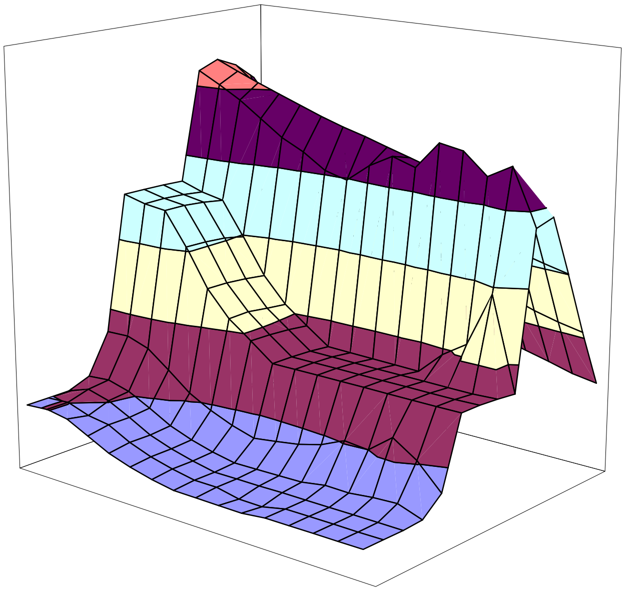
Computer Systems

A Programmer's Perspective [1]



Randal E. Bryant
David R. O'Hallaron

April 22, 2002

Contents

Preface

This book (CS:APP) is for programmers who want to improve their skills by learning what is going on "under the hood" of a computer system.

Our aim is to explain the enduring concepts underlying all computer systems, and to show you the concrete ways that these ideas affect the correctness, performance, and utility of your application programs. Unlike other systems books, which are written primarily for system builders, this book is written for programmers, from a programmer's perspective.

If you study and learn the concepts in this book, you will be on your way to becoming the rare "power programmer" who knows how things work and how to fix them when they break. You will also be prepared to study specific systems topics such as compilers, computer architecture, operating systems, embedded systems, and networking.

Assumptions About the Reader's Background

The examples in the book are based on Intel-compatible processors (called "IA32" by Intel and "x86" colloquially) running C programs on Unix or Unix-like (such as Linux) operating systems. (To simplify our presentation, we will use the term "Unix" as an umbrella term for systems like Solaris and Linux.) The text contains numerous programming examples that have been compiled and run on Linux systems. We assume that you have access to such a machine, and are able to log in and do simple things such as changing directories.

If your computer runs Microsoft Windows, you have two choices. First, you can get a copy of Linux (see `www.linux.org` or `www.redhat.com`) and install it as a "dual boot" option, so that your machine can run either operating system. Alternatively, by installing a copy of the Cygwin tools (`www.cygwin.com`), you can have up a Unix-like shell under Windows and have an environment very close to that provided by Linux. Not all features of Linux are available under Cygwin, however.

We also assume that you have some familiarity with C or C++. If your only prior experience is with Java, the transition will require more effort on your part, but we will help you. Java and C share similar syntax and control statements. However, there are aspects of C, particularly pointers, explicit dynamic memory allocation, and formatted I/O, that do not exist in Java. Fortunately, C is a small language, and it is clearly and beautifully described in the classic "K&R" text by Brian Kernighan and Dennis Ritchie [41]. Regardless of your programming background, consider K&R an essential part of your personal systems library.

Several of the early chapters in the book explore the interactions between C programs and their machine-language counterparts. The machine language examples were all generated by the GNU GCC compiler running on an Intel IA32 processor. We do not assume any prior experience with hardware, machine language, or assembly-language programming.

> **New to C?: Advice on the C Programming Language**
> To help readers whose background in C programming is weak (or nonexistent), we have also included these special notes to highlight features that are especially important in C. We assume you are familiar with C++ or Java. **End.**

How to Read the Book

Learning how computer systems work from a programmer's perspective is great fun, mainly because it can be done so actively. Whenever you learn some new thing, you can try it out right away and see the result first hand. In fact, we believe that the only way to learn systems is to *do* systems, either working concrete problems, or writing and running programs on real systems.

This theme pervades the entire book. When a new concept is introduced, it is followed in the text by one or more *practice problems* that you should work immediately to test your understanding. Solutions to the practice problems are at the end of each chapter (look for the blue edge). As you read, try to solve each problem on your own, and then check the solution to make sure you are on the right track. Each chapter is followed by a set of *homework problems* of varying difficulty. Your instructor has the solutions to the homework problems in an Instructor's Manual. For each homework problem, we show a rating of the amount of effort we feel it will require:

◆ Should require just a few minutes. Little or no programming required.

◆◆ Might require up to 20 minutes. Often involves writing and testing some code. Many of these are derived from problems we have given on exams.

◆◆◆ Requires a significant effort, perhaps 1–2 hours. Generally involves writing and testing a significant amount of code.

◆◆◆◆ A lab assignment, requiring up to 10 hours of effort.

Each code example in the text was formatted directly, without any manual intervention, from a C program compiled with GCC version 2.95.3, and tested on a Linux system with a 2.2.16 kernel. All of the source code is available from the CS:APP Web page at `csapp.cs.cmu.edu`. In the text, the file names of the source programs are documented in horizontal bars that surround the formatted code. For example, the program in Figure 1 can be found in the file `hello.c` in directory `code/intro/`. We encourage you to try running the example programs on your system as you encounter them.

Finally, some sections (denoted by a "*") contain material that you might find interesting, but that can be skipped without any loss of continuity.

> **Aside: What is an aside?**
> You will encounter asides of this form throughout the text. Asides are parenthetical remarks that give you some additional insight into the current topic. Asides serve a number of purposes. Some are little history lessons. For

code/intro/hello.c

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

code/intro/hello.c

Figure 1: **A typical code example.**

example, where did C, Linux, and the Internet come from? Other asides are meant to clarify ideas that students often find confusing. For example, what is the difference between a cache line, set, and block? Other asides give real-world examples. For example, how a floating-point error crashed a French rocket, or what the geometry of a real IBM disk drive looks like. Finally, some asides are just fun stuff. For example, what is a "hoinky"? **End Aside.**

Origins of the Book

The book stems from an introductory course that we developed at Carnegie Mellon University in the Fall of 1998, called *15-213: Introduction to Computer Systems* (ICS) [7]. The ICS course has been taught every semester since then, each time to about 150 students, mostly sophomores in computer science and computer engineering. It has since become a prerequisite for most upper-level systems courses in the CS and ECE departments at Carnegie Mellon.

The idea with ICS was to introduce students to computers in a different way. Few of our students would have the opportunity to build a computer system. On the other hand, most students, even the computer engineers, would be required to use and program computers on a daily basis. So we decided to teach about systems from the point of view of the programmer, using the following filter: We would cover a topic only if it affected the performance, correctness, or utility of user-level C programs.

For example, topics such as hardware adder and bus designs were out. Topics such as machine language were in, but instead of focusing on how to write assembly language, we would look at how C constructs such as pointers, loops, procedure calls and returns, and switch statements were translated by the compiler. Further, we would take a broader and more realistic view of the system as both hardware and systems software, covering such topics as linking, loading, processes, signals, performance optimization, measurement, I/O, and network and concurrent programming.

This approach allowed us to teach the ICS course in a way that was practical, concrete, hands-on, and exciting for the students. The response from our students and faculty colleagues was immediate and overwhelmingly positive, and we realized that others outside of CMU might benefit from using our approach. Hence this book, which we developed over a period of two years from the ICS lecture notes.

Aside: ICS numerology.
The numerology of the ICS course is a little strange. About halfway through the first semester, we realized that the assigned course number (15-213) was also the CMU zip code, hence the motto "15-213: The course that gives CMU its zip!". By chance, the alpha version of the manuscript was printed on February 13, 2001 (2/13/01). When

we presented the course at the SIGCSE education conference, the talk was scheduled in Room 213. And the final version of the book has 13 chapters. It's a good thing we're not superstitious! **End Aside.**

Overview of the Book

The CS:APP book consists of 13 chapters designed to capture the core ideas in computer systems:

- *Chapter 1: A Tour of Computer Systems.* This chapter introduces the major ideas and themes in computer systems by tracing the life cycle of a simple "hello, world" program.

- *Chapter 2: Representing and Manipulating Information.* We cover computer arithmetic, emphasizing the properties of unsigned and two's complement number representations that affect programmers. We consider how numbers are represented and therefore what range of values can be encoded for a given word size. We consider the effect of casting between signed and unsigned numbers. We cover the mathematical properties of arithmetic operations. Students are surprised to learn that the (two's complement) sum or product of two positive numbers can be negative. On the other hand, two's complement arithmetic satisfies ring properties, and hence a compiler can transform multiplication by a constant into a sequence of shifts and adds. We use the bit-level operations of C to demonstrate the principles and applications of Boolean algebra. We cover the IEEE floating point format in terms of how it represents values and the mathematical properties of floating point operations.

  Having a solid understanding of computer arithmetic is critical to writing reliable programs. For example, one cannot replace the expression `(x<y)` with `(x-y<0)` due to the possibility of overflow. One cannot even replace it with the expression `(-y<-x)` due to the asymmetric range of negative and positive numbers in the two's complement representation. Arithmetic overflow is a common source of programming errors, yet few other books cover the properties of computer arithmetic from a programmer's perspective.

- *Chapter 3: Machine-Level Representation of Programs.* We teach students how to read the IA32 assembly language generated by a C compiler. We cover the basic instruction patterns generated for different control constructs, such as conditionals, loops, and switch statements. We cover the implementation of procedures, including stack allocation, register usage conventions and parameter passing. We cover the way different data structures such as structures, unions, and arrays are allocated and accessed. Learning the concepts in this chapter helps students become better programmers, because they understand how their programs are represented on the machine. Another nice benefit is that students develop a concrete understanding of pointers.

- *Chapter 4: Processor Architecture.* This chapter covers basic combinational and sequential logic elements and then shows how these elements can be combined in a datapath that executes a simplified subset of the IA32 instruction set called "Y86." We begin with the design of a single-cycle non-pipelined datapath, which we extend into a five-stage pipelined design. The control logic for the processor designs in this chapter are described using a simple hardware description language called HCL. Hardware designs written in HCL can be compiled and linked into graphical processor simulators provided with the textbook.

- *Chapter 5: Optimizing Program Performance.* In this chapter we introduce a number of techniques for improving code performance. We start with machine-independent program transformations that should be standard practice when writing any program on any machine. We then progress to transformations whose efficacy depends on the characteristics of the target machine and compiler. To motivate these transformation, we introduce a simple operational model of how modern out-of-order processors work, and then show students how to use this model to improve the performance of their C programs.

- *Chapter 6: The Memory Hierarchy.* The memory system is one of the most visible parts of a computer system to application programmers. To this point, the students have relied on a conceptual model of the memory system as a linear array with uniform access times. In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. We cover the different types of RAM and ROM memories and the geometry and organization of modern disk drives. We describe how these storage devices are arranged in a hierarchy. We show how this hierarchy is made possible by locality of reference. We make these ideas concrete by introducing a unique view of a memory system as a "memory mountain" with ridges of temporal locality and slopes of spatial locality. Finally, we show students how to improve the performance of application programs by improving their temporal and spatial locality.

- *Chapter 7: Linking.* This chapter covers both static and dynamic linking, including the ideas of relocatable and executable object files, symbol resolution, relocation, static libraries, shared object libraries, and position-independent code. Linking is not covered in most systems texts, but we cover it for several reasons. First, some of the most confusing errors that students can encounter are related to glitches during linking, especially for large software packages. Second, the object files produced by linkers are tied to concepts such as loading, virtual memory, and memory mapping.

- *Chapter 8: Exceptional Control Flow.* In this part of the course we break the single-program model by introducing the general concept of exceptional control flow (i.e., changes in control flow that are outside the normal branches and procedure calls). We cover examples of exceptional control flow that exist at all levels of the system, from low-level hardware exceptions and interrupts, to context switches between concurrent processes, to abrupt changes in control flow caused by the delivery of Unix signals, to the nonlocal jumps in C that break the stack discipline.

  This is the part of the book where we introduce students to the fundamental idea of a process. Students learn how processes work and how they can be created and manipulated from application programs. We show them how application programmers can make use of multiple processes via Unix system calls. When students finish this chapter, they are able to write a Unix shell with job control.

- *Chapter 9: Measuring Program Execution Time.* This chapter teaches students how computers keep track of time (interval timers, cycle timers, and system clocks), the sources of error when we try to use these times to measure running time, and how to exploit this knowledge to get accurate measurements. To the best of our knowledge, this is unique material that has never been discussed before in any consistent way. We include the topic at this point because it requires an understanding of assembly language, processes, and caches.

- *Chapter 10: Virtual Memory.* Our presentation of the virtual memory system seeks to give students some understanding of how it works and its characteristics. We want students to know how it is that the different simultaneous processes can each use an identical range of addresses, sharing some pages but having individual copies of others. We also cover issues involved in managing and manipulating virtual memory. In particular, we cover the operation of storage allocators such as the Unix `malloc` and `free` operations. Covering this material serves several purposes. It reinforces the concept that the virtual memory space is just an array of bytes that the program can subdivide into different storage units. It helps students understand the effects of programs containing memory referencing errors such as storage leaks and invalid pointer references. Finally, many application programmers write their own storage allocators optimized toward the needs and characteristics of the application.

- *Chapter 11: System-Level I/O.* We cover the basic concepts of Unix I/O such as files and descriptors. We describe how files are shared, how I/O redirection works, and how to access file metadata. We also develop a robust buffered I/O package that deals correctly with short counts. We cover the C standard I/O library and its relationship to Unix I/O, focusing on limitations of standard I/O that make it unsuitable for network programming. In general, the topics covered in this chapter are building blocks for the next two chapters on network and concurrent programming.

- *Chapter 12: Network Programming.* Networks are interesting I/O devices to program, tying together many of the ideas that we have studied earlier in the text, such as processes, signals, byte ordering, memory mapping, and dynamic storage allocation. Network programs also provide a compelling context for concurrency, which is the topic of the next section. This chapter is a thin slice through network programming that gets the students to point where they can write a Web server. We cover the client-server model that underlies all network applications. We present a programmer's view of the Internet, and show students how to write Internet clients and servers using the sockets interface. Finally, we introduce HTTP and develop a simple iterative Web server.

- *Chapter 13: Concurrent Programming.* This chapter introduces students to concurrent programming using Internet server design as the running motivational example. We compare and contrast the three basic mechanisms for writing concurrent programs — processes, I/O multiplexing, and threads — and show how to use them to build concurrent Internet servers. We cover basic principles of synchronization using P and V semaphore operations, thread safety and reentrancy, race conditions, and deadlocks.

Courses Based on the Book

Instructors can use the CS:APP book to teach five different kinds of systems courses (Figure 2). The particular course depends on curriculum requirements, personal taste, and the backgrounds and abilities of the students. From left to right in the figure, the courses are characterized by an increasing emphasis on the programmer's perspective of a system. Here is a brief description:

- **ORG**: A computer organization course with traditional topics covered in an untraditional style. Traditional topics such as logic design, processor architecture, assembly language, and memory systems are covered. However, there is more emphasis on the impact for the programmer. For example, data representations are related back to their impact on C programs. Students learn how C constructs are represented in machine language.

- **ORG+**: The ORG course with additional emphasis on the impact of hardware on the performance of application programs. Compared to ORG, students learn more about code optimization and about improving the memory performance of their C programs.
- **ICS**: The baseline ICS course, designed to produce enlightened programmers who understand the impact of the hardware, operating system, and compilation system on the performance and correctness of their application programs. A significant difference from ORG+ is that low-level processor architecture is not covered. Instead, programmers work with a higher-level model of a modern out-of-order processor. The ICS course fits nicely into a 10-week quarter, and can also be stretched to a 15-week semester if covered at a more leisurely pace.
- **ICS+**: The baseline ICS course with additional coverage of systems programming topics such as system-level I/O, network programming, and concurrent programming. This is the semester-long Carnegie Mellon course, which covers every chapter in CS:APP except low-level processor architecture.
- **SP**: A systems programming course. Similar to the ICS+ course, but drops floating point and performance optimization, and places more emphasis on systems programming, including process control, dynamic linking, system-level I/O, network programming, and concurrent programming. Instructors might want to supplement from other sources for advanced topics such as daemons, terminal control, and Unix IPC.

		Course				
Chapter	Topic	ORG	ORG+	ICS	ICS+	SP
1	Tour of Systems	●	●	●	●	●
2	Data representation	●	●	●	●	⊙ (d)
3	Machine language	●	●	●	●	●
4	Processor architecture	●	●			
5	Code optimization		●	●	●	
6	Memory hierarchy	⊙ (a)	●	●	●	⊙ (a)
7	Linking			⊙ (c)	⊙ (c)	●
8	Exceptional control flow			●	●	●
9	Performance measurement				●	●
10	Virtual memory	⊙ (b)	●	●	●	●
11	System-level I/O				●	●
12	Network programming				●	●
13	Concurrent programming				●	●

Figure 2: **Five systems courses based on the CS:APP book.** Notes: (a) Hardware only, (b) No dynamic storage allocation, (c) No dynamic linking, (d) No floating point. ICS+ is the 15-213 course from Carnegie Mellon.

The main message of Figure 2 is that the CS:APP book gives you a lot of options. If you want your students to be exposed to lower-level processor architecture, then that option is available via the ORG and ORG+ courses. On the other hand, if you want to switch from your current computer organization course to an ICS or ICS+ course, but are wary are making such a drastic change all at once, then you can move towards ICS incrementally. You can start with ORG, which teaches the traditional topics in an non-traditional way. Once you are comfortable with that material, then you can move to ORG+, and eventually to ICS. If students have no experience in C (for example they have only programmed in Java), you could spend several weeks on C and then cover the material of ORG or ICS.

Finally, we note that the ORG+ and SP courses would make a nice two-term (either quarters or semesters) sequence. Or you might consider offering ICS+ as one term of ICS and one term of SP.

Classroom-Tested Laboratory Exercises

The ICS+ course at Carnegie Mellon receives very high evaluations from students. Median scores of $5.0/5.0$ and means of $4.6/5.0$ are typical. Students cite the fun, exciting, and relevant laboratory exercises as the primary reason. Here are examples of the labs that are provided with the book:

- *Data Lab.* This lab requires students to implement simple logical and arithmetic functions, but using a highly restricted subset of C. For example, they must compute the absolute value of a number using only bit-level operations. This lab helps students understand the bit-level representations of C data types and the bit-level behavior of the operations on data.
- *Binary Bomb Lab.* A *binary bomb* is a program provided to students as an object code file. When run, it prompts the user to type in 6 different strings. If any of these is incorrect, the bomb "explodes," printing an error message and logging the event on a grading server. Students must "defuse" their own unique bomb by disassembling and reverse engineering the program to determine what the 6 strings should be. The lab teaches students to understand assembly language, and also forces them to learn how to use a debugger.
- *Buffer Overflow Lab.* Students are required to modify the run-time behavior of a binary executable by exploiting a buffer overflow bug. This lab teaches the students about the stack discipline and teaches them about the danger of writing code that is vulnerable to buffer overflow attacks.
- *Architecture Lab.* Several of the homework problems of Chapter 4 could be combined into a lab assignment, where students modify the HCL description of a processor to add new instructions, change the branch prediction policy, or add or remove bypassing paths and register ports. The resulting processors can be simulated and run through automated tests that will detect most of the possible bugs. This lab lets students experience the exciting parts of processor design without learning and constructing complex, low-level models in a language such as Verilog or VHDL.
- *Performance Lab.* Students must optimize the performance of an application kernel function such as convolution or matrix transposition. This lab provides a very clear demonstration of the properties of cache memories and gives them experience with low-level program optimization.
- *Shell Lab.* Students implement their own Unix shell program with job control, including the `ctrl-c` and `ctrl-z` keystrokes, `fg`, `bg`, and `jobs` commands. This is the student's first introduction to concurrency, and gives them a clear idea of Unix process control, signals, and signal handling.
- *Malloc Lab.* Students implement their own version of `malloc`, `free`, and (optionally) `realloc`. This lab gives students a clear understanding of data layout and organization, and requires them to evaluate different trade-offs between space and time efficiency.

