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 understand how program data are shared or kept private by the different threads and precisely how and where shared data are accessed. Such information is visible at the machine-code level. As another example, many of the ways programs can be attacked, allowing malware 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 machine code has shifted over the years from one of being able to write programs directly in assembly code to one of being able to read and understand the code generated by compilers.

In this chapter, we will learn the details of one particular assembly language and see how C programs get compiled into this form 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 x86-64, the machine language for most of the processors found in today's laptop and desktop machines, as well as those that power very large data centers and supercomputers. This language has evolved over a long history, starting with Intel Corporation's first 16-bit processor in 1978, through to the expansion to 32 bits, and most recently to 64 bits. Along the way, features have been added to make better use of the available semiconductor technology, and to satisfy the demands of the marketplace. Much of the development has been driven by Intel, but its rival Advanced Micro Devices (AMD) has also made important contributions. 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 many of the arcane features of x86-64.

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 x86-64, 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

Web Aside ASM:IA32 IA32 programming

IA32, the 32-bit predecessor to x86-64, was introduced by Intel in 1985. It served as the machine language of choice for several decades. Most x86 microprocessors sold today, and most operating systems installed on these machines, are designed to run x86-64. However, they can also execute IA32 programs in a backward compatibility mode. As a result, many application programs are still based on IA32. In addition, many existing systems cannot execute x86-64, due to limitations of their hardware or system software. IA32 continues to be an important machine language. You will find that having a background in x86-64 will enable you to learn the IA32 machine language quite readily.

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. The chapter concludes with a presentation on machine-program representations of code involving floating-point data and operations.

The computer industry has recently made the transition from 32-bit to 64-bit machines. A 32-bit machine can only make use of around 4 gigabytes (2^{32} bytes) of random access memory, With memory prices dropping at dramatic rates, and our computational demands and data sizes increasing, it has become both economically feasible and technically desirable to go beyond this limitation. Current 64-bit machines can use up to 256 terabytes (2^{48} bytes) of memory, and could readily be extended to use up to 16 exabytes (2^{64} bytes). Although it is hard to imagine having a machine with that much memory, keep in mind that 4 gigabytes seemed like an extreme amount of memory when 32-bit machines became commonplace in the 1970s and 1980s.

Our presentation focuses on the types of machine-level programs generated when compiling C and similar programming languages targeting modern operating systems. As a consequence, we make no attempt to describe many of the features of x86-64 that arise out of its legacy support for the styles of programs written in the early days of microprocessors, when much of the code was written manually and where programmers had to struggle with the limited range of addresses allowed by 16-bit 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. In this table, "K" denotes 1,000 (10^3), "M" denotes 1,000,000 (10^6), and "G" denotes 1,000,000,000 (10^9).

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 operating system. This was the first machine in the series that could fully 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/MMX (1997, 4.5 M transistors). Added new class of instructions to the Pentium processor for manipulating vectors of integers. Each datum can be 1, 2, or 4 bytes long. Each vector totals 64 bits.

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.

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 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, Nehalem (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.

Core i7, Sandy Bridge (2011, 1.17 G transistors). Introduced AVX, an extension of the SSE to support data packed into 256-bit vectors.

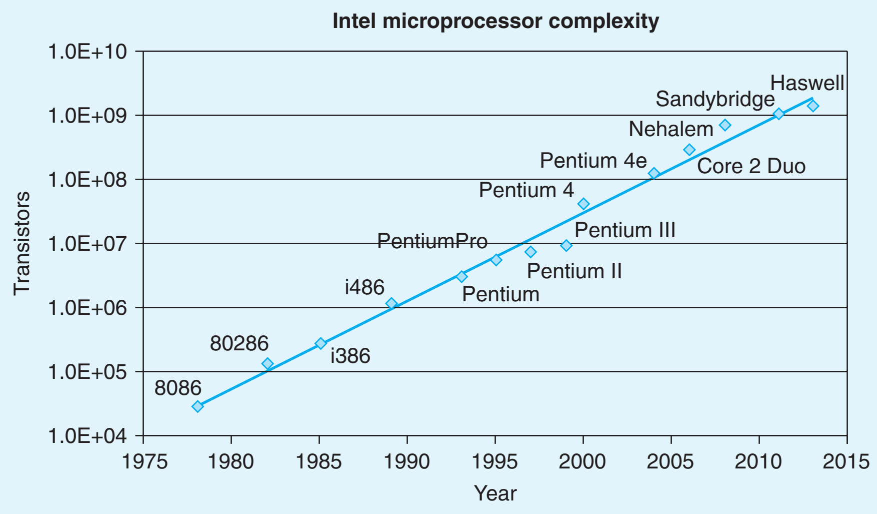
Core i7, Haswell (2013, 1.4 G transistors). Extended AVX to AVX2, adding more instructions and instruction formats.

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.

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 Intel's 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 became obsolete with the i386. The original x87 floating-point instructions became obsolete

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 37%, 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 (by 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 prediction 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 50 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, including the storage capacities of magnetic disks and semiconductor memories. These remarkable growth rates have been the major driving forces of the computer revolution.

with the introduction of SSE2. Although we see vestiges of the historical evolution of x86 in x86-64 programs, many of the most arcane features of x86 do not appear.

3.2 Program Encodings

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

```
linux> gcc -Og -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 -Og[1] instructs the compiler to apply a level of optimization that yields machine code that follows the overall structure of the original C code. 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 -Og optimization as a learning tool and then see what happens as we increase the level of optimization. In practice, higher levels of optimization (e.g., specified with the option -O1 or -O2) are considered a better choice in terms of the resulting program performance.

The gcc command invokes an entire 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 (as specified by the command-line directive -o 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 1.9.3, 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 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 it employs 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

1. This optimization level was introduced in GCC version 4.8. Earlier versions of GCC, as well as non-GNU compilers, will not recognize this option. For these, using optimization level one (specified with the command-line flag -O1) is probably the best choice for generating code that follows the original program structure.

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 provided 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.

The machine code for x86-64 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 `%rip` in x86-64) indicates the address in memory of the next instruction to be executed.
- The integer *register file* contains 16 named locations storing 64-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 arguments and local variables of a procedure, as well as 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 vector registers can each hold one or more integer or floating-point values.

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 (e.g., 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, x86-64 virtual addresses are represented by 64-bit words. In current implementations of these machines, the upper 16 bits must be set to zero, and so an address can potentially specify a byte over a range of 2^{48}, or 64 terabytes. More typical programs will only have access to a few megabytes, or perhaps several gigabytes. The operating system manages

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.

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.

3.2.2 Code Examples

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

```
long mult2(long, long);

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

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

```
linux> gcc -Og -S mstore.c
```

This will cause GCC to run the compiler, generating an assembly file `mstore.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 following set of lines:

```
multstore:
  pushq   %rbx
```

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

To display the binary object code for a program (say, `mstore`), we use a *disassembler* (described below) to determine that the code for the procedure is 14 bytes long. Then we run the GNU debugging tool GDB on file `mstore.o` and give it the command

```
(gdb) x/14xb multstore
```

telling it to display (abbreviated '`x`') 14 hex-formatted (also '`x`') bytes ('`b`') starting at the address where function `multstore` is located. You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 3.10.2.

```
  movq    %rdx, %rbx
  call    mult2
  movq    %rax, (%rbx)
  popq    %rbx
  ret
```

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

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

```
linux> gcc -Og -c mstore.c
```

This will generate an object-code file `mstore.o` that is in binary format and hence cannot be viewed directly. Embedded within the 1,368 bytes of the file `mstore.o` is a 14-byte sequence with the hexadecimal representation

```
53 48 89 d3 e8 00 00 00 00 48 89 03 5b c3
```

This is the object code corresponding to the assembly instructions listed previously. A key lesson to learn from this is that the program 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.

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:

```
linux> objdump -d mstore.o
```

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

```
Disassembly of function sum in binary file mstore.o
0000000000000000 <multstore>:
Offset   Bytes                          Equivalent assembly language
   0:    53                             push   %rbx
   1:    48 89 d3                       mov    %rdx,%rbx
   4:    e8 00 00 00 00                 callq  9 <multstore+0x9>
   9:    48 89 03                       mov    %rax,(%rbx)
   c:    5b                             pop    %rbx
   d:    c3                             retq
```

On the left we see the 14 hexadecimal byte values, listed in the byte sequence shown earlier, partitioned into groups of 1 to 5 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:

- x86-64 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 `pushq %rbx` can start with byte value `53`.
- 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 '`q`' from many of the instructions. These suffixes are size designators and can be omitted in most cases. Conversely, the disassembler adds the suffix '`q`' to the `call` and `ret` instructions. Again, these suffixes can safely be omitted.

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:

```
#include <stdio.h>

void multstore(long, long, long *);

int main() {
    long d;
    multstore(2, 3, &d);
    printf("2 * 3 --> %ld\n", d);
    return 0;
}
```

```
long mult2(long a, long b) {
    long s = a * b;
    return s;
 }
```

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

```
linux> gcc -Og -o prog main.c mstore.c
```

The file `prog` has grown to 8,655 bytes, since it contains not just the machine code for the procedures we provided but also code used to start and terminate the program as well as to interact with the operating system.

We can disassemble the file `prog`:

```
linux> objdump -d prog
```

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

```
     Disassembly of function sum in binary file prog
1    0000000000400540 <multstore>:
2      400540:  53                     push   %rbx
3      400541:  48 89 d3               mov    %rdx,%rbx
4      400544:  e8 42 00 00 00         callq  40058b <mult2>
5      400549:  48 89 03               mov    %rax,(%rbx)
6      40054c:  5b                     pop    %rbx
7      40054d:  c3                     retq
8      40054e:  90                     nop
9      40054f:  90                     nop
```

This code is almost identical to that generated by the disassembly of `mstore.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 filled in the address that the `callq` instruction should use in calling the function `mult2` (line 4 of the disassembly). One task for the linker is to match function calls with the locations of the executable code for those functions. A final difference is that we see two additional lines of code (lines 8–9). These instructions will have no effect on the program, since they occur after the return instruction (line 7). They have been inserted to grow the code for the function to 16 bytes, enabling a better placement of the next block of code in terms of memory system performance.

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 we give the command

```
linux> gcc -Og -S mstore.c
```

to generate the file `mstore.s`. The full content of the file is as follows:

```
        .file   "010-mstore.c"
        .text
        .globl  multstore
        .type   multstore, @function
multstore:
        pushq   %rbx
        movq    %rdx, %rbx
        call    mult2
        movq    %rax, (%rbx)
        popq    %rbx
        ret
        .size   multstore, .-multstore
        .ident  "GCC: (Ubuntu 4.8.1-2ubuntu1~12.04) 4.8.1"
        .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:

```
     void multstore(long x, long y, long *dest)
     x in %rdi, y in %rsi, dest in %rdx
1    multstore:
2      pushq   %rbx             Save %rbx
3      movq    %rdx, %rbx       Copy dest to %rbx
4      call    mult2            Call mult2(x, y)
5      movq    %rax, (%rbx)     Store result at *dest
6      popq    %rbx             Restore %rbx
7      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.

We also provide Web asides to cover material intended for dedicated machine-language enthusiasts. One Web aside describes IA32 machine code. Having a background in x86-64 makes learning IA32 fairly simple. 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

Aside ATT versus Intel assembly-code formats

In our presentation, we show assembly code in ATT format (named after AT&T, the company that operated Bell Laboratories for many years), 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:

```
linux> gcc -Og -S -masm=intel mstore.c
```

This gives the following assembly code:

```
multstore:
  push    rbx
  mov     rbx, rdx
  call    mult2
  mov     QWORD PTR [rbx], rax
  pop     rbx
  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 `push` and `mov` instead of `pushq` and `movq`.
- The Intel code omits the '`%`' character in front of register names, using `rbx` instead of `%rbx`.
- The Intel code has a different way of describing locations in memory—for example, `QWORD PTR [rbx]` rather than `(%rbx)`.
- 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 documentation from Intel and Microsoft.

second is to use GCC's support for embedding assembly code directly within C programs.

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," and 64-bit quantities as "quad words." Figure 3.1 shows the x86-64 representations used for the primitive data types of C. Standard `int` values are stored as double words (32 bits). Pointers (shown here as `char *`) are stored as 8-byte quad words, as would be expected in a 64-bit machine. With x86-64, data type `long` is implemented with 64 bits, allowing a very wide range of values. Most of our code examples in this chapter use pointers and `long` data

Web Aside ASM:EASM Combining assembly code with C programs

Although a C compiler does a good job of converting the computations expressed in a program into machine code, there are some features of a machine that cannot be accessed by a C program. For example, every time an x86-64 processor executes an arithmetic or logical operation, it sets a 1-bit *condition code* flag, named `PF` (for "parity flag"), to 1 when the lower 8 bits in the resulting computation have an even number of ones and to 0 otherwise. Computing this information in C requires at least seven shifting, masking, and EXCLUSIVE-OR operations (see Problem 2.65). Even though the hardware performs this computation as part of every arithmetic or logical operation, there is no way for a C program to determine the value of the `PF` condition code flag. This task can readily be performed by incorporating a small number of assembly-code instructions into the program.

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 x86-64), and so it should only be used when the desired feature can only be accessed in this way.

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

types, and so they will operate on quad words. The x86-64 instruction set includes a full complement of instructions for bytes, words, and double words as well.

Floating-point numbers come in two principal formats: single-precision (4-byte) values, corresponding to C data type `float`, and double-precision (8-byte) values, corresponding to C data type `double`. Microprocessors in the x86 family historically implemented all floating-point operations with a special 80-bit (10-byte) floating-point format (see Problem 2.86). This format can be specified in C programs using the declaration `long double`. We recommend against using this format, however. It is not portable to other classes of machines, and it is typically not implemented with the same high-performance hardware as is the case for single- and double-precision arithmetic.

As the table of Figure 3.1 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 four variants: `movb` (move byte), `movw` (move word), `movl` (move double word), and `movq` (move quad word). The suffix '`l`' is used for double words, since 32-bit quantities are considered to be "long words." The assembly code uses the suffix '`l`' to denote a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating-point code involves an entirely different set of instructions and registers.

3.4 Accessing Information

An x86-64 central processing unit (CPU) contains a set of 16 *general-purpose registers* storing 64-bit values. These registers are used to store integer data as well as pointers. Figure 3.2 diagrams the 16 registers. Their names all begin with `%r`, but otherwise follow multiple different naming conventions, owing to the historical evolution of the instruction set. The original 8086 had eight 16-bit registers, shown in Figure 3.2 as registers `%ax` through `%bp`. Each had a specific purpose, and hence they were given names that reflected how they were to be used. With the extension to IA32, these registers were expanded to 32-bit registers, labeled `%eax` through `%ebp`. In the extension to x86-64, the original eight registers were expanded to 64 bits, labeled `%rax` through `%rbp`. In addition, eight new registers were added, and these were given labels according to a new naming convention: `%r8` through `%r15`.

As the nested boxes in Figure 3.2 indicate, instructions can operate on data of different sizes stored in the low-order bytes of the 16 registers. Byte-level operations can access the least significant byte, 16-bit operations can access the least significant 2 bytes, 32-bit operations can access the least significant 4 bytes, and 64-bit operations can access entire registers.

In later sections, we will present a number of instructions for copying and generating 1-, 2-, 4-, and 8-byte values. When these instructions have registers as destinations, two conventions arise for what happens to the remaining bytes in the register for instructions that generate less than 8 bytes: Those that generate 1- or 2-byte quantities leave the remaining bytes unchanged. Those that generate 4-byte quantities set the upper 4 bytes of the register to zero. The latter convention was adopted as part of the expansion from IA32 to x86-64.

As the annotations along the right-hand side of Figure 3.2 indicate, different registers serve different roles in typical programs. Most unique among them is the stack pointer, `%rsp`, used to indicate the end position in the run-time stack. Some instructions specifically read and write this register. The other 15 registers have more flexibility in their uses. A small number of instructions make specific use of certain registers. More importantly, a set of standard programming conventions governs how the registers are to be used for managing the stack, passing function

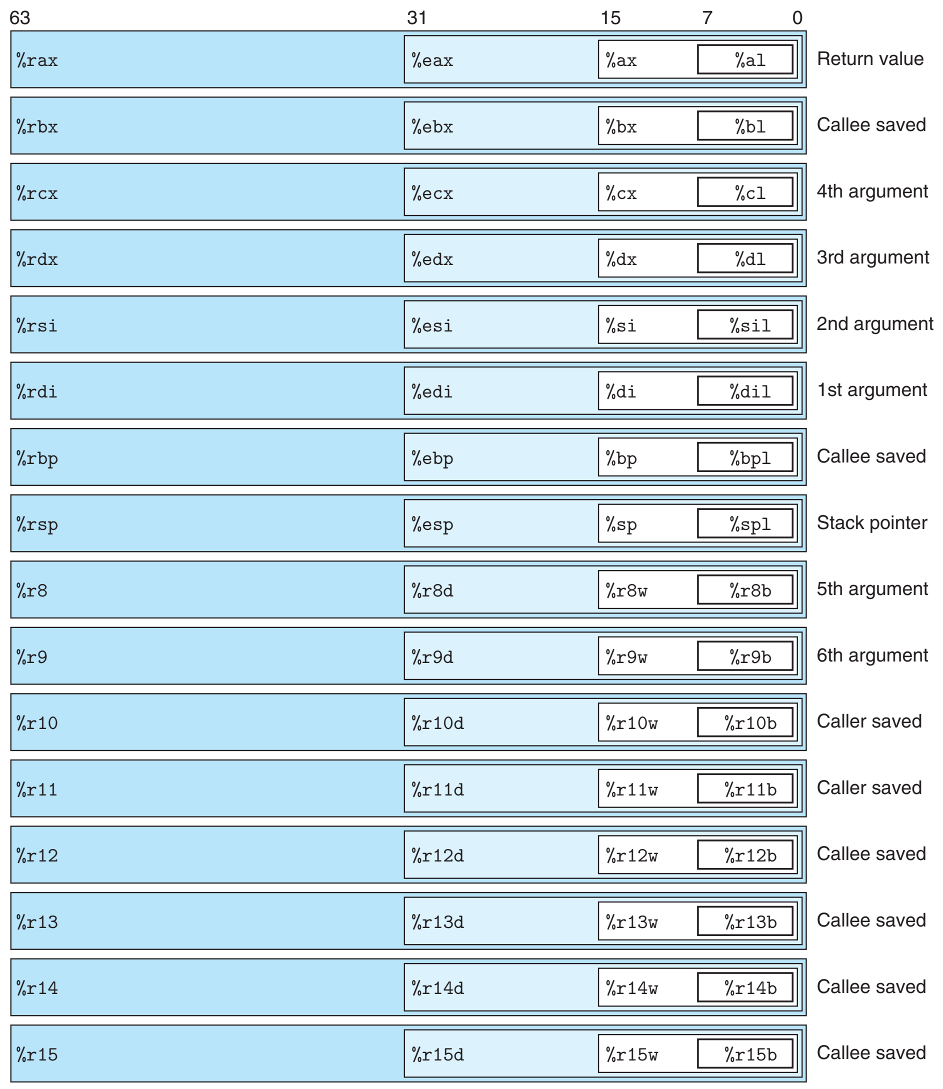


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

arguments, returning values from functions, and storing local and temporary data. We will cover these conventions in our presentation, especially in Section 3.7, where we describe the implementation of procedures.

3.4.1 Operand Specifiers

Most instructions have one or more *operands* specifying the source values to use in performing an operation and the destination location into which to place the

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$\mathtt{r}_a$	$\mathsf{R}[\mathtt{r}_a]$	Register
Memory	Imm	$\mathsf{M}[Imm]$	Absolute
Memory	$(\mathtt{r}_a)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_a]]$	Indirect
Memory	$Imm(\mathtt{r}_b)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b]]$	Base + displacement
Memory	$(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i]]$	Indexed
Memory	$(,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[\mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathtt{r}_b,\mathtt{r}_i,s)$	$\mathsf{M}[Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s]$	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.

result. x86-64 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`. Different instructions allow different ranges of immediate values; the assembler will automatically select the most compact way of encoding a value. The second type, *register*, denotes the contents of a register, one of the sixteen 8-, 4-, 2-, or 1-byte low-order portions of the registers for operands having 64, 32, 16, or 8 bits, respectively. In Figure 3.3, we use the notation $\mathtt{r}_a$ to denote an arbitrary register a and indicate its value with the reference $\mathsf{R}[\mathtt{r}_a]$, 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 $\mathsf{M}_b[Addr]$ 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 $Imm(\mathtt{r}_b,\mathtt{r}_i,s)$. Such a reference has four components: an immediate offset Imm, a base register $\mathtt{r}_b$, an index register $\mathtt{r}_i$, and a scale factor s, where s must be 1, 2, 4, or 8. Both the base and index must be 64-bit registers. The effective address is computed as $Imm + \mathsf{R}[\mathtt{r}_b] + \mathsf{R}[\mathtt{r}_i] \cdot s$. 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 **(solution page 361)**

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

Address	Value	Register	Value
`0x100`	`0xFF`	`%rax`	`0x100`
`0x104`	`0xAB`	`%rcx`	`0x1`
`0x108`	`0x13`	`%rdx`	`0x3`
`0x10C`	`0x11`		

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

Operand	Value
`%rax`	______
`0x104`	______
`$0x108`	______
`(%rax)`	______
`4(%rax)`	______
`9(%rax,%rdx)`	______
`260(%rcx,%rdx)`	______
`0xFC(,%rcx,4)`	______
`(%rax,%rdx,4)`	______

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 express a range of possibilities that in many machines would require a number of different instructions. We present a number of different data movement instructions, differing in their source and destination types, what conversions they perform, and other side effects they may have. In our presentation, we group the many different instructions into *instruction classes*, where the instructions in a class perform the same operation but with different operand sizes.

Figure 3.4 lists the simplest form of data movement instructions—MOV class. These instructions copy data from a source location to a destination location, without any transformation. The class consists of four instructions: `movb`, `movw`, `movl`, and `movq`. All four of these instructions have similar effects; they differ primarily in that they operate on data of different sizes: 1, 2, 4, and 8 bytes, respectively.

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

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. x86-64 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, register operands for these instructions can be the labeled portions of any of the 16 registers, where the size of the register must match the size designated by the last character of the instruction (‘b’, ‘w’, ‘l’, or ‘q’). For most cases, the MOV instructions will only update the specific register bytes or memory locations indicated by the destination operand. The only exception is that when movl has a register as the destination, it will also set the high-order 4 bytes of the register to 0. This exception arises from the convention, adopted in x86-64, that any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

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.

