating the instruction, the source register, and the destination register for the following combinations of source and destination type:

src_t	dest_t	Instruction	*S*	*D*
long	long	movq	%rdi	%rax
int	long	_____	_____	_____
char	long	_____	_____	_____
unsigned int	unsigned long	_____	_____	_____
unsigned char	unsigned long	_____	_____	_____
long	int	_____	_____	_____
unsigned long	unsigned	_____	_____	_____

Arithmetic Instructions

In Figure 3.7, we listed a number of arithmetic and logic instructions, using a class name, such as "ADD", to represent instructions for different operand sizes, such as `addb` (byte), `addw` (word), and `addl` (long word). To each of these classes we now add instructions that operate on quad words with the suffix 'q'. Examples of these quad-word instructions include `leaq` (load effective address), `incq` (increment), `addq` (add), and `salq` (shift left). These quad-word instructions have the same argument types as their shorter counterparts. As mentioned earlier, instructions that generate 32-bit register results, such as `addl`, also set the upper 32 bits of the register to zero. Instructions that generate 16-bit results, such as `addw`, only affect their 16-bit destination registers, and similarly for instructions that generate 8-bit results. As with the `movq` instruction, immediate operands are limited to 32-values, which are sign extended to 64 bits.

When mixing operands of different sizes, GCC must choose the right combinations of arithmetic instructions, sign extensions, and zero extensions. These depend on subtle aspects of type conversion and the behavior of the instructions for different operand sizes. This is illustrated by the following C function:

```
1    long int gfun(int x, int y)
2    {
3        long int t1 = (long) x + y;   /* 64-bit addition */
4        long int t2 = (long) (x + y); /* 32-bit addition */
5        return t1 | t2;
6    }
```

Given that integers are 32 bits and long integers are 64, the two additions in this function proceed as follows. Recall that casting has higher precedence than addition, and so line 3 calls for x to be converted to 64 bits, and by operand

promotion `y` is also converted. Value `t1` is then computed using 64-bit addition. On the other hand, `t2` is computed in line 4 by performing 32-bit addition and then extending this value to 64 bits.

The assembly code generated for this function is as follows:

```
1   gfun:
    x in %rdi, y in %rsi
2     leal    (%rsi,%rdi), %eax   Compute t2 as 32-bit sum of x and y
      cltq is equivalent to movslq %eax,%rax
3     cltq                        Sign extend to 64 bits
4     movslq  %esi,%rsi           Convert y to long
5     movslq  %edi,%rdi           Convert x to long
6     addq    %rdi, %rsi          Compute t1 (64-bit addition)
7     orq     %rsi, %rax          Set t1 | t2 as return value
8     ret                          Return
```

Local value `t2` is computed with an `leal` instruction (line 2), which uses 32-bit arithmetic. It is then sign-extended to 64 bits using the `cltq` instruction, which we will see is a special instruction equivalent to executing the instruction `movslq %eax,%rax`. The `movslq` instructions on lines 4–5 take the lower 32 bits of the arguments and sign extend them to 64 bits in the same registers. The `addq` instruction on line 6 then performs 64-bit addition to get `t1`.

Practice Problem 3.48

A C function `arithprob` with arguments `a`, `b`, `c`, and `d` has the following body:

```
return a*b + c*d;
```

It compiles to the following x86-64 code:

```
1   arithprob:
2     movslq  %ecx,%rcx
3     imulq   %rdx, %rcx
4     movsbl  %sil,%esi
5     imull   %edi, %esi
6     movslq  %esi,%rsi
7     leaq    (%rcx,%rsi), %rax
8     ret
```

The arguments and return value are all signed integers of various lengths. Arguments `a`, `b`, `c`, and `d` are passed in the appropriate regions of registers `%rdi`, `%rsi`, `%rdx`, and `%rcx`, respectively. Based on this assembly code, write a function prototype describing the return and argument types for `arithprob`.

Figure 3.37 show instructions used to generate the full 128-bit product of two 64-bit words, as well as ones to support 64-bit division. They are similar to their 32-bit counterparts (Figure 3.9). Several of these instructions view the combination

Instruction		Effect	Description
`imulq`	*S*	R[%rdx]:R[%rax] ← *S* × R[%rax]	Signed full multiply
`mulq`	*S*	R[%rdx]:R[%rax] ← *S* × R[%rax]	Unsigned full multiply
`cltq`		R[%rax] ← SignExtend(R[%eax])	Convert %eax to quad word
`cqto`		R[%rdx]:R[%rax] ← SignExtend(R[%rax])	Convert to oct word
`idivq`	*S*	R[%rdx] ← R[%rdx]:R[%rax] mod *S*; R[%rax] ← R[%rdx]:R[%rax] ÷ *S*	Signed divide
`divq`	*S*	R[%rdx] ← R[%rdx]:R[%rax] mod *S*; R[%rax] ← R[%rdx]:R[%rax] ÷ *S*	Unsigned divide

Figure 3.37 **Special arithmetic operations.** These operations support full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

of registers %rdx and %rax as forming a 128-bit *oct word*. For example, the `imulq` and `mulq` instructions store the result of multiplying two 64-bit values—the first as given by the source operand and the second from register %rax.

The two divide instructions `idivq` and `divq` start with %rdx:%rax as the 128-bit dividend and the source operand as the 64-bit divisor. They then store the quotient in register %rax and the remainder in register %rdx. Preparing the dividend depends on whether unsigned (`divq`) or signed (`idivq`) division is to be performed. In the former case, register %rdx is simply set to zero. In the latter case, the instruction `cqto` is used to perform sign extension, copying the sign bit of %rax into every bit of %rdx.[6] Figure 3.37 also shows an instruction `cltq` to sign extend register %eax to %rax.[7] This instruction is just a shorthand for the instruction `movslq %eax,%rax`.

3.13.4 Control

The control instructions and methods of implementing control transfers in x86-64 are the same as those in IA32 (Section 3.6.) As shown in Figure 3.38, two new instructions, `cmpq` and `testq`, are added to compare and test quad words, augmenting those for byte, word, and double word sizes (Figure 3.10). GCC uses both conditional data transfer and conditional control transfer, since all x86-64 machines support conditional moves.

To illustrate the similarity between IA32 and x86-64 code, consider the assembly code generated by compiling an integer factorial function implemented with a `while` loop (Figure 3.15), as is shown in Figure 3.39. As can be seen, these

6. ATT-format instruction `cqto` is called `cqo` in Intel and AMD documentation.

7. Instruction `cltq` is called `cdqe` in Intel and AMD documentation.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
cmpq		Compare quad word	
TEST	S_2, S_1	S_1 & S_2	Test
testq		Test quad word	

Figure 3.38 **64-bit comparison and test instructions.** These instructions set the condition codes without updating any other registers.

(a) IA32 version

```
1   fact_while:
    n at %ebp+8
2     pushl   %ebp              Save frame pointer
3     movl    %esp, %ebp        Create new frame pointer
4     movl    8(%ebp), %edx     Get n
5     movl    $1, %eax          Set result = 1
6     cmpl    $1, %edx          Compare n:1
7     jle     .L7               If <=, goto done
8   .L10:                     loop:
9     imull   %edx, %eax        Compute result *= n
10    subl    $1, %edx          Decrement n
11    cmpl    $1, %edx          Compare n:1
12    jg      .L10              If >, goto loop
13  .L7:                      done:
14    popl    %ebp              Restore frame pointer
15    ret                       Return result
```

(b) x86-64 version

```
1   fact_while:
    n in %rdi
2     movl    $1, %eax          Set result = 1
3     cmpl    $1, %edi          Compare n:1
4     jle     .L7               If <=, goto done
5   .L10:                     loop:
6     imull   %edi, %eax        Compute result *= n
7     subl    $1, %edi          Decrement n
8     cmpl    $1, %edi          Compare n:1
9     jg      .L10              If >, goto loop
10  .L7:                      done:
11    rep                       (See explanation in aside)
12    ret                       Return result
```

Figure 3.39 **IA32 and x86-64 versions of factorial.** Both were compiled from the C code shown in Figure 3.15.

two versions are very similar. They differ only in how arguments are passed (on the stack vs. in registers), and the absence of a stack frame or frame pointer in the x86-64 code.

Aside Why is there a `rep` instruction in this code?

On line 11 of the x86-64 code, we see the instruction `rep` precedes the return instruction `ret`. Looking at the Intel and AMD documentation for the `rep` instruction, we find that it is normally used to implement a repeating string operation [3, 29]. It seems completely inappropriate here. The answer to this puzzle can be seen in AMD's guidelines to compiler writers [1]. They recommend using the combination of `rep` followed by `ret` to avoid making the `ret` instruction be the destination of a conditional jump instruction. Without the `rep` instruction, the `jg` instruction would proceed to the `ret` instruction when the branch is not taken. According to AMD, their processors cannot properly predict the destination of a `ret` instruction when it is reached from a jump instruction. The `rep` instruction serves as a form of no-operation here, and so inserting it as the jump destination does not change behavior of the code, except to make it faster on AMD processors.

Practice Problem 3.49

A function `fun_c` has the following overall structure:

```
long fun_c(unsigned long x) {
    long val = 0;
    int i;
    for ( ________ ; ________ ; ________ ) {
        ________;
    }
    ________;
    return ________;
}
```

The GCC C compiler generates the following assembly code:

```
1   fun_c:
    x in %rdi
2     movl    $0, %ecx
3     movl    $0, %edx
4     movabsq $72340172838076673, %rsi
5   .L2:
6     movq    %rdi, %rax
7     andq    %rsi, %rax
8     addq    %rax, %rcx
9     shrq    %rdi              Shift right by 1
10    addl    $1, %edx
11    cmpl    $8, %edx
12    jne     .L2
```

```
13      movq    %rcx, %rax
14      sarq    $32, %rax
15      addq    %rcx, %rax
16      movq    %rax, %rdx
17      sarq    $16, %rdx
18      addq    %rax, %rdx
19      movq    %rdx, %rax
20      sarq    $8, %rax
21      addq    %rdx, %rax
22      andl    $255, %eax
23      ret
```

Reverse engineer the operation of this code. You will find it useful to convert the decimal constant on line 4 to hexadecimal.

A. Use the assembly-code version to fill in the missing parts of the C code.

B. Describe in English what this code computes.

Procedures

We have already seen in our code samples that the x86-64 implementation of procedure calls differs substantially from that of IA32. By doubling the register set, programs need not be so dependent on the stack for storing and retrieving procedure information. This can greatly reduce the overhead for procedure calls and returns.

Here are some of the highlights of how procedures are implemented with x86-64:

- Arguments (up to the first six) are passed to procedures via registers, rather than on the stack. This eliminates the overhead of storing and retrieving values on the stack.
- The `callq` instruction stores a 64-bit return address on the stack.
- Many functions do not require a stack frame. Only functions that cannot keep all local variables in registers need to allocate space on the stack.
- Functions can access storage on the stack up to 128 bytes beyond (i.e., at a lower address than) the current value of the stack pointer. This allows some functions to store information on the stack without altering the stack pointer.
- There is no frame pointer. Instead, references to stack locations are made relative to the stack pointer. Most functions allocate their total stack storage needs at the beginning of the call and keep the stack pointer at a fixed position.
- As with IA32, some registers are designated as callee-save registers. These must be saved and restored by any procedure that modifies them.

Operand size (bits)	Argument Number 1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Figure 3.40 **Registers for passing function arguments.** The registers are used in a specified order and named according to the argument sizes.

Argument Passing

Up to six integral (i.e., integer and pointer) arguments can be passed via registers. The registers are used in a specified order, with the name used for a register depending on the size of the data type being passed. These are shown in Figure 3.40. Arguments are allocated to these registers according to their ordering in the argument list. Arguments smaller than 64 bits can be accessed using the appropriate subsection of the 64-bit register. For example, if the first argument is 32 bits, it can be accessed as %edi.