- *Proxy Lab.* Students implement a concurrent Web proxy that sits between their browser and the rest of the World Wide Web. This lab exposes the students to such topics as web clients and servers, and ties together many of the concepts from the course, such as byte ordering, file I/O, process control, signals, signal handling, memory mapping, sockets, and concurrency.

The labs are available from the CS:APP Web page.

Acknowledgements

We are deeply indebted to many friends and colleagues for their thoughtful criticisms and encouragement. A special thanks to our 15-213 students, whose infectious energy and enthusiasm spurred us on. Nick Carter and Vinny Furia generously provided their malloc package.

Guy Blelloch, Greg Kesden, Bruce Maggs, and Todd Mowry taught the course over multiple semesters, gave us encouragement, and helped improve the course material. Herb Derby provided early spiritual guidance and encouragement. Allan Fisher, Garth Gibson, Thomas Gross, Satya, Peter Steenkiste, and Hui Zhang encouraged us to develop the course from the start. A suggestion from Garth early on got the whole ball rolling, and this was picked up and refined with the help of a group led by Allan Fisher. Mark Stehlik and Peter Lee have been very supportive about building this material into the undergraduate curriculum. Greg Kesden provided helpful feedback on the impact of ICS on the OS course. Greg Ganger and Jiri Schindler graciously provided some disk drive characterizations and answered our questions on modern disks. Tom Stricker showed us the memory mountain. James Hoe provided useful ideas and feedback on how to present processor architecture.

A special group of students, Khalil Amiri, Angela Demke Brown, Chris Colohan, Jason Crawford, Peter Dinda, Julio Lopez, Bruce Lowekamp, Jeff Pierce, Sanjay Rao, Balaji Sarpeshkar, Blake Scholl, Sanjit Seshia, Greg Steffan, Tiankai Tu, Kip Walker, and Yinglian Xie were instrumental in helping us develop the content of the course. In particular, Chris Colohan established a fun (and funny) tone that persists to this day, and invented the legendary “binary bomb” that has proven to be a great tool for teaching machine code and debugging concepts.

Chris Bauer, Alan Cox, Peter Dinda, Sandhya Dwarkadis, John Greiner, Bruce Jacob, Barry Johnson, Don Heller, Bruce Lowekamp, Greg Morrisett, Brian Noble, Bobbie Othmer, Bill Pugh, Michael Scott, Mark Smotherman, Greg Steffan, and Bob Wier took time that they did not have to read and advise us on early drafts of the book. A very special thanks to Peter Dinda (Northwestern University), John Greiner (Rice University), Wei Hsu (University of Minnesota), Bruce Lowekamp (William & Mary), Bobbie Othmer (University of Minnesota), Michael Scott (University of Rochester), and Bob Wier (Rocky Mountain College) for class testing the Beta version. A special thanks to their students as well!

We would also like to thank our colleagues at Prentice Hall. Marcia Horton, Eric Frank, and Harold Stone have been unflagging in their support and vision. Jerry Ralya provided sharp insights and taught us a lot about good writing.

Finally, we would like to acknowledge the great technical writers Brian Kernighan and the late W. Richard Stevens, for showing us that technical books can be beautiful.

Thank you all.

Randy Bryant
Dave O'Hallaron

Pittsburgh, PA
February 1, 2002

Chapter 1

A Tour of Computer Systems

A *computer system* consists of hardware and systems software that work together to run application programs. Specific implementations of systems change over time, but the underlying concepts do not. All computer systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to get better at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

You are poised for an exciting journey. If you dedicate yourself to learning the concepts in this book, then you will be on your way to becoming a rare "power programmer," enlightened by an understanding of the underlying computer system and its impact on your application programs.

You are going to learn practical skills such as how to avoid strange numerical errors caused by the way that computers represent numbers. You will learn how to optimize your C code by using clever tricks that exploit the designs of modern processors and memory systems. You will learn how the compiler implements procedure calls and how to use this knowledge to avoid the security holes from buffer overflow bugs that plague network and Internet software. You will learn how to recognize and avoid the nasty errors during linking that confound the average programmer. You will learn how to write your own Unix shell, your own dynamic storage allocation package, and even your own Web server!

In their classic text on the C programming language [41], Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.1. Although `hello` is a very simple program, every major

code/intro/hello.c

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

code/intro/hello.c

Figure 1.1: **The `hello` program.**

part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why, when you run `hello` on your system.

We begin our study of systems by tracing the lifetime of the `hello` program, from the time it is created by a programmer, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

1.1 Information is Bits + Context

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most modern systems represent text characters using the ASCII standard that represents each character with a unique byte-sized integer value. For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

```
 #   i    n     c   l    u    d    e   <sp>   <   s    t    d    i    o     .
 35  105  110   99  108  117  100  101   32   60  115  116  100  105  111   46

h     >    \n   \n  i    n    t   <sp>  m     a   i    n     (    )    \n  {
104   62   10   10  105  110  116   32  109   97  105  110   40   41   10  123

 \n <sp> <sp> <sp> <sp>  p    r    i    n    t    f     (    "   h    e    l
 10   32   32   32   32  112  114  105  110  116  102   40   34  104  101  108

l    o     ,  <sp>  w    o    r    l    d     \   n     "    )    ;    \n  }
108  111   44   32  119  111  114  108  100   92  110   34   41   59   10  125
```

Figure 1.2: **The ASCII text representation of `hello.c`.**

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '#'. The second byte has the integer value 105, which corresponds to the character '`i`', and so on. Notice that each text line is terminated by the invisible *newline* character '`\n`', which is represented by the integer value 10. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

As programmers, we need to understand machine representations of numbers because they are not the same as integers and real numbers. They are finite approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

Aside: The C programming language.
C was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989. The standard defines the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as "K&R" [41]. In Ritchie's words [65], C is "quirky, flawed, and an enormous success." So why the success?

- *C was closely tied with the Unix operating system.* C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel, and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.
- *C is a small, simple language.* The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
- *C was designed for a practical purpose.* C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed base of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes, objects, and exceptions. Newer languages such as C++ and Java address these issues for application-level programs. **End Aside.**

1.2 Programs Are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the # character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.

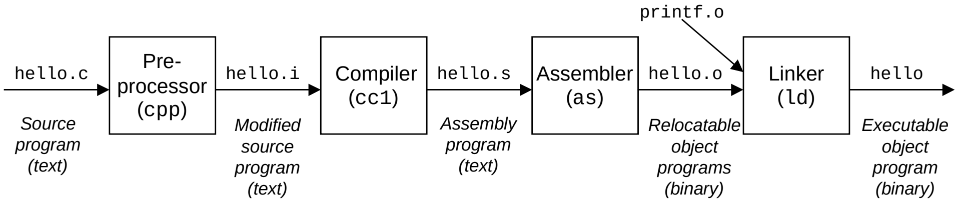


Figure 1.3: **The compilation system.**

- *Compilation phase.* The compiler (cc1) translates the text file hello.i into the text file hello.s, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- *Assembly phase.* Next, the assembler (as) translates hello.s into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file hello.o. The hello.o file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view hello.o with a text editor, it would appear to be gibberish.

- *Linking phase.* Notice that our hello program calls the printf function, which is part of the *standard C library* provided by every C compiler. The printf function resides in a separate precompiled object file called printf.o, which must somehow be merged with our hello.o program. The linker (ld) handles this merging. The result is the hello file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

Aside: The GNU project.
GCC is one of many useful tools developed by the GNU (short for GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. As of 2002, the GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and other components.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open-source movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software* ("free" as in "free speech" not "free beer"). Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel. **End Aside.**

1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as hello.c, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of assembly language and how the compiler translates different C statements into assembly language. For example, is a `switch` statement always more efficient than a sequence of `if-then-else` statements? Just how expensive is a function call? Is a `while` loop more efficient than a `do` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? Why do two functionally equivalent loops have such different running times?

  In Chapter 3, we will introduce the Intel IA32 machine language and describe how compilers translate different C constructs into that language. In Chapter 5 you will learn how to tune the performance of your C programs by making simple transformations to the C code that help the compiler do its job. And in Chapter 6 you will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how your C programs can exploit this knowledge to run more efficiently.

- *Understanding link-time errors.* In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when you are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if you define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run time? You will learn the answers to these kinds of questions in Chapter 7

- *Avoiding security holes.* For many years now, *buffer overflow bugs* have accounted for the majority of security holes in network and Internet servers. These bugs exist because too many programmers are ignorant of the stack discipline that compilers use to generate code for functions. We will describe the stack discipline and buffer overflow bugs in Chapter 3 as part of our study of assembly language.