```
movl $0x4050,%eax        Immediate--Register, 4 bytes
movw %bp,%sp             Register--Register,  2 bytes
movb (%rdi,%rcx),%al     Memory--Register,    1 byte
movb $-17,(%esp)         Immediate--Memory,   1 byte
movq %rax,-12(%rbp)      Register--Memory,    8 bytes
```

A final instruction documented in Figure 3.4 is for dealing with 64-bit immediate data. The regular movq instruction can only have immediate source operands that can be represented as 32-bit two’s-complement numbers. This value is then sign extended to produce the 64-bit value for the destination. The movabsq instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

Figures 3.5 and 3.6 document two classes of data movement instructions for use when copying a smaller source value to a larger destination. All of these instructions copy data from a source, which can be either a register or stored

Aside Understanding how data movement changes a destination register

As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register. This distinction is illustrated by the following code sequence:

```
1    movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2    movb    $-1, %al                     %rax = 00112233445566FF
3    movw    $-1, %ax                     %rax = 001122334455FFFF
4    movl    $-1, %eax                    %rax = 00000000FFFFFFFF
5    movq    $-1, %rax                    %rax = FFFFFFFFFFFFFFFF
```

In the following discussion, we use hexadecimal notation. In the example, the instruction on line 1 initializes register `%rax` to the pattern `0011223344556677`. The remaining instructions have immediate value −1 as their source values. Recall that the hexadecimal representation of −1 is of the form `FF···F`, where the number of `F`'s is twice the number of bytes in the representation. The `movb` instruction (line 2) therefore sets the low-order byte of `%rax` to `FF`, while the `movw` instruction (line 3) sets the low-order 2 bytes to `FFFF`, with the remaining bytes unchanged. The `movl` instruction (line 4) sets the low-order 4 bytes to `FFFFFFFF`, but it also sets the high-order 4 bytes to `00000000`. Finally, the `movq` instruction (line 5) sets the complete register to `FFFFFFFFFFFFFFFF`.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow$ 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
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

in memory, to a register destination. Instructions in the MOVZ class fill out the remaining bytes of the destination with zeros, while those in the MOVS class fill them out by sign extension, replicating copies of the most significant bit of the source operand. Observe that each instruction name has size designators as its final two characters—the first specifying the source size, and the second specifying the destination size. 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, considering only cases where the destination is larger than the source, of course.

Instruction	Effect	Description
MOVS *S*, *R*	*R* ← 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
`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
`cltq`	`%rax` ← SignExtend(`%eax`)	Sign-extend `%eax` to `%rax`

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers `%eax` and `%rax`.

Note the absence of an explicit instruction to zero-extend a 4-byte source value to an 8-byte destination in Figure 3.5. Such an instruction would logically be named `movzlq`, but this instruction does not exist. Instead, this type of data movement can be implemented using a `movl` instruction having a register as the destination. This technique takes advantage of the property that an instruction generating a 4-byte value with a register as the destination will fill the upper 4 bytes with zeros. Otherwise, for 64-bit destinations, moving with sign extension is supported for all three source types, and moving with zero extension is supported for the two smaller source types.

Figure 3.6 also documents the `cltq` instruction. This instruction has no operands—it always uses register `%eax` as its source and `%rax` as the destination for the sign-extended result. It therefore has the exact same effect as the instruction `movslq %eax, %rax`, but it has a more compact encoding.

Practice Problem 3.2 (solution page 361)

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`, `movl`, or `movq`.)

```
mov__   %eax, (%rsp)
mov__   (%rax), %dx
mov__   $0xFF, %bl
mov__   (%rsp,%rdx,4), %dl
mov__   (%rdx), %rax
mov__   %dx, (%rax)
```

Aside Comparing byte movement instructions

The following example illustrates how different data movement instructions either do or do not change the high-order bytes of the destination. Observe that the three byte-movement instructions `movb`, `movsbq`, and `movzbq` differ from each other in subtle ways. Here is an example:

```
1      movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2      movb    $0xAA, %dl                   %dl  = AA
3      movb %dl,%al                         %rax = 00112233445566AA
4      movsbq %dl,%rax                      %rax = FFFFFFFFFFFFFFAA
5      movzbq %dl,%rax                      %rax = 00000000000000AA
```

In the following discussion, we use hexadecimal notation for all of the values. The first two lines of the code initialize registers `%rax` and `%dl` to `0011223344556677` and `AA`, respectively. The remaining instructions all copy the low-order byte of `%rdx` to the low-order byte of `%rax`. The `movb` instruction (line 3) does not change the other bytes. The `movsbq` instruction (line 4) sets the other 7 bytes to either all ones or all zeros depending on the high-order bit of the source byte. Since hexadecimal `A` represents binary value `1010`, sign extension causes the higher-order bytes to each be set to `FF`. The `movzbq` instruction (line 5) always sets the other 7 bytes to zero.

Practice Problem 3.3 **(solution page 362)**

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

```
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax),4(%rsp)
movb %al,%sl
movq %rax,$0x123
movl %eax,%rdx
movb %si, 8(%rbp)
```

3.4.3 Data Movement Example

As an example of code that uses data movement instructions, consider the data exchange routine shown in Figure 3.7, both as C code and as assembly code generated by GCC.

As Figure 3.7(b) shows, function `exchange` is implemented with just three instructions: two data movements (`movq`) plus an instruction to return back to the point from which the function was called (`ret`). We will cover the details of function call and return in Section 3.7. Until then, it suffices to say that arguments are passed to functions in registers. Our annotated assembly code documents these. A function returns a value by storing it in register `%rax`, or in one of the low-order portions of this register.

(a) C code

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
     long exchange(long *xp, long y)
     xp in %rdi, y in %rsi
1    exchange:
2      movq    (%rdi), %rax      Get x at xp. Set as return value.
3      movq    %rsi, (%rdi)      Store y at xp.
4      ret                       Return.
```

Figure 3.7 **C and assembly code for exchange routine.** Registers `%rdi` and `%rsi` hold parameters `xp` and `y`, respectively.

When the procedure begins execution, procedure parameters `xp` and `y` are stored in registers `%rdi` and `%rsi`, respectively. Instruction 2 then reads `x` from memory and stores the value in register `%rax`, a direct implementation of the operation `x = *xp` in the C program. Later, register `%rax` will be used to return a value from the function, and so the return value will be `x`. Instruction 3 writes `y` to the memory location designated by `xp` in register `%rdi`, 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 (line 2), and to write from a register to memory (line 3).

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 (solution page 362)

Assume variables `sp` and `dp` are declared with types

```
src_t  *sp;
dest_t *dp;
```

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

```
*dp = (dest_t) *sp;
```

New to C? Some examples of pointers

Function `exchange` (Figure 3.7(a)) provides a good illustration of the use of pointers in C. Argument `xp` is a pointer to a long integer, while `y` is a long integer itself. The statement

```
long 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:

```
long a = 4;
long b = exchange(&a, 3);
printf("a = %ld, b = %ld\verb@\@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` overwrites the value stored in `a` with 3 but returns the previous value, 4, as the function value. Observe how by passing a pointer to `exchange`, it could modify data held at some remote location.

Assume that the values of `sp` and `dp` are stored in registers `%rdi` and `%rsi`, respectively. For each entry in the table, show the two instructions that implement the specified data movement. The first instruction in the sequence should read from memory, do the appropriate conversion, and set the appropriate portion of register `%rax`. The second instruction should then write the appropriate portion of `%rax` to memory. In both cases, the portions may be `%rax`, `%eax`, `%ax`, or `%al`, and they may differ from one another.

Recall that when performing a cast that involves both a size change and a change of "signedness" in C, the operation should change the size first (Section 2.2.6).

`src_t`	`dest_t`	Instruction
`long`	`long`	`movq (%rdi), %rax`
		`movq %rax, (%rsi)`
`char`	`int`	____________
		____________

`char`	`unsigned`	
`unsigned char`	`long`	
`int`	`char`	
`unsigned`	`unsigned char`	
`char`	`short`	

Practice Problem 3.5 (solution page 363)

You are given the following information. A function with prototype

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

is compiled into assembly code, yielding the following:

```
  void decode1(long *xp, long *yp, long *zp)
  xp in %rdi, yp in %rsi, zp in %rdx
decode1:
  movq    (%rdi), %r8
  movq    (%rsi), %rcx
  movq    (%rdx), %rax
  movq    %r8, (%rsi)
  movq    %rcx, (%rdx)
  movq    %rax, (%rdi)
  ret
```

Parameters `xp`, `yp`, and `zp` are stored in registers `%rdi`, `%rsi`, and `%rdx`, respectively.

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

3.4.4 Pushing and Popping Stack Data

The final two data movement operations are used to push data onto and pop data from the program stack, as documented in Figure 3.8. 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

Instruction	Effect	Description
`pushq` S	R[`%rsp`] $\leftarrow$ R[`%rsp`] $-$ 8;	Push quad word
	M[R[`%rsp`]] $\leftarrow$ S	
`popq` D	D $\leftarrow$ M[R[`%rsp`]];	Pop quad word
	R[`%rsp`] $\leftarrow$ R[`%rsp`] $+$ 8	

Figure 3.8 Push and pop instructions.

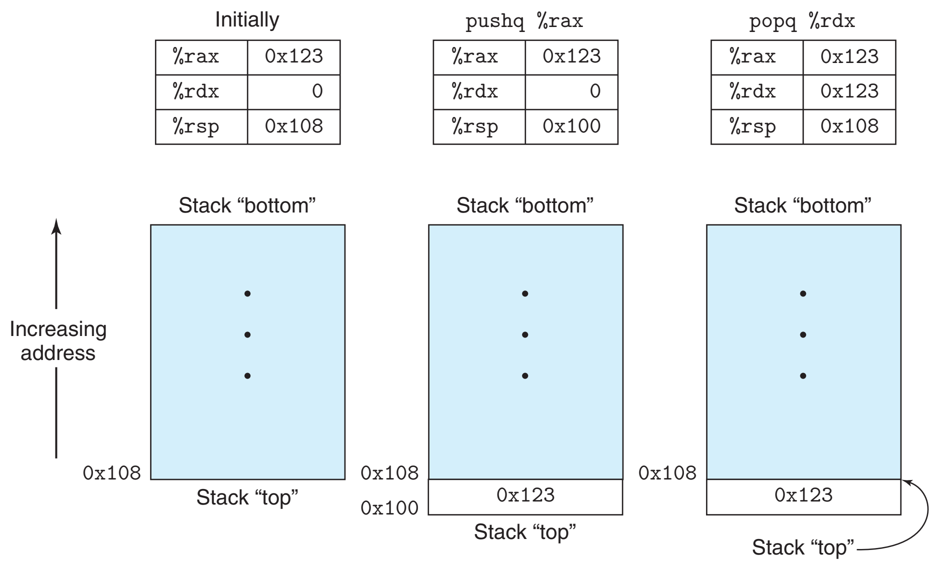


Figure 3.9 Illustration of stack operation. By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register `%rsp`) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

end of the array. This end is called the *top* of the stack. With x86-64, the program stack is stored in some region of memory. As illustrated in Figure 3.9, 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 `%rsp` holds the address of the top stack element.

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

Pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 and then writing the value at the new top-of-stack address.

Therefore, the behavior of the instruction pushq %rbp is equivalent to that of the pair of instructions

```
subq $8,%rsp            Decrement stack pointer
movq %rbp,(%rsp)        Store %rbp on stack
```

except that the pushq instruction is encoded in the machine code as a single byte, whereas the pair of instructions shown above requires a total of 8 bytes. The first two columns in Figure 3.9 illustrate the effect of executing the instruction pushq %rax when %rsp is 0x108 and %rax is 0x123. First %rsp is decremented by 8, giving 0x100, and then 0x123 is stored at memory address 0x100.

Popping a quad word involves reading from the top-of-stack location and then incrementing the stack pointer by 8. Therefore, the instruction popq %rax is equivalent to the following pair of instructions:

```
movq (%rsp),%rax        Read %rax from stack
addq $8,%rsp            Increment stack pointer
```

The third column of Figure 3.9 illustrates the effect of executing the instruction popq %edx immediately after executing the pushq. Value 0x123 is read from memory and written to register %rdx. Register %rsp 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 %rsp.

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 quad word, the instruction movq 8(%rsp),%rdx will copy the second quad word from the stack to register %rdx.

3.5 Arithmetic and Logical Operations

Figure 3.10 lists some of the x86-64 integer and logic operations. Most of the operations are given as instruction classes, as they can have different variants with different operand sizes. (Only leaq has no other size variants.) For example, the instruction class ADD consists of four addition instructions: addb, addw, addl, and addq, adding bytes, words, double words, and quad words, respectively. Indeed, each of the instruction classes shown has instructions for operating on these four different sizes of 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 leaq is actually a variant of the movq instruction. It has the form of an instruction that reads from memory to a register,

Instruction		Effect	Description
`leaq`	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	$D \leftarrow D \mathbin{\char`^} S$	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 3.10 Integer arithmetic operations. The load effective address (`leaq`) 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.

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 3.10 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 `%rdx` contains value x, then the instruction `leaq 7(%rdx,%rdx,4), %rax` will set register `%rax` to $5x + 7$. Compilers often find clever uses of `leaq` that have nothing to do with effective address computations. The destination operand must be a register.

Practice Problem 3.6 (solution page 363)

Suppose register `%rbx` holds value p and `%rdx` holds value q. Fill in the table below with formulas indicating the value that will be stored in register `%rax` for each of the given assembly-code instructions:

Instruction	Result
`leaq 9(%rdx), %rax`	
`leaq (%rdx,%rbx), %rax`	
`leaq (%rdx,%rbx,3), %rax`	
`leaq 2(%rbx,%rbx,7), %rax`	

```
leaq 0xE(,%rdx,3), %rax        ________
leaq 6(%rbx,%rdx,7), %rax      ________
```

As an illustration of the use of `leaq` in compiled code, consider the following C program:

```
long scale(long x, long y, long z) {
    long t = x + 4 * y + 12 * z;
    return t;
}
```

When compiled, the arithmetic operations of the function are implemented by a sequence of three `leaq` functions, as is documented by the comments on the right-hand side:

```
  long scale(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
scale:
  leaq    (%rdi,%rsi,4), %rax     x + 4*y
  leaq    (%rdx,%rdx,2), %rdx     z + 2*z = 3*z
  leaq    (%rax,%rdx,4), %rax     (x+4*y) + 4*(3*z) = x + 4*y + 12*z
  ret
```

The ability of the `leaq` instruction to perform addition and limited forms of multiplication proves useful when compiling simple arithmetic expressions such as this example.

Practice Problem 3.7 (solution page 364)

Consider the following code, in which we have omitted the expression being computed:

```
short scale3(short x, short y, short z) {
  short t = __________;
  return t;
}
```

Compiling the actual function with GCC yields the following assembly code:

```
  short scale3(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
scale3:
  leaq (%rsi,%rsi,9), %rbx
  leaq (%rbx,%rdx), %rbx
  leaq (%rbx,%rdi,%rsi), %rbx
  ret
```

Fill in the missing expression in the C code.

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 `incq (%rsp)` causes the 8-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 `subq %rax,%rdx` decrements register `%rdx` by the value in `%rax`. (It helps to read the instruction as "Subtract `%rax` from `%rdx`.") 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 MOV instructions, the two operands cannot both be memory locations. Note that when the second operand is a memory location, the processor must read the value from memory, perform the operation, and then write the result back to memory.

Practice Problem 3.8 **(solution page 364)**

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

Address	Value	Register	Value
`0x100`	`0xFF`	`%rax`	`0x100`
`0x108`	`0xAB`	`%rcx`	`0x1`
`0x110`	`0x13`	`%rdx`	`0x3`
`0x118`	`0x11`		

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

Instruction	Destination	Value
`addq %rcx,(%rax)`		
`subq %rdx,8(%rax)`		
`imulq $16,(%rax,%rdx,8)`		
`incq 16(%rax)`		
`decq %rcx`		
`subq %rdx,%rax`		

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 different shift instructions can specify the shift amount either as an immediate value or with the single-byte register `%cl`. (These instructions are unusual in only allowing this specific register as the operand.) In principle, having a 1-byte shift amount would make it possible to encode shift amounts ranging up to $2^8 - 1 = 255$. With x86-64, a shift instruction operating on data values that are w bits long determines the shift amount from the low-order m bits of register `%cl`, where $2^m = w$. The higher-order bits are ignored. So, for example, when register `%cl` has hexadecimal value `0xFF`, then instruction `salb` would shift by 7, while `salw` would shift by 15, `sall` would shift by 31, and `salq` would shift by 63.

As Figure 3.10 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 3.10 as $>>_A$ (arithmetic) and $>>_L$ (logical).

Practice Problem 3.9 (solution page 364)

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

```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    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 `%rax`. Two key instructions have been omitted. Parameters `x` and `n` are stored in registers `%rdi` and `%rsi`, respectively.

```
  long shift_left4_rightn(long x, long n)
  x in %rdi, n in %rsi
shift_left4_rightn:
  movq    %rdi, %rax      Get x
  __________________      x <<= 4
  movl    %esi, %ecx      Get n (4 bytes)
  __________________      x >>= n
```

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

(a) C code

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

(b) Assembly code

```
     long arith(long x, long y, long z)
     x in %rdi, y in %rsi, z in %rdx
1    arith:
2      xorq    %rsi, %rdi              t1 = x ^ y
3      leaq    (%rdx,%rdx,2), %rax     3*z
4      salq    $4, %rax                t2 = 16 * (3*z) = 48*z
5      andl    $252645135, %edi        t3 = t1 & 0x0F0F0F0F
6      subq    %rdi, %rax              Return t2 - t3
7      ret
```

Figure 3.11 C and assembly code for arithmetic function.

3.5.4 Discussion

We see that most of the instructions shown in Figure 3.10 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 3.11 shows an example of a function that performs arithmetic operations and its translation into assembly code. Arguments `x`, `y`, and `z` are initially stored in registers `%rdi`, `%rsi`, and `%rdx`, respectively. The assembly-code instructions correspond closely with the lines of C source code. Line 2 computes the value of `x^y`. Lines 3 and 4 compute the expression `z*48` by a combination of `leaq` and shift instructions. Line 5 computes the AND of `t1` and `0x0F0F0F0F`. The final subtraction is computed by line 6. Since the destination of the subtraction is register `%rax`, this will be the value returned by the function.

In the assembly code of Figure 3.11, the sequence of values in register `%rax` corresponds to program values `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 3.10 (solution page 365)

Consider the following code, in which we have omitted the expression being computed:

```
short arith3(short x, short y, short z)
{
    short p1 = ________;
    short p2 = ________;
    short p3 = ________;
    short p4 = ________;
    return p4;
}
```

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

```
  short arith3(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
arith3:
  orq     %rsi, %rdx
  sarq    $9, %rdx
  notq    %rdx
  movq    %rdx, %bax
  subq    %rsi, %rbx
  ret
```

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

Practice Problem 3.11 (solution page 365)

It is common to find assembly-code lines of the form

```
xorq %rcx,%rcx
```

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 any two of these three different implementations of the same operation.

3.5.5 Special Arithmetic Operations

As we saw in Section 2.3, multiplying two 64-bit signed or unsigned integers can yield a product that requires 128 bits to represent. The x86-64 instruction set provides limited support for operations involving 128-bit (16-byte) numbers. Continuing with the naming convention of word (2 bytes), double word (4 bytes), and quad word (8 bytes), Intel refers to a 16-byte quantity as an *oct word*. Figure 3.12

Instruction		Effect	Description
`imulq`	S	R[`%rdx`]:R[`%rax`] $\leftarrow$ $S \times$ R[`%rax`]	Signed full multiply
`mulq`	S	R[`%rdx`]:R[`%rax`] $\leftarrow$ $S \times$ R[`%rax`]	Unsigned full multiply
`cqto`		R[`%rdx`]:R[`%rax`] $\leftarrow$ SignExtend(R[`%rax`])	Convert to oct word
`idivq`	S	R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div$ S	Signed divide
`divq`	S	R[`%rdx`] $\leftarrow$ R[`%rdx`]:R[`%rax`] mod S; R[`%rax`] $\leftarrow$ R[`%rdx`]:R[`%rax`] $\div$ S	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-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.

describes instructions that support generating the full 128-bit product of two 64-bit numbers, as well as integer division.

The `imulq` instruction has two different forms One form, shown in Figure 3.10, is as a member of the IMUL instruction class. In this form, it serves as a "two-operand" multiply instruction, generating a 64-bit product from two 64-bit operands. It implements the operations $*^{u}_{64}$ and $*^{t}_{64}$ described in Sections 2.3.4 and 2.3.5. (Recall that when truncating the product to 64 bits, both unsigned multiply and two's-complement multiply have the same bit-level behavior.)

Additionally, the x86-64 instruction set includes two different "one-operand" multiply instructions to compute the full 128-bit product of two 64-bit values—one for unsigned (`mulq`) and one for two's-complement (`imulq`) multiplication. For both of these instructions, one argument must be in register `%rax`, and the other is given as the instruction source operand. The product is then stored in registers `%rdx` (high-order 64 bits) and `%rax` (low-order 64 bits). Although the name `imulq` is used for two distinct multiplication operations, the assembler can tell which one is intended by counting the number of operands.

As an example, the following C code demonstrates the generation of a 128-bit product of two unsigned 64-bit numbers `x` and `y`:

```
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y) {
    *dest = x * (uint128_t) y;
}
```

In this program, we explicitly declare `x` and `y` to be 64-bit numbers, using definitions declared in the file `inttypes.h`, as part of an extension of the C standard. Unfortunately, this standard does not make provisions for 128-bit values. Instead, we rely on support provided by GCC for 128-bit integers, declared using the name `__int128`. Our code uses a `typedef` declaration to define data type `uint128_t`, following the naming pattern for other data types found in `inttypes.h`. The code specifies that the resulting product should be stored at the 16 bytes designated by pointer `dest`.

The assembly code generated by GCC for this function is as follows:

```
     void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
     dest in %rdi, x in %rsi, y in %rdx
1    store_uprod:
2      movq    %rsi, %rax        Copy x to multiplicand
3      mulq    %rdx              Multiply by y
4      movq    %rax, (%rdi)      Store lower 8 bytes at dest
5      movq    %rdx, 8(%rdi)     Store upper 8 bytes at dest+8
6      ret
```

Observe that storing the product requires two `movq` instructions: one for the low-order 8 bytes (line 4), and one for the high-order 8 bytes (line 5). Since the code is generated for a little-endian machine, the high-order bytes are stored at higher addresses, as indicated by the address specification `8(%rdi)`.

Our earlier table of arithmetic operations (Figure 3.10) 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 its dividend the 128-bit quantity in registers `%rdx` (high-order 64 bits) and `%rax` (low-order 64 bits). The divisor is given as the instruction operand. The instruction stores the quotient in register `%rax` and the remainder in register `%rdx`.

For most applications of 64-bit addition, the dividend is given as a 64-bit value. This value should be stored in register `%rax`. The bits of `%rdx` should then be set to either all zeros (unsigned arithmetic) or the sign bit of `%rax` (signed arithmetic). The latter operation can be performed using the instruction `cqto`.[2] This instruction takes no operands—it implicitly reads the sign bit from `%rax` and copies it across all of `%rdx`.

As an illustration of the implementation of division with x86-64, the following C function computes the quotient and remainder of two 64-bit, signed numbers:

```
void remdiv(long x, long y,
            long *qp, long *rp) {
    long q = x/y;
    long r = x%y;
    *qp = q;
    *rp = r;
}
```

2. This instruction is called `cqo` in the Intel documentation, one of the few cases where the ATT-format name for an instruction does not match the Intel name.

This compiles to the following assembly code:

```
     void remdiv(long x, long y, long *qp, long *rp)
     x in %rdi, y in %rsi, qp in %rdx, rp in %rcx
1    remdiv:
2      movq    %rdx, %r8       Copy qp
3      movq    %rdi, %rax      Move x to lower 8 bytes of dividend
4      cqto                    Sign-extend to upper 8 bytes of dividend
5      idivq   %rsi            Divide by y
6      movq    %rax, (%r8)     Store quotient at qp
7      movq    %rdx, (%rcx)    Store remainder at rp
8      ret
```

In this code, argument rp must first be saved in a different register (line 2), since argument register %rdx is required for the division operation. Lines 3–4 then prepare the dividend by copying and sign-extending x. Following the division, the quotient in register %rax gets stored at qp (line 6), while the remainder in register %rdx gets stored at rp (line 7).

Unsigned division makes use of the divq instruction. Typically, register %rdx is set to zero beforehand.

Practice Problem 3.12 (solution page 365)

Consider the following function for computing the quotient and remainder of two unsigned 64-bit numbers:

```
void uremdiv(unsigned long x, unsigned long y,
             unsigned long *qp, unsigned long *rp) {
    unsigned long q = x/y;
    unsigned long r = x%y;
    *qp = q;
    *rp = r;
}
```

Modify the assembly code shown for signed division to implement this function.

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 get 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 alters either the control flow or the data flow based on the results 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 two ways of implementing conditional operations. We then describe methods for presenting loops and `switch` statements.

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. These condition codes are the most useful:

`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.

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 `leaq` 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 3.10 cause the condition codes to be set. For the logical operations, such as XOR, the carry and overflow flags are set to zero. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero. 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 3.10, there are two instruction classes (having 8-, 16-, 32-, and 64-bit forms) that set condition codes without altering any other registers; these are listed in Figure 3.13. 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,

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

Figure 3.13 Comparison and test instructions. These instructions set the condition codes without updating any other registers.

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., `testq %rax,%rax` to see whether `%rax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

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 3.14 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 low-order 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 or 64-bit result, we must also clear the high-order bits. A typical instruction sequence to compute the C expression `a < b`, where `a` and `b` are both of type `long`, proceeds as follows:

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 3.14 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," that is, alternate names for the same machine instruction.