As an example of argument passing, consider the following C function having eight arguments:

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

The arguments include a range of different-sized integers (64, 32, 16, and 8 bits) as well as different types of pointers, each of which is 64 bits.

This function is implemented in x86-64 as follows:

```
    x86-64 implementation of function proc
    Arguments passed as follows:
      a1  in %rdi       (64 bits)
      a1p in %rsi       (64 bits)
      a2  in %edx       (32 bits)
      a2p in %rcx       (64 bits)
      a3  in %r8w       (16 bits)
      a3p in %r9        (64 bits)
```

```
     a4  at %rsp+8     (8 bits)
     a4p at %rsp+16    (64 bits)
1    proc:
2      movq    16(%rsp), %r10   Fetch a4p  (64 bits)
3      addq    %rdi, (%rsi)     *a1p += a1 (64 bits)
4      addl    %edx, (%rcx)     *a2p += a2 (32 bits)
5      addw    %r8w, (%r9)      *a3p += a3 (16 bits)
6      movzbl  8(%rsp), %eax    Fetch a4   (8 bits)
7      addb    %al, (%r10)      *a4p += a4 (8 bits)
8      ret
```

The first six arguments are passed in registers, while the last two are at positions 8 and 16 relative to the stack pointer. Different versions of the ADD instruction are used according to the sizes of the operands: `addq` for `a1` (`long`), `addl` for `a2` (`int`), `addw` for `a3` (`short`), and `addb` for `a4` (`char`).

Practice Problem 3.50

A C function `incrprob` has arguments `q`, `t`, and `x` of different sizes, and each may be signed or unsigned. The function has the following body:

```
*t += x;
*q += *t;
```

It compiles to the following x86-64 code:

```
1    incrprob:
2      addl    (%rdx), %edi
3      movl    %edi, (%rdx)
4      movslq  %edi,%rdi
5      addq    %rdi, (%rsi)
6      ret
```

Determine all four valid function prototypes for `incrprob` by determining the ordering and possible types of the three parameters.

Stack Frames

We have already seen that many compiled functions do not require a stack frame. If all of the local variables can be held in registers, and the function does not call any other functions (sometimes referred to as a *leaf procedure*, in reference to the tree structure of procedure calls), then the only need for the stack is to save the return address.

On the other hand, there are several reasons a function may require a stack frame:

- There are too many local variables to hold in registers.
- Some local variables are arrays or structures.

- The function uses the address-of operator (`&`) to compute the address of a local variable.
- The function must pass some arguments on the stack to another function.
- The function needs to save the state of a callee-save register before modifying it.

When any of these conditions hold, we find the compiled code for the function creating a stack frame. Unlike the code for IA32, where the stack pointer fluctuates back and forth as values are pushed and popped, the stack frames for x86-64 procedures usually have a fixed size, set at the beginning of the procedure by decrementing the stack pointer (register `%rsp`). The stack pointer remains at a fixed position during the call, making it possible to access data using offsets relative to the stack pointer. As a consequence, the frame pointer (register `%ebp`) seen in IA32 code is no longer needed.

Whenever one function (the *caller*) calls another (the *callee*), the return address gets pushed onto the stack. By convention, we consider this part of the caller's stack frame, in that it encodes part of the caller's state. But this information gets popped from the stack as control returns to the caller, and so it does not affect the offsets used by the caller for accessing values within the stack frame.

The following function illustrates many aspects of the x86-64 stack discipline. Despite the length of this example, it is worth studying carefully.

```
long int call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

GCC generates the following x86-64 code.

```
    x86-64 implementation of call_proc
 1   call_proc:
 2     subq    $32, %rsp         Allocate 32-byte stack frame
 3     movq    $1, 16(%rsp)      Store 1 in &x1
 4     movl    $2, 24(%rsp)      Store 2 in &x2
 5     movw    $3, 28(%rsp)      Store 3 in &x3
 6     movb    $4, 31(%rsp)      Store 4 in &x4
 7     leaq    24(%rsp), %rcx    Pass &x2 as argument 4
 8     leaq    16(%rsp), %rsi    Pass &x1 as argument 2
 9     leaq    31(%rsp), %rax    Compute &x4
10     movq    %rax, 8(%rsp)     Pass &x4 as argument 8
11     movl    $4, (%rsp)        Pass   4 as argument 7
12     leaq    28(%rsp), %r9     Pass &x3 as argument 6
13     movl    $3, %r8d          Pass   3 as argument 5
14     movl    $2, %edx          Pass   2 as argument 3
15     movl    $1, %edi          Pass   1 as argument 1
```

```
16      call    proc              Call
17      movswl  28(%rsp),%eax     Get x3 and convert to int
18      movsbl  31(%rsp),%edx     Get x4 and convert to int
19      subl    %edx, %eax        Compute x3-x4
20      cltq                      Sign extend to long int
21      movslq  24(%rsp),%rdx     Get x2
22      addq    16(%rsp), %rdx    Compute x1+x2
23      imulq   %rdx, %rax        Compute (x1+x2)*(x3-x4)
24      addq    $32, %rsp         Deallocate stack frame
25      ret                       Return
```

Figure 3.41(a) illustrates the stack frame set up during the execution of `call_proc`. Function `call_proc` allocates 32 bytes on the stack by decrementing the stack pointer. It uses bytes 16–31 to hold local variables `x1` (bytes 16–23), `x2` (bytes 24–27), `x3` (bytes 28–29), and `x4` (byte 31). These allocations are sized according to the variable types. Byte 30 is unused. Bytes 0–7 and 8–15 of the stack frame are used to hold the seventh and eighth arguments to `call_proc`, since there are not enough argument registers. The parameters are allocated eight bytes each, even though parameter `x4` requires only a single byte. In the code for `call_proc`, we can see instructions initializing the local variables and setting up the parameters (both in registers and on the stack) for the call to `call_proc`. After `proc` returns, the local variables are combined to compute the final expression, which is returned in register `%rax`. The stack space is deallocated by simply incrementing the stack pointer before the `ret` instruction.

Figure 3.41(b) illustrates the stack during the execution of `proc`. The `call` instruction pushed the return address onto the stack, and so the stack pointer is shifted down by 8 relative to its position during the execution of `call_proc`.

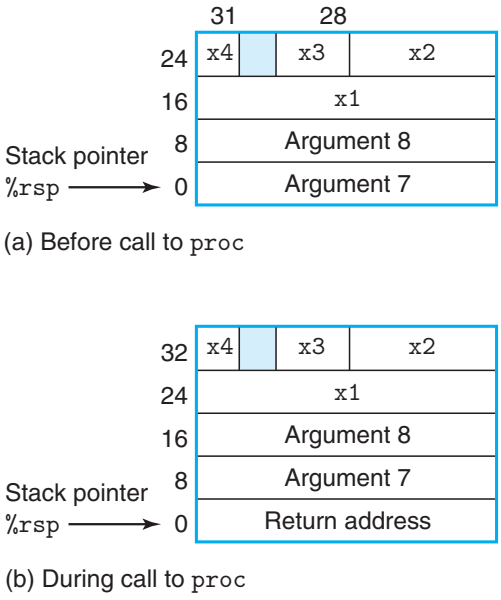


Figure 3.41 **Stack frame structure for `call_proc`.** The frame is required to hold local variables `x1` through `x4`, as well as the seventh and eighth arguments to `proc`. During the execution of `proc` (b), the stack pointer is shifted down by 8.

Hence, within the code for `proc`, arguments 7 and 8 are accessed by offsets of 8 and 16 from the stack pointer.

Observe how `call_proc` changed the stack pointer only once during its execution. GCC determined that 32 bytes would suffice for holding all local variables and for holding the additional arguments to `proc`. Minimizing the amount of movement by the stack pointer simplifies the compiler's task of generating reference to stack elements using offsets from the stack pointer.

Register Saving Conventions

We saw in IA32 (Section 3.7.3) that some registers used for holding temporary values are designated as *caller-saved*, where a function is free to overwrite their values, while others are *callee-saved*, where a function must save their values on the stack before writing to them. With x86-64, the following registers are designated as being callee-saved: `%rbx`, `%rbp`, and `%r12`–`%r15`.

Aside Are there any caller-saved temporary registers?

Of the 16 general-purpose registers, we've seen that 6 are designated for passing arguments, 6 are for callee-saved temporaries, 1 (`%rax`) holds the return value for a function, and 1 (`%rsp`) serves as the stack pointer. Only `%r10` and `%r11` are left as caller-saved temporary registers. Of course, an argument register can be used when there are fewer than six arguments or when the function is done using that argument, and `%rax` can be used multiple times before the final result is generated.

We illustrate the use of callee-saved registers with a somewhat unusual version of a recursive factorial function:

```
/* Compute x! and store at resultp */
void sfact_helper(long int x, long int *resultp)
{
    if (x <= 1)
        *resultp = 1;
    else {
        long int nresult;
        sfact_helper(x-1, &nresult);
        *resultp = x * nresult;
    }
}
```

To compute the factorial of a value x, this function would be called at the top level as follows:

```
long int sfact(long int x)
{
    long int result;
    sfact_helper(x, &result);
    return result;
}
```

The x86-64 code for `sfact_helper` is shown below.

```
   Arguments: x in %rdi, resultp in %rsi
1   sfact_helper:
2     movq    %rbx, -16(%rsp)     Save %rbx (callee save)
3     movq    %rbp, -8(%rsp)      Save %rbp (callee save)
4     subq    $40, %rsp           Allocate 40 bytes on stack
5     movq    %rdi, %rbx          Copy x to %rbx
6     movq    %rsi, %rbp          Copy resultp to %rbp
7     cmpq    $1, %rdi            Compare x:1
8     jg      .L14                If >, goto recur
9     movq    $1, (%rsi)          Store 1 in *resultp
10    jmp     .L16                Goto done
11  .L14:                       recur:
12    leaq    16(%rsp), %rsi      Compute &nresult as second argument
13    leaq    -1(%rdi), %rdi      Compute xm1 = x-1 as first argument
14    call    sfact_helper        Call sfact_helper(xm1, &nresult)
15    movq    %rbx, %rax          Copy x
16    imulq   16(%rsp), %rax      Compute x*nresult
17    movq    %rax, (%rbp)        Store at resultp
18  .L16:                       done:
19    movq    24(%rsp), %rbx      Restore %rbx
20    movq    32(%rsp), %rbp      Restore %rbp
21    addq    $40, %rsp           Deallocate stack
22    ret                         Return
```

Figure 3.42 illustrates how `sfact_helper` uses the stack to store the values of callee-saved registers and to hold the local variable `nresult`. This implementation

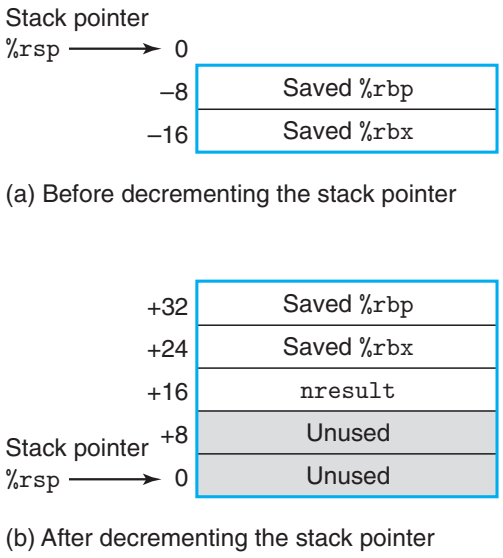


Figure 3.42 **Stack frame for function `sfact_helper`.** This function decrements the stack pointer *after* saving some of the state.

has the interesting feature that the two callee-saved registers it uses (%rbx and %rbp) are saved on the stack (lines 2–3) *before* the stack pointer is decremented (line 4) to allocate the stack frame. As a consequence, the stack offset for %rbx shifts from −16 at the beginning to +24 at the end (line 19). Similarly, the offset for %rbp shifts from −8 to +32.

Being able to access memory beyond the stack pointer is an unusual feature of x86-64. It requires that the virtual memory management system allocate memory for that region. The x86-64 ABI [73] specifies that programs can use the 128 bytes beyond (i.e., at lower addresses than) the current stack pointer. The ABI refers to this area as the *red zone*. It must be kept available for reading and writing as the stack pointer moves.

Practice Problem 3.51

For the C program