1.4 Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell*:

```
unix> ./hello
hello, world
unix>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

1.4.1 Hardware Organization of a System

To understand what happens to our `hello` program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

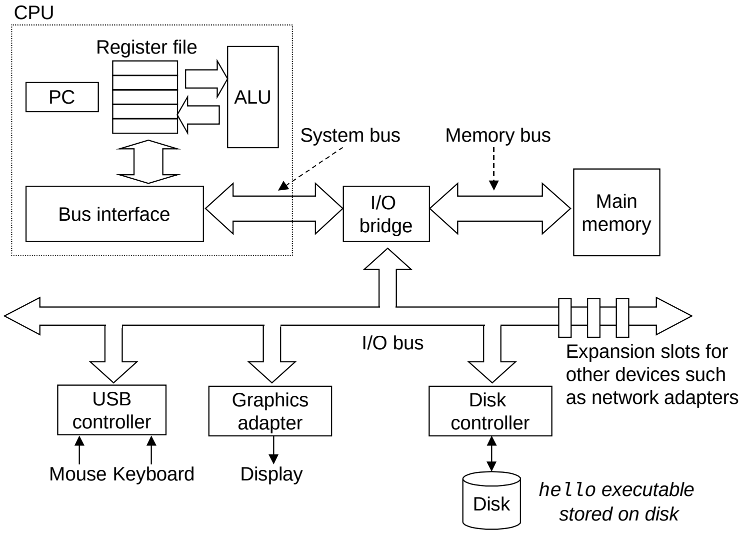


Figure 1.4: **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-sized chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. For example, Intel Pentium systems have a word size of 4 bytes, while server-class systems such as Intel Itaniums and high-end Sun SPARCS have word sizes of 8 bytes. Smaller systems that are used as embedded controllers in automobiles and factories can have word sizes of 1 or 2 bytes. For simplicity, we will assume a word size of 4 bytes, and we will assume that buses transfer only one word at a time.

I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 11, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *Dynamic Random Access Memory (DRAM)* chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an Intel machine running Linux, data of type `short` requires two bytes, types `int`, `float`, and `long` four bytes, and type `double` eight bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory. [1]

From the time that power is applied to the system, until the time that the power is shut off, the processor blindly and repeatedly performs the same basic task, over and over again: It reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple *operation* dictated by the instruction, and then updates the PC to point to the *next* instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and

[1] PC is also a commonly nused acronym for "personal computer". However, the distinction between the two should be clear from the context.

the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-sized registers, each with its own unique name. The ALU computes new data and address values. Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

- *Load:* Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.
- *Store:* Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.
- *Update:* Copy the contents of two registers to the ALU, which adds the two words together and stores the result in a register, overwriting the previous contents of that register.
- *I/O Read:* Copy a byte or a word from an I/O device into a register.
- *I/O Write:* Copy a byte or a word from a register to an I/O device.
- *Jump:* Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

Chapter 4 has much more to say about how processors work.

1.4.2 Running the `hello` Program

Given this simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters "`./hello`" at the keyboard, the shell program reads each one into a register, and then stores it in memory, as shown in Figure 1.5.

When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the `hello` object file from disk to main memory. The data include the string of characters ”`hello, world\n`” that will eventually be printed out.

Using a technique known as *direct memory access* (DMA, discussed in Chapter 6), the data travels directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's `main` routine. These instruction copy the bytes in the ”`hello, world\n`” string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

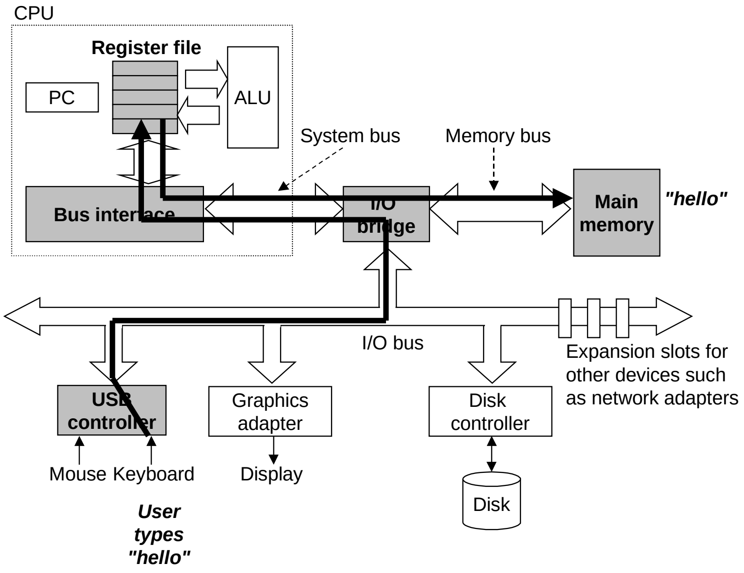


Figure 1.5: **Reading the `hello` command from the keyboard.**

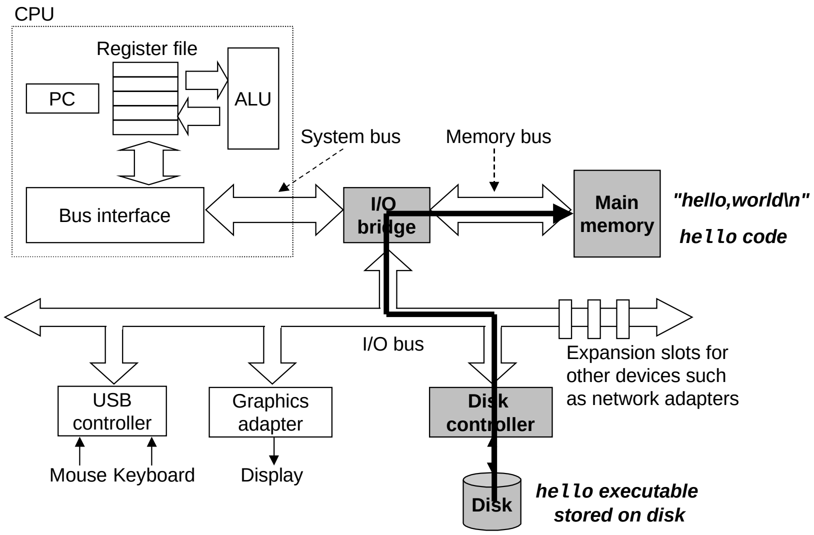


Figure 1.6: **Loading the executable from disk into main memory.**

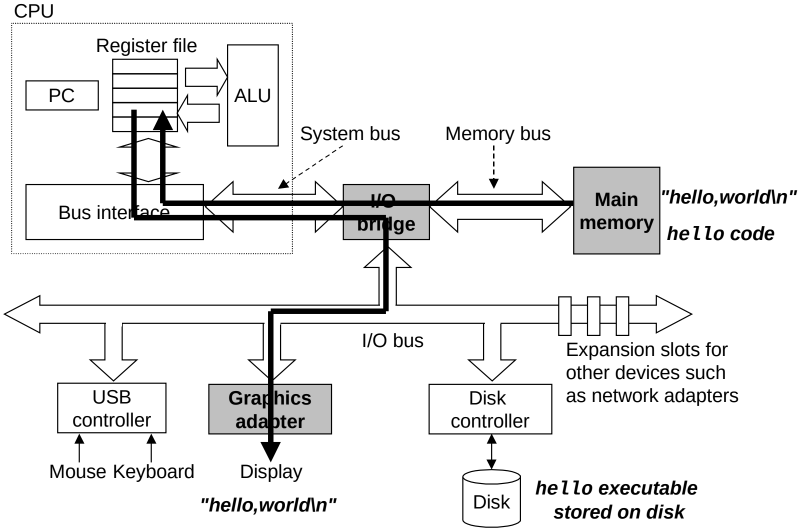


Figure 1.7: **Writing the output string from memory to the display.**

1.5 Caches Matter

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another. The machine instructions in the `hello` program are originally stored on disk. When the program is loaded, they are copied to main memory. As the processor runs the program, instructions are copied from main memory into the processor. Similarly, the data string ”`hello,world\n`”, originally on disk, is copied to main memory, and then copied from main memory to the display device. From a programmer’s perspective, much of this copying is overhead that slows down the “real work” of the program. Thus, a major goal for system designers is make these copy operations run as fast as possible.

Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower counterparts. For example, the disk drive on a typical system might be 100 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred bytes of information, as opposed to millions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this *processor-memory gap* continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

To deal with the processor-memory gap, system designers include smaller faster storage devices called *cache memories* (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. Figure 1.8 shows the cache memories in a typical system. An *L1 cache* on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file. A larger *L2 cache* with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5

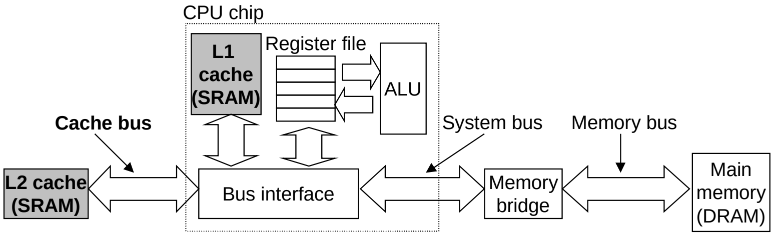


Figure 1.8: **Cache memories.**

to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as *Static Random Access Memory* (SRAM).

One of the most important lessons in this book is that application programmers who are aware of cache memories can exploit them to improve the performance of their programs by an order of magnitude. You will learn more about these important devices and how to exploit them in Chapter 6.

1.6 Storage Devices Form a Hierarchy

This notion of inserting a smaller, faster storage device (e.g., cache memory) between the processor and a larger slower device (e.g., main memory) turns out to be a general idea. In fact, the storage devices in every computer system are organized as a *memory hierarchy* similar to Figure 1.9. As we move from the top of the hierarchy to the bottom, the devices become slower, larger, and less costly per byte. The register file

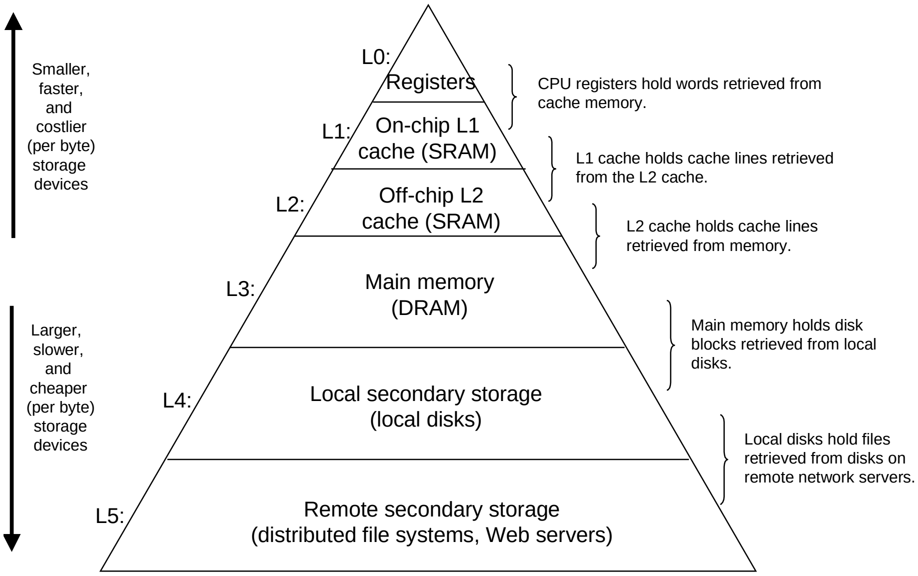

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Registers
L1:
On-chip L1 cache (SRAM)
L2:
Off-chip L2 cache (SRAM)
L3:
Main memory (DRAM)
L4:
Local secondary storage (local disks)
L5:
Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Figure 1.9: **An example of a memory hierarchy.**

occupies the top level in the hierarchy, which is known as level 0 or L0. The L1 cache occupies level 1 (hence the term L1). The L2 cache occupies level 2. Main memory occupies level 3, and so on.

The main idea of a memory hierarchy is that storage at one level serves as a cache for storage at the next lower level. Thus, the register file is a cache for the L1 cache, which is a cache for the L2 cache, which is a cache for the main memory, which is a cache for the disk. On some networked systems with distributed file systems, the local disk serves as a cache for data stored on the disks of other systems.

Just as programmers can exploit knowledge of the L1 and L2 caches to improve performance, programmers can exploit their understanding of the entire memory hierarchy. Chapter 6 will have much more to say about this.

1.7 The Operating System Manages the Hardware

Back to our `hello` example. When the shell loaded and ran the `hello` program, and when the `hello` program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the *operating system*. We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10. All attempts by an application program to manipulate the hardware must go through the operating system.

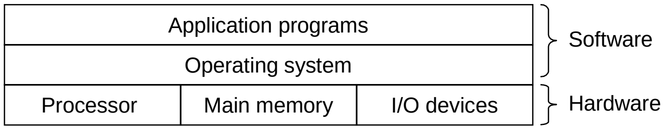


Figure 1.10: **Layered view of a computer system.**

The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications, and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices. The operating system achieves both goals via the fundamental abstractions shown in Figure 1.11: *processes*, *virtual memory*, and *files*. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices. We will discuss each in turn.

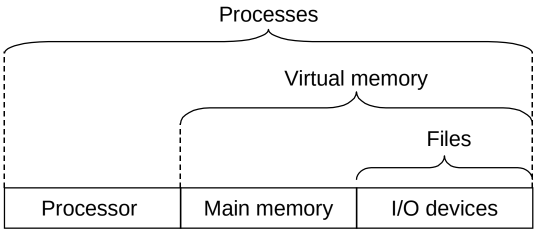


Figure 1.11: **Abstractions provided by an operating system.**

Aside: Unix and Posix.
The 1960s was an era of huge, complex operating systems, such as IBM's OS/360 and Honeywell's Multics systems. While OS/360 was one of the most successful software projects in history, Multics dragged on for years and never achieved wide-scale use. Bell Laboratories was an original partner in the Multics project, but dropped out in 1969 because of concern over the complexity of the project and the lack of progress. In reaction to their unpleasant Multics experience, a group of Bell Labs researchers — Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna — began work in 1969 on a simpler operating system for a DEC PDP-7 computer, written entirely in machine language. Many of the ideas in the new system, such as the hierarchical file system and the notion of a shell as a user-level process, were borrowed from Multics but implemented in a smaller, simpler package. In 1970, Brian Kernighan dubbed the new system "Unix" as a pun on the complexity of "Multics." The kernel was rewritten in C in 1973, and Unix was announced to the outside world in 1974 [66].

Because Bell Labs made the source code available to schools with generous terms, Unix developed a large following at universities. The most influential work was done at the University of California at Berkeley in the late 1970s and early 1980s, with Berkeley researchers adding virtual memory and the Internet protocols in a series of releases called Unix 4.xBSD (Berkeley Software Distribution). Concurrently, Bell Labs was releasing their own versions, which become known as System V Unix. Versions from other vendors, such as the Sun Microsystems Solaris system, were derived from these original BSD and System V versions.

Trouble arose in the mid 1980s as Unix vendors tried to differentiate themselves by adding new and often incompatible features. To combat this trend, IEEE (Institute for Electrical and Electronics Engineers) sponsored an effort to standardize Unix, later dubbed "Posix" by Richard Stallman. The result was a family of standards, known as the Posix standards, that cover such issues as the C language interface for Unix system calls, shell programs and utilities, threads, and network programming. As more systems comply more fully with the Posix standards, the differences between Unix versions are gradually disappearing. **End Aside.**

1.7.1 Processes

When a program such as `hello` runs on a modern system, the operating system provides the illusion that the program is the only one running on the system. The program appears to have exclusive use of both the processor, main memory, and I/O devices. The processor appears to execute the instructions in the program, one after the other, without interruption. And the code and data of the program appear to be the only objects in the system's memory. These illusions are provided by the notion of a process, one of the most important and successful ideas in computer science.

A *process* is the operating system's abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware. By *concurrently*, we mean that the instructions of one process are interleaved with the instructions of another process. The operating system performs this interleaving with a mechanism known as *context switching*.

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the *context*, includes information such as the current values of the PC, the register file, and the contents of main memory. At any point in time, exactly one process is running on the system. When the operating system decides to transfer control from the current process to a some new process, it performs a *context switch* by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example `hello` scenario.

There are two concurrent processes in our example scenario: the shell process and the `hello` process. Initially, the shell process is running alone, waiting for input on the command line. When we ask it to run the `hello` program, the shell carries out our request by invoking a special function known as a *system*

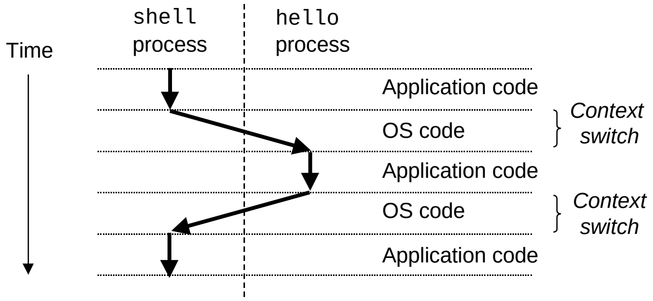


Figure 1.12: **Process context switching.**

call that passes control to the operating system. The operating system saves the shell's context, creates a new `hello` process and its context, and then passes control to the new `hello` process. After `hello` terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command line input.

Implementing the process abstraction requires close cooperation between both the low-level hardware and the operating system software. We will explore how this works, and how applications can create and control their own processes, in Chapter 8.

One of the implications of the process abstraction is that by interleaving different processes, it distorts the notion of time, making it difficult for programmers to obtain accurate and repeatable measurements of running time. Chapter 9 discusses the various notions of time in a modern system and describes techniques for obtaining accurate measurements.

1.7.2 Threads

Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called *threads*, each running in the context of the process and sharing the same code and global data. Threads are an increasingly important programming model because of the requirement for concurrency in network servers, because it is easier to share data between multiple threads than between multiple processes, and because threads are typically more efficient than processes. You will learn the basic concepts of concurrency, including threading, in Chapter 13.

1.7.3 Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its *virtual address space*. The virtual address space for Linux processes is shown in Figure 1.13. (Other Unix systems use a similar layout.) In Linux, the topmost one-fourth of the address space is reserved for code and data in the operating system that is common to all processes. The bottommost three-quarters of the address space holds the code and data defined by the user's process. Note that addresses in the figure increase from bottom to the top.

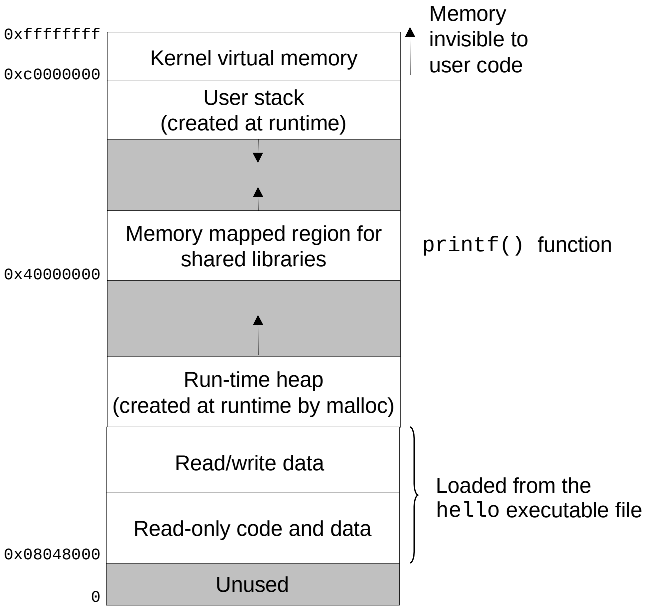


Figure 1.13: **Process virtual address space.**

The virtual address space seen by each process consists of a number of well-defined areas, each with a specific purpose. You will learn more about these areas later in the book, but it will be helpful to look briefly at each, starting with the lowest addresses and working our way up:

- *Program code and data.* Code begins at the same fixed address, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file, in our case the `hello` executable. You will learn more about this part of the address space when we study linking and loading in Chapter 7.

- *Heap.* The code and data areas are followed immediately by the run-time *heap*. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at run time as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in Chapter 10.

- *Shared libraries.* Near the middle of the address space is an area that holds the code and data for *shared libraries* such as the C standard library and the math library. The notion of a shared library is a powerful, but somewhat difficult concept. You will learn how they work when we study dynamic linking in Chapter 7.

- *Stack.* At the top of the user's virtual address space is the *user stack* that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. You will learn how the compiler uses the stack in Chapter 3.

- Kernel virtual memory. The *kernel* is the part of the operating system that is always resident in memory. The top 1/4 of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code.

For virtual memory to work, a sophisticated interaction is required between the hardware and the operating system software, including a hardware translation of every address generated by the processor. The basic idea is to store the contents of a process's virtual memory on disk, and then use the main memory as a cache for the disk. Chapter 10 explains how this works and why it is so important to the operation of modern systems.

1.7.4 Files

A *file* is a sequence of bytes, nothing more and nothing less. Every I/O device, including disks, keyboards, displays, and even networks, is modeled as a file. All input and output in the system is performed by reading and writing files, using a small set of system calls known as *Unix I/O*.

This simple and elegant notion of a file is nonetheless very powerful because it provides applications with a uniform view of all of the varied I/O devices that might be contained in the system. For example, application programmers who manipulate the contents of a disk file are blissfully unaware of the specific disk technology. Further, the same program will run on different systems that use different disk technologies. You will learn about Unix I/O in Chapter 11.

Aside: The Linux project.
In August, 1991, a Finnish graduate student named Linus Torvalds modestly announced a new Unix-like operating system kernel:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Date: 25 Aug 91 20:57:08 GMT

Hello everybody out there using minix -
I’m doing a (free) operating system (just a hobby, won’t be big and
professional like gnu) for 386(486) AT clones. This has been brewing
since April, and is starting to get ready. I’d like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I’ve currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I’ll get something practical within a few months, and
I’d like to know what features most people would want. Any suggestions
are welcome, but I won’t promise I’ll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)
```

The rest, as they say, is history. Linux has evolved into a technical and cultural phenomenon. By combining forces with the GNU project, the Linux project has developed a complete, Posix-compliant version of the Unix operating system, including the kernel and all of the supporting infrastructure. Linux is available on a wide array of computers, from hand-held devices to mainframe computers. A group at IBM has even ported Linux to a wristwatch! **End Aside.**

1.8 Systems Communicate With Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks. From the point of view of an individual system, the network can be viewed as just another I/O device, as shown in Figure 1.14. When the system copies a sequence of bytes from main memory to the network adapter, the data flows across the network to another machine, instead of, say, to a local disk drive. Similarly, the system can read data sent from other machines and copy this data to its main memory.

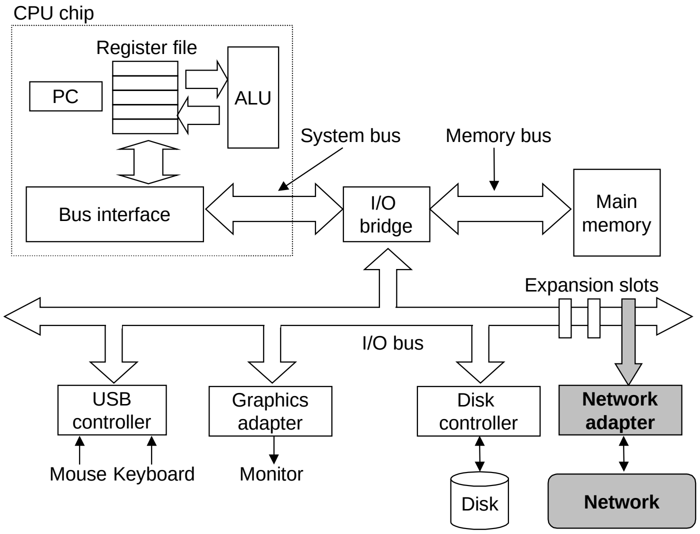


Figure 1.14: **A network is another I/O device.**

With the advent of global networks such as the Internet, copying information from one machine to another has become one of the most important uses of computer systems. For example, applications such as email, instant messaging, the World Wide Web, FTP, and telnet are all based on the ability to copy information over a network.

Returning to our `hello` example, we could use the familiar telnet application to run `hello` on a remote machine. Suppose we use a telnet *client* running on our local machine to connect to a telnet *server* on a remote machine. After we log in to the remote machine and run a shell, the remote shell is waiting to receive an input command. From this point, running the `hello` program remotely involves the five basic steps shown in Figure 1.15.

After we type the ”`hello`” string to the telnet client and hit the `enter` key, the client sends the string to the telnet server. After the telnet server receives the string from the network, it passes it along to the remote shell program. Next, the remote shell runs the `hello` program, and passes the output line back to the telnet server. Finally, the telnet server forwards the output string across the network to the telnet client, which prints the output string on our local terminal.

This type of exchange between clients and servers is typical of all network applications. In Chapter 12 you will learn how to build network applications, and apply this knowledge to build a simple Web server.

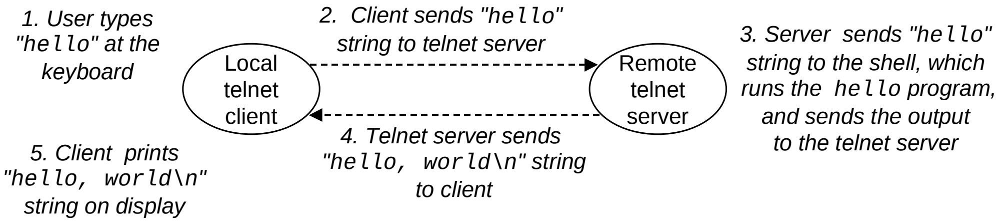

1. User types "hello" at the keyboard
Local telnet client
2. Client sends "hello" string to telnet server
Remote telnet server
3. Server sends "hello" string to the shell, which runs the hello program, and sends the output to the telnet server
4. Telnet server sends "hello, world\n" string to client
5. Client prints "hello, world\n" string on display

Figure 1.15: **Using telnet to run `hello` remotely over a network.**

1.9 The Next Step

This concludes our initial whirlwind tour of systems. An important idea to take away from this discussion is that a system is more than just hardware. It is a collection of intertwined hardware and systems software that must cooperate in order to achieve the ultimate goal of running application programs. The rest of this book will expand on this theme.

1.10 Summary

A computer system consists of hardware and systems software that cooperate to run application programs. Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on the context. Programs are translated by other programs into different forms, beginning as ASCII text and then translated by compilers and linkers into binary executable files.

Processors read and interpret binary instructions that are stored in main memory. Since computers spend most of their time copying data between memory, I/O devices, and the CPU registers, the storage devices in a system are arranged in a hierarchy, with the CPU registers at the top, followed by multiple levels of hardware cache memories, DRAM main memory, and disk storage. Storage devices that are higher in the hierarchy are faster and more costly per bit than those lower in the hierarchy. Storage devices that are higher in the hierarchy serve as caches for devices that are lower in the hierarchy. Programmers can optimize the performance of their C programs by understanding and exploiting the memory hierarchy.

The operating system kernel serves an intermediary between the application and the hardware. It provides three fundamental abstractions: (1) Files are abstractions for I/O devices. (2) Virtual memory is an abstraction for both main memory and disks. (3) Processes are abstractions for the processor, main memory, and I/O devices.

Finally, networks provide ways for computer systems to communicate with one another. From the viewpoint of a particular system, the network is just another I/O device.

Bibliographic Notes

Ritchie has written interesting first hand accounts of the early days of C and Unix [64, 65]. Ritchie and Thompson presented the first published account of Unix [66]. Silberschatz and Gavin [72] provide a comprehensive history of the different flavors of Unix. The GNU (`www.gnu.org`) and Linux (`www.linux.org`) Web pages have loads of current and historical information. Unfortunately, the Posix standards are not available online. They must be ordered for a fee from IEEE (`standards.ieee.org`).

Part I

Program Structure and Execution

Our exploration of computer systems starts by studying the computer itself, comprising a processor and a memory subsystem. At the core, we require ways to represent basic data types, such as approximations to integer and real arithmetic. From there we can consider how machine-level instructions manipulate this data and how a compiler translates C programs into these instructions. Next, we study several methods of implementing a processor to gain a better understanding of how hardware resources are used to execute instructions. Once we understand compilers and machine-level code, we can examine how to maximize program performance by writing source code that will compile efficiently. We conclude with the design of the memory subsystem, one of the most complex components of a modern computer system.

This part of the book will give you a deep understanding of how application programs are represented and executed. You will gain skills that help you write programs that are reliable and that make the best use of the computing resources.

Chapter 2

Representing and Manipulating Information

Modern computers store and process information represented as two-valued signals. These lowly binary digits, or *bits*, form the basis of the digital revolution. The familiar decimal, or base-10, representation has been in use for over 1000 years, having been developed in India, improved by Arab mathematicians in the 12th century, and brought to the West in the 13th century by the Italian mathematician Leonardo Pisano, better known as Fibonacci. Using decimal notation is natural for ten-fingered humans, but binary values work better when building machines that store and process information. Two-valued signals can readily be represented, stored, and transmitted, for example, as the presence or absence of a hole in a punched card, as a high or low voltage on a wire, or as a magnetic domain oriented clockwise or counterclockwise. The electronic circuitry for storing and performing computations on two-valued signals is very simple and reliable, enabling manufacturers to integrate millions of such circuits on a single silicon chip.

In isolation, a single bit is not very useful. When we group bits together and apply some *interpretation* that gives meaning to the different possible bit patterns, however, we can represent the elements of any finite set. For example, using a binary number system, we can use groups of bits to encode nonnegative numbers. By using a standard character code, we can encode the letters and symbols in a document. We cover both of these encodings in this chapter, as well as encodings to represent negative numbers and to approximate real numbers.

We consider the three most important encodings of numbers. *Unsigned* encodings are based on traditional binary notation, representing numbers greater than or equal to 0. *Two's-complement* encodings are the most common way to represent signed integers, that is, numbers that may be either positive or negative. *Floating-point* encodings are a base-two version of scientific notation for representing real numbers. Computers implement arithmetic operations, such as addition and multiplication, with these different representations, similar to the corresponding operations on integers and real numbers.