```
   int comp(data_t a, data_t b)
   a in %rdi, b in %rsi
1  comp:
2    cmpq    %rsi, %rdi     Compare a:b
3    setl    %al            Set low-order byte of %eax to 0 or 1
4    movzbl  %al, %eax      Clear rest of %eax (and rest of %rax)
5    ret
```

Note the comparison order of the `cmpq` instruction (line 2). Although the arguments are listed in the order `%rsi` (`b`), then `%rdi` (`a`), the comparison is really between `a` and `b`. Recall also, as discussed in Section 3.4.2, that the `movzbl` instruction (line 4) clears not just the high-order 3 bytes of `%eax`, but the upper 4 bytes of the entire register, `%rax`, as well.

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 $t = a -^{t}_{w} b$, 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 $a -^{t}_{w} b < 0$, indicated by having `SF` set to 1, and $a \geq b$ when $a -^{t}_{w} b \geq 0$, indicated by having `SF` set to 0. On the other hand, when overflow occurs, we will have $a < b$ when $a -^{t}_{w} b > 0$ (negative overflow) and $a > b$ when $a -^{t}_{w} b < 0$ (positive 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 $a - b < 0$, and so the unsigned comparisons use combinations of the carry and zero flags.

It is important to note how machine code does or does not distinguish 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 (solution page 366)

The C code

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

shows a general comparison between arguments `a` and `b`, where `data_t`, the data type of the arguments, is defined (via `typedef`) to be one of the integer data types listed in Figure 3.1 and either signed or unsigned. The comparison `COMP` is defined via `#define`.

Suppose `a` is in some portion of `%rdx` while `b` is in some portion of `%rsi`. 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    %esi, %edi
setl    %al
```

B.
```
cmpw    %si, %di
setge   %al
```

C.
```
cmpb    %sil, %dil
setbe    %al
```

D.
```
cmpq    %rsi, %rdi
setne   %a
```

Practice Problem 3.14 (solution page 366)

The 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. The following instruction sequences implement the comparison, where `a` is held in some portion of register `%rdi`. For each sequence, 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.
```
testq  %rdi, %rdi
setge  %al
```

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

C.
```
testb    %dil, %dil
seta     %al
```

D.
```
testl    %edi, %edi
setle    %al
```

3.6.3 Jump Instructions

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

```
  movq $0,%rax              Set %rax to 0
  jmp .L1                   Goto .L1
  movq (%rax),%rdx          Null pointer dereference (skipped)
.L1:
  popq %rdx                 Jump target
```

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 3.15 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.

The instruction `jmp .L1` will cause the program to skip over the `movq` instruction and instead resume execution with the `popq` 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 3.15 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 code by giving a label as the jump target, for example, the label `.L1` in the code shown. Indirect jumps are written using '`*`' followed by an operand specifier using one of the memory operand formats described in Figure 3.3. As examples, the instruction

```
jmp *%rax
```

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

```
jmp *(%rax)
```

reads the jump target from memory, using the value in `%rax` 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 3.14). As with the SET instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

3.6.4 Jump Instruction Encodings

For the most part, we will not concern ourselves with the detailed format of machine code. On the other hand, 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 assembly code for a function was generated by compiling a file `branch.c`. It contains two jumps: the `jmp` instruction on line 2 jumps forward to a higher address, while the `jg` instruction on line 7 jumps back to a lower one.

```
1     movq    %rdi, %rax
2     jmp     .L2
3   .L3:
4     sarq    %rax
5   .L2:
6     testq   %rax, %rax
7     jg      .L3
8     rep; ret
```

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

```
1      0:   48 89 f8                mov    %rdi,%rax
2      3:   eb 03                   jmp    8 <loop+0x8>
3      5:   48 d1 f8                sar    %rax
4      8:   48 85 c0                test   %rax,%rax
5      b:   7f f8                   jg     5 <loop+0x5>
6      d:   f3 c3                   repz retq
```

In the annotations on the right generated by the disassembler, the jump targets are indicated as `0x8` for the jump instruction on line 2 and `0x5` for the jump instruction on line 5 (the disassembler lists all numbers in hexadecimal). 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 `0x03`. Adding this to `0x5`, the

Aside What do the instructions `rep` and `repz` do?

Line 8 of the assembly code shown on page 243 contains the instruction combination `rep; ret`. These are rendered in the disassembled code (line 6) as `repz retq`. One can infer that `repz` is a synonym for `rep`, just as `retq` is a synonym for `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, 51]. 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 the destination of a conditional jump instruction. Without the `rep` instruction, the `jg` instruction (line 7 of the assembly code) 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. We can safely ignore any `rep` or `repz` instruction we see in the rest of the code presented in this book.

address of the following instruction, we get jump target address `0x8`, the address of the instruction on line 4.

Similarly, the target of the second jump instruction is encoded as `0xf8` (decimal −8) using a single-byte two's-complement representation. Adding this to `0xd` (decimal 13), the address of the instruction on line 6, we get `0x5`, the address of the instruction on line 3.

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   4004d0:  48 89 f8          mov    %rdi,%rax
2   4004d3:  eb 03             jmp    4004d8 <loop+0x8>
3   4004d5:  48 d1 f8          sar    %rax
4   4004d8:  48 85 c0          test   %rax,%rax
5   4004db:  7f f8             jg     4004d5 <loop+0x5>
6   4004dd:  f3 c3             repz retq
```

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 2 and 5 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 (solution page 366)

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 do not need to know anything about the `callq` instruction here.)

```
4003fa: 74 02                    je     XXXXXX
4003fc: ff d0                    callq  *%rax
```

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

```
40042f: 74 f4                    je     XXXXXX
400431: 5d                       pop    %rbp
```

C. What is the address of the `ja` and `pop` instructions?

```
XXXXXX: 77 02                    ja     400547
XXXXXX: 5d                       pop    %rbp
```

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 x86-64. What is the address of the jump target?

```
4005e8: e9 73 ff ff ff           jmpq   XXXXXXX
4005ed: 90                       nop
```

The jump instructions provide a means to implement conditional execution (`if`), as well as several different loop constructs.

3.6.5 Implementing Conditional Branches with Conditional Control

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 3.16(a) shows the C code for a function that computes the absolute value of the difference of two numbers.[3] The function also has a side effect of incrementing one of two counters, encoded as global variables `lt_cnt` and `ge_cnt`. Gcc generates the assembly code shown as Figure 3.16(c). Our rendition of the machine code into C is shown as the function `gotodiff_se` (Figure 3.16(b)). It uses the `goto` statement in C, which is similar to the unconditional jump of

3. 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

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

(b) Equivalent goto version

```
1   long gotodiff_se(long x, long y)
2   {
3       long result;
4       if (x >= y)
5           goto x_ge_y;
6       lt_cnt++;
7       result =  y - x;
8       return result;
9    x_ge_y:
10      ge_cnt++;
11      result = x - y;
12      return result;
13  }
```

(c) Generated assembly code

```
long absdiff_se(long x, long y)
x in %rdi, y in %rsi
absdiff_se:
  cmpq    %rsi, %rdi             Compare x:y
  jge     .L2                    If >= goto x_ge_y
  addq    $1, lt_cnt(%rip)       lt_cnt++
  movq    %rsi, %rax
  subq    %rdi, %rax             result = y - x
  ret                            Return
.L2:                           x_ge_y:
  addq    $1, ge_cnt(%rip)       ge_cnt++
  movq    %rdi, %rax
  subq    %rsi, %rax             result = x - y
  ret                            Return
```

Figure 3.16 Compilation of conditional statements. (a) C procedure `absdiff_se` contains an if-else statement. The generated assembly code is shown (c), along with (b) a C procedure `gotodiff_se` that mimics the control flow of the assembly code.

assembly 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 machine code. We call this style of programming "goto code."

In the goto code (Figure 3.16(b)), the statement `goto x_ge_y` on line 5 causes a jump to the label `x_ge_y` (since it occurs when $x \geq y$) on line 9. Continuing the

Aside Describing machine code with C code

Figure 3.16 shows an example of how we will demonstrate the translation of C language control constructs into machine code. The figure contains an example C function (a) and an annotated version of the assembly code generated by GCC (c). It also contains a version in C that closely matches the structure of the assembly code (b). Although these versions were generated in the sequence (a), (c), and (b), we recommend that you read them in the order (a), (b), and then (c). That is, the C rendition of the machine code will help you understand the key points, and this can guide you in understanding the actual assembly code.

execution from this point, it completes the computations specified by the `else` portion of function `absdiff_se` and returns. On the other hand, if the test `x >= y` fails, the program procedure will carry out the steps specified by the `if` portion of `absdiff_se` and return.

The assembly-code implementation (Figure 3.16(c)) first compares the two operands (line 2), setting the condition codes. If the comparison result indicates that *x* is greater than or equal to *y*, it then jumps to a block of code starting at line 8 that increments global variable `ge_cnt`, computes `x-y` as the return value, and returns. Otherwise, it continues with the execution of code beginning at line 4 that increments global variable `lt_cnt`, computes `y-x` as the return value, and returns. We can see, then, that the control flow of the assembly code generated for `absdiff_se` closely follows the goto code of `gotodiff_se`.

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 zero (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 (solution page 367)

When given the C code

```
void cond(short a, short *p)
{
    if (a && *p < a)
        *p = a;
}
```

GCC generates the following assembly code:

```
  void cond(short a, short *p)
  a in %rdi, p in %rsi
cond:
  testq   %rdi, %rdi
  je      .L1
  cmpq    %rsi, (%rdi)
  jle     .L1
  movq    %rdi, (%rsi)
.L1:
  rep; ret
```

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 3.16(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 (solution page 367)

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_se` based on this alternate rule.

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

Practice Problem 3.18 (solution page 368)

Starting with C code of the form

```
short test(short x, short y, short z) {
    short val = _________;
    if (_________) {
        if (_________)
            val = _________;
        else
            val = _________;
    } else if (_________)
        val = _________;
    return val;
}
```

GCC generates the following assembly code:

```
  short test(short x, short y, short z)
  x in %rdi, y in %rsi, z in %rdx
test:
  leaq    (%rdx,%rsi), %rax
  subq    %rdi, %rax
  cmpq    $5, %rdx
  jle     .L2
  cmpq    $2, %rsi
  jle     .L3
  movq    %rdi, %rax
  idivq   %rdx, %rax
  ret
.L3:
  movq    %rdi, %rax
  idivq   %rsi, %rax
  ret
.L2:
  cmpq    $3, %rdx
  jge     .L4
  movq    %rdx, %rax
  idivq   %rsi, %rax
.L4:
  rep; ret
```

Fill in the missing expressions in the C code.

3.6.6 Implementing Conditional Branches with Conditional Moves

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. Here, we examine this strategy and its implementation with x86-64.

Figure 3.17(a) shows an example of code that can be compiled using a conditional move. The function computes the absolute value of its arguments `x` and `y`, as did our earlier example (Figure 3.16). Whereas the earlier example had side effects in the branches, modifying the value of either `lt_cnt` or `ge_cnt`, this version simply computes the value to be returned by the function.

(a) Original C code

```
long absdiff(long x, long y)
{
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}
```

(b) Implementation using conditional assignment

```
long cmovdiff(long x, long y)
{
    long rval = y-x;
    long eval = x-y;
    long ntest = x >= y;
    /* Line below requires
       single instruction: */
    if (ntest) rval = eval;
    return rval;
}
```

(c) Generated assembly code

```
long absdiff(long x, long y)
x in %rdi, y in %rsi
absdiff:
  movq    %rsi, %rax
  subq    %rdi, %rax        rval = y-x
  movq    %rdi, %rdx
  subq    %rsi, %rdx        eval = x-y
  cmpq    %rsi, %rdi        Compare x:y
  cmovge  %rdx, %rax        If >=, rval = eval
  ret                       Return tval
```

Figure 3.17 Compilation of conditional statements using conditional assignment. (a) C function `absdiff` contains a conditional expression. The generated assembly code is shown (c), along with (b) a C function `cmovdiff` that mimics the operation of the assembly code.

For this function, GCC generates the assembly code shown in Figure 3.17(c), having an approximate form shown by the C function `cmovdiff` shown in Figure 3.17(b). Studying the C version, we can see that it computes both `y-x` and `x-y`, naming these `rval` and `eval`, respectively. It then tests whether *x* is greater than or equal to *y*, and if so, copies `eval` to `rval` before returning `rval`. The assembly code in Figure 3.17(c) follows the same logic. The key is that the single `cmovge` instruction (line 7) of the assembly code implements the conditional assignment (line 8) of `cmovdiff`. It will transfer the data from the source register to the destination, only if the `cmpq` instruction of line 6 indicates that one value is greater than or equal to the other (as indicated by the suffix `ge`).

To understand why code based on conditional data transfers can outperform code based on conditional control transfers (as in Figure 3.16), 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 cannot determine which way the branch will go until it has evaluated the branch condition. 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, 15–30 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 Haswell 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 x86-64 code with conditional jumps, we found that the function requires around 8 clock cycles per call when the branching pattern is easily predictable, and around 17.50 clock cycles per call when the branching pattern is random. From this, we can infer that the branch misprediction penalty is around 19 clock cycles. That means time required by the function ranges between around 8 and 27 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 $T_{\text{avg}}(p) = (1 - p)T_{\text{OK}} + p(T_{\text{OK}} + T_{\text{MP}}) = T_{\text{OK}} + pT_{\text{MP}}$. 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 $T_{\text{ran}} = T_{\text{avg}}(0.5) = T_{\text{OK}} + 0.5T_{\text{MP}}$, and therefore $T_{\text{MP}} = 2(T_{\text{ran}} - T_{\text{OK}})$. So, for $T_{\text{OK}} = 8$ and $T_{\text{ran}} = 17.5$, we get $T_{\text{MP}} = 19$.

On the other hand, the code compiled using conditional moves requires around 8 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 3.19 (solution page 368)

Running on a new processor model, our code required around 45 cycles when the branching pattern was random, and around 25 cycles when the pattern was highly predictable.

A. What is the approximate miss penalty?

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

Figure 3.18 illustrates some of the conditional move instructions available with x86-64. 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 (Section 3.6.3) instructions, 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.

The source and destination values can be 16, 32, or 64 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 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:

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 3.18 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.

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

The standard way to compile this expression using conditional control transfer would have the following form:

```
    if (!test-expr)
        goto false;
    v = then-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 a 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:

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

The final statement in this sequence is implemented with a conditional move—value `ve` is copied to `v` only if test condition `t` does not hold.

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. Such is the case for our earlier example (Figure 3.16). Indeed, we put the side effects into this example specifically to force GCC to implement this function using conditional transfers.

As a second illustration, consider the following C function:

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

At first, this seems like a good candidate to compile using a conditional move to set the result to zero when the pointer is null, as shown in the following assembly code:

```
     long cread(long *xp)
     Invalid implementation of function cread
     xp in register %rdi
1    cread:
2      movq    (%rdi), %rax     v = *xp
3      testq   %rdi, %rdi       Test x
4      movl    $0, %edx         Set ve = 0
5      cmove   %rdx, %rax       If x==0, v = ve
6      ret                      Return v
```

This implementation is invalid, however, since the dereferencing of `xp` by the `movq` instruction (line 2) occurs even when the test fails, causing a null pointer dereferencing error. Instead, this code must be compiled using branching code.

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 3.20 (solution page 369)

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

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

short arith(short x) {
    return x OP 16;
}
```

When compiled, GCC generates the following assembly code:

```
  short arith(short x)
  x in %rdi
arith:
  leaq    15(%rdi), %rbx
  testq   %rdi, %rdi
  cmovns  %rdi, %rbx
  sarq    $4, %rbx
  ret
```

A. What operation is `OP`?

B. Annotate the code to explain how it works.

Practice Problem 3.21 (solution page 369)

Starting with C code of the form

```
short test(short x, short y) {
    short val = __________;
    if (__________) {
        if (__________)
            val = __________;
        else
            val = __________;
    } else if (__________)
        val = __________;
    return val;
}
```

GCC generates the following assembly code:

```
  short test(short x, short y)
  x in %rdi, y in %rsi
test:
  leaq    12(%rsi), %rbx
  testq   %rdi, %rdi
  jge     .L2
  movq    %rdi, %rbx
  imulq   %rsi, %rbx
  movq    %rdi, %rdx
  orq     %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge  %rdx, %rbx
  ret
.L2:
  idivq   %rsi, %rdi
  cmpq    $10, %rsi
  cmovge  %rdi, %rbx
  ret
```

Fill in the missing expressions in the C code.

3.6.7 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. Gcc and other compilers generate loop code based on the two basic loop patterns. We will study the translation of loops as a progression, starting with `do-while` and then working toward ones with more complex implementations, covering both patterns.

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, the program goes back for another iteration.

(a) C code

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Equivalent goto version

```
long fact_do_goto(long n)
{
    long result = 1;
 loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
    long fact_do(long n)
    n in %rdi
    fact_do:
      movl    $1, %eax          Set result = 1
    .L2:                      loop:
      imulq   %rdi, %rax        Compute result *= n
      subq    $1, %rdi          Decrement n
      cmpq    $1, %rdi          Compare n:1
      jg      .L2               If >, goto loop
      rep; ret                  Return
```

Figure 3.19 **Code for `do-while` version of factorial program.** A conditional jump causes the program to loop.

As an example, Figure 3.19(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 $n > 0$.

Practice Problem 3.22 (solution page 369)

A. Try to calculate 14! with a 32-bit int. Verify whether the computation of 14! overflows.

B. What if the computation is done with a 64-bit long int?

The goto code shown in Figure 3.19(b) shows how the loop gets turned into a lower-level combination of tests and conditional jumps. Following the initialization of `result`, the program begins looping. First it executes the body of the loop, consisting here of updates to variables `result` and n. It then tests whether $n > 1$, and, if so, it jumps back to the beginning of the loop. Figure 3.19(c) shows

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 3.19, 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.

the assembly code from which the goto code was generated. The conditional jump instruction `jg` (line 7) is the key instruction in implementing a loop. It determines whether to continue iterating or to exit the loop.

Reverse engineering assembly code, such as that of Figure 3.19(c), requires determining which registers are used for which program values. In this case, the mapping is fairly simple to determine: We know that *n* will be passed to the function in register `%rdi`. We can see register `%rax` getting initialized to 1 (line 2). (Recall that, although the instruction has `%eax` as its destination, it will also set the upper 4 bytes of `%rax` to 0.) We can see that this register is also updated by multiplication on line 4. Furthermore, since `%rax` is used to return the function value, it is often chosen to hold program values that are returned. We therefore conclude that `%rax` corresponds to program value `result`.

Practice Problem 3.23 (solution page 370)

For the C code

```
short dw_loop(short x) {
    short y = x/9;
    short *p = &x;
    short n = 4*x;
    do {
        x += y;
        (*p) += 5;
        n -= 2;
    } while (n > 0);
    return x;
}
```

GCC generates the following assembly code:

```
short dw_loop(short x)
x initially in %rdi
dw_loop:
  movq    %rdi, %rbx
  movq    %rdi, %rcx
  idivq   $9, %rcx
  leaq    (,%rdi,4), %rdx
.L2:
  leaq    5(%rbx,%rcx), %rcx
  subq    $1, %rdx
  testq   %rdx, %rdx
  jg      .L2
  rep; ret
```

A. Which registers are used to hold program values `x`, `y`, and `n`?

B. How has the compiler eliminated the need for pointer variable `p` and the pointer dereferencing implied by the expression `(*p)+=5`?

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.19(c).

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, two of which are used in code generated by GCC. Both use the same loop structure as we saw for `do-while` loops but differ in how to implement the initial test.

The first translation method, which we refer to as *jump to middle*, performs the initial test by performing an unconditional jump to the test at the end of the loop. It can be expressed by the following template for translating from the general `while` loop form to goto code:

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

As an example, Figure 3.20(a) shows an implementation of the factorial function using a `while` loop. This function correctly computes $0! = 1$. The adjacent

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
 loop:
    result *= n;
    n = n-1;
 test:
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
  long fact_while(long n)
  n in %rdi
fact_while:
  movl    $1, %eax        Set result = 1
  jmp     .L5             Goto test
.L6:                    loop:
  imulq   %rdi, %rax      Compute result *= n
  subq    $1, %rdi        Decrement n
.L5:                    test:
  cmpq    $1, %rdi        Compare n:1
  jg      .L6             If >, goto loop
  rep; ret                Return
```

Figure 3.20 C and assembly code for `while` version of factorial using jump-to-middle translation. The C function `fact_while_jm_goto` illustrates the operation of the assembly-code version.

function `fact_while_jm_goto` (Figure 3.20(b)) is a C rendition of the assembly code generated by GCC when optimization is specified with the command-line option `-Og`. Comparing the goto code generated for `fact_while` (Figure 3.20(b)) to that for `fact_do` (Figure 3.19(b)), we see that they are very similar, except that the statement `goto test` before the loop causes the program to first perform the test of `n` before modifying the values of `result` or `n`. The bottom portion of the figure (Figure 3.20(c)) shows the actual assembly code generated.