```
long int local_array(int i)
{
    long int a[4] = {2L, 3L, 5L, 7L};
    int idx = i & 3;
    return a[idx];
}
```

GCC generates the following code:

```
    x86-64 implementation of local_array
    Argument: i in %edi
1   local_array:
2     movq    $2, -40(%rsp)
3     movq    $3, -32(%rsp)
4     movq    $5, -24(%rsp)
5     movq    $7, -16(%rsp)
6     andl    $3, %edi
7     movq    -40(%rsp,%rdi,8), %rax
8     ret
```

A. Draw a diagram indicating the stack locations used by this function and their offsets relative to the stack pointer.

B. Annotate the assembly code to describe the effect of each instruction.

C. What interesting feature does this example illustrate about the x86-64 stack discipline?

Practice Problem 3.52

For the recursive factorial program

```
long int rfact(long int x)
{
    if (x <= 0)
        return 1;
    else {
        long int xm1 = x-1;
        return x * rfact(xm1);
    }
}
```

GCC generates the following code:

```
    x86-64 implementation of recursive factorial function rfact
    Argument x in %rdi
 1   rfact:
 2     pushq   %rbx
 3     movq    %rdi, %rbx
 4     movl    $1, %eax
 5     testq   %rdi, %rdi
 6     jle     .L11
 7     leaq    -1(%rdi), %rdi
 8     call    rfact
 9     imulq   %rbx, %rax
10   .L11:
11     popq    %rbx
12     ret
```

A. What value does the function store in `%rbx`?

B. What are the purposes of the `pushq` (line 2) and `popq` (line 11) instructions?

C. Annotate the assembly code to describe the effect of each instruction.

D. How does this function manage the stack frame differently from others we have seen?

3.13.5 Data Structures

Data structures follow the same principles in x86-64 as they do in IA32: arrays are allocated as sequences of identically sized blocks holding the array elements, structures are allocated as sequences of variably sized blocks holding the structure elements, and unions are allocated as a single block big enough to hold the largest union element.

One difference is that x86-64 follows a more stringent set of alignment requirements. For any scalar data type requiring K bytes, its starting address must be a multiple of K. Thus, data types `long` and `double` as well as pointers, must be aligned on 8-byte boundaries. In addition, data type `long double` uses a 16-byte alignment (and size allocation), even though the actual representation requires only 10 bytes. These alignment conditions are imposed to improve memory system performance—the memory interface is designed in most processors to read or write aligned blocks that are 8 or 16 bytes long.

Practice Problem 3.53

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement under x86-64.

A. `struct P1 { int i; char c; long j; char d; };`

B. `struct P2 { long i; char c; char d; int j; };`

C. `struct P3 { short w[3]; char c[3] };`

D. `struct P4 { short w[3]; char *c[3] };`

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

3.13.6 Concluding Observations about x86-64

Both AMD and the authors of GCC deserve credit for moving x86 processors into a new era. The formulation of both the x86-64 hardware and the programming conventions changed the processor from one that relied heavily on the stack to hold program state to one where the most heavily used part of the state is held in the much faster and expanded register set. Finally, x86 has caught up to ideas developed for RISC processors in the early 1980s!

Processors capable of running either IA32 or x86-64 code are becoming commonplace. Many current desktop and laptop systems are still running 32-bit versions of their operating systems, and these machines are restricted to running only 32-bit applications, as well. Machines running 64-bit operating systems, and therefore capable of running both 32- and 64-bit applications, have become the widespread choice for high-end machines, such as for database servers and scientific computing. The biggest drawback in transforming applications from 32 bits to 64 bits is that the pointer variables double in size, and since many data structures contain pointers, this means that the overall memory requirement can nearly double. The transition from 32- to 64-bit applications has only occurred for ones having memory needs that exceed the 4-gigabyte address space limitation of IA32. History has shown that applications grow to use all available processing power and memory size, and so we can reliably predict that 64-bit processors running 64-bit operating systems and applications will become increasingly more commonplace.

3.14 Machine-Level Representations of Floating-Point Programs

Thus far, we have only considered programs that represent and operate on integer data types. In order to implement programs that make use of floating-point data, we must have some method of storing floating-point data and additional instructions to operate on floating-point values, to convert between floating-point and integer values, and to perform comparisons between floating-point values. We also require conventions on how to pass floating-point values as function arguments and to return them as function results. We call this combination of storage model, instructions, and conventions the *floating-point architecture* for a machine.

Due to its long evolutionary heritage, x86 processors provide multiple floating-point architectures, of which two are in current use. The first, referred to as "x87," dates back to the earliest days of Intel microprocessors and until recently was the standard implementation. The second, referred to as "SSE," is based on recent additions to x86 processors to support multimedia applications.

Web Aside ASM:X87 The x87 floating-point architecture

The historical x87 floating-point architecture is one of the least elegant features of the x87 architecture. In the original Intel machines, floating point was performed by a separate *coprocessor*, a unit with its own registers and processing capabilities that executes a subset of the instructions. This coprocessor was implemented as a separate chip, named the 8087, 80287, and i387, to accompany the processor chips 8086, 80286, and i386, respectively, and hence the colloquial name "x87." All x86 processors support the x87 architecture, and so this continues to be a possible target for compiling floating-point code.

x87 instructions operate on a shallow stack of floating-point registers. In a stack model, some instructions read values from memory and push them onto the stack; others pop operands from the stack, perform an operation, and then push the result; while others pop values from the stack and store them to memory. This approach has the advantage that there is a simple algorithm by which a compiler can map the evaluation of arithmetic expressions into stack code.

Modern compilers can make many optimizations that do not fit well within a stack model, for example, making use of a single computed result multiple times. Consequently, the x87 architecture implements an odd hybrid between a stack and a register model, where the different elements of the stack can be read and written explicitly, as well as shifted up and down by pushing and popping. In addition, the x87 stack is limited to a depth of eight values; when additional values are pushed, the ones at the bottom are simply discarded. Hence, the compiler must keep track of the stack depth. Furthermore, a compiler must treat all floating-point registers as being caller-save, since their values might disappear off the bottom if other procedures push more values onto the stack.

Web Aside ASM:SSE The SSE floating-point architecture

Starting with the Pentium 4, the SSE2 instruction set, added to support multimedia applications, becomes a viable floating-point architecture for compiled C code. Unlike the stack-based architecture of x87, SSE-based floating point uses a straightforward register-based approach, a much better target

for optimizing compilers. With SSE2, floating-point code is similar to integer code, except that it uses a different set of registers and instructions. When compiling for x86-64, GCC generates SSE code. On the other hand, its default is to generate x87 code for IA32, but it can be directed to generate SSE code by a suitable setting of the command-line parameters.

3.15 Summary

In this chapter, we have peered beneath the layer of abstraction provided by the C language to get a view of machine-level programming. By having the compiler generate an assembly-code representation of the machine-level program, we gain insights into both the compiler and its optimization capabilities, along with the machine, its data types, and its instruction set. In Chapter 5, we will see that knowing the characteristics of a compiler can help when trying to write programs that have efficient mappings onto the machine. We have also gotten a more complete picture of how the program stores data in different memory regions. In Chapter 12, we will see many examples where application programmers need to know whether a program variable is on the run-time stack, in some dynamically allocated data structure, or part of the global program data. Understanding how programs map onto machines makes it easier to understand the differences between these kinds of storage.

Machine-level programs, and their representation by assembly code, differ in many ways from C programs. There is minimal distinction between different data types. The program is expressed as a sequence of instructions, each of which performs a single operation. Parts of the program state, such as registers and the run-time stack, are directly visible to the programmer. Only low-level operations are provided to support data manipulation and program control. The compiler must use multiple instructions to generate and operate on different data structures and to implement control constructs such as conditionals, loops, and procedures. We have covered many different aspects of C and how it gets compiled. We have seen that the lack of bounds checking in C makes many programs prone to buffer overflows. This has made many systems vulnerable to attacks by malicious intruders, although recent safeguards provided by the run-time system and the compiler help make programs more secure.

We have only examined the mapping of C onto IA32 and x86-64, but much of what we have covered is handled in a similar way for other combinations of language and machine. For example, compiling C++ is very similar to compiling C. In fact, early implementations of C++ first performed a source-to-source conversion from C++ to C and generated object-code by running a C compiler on the result. C++ objects are represented by structures, similar to a C `struct`. Methods are represented by pointers to the code implementing the methods. By contrast, Java is implemented in an entirely different fashion. The object code of Java is a special binary representation known as *Java byte code*. This code can be viewed as a machine-level program for a *virtual machine*. As its name suggests, this machine is not implemented directly in hardware. Instead, software interpreters process

the byte code, simulating the behavior of the virtual machine. Alternatively, an approach known as *just-in-time compilation* dynamically translates byte code sequences into machine instructions. This approach provides faster execution when code is executed multiple times, such as in loops. The advantage of using byte code as the low-level representation of a program is that the same code can be "executed" on many different machines, whereas the machine code we have considered runs only on x86 machines.

Bibliographic Notes

Both Intel and AMD provide extensive documentation on their processors. This includes general descriptions of an assembly-language programmer's view of the hardware [2, 27], as well as detailed references about the individual instructions [3, 28, 29]. Reading the instruction descriptions is complicated by the facts that (1) all documentation is based on the Intel assembly-code format, (2) there are many variations for each instruction due to the different addressing and execution modes, and (3) there are no illustrative examples. Still, these remain the authoritative references about the behavior of each instruction.

The organization `amd64.org` has been responsible for defining the *Application Binary Interface* (ABI) for x86-64 code running on Linux systems [73]. This interface describes details for procedure linkages, binary code files, and a number of other features that are required for machine-code programs to execute properly.

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). Blum's book [9] is one of the few references based on ATT format, and it provides an extensive description of how to embed assembly code into C programs using the `asm` directive.

Muchnick's book on compiler design [76] is considered the most comprehensive reference on code-optimization techniques. It covers many of the techniques we discuss here, such as register usage conventions and the advantages of generating code for loops based on their `do-while` form.

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 [102] as well as by members of the team at MIT who helped stop its spread [40]. Since then a number of papers and projects have generated ways both to create and to prevent buffer overflow attacks. Seacord's book [94] provides a wealth of information about buffer overflow and other attacks on code generated by C compilers.

Homework Problems

3.54 ◆

A function with prototype

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

is compiled into IA32 assembly code. The body of the code is as follows:

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
 1      movl    12(%ebp), %edx
 2      subl    16(%ebp), %edx
 3      movl    %edx, %eax
 4      sall    $31, %eax
 5      sarl    $31, %eax
 6      imull   8(%ebp), %edx
 7      xorl    %edx, %eax
```

Parameters x, y, and z are stored at memory locations with offsets 8, 12, and 16 relative to the address in register %ebp. The code stores the return value in register %eax.

Write C code for decode2 that will have an effect equivalent to our assembly code.