Computer representations use a limited number of bits to encode a number, and hence some operations can *overflow* when the results are too large to be represented. This can lead to some surprising results. For example, on most of today's computers, computing the expression

```
200 * 300 * 400 * 500
```

yields $-884{,}901{,}888$. This runs counter to the properties of integer arithmetic—computing the product of a set of positive numbers has yielded a negative result.

On the other hand, integer computer arithmetic satisfies many of the familiar properties of true integer arithmetic. For example, multiplication is associative and commutative, so that computing any of the following C expressions yields $-884{,}901{,}888$:

```
(500  *  400) * (300 * 200)
((500 *  400) * 300) * 200
((200 *  500) * 300) * 400
400   * (200 * (300 * 500))
```

The computer might not generate the expected result, but at least it is consistent!

Floating-point arithmetic has altogether different mathematical properties. The product of a set of positive numbers will always be positive, although overflow will yield the special value $+\infty$. On the other hand, floating-point arithmetic is not associative due to the finite precision of the representation. For example, the C expression `(3.14+1e20)-1e20` will evaluate to `0.0` on most machines, while `3.14+(1e20-1e20)` will evaluate to `3.14`.

By studying the actual number representations, we can understand the ranges of values that can be represented and the properties of the different arithmetic operations. This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler.

Computers use several different binary representations to encode numeric values. You will need to be familiar with these representations as you progress into machine-level programming in Chapter 3. We describe these encodings in this chapter and give you some practice reasoning about number representations.

We derive several ways to perform arithmetic operations by directly manipulating the bit-level representations of numbers. Understanding these techniques will be important for understanding the machine-level code generated when compiling arithmetic expressions.

Our treatment of this material is very mathematical. We start with the basic definitions of the encodings and then derive such properties as the range of representable numbers, their bit-level representations, and the properties of the arithmetic operations. We believe it is important for you to examine this material from such an abstract viewpoint, because programmers need to have a solid understanding of how computer arithmetic relates to the more familiar integer and real arithmetic. Although it may appear intimidating, the mathematical treatment requires just an understanding of basic algebra. We recommend you work the practice problems as a way to solidify the connection between the formal treatment and some real-life examples.

Aside: How to read this chapter.
If you find equations and formulas daunting, do not let that stop you from getting the most out of this chapter! We provide full derivations of mathematical ideas for completeness, but the best way to read this material is often to skip over the derivation on your initial reading. Instead, try working out a few simple examples (for example, the practice problems) to build your intuition, and then see how the mathematical derivation reinforces your intuition.
End Aside.

The C++ programming language is built upon C, using the exact same numeric representations and operations. Everything said in this chapter about C also holds for C++. The Java language definition, on the other hand, created a new set of standards for numeric representations and operations. Whereas the C standard is designed to allow a wide range of implementations, the Java standard is quite specific on the formats and encodings of data. We highlight the representations and operations supported by Java at several places in the chapter.

2.1 Information Storage

Rather than accessing individual bits in a memory, most computers use blocks of eight bits, or *bytes*, as the smallest addressable unit of memory. A machine-level program views memory as a very large array of bytes, referred to as *virtual memory*. Every byte of memory is identified by a unique number, known as its *address*, and the set of all possible addresses is known as the *virtual address space*. As indicated by its name, this virtual address space is just a conceptual image presented to the machine-level program. The actual implementation (presented in Chapter 10) uses a combination of random-access memory (RAM), disk storage, special hardware, and operating system software to provide the program with what appears to be a monolithic byte array.

One task of a compiler and the run-time system is to subdivide this memory space into more manageable units to store the different *program objects*, that is, program data, instructions, and control information. Various mechanisms are used to allocate and manage the storage for different parts of the program. This management is all performed within the virtual address space. For example, the value of a pointer in C—whether it points to an integer, a structure, or some other program unit—is the virtual address of the first byte of some block of storage. The C compiler also associates *type* information with each pointer, so that it can generate different machine-level code to access the value stored at the location designated by the pointer depending on the type of that value. Although the C compiler maintains this type information, the actual machine-level program it generates has no information about data types. It simply treats each program object as a block of bytes, and the program itself as a sequence of bytes.

New to C?: The role of pointers in C.
Pointers are a central feature of C. They provide the mechanism for referencing elements of data structures, including arrays. Just like a variable, a pointer has two aspects: its *value* and its *type*. The value indicates the location of some object, while its type indicates what kind of object (e.g., integer or floating-point number) is stored at that location. **End.**

2.1.1 Hexadecimal Notation

A single byte consists of eight bits. In binary notation, its value ranges from 00000000_2 to 11111111_2. When viewed as a decimal integer, its value ranges from 0_{10} to 255_{10}. Neither notation is very convenient for describing bit patterns. Binary notation is too verbose, while with decimal notation, it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base-16, or *hexadecimal* numbers. Hexadecimal (or simply “Hex”) uses digits ‘0’ through ‘9’, along with characters ‘A’ through ‘F’ to represent 16 possible values. Figure 2.1 shows the decimal and binary values associated with the 16 hexadecimal digits. Written in hexadecimal, the value of a single byte can range from 00_{16} to FF_{16}.

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.1: **Hexadecimal notation.** Each Hex digit encodes one of 16 values.

In C, numeric constants starting with `0x` or `0X` are interpreted as being in hexadecimal. The characters ‘A’ through ‘F’ may be written in either upper or lower case. For example, we could write the number FA1D37B_{16} as `0xFA1D37B`, as `0xfa1d37b`, or even mixing upper and lower case, e.g., `0xFa1D37b`. We will use the C notation for representing hexadecimal values in this book.

A common task in working with machine-level programs is to manually convert between decimal, binary, and hexadecimal representations of bit patterns. Converting between binary and hexadecimal is straightforward, since it can be performed one hexadecimal digit at a time. Digits can be converted by referring to a chart such as that shown in Figure 2.1. One simple trick for doing the conversion in your head is to memorize the decimal equivalents of hex digits `A`, `C`, and `F`. The hex values `B`, `D`, and `E` can be translated to decimal by computing their values relative to the first three.

For example, suppose you are given the number `0x173A4C`. You can convert this to binary format by expanding each hexadecimal digit, as follows:

Hexadecimal	`1`	`7`	`3`	`A`	`4`	`C`
Binary	0001	0111	0011	1010	0100	1100

This gives the binary representation 000101110011101001001100.

Conversely, given a binary number 1111001010110110110011 you convert it to hexadecimal by first splitting it into groups of four bits each. Note, however, that if the total number of bits is not a multiple of four, you should make the *leftmost* group be the one with fewer than four bits, effectively padding the number with leading 0s. Then you translate each group of four bits into the corresponding hexadecimal digit:

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	`3`	`C`	`A`	`D`	`B`	`3`

Practice Problem 2.1:

Perform the following number conversions:

A. `0x8F7A93` to binary.

B. Binary 1011011110011100 to hexadecimal.

C. `0xC4E5D` to binary.

D. Binary 1101011011011111100110 to hexadecimal.

When a value x is a power of two, that is, $x = 2^n$ for some n, we can readily write x in hexadecimal form by remembering that the binary representation of x is simply 1 followed by n 0s. The hexadecimal digit `0` represents four binary 0s. So, for n written in the form $i + 4j$, where $0 \leq i \leq 3$, we can write x with a leading hex digit of `1` ($i = 0$), `2` ($i = 1$), `4` ($i = 2$), or `8` ($i = 3$), followed by j hexadecimal `0`s. As an example, for $x = 2048 = 2^{11}$, we have $n = 11 = 3 + 4 \cdot 2$, giving hexadecimal representation `0x800`.

Practice Problem 2.2:

Fill in the blank entries in the following table, giving the decimal and hexadecimal representations of different powers of 2:

n	2^n (Decimal)	2^n (Hexadecimal)
11	2048	`0x800`
7		
	8192	
		`0x2000`
16		
	256	
		`0x20`

Converting between decimal and hexadecimal representations requires using multiplication or division to handle the general case. To convert a decimal number x to hexadecimal, we can repeatedly divide x by 16, giving a quotient q and a remainder r, such that $x = q \cdot 16 + r$. We then use the hexadecimal digit representing r as the least significant digit and generate the remaining digits by repeating the process on q. As an example, consider the conversion of decimal 314156:

$$\begin{array}{rcll}
314156 & = & 19634 \cdot 16 + 12 & (\texttt{C}) \\
19634 & = & 1227 \cdot 16 + 2 & (\texttt{2}) \\
1227 & = & 76 \cdot 16 + 11 & (\texttt{B}) \\
76 & = & 4 \cdot 16 + 12 & (\texttt{C}) \\
4 & = & 0 \cdot 16 + 4 & (\texttt{4})
\end{array}$$

From this we can read off the hexadecimal representation as `0x4CB2C`.

Conversely, to convert a hexadecimal number to decimal, we can multiply each of the hexadecimal digits by the appropriate power of 16. For example, given the number `0x7AF`, we compute its decimal equivalent as $7 \cdot 16^2 + 10 \cdot 16 + 15 = 7 \cdot 256 + 10 \cdot 16 + 15 = 1792 + 160 + 15 = 1967$.

Practice Problem 2.3:

A single byte can be represented by two hexadecimal digits. Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns:

Decimal	Binary	Hexadecimal
0	00000000	00
55		
136		
243		
	01010010	
	10101100	
	11100111	
		A7
		3E
		BC

Aside: Converting between decimal and hexadecimal.
For converting larger values between decimal and hexadecimal, it is best to let a computer or calculator do the work. For example, the following script in the Perl language converts a list of numbers from decimal to hexadecimal:

code/../bin/d2h

```
#!/usr/local/bin/perl
# Convert list of decimal numbers into hex

for ($i = 0; $i < @ARGV; $i++) {
    printf("%d\t= 0x%x\n", $ARGV[$i], $ARGV[$i]);
}
```

code/../bin/d2h

Once this file has been set to be executable, the command:

```
unix> ./d2h 100 500 751
```

yields output:

```
100 = 0x64
500 = 0x1f4
751 = 0x2ef
```

Similarly, the following script converts from hexadecimal to decimal:

code/../bin/h2d

```
#!/usr/local/bin/perl
# Convert list of hex numbers into decimal

for ($i = 0; $i < @ARGV; $i++) {
  $val = hex($ARGV[$i]);
  printf("0x%x = %d\n", $val, $val);
}
```

code/../bin/h2d **End Aside.**

C declaration	Typical 32-bit	Compaq Alpha
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`char *`	4	8
`float`	4	4
`double`	8	8

Figure 2.2: **Sizes (in bytes) of C numeric data types.** The number of bytes allocated varies with machine and compiler.

Practice Problem 2.4:

Without converting the numbers to decimal or binary, try to solve the following arithmetic problems, giving the answers in hexadecimal. Hint: just modify the methods you use for performing decimal addition and subtraction to use base 16.

A. `0x502c` $+$ `0x8` $=$

B. `0x502c` $-$ `0x30` $=$

C. `0x502c` $+ 64 =$

D. `0x50da` $-$ `0x502c`

2.1.2 Words

Every computer has a *word size*, indicating the nominal size of integer and pointer data. Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is the maximum size of the virtual address space. That is, for a machine with an n-bit word size, the virtual addresses can range from 0 to $2^n - 1$, giving the program access to at most 2^n bytes.

Most computers today have a 32-bit word size. This limits the virtual address space to 4 gigabytes (written 4 GB), that is, just over 4×10^9 bytes. Although this is ample space for most applications, we have reached the point where many large-scale scientific and database applications require larger amounts of storage. Consequently, high-end machines with 64-bit word sizes are becoming increasingly commonplace as storage costs decrease.

2.1.3 Data Sizes

Computers and compilers support multiple data formats using different ways to encode data, such as integers and floating point, as well as different lengths. For example, many machines have instructions for manipulating single bytes, as well as integers represented as two-, four-, and eight-byte quantities. They also support floating-point numbers represented as four and eight-byte quantities.

The C language supports multiple data formats for both integer and floating-point data. The C data type `char` represents a single byte. Although the name "char" derives from the fact that it is used to store a single character in a text string, it can also be used to store integer values. The C data type `int` can also be prefixed by the qualifiers `long` and `short`, providing integer representations of various sizes. Figure 2.2 shows the number of bytes allocated for various C data types. The exact number depends on both the machine and the compiler. We show two representative cases: a typical 32-bit machine, and the Compaq Alpha architecture, a 64-bit machine targeting high end applications. Most 32-bit machines use the allocations indicated as "typical." Observe that "short" integers have two-byte allocations, while an unqualified `int` is 4 bytes. A "long" integer uses the full word size of the machine.

Figure 2.2 also shows that a pointer (e.g., a variable declared as being of type "`char *`") uses the full word size of the machine. Most machines also support two different floating-point formats: single precision, declared in C as `float`, and double precision, declared in C as `double`. These formats use four and eight bytes, respectively.

New to C?: Declaring pointers.
For any data type T, the declaration

```
T *p;
```

indicates that `p` is a pointer variable, pointing to an object of type T. For example

```
char *p;
```

is the declaration of a pointer to an object of type `char`. **End.**

Programmers should strive to make their programs portable across different machines and compilers. One aspect of portability is to make the program insensitive to the exact sizes of the different data types. The C standard sets lower bounds on the numeric ranges of the different data types, as will be covered later, but there are no upper bounds. Since 32-bit machines have been the standard for the last 20 years, many programs have been written assuming the allocations listed as "typical 32-bit" in Figure 2.2. Given the increasing prominence of 64-bit machines in the near future, many hidden word size dependencies will show up as bugs in migrating these programs to new machines. For example, many programmers assume that a program object declared as type `int` can be used to store a pointer. This works fine for most 32-bit machines but leads to problems on an Alpha.

2.1.4 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what will be the address of the object, and how will we order the bytes in memory. In virtually all machines, a multibyte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used. For example, suppose a variable `x` of type `int` has address `0x100`, that is, the value of the address expression `&x` is `0x100`. Then the four bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.

For ordering the bytes representing an object, there are two common conventions. Consider a w-bit integer having a bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit, and x_0 is the least. Assuming w is a multiple of eight, these bits can be grouped as bytes, with the most significant byte having bits $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$, the least significant byte having bits $[x_7, x_6, \ldots, x_0]$, and the other bytes having bits from the middle. Some machines choose to store the object in memory ordered from least significant byte to most, while other machines store them from most to least. The former convention—where the least significant byte comes first—is referred to as *little endian*. This convention is followed by most machines from the former Digital Equipment Corporation (now part of Compaq Corporation), as well as by Intel. The latter convention—where the most significant byte comes first—is referred to as *big endian*. This convention is followed by most machines from IBM, Motorola, and Sun Microsystems. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, personal computers manufactured by IBM use Intel-compatible processors and hence are little endian. Many microprocessor chips, including Alpha and the PowerPC by Motorola, can be run in either mode, with the byte ordering convention determined when the chip is powered up.

Continuing our earlier example, suppose the variable `x` of type `int` and at address `0x100` has a hexadecimal value of `0x01234567`. The ordering of the bytes within the address range `0x100` through `0x103` depends on the type of machine:

Big endian

	0x100	0x101	0x102	0x103	
···	01	23	45	67	···

Little endian

	0x100	0x101	0x102	0x103	
···	67	45	23	01	···

Note that in the word `0x01234567` the high-order byte has hexadecimal value `0x01`, while the low-order byte has value `0x67`.

People get surprisingly emotional about which byte ordering is the proper one. In fact, the terms "little endian" and "big endian" come from the book *Gulliver's Travels* by Jonathan Swift, where two warring factions could not agree by which end a soft-boiled egg should be opened—the little end or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, and hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

Aside: Origin of "endian."
Here is how Jonathan Swift, writing in 1726, described the history of the controversy between big and little endians:

> ...Lilliput and Blefuscu ...have, as I was going to tell you, been engaged in a most obstinate war for six-and-thirty moons past. It began upon the following occasion. It is allowed on all hands, that the primitive way of breaking eggs, before we eat them, was upon the larger end; but his present majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the emperor his father published an edict, commanding all his subjects, upon great penalties, to break the smaller end of their eggs. The people so highly resented this law, that our histories tell us, there have been six rebellions raised on that account; wherein one emperor lost his life, and another his crown. These civil commotions were

> constantly fomented by the monarchs of Blefuscu; and when they were quelled, the exiles always fled for refuge to that empire. It is computed that eleven thousand persons have at several times suffered death, rather than submit to break their eggs at the smaller end. Many hundred large volumes have been published upon this controversy: but the books of the Big-endians have been long forbidden, and the whole party rendered incapable by law of holding employments.

In his day, Swift was satirizing the continued conflicts between England (Lilliput) and France (Blefuscu). Danny Cohen, an early pioneer in networking protocols, first applied these terms to refer to byte ordering [17], and the terminology has been widely adopted. **End Aside.**

For most application programmers, the byte orderings used by their machines are totally invisible. Programs compiled for either class of machine give identical results. At times, however, byte ordering becomes an issue. The first is when binary data is communicated over a network between different machines. A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice-versa, leading to the bytes within the words being in reverse order for the receiving program. To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation. We will see examples of these conversions in Chapter 12.

A second case where byte ordering becomes important is when looking at the byte sequences representing integer data. This occurs often when inspecting machine-level programs. As an example, the following line occurs in a file that gives a text representation of the machine-level code for an Intel processor:

```
 80483bd:  01 05 64 94 04 08       add    %eax,0x8049464
```

This line was generated by a *disassembler*, a tool that determines the instruction sequence represented by an executable program file. We will learn more about these tools and how to interpret lines such as this in the next chapter. For now, we simply note that this line states that the hexadecimal byte sequence `01 05 64 94 04 08` is the byte-level representation of an instruction that adds `0x8049464` to some program value. If we take the the final four bytes of the sequence: `64 94 04 08`, and write them in reverse order, we have `08 04 94 64`. Dropping the leading 0, we have the value `0x8049464`, the numeric value written on the right. Having bytes appear in reverse order is a common occurrence when reading machine-level program representations generated for little-endian machines such as this one. The natural way to write a byte sequence is to have the lowest numbered byte on the left and the highest on the right, but this is contrary to the normal way of writing numbers with the most significant digit on the left and the least on the right.

A third case where byte ordering becomes visible is when programs are written that circumvent the normal type system. In the C language, this can be done using a *cast* to allow an object to be referenced according to a different data type from which it was created. Such coding tricks are strongly discouraged for most application programming, but they can be quite useful and even necessary for system-level programming.

Figure 2.3 shows C code that uses casting to access and print the byte representations of different program objects. We use `typedef` to define data type `byte_pointer` as a pointer to an object of type "`unsigned char`." Such a byte pointer references a sequence of bytes where each byte is considered to be a nonnegative integer. The first routine `show_bytes` is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count. It prints the individual bytes in hexadecimal. The C formatting directive "`%.2x`" indicates that an integer should be printed in hexadecimal with at least two digits.

code/data/show-bytes.c

```
#include <stdio.h>

typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf(" %.2x", start[i]);
    printf("\n");
}

void show_int(int x)
{
    show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x)
{
    show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x)
{
    show_bytes((byte_pointer) &x, sizeof(void *));
}
```

code/data/show-bytes.c

Figure 2.3: **Code to print the byte representation of program objects.** This code uses casting to circumvent the type system. Similar functions are easily defined for other data types.

New to C?: Naming data types with `typedef`.
The `typedef` declaration in C provides a way of giving a name to a data type. This can be a great help in improving code readability, since deeply nested type declarations can be difficult to decipher.