Practice Problem 3.24 (solution page 371)

For C code having the general form

```
short loop_while(short a, short b)
{
    short result = __________;
    while (__________) {
        result = __________;
        a = __________;
    }
    return result;
}
```

Gcc, run with command-line option `-Og`, produces the following code:

```
     short loop_while(short a, short b)
     a in %rdi, b in %rsi
1    loop_while:
2      movl    $0, %eax
3      jmp     .L2
4    .L3:
5      leaq    (,%rsi,%rdi), %rdx
6      addq    %rdx, %rax
7      subq    $1, %rdi
8    .L2:
9      cmpq    %rsi, %rdi
10     jg      .L3
11     rep; ret
```

We can see that the compiler used a jump-to-middle translation, using the `jmp` instruction on line 3 to jump to the test starting with label `.L2`. Fill in the missing parts of the C code.

The second translation method, which we refer to as *guarded do*, first transforms the code into a `do-while` loop by using a conditional branch to skip over the loop if the initial test fails. Gcc follows this strategy when compiling with higher levels of optimization, for example, with command-line option `-O1`. This method can be expressed by the following template for translating from the general `while` loop form to a `do-while` loop:

```
t = test-expr;
if (!t)
    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 3.21 shows the same C code for a factorial function as in Figure 3.20, but demonstrates the compilation that occurs when GCC is given command-line option `-O1`. Figure 3.21(c) shows the actual assembly code generated, while Figure 3.21(b) renders this assembly code in a more readable C representation. Referring to this goto code, we see that the loop will be skipped if $n \leq 1$, for the initial value of n. The loop itself has the same general structure as that generated for the `do-while` version of the function (Figure 3.19). One interesting feature, however, is that the loop test (line 9 of the assembly code) has been changed from $n > 1$ in the original C code to $n \neq 1$. The compiler has determined that the loop can only be entered when $n > 1$, and that decrementing n will result in either $n > 1$ or $n = 1$. Therefore, the test $n \neq 1$ will be equivalent to the test $n \leq 1$.

Practice Problem 3.25 (solution page 371)

For C code having the general form

```
long loop_while2(long a, long b)
{
    long result = __________;
    while (__________) {
        result = __________;
        b = __________;
    }
    return result;
}
```

GCC, run with command-line option `-O1`, produces the following code:

```
     a in %rdi, b in %rsi
     loop_while2:
       testq   %rsi, %rsi
       jle     .L8
       movq    %rsi, %rax
     .L7:
       imulq   %rdi, %rax
       subq    %rdi, %rsi
       testq   %rsi, %rsi
```

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_gd_goto(long n)
{
    long result = 1;
    if (n <= 1)
        goto done;
 loop:
    result *= n;
    n = n-1;
    if (n != 1)
        goto loop;
 done:
    return result;
}
```

(c) Corresponding assembly-language code

```
  long fact_while(long n)
  n in %rdi
fact_while:
  cmpq    $1, %rdi         Compare n:1
  jle     .L7              If <=, goto done
  movl    $1, %eax         Set result = 1
.L6:                     loop:
  imulq   %rdi, %rax       Compute result *= n
  subq    $1, %rdi         Decrement n
  cmpq    $1, %rdi         Compare n:1
  jne     .L6              If !=, goto loop
  rep; ret                 Return
.L7:                     done:
  movl    $1, %eax         Compute result = 1
  ret                      Return
```

Figure 3.21 **C and assembly code for `while` version of factorial using guarded-do translation.** The `fact_while_gd_goto` function illustrates the operation of the assembly-code version.

```
 9    jg      .L7
10    rep; ret
11  .L8:
12    movq    %rsi, %rax
13    ret
```

We can see that the compiler used a guarded-do translation, using the `jle` instruction on line 3 to skip over the loop code when the initial test fails. Fill in the missing parts of the C code. Note that the control structure in the assembly code does not exactly match what would be obtained by a direct translation of the C code according to our translation rules. In particular, it has two different `ret` instructions (lines 10 and 13). However, you can fill out the missing portions of the C code in a way that it will have equivalent behavior to the assembly code.

Practice Problem 3.26 (solution page 372)

A function `test_one` has the following overall structure:

```
short test_one(unsigned short x) {
    short val = 1;
    while ( ... ) {
        .
        .
        .
    }
    return ...;
}
```

The GCC C compiler generates the following assembly code:

```
    short test_one(unsigned short x)
    x in %rdi
test_one:
  movl    $1, %eax
  jmp     .L5
.L6:
  xorq    %rdi, %rax
  shrq    %rdi              Shift right by 1
.L5:
  testq   %rdi, %rdi
  jne     .L6
  andl    $0, %eax
  ret
```

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

A. Determine what loop translation method was used.

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

C. 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 3.29) that the behavior of such a loop is identical to the following code using 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 code generated by GCC for a `for` loop then follows one of our two translation strategies for `while` loops, depending on the optimization level. That is, the jump-to-middle strategy yields the goto code

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

while the guarded-do strategy yields

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

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

```
long fact_for(long n)
{
    long i;
    long result = 1;
    for (i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

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 to transform a `for` loop into a `while` loop yields the following:

```
long fact_for_while(long n)
{
    long i = 2;
    long result = 1;
    while (i <= n) {
        result *= i;
        i++;
    }
    return result;
}
```

Applying the jump-to-middle transformation to the `while` loop then yields the following version in goto code:

```
long fact_for_jm_goto(long n)
{
    long i = 2;
    long result = 1;
    goto test;
 loop:
    result *= i;
    i++;
 test:
    if (i <= n)
        goto loop;
    return result;
}
```

Indeed, a close examination of the assembly code produced by GCC with command-line option -Og closely follows this template:

```
  long fact_for(long n)
  n in %rdi
fact_for:
  movl    $1, %eax        Set result = 1
  movl    $2, %edx        Set i = 2
  jmp     .L8             Goto test
.L9:                    loop:
  imulq   %rdx, %rax      Compute result *= i
  addq    $1, %rdx        Increment i
.L8:                    test:
  cmpq    %rdi, %rdx      Compare i:n
  jle     .L9             If <=, goto loop
  rep; ret                Return
```

Practice Problem 3.27 (solution page 372)

Write goto code for a function called `fibonacci` to print fibonacci numbers using a `while` loop. Apply the guarded-do transformation.

We see from this presentation that all three forms of loops in C—`do-while`, `while`, and `for`—can be translated by a simple 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 3.28 (solution page 372)

A function `test_two` has the following overall structure:

```
short test_two(unsigned short x) {
    short val = 0;
    short i;
    for ( ... ; ... ; ... ) {
        .
        .
        .
    }
    return val;
}
```

The GCC C compiler generates the following assembly code:

```
   test fun_b(unsigned test x)
   x in %rdi
1  test_two:
2    movl    $1, %edx
```

```
  movl    $65, %eax
.L10:
  movq    %rdi, %rcx
  andl    $1, %ecx
  addq    %rax, %rax
  orq     %rcx, %rax
  shrq    %rdi           Shift right by 1
  addq    $1, %rdx
  jne     .L10
  rep; ret
```

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. Explain why there is neither an initial test before the loop nor an initial jump to the test portion of the loop.

C. Describe in English what this function computes.

Practice Problem 3.29 (solution page 373)

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 containing a continue statement */
/* Sum even numbers between 0 and 9 */
long sum = 0;
long 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?

3.6.8 Switch Statements

A `switch` statement provides a multiway 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, but 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 3.22(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 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 3.23 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 3.22(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 to the 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 integer. 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` (`.L5`), `loc_C` (`.L6`), `loc_D` (`.L7`), and `loc_def` (`.L8`), 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 5, where the `jmp` instruction's operand is prefixed with '`*`', indicating

(a) Switch statement

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

    case 100:
        val *= 13;
        break;

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

    case 103:
        val += 11;
        break;

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

    default:
        val = 0;
    }
    *dest = val;
}
```

(b) Translation into extended C

```
void switch_eg_impl(long x, long n,
                    long *dest)
{
    /* Table of code pointers */
    static void *jt[7] = {
        &&loc_A, &&loc_def, &&loc_B,
        &&loc_C, &&loc_D, &&loc_def,
        &&loc_D
    };
    unsigned long index = n - 100;
    long val;

    if (index > 6)
        goto loc_def;
    /* Multiway branch */
    goto *jt[index];

 loc_A:    /* Case 100 */
    val = x * 13;
    goto done;
 loc_B:    /* Case 102 */
    x = x + 10;
    /* Fall through */
 loc_C:     /* Case 103 */
    val = x + 11;
    goto done;
 loc_D:    /* Cases 104, 106 */
    val = x * x;
    goto done;
 loc_def:  /* Default case */
    val = 0;
 done:
    *dest = val;
}
```

Figure 3.22 Example `switch` statement and its 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.

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

```
    void switch_eg(long x, long n, long *dest)
    x in %rdi, n in %rsi, dest in %rdx
switch_eg:
  subq    $100, %rsi                 Compute index = n-100
  cmpq    $6, %rsi                   Compare index:6
  ja      .L8                        If >, goto loc_def
  jmp     *.L4(,%rsi,8)              Goto *jg[index]
.L3:                               loc_A:
  leaq    (%rdi,%rdi,2), %rax        3*x
  leaq    (%rdi,%rax,4), %rdi        val = 13*x
  jmp     .L2                        Goto done
.L5:                               loc_B:
  addq    $10, %rdi                  x = x + 10
.L6:                               loc_C:
  addq    $11, %rdi                  val = x + 11
  jmp     .L2                        Goto done
.L7:                               loc_D:
  imulq   %rdi, %rdi                 val = x * x
  jmp     .L2                        Goto done
.L8:                               loc_def:
  movl    $0, %edi                   val = 0
.L2:                               done:
  movq    %rdi, (%rdx)               *dest = val
  ret                                Return
```

Figure 3.23 **Assembly code for `switch` statement example in Figure 3.22.**

`index`, corresponding to values 100–106 of `n`. Observe that 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:

```
  .section        .rodata
  .align 8           Align address to multiple of 8
.L4:
  .quad   .L3        Case 100: loc_A
  .quad   .L8        Case 101: loc_def
  .quad   .L5        Case 102: loc_B
  .quad   .L6        Case 103: loc_C
  .quad   .L7        Case 104: loc_D
  .quad   .L8        Case 105: loc_def
  .quad   .L7        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 "quad" (8-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 `.L4` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (line 5).

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 `val` and then go to the end of the function. Similarly, the assembly-code blocks compute a value for register `%rdi` and jump to the position indicated by label `.L2` at the end of the function. Only the code for case label 102 does not follow this pattern, to account for the way the code for this case falls through to the block with label 103 in the original C code. This is handled in the assembly-code block starting with label `.L5`, by omitting the `jmp` instruction at the end of the block, so that the code continues execution of the next block. Similarly, the C version `switch_eg_impl` has no `goto` statement at the end of the block starting with label `loc_B`.

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 3.30 (solution page 374)

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.

```
void switch2(short x, short *dest) {
    short val = 0;
    switch (x) {
        ⋮  Body of switch statement omitted
    }
    *dest = val;
}
```

In compiling the function, GCC generates the assembly code that follows for the initial part of the procedure, with variable `x` in `%rdi`:

```
   void switch2(short x, short *dest)
   x in %rdi
1  switch2:
2    addq    $2, %rdi
3    cmpq    $8, %rdi
4    ja      .L2
5    jmp     *.L4(,%rdi,8)
```

It generates the following code for the jump table:

```
.L4:
  .quad   .L9
  .quad   .L5
  .quad   .L6
  .quad   .L7
  .quad   .L2
  .quad   .L7
  .quad   .L8
  .quad   .L2
  .quad   .L5
```

Based on this information, answer the following questions:

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

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

Practice Problem 3.31 (solution page 374)

For a C function `switcher` with the general structure

```
void switcher(long a, long b, long c, long *dest)
{
    long val;
    switch(a) {
    case _________:       /* Case A */
        c = _________;
        /* Fall through */
    case _________:       /* Case B */
        val = _________;
        break;
    case _________:       /* Case C */
    case _________:       /* Case D */
        val = _________;
        break;
    case _________:       /* Case E */
        val = _________;
        break;
    default:
        val = _________;
    }
    *dest = val;
}
```

GCC generates the assembly code and jump table shown in Figure 3.24.

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) Code

```
void switcher(long a, long b, long c, long *dest)
a in %rsi, b in %rdi, c in %rdx, d in %rcx
switcher:
  cmpq    $7, %rdi
  ja      .L2
  jmp     *.L4(,%rdi,8)
  .section        .rodata
.L7:
  xorq    $15, %rsi
  movq    %rsi, %rdx
.L3:
  leaq    112(%rdx), %rdi
  jmp     .L6
.L5:
  leaq    (%rdx,%rsi), %rdi
  salq    $2, %rdi
  jmp     .L6
.L2:
  movq    %rsi, %rdi
.L6:
  movq    %rdi, (%rcx)
  ret
```

(b) Jump table

```
.L4:
  .quad   .L3
  .quad   .L2
  .quad   .L5
  .quad   .L2
  .quad   .L6
  .quad   .L7
  .quad   .L2
  .quad   .L5
```

Figure 3.24 Assembly code and jump table for Problem 3.31.

3.7 Procedures

Procedures are a key abstraction in software. They provide a way to package code that implements some functionality with a designated set of arguments and an optional return value. This function can then be invoked from different points in a program. Well-designed software uses procedures as an abstraction mechanism, hiding the detailed implementation of some action while providing a clear and concise interface definition of what values will be computed and what effects the procedure will have on the program state. Procedures come in many guises in different programming languages—functions, methods, subroutines, handlers, and so on—but they all share a general set of features.

There are many different attributes that must be handled when providing machine-level support for procedures. For discussion purposes, suppose procedure `P` calls procedure `Q`, and `Q` then executes and returns back to `P`. These actions involve one or more of the following mechanisms:

Passing control. The program counter must be set to the starting address of the code for `Q` upon entry and then set to the instruction in `P` following the call to `Q` upon return.

Passing data. `P` must be able to provide one or more parameters to `Q`, and `Q` must be able to return a value back to `P`.

Allocating and deallocating memory. `Q` may need to allocate space for local variables when it begins and then free that storage before it returns.

The x86-64 implementation of procedures involves a combination of special instructions and a set of conventions on how to use the machine resources, such as the registers and the program memory. Great effort has been made to minimize the overhead involved in invoking a procedure. As a consequence, it follows what can be seen as a minimalist strategy, implementing only as much of the above set of mechanisms as is required for each particular procedure. In our presentation, we build up the different mechanisms step by step, first describing control, then data passing, and, finally, memory management.

3.7.1 The Run-Time Stack

A key feature of the procedure-calling mechanism of C, and of most other languages, is that it can make use of the last-in, first-out memory management discipline provided by a stack data structure. Using our example of procedure `P` calling procedure `Q`, we can see that while `Q` is executing, `P`, along with any of the procedures in the chain of calls up to `P`, is temporarily suspended. While `Q` is running, only it will need the ability to allocate new storage for its local variables or to set up a call to another procedure. On the other hand, when `Q` returns, any local storage it has allocated can be freed. Therefore, a program can manage the storage required by its procedures using a stack, where the stack and the program registers store the information required for passing control and data, and for allocating memory. As `P` calls `Q`, control and data information are added to the end of the stack. This information gets deallocated when `P` returns.

As described in Section 3.4.4, the x86-64 stack grows toward lower addresses and the stack pointer `%rsp` points to the top element of the stack. Data can be stored on and retrieved from the stack using the `pushq` and `popq` 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.

When an x86-64 procedure requires storage beyond what it can hold in registers, it allocates space on the stack. This region is referred to as the procedure's

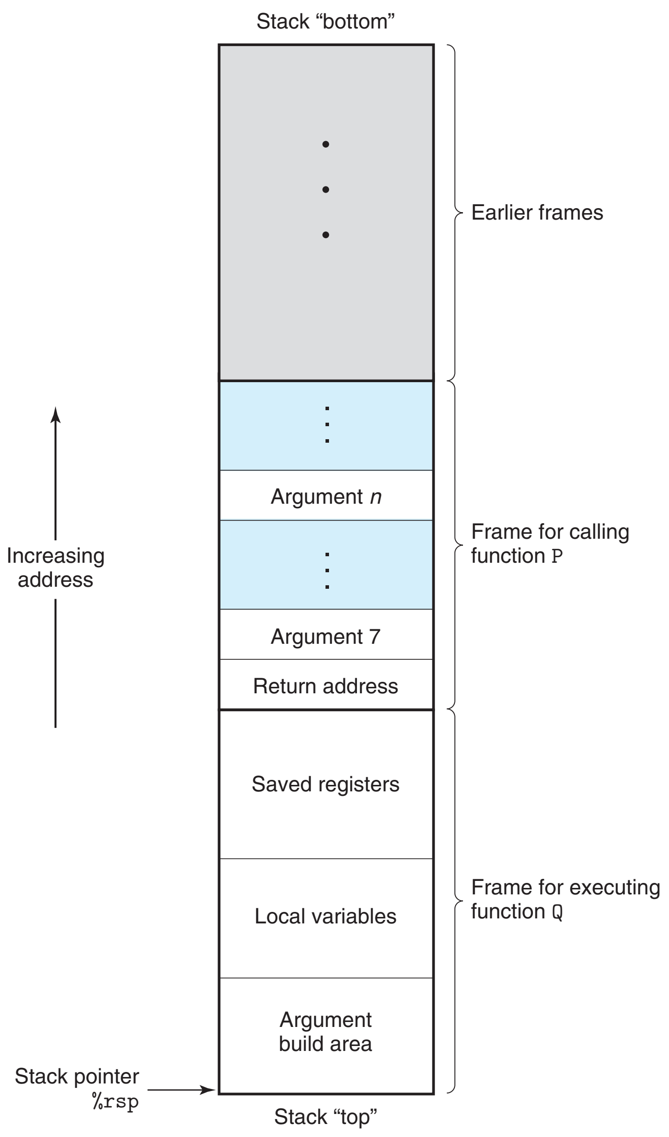


Figure 3.25 General stack frame structure. The stack can be used for passing arguments, for storing return information, for saving registers, and for local storage. Portions may be omitted when not needed.

stack frame. Figure 3.25 shows the overall structure of the run-time stack, including its partitioning into stack frames, in its most general form. The frame for the currently executing procedure is always at the top of the stack. When procedure P calls procedure Q, it will push the *return address* onto the stack, indicating where within P the program should resume execution once Q returns. We consider the return address to be part of P's stack frame, since it holds state relevant to P. The code for Q allocates the space required for its stack frame by extending the current stack boundary. Within that space, it can save the values of registers, allocate space for local variables, and set up arguments for the procedures it calls. The stack frames for most procedures are of fixed size, allocated at the beginning of the procedure. Some procedures, however, require variable-size frames. This issue is discussed in Section 3.10.5. Procedure `P` can pass up to six integral values (i.e., pointers and integers) on the stack, but if `Q` requires more arguments, these can be stored by `P` within its stack frame prior to the call.

In the interest of space and time efficiency, x86-64 procedures allocate only the portions of stack frames they require. For example, many procedures have six or fewer arguments, and so all of their parameters can be passed in registers. Thus, parts of the stack frame diagrammed in Figure 3.25 may be omitted. Indeed, many functions do not even require a stack frame. This occurs when 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). For example, none of the functions we have examined thus far required stack frames.

3.7.2 Control Transfer

Passing control from function `P` to function `Q` involves simply setting the program counter (PC) to the starting address of the code for `Q`. However, when it later comes time for `Q` to return, the processor must have some record of the code location where it should resume the execution of `P`. This information is recorded in x86-64 machines by invoking procedure `Q` with the instruction `call Q`. This instruction pushes an address *A* onto the stack and sets the PC to the beginning of `Q`. The pushed address *A* is referred to as the *return address* and is computed as the address of the instruction immediately following the `call` instruction. The counterpart instruction `ret` pops an address *A* off the stack and sets the PC to *A*.

The general forms of the `call` and `ret` instructions are described as follows:

Instruction	Description
`call` *Label*	Procedure call
`call` **Operand*	Procedure call
`ret`	Return from call

(These instructions are referred to as `callq` and `retq` in the disassembly outputs generated by the program OBJDUMP. The added suffix '`q`' simply emphasizes that these are x86-64 versions of call and return instructions, not IA32. In x86-64 assembly code, both versions can be used interchangeably.)

The `call` instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can be either 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 '`*`' followed by an operand specifier using one of the formats described in Figure 3.3.

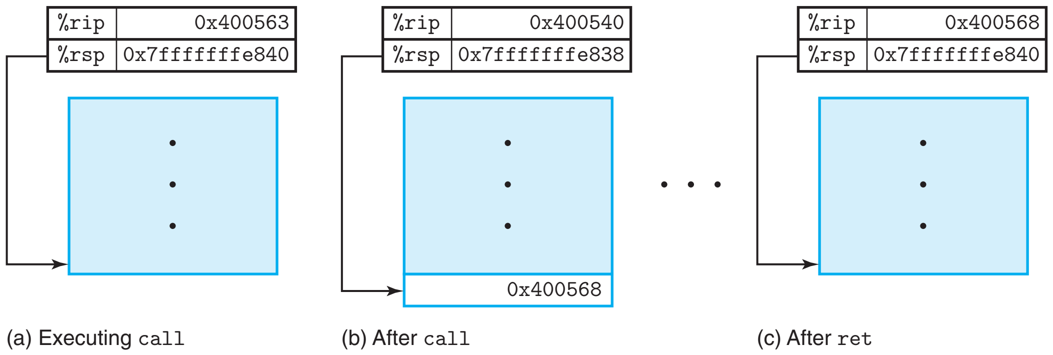


Figure 3.26 Illustration of `call` and `ret` functions. The `call` instruction transfers control to the start of a function, while the `ret` instruction returns back to the instruction following the call.

Figure 3.26 illustrates the execution of the `call` and `ret` instructions for the `multstore` and `main` functions introduced in Section 3.2.2. The following are excerpts of the disassembled code for the two functions:

```
    Beginning of function multstore
1   0000000000400540 <multstore>:
2     400540:  53                        push   %rbx
3     400541:  48 89 d3                  mov    %rdx,%rbx
    . . .
    Return from function multstore
4     40054d:  c3                        retq
    . . .
    Call to multstore from main
5     400563:  e8 d8 ff ff ff            callq  400540 <multstore>
6     400568:  48 8b 54 24 08            mov    0x8(%rsp),%rdx
```

In this code, we can see that the `call` instruction with address `0x400563` in `main` calls function `multstore`. This status is shown in Figure 3.26(a), with the indicated values for the stack pointer `%rsp` and the program counter `%rip`. The effect of the `call` is to push the return address `0x400568` onto the stack and to jump to the first instruction in function `multstore`, at address `0x0400540` (3.26(b)). The execution of function `multstore` continues until it hits the `ret` instruction at address `0x40054d`. This instruction pops the value `0x400568` from the stack and jumps to this address, resuming the execution of `main` just after the `call` instruction (3.26(c)).

As a more detailed example of passing control to and from procedures, Figure 3.27(a) shows the disassembled code for two functions, `top` and `leaf`, as well as the portion of code in function `main` where `top` gets called. Each instruction is identified by labels `L1`–`L2` (in `leaf`), `T1`–`T4` (in `top`), and `M1`–`M2` in `main`. Part (b) of the figure shows a detailed trace of the code execution, in which `main` calls `top(100)`, causing `top` to call `leaf(95)`. Function `leaf` returns 97 to `top`, which

(a) Disassembled code for demonstrating procedure calls and returns