3.55 ◆

The following code computes the product of x and y and stores the result in memory. Data type ll_t is defined to be equivalent to long long.

```
typedef long long ll_t;

void store_prod(ll_t *dest, int x, ll_t y) {
    *dest = x*y;
}
```

GCC generates the following assembly code implementing the computation:

```
    dest at %ebp+8, x at %ebp+12, y at %ebp+16
 1      movl    16(%ebp), %esi
 2      movl    12(%ebp), %eax
 3      movl    %eax, %edx
 4      sarl    $31, %edx
 5      movl    20(%ebp), %ecx
 6      imull   %eax, %ecx
 7      movl    %edx, %ebx
 8      imull   %esi, %ebx
 9      addl    %ebx, %ecx
10      mull    %esi
11      leal    (%ecx,%edx), %edx
12      movl    8(%ebp), %ecx
13      movl    %eax, (%ecx)
14      movl    %edx, 4(%ecx)
```

This code uses three multiplications to implement the multiprecision arithmetic required to implement 64-bit arithmetic on a 32-bit machine. Describe the algorithm used to compute the product, and annotate the assembly code to show how it realizes your algorithm. **Hint:** See Problem 3.12 and its solution.

3.56 ◆◆

Consider the following assembly code:

```
    x at %ebp+8, n at %ebp+12
1     movl    8(%ebp), %esi
2     movl    12(%ebp), %ebx
3     movl    $-1, %edi
4     movl    $1, %edx
5   .L2:
6     movl    %edx, %eax
7     andl    %esi, %eax
8     xorl    %eax, %edi
9     movl    %ebx, %ecx
10    sall    %cl, %edx
11    testl   %edx, %edx
12    jne     .L2
13    movl    %edi, %eax
```

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

```
1   int loop(int x, int n)
2   {
3       int result = __________;
4       int 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 `%eax`. 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.

3.57 ◆◆

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

```
int cread(int *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 with the command-line option '`-march=i686`', the generated code should use a conditional move instruction rather than one of the jump instructions.

3.58 ◆◆

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 go 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;

int switch3(int *p1, int *p2, mode_t action)
{
    int 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 3.43. The annotations indicate the argument locations, the register values, and the case labels for the different jump destinations. Register `%edx` corresponds to program variable `result` and is initialized to −1.

Fill in the missing parts of the C code. Watch out for cases that fall through.

```
   Arguments: p1 at %ebp+8, p2 at %ebp+12, action at %ebp+16
   Registers: result in %edx (initialized to -1)
   The jump targets:
1   .L17:                       MODE_E
2     movl    $17, %edx
3     jmp     .L19
4   .L13:                       MODE_A
5     movl    8(%ebp), %eax
6     movl    (%eax), %edx
7     movl    12(%ebp), %ecx
8     movl    (%ecx), %eax
9     movl    8(%ebp), %ecx
10    movl    %eax, (%ecx)
11    jmp     .L19
12  .L14:                       MODE_B
13    movl    12(%ebp), %edx
14    movl    (%edx), %eax
15    movl    %eax, %edx
16    movl    8(%ebp), %ecx
17    addl    (%ecx), %edx
18    movl    12(%ebp), %eax
19    movl    %edx, (%eax)
20    jmp     .L19
21  .L15:                       MODE_C
22    movl    12(%ebp), %edx
23    movl    $15, (%edx)
24    movl    8(%ebp), %ecx
25    movl    (%ecx), %edx
26    jmp     .L19
27  .L16:                       MODE_D
28    movl    8(%ebp), %edx
29    movl    (%edx), %eax
30    movl    12(%ebp), %ecx
31    movl    %eax, (%ecx)
32    movl    $17, %edx
33  .L19:                       default
34    movl    %edx, %eax        Set return value
```

Figure 3.43 **Assembly code for Problem 3.58.** This code implements the different branches of a `switch` statement.

3.59 ◆◆

This problem will give you a chance to reverse engineer a `switch` statement from machine code. In the following procedure, the body of the `switch` statement has been removed:

```
1   int switch_prob(int x, int n)
2   {
3       int result = x;
4
5       switch(n) {
6
7           /* Fill in code here */
8       }
9
10      return result;
11  }
```

Figure 3.44 shows the disassembled machine code for the procedure. We can see in lines 4 and 5 that parameters `x` and `n` are loaded into registers `%eax` and `%edx`, respectively.

The jump table resides in a different area of memory. We can see from the indirect jump on line 9 that the jump table begins at address `0x80485d0`. Using the GDB debugger, we can examine the six 4-byte words of memory comprising the jump table with the command `x/6w 0x80485d0`. GDB prints the following:

```
(gdb) x/6w 0x80485d0
0x80485d0: 0x08048438 0x08048448 0x08048438 0x0804843d
0x80485e0: 0x08048442 0x08048445
```

Fill in the body of the switch statement with C code that will have the same behavior as the machine code.

```
1   08048420 <switch_prob>:
2    8048420:  55                      push   %ebp
3    8048421:  89 e5                   mov    %esp,%ebp
4    8048423:  8b 45 08                mov    0x8(%ebp),%eax
5    8048426:  8b 55 0c                mov    0xc(%ebp),%edx
6    8048429:  83 ea 32                sub    $0x32,%edx
7    804842c:  83 fa 05                cmp    $0x5,%edx
8    804842f:  77 17                   ja     8048448 <switch_prob+0x28>
9    8048431:  ff 24 95 d0 85 04 08    jmp    *0x80485d0(,%edx,4)
10   8048438:  c1 e0 02                shl    $0x2,%eax
11   804843b:  eb 0e                   jmp    804844b <switch_prob+0x2b>
12   804843d:  c1 f8 02                sar    $0x2,%eax
13   8048440:  eb 09                   jmp    804844b <switch_prob+0x2b>
14   8048442:  8d 04 40                lea    (%eax,%eax,2),%eax
15   8048445:  0f af c0                imul   %eax,%eax
16   8048448:  83 c0 0a                add    $0xa,%eax
17   804844b:  5d                      pop    %ebp
18   804844c:  c3                      ret
```

Figure 3.44 **Disassembled code for Problem 3.59.**

3.60 ◆◆◆

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

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

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

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

```
    i at %ebp+8, j at %ebp+12, k at %ebp+16, dest at %ebp+20
 1     movl    12(%ebp), %edx
 2     leal    (%edx,%edx,4), %eax
 3     leal    (%edx,%eax,2), %eax
 4     imull   $99, 8(%ebp), %edx
 5     addl    %edx, %eax
 6     addl    16(%ebp), %eax
 7     movl    A(,%eax,4), %edx
 8     movl    20(%ebp), %eax
 9     movl    %edx, (%eax)
10     movl    $1980, %eax
```

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.

3.61 ◆◆

The code generated by the C compiler for `var_prod_ele` (Figure 3.29) cannot fit all of the values it uses in the loop in registers, and so it must retrieve the value of n from memory on each iteration. Write C code for this function that incorporates optimizations similar to those performed by GCC, but such that the compiled code does not spill any loop values into memory.

Recall that the processor only has six registers available to hold temporary data, since registers `%ebp` and `%esp` cannot be used for this purpose. One of these registers must be used to hold the result of the multiply instruction. Hence, you must reduce the number of values in the loop from six (`result`, `Arow`, `Bcol`, `j`, `n`, and `4*n`) to five.

You will need to find a strategy that works for your particular compiler. Keep trying different strategies until you find one that works.

3.62 ◆◆

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

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

When compiled with optimization level `-O2`, GCC generates the following code for the inner loop of the function:

```
1   .L3:
2     movl    (%ebx), %eax
3     movl    (%esi,%ecx,4), %edx
4     movl    %eax, (%esi,%ecx,4)
5     addl    $1, %ecx
6     movl    %edx, (%ebx)
7     addl    $52, %ebx
8     cmpl    %edi, %ecx
9     jl      .L3
```

A. What is the value of M?

B. What registers hold program values i and j?

C. Write a C code version of `transpose` that makes use of the optimizations that occur in this loop. Use the parameter `M` in your code rather than numeric constants.

3.63 ◆◆

Consider the following source code, where `E1` and `E2` 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.

```
1   int sum_col(int n, int A[E1(n)][E2(n)], int j) {
2       int i;
3       int result = 0;
4       for (i = 0; i < E1(n); i++)
5           result += A[i][j];
6       return result;
7   }
```

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

```
   n at %ebp+8, A at %ebp+12, j at %ebp+16
1     movl    8(%ebp), %eax
2     leal    (%eax,%eax), %edx
```

```
3     leal    (%edx,%eax), %ecx
4     movl    %edx, %ebx
5     leal    1(%edx), %eax
6     movl    $0, %edx
7     testl   %eax, %eax
8     jle     .L3
9     leal    0(,%ecx,4), %esi
10    movl    16(%ebp), %edx
11    movl    12(%ebp), %ecx
12    leal    (%ecx,%edx,4), %eax
13    movl    $0, %edx
14    movl    $1, %ecx
15    addl    $2, %ebx
16  .L4:
17    addl    (%eax), %edx
18    addl    $1, %ecx
19    addl    %esi, %eax
20    cmpl    %ebx, %ecx
21    jne     .L4
22  .L3:
23    movl    %edx, %eax
```

Use your reverse engineering skills to determine the definitions of `E1` and `E2`.

3.64 ◆◆

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 `word_sum` having structures as argument and return values, and a function `prod` that calls `word_sum`:

```
typedef struct {
    int a;
    int *p;
} str1;

typedef struct {
    int sum;
    int diff;
} str2;

str2 word_sum(str1 s1) {
    str2 result;
    result.sum  = s1.a + *s1.p;
    result.diff = s1.a - *s1.p;
```

```
    return result;
}

int prod(int x, int y)
{
    str1 s1;
    str2 s2;
    s1.a = x;
    s1.p = &y;
    s2 = word_sum(s1);
    return s2.sum * s2.diff;
}
```

GCC generates the following code for these two functions:

```
1   word_sum:
2     pushl   %ebp
3     movl    %esp, %ebp
4     pushl   %ebx
5     movl    8(%ebp), %eax
6     movl    12(%ebp), %ebx
7     movl    16(%ebp), %edx
8     movl    (%edx), %edx
9     movl    %ebx, %ecx
10    subl    %edx, %ecx
11    movl    %ecx, 4(%eax)
12    addl    %ebx, %edx
13    movl    %edx, (%eax)
14    popl    %ebx
15    popl    %ebp
16    ret     $4
```

```
1   prod:
2     pushl   %ebp
3     movl    %esp, %ebp
4     subl    $20, %esp
5     leal    12(%ebp), %edx
6     leal    -8(%ebp), %ecx
7     movl    8(%ebp), %eax
8     movl    %eax, 4(%esp)
9     movl    %edx, 8(%esp)
10    movl    %ecx, (%esp)
11    call    word_sum
12    subl    $4, %esp
13    movl    -4(%ebp), %eax
14    imull   -8(%ebp), %eax
15    leave
16    ret
```

The instruction `ret $4` is like a normal return instruction, but it increments the stack pointer by 8 (4 for the return address plus 4 additional), rather than 4.

A. We can see in lines 5–7 of the code for `word_sum` that it appears as if three values are being retrieved from the stack, even though the function has only a single argument. Describe what these three values are.

B. We can see in line 4 of the code for `prod` that 20 bytes are allocated in the stack frame. These get used as five fields of 4 bytes each. Describe how each of these fields gets used.