The syntax for `typedef` is exactly like that of declaring a variable, except that it uses a type name rather than a variable name. Thus, the declaration of `byte_pointer` in Figure 2.3 has the same form as would the declaration of a variable to type “`unsigned char`.”

For example, the declaration:

```
typedef int *int_pointer;
int_pointer ip;
```

defines type “`int_pointer`” to be a pointer to an `int`, and declares a variable `ip` of this type. Alternatively, we could declare this variable directly as:

```
int *ip;
```

End.

New to C?: Formatted printing with `printf`.
The `printf` function (along with its cousins `fprintf` and `sprintf`) provides a way to print information with considerable control over the formatting details. The first argument is a *format string*, while any remaining arguments are values to be printed. Within the format string, each character sequence starting with ‘`%`’ indicates how to format the next argument. Typical examples include ‘`%d`’ to print a decimal integer and ‘`%f`’ to print a floating-point number, and ‘`%c`’ to print a character having the character code given by the argument. **End.**

New to C?: Pointers and arrays.
In function `show_bytes` (Figure 2.3) we see the close connection between pointers and arrays, as will be discussed in detail in Section 3.8. We see that this function has an argument `start` of type `byte_pointer` (which has been defined to be a pointer to `unsigned char`,) but we see the array reference `start[i]` on line 9. In C, we can use reference a pointer with array notation, and we can reference arrays with pointer notation. In this example, the reference `start[i]` indicates that we want to read the byte that is `i` positions beyond the location pointed to by `start`. **End.**

Procedures `show_int`, `show_float`, and `show_pointer` demonstrate how to use procedure `show_bytes` to print the byte representations of C program objects of type `int`, `float`, and `void *`, respectively. Observe that they simply pass `show_bytes` a pointer `&x` to their argument `x`, casting the pointer to be of type “`unsigned char *`.” This cast indicates to the compiler that the program should consider the pointer to be to a sequence of bytes rather than to an object of the original data type. This pointer will then be to the lowest byte address used by the object.

New to C?: Pointer creation and dereferencing.
In lines 15, 20, and 25 of Figure 2.3 we see uses of two operations that are unique to C and C++. The C “address of” operator & creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding variable `x`. The type of this pointer depends on the type of `x`, and hence these three pointers are of type `int *`, `float *`, and `void **`, respectively. (Data type `void *` is a special kind of pointer with no associated type information.)

The cast operator converts from one data type to another. Thus, the cast `(byte_pointer) &x` indicates that whatever type the pointer `&x` had before, it now is a pointer to data of type `unsigned char`. **End.**

code/data/show-bytes.c

```
void test_show_bytes(int val)
{
    int ival = val;
    float fval = (float) ival;
    int *pval = &ival;
    show_int(ival);
    show_float(fval);
    show_pointer(pval);
}
```

code/data/show-bytes.c

Figure 2.4: **Byte representation examples.** This code prints the byte representations of sample data objects.

These procedures use the C operator `sizeof` to determine the number of bytes used by the object. In general, the expression `sizeof(`T`)` returns the number of bytes required to store an object of type T. Using `sizeof` rather than a fixed value is one step toward writing code that is portable across different machine types.

We ran the code shown in Figure 2.4 on several different machines, giving the results shown in Figure 2.5. The following machines were used:

Linux: Intel Pentium II running Linux.

NT: Intel Pentium II running Windows-NT.

Sun: Sun Microsystems UltraSPARC running Solaris.

Alpha: Compaq Alpha 21164 running Tru64 Unix.

Our sample integer argument 12,345 has hexadecimal representation `0x00003039`. For the `int` data, we get identical results for all machines, except for the byte ordering. In particular, we can see that the least significant byte value of `0x39` is printed first for Linux, NT, and Alpha, indicating little-endian machines, and last for Sun, indicating a big-endian machine. Similarly, the bytes of the `float` data are identical, except for the byte ordering. On the other hand, the pointer values are completely different. The different machine/operating system configurations use different conventions for storage allocation. One feature to note is that the Linux and Sun machines use four-byte addresses, while the Alpha uses eight-byte addresses.

Observe that although the floating point and the integer data both encode the numeric value 12,345, they have very different byte patterns: `0x00003039` for the integer, and `0x4640E400` for floating point. In general, these two formats use different encoding schemes. If we expand these hexadecimal patterns into binary form and shift them appropriately, we find a sequence of 13 matching bits, indicated by a sequence of asterisks, as follows:

```
   0   0   0   0   3   0   3   9
00000000000000000011000000111001
```

Machine	Value	Type	Bytes (hex)
Linux	12,345	`int`	`39 30 00 00`
NT	12,345	`int`	`39 30 00 00`
Sun	12,345	`int`	`00 00 30 39`
Alpha	12,345	`int`	`39 30 00 00`
Linux	$12,345.0$	`float`	`00 e4 40 46`
NT	$12,345.0$	`float`	`00 e4 40 46`
Sun	$12,345.0$	`float`	`46 40 e4 00`
Alpha	$12,345.0$	`float`	`00 e4 40 46`
Linux	`&ival`	`int *`	`3c fa ff bf`
NT	`&ival`	`int *`	`1c ff 44 02`
Sun	`&ival`	`int *`	`ef ff fc e4`
Alpha	`&ival`	`int *`	`80 fc ff 1f 01 00 00 00`

Figure 2.5: **Byte representations of different data values.** Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine-dependent.

```
         *************
   4   6   4   0   E   4   0   0
01000110010000001110010000000000
```

This is not coincidental. We will return to this example when we study floating-point formats.

Practice Problem 2.5:

Consider the following three calls to `show_bytes`:

```
int val = 0x12345678;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate which of the following values would be printed by each call on a little-endian machine and on a big-endian machine:

A. Little endian: Big endian:

B. Little endian: Big endian:

C. Little endian: Big endian:

Practice Problem 2.6:

Using `show_int` and `show_float`, we determine that the integer 3490593 has hexadecimal representation `0x00354321`, while the floating-point number 3490593.0 has hexadecimal representation representation `0x4A550C84`.

}

A. Write the binary representations of these two hexadecimal values.

B. Shift these two strings relative to one another to maximize the number of matching bits.

C. How many bits match? What parts of the strings do not match?

2.1.5 Representing Strings

A string in C is encoded by an array of characters terminated by the null (having value 0) character. Each character is represented by some standard encoding, with the most common being the ASCII character code. Thus, if we run our routine `show_bytes` with arguments `"12345"` and `6` (to include the terminating character), we get the result `31 32 33 34 35 00`. Observe that the ASCII code for decimal digit x happens to be `0x3`x, and that the terminating byte has the hex representation `0x00`. This same result would be obtained on any system using ASCII as its character code, independent of the byte ordering and word size conventions. As a consequence, text data is more platform-independent than binary data.

Aside: Generating an ASCII table.
You can display a table showing the ASCII character code by executing the command `man ascii`. **End Aside.**

Practice Problem 2.7:

What would be printed as a result of the following call to `show_bytes`?

```
char *s = "ABCDEF";
show_bytes(s, strlen(s));
```

Note that letters '`A`' through '`Z`' have ASCII codes `0x41` through `0x5A`.

Aside: The Unicode character set.
The ASCII character set is suitable for encoding English language documents, but it does not have much in the way of special characters, such as the French 'ç.' It is wholly unsuited for encoding documents in languages such as Greek, Russian, and Chinese. Recently, the 16-bit *Unicode* character set has been adopted to support documents in all languages. This doubling of the character set representation enables a very large number of different characters to be represented. The Java programming language uses Unicode when representing character strings. Program libraries are also available for C that provide Unicode versions of the standard string functions such as `strlen` and `strcpy`. **End Aside.**

2.1.6 Representing Code

Consider the following C function:

```
int sum(int x, int y)
{
    return x + y;
}
```

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Figure 2.6: **Operations of Boolean algebra.** Binary values 1 and 0 encode logic values TRUE and FALSE, while operations ~, &, |, and ^ encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

When compiled on our sample machines, we generate machine code having the following byte representations:

Linux:	`55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`
NT:	`55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`
Sun:	`81 C3 E0 08 90 02 00 09`
Alpha:	`00 00 30 42 01 80 FA 6B`

Here we find that the instruction codings are different, except for the NT and Linux machines. Different machine types use different and incompatible instructions and encodings. The NT and Linux machines both have Intel processors and hence support the same machine-level instructions. In general, however, the structure of an executable NT program differs from a Linux program, and hence the machines are not fully binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that a program, from the perspective of the machine, is simply a sequence of bytes. The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging. We will see this more clearly when we study machine-level programming in Chapter 3.

2.1.7 Boolean Algebras and Rings

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole around 1850 and thus is known as *Boolean algebra*. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the properties of propositional logic.

There is an infinite number of different Boolean algebras, where the simplest is defined over the two-element set $\{0, 1\}$. Figure 2.6 defines several operations in this Boolean algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations, as will be discussed later. The Boolean operation ~ corresponds to the logical operation NOT, denoted in propositional logic as $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice-versa. Correspondingly, ~p equals 1 when p equals 0, and vice-versa. Boolean operation & corresponds to the logical operation AND, denoted in propositional

Shared properties

Property	Integer ring	Boolean algebra
Commutativity	$a + b = b + a$ $a \times b = b \times a$	$a \mid b = b \mid a$ $a \,\&\, b = b \,\&\, a$
Associativity	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a \mid b) \mid c = a \mid (b \mid c)$ $(a \,\&\, b) \,\&\, c = a \,\&\, (b \,\&\, c)$
Distributivity	$a \times (b + c) = (a \times b) + (a \times c)$	$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$
Identities	$a + 0 = a$ $a \times 1 = a$	$a \mid 0 = a$ $a \,\&\, 1 = a$
Annihilator	$a \times 0 = 0$	$a \,\&\, 0 = 0$
Cancellation	$-(-a) = a$	$\sim(\sim a) = a$

Unique to Rings

Inverse	$a + -a = 0$	—

Unique to Boolean Algebras

Distributivity	—	$a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$
Complement	— —	$a \mid \sim a = 1$ $a \,\&\, \sim a = 0$
Idempotency	— —	$a \,\&\, a = a$ $a \mid a = a$
Absorption	— —	$a \mid (a \,\&\, b) = a$ $a \,\&\, (a \mid b) = a$
DeMorgan's laws	— —	$\sim(a \,\&\, b) = \sim a \mid \sim b$ $\sim(a \mid b) = \sim a \,\&\, \sim b$

Figure 2.7: **Comparison of integer ring and Boolean algebra.** The two mathematical structures share many properties, but there are key differences, particularly between $-$ and $\sim$.

logic as $\wedge$. We say that $P \wedge Q$ holds when both P and Q are true. Correspondingly, $p \,\&\, q$ equals 1 only when $p = 1$ and $q = 1$. Boolean operation $\mid$ corresponds to the logical operation OR, denoted in propositional logic as $\vee$. We say that $P \vee Q$ holds when either P or Q are true. Correspondingly, $p \mid q$ equals 1 when either $p = 1$ or $q = 1$. Boolean operation $\hat{}$ corresponds to the logical operation EXCLUSIVE-OR, denoted in propositional logic as $\oplus$. We say that $P \oplus Q$ holds when either P or Q are true, but not both. Correspondingly, $p \mathbin{\hat{}} q$ equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon, who later founded the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 master's thesis, he showed that Boolean algebra could be applied to the design and analysis of networks of electromechanical relays. Although computer technology has advanced considerably since, Boolean algebra still plays a central role in the design and analysis of digital systems.

There are many parallels between integer arithmetic and Boolean algebra, as well as several important differences. In particular, the set of integers, denoted $\mathcal{Z}$, forms a mathematical structure known as a *ring*, denoted $\langle \mathcal{Z}, +, \times, -, 0, 1\rangle$, with addition serving as the *sum* operation, multiplication as the *product* operation, negation as the additive inverse, and elements 0 and 1 serving as the additive and multiplicative identities. The Boolean algebra $\langle \{0,1\}, \mid, \&, \tilde{}, 0, 1\rangle$ has similar properties. Figure 2.7 highlights properties of these two structures, showing the properties that are common to both and those that are unique to one or the other. One important difference is that $\tilde{}a$ is not an inverse for a under $\mid$.

Aside: What good is abstract algebra?
Abstract algebra involves identifying and analyzing the common properties of mathematical operations in different domains. Typically, an algebra is characterized by a set of elements, some of its key operations, and some important elements. As an example, modular arithmetic also forms a ring. For modulus n, the algebra is denoted $\langle \mathcal{Z}_n, +_n, \times_n, -_n, 0, 1\rangle$, with components defined as follows:

$$\begin{array}{rcl} \mathcal{Z}_n & = & \{0, 1, \ldots, n-1\} \\ a +_n b & = & a + b \bmod n \\ a \times_n b & = & a \times b \bmod n \\ -_n a & = & \left\{ \begin{array}{ll} 0, & a = 0 \\ n - a, & a > 0 \end{array} \right. \end{array}$$

Even though modular arithmetic yields different results from integer arithmetic, it has many of the same mathematical properties. Other well-known rings include rational and real numbers. **End Aside.**

If we replace the OR operation of Boolean algebra by the EXCLUSIVE-OR operation, and the complement operation ~ with the identity operation I—where $I(a) = a$ for all a—we have a structure $\langle \{0,1\}, \hat{}, \&, I, 0, 1\rangle$. This structure is no longer a Boolean algebra—in fact it's a ring. It can be seen to be a particularly simple form of the ring consisting of all integers $\{0, 1, \ldots, n-1\}$ with both addition and multiplication performed modulo n. In this case, we have $n = 2$. That is, the Boolean AND and EXCLUSIVE-OR operations correspond to multiplication and addition modulo 2, respectively. One curious property of this algebra is that every element is its own additive inverse: $a \mathbin{\hat{}} I(a) = a \mathbin{\hat{}} a = 0$.

Aside: Who, besides mathematicians, care about Boolean rings?
Every time you enjoy the clarity of music recorded on a CD or the quality of video recorded on a DVD, you are taking advantage of Boolean rings. These technologies rely on *error-correcting codes* to reliably retrieve the bits from a disk even when dirt and scratches are present. The mathematical basis for these error-correcting codes is a linear algebra based on Boolean rings. **End Aside.**

We can extend the four Boolean operations to also operate on bit vectors, i.e., strings of 0s and 1s of some fixed length w. We define the operations over bit vectors according their applications to the matching elements of the arguments. For example, we define $[a_{w-1}, a_{w-2}, \ldots, a_0]\&[b_{w-1}, b_{w-2}, \ldots, b_0]$ to be $[a_{w-1}\& b_{w-1}, a_{w-2} \& b_{w-2}, \ldots, a_0 \& b_0]$, and similarly for operations ~, |, and ^. Letting $\{0,1\}^w$ denote the set of all strings of 0s and 1s having length w, and a^w denote the string consisting of w repetitions of symbol a, then one can see that the resulting algebras: $\langle \{0,1\}^w, \mid, \&, \tilde{}, 0^w, 1^w\rangle$ and $\langle \{0,1\}^w, \hat{}, \&, I, 0^w, 1^w\rangle$ form Boolean algebras and rings, respectively. Each value of w defines a different Boolean algebra and a different Boolean ring.

Aside: Are Boolean rings the same as modular arithmetic?
The two-element Boolean ring $\langle \{0,1\}, \hat{}, \&, I, 0, 1\rangle$ is identical to the ring of integers modulo two $\langle \mathcal{Z}_2, +_2, \times_2, -_2, 0, 1\rangle$. The generalization to bit vectors of length w, however, yields a very different ring from modular arithmetic. **End Aside.**

Practice Problem 2.8:

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

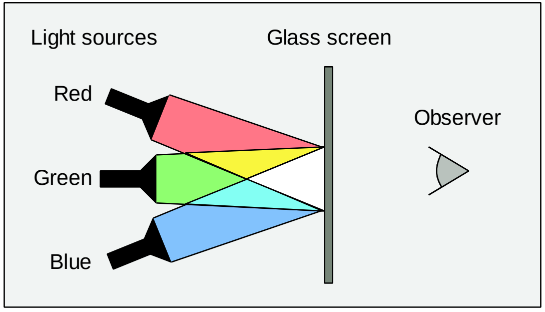
Operation	Result
a	$[01101001]$
b	$[01010101]$
~a	
~b	
a & b	
$a \mid b$	
a ^ b	

One useful application of bit vectors is to represent finite sets. For example, we can denote any subset $A \subseteq \{0, 1, \ldots, w-1\}$ as a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where $a_i = 1$ if and only if $i \in A$. For example, (recalling that we write a_{w-1} on the left and a_0 on the right), we have $a = [01101001]$ representing the set $A = \{0, 3, 5, 6\}$, and $b = [01010101]$ representing the set $B = \{0, 2, 4, 6\}$. Under this interpretation, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. For example, the operation a & b yields bit vector $[01000001]$, while $A \cap B = \{0, 6\}$.

In fact, for any set S, the structure $\langle \mathcal{P}(S), \cup, \cap, \overline{\ }, \emptyset, S \rangle$ forms a Boolean algebra, where $\mathcal{P}(S)$ denotes the set of all subsets of S, and $\overline{\ }$ denotes the set complement operator. That is, for any set A, its complement is the set $\overline{A} = \{a \in S | a \notin A\}$. The ability to represent and manipulate finite sets using bit vector operations is a practical outcome of a deep mathematical principle.

Practice Problem 2.9:

Computers generate color pictures on a video screen or liquid crystal display by mixing three different colors of light: red, green, and blue. Imagine a simple scheme, with three different lights, each of which can be turned on or off, projecting onto a glass screen:



We can then create eight different colors based on the absence (0) or presence (1) of light sources R, G, and B:

R	G	B	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

This set of colors forms an eight-element Boolean algebra.

A. The complement of a color is formed by turning off the lights that are on and turning on the lights that are off. What would be the complements of the eight colors listed above?

B. What colors correspond to Boolean values 0^w and 1^w for this algebra?

C. Describe the effect of applying Boolean operations on the following colors:

Blue | Red =
Magenta & Cyan =
Green ^ White =

2.1.8 Bit-Level Operations in C

One useful feature of C is that it supports bit-wise Boolean operations. In fact, the symbols we have used for the Boolean operations are exactly those used by C: `|` for OR, `&` for AND, `~` for NOT, and `^` for EXCLUSIVE-OR. These can be applied to any "integral" data type, that is, one declared as type `char` or `int`, with or without qualifiers such as `short`, `long`, or `unsigned`. Here are some examples of expression evaluations:

C expression	Binary expression	Binary result	C result
`~0x41`	~[01000001]	[10111110]	`0xBE`
`~0x00`	~[00000000]	[11111111]	`0xFF`
`0x69 & 0x55`	[01101001] & [01010101]	[01000001]	`0x41`
`0x69 | 0x55`	[01101001] | [01010101]	[01111101]	`0x7D`

As our examples show, the best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

Practice Problem 2.10:

To show how the ring properties of ^ can be useful, consider the following program:

```
void inplace_swap(int *x, int *y)
{
    *x = *x ^ *y;  /* Step 1 */
    *y = *x ^ *y;  /* Step 2 */
    *x = *x ^ *y;  /* Step 3 */
}
```

As the name implies, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables x and y. Note that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is no performance advantage to this way of swapping. It is merely an intellectual amusement.

Starting with values a and b in the locations pointed to by x and y, respectively, fill in the table that follows giving the values stored at the two locations after each step of the procedure. Use the ring properties to show that the desired effect is achieved. Recall that every element is its own additive inverse (that is, a ^ $a = 0$).

Step	*x	*y
Initially	a	b
Step 1		
Step 2		
Step 3		

One common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that indicates a selected set of bits within a word. As an example, the mask 0xFF (having 1s for the least significant eight bits) indicates the low-order byte of a word. The bit-level operation x & 0xFF yields a value consisting of the least significant byte of x, but with all other bytes set to 0. For example, with x = 0x89ABCDEF, the expression would yield 0x000000EF. The expression ~0 will yield a mask of all 1s, regardless of the word size of the machine. Although the same mask can be written 0xFFFFFFFF for a 32-bit machine, such code is not as portable.

Practice Problem 2.11:

Write C expressions for the following values, with the results for x = 0x98FDECBA and a 32-bit word size shown in square brackets:

A. The least significant byte of x, with all other bits set to 1 [0xFFFFFFBA].

B. The complement of the least significant byte of x, with all other bytes left unchanged [0x98FDEC45].

C. All but the least significant byte of x, with the least significant byte set to 0 [0x98FDEC00].

Although our examples assume a 32-bit word size, your code should work for any word size $w \geq 8$.

Practice Problem 2.12:

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations AND and OR, it had instructions bis (bit set) and bic (bit clear). Both instructions take a data word x and a mask word m. They generate a result z consisting of the bits of x modified according to the bits of m. With bis, the modification involves setting z to 1 at each bit position where m is 1. With bic, the modification involves setting z to 0 at each bit position where m is 1.

We would like to write C functions bis and bic to compute the effect of these two instructions. Fill in the missing expressions in the following code using the bit-level operations of C:

```
/* Bit Set */
int bis(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ___________;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  /* Write an expression in C that computes the effect of bit clear */
  int result = ___________;
  return result;
}
```

2.1.9 Logical Operations in C

C also provides a set of *logical* operators `||`, `&&`, and `!`, which correspond to the OR, AND, and NOT operations of propositional logic. These can easily be confused with the bit-level operations, but their function is quite different. The logical operations treat any nonzero argument as representing TRUE and argument 0 as representing FALSE. They return either 1 or 0, indicating a result of either TRUE or FALSE, respectively. Here are some examples of expression evaluations:

<table>
<tr><th>Expression</th><th>Result</th></tr>
<tr><td>!0x41</td><td>0x00</td></tr>
<tr><td>!0x00</td><td>0x01</td></tr>
<tr><td>!!0x41</td><td>0x01</td></tr>
<tr><td>0x69 && 0x55</td><td>0x01</td></tr>
<tr><td>0x69 || 0x55</td><td>0x01</td></tr>
</table>

Observe that a bit-wise operation will have behavior matching that of its logical counterpart only in the special case in which the arguments are restricted to 0 or 1.

A second important distinction between the logical operators `&&` and `||` versus their bit-level counterparts `&` and `|` is that the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument. Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Practice Problem 2.13:

Suppose that `x` and `y` have byte values `0x66` and `0x93`, respectively. Fill in the following table indicating the byte values of the different C expressions:

<table>
<tr><th>Expression</th><th>Value</th><th>Expression</th><th>Value</th></tr>
<tr><td>x & y</td><td></td><td>x && y</td><td></td></tr>
<tr><td>x | y</td><td></td><td>x || y</td><td></td></tr>
<tr><td>~x | ~y</td><td></td><td>!x || !y</td><td></td></tr>
<tr><td>x & !y</td><td></td><td>x && ~y</td><td></td></tr>
</table>

Practice Problem 2.14:

Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. In other words, it will return 1 when `x` and `y` are equal and 0 otherwise.

2.1.10 Shift Operations in C

C also provides a set of *shift* operations for shifting bit patterns to the left and to the right. For an operand `x` having bit representation $[x_{n-1}, x_{n-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{n-k-1}, x_{n-k-2}, \ldots, x_0, 0, \ldots 0]$. That is, `x` is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k 0s. The shift amount should be a value between 0 and $n-1$. Shift operations group from left to right, so `x << j << k` is equivalent to `(x << j) << k`. Be careful about operator precedence: `1<<5 - 1` is evaluated as `1 << (5-1)`, not as `(1<<5) - 1`.

There is a corresponding right shift operation `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift: *logical* and *arithmetic*. A logical right shift fills the left end with k 0s, giving a result $[0, \ldots, 0, x_{n-1}, x_{n-2}, \ldots x_k]$. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{n-1}, \ldots, x_{n-1}, x_{n-1}, x_{n-2}, \ldots x_k]$. This convention might seem peculiar, but as we will see it is useful for operating on signed integer data.

The C standard does not precisely define which type of right shift should be used. For unsigned data (i.e., integral objects declared with the qualifier `unsigned`), right shifts must be logical. For signed data (the default), either arithmetic or logical shifts may be used. This unfortunately means that any code assuming one form or the other will potentially encounter portability problems. In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case.

Practice Problem 2.15:

Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.

x		x << 3		x >> 2		x >> 2	
				(Logical)		(Arithmetic)	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
0xF0							
0x0F							
0xCC							
0x55							

C declaration	Guaranteed		Typical 32-bit	
	Minimum	Maximum	Minimum	Maximum
`char`	−127	127	−128	127
`unsigned char`	0	255	0	255
`short [int]`	−32,767	32,767	−32,768	32,767
`unsigned short [int]`	0	63,535	0	63,535
`int`	−32,767	32,767	−2,147,483,648	2,147,483,647
`unsigned [int]`	0	65,535	0	4,294,967,295
`long [int]`	−2,147,483,647	2,147,483,647	−2,147,483,648	$2,147,483,647$
`unsigned long [int]`	0	4,294,967,295	0	4,294,967,295

Figure 2.8: **C Integral data types.** Text in square brackets is optional.

2.2 Integer Representations

In this section we describe two different ways bits can be used to encode integers—one that can only represent nonnegative numbers, and one that can represent negative, zero, and positive numbers. We will see later that they are strongly related both in their mathematical properties and their machine-level implementations. We also investigate the effect of expanding or shrinking an encoded integer to fit a representation with a different length.

2.2.1 Integral Data Types

C supports a variety of *integral* data types—ones that represent a finite range of integers. These are shown in Figure 2.8. Each type has a size designator: `char`, `short`, `int`, and `long`, as well as an indication of whether the represented number is nonnegative (declared as `unsigned`), or possibly negative (the default). The typical allocations for these different sizes were given in Figure 2.2. As indicated in Figure 2.8, these different sizes allow different ranges of values to be represented. The C standard defines a minimum range of values each data type must be able to represent. As shown in the figure, a typical 32-bit machine uses a 32-bit representation for data types `int` and `unsigned`, even though the C standard allows 16-bit representations. As described in Figure 2.2, the Compaq Alpha uses a 64-bit word to represent `long` integers, giving an upper limit of over 1.84×10^{19} for unsigned values, and a range of over $\pm 9.22 \times 10^{18}$ for signed values.

New to C?: Signed and unsigned numbers in C, C++, and Java.
Both C and C++ support signed (the default) and unsigned numbers. Java supports only signed numbers. **End.**

2.2.2 Unsigned and Two's-Complement Encodings

Assume we have an integer data type of w bits. We write a bit vector as either $\vec{x}$, to denote the entire vector, or as $[x_{w-1}, x_{w-2}, \ldots, x_0]$ to denote the individual bits within the vector. Treating $\vec{x}$ as a number written in binary notation, we obtain the *unsigned* interpretation of $\vec{x}$. We express this interpretation as a function

$B2U_w$ (for "binary to unsigned," length w):

$$B2U_w(\vec{x}) \quad \doteq \quad \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

In this equation, the notation "$\doteq$" means that the left hand side is defined to be equal to the right hand side. The function $B2U_w$ maps strings of 0s and 1s of length w to nonnegative integers. The least value is given by bit vector $[00 \cdots 0]$ having integer value 0, and the greatest value is given by bit vector $[11 \cdots 1]$ having integer value $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$. Thus, the function $B2U_w$ can be defined as a mapping $B2U_w \colon \{0,1\}^w \rightarrow \{0, \ldots, 2^w - 1\}$. Note that $B2U_w$ is a *bijection*—it associates a unique value to each bit vector of length w; conversely, each integer between 0 and $2^w - 1$ has a unique binary representation as a bit vector of length w.

For many applications, we wish to represent negative values as well. The most common computer representation of signed numbers is known as *two's-complement* form. This is defined by interpreting the most significant bit of the word to have negative weight. We express this interpretation as a function $B2T_w$ (for "binary to two's complement" length w):

$$B2T_w(\vec{x}) \quad \doteq \quad -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.2}$$

The most significant bit is also called the *sign bit*. When set to 1, the represented value is negative, and when set to 0 the value is nonnegative. The least representable value is given by bit vector $[10 \cdots 0]$ (i.e., set the bit with negative weight but clear all others) having integer value $TMin_w \doteq -2^{w-1}$. The greatest value is given by bit vector $[01 \cdots 1]$, having integer value $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. Again, one can see that $B2T_w$ is a bijection $B2T_w \colon \{0,1\}^w \rightarrow \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$, associating a unique integer in the representable range for each bit pattern.

Practice Problem 2.16:

Assuming $w = 4$, we can assign a numeric value to each possible hexadecimal digit, assuming either an unsigned or two's-complement interpretation. Fill in the following table according to these interpretations by writing out the nonzero powers of two in the summations shown in Equations 2.1 and 2.2:

$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
Hexadecimal	Binary		
A	[1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
0			
3			
8			
C			
F			

Figure 2.9 shows the bit patterns and numeric values for several "interesting" numbers for different word sizes. The first three give the ranges of representable integers. A few points are worth highlighting. First, the

Quantity	Word size w			
	8	16	32	64
$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	-128	$-32,768$	$-2,147,483,648$	$-9,223,372,036,854,775,808$
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Figure 2.9: **"Interesting" numbers.** Both numeric values and hexadecimal representations are shown.

two's-complement range is asymmetric: $|TMin_w| = |TMax_w| + 1$, that is, there is no positive counterpart to $TMin_w$. As we shall see, this leads to some peculiar properties of two's-complement arithmetic and can be the source of subtle program bugs. Second, the maximum unsigned value is just over twice the maximum two's-complement value: $UMax_w = 2TMax_w + 1$. This follows from the fact that two's-complement notation reserves half of the bit patterns to represent negative values. The other cases are the constants -1 and 0. Note that -1 has the same bit representation as $UMax_w$—a string of all 1s. Numeric value 0 is represented as a string of all 0s in both representations.

The C standard does not require signed integers to be represented in two's-complement form, but nearly all machines do so. To keep code portable, one should not assume any particular range of representable values or how they are represented, beyond the ranges indicated in Figure 2.2. The C library file `<limits.h>` defines a set of constants delimiting the ranges of the different integer data types for the particular machine on which the compiler is running. For example, it defines constants `INT_MAX`, `INT_MIN`, and `UINT_MAX` describing the ranges of signed and unsigned integers. For a two's-complement machine in which data type `int` has w bits, these constants correspond to the values of $TMax_w$, $TMin_w$, and $UMax_w$.

Aside: Alternative representations of signed numbers.
There are two other standard representations for signed numbers:

Ones' Complement: This is the same as two's complement, except that the most significant bit has weight $-(2^{w-1} - 1)$ rather than -2^{w-1}:

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

Sign-Magnitude: The most significant bit is a sign bit that determines whether the remaining bits should be given negative or positive weight:

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i\right)$$

Both of these representations have the curious property that there are two different encodings of the number 0. For both representations, $[00 \cdots 0]$ is interpreted as $+0$. The value -0 can be represented in sign-magnitude form as $[10 \cdots 0]$ and in ones' complement as $[11 \cdots 1]$. Although machines based on ones' complement representations

Weight	12,345		−12,345		53,191	
	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4096	0	0	0	0
8,192	1	8192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32, 768	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

Figure 2.10: **two's-complement representations of 12,345 and −12,345, and unsigned representation of 53,191.** Note that the latter two have identical bit representations.

were built in the past, almost all modern machines use two's complement. We will see that sign-magnitude encoding is used with floating-point numbers.

Note the different position of apostrophes: *Two's* complement versus *Ones'* complement. **End Aside.**

As an example, consider the following code:

```
short int x = 12345;
short int mx = -x;

show_bytes((byte_pointer) &x, sizeof(short int));
show_bytes((byte_pointer) &mx, sizeof(short int));
```

When run on a big-endian machine, this code prints `30 39` and `cf c7`, indicating that `x` has hexadecimal representation `0x3039`, while `mx` has hexadecimal representation `0xCFC7`. Expanding these into binary we get bit patterns [0011000000111001] for `x` and [1100111111000111] for `mx`. As Figure 2.10 shows, Equation 2.2 yields values 12,345 and −12,345 for these two bit patterns.

Practice Problem 2.17:

In Chapter 3, we will look at listings generated by a *disassembler*, a program that converts an executable program file back to a more readable ASCII form. These files contain many hexadecimal numbers, typically representing values in two's-complement form. Being able to recognize these numbers and understand their significance (for example, whether they are negative or positive) is an important skill.

For the lines labeled A–K (on the right) in the following listing, convert the hexadecimal values shown to the right of the instruction names (`sub`, `push`, `mov`, and `add`) into their decimal equivalents.

```
80483b7:  81 ec 84 01 00 00       sub    $0x184,%esp             A.
80483bd:  53                      push   %ebx
80483be:  8b 55 08                mov    0x8(%ebp),%edx          B.
80483c1:  8b 5d 0c                mov    0xc(%ebp),%ebx          C.
80483c4:  8b 4d 10                mov    0x10(%ebp),%ecx         D.
80483c7:  8b 85 94 fe ff ff       mov    0xfffffe94(%ebp),%eax   E.
80483cd:  01 cb                   add    %ecx,%ebx
80483cf:  03 42 10                add    0x10(%edx),%eax         F.
80483d2:  89 85 a0 fe ff ff       mov    %eax,0xfffffea0(%ebp)   G.
80483d8:  8b 85 10 ff ff ff       mov    0xffffff10(%ebp),%eax   H.
80483de:  89 42 1c                mov    %eax,0x1c(%edx)         I.
80483e1:  89 9d 7c ff ff ff       mov    %ebx,0xffffff7c(%ebp)   J.
80483e7:  8b 42 18                mov    0x18(%edx),%eax         K.
```

2.2.3 Conversions Between Signed and Unsigned

Since both $B2U_w$ and $B2T_w$ are bijections, they have well-defined inverses. Define $U2B_w$ to be $B2U_w^{-1}$, and $T2B_w$ to be $B2T_2^{-1}$. These functions give the unsigned or two's-complement bit patterns for a numeric value. Given an integer x in the range $0 \leq x < 2^w$, the function $U2B_w(x)$ gives the unique w-bit unsigned representation of x. Similarly, when x is in the range $-2^{w-1} \leq x < 2^{w-1}$, the function $T2B_w(x)$ gives the unique w-bit two's-complement representation of x. Observe that for values in the range $0 \leq x < 2^{w-1}$, both of these functions will yield the same bit representation—the most significant bit will be 0, and hence it does not matter whether this bit has positive or negative weight.

Consider the function $U2T_w(x) \doteq B2T_w(U2B_w(x))$, which takes a number between 0 and $2^w - 1$ and yields a number between -2^{w-1} and $2^{w-1} - 1$, where the two numbers have identical bit representations, except that the argument is unsigned, while the result has a two's-complement representation. Conversely, the function $T2U_w(x) \doteq B2U_w(T2B_w(x))$ yields the unsigned number having the same bit representation as the two's-complement value of `x`. For example, as Figure 2.10 indicates, the 16-bit, two's-complement representation of $-$12,345 is identical to the 16-bit, unsigned representation of 53,191. Therefore, $T2U_{16}(-12,345) = 53,191$, and $U2T_{16}(53,191) = -12,345$.

These two functions might seem to be of only academic interest, but they actually have great practical importance—they formally define the effect of casting between signed and unsigned values in C. For example, consider executing the following code on a two's-complement machine:

```
int x = -1;
unsigned ux = (unsigned) x;
```

This code will set `ux` to $UMax_w$, where w is the number of bits in data type `int`, since by Figure 2.9 we can see that the w-bit two's-complement representation of -1 has the same bit representation as $UMax_w$. In

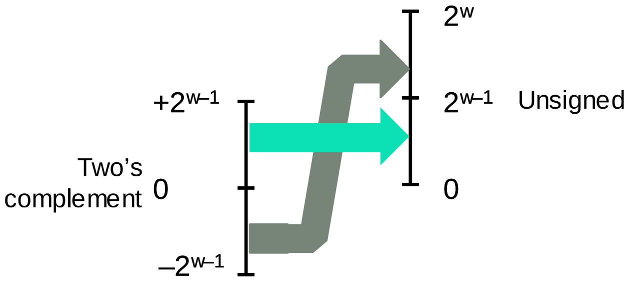


Figure 2.11: **Conversion from two's complement to unsigned.** Function $T2U$ converts negative numbers to large positive numbers.

general, casting from a signed value `x` to unsigned value `(unsigned) x` is equivalent to applying function $T2U$. The cast does not change the bit representation of the argument, just how these bits are interpreted as a number. Similarly, casting from unsigned value `u` to signed value `(int) u` is equivalent to applying function $U2T$.

Practice Problem 2.18:

Using the table you filled in when solving Problem 2.16, fill in the following table describing the function $T2U_4$:

x	$T2U_4(x)$
-8	
-6	
-4	
-1	
0	
3	

To get a better understanding of the relation between a signed number x and its unsigned counterpart $T2U_w(x)$, we can use the fact that they have identical bit representations to derive a numerical relationship. Comparing Equations 2.1 and 2.2, we can see that for bit pattern $\vec{x}$, if we compute the difference $B2U_w(\vec{x}) - B2T_w(\vec{x})$, the weighted sums for bits from 0 to $w-2$ will cancel each other, leaving a value: $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$. This gives a relationship $B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$. If we let $x = B2T_w(\vec{x})$, we then have

$$B2U_w(T2B_w(x)) \quad = \quad T2U_w(x) \quad = \quad x_{w-1}2^w + x \tag{2.3}$$

This relationship is useful for proving relationships between unsigned and two's-complement arithmetic. In the two's-complement representation of x, bit x_{w-1} determines whether or not x is negative, giving

$$T2U_w(x) \quad = \quad \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.4}$$

Figure 2.11 illustrates the behavior of function $T2U$. As it illustrates, when mapping a signed number to its unsigned counterpart, negative numbers are converted to large positive numbers, while nonnegative numbers remain unchanged.

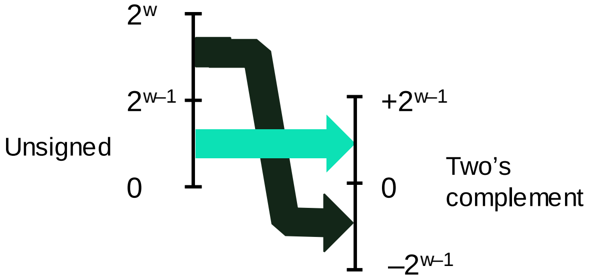


Figure 2.12: **Conversion from unsigned to two's complement.** Function $U2T$ converts numbers greater than $2^{w-1} - 1$ to negative values.

Practice Problem 2.19:

Explain how Equation 2.4 applies to the entries in the table you generated when solving Problem 2.18.

Going in the other direction, we wish to derive the relationship between an unsigned number x and its signed counterpart $U2T_w(x)$. If we let $x = B2U_w(\vec{x})$, we have

$$B2T_w(U2B_w(x)) \quad = \quad U2T_w(x) \quad = \quad -x_{w-1}2^w + x \tag{2.5}$$

In the unsigned representation of x, bit x_{w-1} determines whether or not x is greater than or equal to 2^{w-1}, giving

$$U2T_w(x) \quad = \quad \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases} \tag{2.6}$$

This behavior is illustrated in Figure 2.12. For small ($< 2^{w-1}$) numbers, the conversion from unsigned to signed preserves the numeric value. For large ($\geq 2^{w-1}$) the number is converted to a negative value.

To summarize, we can consider the effects of converting in both directions between unsigned and two's-complement representations. For values in the range $0 \leq x < 2^{w-1}$, we have $T2U_w(x) = x$ and $U2T_w(x) = x$. That is, numbers in this range have identical unsigned and two's-complement representations. For values outside of this range, the conversions either add or subtract 2^w. For example, we have $T2U_w(-1) = -1 + 2^w = UMax_w$—the negative number closest to 0 maps to the largest unsigned number. At the other extreme, one can see that $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$—the most negative number maps to an unsigned number just outside the range of positive, two's-complement numbers. Using the example of Figure 2.10, we can see that $T2U_{16}(-12,345) = 65,536 + -12,345 = 53,191$.

2.2.4 Signed vs. Unsigned in C

As indicated in Figure 2.8, C supports both signed and unsigned arithmetic for all of its integer data types. Although the C standard does not specify a particular representation of signed numbers, almost all machines use two's complement. Generally, most numbers are signed by default. For example, when declaring a constant such as `12345` or `0x1A2B`, the value is considered signed. To create an unsigned constant, the character '`U`' or '`u`' must be added as suffix (e.g., `12345U` or `0x1A2Bu`).

C allows conversion between unsigned and signed. The rule is that the underlying bit representation is not changed. Thus, on a two's-complement machine, the effect is to apply the function $\mathit{U2T}_w$ when converting from unsigned to signed, and $\mathit{T2U}_w$ when converting from signed to unsigned, where w is the number of bits for the data type.

Conversions can happen due to explicit casting, such as in the following code:

```
int tx, ty;
unsigned ux, uy;

tx = (int) ux;
uy = (unsigned) ty;
```

Alternatively, they can happen implicitly when an expression of one type is assigned to a variable of another, as in the following code:

```
int tx, ty;
unsigned ux, uy;

tx = ux; /* Cast to signed */
uy = ty; /* Cast to unsigned */
```

When printing numeric values with `printf`, the directives `%d`, `%u`, and `%x` should be used to print a number as a signed decimal, an unsigned decimal, and in hexadecimal format, respectively. Note that `printf` does not make use of any type information, and so it is possible to print a value of type `int` with directive `%u` and a value of type `unsigned` with directive `%d`. For example, consider the following code:

```
int x = -1;
unsigned u = 2147483648; /* 2 to the 31st */

printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);
```

When run on a 32-bit machine, it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

In both cases, `printf` prints the word first as if it represented an unsigned number and second as if it represented a signed number. We can see the conversion routines in action: $\mathit{T2U}_{32}(-1) = \mathit{UMax}_{32} = 4,294,967,295$ and $\mathit{U2T}_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = \mathit{TMin}_{32}$.

Some peculiar behavior arises due to C's handling of expressions containing combinations of signed and unsigned quantities. When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative. As we will see, this convention makes little difference for standard arithmetic operations, but

Expression	Type	Evaluation
`0 == 0U`	unsigned	1
`-1 < 0`	signed	1
`-1 < 0U`	unsigned	0 *
`2147483647 > -2147483648`	signed	1
`2147483647U > -2147483648`	unsigned	0 *
`2147483647 > (int) 2147483648U`	signed	1 *
`-1 > -2`	signed	1
`(unsigned) -1 > -2`	unsigned	1

Figure 2.13: **Effects of C promotion rules on 32-bit machine.** Nonintuitive cases marked by '*'. When either operand of a comparison is unsigned, the other operand is implicitly cast to unsigned.

it leads to nonintuitive results for relational operators such as `<` and `>`. Figure 2.13 shows some sample relational expressions and their resulting evaluations, assuming a 32-bit machine using two's-complement representation. The nonintuitive cases are marked by '*'. Consider the comparison `-1 < 0U`. Since the second operand is unsigned, the first one is implicitly cast to unsigned, and hence the expression is equivalent to the comparison `4294967295U < 0U` (recall that $T2U_w(-1) = UMax_w$), which of course is false. The other cases can be understood by similar analyses.

Practice Problem 2.20:

Assuming the expressions are evaluated on a 32-bit machine that uses two's-complement arithmetic, fill in the following table describing the effect of casting and relational operations, in the style of Figure 2.13:

Expression	Type	Evaluation
`-2147483648 == 2147483648U`		
`-2147483648 < -21474836487`		
`(unsigned) -2147483648 < -21474836487`		
`-2147483648 < 21474836487`		
`(unsigned) -2147483648 < 21474836487`		

2.2.5 Expanding the Bit Representation of a Number

One common operation is to convert between integers having different word sizes, while retaining the same numeric value. Of course, this may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to a larger data type, however, should always be possible. To convert an unsigned number to a larger data type, we can simply add leading 0s to the representation. this operation is known as *zero extension*. For converting a two's-complement number to a larger data type, the rule is to perform a *sign extension*, adding copies of the most significant bit to the representation. Thus, if our original value has bit representation $[x_{w-1}, x_{w-2}, \ldots, x_0]$, the expanded representation would be $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$.

As an example, consider the following code:

```
    short sx = val;          /* -12345 */
    unsigned short usx = sx; /* 53191 */
    int   x = sx;            /* -12345 */
    unsigned  ux = usx;      /* 53191 */

    printf("sx  = %d:\t", sx);
    show_bytes((byte_pointer) &sx, sizeof(short));
    printf("usx = %u:\t", usx);
    show_bytes((byte_pointer) &usx, sizeof(unsigned short));
    printf("x   = %d:\t", x);
    show_bytes((byte_pointer) &x, sizeof(int));
    printf("ux  = %u:\t", ux);
    show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run on a 32-bit big-endian machine using two's-complement representations, this code prints the following output:

```
sx  = -12345:  cf c7
usx = 53191:   cf c7
x   = -12345:  ff ff cf c7
ux  = 53191:   00 00 cf c7
```

We see that, although the two's-complement representation of $-12{,}345$ and the unsigned representation of 53,191 are identical for a 16-bit word size, they differ for a 32-bit word size. In particular, $-12{,}345$ has hexadecimal representation `0xFFFFCFC7`, while 53,191 has hexadecimal representation `0x0000CFC7`. The former has been sign extended—16 copies of the most significant bit 1, having hexadecimal representation `0xFFFF`, have been added as leading bits. The latter has been extended with 16 leading 0s, having hexadecimal representation `0x0000`.

Can we justify that sign extension works? What we want to prove is that

$$B2T_{w+k}([x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) \quad = \quad B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

where in the expression on the left hand side, we have made k additional copies of bit x_{w-1}. The proof follows by induction on k. That is, if we can prove that sign extending by one bit preserves the numeric value, then this property will hold when sign extending by an arbitrary number of bits. Thus, the task reduces to proving that

$$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) \quad = \quad B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

Expanding the left hand expression with Equation 2.2 gives the following:

$$\begin{aligned} B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\ &= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \end{aligned}$$

$$\begin{aligned}
&= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])
\end{aligned}$$

The key property we exploit is that $-2^w + 2^{w-1} = -2^{w-1}$. Thus, the combined effect of adding a bit of weight -2^w and of converting the bit having weight -2^{w-1} to be one with weight 2^{w-1} is to preserve the original numeric value.

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program. Consider the following additional code for our previous example:

```
    unsigned  uy = x;         /* Mystery! */

    printf("uy  = %u:\t", uy);
    show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

This portion of the code causes the following output to be printed:

```
uy = 4294954951:  ff ff cf c7
```

This shows that the expressions:

```
(unsigned) (int) sx           /* 4294954951 */
```

and

```
(unsigned) (unsigned short) sx     /* 53191 */
```

produce different values, even though the original and the final data types are the same. In the former expression, we first sign extend the 16-bit `short` to a 32-bit `int`, whereas zero extension is performed in the latter expression.

Practice Problem 2.21:

Consider the following C functions:

```
int fun1(unsigned word)
{
    return (int) ((word << 24) >> 24);
}

int fun2(unsigned word)
{
    return ((int) word << 24) >> 24;
}
```

Assume these are executed on a machine with a 32-bit word size that uses two's-complement arithmetic. Assume also that right shifts of signed values are performed arithmetically, while right shifts of unsigned values are performed logically.

A. Fill in the following table showing the effect of these functions for several example arguments:

w	fun1(w)	fun2(w)
127		
128		
255		
256		

B. Describe in words the useful computation each of these functions performs.

2.2.6 Truncating Numbers

Suppose that rather than extending a value with extra bits, we reduce the number of bits representing a number. This occurs, for example, in the code:

```
int   x = 53191;
short sx = (short) x;  /* -12345 */
int   y = sx;          /* -12345 */
```

On a typical 32-bit machine, when we cast `x` to be `short`, we truncate the 32-bit `int` to be a 16-bit `short int`. As we saw before, this 16-bit pattern is the two's-complement representation of $-12{,}345$. When we cast this back to `int`, sign extension will set the high-order 16 bits to 1s, yielding the 32-bit two's-complement representation of $-12{,}345$.

When truncating a w-bit number $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ to a k-bit number, we drop the high-order $w-k$ bits, giving a bit vector $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Truncating a number can alter its value—a form of overflow. We now investigate what numeric value will result. For an unsigned number x, the result of truncating it to k bits is equivalent to computing $x \bmod 2^k$. This can be seen by applying the modulus operation to Equation 2.1:

$$\begin{aligned}
B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\
&= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\
&= \sum_{i=0}^{k-1} x_i 2^i \\
&= B2U_k([x_k, x_{k-1}, \ldots, x_0])
\end{aligned}$$

In the above derivation we make use of the property that $2^i \bmod 2^k = 0$ for any $i \geq k$, and that $\sum_{i=0}^{k-1} x_i 2^i \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$.

For a two's-complement number x, a similar argument shows that $B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k = B2U_k([x_k, x_{k-1}, \ldots, x_0])$. That is, $x \bmod 2^k$ can be represented by an unsigned number having bit-level representation $[x_{k-1}, \ldots, x_0]$. In general, however, we treat the truncated number as being signed. This will have numeric value $U2T_k(x \bmod 2^k)$.

Summarizing, the effects of truncation are:

$$B2U_k([x_k, x_{k-1}, \ldots, x_0]) = B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k \tag{2.7}$$

$$B2T_k([x_k, x_{k-1}, \ldots, x_0]) = U2T_k(B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k) \tag{2.8}$$

Practice Problem 2.22:

Suppose we truncate a four-bit value (represented by hex digits `0` through `F`) to a three-bit value (represented as hex digits `0` through `7`). Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's-complement interpretations of those bit patterns.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
`0`	`0`	0		0	
`3`	`3`	3		3	
`8`	`0`	8		-8	
`A`	`2`	10		-6	
`F`	`7`	15		-1	

Explain how Equations 2.7 and 2.8 apply to these cases.

2.2.7 Advice on Signed vs. Unsigned

As we have seen, the implicit casting of signed to unsigned leads to some nonintuitive behavior. Nonintuitive features often lead to program bugs, and ones involving the nuances of implicit casting can be especially difficult to see. Since the casting is invisible, we can often overlook its effects.

Practice Problem 2.23:

Consider the following code that attempts to sum the elements of an array `a`, where the number of elements is given by parameter `length`:

```
/* WARNING: This is buggy code */
float sum_elements(float a[], unsigned length)
{
    int i;
    float result = 0;