```
    Disassembly of leaf(long y)
    y in %rdi
  0000000000400540 <leaf>:
    400540:  48 8d 47 02            lea    0x2(%rdi),%rax    L1: z+2
    400544:  c3                     retq                     L2: Return

  0000000000400545 <top>:
    Disassembly of top(long x)
    x in %rdi
    400545:  48 83 ef 05            sub    $0x5,%rdi         T1: x-5
    400549:  e8 f2 ff ff ff         callq  400540 <leaf>     T2: Call leaf(x-5)
    40054e:  48 01 c0               add    %rax,%rax         T3: Double result
    400551:  c3                     retq                     T4: Return

    . . .
      Call to top from function main
    40055b:  e8 e5 ff ff ff         callq  400545 <top>      M1: Call top(100)
    400560:  48 89 c2               mov    %rax,%rdx         M2: Resume
```

(b) Execution trace of example code

Instruction			State values (at beginning)				
Label	PC	Instruction	`%rdi`	`%rax`	`%rsp`	`*%rsp`	Description
M1	`0x40055b`	`callq`	100	—	`0x7fffffffe820`	—	Call `top(100)`
T1	`0x400545`	`sub`	100	—	`0x7fffffffe818`	`0x400560`	Entry of `top`
T2	`0x400549`	`callq`	95	—	`0x7fffffffe818`	`0x400560`	Call `leaf(95)`
L1	`0x400540`	`lea`	95	—	`0x7fffffffe810`	`0x40054e`	Entry of `leaf`
L2	`0x400544`	`retq`	—	97	`0x7fffffffe810`	`0x40054e`	Return 97 from `leaf`
T3	`0x40054e`	`add`	—	97	`0x7fffffffe818`	`0x400560`	Resume `top`
T4	`0x400551`	`retq`	—	194	`0x7fffffffe818`	`0x400560`	Return 194 from `top`
M2	`0x400560`	`mov`	—	194	`0x7fffffffe820`	—	Resume `main`

Figure 3.27 Detailed execution of program involving procedure calls and returns. Using the stack to store return addresses makes it possible to return to the right point in the procedures.

then returns 194 to `main`. The first three columns describe the instruction being executed, including the instruction label, the address, and the instruction type. The next four columns show the state of the program *before* the instruction is executed, including the contents of registers `%rdi`, `%rax`, and `%rsp`, as well as the value at the top of the stack. The contents of this table should be studied carefully, as they demonstrate the important role of the run-time stack in managing the storage needed to support procedure calls and returns.

Instruction `L1` of `leaf` sets `%rax` to 97, the value to be returned. Instruction `L2` then returns. It pops `0x400054e` from the stack. In setting the PC to this popped value, control transfers back to instruction `T3` of `top`. The program has successfully completed the call to `leaf` and returned to `top`.

Instruction `T3` sets `%rax` to 194, the value to be returned from `top`. Instruction `T4` then returns. It pops `0x4000560` from the stack, thereby setting the PC to instruction `M2` of `main`. The program has successfully completed the call to `top` and returned to `main`. We see that the stack pointer has also been restored to `0x7fffffffe820`, the value it had before the call to `top`.

We can see that this simple mechanism of pushing the return address onto the stack makes it possible for the function to later return to the proper point in the program. The standard call/return mechanism of C (and of most programming languages) conveniently matches the last-in, first-out memory management discipline provided by a stack.

Practice Problem 3.32 (solution page 375)

The disassembled code for two functions `first` and `last` is shown below, along with the code for a call of `first` by function `main`:

```
 Disassembly of last(long u, long v)
 u in %rdi, v in %rsi
0000000000400540 <last>:
  400540:  48 89 f8               mov    %rdi,%rax          L1: u
  400543:  48 0f af c6            imul   %rsi,%rax          L2: u*v
  400547:  c3                     retq                      L3: Return

 Disassembly of last(long x)
 x in %rdi
0000000000400548 <first>:
  400548:  48 8d 77 01            lea    0x1(%rdi),%rsi     F1: x+1
  40054c:  48 83 ef 01            sub    $0x1,%rdi          F2: x-1
  400550:  e8 eb ff ff ff         callq  400540 <last>      F3: Call last(x-1,x+1)
  400555:  f3 c3                  repz retq                 F4: Return
   .
   .
   .
  400560:  e8 e3 ff ff ff         callq  400548 <first>     M1: Call first(10)
  400565:  48 89 c2               mov    %rax,%rdx          M2: Resume
```

Each of these instructions is given a label, similar to those in Figure 3.27(a). Starting with the calling of `first(10)` by `main`, fill in the following table to trace instruction execution through to the point where the program returns back to `main`.

Instruction			State values (at beginning)					
Label	PC	Instruction	`%rdi`	`%rsi`	`%rax`	`%rsp`	`*%rsp`	Description
M1	`0x400560`	`callq`	10	—	—	`0x7fffffffe820`	—	Call `first(10)`
F1	______	______	______	______	______	______	______	______
F2	______	______	______	______	______	______	______	______
F3	______	______	______	______	______	______	______	______
L1	______	______	______	______	______	______	______	______
L2	______	______	______	______	______	______	______	______
L3	______	______	______	______	______	______	______	______
F4	______	______	______	______	______	______	______	______
M2	______	______	______	______	______	______	______	______

3.7.3 Data Transfer

In addition to passing control to a procedure when called, and then back again when the procedure returns, procedure calls may involve passing data as arguments, and returning from a procedure may also involve returning a value. With x86-64, most of these data passing to and from procedures take place via registers. For example, we have already seen numerous examples of functions where arguments are passed in registers `%rdi`, `%rsi`, and others, and where values are returned in register `%rax`. When procedure `P` calls procedure `Q`, the code for `P` must first copy the arguments into the proper registers. Similarly, when `Q` returns back to `P`, the code for `P` can access the returned value in register `%rax`. In this section, we explore these conventions in greater detail.

With x86-64, 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 3.28. Arguments are allocated to these registers according to their

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 3.28 **Registers for passing function arguments.** The registers are used in a specified order and named according to the argument sizes.

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`.

When a function has more than six integral arguments, the other ones are passed on the stack. Assume that procedure `P` calls procedure `Q` with n integral arguments, such that $n > 6$. Then the code for `P` must allocate a stack frame with enough storage for arguments 7 through n, as illustrated in Figure 3.25. It copies arguments 1–6 into the appropriate registers, and it puts arguments 7 through n onto the stack, with argument 7 at the top of the stack. When passing parameters on the stack, all data sizes are rounded up to be multiples of eight. With the arguments in place, the program can then execute a `call` instruction to transfer control to procedure `Q`. Procedure `Q` can access its arguments via registers and possibly from the stack. If `Q`, in turn, calls some function that has more than six arguments, it can allocate space within its stack frame for these, as is illustrated by the area labeled "Argument build area" in Figure 3.25.

As an example of argument passing, consider the C function `proc` shown in Figure 3.29(a). This function has eight arguments, including integers with different numbers of bytes (8, 4, 2, and 1), as well as different types of pointers, each of which is 8 bytes.

The assembly code generated for `proc` is shown in Figure 3.29(b). The first six arguments are passed in registers. The last two are passed on the stack, as documented by the diagram of Figure 3.30. This diagram shows the state of the stack during the execution of `proc`. We can see that the return address was pushed onto the stack as part of the procedure call. The two arguments, therefore, are at positions 8 and 16 relative to the stack pointer. Within the code, we can see that 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`). Observe that the `movl` instruction of line 6 reads 4 bytes from memory; the following `addb` instruction only makes use of the low-order byte.

Practice Problem 3.33 (solution page 375)

A C function `procprob` has four arguments `u`, `a`, `v`, and `b`. Each is either a signed number or a pointer to a signed number, where the numbers have different sizes. The function has the following body:

```
*u += a;
*v += b;
return sizeof(a) + sizeof(b);
```

It compiles to the following x86-64 code:

```
procprob:
  movslq  %edi, %rdi
  addq    %rdi, (%rdx)
  addb    %sil, (%rcx)
```

(a) C code

```
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;
}
```

(b) Generated assembly code

```
    void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
    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)
proc:
  movq    16(%rsp), %rax    Fetch a4p    (64 bits)
  addq    %rdi, (%rsi)      *a1p += a1   (64 bits)
  addl    %edx, (%rcx)      *a2p += a2   (32 bits)
  addw    %r8w, (%r9)       *a3p += a3   (16 bits)
  movl    8(%rsp), %edx     Fetch a4     ( 8 bits)
  addb    %dl, (%rax)       *a4p += a4   ( 8 bits)
  ret                       Return
```

Figure 3.29 Example of function with multiple arguments of different types. Arguments 1–6 are passed in registers, while arguments 7–8 are passed on the stack.

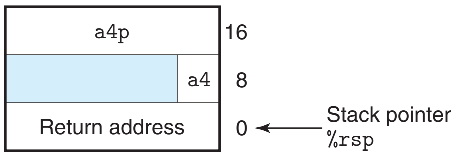


Figure 3.30
Stack frame structure for function proc. Arguments a4 and a4p are passed on the stack.

```
    movl    $6, %eax
    ret
```

Determine a valid ordering and types of the four parameters. There are two correct answers.

3.7.4 Local Storage on the Stack

Most of the procedure examples we have seen so far did not require any local storage beyond what could be held in registers. At times, however, local data must be stored in memory. Common cases of this include these:

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

Typically, a procedure allocates space on the stack frame by decrementing the stack pointer. This results in the portion of the stack frame labeled "Local variables" in Figure 3.25.

As an example of the handling of the address operator, consider the two functions shown in Figure 3.31(a). The function `swap_add` swaps the two values designated by pointers `xp` and `yp` and also returns the sum of the two values. The function `caller` creates pointers to local variables `arg1` and `arg2` and passes these to `swap_add`. Figure 3.31(b) shows how `caller` uses a stack frame to implement these local variables. The code for `caller` starts by decrementing the stack pointer by 16; this effectively allocates 16 bytes on the stack. Letting S denote the value of the stack pointer, we can see that the code computes `&arg2` as $S + 8$ (line 5), `&arg1` as S (line 6). We can therefore infer that local variables `arg1` and `arg2` are stored within the stack frame at offsets 0 and 8 relative to the stack pointer. When the call to `swap_add` completes, the code for `caller` then retrieves the two values from the stack (lines 8–9), computes their difference, and multiplies this by the value returned by `swap_add` in register `%rax` (line 10). Finally, the function deallocates its stack frame by incrementing the stack pointer by 16 (line 11.) We can see with this example that the run-time stack provides a simple mechanism for allocating local storage when it is required and deallocating it when the function completes.

As a more complex example, the function `call_proc`, shown in Figure 3.32, illustrates many aspects of the x86-64 stack discipline. Despite the length of this example, it is worth studying carefully. It shows a function that must allocate storage on the stack for local variables, as well as to pass values to the 8-argument function `proc` (Figure 3.29). The function creates a stack frame, diagrammed in Figure 3.33.

Looking at the assembly code for `call_proc` (Figure 3.32(b)), we can see that a large portion of the code (lines 2–15) involves preparing to call function

(a) Code for `swap_add` and calling function

```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

(b) Generated assembly code for calling function

```
  long caller()
  caller:
    subq    $16, %rsp          Allocate 16 bytes for stack frame
    movq    $534, (%rsp)       Store 534 in arg1
    movq    $1057, 8(%rsp)     Store 1057 in arg2
    leaq    8(%rsp), %rsi      Compute &arg2 as second argument
    movq    %rsp, %rdi         Compute &arg1 as first argument
    call    swap_add           Call swap_add(&arg1, &arg2)
    movq    (%rsp), %rdx       Get arg1
    subq    8(%rsp), %rdx      Compute diff = arg1 - arg2
    imulq   %rdx, %rax         Compute sum * diff
    addq    $16, %rsp          Deallocate stack frame
    ret                        Return
```

Figure 3.31 **Example of procedure definition and call.** The calling code must allocate a stack frame due to the presence of address operators.

`proc`. This includes setting up the stack frame for the local variables and function parameters, and for loading function arguments into registers. As Figure 3.33 shows, local variables `x1`–`x4` are allocated on the stack and have different sizes. Expressing their locations as offsets relative to the stack pointer, they occupy bytes 24–31 (`x1`), 20–23 (`x2`), 18–19 (`x3`), and 17 (`s3`). Pointers to these locations are generated by `leaq` instructions (lines 7, 10, 12, and 14). Arguments 7 (with value 4) and 8 (a pointer to the location of `x4`) are stored on the stack at offsets 0 and 8 relative to the stack pointer.

(a) C code for calling function

```
long 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);
}
```

(b) Generated assembly code

```
  long call_proc()
call_proc:
    Set up arguments to proc
    subq    $32, %rsp              Allocate 32-byte stack frame
    movq    $1, 24(%rsp)           Store 1 in &x1
    movl    $2, 20(%rsp)           Store 2 in &x2
    movw    $3, 18(%rsp)           Store 3 in &x3
    movb    $4, 17(%rsp)           Store 4 in &x4
    leaq    17(%rsp), %rax         Create &x4
    movq    %rax, 8(%rsp)          Store &x4 as argument 8
    movl    $4, (%rsp)             Store 4 as argument 7
    leaq    18(%rsp), %r9          Pass &x3 as argument 6
    movl    $3, %r8d               Pass 3 as argument 5
    leaq    20(%rsp), %rcx         Pass &x2 as argument 4
    movl    $2, %edx               Pass 2 as argument 3
    leaq    24(%rsp), %rsi         Pass &x1 as argument 2
    movl    $1, %edi               Pass 1 as argument 1
    Call proc
    call    proc
    Retrieve changes to memory
    movslq  20(%rsp), %rdx         Get x2 and convert to long
    addq    24(%rsp), %rdx         Compute x1+x2
    movswl  18(%rsp), %eax         Get x3 and convert to int
    movsbl  17(%rsp), %ecx         Get x4 and convert to int
    subl    %ecx, %eax             Compute x3-x4
    cltq                           Convert to long
    imulq   %rdx, %rax             Compute (x1+x2) * (x3-x4)
    addq    $32, %rsp              Deallocate stack frame
    ret                            Return
```

Figure 3.32 Example of code to call function `proc`**, defined in Figure 3.29.** This code creates a stack frame.

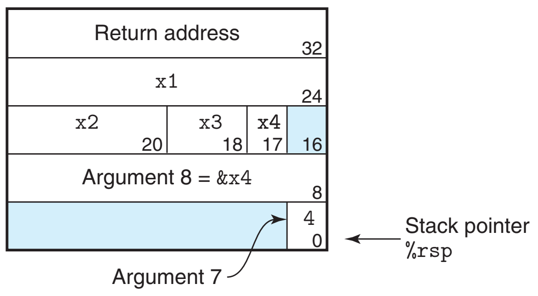


Figure 3.33
Stack frame for function `call_proc`. The stack frame contains local variables, as well as two of the arguments to pass to function `proc`.

When procedure `proc` is called, the program will begin executing the code shown in Figure 3.29(b). As shown in Figure 3.30, arguments 7 and 8 are now at offsets 8 and 16 relative to the stack pointer, because the return address was pushed onto the stack.

When the program returns to `call_proc`, the code retrieves the values of the four local variables (lines 17–20) and performs the final computations. It finishes by incrementing the stack pointer by 32 to deallocate the stack frame.

3.7.5 Local Storage in Registers

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, x86-64 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

By convention, registers `%rbx`, `%rbp`, and `%r12`–`%r15` are classified as *callee-saved* registers. When procedure `P` calls procedure `Q`, `Q` must *preserve* the values of these registers, ensuring that they have the same values when `Q` returns to `P` as they did when `Q` was called. Procedure `Q` can preserve a register value by either not changing it at all or by pushing the original value on the stack, altering it, and then popping the old value from the stack before returning. The pushing of register values has the effect of creating the portion of the stack frame labeled "Saved registers" in Figure 3.25. With this convention, the code for `P` can safely store a value in a callee-saved register (after saving the previous value on the stack, of course), call `Q`, and then use the value in the register without risk of it having been corrupted.

All other registers, except for the stack pointer `%rsp`, are classified as *caller-saved* registers. This means that they can be modified by any function. The name "caller saved" can be understood in the context of a procedure `P` having some local data in such a register and calling procedure `Q`. Since `Q` is free to alter this register, it is incumbent upon `P` (the caller) to first save the data before it makes the call.

As an example, consider the function `P` shown in Figure 3.34(a). It calls `Q` twice. During the first call, it must retain the value of `x` for use later. Similarly, during the second call, it must retain the value computed for `Q(y)`. In Figure 3.34(b),

(a) Calling function

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

(b) Generated assembly code for the calling function

```
     long P(long x, long y)
     x in %rdi, y in %rsi
1    P:
2      pushq   %rbp              Save %rbp
3      pushq   %rbx              Save %rbx
4      subq    $8, %rsp          Align stack frame
5      movq    %rdi, %rbp        Save x
6      movq    %rsi, %rdi        Move y to first argument
7      call    Q                 Call Q(y)
8      movq    %rax, %rbx        Save result
9      movq    %rbp, %rdi        Move x to first argument
10     call    Q                 Call Q(x)
11     addq    %rbx, %rax        Add saved Q(y) to Q(x)
12     addq    $8, %rsp          Deallocate last part of stack
13     popq    %rbx              Restore %rbx
14     popq    %rbp              Restore %rbp
15     ret
```

Figure 3.34 Code demonstrating use of callee-saved registers. Value `x` must be preserved during the first call, and value `Q(y)` must be preserved during the second.

we can see that the code generated by GCC uses two callee-saved registers: `%rbp` to hold `x`, and `%rbx` to hold the computed value of `Q(y)`. At the beginning of the function, it saves the values of these two registers on the stack (lines 2–3). It copies argument `x` to `%rbp` before the first call to `Q` (line 5). It copies the result of this call to `%rbx` before the second call to `Q` (line 8). At the end of the function (lines 13–14), it restores the values of the two callee-saved registers by popping them off the stack. Note how they are popped in the reverse order from how they were pushed, to account for the last-in, first-out discipline of a stack.

Practice Problem 3.34 (solution page 376)

Consider a function `P`, which generates local values, named `a0`–`a8`. It then calls function `Q` using these generated values as arguments. GCC produces the following code for the first part of `P`:

```
long P(long x)
x in %rdi
P:
  pushq   %r15
  pushq   %r14
  pushq   %r13
  pushq   %r12
  pushq   %rbp
  pushq   %rbx
  subq    $24, %rsp
  movq    %rdi, %rbx
  leaq    1(%rdi), %r15
  leaq    2(%rdi), %r14
  leaq    3(%rdi), %r13
  leaq    4(%rdi), %r12
  leaq    5(%rdi), %rbp
  leaq    6(%rdi), %rax
  movq    %rax, (%rsp)
  leaq    7(%rdi), %rdx
  movq    %rdx, 8(%rsp)
  movl    $0, %eax
  call    Q
      . . .
```

A. Identify which local values get stored in callee-saved registers.

B. Identify which local values get stored on the stack.

C. Explain why the program could not store all of the local values in callee-saved registers.

3.7.6 Recursive Procedures

The conventions we have described for using the registers and the stack allow x86-64 procedures to call themselves recursively. Each procedure call has its own private space on the stack, and so 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 before returning.

Figure 3.35 shows both the C code and the generated assembly code for a recursive factorial function. We can see that the assembly code uses register `%rbx` to hold the parameter `n`, after first saving the existing value on the stack (line 2) and later restoring the value before returning (line 11). Due to the stack discipline, and the register-saving conventions, we can be assured that when the recursive call to `rfact(n-1)` returns (line 9) that (1) the result of the call will be held in register

(a) C code

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

(b) Generated assembly code

```
    long rfact(long n)
    n in %rdi
rfact:
  pushq   %rbx                 Save %rbx
  movq    %rdi, %rbx           Store n in callee-saved register
  movl    $1, %eax             Set return value = 1
  cmpq    $1, %rdi             Compare n:1
  jle     .L35                 If <=, goto done
  leaq    -1(%rdi), %rdi       Compute n-1
  call    rfact                Call rfact(n-1)
  imulq   %rbx, %rax           Multiply result by n
.L35:                        done:
  popq    %rbx                 Restore %rbx
  ret                          Return
```

Figure 3.35 Code for recursive factorial program. The standard procedure handling mechanisms suffice for implementing recursive functions.

`%rax`, and (2) the value of argument `n` will held in register `%rbx`. Multiplying these two values then computes the desired result.

We can see from this example 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 and callee-saved 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 (e.g., when procedure `P` calls `Q`, which in turn calls `P`).

Practice Problem 3.35 (solution page 376)

For a C function having the general structure

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

GCC generates the following assembly code:

```
    long rfun(unsigned long x)
    x in %rdi
rfun:
  pushq   %rbx
  movq    %rdi, %rbx
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L2
  shrq    $2, %rdi
  call    rfun
  addq    %rbx, %rax
.L2:
  popq    %rbx
  ret
```

A. What value does `rfun` store in the callee-saved register `%rbx`?

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

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, consider a declaration of the form

T `A[`N`];`

Let us denote the starting location as $x_{\mathtt{A}}$. The declaration 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. 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];
int     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	$x_{\mathtt{A}}$	$x_{\mathtt{A}} + i$
B	8	64	$x_{\mathtt{B}}$	$x_{\mathtt{B}} + 8i$
C	4	24	$x_{\mathtt{C}}$	$x_{\mathtt{C}} + 4i$
D	8	40	$x_{\mathtt{D}}$	$x_{\mathtt{D}} + 8i$

Array `A` consists of 12 single-byte (`char`) elements. Array `C` consists of 6 integers, each requiring 4 bytes. `B` and `D` are both arrays of pointers, and hence the array elements are 8 bytes each.

The memory referencing instructions of x86-64 are designed to simplify array access. For example, suppose `E` is an array of values of type `int` and we wish to evaluate `E[i]`, where the address of `E` is stored in register `%rdx` and i is stored in register `%rcx`. Then the instruction

```
movl (%rdx,%rcx,4),%eax
```

will perform the address computation $x_{\mathtt{E}} + 4i$, 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 3.36 (solution page 377)

Consider the following declarations:

```
int    P[5];
short  Q[2];
int    **R[9];
double *S[10];
short  *T[2];
```

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
P	______	______	$x_{\mathtt{P}}$	______
Q	______	______	$x_{\mathtt{Q}}$	______
R	______	______	$x_{\mathtt{R}}$	______
S	______	______	$x_{\mathtt{S}}$	______
T	______	______	$x_{\mathtt{T}}$	______

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 $x_{\mathtt{p}}$, then the expression `p+i` has value $x_{\mathtt{p}} + L \cdot i$, 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 ith 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 `%rdx` and `%rcx`, respectively. The following are some expressions involving `E`. We also show an assembly-code implementation of each expression, with the result being stored in either register `%eax` (for data) or register `%rax` (for pointers).