C. How would you describe the general strategy for passing structures as arguments to a function?

D. How would you describe the general strategy for handling a structure as a return value from a function?

3.65 ◆◆◆

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

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

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

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

GCC generates the following code for the body of `setVal`:

```
1     movl    12(%ebp), %eax
2     movl    36(%eax), %edx
3     addl    12(%eax), %edx
4     movl    8(%ebp), %eax
5     movl    %edx, 92(%eax)
```

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

3.66 ◆◆◆

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

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

```
00000000 <test>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 45 08                mov    0x8(%ebp),%eax
   6:   8b 4d 0c                mov    0xc(%ebp),%ecx
   9:   8d 04 80                lea    (%eax,%eax,4),%eax
   c:   03 44 81 04             add    0x4(%ecx,%eax,4),%eax
  10:   8b 91 b8 00 00 00       mov    0xb8(%ecx),%edx
  16:   03 11                   add    (%ecx),%edx
  18:   89 54 81 08             mov    %edx,0x8(%ecx,%eax,4)
  1c:   5d                      pop    %ebp
  1d:   c3                      ret
```

Figure 3.45 **Disassembled code for Problem 3.66.**

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 disassembly shown in Figure 3.45.

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`.

3.67 ◆◆◆

Consider the following union declaration:

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

This declaration illustrates that structures can be embedded within unions.

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

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

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

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

B. How many total bytes would the structure require?

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

```
   up at %ebp+8
1     movl    8(%ebp), %edx
2     movl    4(%edx), %ecx
3     movl    (%ecx), %eax
4     movl    (%eax), %eax
5     subl    (%edx), %eax
6     movl    %eax, 4(%ecx)
```

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.

3.68 ◆

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 [48, 58].

3.69 ◆

The following declaration defines a class of structures for use in constructing binary trees:

```
1   typedef struct ELE *tree_ptr;
2
3   struct ELE {
4       long val;
5       tree_ptr left;
6       tree_ptr right;
7   };
```

For a function with the following prototype:

```
long trace(tree_ptr tp);
```

GCC generates the following x86-64 code:

```
1   trace:
    tp in %rdi
2     movl    $0, %eax
3     testq   %rdi, %rdi
4     je      .L3
5   .L5:
6     movq    (%rdi), %rax
7     movq    16(%rdi), %rdi
8     testq   %rdi, %rdi
9     jne     .L5
10  .L3:
11    rep
12    ret
```

A. Generate a C version of the function, using a `while` loop.

B. Explain in English what this function computes.

3.70 ◆◆

Using the same tree data structure we saw in Problem 3.69, and a function with the prototype

```
long traverse(tree_ptr tp);
```

GCC generates the following x86-64 code:

```
1   traverse:
    tp in %rdi
2     movq    %rbx, -24(%rsp)
3     movq    %rbp, -16(%rsp)
4     movq    %r12, -8(%rsp)
5     subq    $24, %rsp
6     movq    %rdi, %rbp
7     movabsq $-9223372036854775808, %rax
8     testq   %rdi, %rdi
9     je      .L9
10    movq    (%rdi), %rbx
11    movq    8(%rdi), %rdi
12    call    traverse
13    movq    %rax, %r12
14    movq    16(%rbp), %rdi
15    call    traverse
```

```
16      cmpq    %rax, %r12
17      cmovge  %r12, %rax
18      cmpq    %rbx, %rax
19      cmovl   %rbx, %rax
20    .L9:
21      movq    (%rsp), %rbx
22      movq    8(%rsp), %rbp
23      movq    16(%rsp), %r12
24      addq    $24, %rsp
25      ret
```

A. Generate a C version of the function.

B. Explain in English what this function computes.

Solutions to Practice Problems

Solution to Problem 3.1 (page 170)

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100
4(%eax)	0xAB	Address 0x104
9(%eax,%edx)	0x11	Address 0x10C
260(%ecx,%edx)	0x13	Address 0x108
0xFC(,%ecx,4)	0xFF	Address 0x100
(%eax,%edx,4)	0x11	Address 0x10C

Solution to Problem 3.2 (page 174)

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 two forms is an important skill to learn. One important feature is that memory references in IA32 are always given with double-word registers, such as %eax, even if the operand is a byte or single word.

Here is the code written with suffixes:

```
1       movl    %eax, (%esp)
2       movw    (%eax), %dx
3       movb    $0xFF, %bl
4       movb    (%esp,%edx,4), %dh
5       pushl   $0xFF
6       movw    %dx, (%eax)
7       popl    %edi
```

Solution to Problem 3.3 (page 174)

Since we will rely on GCC to generate most of our assembly code, being able to write correct assembly code is not a critical skill. Nonetheless, this exercise will help you become more familiar with the different instruction and operand types.

Here is the code with explanations of the errors:

```
1     movb $0xF, (%bl)        Cannot use %bl as address register
2     movl %ax, (%esp)        Mismatch between instruction suffix and register ID
3     movw (%eax),4(%esp)     Cannot have both source and destination be memory references
4     movb %ah,%sh            No register named %sh
5     movl %eax,$0x123        Cannot have immediate as destination
6     movl %eax,%dx           Destination operand incorrect size
7     movb %si, 8(%ebp)       Mismatch between instruction suffix and register ID
```

Solution to Problem 3.4 (page 176)

This exercise gives you more experience with the different data movement instructions and how they relate to the data types and conversion rules of C.

src_t	dest_t	Instruction
int	int	movl %eax, (%edx)
char	int	movsbl %al, (%edx)
char	unsigned	movsbl %al, (%edx)
unsigned char	int	movzbl %al, (%edx)
int	char	movb %al, (%edx)
unsigned	unsigned char	movb %al, (%edx)
unsigned	int	movl %eax, (%edx)

Solution to Problem 3.5 (page 176)

Reverse engineering is a good way to understand systems. In this case, we want to reverse the effect of the C compiler to determine what C code gave rise to this assembly code. The best way is to run a "simulation," starting with values x, y, and z at the locations designated by pointers xp, yp, and zp, respectively. We would then get the following behavior:

```
   xp at %ebp+8, yp at %ebp+12, zp at %ebp+16
1     movl    8(%ebp), %edi     Get xp
2     movl    12(%ebp), %edx    Get yp
3     movl    16(%ebp), %ecx    Get zp
4     movl    (%edx), %ebx      Get y
5     movl    (%ecx), %esi      Get z
6     movl    (%edi), %eax      Get x
7     movl    %eax, (%edx)      Store x at yp
8     movl    %ebx, (%ecx)      Store y at zp
9     movl    %esi, (%edi)      Store z at xp
```

From this, we can generate the following C code:

```
void decode1(int *xp, int *yp, int *zp)
{
    int tx = *xp;
    int ty = *yp;
    int tz = *zp;

    *yp = tx;
    *zp = ty;
    *xp = tz;
}
```

Solution to Problem 3.6 (page 178)

This exercise demonstrates the versatility of the `leal` instruction and gives you more practice in deciphering the different operand forms. Although the operand forms are classified as type "Memory" in Figure 3.3, no memory access occurs.

Instruction	Result
`leal 6(%eax), %edx`	$6 + x$
`leal (%eax,%ecx), %edx`	$x + y$
`leal (%eax,%ecx,4), %edx`	$x + 4y$
`leal 7(%eax,%eax,8), %edx`	$7 + 9x$
`leal 0xA(,%ecx,4), %edx`	$10 + 4y$
`leal 9(%eax,%ecx,2), %edx`	$9 + x + 2y$

Solution to Problem 3.7 (page 179)

This problem gives you a chance to test your understanding of operands and the arithmetic instructions. The instruction sequence is designed so that the result of each instruction does not affect the behavior of subsequent ones.

Instruction	Destination	Value
`addl %ecx,(%eax)`	`0x100`	`0x100`
`subl %edx,4(%eax)`	`0x104`	`0xA8`
`imull $16,(%eax,%edx,4)`	`0x10C`	`0x110`
`incl 8(%eax)`	`0x108`	`0x14`
`decl %ecx`	`%ecx`	`0x0`
`subl %edx,%eax`	`%eax`	`0xFD`

Solution to Problem 3.8 (page 180)

This exercise gives you a chance to generate a little bit of assembly code. The solution code was generated by GCC. By loading parameter `n` in register `%ecx`, it can then use byte register `%cl` to specify the shift amount for the `sarl` instruction:

```
1      movl    8(%ebp), %eax     Get x
2      sall    $2, %eax          x <<= 2
3      movl    12(%ebp), %ecx    Get n
4      sarl    %cl, %eax         x >>= n
```

Solution to Problem 3.9 (page 181)

This problem is fairly straightforward, since each of the expressions is implemented by a single instruction and there is no reordering of the expressions.

```
5        int t1 = x^y;
6        int t2 = t1 >> 3;
7        int t3 = ~t2;
8        int t4 = t3-z;
```

Solution to Problem 3.10 (page 182)

A. This instruction is used to set register `%edx` to zero, exploiting the property that $x \hat{} x = 0$ for any x. It corresponds to the C statement `x = 0`.

B. A more direct way of setting register `%edx` to zero is with the instruction `movl $0,%edx`.

C. Assembling and disassembling this code, however, we find that the version with `xorl` requires only 2 bytes, while the version with `movl` requires 5.

Solution to Problem 3.11 (page 184)

We can simply replace the `cltd` instruction with one that sets register `%edx` to 0, and use `divl` rather than `idivl` as our division instruction, yielding the following code:

```
x at %ebp+8, y at %ebp+12
 movl    8(%ebp),%eax     Load x into %eax
 movl    $0,%edx          Set high-order bits to 0
 divl    12(%ebp)         Unsigned divide by y
 movl    %eax, 4(%esp)    Store x / y
 movl    %edx, (%esp)     Store x % y
```

Solution to Problem 3.12 (page 184)

A. We can see that the program is performing multiprecision operations on 64-bit data. We can also see that the 64-bit multiply operation (line 4) uses unsigned arithmetic, and so we conclude that `num_t` is `unsigned long long`.

B. Let x denote the value of variable `x`, and let y denote the value of `y`, which we can write as $y = y_h \cdot 2^{32} + y_l$, where y_h and y_l are the values represented by the high- and low-order 32 bits, respectively. We can therefore compute $x \cdot y = x \cdot y_h \cdot 2^{32} + x \cdot y_l$. The full representation of the product would be 96 bits long, but we require only the low-order 64 bits. We can therefore let s be the low-order 32 bits of $x \cdot y_h$ and t be the full 64-bit product $x \cdot y_l$, which

we can split into high- and low-order parts t_h and t_l. The final result has t_l as the low-order part, and $s + t_h$ as the high-order part.

Here is the annotated assembly code:

```
   dest at %ebp+8, x at %ebp+12, y at %ebp+16
1     movl    12(%ebp), %eax              Get x
2     movl    20(%ebp), %ecx              Get y_h
3     imull   %eax, %ecx                  Compute s = x*y_h
4     mull    16(%ebp)                    Compute t = x*y_l
5     leal    (%ecx,%edx), %edx           Add s to t_h
6     movl    8(%ebp), %ecx               Get dest
7     movl    %eax, (%ecx)                Store t_l
8     movl    %edx, 4(%ecx)               Store s+t_h
```

Solution to Problem 3.13 (page 188)

It is important to understand that assembly code does not keep track of the type of a program value. Instead, the different instructions determine the operand sizes and whether they are signed or unsigned. When mapping from instruction sequences back to C code, we must do a bit of detective work to infer the data types of the program values.

A. The suffix '`l`' and the register identifiers indicate 32-bit operands, while the comparison is for a two's complement '`<`'. We can infer that `data_t` must be `int`.

B. The suffix '`w`' and the register identifiers indicate 16-bit operands, while the comparison is for a two's-complement '`>=`'. We can infer that `data_t` must be `short`.

C. The suffix '`b`' and the register identifiers indicate 8-bit operands, while the comparison is for an unsigned '`<`'. We can infer that `data_t` must be `unsigned char`.

D. The suffix '`l`' and the register identifiers indicate 32-bit operands, while the comparison is for '`!=`', which is the same whether the arguments are signed, unsigned, or pointers. We can infer that `data_t` could be either `int`, `unsigned`, or some form of pointer. For the first two cases, they could also have the `long` size designator.