Expression	Type	Value	Assembly code
`E`	`int *`	$x_{\mathtt{E}}$	`movl %rdx,%rax`
`E[0]`	`int`	$M[x_{\mathtt{E}}]$	`movl (%rdx),%eax`
`E[i]`	`int`	$M[x_{\mathtt{E}} + 4i]$	`movl (%rdx,%rcx,4),%eax`
`&E[2]`	`int *`	$x_{\mathtt{E}} + 8$	`leaq 8(%rdx),%rax`
`E+i-1`	`int *`	$x_{\mathtt{E}} + 4i - 4$	`leaq -4(%rdx,%rcx,4),%rax`
`*(E+i-3)`	`int`	$M[x_{\mathtt{E}} + 4i - 12]$	`movl -12(%rdx,%rcx,4),%eax`
`&E[i]-E`	`long`	i	`movq %rcx,%rax`

In these examples, we see that operations that return array values have type `int`, and hence involve 4-byte operations (e.g., `movl`) and registers (e.g., `%eax`). Those that return pointers have type `int *`, and hence involve 8-byte operations (e.g., `leaq`) and registers (e.g., `%rax`). The final example shows that one can compute the difference of two pointers within the same data structure, with the result being data having type `long` and value equal to the difference of the two addresses divided by the size of the data type.

Practice Problem 3.37 (solution page 377)

Suppose $x_{\mathtt{P}}$, the address of short integer array `P`, and long integer index i are stored in registers `%rdx` and `%rcx`, 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 `%rax` if it is a pointer and register element `%ax` if it has data type `short`.

Expression	Type	Value	Assembly code
`P[1]`			
`P + 3 + i`			
`P[i * 6 - 5]`			
`P[2]`			
`&P[i + 2]`			

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. This is illustrated in Figure 3.36.

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

$$\&\mathtt{D}[\mathtt{i}][\mathtt{j}] = x_{\mathtt{D}} + L(C \cdot i + j) \qquad (3.1)$$

Figure 3.36
Elements of array in row-major order.

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[4][2]`	$x_A + 44$
`A[4]`	`A[4][0]`	$x_A + 48$
	`A[4][1]`	$x_A + 52$
	`A[4][2]`	$x_A + 56$

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 in registers `%rdi`, `%rsi`, and `%rdx`, respectively. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
  A in %rdi, i in %rsi, and j in %rdx
    leaq    (%rsi,%rsi,2), %rax      Compute 3i
    leaq    (%rdi,%rax,4), %rax      Compute x_A + 12i
    movl    (%rax,%rdx,4), %eax      Read from M[x_A + 12i + 4]
```

As can be seen, this code computes the element's address as $x_A + 12i + 4j = x_A + 4(3i + j)$ using the scaling and addition capabilities of x86-64 address arithmetic.

Practice Problem 3.38 (solution page 377)

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

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

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

```
long sum_element(long i, long j)
i in %rdi, j in %rsi
sum_element:
  leaq    0(,%rdi,8), %rdx
  subq    %rdi, %rdx
  addq    %rsi, %rdx
  leaq    (%rsi,%rsi,4), %rax
  addq    %rax, %rdi
  movq    Q(,%rdi,8), %rax
  addq    P(,%rdx,8), %rax
  ret
```

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. Here we demonstrate some of the optimizations made by GCC when the optimization level is set with the flag `-O1`. Suppose we declare data type `fix_matrix` to be 16×16 arrays of integers as follows:

```
#define N 16
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 3.37(a) computes element i, k of the product of arrays `A` and `B`—that is, the inner product of row i from `A` and column k from `B`. This product is given by the formula $\sum_{0 \le j < N} a_{i,j} \cdot b_{j,k}$. GCC generates code that we then recoded into C, shown as function `fix_prod_ele_opt` in Figure 3.37(b). This code contains a number of clever optimizations. It removes the integer index `j` and converts all array references to pointer dereferences. This involves (1) generating a pointer, which we have named `Aptr`, that points to successive elements in row i of `A`, (2) generating a pointer, which we have named `Bptr`, that points to successive elements in column k of `B`, and (3) generating a pointer, which we have named `Bend`, that equals the value `Bptr` will have when it is time to terminate the loop. The initial value for `Aptr` is the address of the first element of row i of `A`, given by the C expression `&A[i][0]`. The initial value for `Bptr` is the address of the first element of column k of `B`, given by the C expression `&B[0][k]`. The value for `Bend` is the index of what would be the $(n + 1)$st element in column j of `B`, given by the C expression `&B[N][k]`.

(a) Original C code

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele (fix_matrix A, fix_matrix B, long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

(b) Optimized C code

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k) {
    int *Aptr = &A[i][0];    /* Points to elements in row i of A    */
    int *Bptr = &B[0][k];    /* Points to elements in column k of B */
    int *Bend = &B[N][k];    /* Marks stopping point for Bptr       */
    int result = 0;
    do {                          /* No need for initial test */
        result += *Aptr * *Bptr;  /* Add next product to sum  */
        Aptr ++;                  /* Move Aptr to next column */
        Bptr += N;                /* Move Bptr to next row    */
    } while (Bptr != Bend);       /* Test for stopping point  */
    return result;
}
```

Figure 3.37 Original and optimized code to compute element *i*, *k* of matrix product for fixed-length arrays. The compiler performs these optimizations automatically.

The following is the actual assembly code generated by GCC for function `fix_prod_ele`. We see that four registers are used as follows: `%eax` holds `result`, `%rdi` holds `Aptr`, `%rcx` holds `Bptr`, and `%rsi` holds `Bend`.

```
  int fix_prod_ele_opt(fix_matrix A, fix_matrix B, long i, long k)
  A in %rdi, B in %rsi, i in %rdx, k in %rcx
fix_prod_ele:
  salq    $6, %rdx                Compute 64 * i
  addq    %rdx, %rdi              Compute Aptr = xA + 64i = &A[i][0]
  leaq    (%rsi,%rcx,4), %rcx     Compute Bptr = xB + 4k = &B[0][k]
  leaq    1024(%rcx), %rsi        Compute Bend = xB + 4k + 1024 = &B[N][k]
  movl    $0, %eax                Set result = 0
.L7:                            loop:
  movl    (%rdi), %edx            Read *Aptr
  imull   (%rcx), %edx            Multiply by *Bptr
  addl    %edx, %eax              Add to result
  addq    $4, %rdi                    Increment Aptr ++
  addq    $64, %rcx                   Increment Bptr += N
  cmpq    %rsi, %rcx                  Compare Bptr:Bend
  jne     .L7                         If !=, goto loop
  rep; ret                            Return
```

Practice Problem 3.39 (solution page 378)

Use Equation 3.1 to explain how the computations of the initial values for `Aptr`, `Bptr`, and `Bend` in the C code of Figure 3.37(b) (lines 3–5) correctly describe their computations in the assembly code generated for `fix_prod_ele` (lines 3–5).

Practice Problem 3.40 (solution page 378)

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

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

When compiled with optimization level `-O1`, GCC generates the following assembly code:

```
fix_set_diag:
void fix_set_diag(fix_matrix A, int val)
A in %rdi, val in %rsi
  movl    $0, %eax
.L13:
  movl    %esi, (%rdi,%rax)
  addq    $68, %rax
  cmpq    $1088, %rax
  jne     .L13
  rep; ret
```

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 3.37(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-size arrays had to allocate storage for these arrays using functions such as `malloc` or `calloc`, and they 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 of having array dimension expressions that are computed as the array is being allocated.

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:

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

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

```
    int var_ele(long n, int A[n][n], long i, long j)
    n in %rdi, A in %rsi, i in %rdx, j in %rcx
1   var_ele:
2     imulq   %rdx, %rdi            Compute n · i
3     leaq    (%rsi,%rdi,4), %rax   Compute x_A + 4(n · i)
4     movl    (%rax,%rcx,4), %eax   Read from M[x_A + 4(n · i) + 4j]
5     ret
```

As the annotations show, this code computes the address of element i, j as $x_A + 4(n \cdot i) + 4j = x_A + 4(n \cdot i + j)$. The address computation is similar to that of the fixed-size array (Section 3.8.3), except that (1) the register usage changes due to added parameter `n`, and (2) a multiply instruction is used (line 2) to compute $n \cdot i$, rather than an `leaq` 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-size 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 3.38(a) shows C code to compute element i, k of the product of two $n \times n$ arrays `A` and `B`. Gcc generates assembly code, which we have recast into C (Figure 3.38(b)). This code follows a different style from the optimized code for the fixed-size array (Figure 3.37), but that is more an artifact of the choices made by the compiler, rather than a fundamental requirement for the two different functions. The code of Figure 3.38(b) retains loop variable `j`, both to detect when

(a) Original C code

```
/* Compute i,k of variable matrix product */
int var_prod_ele(long n, int A[n][n], int B[n][n], long i, long k) {
    long j;
    int result = 0;

    for (j = 0; j < n; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

(b) Optimized C code

```
/* Compute i,k of variable matrix product */
int var_prod_ele_opt(long n, int A[n][n], int B[n][n], long i, long k) {
    int *Arow = A[i];
    int *Bptr = &B[0][k];
    int result = 0;
    long j;
    for (j = 0; j < n; j++) {
        result += Arow[j] * *Bptr;
        Bptr += n;
    }
    return result;
}
```

Figure 3.38 Original and optimized code to compute element i, k of matrix product for variable-size arrays. The compiler performs these optimizations automatically.

the loop has terminated and to index into an array consisting of the elements of row i of A.

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

```
  Registers: n in %rdi, Arow in %rsi, Bptr in %rcx
             4n in %r9, result in %eax, j in %edx
.L24:                                        loop:
  movl    (%rsi,%rdx,4), %r8d                  Read Arow[j]
  imull   (%rcx), %r8d                         Multiply by *Bptr
  addl    %r8d, %eax                           Add to result
  addq    $1, %rdx                             j++
  addq    %r9, %rcx                            Bptr += n
  cmpq    %rdi, %rdx                           Compare j:n
  jne     .L24                                 If !=, goto loop
```

We see that the program makes use of both a scaled value $4n$ (register %r9) for incrementing Bptr as well as the value of n (register %rdi) 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.

We have seen that, with optimizations enabled, GCC is able to recognize patterns that arise when a program steps through the elements of a multidimensional array. It can then generate code that avoids the multiplication that would result from a direct application of Equation 3.1. Whether it generates the pointer-based code of Figure 3.37(b) or the array-based code of Figure 3.38(b), these optimizations will significantly improve program performance.

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.

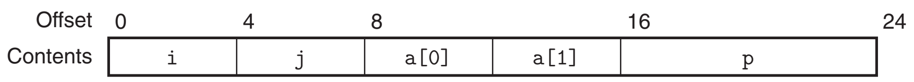
3.9.1 Structures

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.

As an example, consider the following structure declaration:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

This structure contains four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 8-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`

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 to reference that information with names.

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

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

We can 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:

```
long 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 can write a function that rotates a rectangle counterclockwise by 90 degrees as

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

New to C? Representing an object as a `struct` *(continued)*

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 previously.

of type `struct rec *` is in register `%rdi`. Then the following code copies element `r->i` to element `r->j`:

```
    Registers: r in %rdi
    movl    (%rdi), %eax        Get r->i
    movl    %eax, 4(%rdi)       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 `%rdi` and long integer variable `i` in register `%rsi`, we can generate the pointer value `&(r->a[i])` with the single instruction

```
    Registers: r in %rdi, i %rsi
    leaq    8(%rdi,%rsi,4), %rax    Set %rax 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 `%rdi`:

```
    Registers: r in %rdi
    movl    4(%rdi), %eax           Get r->j
    addl    (%rdi), %eax            Add r->i
    cltq                            Extend to 8 bytes
    leaq    8(%rdi,%rax,4), %rax    Compute &r->a[r->i + r->j]
    movq    %rax, 16(%rdi)          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 3.41 (solution page 379)

Consider the following structure declaration:

```
struct test {
    short *p;
    struct {
        short x;
        short y;
    } s;
    struct test *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 st_init(struct test *st) {
    st->s.y  = _________;
    st->p    = _________;
    st->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 `st_init`:

```
   void st_init(struct test *st)
   st in %rdi
st_init:
  movl    8(%rdi), %eax
  movl    %eax, 10(%rdi)
  leaq    10(%rdi), %rax
  movq    %rax, (%rdi)
  movq    %rdi, 12(%rdi)
  ret
```

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

Practice Problem 3.42 (solution page 379)

The following code shows the declaration of a structure of type ACE and the prototype for a function test:

```
struct ACE {
    short      v;
    struct ACE *p;
};

short test(struct ACE *ptr);
```

When the code for fun is compiled, GCC generates the following assembly code:

```
    short test(struct ACE *ptr)
    ptr in %rdi
test:
  movl    $1, %eax
  jmp     .L2
.L3:
  imulq   (%rdi), %rax
  movq    2(%rdi), %rdi
.L2:
  testq   %rdi, %rdi
  jne     .L3
  rep; ret
```

A. Use your reverse engineering skills to write C code for test.

B. Describe the data structure that this structure implements and the operation performed by test.

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 x86-64 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	16	24
U3	0	0	0	8

(We will see shortly why i has offset 4 in S3 rather than 1, and why v has offset 16, rather than 9 or 12.) 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 two double data values and 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[2];
};
```

then every node requires 32 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[2];
};
```

then every node will require just 16 bytes. If n is a pointer to a node of type union node_u *, we would reference the data of a leaf node as n->data[0] and n->data[1], 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[2];
    } info;
};
```

This structure requires a total of 24 bytes: 4 for `type`, and either 8 each for `info.internal.left` and `info.internal.right` or 16 for `info.data`. As we will discuss shortly, an additional 4 bytes of padding is required between the field for `type` and the union elements, bringing the total structure size to $4 + 4 + 16 = 24$. 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, suppose we use a simple cast to convert a value `d` of type `double` to a value `u` of type `unsigned long`:

```
    unsigned long u = (unsigned long) d;
```

Value `u` will be an integer representation of `d`. Except for the case where `d` is 0.0, the bit representation of `u` will be very different from that of `d`. Now consider the following code to generate a value of type `unsigned long` from a `double`:

```
unsigned long double2bits(double d) {
    union {
        double d;
        unsigned long u;
    } temp;
    temp.d = d;
    return temp.u;
};
```

In this code, we store the argument in the union using one data type and access it using another. The result will be that `u` will have the same bit representation as `d`, including fields for the sign bit, the exponent, and the significand, as described in

Section 3.11. The numeric value of `u` will bear no relation to that of `d`, except for the case when `d` is 0.0.

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` values:

```
double uu2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

On a little-endian machine, such as an x86-64 processor, 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 3.43 (solution page 380)

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 {
        long   u;
        short  v;
        char   w;
    } 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 `%rdi` and `%rsi`, respectively. Fill in the following table with data type *type* and sequences of one to three instructions to compute the expression and store the result at `dest`.

expr	*type*	Code

`up->t1.u`	`long`	`movq (%rdi), %rax`
		`movq %rax, (%rsi)`

`up->t1.v`		
`&up->t1.w`		
`up->t2.a`		
`up->t2.a[up->t1.u]`		
`*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 objects 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 x86-64 hardware will work correctly regardless of the alignment of data. However, Intel recommends that data be aligned to improve memory system performance. Their alignment rule is based on the principle that any primitive object of K bytes must have an address that is a multiple of K. We can see that this rule leads to the following alignments:

K	Types
1	`char`
2	`short`
4	`int`, `float`
8	`long`, `double`, `char *`

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 on page 271 contains the following directive on line 2:

```
.align 8
```

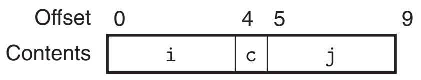
This ensures that the data following it (in this case the start of the jump table) will start with an address that is a multiple of 8. Since each table entry is 8 bytes long, the successive elements will obey the 8-byte alignment restriction.

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 will then have some required alignment for its starting address.

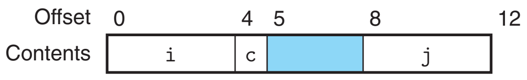
For example, consider the structure declaration

```
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 $x_{\mathtt{p}}$. Then $x_{\mathtt{p}}$ must be a multiple of 4. This guarantees that both `p->i` (address $x_{\mathtt{p}}$) and `p->j` (address $x_{\mathtt{p}} + 8$) will satisfy their 4-byte alignment requirements.

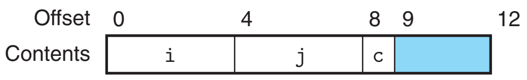
In addition, the compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement. For example, consider the following structure declaration:

```
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 $x_{\mathtt{d}}$, $x_{\mathtt{d}} + 9$, $x_{\mathtt{d}} + 18$, and $x_{\mathtt{d}} + 27$. 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 $x_{\mathtt{d}}$, $x_{\mathtt{d}} + 12$, $x_{\mathtt{d}} + 24$, and $x_{\mathtt{d}} + 36$. As long as $x_{\mathtt{d}}$ is a multiple of 4, all of the alignment restrictions will be satisfied.

Practice Problem 3.44 (solution page 381)

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64:

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

B. `struct P2 { int i[2]; char c[8]; short s[4]; long *j; };`

C. `struct P3 { long w[2]; int *c[2] };`

D. `struct P4 { char w[16]; char *c[2] };`

E. `struct P5 { struct P4 a[2]; struct P1 t };`

Practice Problem 3.45 (solution page 381)

Answer the following for the structure declaration

```
struct {
  int    *a;
  float  b;
  char   c;
  short  d;
  long   e;
  double f;
```

Aside A case of mandatory alignment

For most x86-64 instructions, keeping data aligned improves efficiency, but it does not affect program behavior. On the other hand, some models of Intel and AMD processors will not work correctly with unaligned data for some of the SSE instructions implementing multimedia operations. 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* (see Section 8.1), with the default behavior for the program to terminate.

As a result, any compiler and run-time system for an x86-64 processor must ensure that any memory allocated to hold a data structure that may be read from or stored into an SSE register must satisfy a 16-byte alignment. This requirement has the following two consequences:

- The starting address for any block generated by a memory allocation function (`alloca`, `malloc`, `calloc`, or `realloc`) must be a multiple of 16.
- The stack frame for most functions must be aligned on a 16-byte boundary. (This requirement has a number of exceptions.)

More recent versions of x86-64 processors implement the AVX multimedia instructions. In addition to providing a superset of the SSE instructions, processors supporting AVX also do not have a mandatory alignment requirement.

```
    int     g;
    char    *h;
} rec;
```

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 Combining Control and Data in Machine-Level Programs

So far, we have looked separately at how machine-level code implements the control aspects of a program and how it implements different data structures. In this section, we look at ways in which data and control interact with each other. We start by taking a deep look into pointers, one of the most important concepts in the C programming language, but one for which many programmers only have a shallow understanding. We review the use of the symbolic debugger GDB for examining the detailed operation of machine-level programs. Next, we see how understanding machine-level programs enables us to study buffer overflow, an important security vulnerability in many real-world systems. Finally, we examine how machine-level programs implement cases where the amount of stack storage required by a function can vary from one execution to another.

3.10.1 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 `leaq` 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 and Ritchie [61, Sect. 5.12] present a helpful tutorial on reading C declarations.

3.10.2 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 3.39 shows examples of some GDB commands that help when working with machine-level x86-64 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 211. We start GDB with the following command line:

```
linux> 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 graphical user interface.

3.10.3 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:

```
/* Implementation of library function gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;
```

Command	Effect
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command-line arguments here)
`kill`	Stop your program
Breakpoints	
`break multstore`	Set breakpoint at entry to function `multstore`
`break *0x400540`	Set breakpoint at address `0x400540`
`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 multstore`	Disassemble function `multstore`
`disas 0x400544`	Disassemble function around address `0x400544`
`disas 0x400540, 0x40054d`	Disassemble code within specified address range
`print /x $rip`	Print program counter in hex
Examining data	
`print $rax`	Print contents of `%rax` in decimal
`print /x $rax`	Print contents of `%rax` in hex
`print /t $rax`	Print contents of `%rax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($rsp+8)`	Print contents of `%rsp` plus 8 in hex
`print *(long *) 0x7fffffffe818`	Print long integer at address `0x7fffffffe818`
`print *(long *) ($rsp+8)`	Print long integer at address `%rsp` + 8
`x/2g 0x7fffffffe818`	Examine two (8-byte) words starting at address `0x7fffffffe818`
`x/20b multstore`	Examine first 20 bytes of function `multstore`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.39 **Example GDB commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

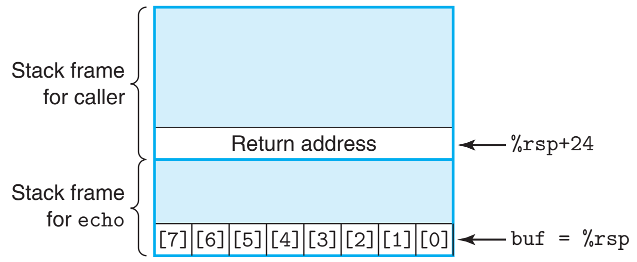


Figure 3.40
Stack organization for `echo` function. Character array `buf` is just part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

```
    while ((c = getchar()) != '\n' && c != EOF)
        *dest++ = c;
    if (c == EOF && dest == s)
        /* No characters read */
        return NULL;
    *dest++ = '\0'; /* Terminate string */
    return s;
}

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

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 echos 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.

By examining the assembly code generated by GCC for `echo`, we can infer how the stack is organized:

```
  void echo()
1 echo:
2   subq    $24, %rsp         Allocate 24 bytes on stack
3   movq    %rsp, %rdi        Compute buf as %rsp
4   call    gets              Call gets
5   movq    %rsp, %rdi        Compute buf as %rsp
```

```
6    call    puts          Call puts
7    addq    $24, %rsp     Deallocate stack space
8    ret                   Return
```

Figure 3.40 illustrates the stack organization during the execution of `echo`. The program allocates 24 bytes on the stack by subtracting 24 from the stack pointer (line 2). Character `buf` is positioned at the top of the stack, as can be seen by the fact that `%rsp` is copied to `%rdi` to be used as the argument to the calls to both `gets` and `puts`. The 16 bytes between `buf` and the stored return pointer are not used. 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 stored on the stack. As the string gets longer, the following information will get corrupted:

Characters typed	Additional corrupted state
0–7	None
9–23	Unused stack space
24–31	Return address
32+	Saved state in caller

No serious consequence occurs for strings of up to 23 characters, but beyond that, the value of the return pointer, and possibly additional saved state, will be corrupted. If the stored value of the return address is corrupted, then the `ret` instruction (line 8) 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 3.71 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. 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 [97]. Such conditions can lead to vulnerabilities to buffer overflow.

Practice Problem 3.46 (solution page 382)

Figure 3.41 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 `get_line` is called with the return address equal to `0x400776` and register `%rbx` equal to `0x0123456789ABCDEF`. You type in the string

```
0123456789012345678901234
```

(a) C code

```
/* This is very low-quality code.
   It is intended to illustrate bad programming practices.
   See Practice Problem 3.46. */
char *get_line()
{
    char buf[4];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

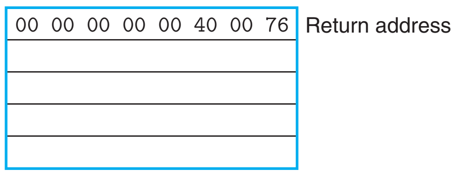
(b) Disassembly up through call to `gets`

```
     char *get_line()
1    0000000000400720 <get_line>:
2      400720:  53                       push   %rbx
3      400721:  48 83 ec 10              sub    $0x10,%rsp
     Diagram stack at this point
4      400725:  48 89 e7                 mov    %rsp,%rdi
5      400728:  e8 73 ff ff ff           callq  4006a0 <gets>
     Modify diagram to show stack contents at this point
```

Figure 3.41 **C and disassembled code for Practice Problem 3.46.**

The program terminates with a segmentation fault. You run GDB and determine that the error occurs during the execution of the `ret` instruction of `get_line`.

A. Fill in the diagram that follows, indicating as much as you can about the stack just after executing the instruction at line 3 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 8 bytes. Indicate the position of `%rsp`. Recall that the ASCII codes for characters 0–9 are 0x30–0x39.



B. Modify your diagram to show the effect of the call to `gets` (line 5).

C. To what address does the program attempt to return?

D. What register(s) have corrupted value(s) when `get_line` returns?

E. Besides the potential for buffer overflow, what two other things are wrong with the code for `get_line`?

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 "bulletproof" so that no behavior by an external agent can cause the system to misbehave.

3.10.4 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 mechanisms 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

Aside Worms and viruses

Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [105], 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.”

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* [96].

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:

```
int main() {
    long local;
    printf("local at %p\n", &local);
    return 0;
}
```

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 `0xff7fc59c` to `0xffffd09c`, a range of around 2^{23}. Running in 64-bit mode on the newer machine, the addresses ranged from `0x7fff0001b698` to `0x7ffffffaa4a8`, 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 [99]. 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" [97], 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 3.47 **(solution page 383)**

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, about how many attempts would it take to test all starting addresses?

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 3.40) that the corruption typically occurs when the program overruns the bounds of a local buffer. In C, there is no reliable way to prevent writing beyond the bounds of an array. Instead, the program can attempt to detect when such a write has occurred before it can have any harmful effects.

Recent versions of GCC incorporate a mechanism known as a *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 3.42 [26, 97]. This canary value, also referred to as a *guard value*, is generated randomly each time the program runs, and so there is no

4. The term "canary" refers to the historic use of these birds to detect the presence of dangerous gases in coal mines.

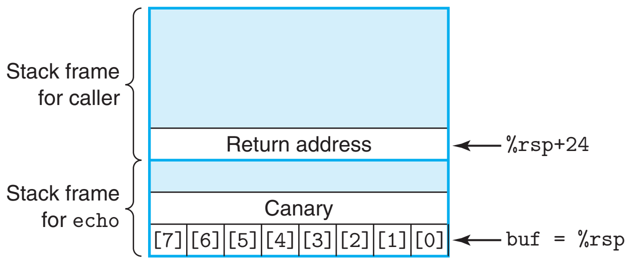


Figure 3.42 Stack organization for `echo` function with stack protector enabled. A special "canary" value is positioned between array `buf` and the saved state. The code checks the canary value to determine whether or not the stack state has been corrupted.

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. Compiling the function `echo` without this option, and hence with the stack protector enabled, gives the following assembly code:

```
     void echo()
1    echo:
2      subq    $24, %rsp               Allocate 24 bytes on stack
3      movq    %fs:40, %rax            Retrieve canary
4      movq    %rax, 8(%rsp)           Store on stack
5      xorl    %eax, %eax              Zero out register
6      movq    %rsp, %rdi              Compute buf as %rsp
7      call    gets                    Call gets
8      movq    %rsp, %rdi              Compute buf as %rsp
9      call    puts                    Call puts
10     movq    8(%rsp), %rax           Retrieve canary
11     xorq    %fs:40, %rax            Compare to stored value
12     je      .L9                     If =, goto ok
13     call    __stack_chk_fail        Stack corrupted!
14   .L9:                            ok:
15     addq    $24, %rsp               Deallocate stack space
16     ret
```

We see that this version of the function retrieves a value from memory (line 3) and stores it on the stack at offset 8 from `%rsp`, just beyond the region allocated for `buf`. The instruction argument `%fs:40` 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 `xorq` instruction on line 11). If the two are identical, the `xorq` instruction will yield zero, 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 3.48 (solution page 383)

The functions `intlen`, `len`, and `iptoa` provide a very convoluted way to compute 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, long *p) {
    long val = *p;
    sprintf(s, "%ld", val);
}

int intlen(long x) {
    long 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:

(a) Without protector

```
    int intlen(long x)
    x in %rdi
    intlen:
      subq    $40, %rsp
      movq    %rdi, 24(%rsp)
```

```
  leaq    24(%rsp), %rsi
  movq    %rsp, %rdi
  call    iptoa
```

(b) With protector

```
int intlen(long x)
x in %rdi
intlen:
  subq    $56, %rsp
  movq    %fs:40, %rax
  movq    %rax, 40(%rsp)
  xorl    %eax, %eax
  movq    %rdi, 8(%rsp)
  leaq    8(%rsp), %rsi
  leaq    16(%rsp), %rdi
  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 does 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 2,048 or 4,096 bytes per page. The hardware supports different forms of *memory protection*, indicating the forms of access allowed by both user programs and 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 the run-time system 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 [85, 97], and so worms and viruses continue to compromise the integrity of many machines.

3.10.5 Supporting Variable-Size Stack Frames

We have examined the machine-level code for a variety of functions so far, but they all have the property that the compiler can determine in advance the amount of space that must be allocated for their stack frames. Some functions, however, require a variable amount of local storage. This can occur, for example, when the function calls `alloca`, a standard library function that can allocate an arbitrary number of bytes of storage on the stack. It can also occur when the code declares a local array of variable size.

Although the information presented in this section should rightfully be considered an aspect of how procedures are implemented, we have deferred the presentation to this point, since it requires an understanding of arrays and alignment.

The code of Figure 3.43(a) gives an example of a function containing a variable-size array. The function declares local array `p` of n pointers, where n is given by the first argument. This requires allocating $8n$ bytes on the stack, where the value of n may vary from one call of the function to another. The compiler therefore cannot determine how much space it must allocate for the function's stack frame. In addition, the program generates a reference to the address of local variable `i`, and so this variable must also be stored on the stack. During execution, the program must be able to access both local variable `i` and the elements of array `p`. On returning, the function must deallocate the stack frame and set the stack pointer to the position of the stored return address.

To manage a variable-size stack frame, x86-64 code uses register `%rbp` to serve as a *frame pointer* (sometimes referred to as a *base pointer*, and hence the letters `bp` in `%rbp`). When using a frame pointer, the stack frame is organized as shown for the case of function `vframe` in Figure 3.44. We see that the code must save the previous version of `%rbp` on the stack, since it is a callee-saved register. It then keeps `%rbp` pointing to this position throughout the execution of the function, and it references fixed-length local variables, such as `i`, at offsets relative to `%rbp`.

(a) C code

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

(b) Portions of generated assembly code

```
  long vframe(long n, long idx, long *q)
  n in %rdi, idx in %rsi, q in %rdx
  Only portions of code shown
vframe:
  pushq   %rbp                    Save old %rbp
  movq    %rsp, %rbp              Set frame pointer
  subq    $16, %rsp               Allocate space for i (%rsp = s1)
  leaq    22(,%rdi,8), %rax
  andq    $-16, %rax
  subq    %rax, %rsp              Allocate space for array p (%rsp = s2)
  leaq    7(%rsp), %rax
  shrq    $3, %rax
  leaq    0(,%rax,8), %r8         Set %r8 to &p[0]
  movq    %r8, %rcx               Set %rcx to &p[0] (%rcx = p)
      . . .
  Code for initialization loop
  i in %rax and on stack, n in %rdi, p in %rcx, q in %rdx
.L3:                            loop:
  movq    %rdx, (%rcx,%rax,8)     Set p[i] to q
  addq    $1, %rax                Increment i
  movq    %rax, -8(%rbp)          Store on stack
.L2:
  movq    -8(%rbp), %rax          Retrieve i from stack
  cmpq    %rdi, %rax              Compare i:n
  jl      .L3                     If <, goto loop
      . . .
  Code for function exit
  leave                            Restore %rbp and %rsp
  ret                              Return
```

Figure 3.43 **Function requiring the use of a frame pointer.** The variable-size array implies that the size of the stack frame cannot be determined at compile time.

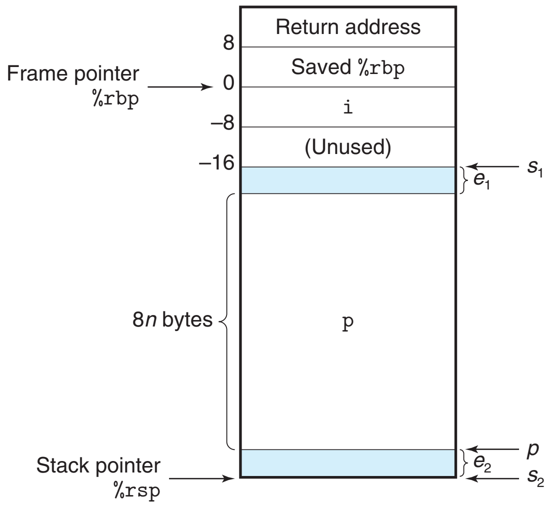


Figure 3.44
Stack frame structure for function `vframe`. The function uses register `%rbp` as a frame pointer. The annotations along the right-hand side are in reference to Practice Problem 3.49.

Figure 3.43(b) shows portions of the code GCC generates for function `vframe`. At the beginning of the function, we see code that sets up the stack frame and allocates space for array `p`. The code starts by pushing the current value of `%rbp` onto the stack and setting `%rbp` to point to this stack position (lines 2–3). Next, it allocates 16 bytes on the stack, the first 8 of which are used to store local variable `i`, and the second 8 of which are unused. Then it allocates space for array `p` (lines 5–11). The details of how much space it allocates and where it positions `p` within this space are explored in Practice Problem 3.49. Suffice it to say that by the time the program reaches line 11, it has (1) allocated at least $8n$ bytes on the stack and (2) positioned array `p` within the allocated region such that at least $8n$ bytes are available for its use.

The code for the initialization loop shows examples of how local variables `i` and `p` are referenced. Line 13 shows array element `p[i]` being set to `q`. This instruction uses the value in register `%rcx` as the address for the start of `p`. We can see instances where local variable `i` is updated (line 15) and read (line 17). The address of `i` is given by reference `-8(%rbp)`—that is, at offset −8 relative to the frame pointer.

At the end of the function, the frame pointer is restored to its previous value using the `leave` instruction (line 20). This instruction takes no arguments. It is equivalent to executing the following two instructions:

```
movq %rbp, %rsp    Set stack pointer to beginning of frame
popq %rbp          Restore saved %rbp and set stack ptr
                     to end of caller's frame
```

That is, the stack pointer is first set to the position of the saved value of `%rbp`, and then this value is popped from the stack into `%rbp`. This instruction combination has the effect of deallocating the entire stack frame.

In earlier versions of x86 code, the frame pointer was used with every function call. With x86-64 code, it is used only in cases where the stack frame may be of variable size, as is the case for function `vframe`. Historically, most compilers used frame pointers when generating IA32 code. Recent versions of GCC have dropped this convention. Observe that it is acceptable to mix code that uses frame pointers with code that does not, as long as all functions treat `%rbp` as a callee-saved register.

Practice Problem 3.49 (solution page 383)

In this problem, we will explore the logic behind the code in lines 5–11 of Figure 3.43(b), where space is allocated for variable-size array `p`. As the annotations of the code indicate, let us let s_1 denote the address of the stack pointer after executing the `subq` instruction of line 4. This instruction allocates the space for local variable `i`. Let s_2 denote the value of the stack pointer after executing the `subq` instruction of line 7. This instruction allocates the storage for local array `p`. Finally, let p denote the value assigned to registers `%r8` and `%rcx` in the instructions of lines 10–11. Both of these registers are used to reference array `p`.

The right-hand side of Figure 3.44 diagrams the positions of the locations indicated by s_1, s_2, and p. It also shows that there may be an offset of e_2 bytes between the values of s_1 and p. This space will not be used. There may also be an offset of e_1 bytes between the end of array `p` and the position indicated by s_1.

A. Explain, in mathematical terms, the logic in the computation of s_2 on lines 5–7. *Hint:* Think about the bit-level representation of −16 and its effect in the `andq` instruction of line 6.

B. Explain, in mathematical terms, the logic in the computation of p on lines 8–10. *Hint:* You may want to refer to the discussion on division by powers of 2 in Section 2.3.7.

C. For the following values of n and s_1, trace the execution of the code to determine what the resulting values would be for s_2, p, e_1, and e_2.

n	s_1	s_2	p	e_1	e_2
5	2,065				
6	2,064				

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

3.11 Floating-Point Code

The *floating-point architecture* for a processor consists of the different aspects that affect how programs operating on floating-point data are mapped onto the machine, including

- How floating-point values are stored and accessed. This is typically via some form of registers.

- The instructions that operate on floating-point data.
- The conventions used for passing floating-point values as arguments to functions and for returning them as results.
- The conventions for how registers are preserved during function calls—for example, with some registers designated as caller saved, and others as callee saved.

To understand the x86-64 floating-point architecture, it is helpful to have a brief historical perspective. Since the introduction of the Pentium/MMX in 1997, both Intel and AMD have incorporated successive generations of *media* instructions to support graphics and image processing. These instructions originally focused on allowing multiple operations to be performed in a parallel mode known as *single instruction, multiple data,* or *SIMD* (pronounced sim-dee). In this mode the same operation is performed on a number of different data values in parallel. Over the years, there has been a progression of these extensions. The names have changed through a series of major revisions from MMX to SSE (for "streaming SIMD extensions") and most recently AVX (for "advanced vector extensions"). Within each generation, there have also been different versions. Each of these extensions manages data in sets of registers, referred to as "MM" registers for MMX, "XMM" for SSE, and "YMM" for AVX, ranging from 64 bits for MM registers, to 128 for XMM, to 256 for YMM. So, for example, each YMM register can hold eight 32-bit values, or four 64-bit values, where these values can be either integer or floating point.

Starting with SSE2, introduced with the Pentium 4 in 2000, the media instructions have included ones to operate on *scalar* floating-point data, using single values in the low-order 32 or 64 bits of XMM or YMM registers. This scalar mode provides a set of registers and instructions that are more typical of the way other processors support floating point. All processors capable of executing x86-64 code support SSE2 or higher, and hence x86-64 floating point is based on SSE or AVX, including conventions for passing procedure arguments and return values [77].

Our presentation is based on AVX2, the second version of AVX, introduced with the Core i7 Haswell processor in 2013. GCC will generate AVX2 code when given the command-line parameter `-mavx2`. Code based on the different versions of SSE, as well as the first version of AVX, is conceptually similar, although they differ in the instruction names and formats. We present only instructions that arise in compiling floating-point programs with GCC. These are, for the most part, the scalar AVX instructions, although we document occasions where instructions intended for operating on entire data vectors arise. A more complete coverage of how to exploit the SIMD capabilities of SSE and AVX is presented in Web Aside OPT:SIMD on page 582. Readers may wish to refer to the AMD and Intel documentation for the individual instructions [4, 51]. As with integer operations, note that the ATT format we use in our presentation differs from the Intel format used in these documents. In particular, the instruction operands are listed in a different order in these two versions.

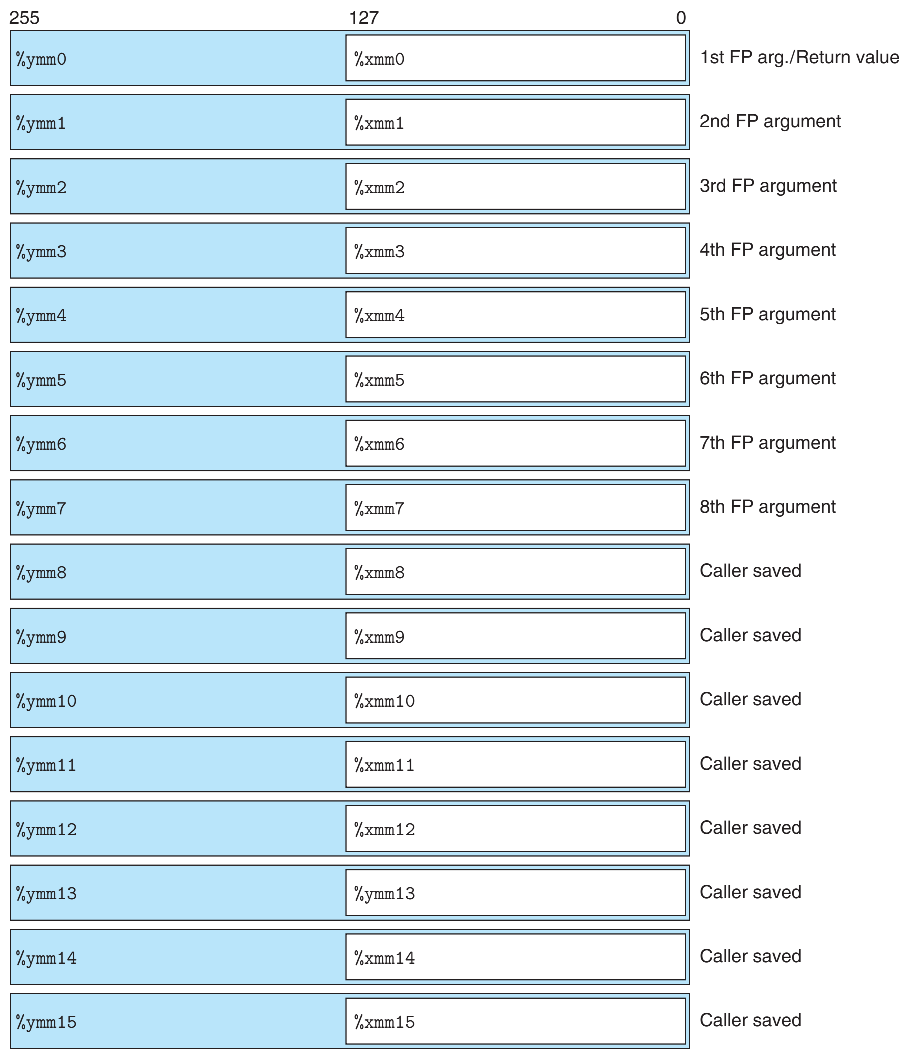


Figure 3.45 **Media registers.** These registers are used to hold floating-point data. Each YMM register holds 32 bytes. The low-order 16 bytes can be accessed as an XMM register.

As is illustrated in Figure 3.45, the AVX floating-point architecture allows data to be stored in 16 YMM registers, named `%ymm0`–`%ymm15`. Each YMM register is 256 bits (32 bytes) long. When operating on scalar data, these registers only hold floating-point data, and only the low-order 32 bits (for `float`) or 64 bits (for `double`) are used. The assembly code refers to the registers by their SSE XMM register names `%xmm0`–`%xmm15`, where each XMM register is the low-order 128 bits (16 bytes) of the corresponding YMM register.

Instruction	Source	Destination	Description
`vmovss`	M_{32}	X	Move single precision
`vmovss`	X	M_{32}	Move single precision
`vmovsd`	M_{64}	X	Move double precision
`vmovsd`	X	M_{64}	Move double precision
`vmovaps`	X	X	Move aligned, packed single precision
`vmovapd`	X	X	Move aligned, packed double precision

Figure 3.46 Floating-point movement instructions. These operations transfer values between memory and registers, as well as between pairs of registers. (X: XMM register (e.g., `%xmm3`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

3.11.1 Floating-Point Movement and Conversion Operations

Figure 3.46 shows a set of instructions for transferring floating-point data between memory and XMM registers, as well as from one XMM register to another without any conversions. Those that reference memory are *scalar* instructions, meaning that they operate on individual, rather than packed, data values. The data are held either in memory (indicated in the table as M_{32} and M_{64}) or in XMM registers (shown in the table as X). These instructions will work correctly regardless of the alignment of data, although the code optimization guidelines recommend that 32-bit memory data satisfy a 4-byte alignment and that 64-bit data satisfy an 8-byte alignment. Memory references are specified in the same way as for the integer MOV instructions, with all of the different possible combinations of displacement, base register, index register, and scaling factor.

Gcc uses the scalar movement operations only to transfer data from memory to an XMM register or from an XMM register to memory. For transferring data between two XMM registers, it uses one of two different instructions for copying the entire contents of one XMM register to another—namely, `vmovaps` for single-precision and `vmovapd` for double-precision values. For these cases, whether the program copies the entire register or just the low-order value affects neither the program functionality nor the execution speed, and so using these instructions rather than ones specific to scalar data makes no real difference. The letter ‘a’ in these instruction names stands for “aligned.” When used to read and write memory, they will cause an exception if the address does not satisfy a 16-byte alignment. For transferring between two registers, there is no possibility of an incorrect alignment.

As an example of the different floating-point move operations, consider the C function

```
float float_mov(float v1, float *src, float *dst) {
    float v2 = *src;
    *dst = v1;
    return v2;
}
```

Instruction	Source	Destination	Description
`vcvttss2si`	X/M_{32}	R_{32}	Convert with truncation single precision to integer
`vcvttsd2si`	X/M_{64}	R_{32}	Convert with truncation double precision to integer
`vcvttss2siq`	X/M_{32}	R_{64}	Convert with truncation single precision to quad word integer
`vcvttsd2siq`	X/M_{64}	R_{64}	Convert with truncation double precision to quad word integer

Figure 3.47 Two-operand floating-point conversion operations. These convert floating-point data to integers. (X: XMM register (e.g., `%xmm3`); R_{32}: 32-bit general-purpose register (e.g., `%eax`); R_{64}: 64-bit general-purpose register (e.g., `%rax`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

Instruction	Source 1	Source 2	Destination	Description
`vcvtsi2ss`	M_{32}/R_{32}	X	X	Convert integer to single precision
`vcvtsi2sd`	M_{32}/R_{32}	X	X	Convert integer to double precision
`vcvtsi2ssq`	M_{64}/R_{64}	X	X	Convert quad word integer to single precision
`vcvtsi2sdq`	M_{64}/R_{64}	X	X	Convert quad word integer to double precision

Figure 3.48 Three-operand floating-point conversion operations. These instructions convert from the data type of the first source to the data type of the destination. The second source value has no effect on the low-order bytes of the result. (X: XMM register (e.g., `%xmm3`); M_{32}: 32-bit memory range; M_{64}: 64-bit memory range)

and its associated x86-64 assembly code