Solution to Problem 3.14 (page 189)

This problem is similar to Problem 3.13, except that it involves TEST instructions rather than CMP instructions.

A. The suffix '`l`' and the register identifiers indicate 32-bit operands, while the comparison is for '`!=`', which is the same for signed or unsigned. We can infer that `data_t` must be either `int`, `unsigned`, or some type of pointer. For the first two cases, they could also have the `long` size designator.

B. The suffix '`w`' and the register identifier indicate 16-bit operands, while the comparison is for '`==`', which is the same for signed or unsigned. We can infer that `data_t` must be either `short` or `unsigned short`.

C. The suffix 'b' and the register identifier indicate an 8-bit operand, while the comparison is for two's complement '>'. We can infer that `data_t` must be `char`.

D. The suffix 'w' and the register identifier indicate 16-bit operands, while the comparison is for unsigned '>'. We can infer that `data_t` must be `unsigned short`.

Solution to Problem 3.15 (page 192)

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `je` instruction has as target `0x8048291 + 0x05`. As the original disassembled code shows, this is `0x8048296`:

```
804828f:        74 05                   je     8048296
8048291:        e8 1e 00 00 00          call   80482b4
```

B. The `jb` instruction has as target `0x8048359 − 25` (since `0xe7` is the 1-byte, two's-complement representation of −25). As the original disassembled code shows, this is `0x8048340`:

```
8048357:        72 e7                   jb     8048340
8048359:        c6 05 10 a0 04 08 01    movb   $0x1,0x804a010
```

C. According to the annotation produced by the disassembler, the jump target is at absolute address `0x8048391`. According to the byte encoding, this must be at an address `0x12` bytes beyond that of the `mov` instruction. Subtracting these gives address `0x804837f`, as confirmed by the disassembled code:

```
804837d:        74 12                   je     8048391
804837f:        b8 00 00 00 00          mov    $0x0,%eax
```

D. Reading the bytes in reverse order, we see that the target offset is `0xfffffffe0`, or decimal −32. Adding this to `0x80482c4` (the address of the `nop` instruction) gives address `0x80482a4`:

```
80482bf:        e9 e0 ff ff ff          jmp    80482a4
80482c4:        90                      nop
```

E. An indirect jump is denoted by instruction code `ff 25`. The address from which the jump target is to be read is encoded explicitly by the following 4 bytes. Since the machine is little endian, these are given in reverse order as `fc 9f 04 08`.

Solution to Problem 3.16 (page 195)

Annotating assembly code and writing C code that mimics its control flow are good first steps in understanding assembly-language programs. This problem gives you practice for an example with simple control flow. It also gives you a chance to examine the implementation of logical operations.

A. Here is the C code:

```
1   void goto_cond(int a, int *p) {
2       if (p == 0)
3           goto done;
4       if (a <= 0)
5           goto done;
6       *p += a;
7    done:
8       return;
9   }
```

B. The first conditional branch is part of the implementation of the `&&` expression. If the test for `p` being non-null fails, the code will skip the test of `a > 0`.

Solution to Problem 3.17 (page 196)

This is an exercise to help you think about the idea of a general translation rule and how to apply it.

A. Converting to this alternate form involves only switching around a few lines of the code:

```
1   int gotodiff_alt(int x, int y) {
2       int result;
3       if (x < y)
4           goto true;
5       result = x - y;
6       goto done;
7    true:
8       result =  y - x;
9    done:
10      return result;
11  }
```

B. In most respects, the choice is arbitrary. But the original rule works better for the common case where there is no `else` statement. For this case, we can simply modify the translation rule to be as follows:

```
    t = test-expr;
    if (!t)
        goto done;
    then-statement
done:
```

A translation based on the alternate rule is more cumbersome.

Solution to Problem 3.18 (page 196)

This problem requires that you work through a nested branch structure, where you will see how our rule for translating `if` statements has been applied. For the

most part, the machine code is a straightforward translation of the C code. The only difference is that the initialization expression (line 2 in the C code) has been moved down (line 15 in the assembly code) so that it only gets computed when it is certain that this will be the returned value.

```
1   int test(int x, int y) {
2       int val = x^y;
3       if (x < -3) {
4           if (y < x)
5               val = x*y;
6           else
7               val = x+y;
8       } else if (x > 2)
9           val = x-y;
10      return val;
11  }
```

Solution to Problem 3.19 (page 198)

A. If we build up a table of factorials computed with data type `int`, we get the following:

n	$n!$	OK?
1	1	Y
2	2	Y
3	6	Y
4	24	Y
5	120	Y
6	720	Y
7	5,040	Y
8	40,320	Y
9	362,880	Y
10	3,628,800	Y
11	39,916,800	Y
12	479,001,600	Y
13	1,932,053,504	Y
14	1,278,945,280	N

We can see that 14! has overflowed, since the numbers stopped growing. As we learned in Problem 2.35, we can also test whether or not the computation of $n!$ has overflowed by computing $n!/n$ and seeing whether it equals $(n-1)!$.

B. Doing the computation with data type `long long` lets us go up to 20!, yielding 2,432,902,008,176,640,000.

Solution to Problem 3.20 (page 199)

The code generated when compiling loops can be tricky to analyze, because the compiler can perform many different optimizations on loop code, and because it can be difficult to match program variables with registers. We start practicing this skill with a fairly simple loop.

A. The register usage can be determined by simply looking at how the arguments get fetched.

Register usage

Register	Variable	Initially
%eax	x	x
%ecx	y	y
%edx	n	n

B. The *body-statement* portion consists of lines 3 through 5 in the C code and lines 5 through 7 in the assembly code. The *test-expr* portion is on line 6 in the C code. In the assembly code, it is implemented by the instructions on lines 8 through 11.

C. The annotated code is as follows:

```
    x at %ebp+8, y at %ebp+12, n at %ebp+16
 1     movl    8(%ebp), %eax    Get x
 2     movl    12(%ebp), %ecx   Get y
 3     movl    16(%ebp), %edx   Get n
 4   .L2:                     loop:
 5     addl    %edx, %eax       x += n
 6     imull   %edx, %ecx       y *= n
 7     subl    $1, %edx         n--
 8     testl   %edx, %edx       Test n
 9     jle     .L5              If <= 0, goto done
10     cmpl    %edx, %ecx       Compare y:n
11     jl      .L2              If <, goto loop
12   .L5:                     done:
```

As with the code of Problem 3.16, two conditional branches are required to implement the && operation.

Solution to Problem 3.21 (page 201)

This problem demonstrates how the transformations made by the compiler can make it difficult to decipher the generated assembly code.

A. We can see that the register is initialized to $a + b$ and then incremented on each iteration. Similarly, the value of a (held in register %ecx) is incremented on each iteration. We can therefore see that the value in register %edx will always equal $a + b$. Let us call this apb (for "a plus b").

B. Here is a table of register usage:

Register	Program value	Initial value
%ecx	a	a
%ebx	b	b
%eax	result	1
%edx	apb	$a + b$

C. The annotated code is as follows:

```
  Arguments: a at %ebp+8, b at %ebp+12
  Registers: a in %ecx, b in %ebx, result in %eax, %edx set to apb (a+b)
1      movl    8(%ebp), %ecx            Get a
2      movl    12(%ebp), %ebx           Get b
3      movl    $1, %eax                 Set result = 1
4      cmpl    %ebx, %ecx               Compare a:b
5      jge     .L11                     If >=, goto done
6      leal    (%ebx,%ecx), %edx        Compute apb = a+b
7      movl    $1, %eax                 Set result = 1
8    .L12:                            loop:
9      imull   %edx, %eax               Compute result *= apb
10     addl    $1, %ecx                 Compute a++
11     addl    $1, %edx                 Compute apb++
12     cmpl    %ecx, %ebx               Compare b:a
13     jg      .L12                     If >, goto loop
14   .L11:                            done:
       Return result
```

D. The equivalent `goto` code is as follows:

```
1   int loop_while_goto(int a, int b)
2   {
3       int result = 1;
4       if (a >= b)
5           goto done;
6       /* apb has same value as a+b in original code */
7       int apb = a+b;
8    loop:
9       result *= apb;
10      a++;
11      apb++;
12      if (b > a)
13          goto loop;
14   done:
15      return result;
16  }
```

Solution to Problem 3.22 (page 202)

Being able to work backward from assembly code to C code is a prime example of reverse engineering.

A. Here is the original C code:

```
int fun_a(unsigned x) {
    int val = 0;
    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0x1;
}
```

B. This code computes the *parity* of argument `x`. That is, it returns 1 if there is an odd number of ones in `x` and 0 if there is an even number.

Solution to Problem 3.23 (page 205)

This problem is trickier than Problem 3.22, since the code within the loop is more complex and the overall operation is less familiar.

A. Here is the original C code:

```
int fun_b(unsigned x) {
    int val = 0;
    int i;
    for (i = 0; i < 32; i++) {
        val = (val << 1) | (x & 0x1);
        x >>= 1;
    }
    return val;
}
```

B. This code reverses the bits in `x`, creating a mirror image. It does this by shifting the bits of `x` from left to right, and then filling these bits in as it shifts `val` from right to left.

Solution to Problem 3.24 (page 206)

Our stated rule for translating a `for` loop into a `while` loop is just a bit too simplistic—this is the only aspect that requires special consideration.

A. Applying our translation rule would yield the following code:

```
/* Naive translation of for loop into while loop */
/* WARNING: This is buggy code */
int sum = 0;
int i = 0;
```

```
while (i < 10) {
    if (i & 1)
        /* This will cause an infinite loop */
        continue;
    sum += i;
    i++;
}
```

This code has an infinite loop, since the `continue` statement would prevent index variable `i` from being updated.

B. The general solution is to replace the `continue` statement with a `goto` statement that skips the rest of the loop body and goes directly to the update portion:

```
/* Correct translation of for loop into while loop */
int sum = 0;
int i = 0;
while (i < 10) {
    if (i & 1)
        goto update;
    sum += i;
update:
    i++;
}
```

Solution to Problem 3.25 (page 209)

This problem reinforces our method of computing the misprediction penalty.

A. We can apply our formula directly to get $T_{MP} = 2(31 - 16) = 30$.

B. When misprediction occurs, the function will require around $16 + 30 = 46$ cycles.

Solution to Problem 3.26 (page 212)

This problem provides a chance to study the use of conditional moves.

A. The operator is '/'. We see this is an example of dividing by a power of 2 by right shifting (see Section 2.3.7). Before shifting by $k = 2$, we must add a bias of $2^k - 1 = 3$ when the dividend is negative.

B. Here is an annotated version of the assembly code:

```
   Computation by function arith
   Register: x in %edx
1     leal    3(%edx), %eax     temp = x+3
2     testl   %edx, %edx        Test x
3     cmovns  %edx, %eax        If >= 0, temp = x
4     sarl    $2, %eax          Return temp >> 2 (= x/4)
```

The program creates a temporary value equal to $x + 3$, in anticipation of x being negative and therefore requiring biasing. The `cmovns` instruction conditionally changes this number to x when $x \geq 0$, and then it is shifted by 2 to generate $x/4$.

Solution to Problem 3.27 (page 212)

This problem is similar to Problem 3.18, except that some of the conditionals have been implemented by conditional data transfers. Although it might seem daunting to fit this code into the framework of the original C code, you will find that it follows the translation rules fairly closely.

```
1   int test(int x, int y) {
2       int val = 4*x;
3       if (y > 0) {
4           if (x < y)
5               val = x-y;
6           else
7               val = x^y;
8       } else if (y < -2)
9           val = x+y;
10      return val;
11  }
```

Solution to Problem 3.28 (page 217)

This problem gives you a chance to reason about the control flow of a `switch` statement. Answering the questions requires you to combine information from several places in the assembly code.