```
   float float_mov(float v1, float *src, float *dst)
   v1 in %xmm0, src in %rdi, dst in %rsi
1  float_mov:
2    vmovaps %xmm0, %xmm1      Copy v1
3    vmovss  (%rdi), %xmm0     Read v2 from src
4    vmovss  %xmm1, (%rsi)     Write v1 to dst
5    ret                       Return v2 in %xmm0
```

We can see in this example the use of the `vmovaps` instruction to copy data from one register to another and the use of the `vmovss` instruction to copy data from memory to an XMM register and from an XMM register to memory.

Figures 3.47 and 3.48 show sets of instructions for converting between floating-point and integer data types, as well as between different floating-point formats. These are all scalar instructions operating on individual data values. Those in Figure 3.47 convert from a floating-point value read from either an XMM register or memory and write the result to a general-purpose register (e.g., `%rax`, `%ebx`, etc.). When converting floating-point values to integers, they perform *truncation*, rounding values toward zero, as is required by C and most other programming languages.

The instructions in Figure 3.48 convert from integer to floating point. They use an unusual three-operand format, with two sources and a destination. The first operand is read from memory or from a general-purpose register. For our purposes, we can ignore the second operand, since its value only affects the upper bytes of the result. The destination must be an XMM register. In common usage, both the second source and the destination operands are identical, as in the instruction

```
vcvtsi2sdq   %rax, %xmm1, %xmm1
```

This instruction reads a long integer from register `%rax`, converts it to data type `double`, and stores the result in the lower bytes of XMM register `%xmm1`.

Finally, for converting between two different floating-point formats, current versions of GCC generate code that requires separate documentation. Suppose the low-order 4 bytes of `%xmm0` hold a single-precision value; then it would seem straightforward to use the instruction

```
vcvtss2sd   %xmm0, %xmm0, %xmm0
```

to convert this to a double-precision value and store the result in the lower 8 bytes of register `%xmm0`. Instead, we find the following code generated by GCC:

```
     Conversion from single to double precision
1    vunpcklps  %xmm0, %xmm0, %xmm0   Replicate first vector element
2    vcvtps2pd  %xmm0, %xmm0          Convert two vector elements to double
```

The `vunpcklps` instruction is normally used to interleave the values in two XMM registers and store them in a third. That is, if one source register contains words $[s_3, s_2, s_1, s_0]$ and the other contains words $[d_3, d_2, d_1, d_0]$, then the value of the destination register will be $[s_1, d_1, s_0, d_0]$. In the code above, we see the same register being used for all three operands, and so if the original register held values $[x_3, x_2, x_1, x_0]$, then the instruction will update the register to hold values $[x_1, x_1, x_0, x_0]$. The `vcvtps2pd` instruction expands the two low-order single-precision values in the source XMM register to be the two double-precision values in the destination XMM register. Applying this to the result of the preceding `vunpcklps` instruction would give values $[dx_0, dx_0]$, where dx_0 is the result of converting x to double precision. That is, the net effect of the two instructions is to convert the original single-precision value in the low-order 4 bytes of `%xmm0` to double precision and store two copies of it in `%xmm0`. It is unclear why GCC generates this code. There is neither benefit nor need to have the value duplicated within the XMM register.

GCC generates similar code for converting from double precision to single precision:

```
     Conversion from double to single precision
1    vmovddup     %xmm0, %xmm0     Replicate first vector element
2    vcvtpd2psx   %xmm0, %xmm0     Convert two vector elements to single
```

Suppose these instructions start with register `%xmm0` holding two double-precision values $[x_1, x_0]$. Then the `vmovddup` instruction will set it to $[x_0, x_0]$. The `vcvtpd2psx` instruction will convert these values to single precision, pack them into the low-order half of the register, and set the upper half to 0, yielding a result $[0.0, 0.0, x_0, x_0]$ (recall that floating-point value 0.0 is represented by a bit pattern of all zeros). Again, there is no clear value in computing the conversion from one precision to another this way, rather than by using the single instruction

```
vcvtsd2ss %xmm0, %xmm0, %xmm0
```

As an example of the different floating-point conversion operations, consider the C function

```
double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long)    d;
    *fp = (float)   i;
    *dp = (double)  l;
    return (double) f;
}
```

and its associated x86-64 assembly code

```
    double fcvt(int i, float *fp, double *dp, long *lp)
    i in %edi, fp in %rsi, dp in %rdx, lp in %rcx
    fcvt:
      vmovss  (%rsi), %xmm0                   Get f = *fp
      movq    (%rcx), %rax                    Get l = *lp
      vcvttsd2siq     (%rdx), %r8             Get d = *dp and convert to long
      movq    %r8, (%rcx)                     Store at lp
      vcvtsi2ss       %edi, %xmm1, %xmm1      Convert i to float
      vmovss  %xmm1, (%rsi)                   Store at fp
      vcvtsi2sdq      %rax, %xmm1, %xmm1      Convert l to double
      vmovsd  %xmm1, (%rdx)                   Store at dp
      The following two instructions convert f to double
      vunpcklps       %xmm0, %xmm0, %xmm0
      vcvtps2pd       %xmm0, %xmm0
      ret                                     Return f
```

All of the arguments to `fcvt` are passed through the general-purpose registers, since they are either integers or pointers. The result is returned in register `%xmm0`. As is documented in Figure 3.45, this is the designated return register for `float` or `double` values. In this code, we see a number of the movement and conversion instructions of Figures 3.46–3.48, as well as GCC's preferred method of converting from single to double precision.

Practice Problem 3.50 (solution page 383)

For the following C code, the expressions `val1`–`val4` all map to the program values `i`, `f`, `d`, and `l`:

```
double fcvt2(int *ip, float *fp, double *dp, long l)
{
    int i = *ip; float f = *fp; double d = *dp;
    *ip = (int)     val1;
    *fp = (float)   val2;
    *dp = (double)  val3;
    return (double) val4;
}
```

Determine the mapping, based on the following x86-64 code for the function:

```
double fcvt2(int *ip, float *fp, double *dp, long l)
ip in %rdi, fp in %rsi, dp in %rdx, l in %rcx
Result returned in %xmm0
fcvt2:
  movl    (%rdi), %eax
  vmovss  (%rsi), %xmm0
  vcvttsd2si      (%rdx), %r8d
  movl    %r8d, (%rdi)
  vcvtsi2ss       %eax, %xmm1, %xmm1
  vmovss  %xmm1, (%rsi)
  vcvtsi2sdq      %rcx, %xmm1, %xmm1
  vmovsd  %xmm1, (%rdx)
  vunpcklps       %xmm0, %xmm0, %xmm0
  vcvtps2pd       %xmm0, %xmm0
  ret
```

Practice Problem 3.51 (solution page 384)

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;
}
```

For execution on x86-64, assume that argument `x` is either in `%xmm0` or in the appropriately named portion of register `%rdi` (i.e., `%rdi` or `%edi`). One or two instructions are to be used to perform the type conversion and to copy the value to the appropriately named portion of register `%rax` (integer result) or

`%xmm0` (floating-point result). Show the instruction(s), including the source and destination registers.

T_x	T_y	Instruction(s)
`long`	`double`	`vcvtsi2sdq %rdi, %xmm0`
`double`	`int`	
`double`	`float`	
`long`	`float`	
`float`	`long`	

3.11.2 Floating-Point Code in Procedures

With x86-64, the XMM registers are used for passing floating-point arguments to functions and for returning floating-point values from them. As is illustrated in Figure 3.45, the following conventions are observed:

- Up to eight floating-point arguments can be passed in XMM registers `%xmm0`–`%xmm7`. These registers are used in the order the arguments are listed. Additional floating-point arguments can be passed on the stack.
- A function that returns a floating-point value does so in register `%xmm0`.
- All XMM registers are caller saved. The callee may overwrite any of these registers without first saving it.

When a function contains a combination of pointer, integer, and floating-point arguments, the pointers and integers are passed in general-purpose registers, while the floating-point values are passed in XMM registers. This means that the mapping of arguments to registers depends on both their types and their ordering. Here are several examples:

```
double f1(int x, double y, long z);
```

This function would have `x` in `%edi`, `y` in `%xmm0`, and `z` in `%rsi`.

```
double f2(double y, int x, long z);
```

This function would have the same register assignment as function `f1`.

```
double f1(float x, double *y, long *z);
```

This function would have `x` in `%xmm0`, `y` in `%rdi`, and `z` in `%rsi`.

Practice Problem 3.52 (solution page 384)

For each of the following function declarations, determine the register assignments for the arguments:

A. `double g1(double a, long b, float c, int d);`

B. `double g2(int a, double *b, float *c, long d);`

C. `double g3(double *a, double b, int c, float d);`

D. `double g4(float a, int *b, float c, double d);`

3.11.3 Floating-Point Arithmetic Operations

Figure 3.49 documents a set of scalar AVX2 floating-point instructions that perform arithmetic operations. Each has either one (S_1) or two (S_1, S_2) source operands and a destination operand D. The first source operand S_1 can be either an XMM register or a memory location. The second source operand and the destination operands must be XMM registers. Each operation has an instruction for single precision and an instruction for double precision. The result is stored in the destination register.

As an example, consider the following floating-point function:

```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

The x86-64 code is as follows:

```
    double funct(double a, float x, double b, int i)
    a in %xmm0, x in %xmm1, b in %xmm2, i in %edi
    funct:
      The following two instructions convert x to double
      vunpcklps      %xmm1, %xmm1, %xmm1
      vcvtps2pd      %xmm1, %xmm1
      vmulsd  %xmm0, %xmm1, %xmm0                Multiply a by x
      vcvtsi2sd      %edi, %xmm1, %xmm1          Convert i to double
      vdivsd  %xmm1, %xmm2, %xmm2                Compute b/i
```

Single	Double	Effect	Description
vaddss	vaddsd	$D \leftarrow S_2 + S_1$	Floating-point add
vsubss	vsubsd	$D \leftarrow S_2 - S_1$	Floating-point subtract
vmulss	vmulsd	$D \leftarrow S_2 \times S_1$	Floating-point multiply
vdivss	vdivsd	$D \leftarrow S_2/S_1$	Floating-point divide
vmaxss	vmaxsd	$D \leftarrow \max(S_2, S_1)$	Floating-point maximum
vminss	vminsd	$D \leftarrow \min(S_2, S_1)$	Floating-point minimum
sqrtss	sqrtsd	$D \leftarrow \sqrt{S_1}$	Floating-point square root

Figure 3.49 Scalar floating-point arithmetic operations. These have either one or two source operands and a destination operand.

```
7     vsubsd  %xmm2, %xmm0, %xmm0             Subtract from a*x
8     ret                                     Return
```

The three floating-point arguments `a`, `x`, and `b` are passed in XMM registers `%xmm0`–`%xmm2`, while integer argument `i` is passed in register `%edi`. The standard two-instruction sequence is used to convert argument `x` to double (lines 2–3). Another conversion instruction is required to convert argument `i` to double (line 5). The function value is returned in register `%xmm0`.

Practice Problem 3.53 (solution page 384)

For the following C function, the types of the four arguments are defined by `typedef`:

```
double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
{
    return p/(q+r) - s;
}
```

When compiled, GCC generates the following code:

```
    double funct1(arg1_t p, arg2_t q, arg3_t r, arg4_t s)
    funct1:
      vcvtsi2ssq      %rsi, %xmm2, %xmm2
      vaddss  %xmm0, %xmm2, %xmm0
      vcvtsi2ss       %edi, %xmm2, %xmm2
      vdivss  %xmm0, %xmm2, %xmm0
      vunpcklps       %xmm0, %xmm0, %xmm0
      vcvtps2pd       %xmm0, %xmm0
      vsubsd  %xmm1, %xmm0, %xmm0
      ret
```

Determine the possible combinations of types of the four arguments (there may be more than one).

Practice Problem 3.54 (solution page 385)

Function `funct2` has the following prototype:

```
double funct2(double w, int x, float y, long z);
```

GCC generates the following code for the function:

```
    double funct2(double w, int x, float y, long z)
    w in %xmm0, x in %edi, y in %xmm1, z in %rsi
    funct2:
      vcvtsi2ss       %edi, %xmm2, %xmm2
      vmulss  %xmm1, %xmm2, %xmm1
```

```
  vunpcklps      %xmm1, %xmm1, %xmm1
  vcvtps2pd      %xmm1, %xmm2
  vcvtsi2sdq     %rsi, %xmm1, %xmm1
  vdivsd  %xmm1, %xmm0, %xmm0
  vsubsd  %xmm0, %xmm2, %xmm0
  ret
```

Write a C version of funct2.

3.11.4 Defining and Using Floating-Point Constants

Unlike integer arithmetic operations, AVX floating-point operations cannot have immediate values as operands. Instead, the compiler must allocate and initialize storage for any constant values. The code then reads the values from memory. This is illustrated by the following Celsius to Fahrenheit conversion function:

```
double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}
```

The relevant parts of the x86-64 assembly code are as follows:

```
double cel2fahr(double temp)
temp in %xmm0
cel2fahr:
  vmulsd  .LC2(%rip), %xmm0, %xmm0    Multiply by 1.8
  vaddsd  .LC3(%rip), %xmm0, %xmm0    Add 32.0
  ret
.LC2:
  .long   3435973837                  Low-order 4 bytes of 1.8
  .long   1073532108                  High-order 4 bytes of 1.8
.LC3:
  .long   0                           Low-order 4 bytes of 32.0
  .long   1077936128                  High-order 4 bytes of 32.0
```

We see that the function reads the value 1.8 from the memory location labeled .LC2 and the value 32.0 from the memory location labeled .LC3. Looking at the values associated with these labels, we see that each is specified by a pair of .long declarations with the values given in decimal. How should these be interpreted as floating-point values? Looking at the declaration labeled .LC2, we see that the two values are 3435973837 (0xcccccccd) and 1073532108 (0x3ffccccc.) Since the machine uses little-endian byte ordering, the first value gives the low-order 4 bytes, while the second gives the high-order 4 bytes. From the high-order bytes, we can extract an exponent field of 0x3ff (1023), from which we subtract a bias of 1023 to get an exponent of 0. Concatenating the fraction bits of the two values, we get a fraction field of 0xccccccccccccd, which can be shown to be the fractional binary representation of 0.8, to which we add the implied leading one to get 1.8.

Single	Double	Effect	Description
`vxorps`	`xorpd`	$D \leftarrow S_2 \mathbin{\char`\^} S_1$	Bitwise EXCLUSIVE-OR
`vandps`	`andpd`	$D \leftarrow S_2 \mathbin{\&} S_1$	Bitwise AND

Figure 3.50 **Bitwise operations on packed data.** These instructions perform Boolean operations on all 128 bits in an XMM register.

Practice Problem 3.55 (solution page 385)

Show how the numbers declared at label `.LC3` encode the number 32.0.

3.11.5 Using Bitwise Operations in Floating-Point Code

At times, we find GCC generating code that performs bitwise operations on XMM registers to implement useful floating-point results. Figure 3.50 shows some relevant instructions, similar to their counterparts for operating on general-purpose registers. These operations all act on packed data, meaning that they update the entire destination XMM register, applying the bitwise operation to all the data in the two source registers. Once again, our only interest for scalar data is the effect these instructions have on the low-order 4 or 8 bytes of the destination. These operations are often simple and convenient ways to manipulate floating-point values, as is explored in the following problem.

Practice Problem 3.56 (solution page 386)

Consider the following C function, where `EXPR` is a macro defined with `#define`:

```
double simplefun(double x) {
    return EXPR(x);
}
```

Below, we show the AVX2 code generated for different definitions of `EXPR`, where value `x` is held in `%xmm0`. All of them correspond to some useful operation on floating-point values. Identify what the operations are. Your answers will require you to understand the bit patterns of the constant words being retrieved from memory.

A.
```
    vmovsd  .LC1(%rip), %xmm1
    vandpd  %xmm1, %xmm0, %xmm0
  .LC1:
    .long   4294967295
    .long   2147483647
    .long   0
    .long   0
```

B.
```
    vxorpd  %xmm0, %xmm0, %xmm0
```

C.
```
    vmovsd  .LC2(%rip), %xmm1
    vxorpd  %xmm1, %xmm0, %xmm0
  .LC2:
    .long   0
    .long   -2147483648
    .long   0
    .long   0
```

3.11.6 Floating-Point Comparison Operations

AVX2 provides two instructions for comparing floating-point values:

Instruction		Based on	Description
`ucomiss`	S_1, S_2	$S_2 - S_1$	Compare single precision
`ucomisd`	S_1, S_2	$S_2 - S_1$	Compare double precision

These instructions are similar to the CMP instructions (see Section 3.6), in that they compare operands S_1 and S_2 (but in the opposite order one might expect) and set the condition codes to indicate their relative values. As with `cmpq`, they follow the ATT-format convention of listing the operands in reverse order. Argument S_2 must be in an XMM register, while S_1 can be either in an XMM register or in memory.

The floating-point comparison instructions set three condition codes: the zero flag `ZF`, the carry flag `CF`, and the parity flag `PF`. We did not document the parity flag in Section 3.6.1, because it is not commonly found in GCC-generated x86 code. For integer operations, this flag is set when the most recent arithmetic or logical operation yielded a value where the least significant byte has even parity (i.e., an even number of ones in the byte). For floating-point comparisons, however, the flag is set when either operand is *NaN*. By convention, any comparison in C is considered to fail when one of the arguments is *NaN*, and this flag is used to detect such a condition. For example, even the comparison `x == x` yields 0 when `x` is *NaN*.

The condition codes are set as follows:

Ordering S_2:S_1	`CF`	`ZF`	`PF`
Unordered	1	1	1
$S_2 < S_1$	1	0	0
$S_2 = S_1$	0	1	0
$S_2 > S_1$	0	0	0

The *unordered* case occurs when either operand is *NaN*. This can be detected with the parity flag. Commonly, the `jp` (for "jump on parity") instruction is used to conditionally jump when a floating-point comparison yields an unordered result. Except for this case, the values of the carry and zero flags are the same as those for an unsigned comparison: `ZF` is set when the two operands are equal, and `CF` is

(a) C code

```
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
    int result;
    if (x < 0)
        result = NEG;
    else if (x == 0)
        result = ZERO;
    else if (x > 0)
        result = POS;
    else
        result = OTHER;
    return result;
}
```

(b) Generated assembly code

```
    range_t find_range(float x)
    x in %xmm0
find_range:
  vxorps   %xmm1, %xmm1, %xmm1           Set %xmm1 = 0
  vucomiss        %xmm0, %xmm1           Compare 0:x
  ja       .L5                           If >, goto neg
  vucomiss        %xmm1, %xmm0           Compare x:0
  jp       .L8                           If NaN, goto posornan
  movl     $1, %eax                      result = ZERO
  je       .L3                           If =, goto done
.L8:                                   posornan:
  vucomiss        .LC0(%rip), %xmm0      Compare x:0
  setbe    %al                           Set result = NaN ? 1 : 0
  movzbl   %al, %eax                     Zero-extend
  addl     $2, %eax                      result += 2 (POS for > 0, OTHER for NaN)
  ret                                    Return
.L5:                                   neg:
  movl     $0, %eax                      result = NEG
.L3:                                   done:
  rep; ret                               Return
```

Figure 3.51 Illustration of conditional branching in floating-point code.

set when $S_2 < S_1$. Instructions such as `ja` and `jb` are used to conditionally jump on various combinations of these flags.

As an example of floating-point comparisons, the C function of Figure 3.51(a) classifies argument `x` according to its relation to 0.0, returning an enumerated type as the result. Enumerated types in C are encoded as integers, and so the possible function values are: 0 (`NEG`), 1 (`ZERO`), 2 (`POS`), and 3 (`OTHER`). This final outcome occurs when the value of `x` is *NaN*.

Gcc generates the code shown in Figure 3.51(b) for `find_range`. The code is not very efficient—it compares `x` to 0.0 three times, even though the required information could be obtained with a single comparison. It also generates floating-point constant 0.0 twice—once using `vxorps`, and once by reading the value from memory. Let us trace the flow of the function for the four possible comparison results:

`x` < 0.0 The `ja` branch on line 4 will be taken, jumping to the end with a return value of 0.

`x` $= 0.0$ The `ja` (line 4) and `jp` (line 6) branches will not be taken, but the `je` branch (line 8) will, returning with `%eax` equal to 1.

`x` > 0.0 None of the three branches will be taken. The `setbe` (line 11) will yield 0, and this will be incremented by the `addl` instruction (line 13) to give a return value of 2.

`x` $= NaN$ The `jp` branch (line 6) will be taken. The third `vucomiss` instruction (line 10) will set both the carry and the zero flag, and so the `setbe` instruction (line 11) and the following instruction will set `%eax` to 1. This gets incremented by the `addl` instruction (line 13) to give a return value of 3.

In Homework Problems 3.73 and 3.74, you are challenged to hand-generate more efficient implementations of `find_range`.

Practice Problem 3.57 (solution page 386)

Function `funct3` has the following prototype:

```
double funct3(int *ap, double b, long c, float *dp);
```

For this function, GCC generates the following code:

```
    double funct3(int *ap, double b, long c, float *dp)
    ap in %rdi, b in %xmm0, c in %rsi, dp in %rdx
funct3:
  vmovss  (%rdx), %xmm1
  vcvtsi2sd      (%rdi), %xmm2, %xmm2
  vucomisd       %xmm2, %xmm0
  jbe     .L8
  vcvtsi2ssq     %rsi, %xmm0, %xmm0
  vmulss  %xmm1, %xmm0, %xmm1
  vunpcklps       %xmm1, %xmm1, %xmm1
  vcvtps2pd       %xmm1, %xmm0
  ret
.L8:
  vaddss  %xmm1, %xmm1, %xmm1
  vcvtsi2ssq      %rsi, %xmm0, %xmm0
  vaddss  %xmm1, %xmm0, %xmm0
  vunpcklps       %xmm0, %xmm0, %xmm0
  vcvtps2pd       %xmm0, %xmm0
  ret
```

Write a C version of `funct3`.

3.11.7 Observations about Floating-Point Code

We see that the general style of machine code generated for operating on floating-point data with AVX2 is similar to what we have seen for operating on integer data. Both use a collection of registers to hold and operate on values, and they use these registers for passing function arguments.

Of course, there are many complexities in dealing with the different data types and the rules for evaluating expressions containing a mixture of data types, and AVX2 code involves many more different instructions and formats than is usually seen with functions that perform only integer arithmetic.

AVX2 also has the potential to make computations run faster by performing parallel operations on packed data. Compiler developers are working on automating the conversion of scalar code to parallel code, but currently the most reliable way to achieve higher performance through parallelism is to use the extensions to the C language supported by GCC for manipulating vectors of data. See Web Aside OPT:SIMD on page 582 to see how this can be done.

3.12 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 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 x86-64 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, 50], as well as detailed references about the individual instructions [3, 51]. 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 x86-64.org has been responsible for defining the *application binary interface* (ABI) for x86-64 code running on Linux systems [77]. 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.