1. Line 2 of the assembly code adds 2 to x to set the lower range of the cases to zero. That means that the minimum case label is -2.
2. Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 6. This implies that the maximum case label is $-2 + 6 = 4$.
3. In the jump table, we see that the entry on line 3 (case value -1) has the same destination (`.L2`) as the jump instruction on line 4, indicating the default case behavior. Thus, case label -1 is missing in the `switch` statement body.
4. In the jump table, we see that the entries on lines 6 and 7 have the same destination. These correspond to case labels 2 and 3.

From this reasoning, we draw the following two conclusions:

A. The case labels in the `switch` statement body had values $-2, 0, 1, 2, 3$, and 4.

B. The case with destination `.L6` had labels 2 and 3.

Solution to Problem 3.29 (page 218)

The key to reverse engineering compiled `switch` statements is to combine the information from the assembly code and the jump table to sort out the different cases. We can see from the `ja` instruction (line 3) that the code for the default case

has label .L2. We can see that the only other repeated label in the jump table is .L4, and so this must be the code for the cases C and D. We can see that the code falls through at line 14, and so label .L6 must match case A and label .L3 must match case B. That leaves only label .L2 to match case E.

The original C code is as follows. Observe how the compiler optimized the case where a equals 4 by setting the return value to be 4, rather than a.

```
1   int switcher(int a, int b, int c)
2   {
3       int answer;
4       switch(a) {
6       case 5:
7           c = b ^ 15;
8           /* Fall through */
9       case 0:
10          answer = c + 112;
11          break;
12      case 2:
13      case 7:
14          answer = (c + b) << 2;
15          break;
16      case 4:
17          answer = a;  /* equivalently, answer = 4 */
18          break;
19      default:
20          answer = b;
21      }
22      return answer;
23  }
```

Solution to Problem 3.30 (page 223)

This is another example of an assembly-code idiom. At first it seems quite peculiar—a call instruction with no matching ret. Then we realize that it is not really a procedure call after all.

A. %eax is set to the address of the popl instruction.

B. This is not a true procedure call, since the control follows the same ordering as the instructions and the return address is popped from the stack.

C. This is the only way in IA32 to get the value of the program counter into an integer register.

Solution to Problem 3.31 (page 224)

This problem makes concrete the discussion of register usage conventions. Registers %edi, %esi, and %ebx are callee-save. The procedure must save them on the stack before altering their values and restore them before returning. The other three registers are caller-save. They can be altered without affecting the behavior of the caller.

Solution to Problem 3.32 (page 228)

One step in learning to read IA32 code is to become very familiar with the way arguments are passed on the stack. The key to solving this problem is to note that the storage of `d` at `p` is implemented by the instruction at line 3 of the assembly code, from which you work backward to determine the types and positions of arguments `d` and `p`. Similarly, the subtraction is performed at line 6, and from this you can work backward to determine the types and positions of arguments `x` and `c`.

The following is the function prototype:

```
int fun(short c, char d, int *p, int x);
```

As this example shows, reverse engineering is like solving a puzzle. It's important to identify the points where there is a unique choice, and then work around these points to fill in the rest of the details.

Solution to Problem 3.33 (page 228)

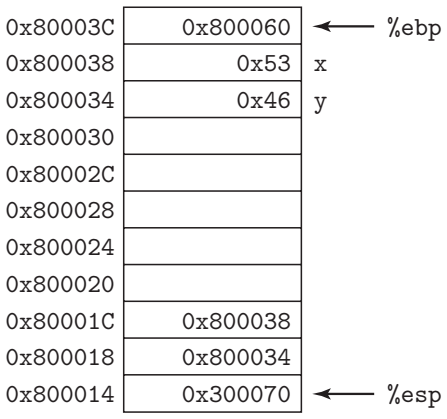
Being able to reason about how functions use the stack is a critical part of understanding compiler-generated code. As this example illustrates, the compiler may allocate a significant amount of space that never gets used.

A. We started with `%esp` having value `0x800040`. The `pushl` instruction on line 2 decrements the stack pointer by 4, giving `0x80003C`, and this becomes the new value of `%ebp`.

B. Line 4 decrements the stack pointer by 40 (hex `0x28`), yielding `0x800014`.

C. We can see how the two `leal` instructions (lines 5 and 7) compute the arguments to pass to `scanf`, while the two `movl` instructions (lines 6 and 8) store them on the stack. Since the function arguments appear on the stack at increasingly positive offsets from `%esp`, we can conclude that line 5 computes `&x`, while line 7 computes line `&y`. These have values `0x800038` and `0x800034`, respectively.

D. The stack frame has the following structure and contents:



E. Byte addresses `0x800020` through `0x800033` are unused.

Solution to Problem 3.34 (page 231)

This problem provides a chance to examine the code for a recursive function. An important lesson to learn is that recursive code has the exact same structure as the other functions we have seen. The stack and register-saving disciplines suffice to make recursive functions operate correctly.

A. Register `%ebx` holds the value of parameter `x`, so that it can be used to compute the result expression.

B. The assembly code was generated from the following C code:

```
int rfun(unsigned x) {
    if (x == 0)
        return 0;
    unsigned nx = x>>1;
    int rv = rfun(nx);
    return (x & 0x1) + rv;
}
```

C. Like the code of Problem 3.49, this function computes the sum of the bits in argument `x`. It recursively computes the sum of all but the least significant bit, and then it adds the least significant bit to get the result.

Solution to Problem 3.35 (page 233)

This exercise tests your understanding of data sizes and array indexing. Observe that a pointer of any kind is 4 bytes long. For IA32, GCC allocates 12 bytes for data type `long double`, even though the actual format requires only 10 bytes.

Array	Element size	Total size	Start address	Element i
`S`	2	14	x_S	$x_S + 2i$
`T`	4	12	x_T	$x_T + 4i$
`U`	4	24	x_U	$x_U + 4i$
`V`	12	96	x_V	$x_V + 12i$
`W`	4	16	x_W	$x_W + 4i$

Solution to Problem 3.36 (page 234)

This problem is a variant of the one shown for integer array `E`. It is important to understand the difference between a pointer and the object being pointed to. Since data type `short` requires 2 bytes, all of the array indices are scaled by a factor of 2. Rather than using `movl`, as before, we now use `movw`.

Expression	Type	Value	Assembly
`S+1`	`short *`	$x_S + 2$	`leal 2(%edx),%eax`
`S[3]`	`short`	M[$x_S + 6$]	`movw 6(%edx),%ax`
`&S[i]`	`short *`	$x_S + 2i$	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	M[$x_S + 8i + 2$]	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short *`	$x_S + 2i - 10$	`leal -10(%edx,%ecx,2),%eax`

Solution to Problem 3.37 (page 236)

This problem requires you to work through the scaling operations to determine the address computations, and to apply Equation 3.1 for row-major indexing. The first step is to annotate the assembly code to determine how the address references are computed:

```
1     movl    8(%ebp), %ecx           Get i
2     movl    12(%ebp), %edx          Get j
3     leal    0(,%ecx,8), %eax        8*i
4     subl    %ecx, %eax              8*i-i = 7*i
5     addl    %edx, %eax              7*i+j
6     leal    (%edx,%edx,4), %edx     5*j
7     addl    %ecx, %edx              5*j+i
8     movl    mat1(,%eax,4), %eax     mat1[7*i+j]
9     addl    mat2(,%edx,4), %eax     mat2[5*j+i]
```

We can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this, we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $M = 5$ and $N = 7$.

Solution to Problem 3.38 (page 238)

This exercise requires that you be able to study compiler-generated assembly code to understand what optimizations have been performed. In this case, the compiler was clever in its optimizations.

Let us first study the following C code, and then see how it is derived from the assembly code generated for the original function.

```
1  /* Set all diagonal elements to val */
2  void fix_set_diag_opt(fix_matrix A, int val) {
3    int *Abase = &A[0][0];
4    int index = 0;
5    do {
6      Abase[index] = val;
7      index += (N+1);
8    } while (index != (N+1)*N);
9  }
```

This function introduces a variable `Abase`, of type `int *`, pointing to the start of array `A`. This pointer designates a sequence of 4-byte integers consisting of elements of `A` in row-major order. We introduce an integer variable `index` that steps through the diagonal elements of `A`, with the property that diagonal elements i and $i + 1$ are spaced $N + 1$ elements apart in the sequence, and that once we reach diagonal element N (index value $N(N + 1)$), we have gone beyond the end.

The actual assembly code follows this general form, but now the pointer increments must be scaled by a factor of 4. We label register `%eax` as holding a

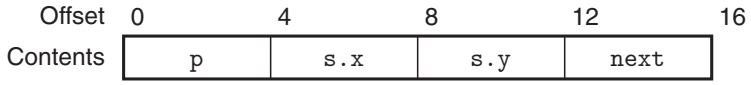
value index4 equal to index in our C version, but scaled by a factor of 4. For $N = 16$, we can see that our stopping point for index4 will be $4 \cdot 16(16 + 1) = 1088$.

```
   A at %ebp+8, val at %ebp+12
1     movl    8(%ebp), %ecx           Get Abase = &A[0][0]
2     movl    12(%ebp), %edx          Get val
3     movl    $0, %eax                Set index4 to 0
4   .L14:                           loop:
5     movl    %edx, (%ecx,%eax)       Set Abase[index4/4] to val
6     addl    $68, %eax               index4 += 4(N+1)
7     cmpl    $1088, %eax             Compare index4:4N(N+1)
8     jne     .L14                    If !=, goto loop
```

Solution to Problem 3.39 (page 243)

This problem gets you to think about structure layout and the code used to access structure fields. The structure declaration is a variant of the example shown in the text. It shows that nested structures are allocated by embedding the inner structures within the outer ones.

A. The layout of the structure is as follows:



B. It uses 16 bytes.

C. As always, we start by annotating the assembly code:

```
   sp at %ebp+8
1     movl    8(%ebp), %eax    Get sp
2     movl    8(%eax), %edx    Get sp->s.y
3     movl    %edx, 4(%eax)    Store in sp->s.x
4     leal    4(%eax), %edx    Compute &(sp->s.x)
5     movl    %edx, (%eax)     Store in sp->p
6     movl    %eax, 12(%eax)   Store sp in sp->next
```

From this, we can generate C code as follows:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = sp->s.y;
    sp->p     = &(sp->s.x);
    sp->next  = sp;
}
```

Solution to Problem 3.40 (page 247)

Structures and unions involve a simple set of concepts, but it takes practice to be comfortable with the different referencing patterns and their implementations.

EXPR	TYPE	Code
`up->t1.s`	`int`	`movl 4(%eax), %eax` `movl %eax, (%edx)`
`up->t1.v`	`short`	`movw (%eax), %ax` `movw %ax, (%edx)`
`&up->t1.d`	`short *`	`leal 2(%eax), %eax` `movl %eax, (%edx)`
`up->t2.a`	`int *`	`movl %eax, (%edx)`

`up->t2.a[up->t1.s]`	`int`	`movl 4(%eax), %ecx`
		`movl (%eax,%ecx,4), %eax`
		`movl %eax, (%edx)`

`*up->t2.p`	`char`	`movl 8(%eax), %eax`
		`movb (%eax), %al`
		`movb %al, (%edx)`

Solution to Problem 3.41 (page 251)

Understanding structure layout and alignment is very important for understanding how much storage different data structures require and for understanding the code generated by the compiler for accessing structures. This problem lets you work out the details of some example structures.

A. `struct P1 { int i; char c; int j; char d; };`

i	c	j	d	Total	Alignment
0	4	8	12	16	4

B. `struct P2 { int i; char c; char d; int j; };`

i	c	j	d	Total	Alignment
0	4	5	8	12	4

C. `struct P3 { short w[3]; char c[3] };`

w	c	Total	Alignment
0	6	10	2

D. `struct P4 { short w[3]; char *c[3] };`

w	c	Total	Alignment
0	8	20	4

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

a	p	Total	Alignment
0	32	36	4

Solution to Problem 3.42 (page 251)

This is an exercise in understanding structure layout and alignment.

A. Here are the object sizes and byte offsets:

Field	a	b	c	d	e	f	g	h
Size	4	2	8	1	4	1	8	4
Offset	0	4	8	16	20	24	32	40

B. The structure is a total of 48 bytes long. The end of the structure must be padded by 4 bytes to satisfy the 8-byte alignment requirement.

C. One strategy that works, when all data elements have a length equal to a power of two, is to order the structure elements in descending order of size. This leads to a declaration,

```
struct {
    double    c;
    long long g;
    float     e;
    char     *a;
    void     *h;
    short     b;
    char      d;
    char      f;
} foo;
```

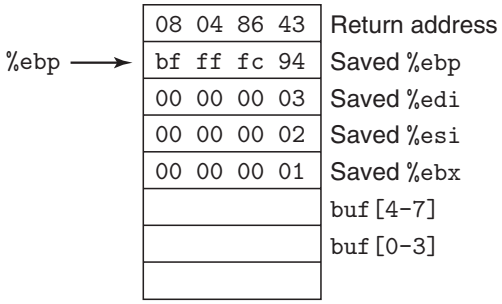
with the following offsets, for a total of 32 bytes:

Field	c	g	e	a	h	b	d	f
Size	8	8	4	4	4	2	1	1
Offset	0	8	16	20	24	28	30	31

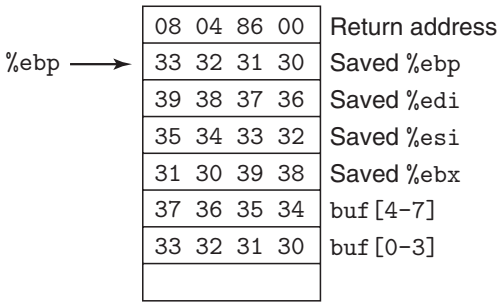
Solution to Problem 3.43 (page 259)

This problem covers a wide range of topics, such as stack frames, string representations, ASCII code, and byte ordering. It demonstrates the dangers of out-of-bounds memory references and the basic ideas behind buffer overflow.

A. Stack after line 7:



B. Stack after line 10:



C. The program is attempting to return to address `0x08048600`. The low-order byte was overwritten by the terminating null character.

D. The saved values of the following registers were altered:

Register	Value
%ebp	33323130
%edi	39383736
%esi	35343332
%ebx	31303938

These values will be loaded into the registers before `getline` returns.

E. The call to `malloc` should have had `strlen(buf)+1` as its argument, and the code should also check that the returned value is not equal to `NULL`.

Solution to Problem 3.44 (page 262)

A. This corresponds to a range of around 2^{13} addresses.

B. A 128-byte nop sled would cover 2^7 addresses with each test, and so we would only require $2^6 = 64$ attempts.

This example clearly shows that the degree of randomization in this version of Linux would provide only minimal deterrence against an overflow attack.

Solution to Problem 3.45 (page 264)

This problem gives you another chance to see how IA32 code manages the stack, and to also better understand how to defend against buffer overflow attacks.

A. For the unprotected code, we can see that lines 4 and 6 compute the positions of `v` and `buf` to be at offsets −8 and −20 relative to `%ebp`. In the protected code, the canary is stored at offset −8 (line 3), while `v` and `buf` are at offsets −24 and −20 (lines 7 and 9).

B. In the protected code, local variable `v` is positioned closer to the top of the stack than `buf`, and so an overrun of `buf` will not corrupt the value of `v`.

In fact, `buf` is positioned so that any buffer overrun will corrupt the canary value.

Solution to Problem 3.46 (page 271)

Achieving a factor of 51 price improvement every 10 years over 3 decades has been truly remarkable, and it helps explain why computers have become so pervasive in our society.

A. Assuming the baseline of 16.3 gigabytes in 2010, 256 terabytes represents an increase by a factor of 1.608×10^4, which would take around 25 years, giving us 2035.

B. Sixteen exabytes is an increase of 1.054×10^9 over 16.3 gigabytes. This would take around 53 years, giving us 2063.

C. Increasing the budget by a factor of 10 cuts about 6 years off our schedule, making it possible to meet the two memory-size goals in years 2029 and 2057, respectively.

These numbers, of course, should not be taken too literally. It would require scaling memory technology well beyond what are believed to be fundamental physical limits of the current technology. Nonetheless, it indicates that, within the lifetimes of many readers of this book, there will be systems with exabyte-scale memory systems.

Solution to Problem 3.47 (page 276)

This problem illustrates some of the subtleties of type conversion and the different move instructions. In some cases, we make use of the property that the `movl` instruction will set the upper 32 bits of the destination register to zeros. Some of the problems have multiple solutions.

`src_t`	`dest_t`	Instruction	*S*	*D*	Explanation
`long`	`long`	`movq`	`%rdi`	`%rax`	No conversion
`int`	`long`	`movslq`	`%edi`	`%rax`	Sign extend
`char`	`long`	`movsbq`	`%dil`	`%rax`	Sign extend
`unsigned int`	`unsigned long`	`movl`	`%edi`	`%eax`	Zero extend to 64 bits
`unsigned char`	`unsigned long`	`movzbq`	`%dil`	`%rax`	Zero extend to 64
`unsigned char`	`unsigned long`	`movzbl`	`%dil`	`%eax`	Zero extend to 64 bits
`long`	`int`	`movslq`	`%edi`	`%rax`	Sign extend to 64 bits
`long`	`int`	`movl`	`%edi`	`%eax`	Zero extend to 64 bits
`unsigned long`	`unsigned`	`movl`	`%edi`	`%eax`	Zero extend to 64 bits

We show that the `long` to `int` conversion can use either `movslq` or `movl`, even though one will sign extend the upper 32 bits, while the other will zero extend it. This is because the values of the upper 32 bits are ignored for any subsequent instruction having `%eax` as an operand.

Solution to Problem 3.48 (page 278)

We can step through the code for `arithprob` and determine the following:

1. The first `movslq` instruction sign extends `d` to a long integer prior to its multiplication by `c`. This implies that `d` has type `int` and `c` has type `long`.
2. The `movsbl` instruction (line 4) sign extends `b` to an integer prior to its multiplication by `a`. This means that `b` has type `char` and `a` has type `int`.
3. The sum is computed using a `leaq` instruction, indicating that the return value has type `long`.

From this, we can determine that the unique prototype for `arithprob` is

```
long arithprob(int a, char b, long c, int d);
```

Solution to Problem 3.49 (page 281)

This problem demonstrates a clever way to count the number of 1 bits in a word. It uses several tricks that look fairly obscure at the assembly-code level.

A. Here is the original C code:

```
long fun_c(unsigned long x) {
    long val = 0;
    int i;
    for (i = 0; i < 8; i++) {
        val += x & 0x0101010101010101L;
        x >>= 1;
    }
    val += (val >> 32);
    val += (val >> 16);
    val += (val >> 8);
    return val & 0xFF;
}
```

B. This code sums the bits in `x` by computing 8 single-byte sums in parallel, using all 8 bytes of `val`. It then sums the two halves of `val`, then the two low-order 16 bits, and then the 2 low-order bytes of this sum to get the final amount in the low-order byte. It masks off the high-order bits to get the final result. This approach has the advantage that it requires only 8 iterations, rather than the more typical 64.

Solution to Problem 3.50 (page 284)

We can step through the code for `incrprob` and determine the following:

1. The `addl` instruction fetches a 32-bit integer from the location given by the third argument register and adds it to the 32-bit version of the first argument register. From this, we can infer that `t` is the third argument and `x` is the first argument. We can see that `t` must be a pointer to a signed or unsigned integer, but `x` could be either signed or unsigned, and it could either be 32 bits or 64 (since when adding it to `*t`, the code should truncate it to 32 bits).

2. The `movslq` instruction sign extends the sum (a copy of `*t`) to a long integer. From this, we can infer that `t` must be a pointer to a signed integer.
3. The `addq` instruction adds the sign-extended value of the previous sum to the location indicated by the second argument register. From this, we can infer that `q` is the second argument and that it is a pointer to a long integer.

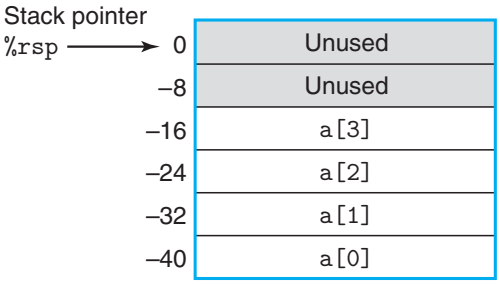
There are four valid prototypes for `incrprob`, depending on whether or not `x` is long, and whether it is signed or unsigned. We show these as four different prototypes:

```
void incrprob_s(int x, long *q, int *t);
void incrprob_u(unsigned x, long *q, int *t);
void incrprob_sl(long x, long *q, int *t);
void incrprob_ul(unsigned long x, long *q, int *t);
```

Solution to Problem 3.51 (page 289)

This function is an example of a leaf function that requires local storage. It can use space beyond the stack pointer for its local storage, never altering the stack pointer.

A. Stack locations used:



B.

```
    x86-64 implementation of local_array
    Argument i in %edi
1   local_array:
2     movq    $2, -40(%rsp)            Store 2 in a[0]
3     movq    $3, -32(%rsp)            Store 3 in a[1]
4     movq    $5, -24(%rsp)            Store 5 in a[2]
5     movq    $7, -16(%rsp)            Store 7 in a[3]
6     andl    $3, %edi                 Compute idx = i & 3
7     movq    -40(%rsp,%rdi,8), %rax   Compute a[idx] as return value
8     ret                              Return
```

C. The function never changes the stack pointer. It stores all of its local values in the region beyond the stack pointer.

Solution to Problem 3.52 (page 290)

A. Register `%rbx` is used to hold the parameter `x`.

B. Since %rbx is callee-saved, it must be stored on the stack. Since this is the only use of the stack for this function, the code uses push and pop instructions to save and restore the register.

C.
```
    x86-64 implementation of recursive factorial function rfact
    Argument: x in %rdi
 1   rfact:
 2     pushq   %rbx              Save %rbx (callee save)
 3     movq    %rdi, %rbx        Copy x to %rbx
 4     movl    $1, %eax          result = 1
 5     testq   %rdi, %rdi        Test x
 6     jle     .L11              If <=0, goto done
 7     leaq    -1(%rdi), %rdi    Compute xm1 = x-1
 8     call    rfact             Call rfact(xm1)
 9     imulq   %rbx, %rax        Compute result = x*rfact(xm1)
10   .L11:                     done:
11     popq    %rbx              Restore %rbx
12     ret                       Return
```

D. Instead of explicitly decrementing and incrementing the stack pointer, the code can use pushq and popq to both modify the stack pointer and to save and restore register state.

Solution to Problem 3.53 (page 291)

This problem is similar to Problem 3.41, but updated for x86-64.

A. `struct P1 { int i; char c; long j; char d; };`

i	c	j	d	Total	Alignment
0	4	8	16	24	8

B. `struct P2 { long i; char c; char d; int j; };`

i	c	d	j	Total	Alignment
0	8	9	12	16	8

C. `struct P3 { short w[3]; char c[3] };`

w	c	Total	Alignment
0	6	10	2

D. `struct P4 { short w[3]; char *c[3] };`

w	c	Total	Alignment
0	8	32	8

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

a	p	Total	Alignment
0	48	56	8