    for (i = 0; i <= length-1; i++)
        result += a[i];
    return result;
}
```

> When run with argument `length` equal to 0, this code should return 0.0. Instead it encounters a memory error. Explain why this happens. Show how this code can be corrected.

One way to avoid such bugs is to never use unsigned numbers. In fact, few languages other than C support unsigned integers. Apparently these other language designers viewed them as more trouble than they are worth. For example, Java supports only signed integers, and it requires that they be implemented with two's-complement arithmetic. The normal right shift operator `>>` is guaranteed to perform an arithmetic shift. The special operator `>>>` is defined to perform a logical right shift.

Unsigned values are very useful when we want to think of words as just collections of bits with no numeric interpretation. This occurs, for example, when packing a word with *flags* describing various Boolean conditions. Addresses are naturally unsigned, so systems programmers find unsigned types to be helpful. Unsigned values are also useful when implementing mathematical packages for modular arithmetic and for multiprecision arithmetic, in which numbers are represented by arrays of words.

2.3 Integer Arithmetic

Many beginning programmers are surprised to find that adding two positive numbers can yield a negative result, and that the comparison `x < y` can yield a different result than the comparison `x-y < 0`. These properties are artifacts of the finite nature of computer arithmetic. Understanding the nuances of computer arithmetic can help programmers write more reliable code.

2.3.1 Unsigned Addition

Consider two nonnegative integers x and y, such that $0 \leq x, y \leq 2^w - 1$. Each of these numbers can be represented by w-bit unsigned numbers. If we compute their sum, however, we have a possible range $0 \leq x+y \leq 2^{w+1}-2$. Representing this sum could require $w+1$ bits. For example, Figure 2.14 shows a plot of the function $x + y$ when x and y have four-bit representations. The arguments (shown on the horizontal axes) range from 0 to 15, but the sum ranges from 0 to 30. The shape of the function is a sloping plane. If we were to maintain the sum as a $w + 1$ bit number and add it to another value, we may require $w + 2$ bits, and so on. This continued "word size inflation" means we cannot place any bound on the word size required to fully represent the results of arithmetic operations. Some programming languages, such as Lisp, actually support *infinite precision* arithmetic to allow arbitrary (within the memory limits of the machine, of course) integer arithmetic. More commonly, programming languages support fixed-precision arithmetic, and hence operations such as "addition" and "multiplication" differ from their counterpart operations over integers.

Unsigned arithmetic can be viewed as a form of modular arithmetic. Unsigned addition is equivalent to computing the sum modulo 2^w. This value can be computed by simply discarding the high-order bit in the $w + 1$-bit representation of $x + y$. For example, consider a four-bit number representation with $x = 9$ and $y = 12$, having bit representations $[1001]$ and $[1100]$, respectively. Their sum is 21, having a 5-bit representation $[10101]$. But if we discard the high-order bit, we get $[0101]$, that is, decimal value 5. This matches the value $21 \bmod 16 = 5$.

In general, we can see that if $x+y < 2^w$, the leading bit in the $w+1$-bit representation of the sum will equal

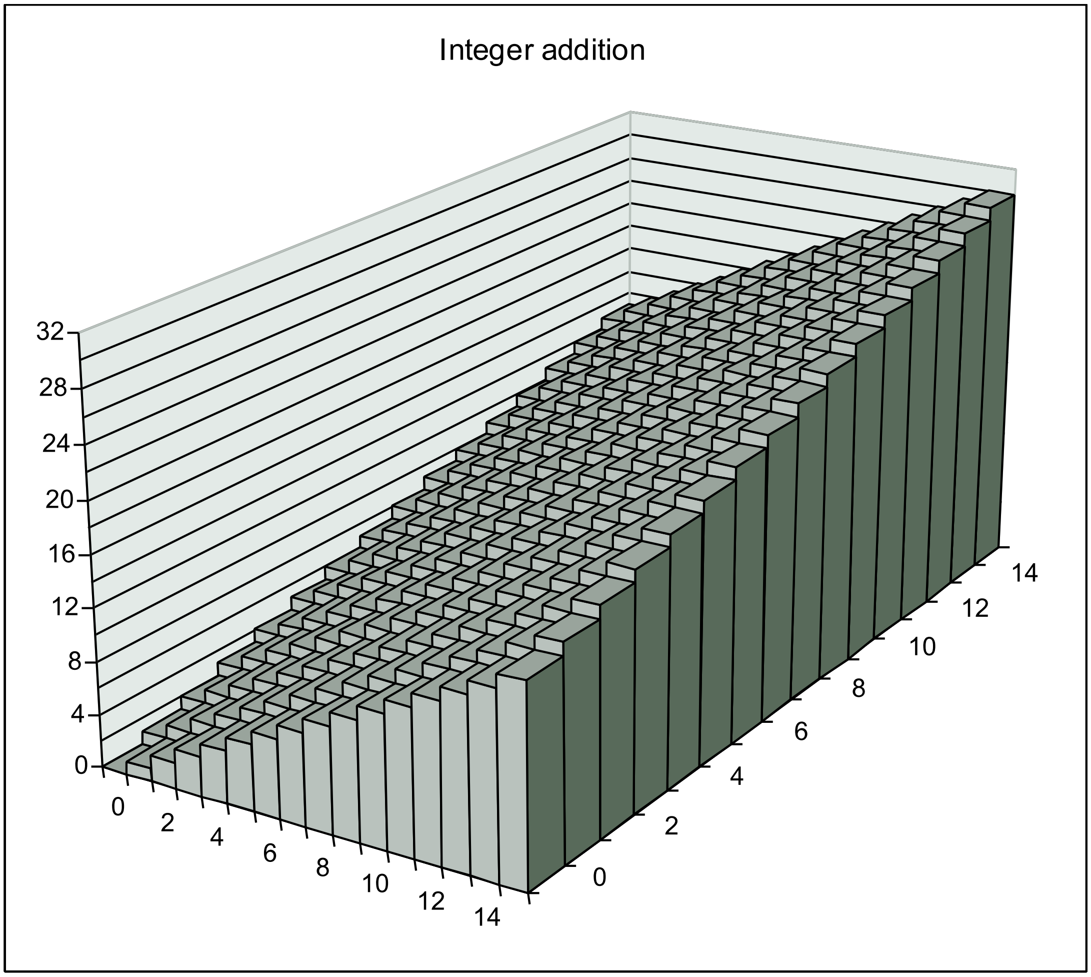


Figure 2.14: **Integer addition.** With a four-bit word size, the sum could require 5 bits.

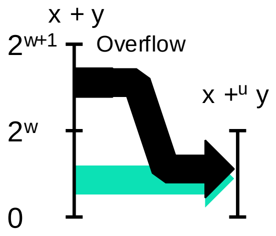


Figure 2.15: **Relation between integer addition and unsigned addition.** When $x + y$ is greater than $2^w - 1$, the sum overflows.

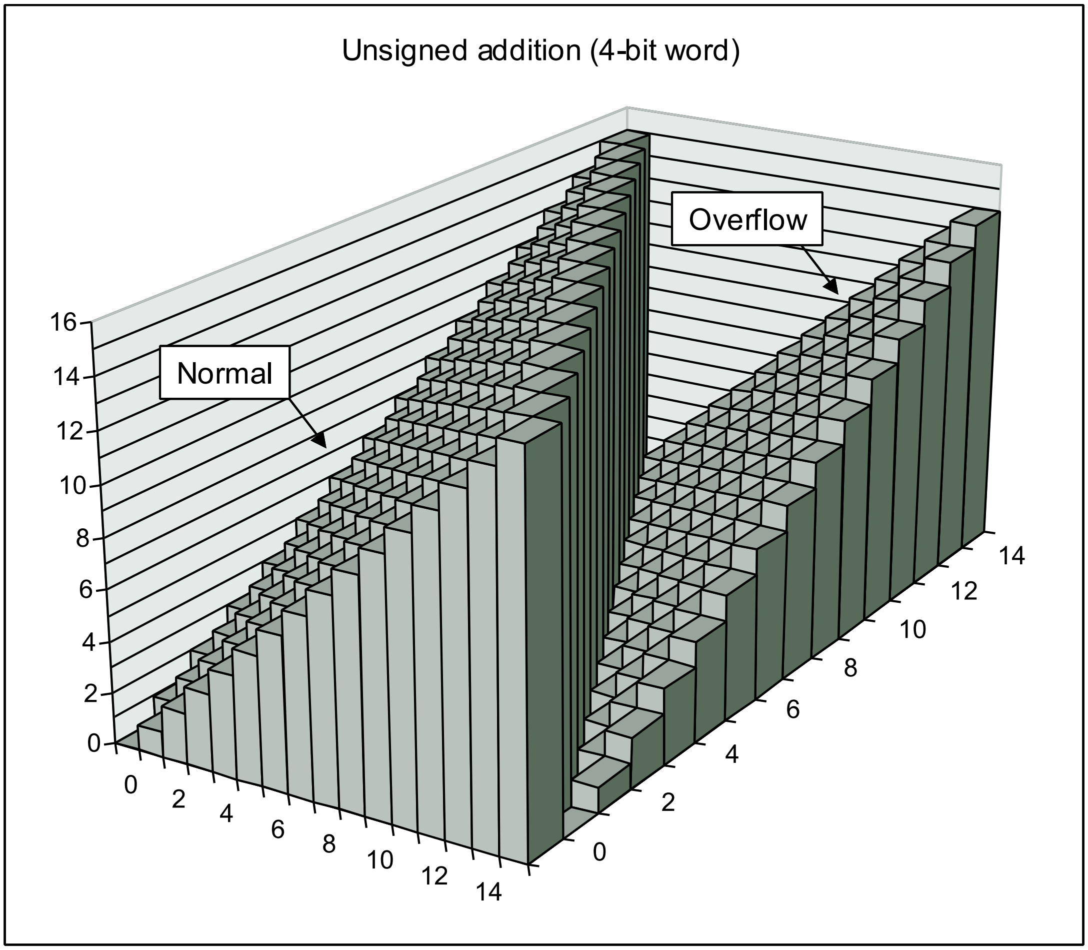


Figure 2.16: **Unsigned addition.** With a four-bit word size, addition is performed modulo 16.

0, and hence discarding it will not change the numeric value. On the other hand, if $2^w \leq x + y < 2^{w+1}$, the leading bit in the $w + 1$-bit representation of the sum will equal 1, and hence discarding it is equivalent to subtracting 2^w from the sum. These two cases are illustrated in Figure 2.15. This will give us a value in the range $0 \leq x + y - 2^w < 2^{w+1} - 2^w = 2^w$, which is precisely the modulo 2^w sum of x and y. Let us define the operation $+^{\mathrm{u}}_w$ for arguments x and y such that $0 \leq x, y < 2^w$ as:

$$x +^{\mathrm{u}}_w y \quad = \quad \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \end{cases} \tag{2.9}$$

This is precisely the result we get in C when performing addition on two w-bit unsigned values.

An arithmetic operation is said to *overflow* when the full integer result cannot fit within the word size limits of the data type. As Equation 2.9 indicates, overflow occurs when the two operands sum to 2^w or more. Figure 2.16 shows a plot of the unsigned addition function for word size $w = 4$. The sum is computed modulo $2^4 = 16$. When $x + y < 16$, there is no overflow, and $x +^{\mathrm{u}}_4 y$ is simply $x + y$. This is shown as the region forming a sloping plane labeled "Normal." When $x + y \geq 16$, the addition overflows, having the effect of decrementing the sum by 16. This is shown as the region forming a sloping plane labeled "Overflow."

When executing C programs, overflows are not signalled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute $s \doteq x +^{\mathrm{u}}_{w} y$, and we wish to determine whether s equals $x + y$. We claim that overflow has occurred if and only if $s < x$ (or equivalently $s < y$.) To see this, observe that $x + y \geq x$, and hence if s did not overflow, we will surely have $s \geq x$. On the other hand, if s did overflow, we have $s = x + y - 2^w$. Given that $y < 2^w$, we have $y - 2^w < 0$, and hence $s = x + y - 2^w < x$. In our earlier example, we saw that $9 +^{\mathrm{u}}_{4} 12 = 5$. We can see that overflow occurred, since $5 < 9$.

Modular addition forms a mathematical structure known as an *Abelian group*, named after the Danish mathematician Niels Henrik Abel (1802–1829). That is, it is commutative (that's where the "Abelian" part comes in) and associative. It has an identity element 0, and every element has an additive inverse. Let us consider the set of w-bit unsigned numbers with addition operation $+^{\mathrm{u}}_{w}$. For every value x, there must be some value $-^{\mathrm{u}}_{w} x$ such that $-^{\mathrm{u}}_{w} x +^{\mathrm{u}}_{w} x = 0$. When $x = 0$, the additive inverse is clearly 0. For $x > 0$, consider the value $2^w - x$. Observe that this number is in the range $0 \leq 2^w - x < 2^w$, and $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Hence it is the inverse of x under $+^{\mathrm{u}}_{w}$. These two cases lead to the following equation for $0 \leq x < 2^w$:

$$-^{\mathrm{u}}_{w} x \quad = \quad \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{2.10}$$

Practice Problem 2.24:

We can represent a bit pattern of length $w = 4$ with a single hex digit. For an unsigned interpretation of these digits, use Equation 2.10 fill in the following table giving the values and the bit representations (in hex) of the unsigned additive inverses of the digits shown.

x		$-^{\mathrm{u}}_{4} x$	
Hex	Decimal	Decimal	Hex
`0`			
`3`			
`8`			
`A`			
`F`			

2.3.2 Two's-Complement Addition

A similar problem arises for two's-complement addition. Given integer values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, their sum is in the range $-2^w \leq x + y \leq 2^w - 2$, potentially requiring $w + 1$ bits to represent exactly. As before, we avoid ever-expanding data sizes by truncating the representation to w bits. The result is not as familiar mathematically as modular addition, however.

The w-bit two's-complement sum of two numbers has the exact same bit-level representation as the unsigned sum. In fact, most computers use the same machine instruction to perform either unsigned or signed

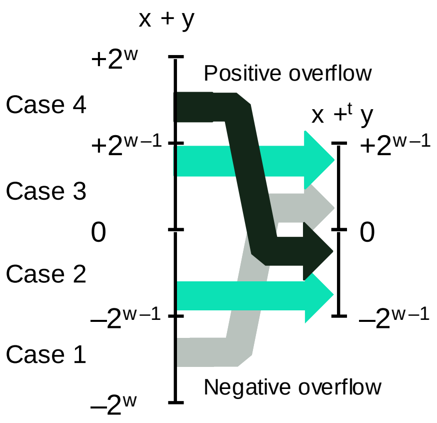


Figure 2.17: **Relation between integer and two's-complement addition.** When $x+y$ is less than -2^{w-1}, there is a negative overflow. When it is greater than $2^{w-1}+1$, there is a positive overflow.

addition. Thus, we can define two's-complement addition for word size w, denoted as $+^{\text{t}}_{w}$ on operands x and y such that $-2^{w-1} \leq x, y < 2^{w-1}$ as

$$x +^{\text{t}}_{w} y \quad \doteq \quad U2T_w(T2U_w(x) +^{\text{u}}_{w} T2U_w(y)) \tag{2.11}$$

By Equation 2.3 we can write $T2U_w(x)$ as $x_{w-1}2^w + x$, and $T2U_w(y)$ as $y_{w-1}2^w + y$. Using the property that $+^{\text{u}}_{w}$ is simply addition modulo 2^w, along with the properties of modular addition, we then have

$$\begin{aligned} x +^{\text{t}}_{w} y &= U2T_w(T2U_w(x) +^{\text{u}}_{w} T2U_w(y)) \\ &= U2T_w[(-x_{w-1}2^w + x + -y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x+y) \bmod 2^w] \end{aligned}$$

The terms $x_{w-1}2^w$ and $y_{w-1}2^w$ drop out since they equal 0 modulo 2^w.

To better understand this quantity, let us define z as the integer sum $z \doteq x+y$, z' as $z' \doteq z \bmod 2^w$, and z'' as $z'' \doteq U2T_w(z')$. The value z'' is equal to $x +^{\text{t}}_{w} y$. We can divide the analysis into four cases as illustrated in Figure 2.17:

1. $-2^w \leq z < -2^{w-1}$. Then we will have $z' = z + 2^w$. This gives $0 \leq z' < -2^{w-1} + 2^w = 2^{w-1}$. Examining Equation 2.6, we see that z' is in the range such that $z'' = z'$. This case is referred to as *negative overflow*. We have added two negative numbers x and y (that's the only way we can have $z < -2^{w-1}$) and obtained a nonnegative result $z'' = x + y + 2^w$.

2. $-2^{w-1} \leq z < 0$. Then we will again have $z' = z + 2^w$, giving $-2^{w-1} + 2^w = 2^{w-1} \leq z' < 2^w$. Examining Equation 2.6, we see that z' is in such a range that $z'' = z' - 2^w$, and therefore $z'' = z' - 2^w = z + 2^w - 2^w = z$. That is, our two's-complement sum z'' equals the integer sum $x+y$.

3. $0 \leq z < 2^{w-1}$. Then we will have $z' = z$, giving $0 \leq z' < 2^{w-1}$, and hence $z'' = z' = z$. Again, the two's-complement sum z'' equals the integer sum $x+y$.

x	y	$x+y$	$x +^{\mathrm{t}}_{4} y$	Case
-8 $[1000]$	-5 $[1011]$	-13	3 $[0011]$	1
-8 $[1000]$	-8 $[1000]$	-16	0 $[0000]$	1
-8 $[1000]$	5 $[0101]$	-3	-3 $[1101]$	2
2 $[0010]$	5 $[0101]$	7	7 $[0111]$	3
5 $[0101]$	5 $[0101]$	10	-6 $[1010]$	4

Figure 2.18: **two's-complement addition examples.** The bit-level representation of the four-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

4. $2^{w-1} \leq z < 2^w$. We will again have $z' = z$, giving $2^{w-1} \leq z' < 2^w$. But in this range we have $z'' = z' - 2^w$, giving $z'' = x + y - 2^w$. This case is referred to as *positive overflow*. We have added two positive numbers x and y (that's the only way we can have $z \geq 2^{w-1}$) and obtained a negative result $z'' = x + y - 2^w$.

By the preceding analysis, we have shown that when operation $+^{\mathrm{t}}_{w}$ is applied to values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, we have

$$x +^{\mathrm{t}}_{w} y \quad = \quad \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \quad \text{Positive Overflow} \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \quad \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} \quad \text{Negative Overflow} \end{cases} \tag{2.12}$$

As an illustration, Figure 2.18 shows some examples of four-bit two's-complement addition. Each example is labeled by the case to which it corresponds in the derivation of Equation 2.12. Note that $2^4 = 16$, and hence negative overflow yields a result 16 more than the integer sum, and positive overflow yields a result 16 less. We include bit-level representations of the operands and the result. Observe that the result can be obtained by performing binary addition of the operands and truncating the result to four bits.

Figure 2.19 illustrates two's-complement addition for word size $w = 4$. The operands range between -8 and 7. When $x + y < -8$, two's-complement addition has a negative underflow, causing the sum to be incremented by 16. When $-8 \leq x + y < 8$, the addition yields $x + y$. When $x + y \geq 8$, the addition has a positive overflow, causing the sum to be decremented by 16. Each of these three ranges forms a sloping plane in the figure.

Equation 2.12 also lets us identify the cases where overflow has occurred. When both x and y are negative, but $x +^{\mathrm{t}}_{w} y \geq 0$, we have negative overflow. When both x and y are positive, but $x +^{\mathrm{t}}_{w} y < 0$, we have positive overflow.

Practice Problem 2.25:

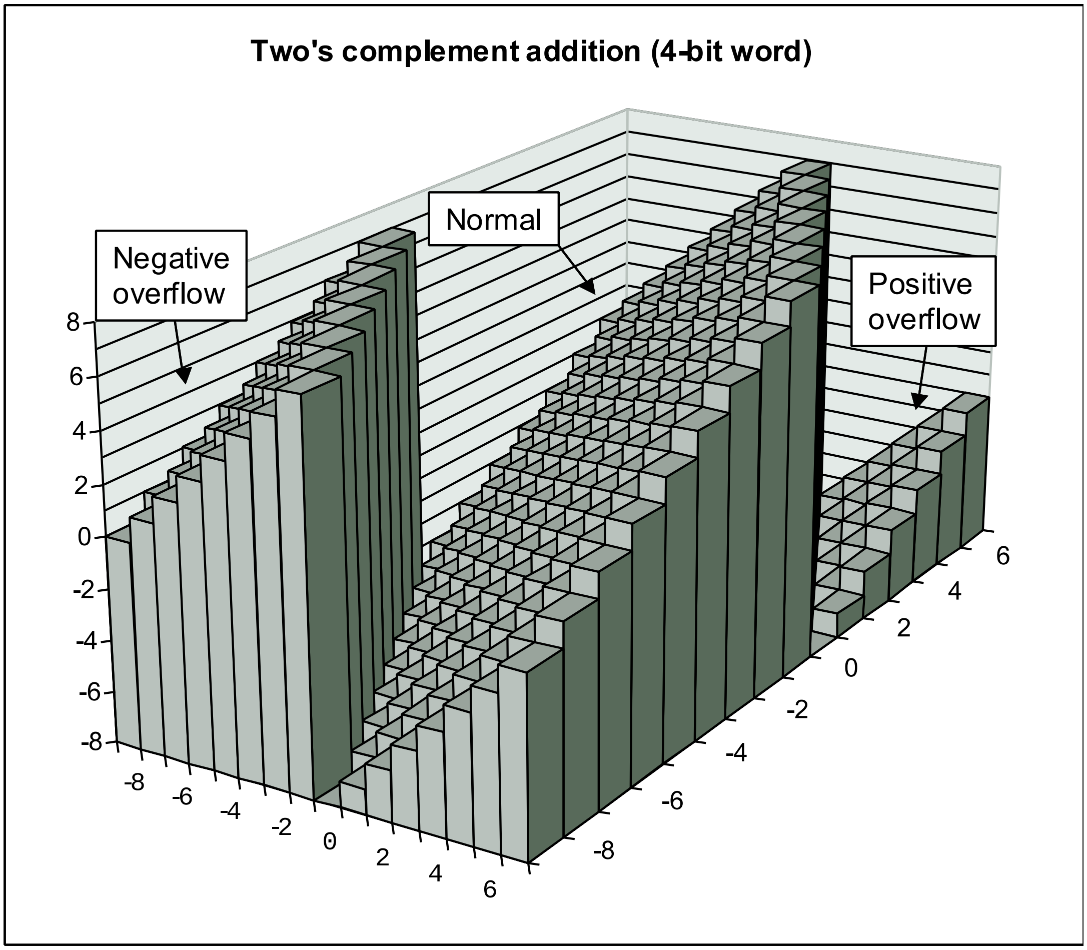


Figure 2.19: **two's-complement addition.** With a four-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

Fill in the following table in the style of Figure 2.18. Give the integer values of the 5-bit arguments, the values of both their integer and two's-complement sums, the bit-level representation of the two's-complement sum, and the case from the derivation of Equation 2.12.

x	y	$x + y$	$x +^{\mathrm{t}}_4 y$	Case
[10000]	[10101]			
[10000]	[10000]			
[11000]	[00111]			
[11110]	[00101]			
[01000]	[01000]			

2.3.3 Two's-Complement Negation

We can see that every number x in the range $-2^{w-1} \leq x < 2^{w-1}$ has an additive inverse under $+^{\mathrm{t}}_w$ as follows. First, for $x \neq -2^{w-1}$, we can see that its additive inverse is simply $-x$. That is, we have $-2^{w-1} < -x < 2^{w-1}$ and $-x +^{\mathrm{t}}_w x = -x + x = 0$. For $x = -2^{w-1} = TMin_w$, on the other hand, $-x = 2^{w-1}$ cannot be represented as a w-bit number. We claim that this special value has itself as the additive inverse under $+^{\mathrm{t}}_w$. The value of $-2^{w+1} +^{\mathrm{t}}_w -2^{w+1}$ is given by the third case of Equation 2.12, since $-2^{w-1} + -2^{w-1} = -2^w$. This gives $-2^{w+1} +^{\mathrm{t}}_w -2^{w+1} = -2^w + 2^w = 0$. From this analysis we can define the two's-complement negation operation $-^{\mathrm{t}}_w$ for x in the range $-2^{2-1} \leq x < 2^{w-1}$ as:

$$-^{\mathrm{t}}_w\, x \;=\; \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \tag{2.13}$$

Practice Problem 2.26:

We can represent a bit pattern of length $w = 4$ with a single hex digit. For a two's-complement interpretation of these digits, fill in the following table to determine the additive inverses of the digits shown.

x		$-^{\mathrm{t}}_4\, x$	
Hex	Decimal	Decimal	Hex
`0`			
`3`			
`8`			
`A`			
`F`			

What do you observe about the bit patterns generated by two's-complement and unsigned (Problem 2.24) negation?

A well-known technique for performing two's-complement negation at the bit level is to complement the bits and then increment the result. In C, this can be written as `~x + 1`. To justify the correctness of this technique, observe that for any single bit x_i, we have ~$x_i = 1 - x_i$. Let $\vec{x}$ be a bit vector of length w and $x \doteq B2T_w(\vec{x})$ be the two's-complement number it represents. By Equation 2.2, the complemented bit vector ~$\vec{x}$ has numeric value

$$\begin{aligned}
B2T_w(\text{~}\vec{x}) &= -(1 - x_{w-1})2^{w-1} + \sum_{i=0}^{w-2}(1 - x_i)2^i \\
&= \left[-2^{w-1} + \sum_{i=0}^{w-2} 2^i\right] - \left[-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i\right] \\
&= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\vec{x}) \\
&= -1 - x
\end{aligned}$$

The key simplification in the above derivation is that $\sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. It follows that by incrementing ~$\vec{x}$ we obtain $-x$.

To increment a number x represented at the bit-level as $\vec{x} \doteq [x_{w-1}, x_{w-2}, \ldots, x_0]$, define the operation incr as follows. Let k be the position of the rightmost zero, such that $\vec{x}$ is of the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 0, 1, \ldots, 1]$. We then define $\mathit{incr}(\vec{x})$ to be $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0]$. For the special case where the bit-level representation of x is $[1, 1, \ldots, 1]$, define $\mathit{incr}(\vec{x})$ to be $[0, \ldots, 0]$. To show that $\mathit{incr}(\vec{x})$ yields the bit-level representation of $x +^{\text{t}}_{w} 1$, consider the following cases:

1. When $\vec{x} = [1, 1, \ldots, 1]$, we have $x = -1$. The incremented value $\mathit{incr}(\vec{x}) \doteq [0, \ldots, 0]$ has numeric value 0.

2. When $k = w - 1$, i.e., $\vec{x} = [0, 1, \ldots, 1]$, we have $x = \mathit{TMax}_w$. The incremented value $\mathit{incr}(\vec{x}) = [1, 0, \ldots, 0]$ has numeric value TMin_w. From Equation 2.12, we can see that $\mathit{TMax}_w +^{\text{t}}_{w} 1$ is one of the positive overflow cases, yielding TMin_w.

3. When $k < w - 1$, i.e., $x \neq \mathit{TMax}_w$ and $x \neq -1$, we can see that the low-order $k + 1$ bits of $\mathit{incr}(\vec{x})$ has numeric value 2^k, while the low-order $k + 1$ bits of $\vec{x}$ has numeric value $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. The high-order $w - k + 1$ bits have matching numeric values. Thus, $\mathit{incr}(\vec{x})$ has numeric value $x + 1$. In addition, for $x \neq \mathit{TMax}_w$, adding 1 to x will not cause an overflow, and hence $x +^{\text{t}}_{w} 1$ has numeric value $x + 1$ as well.

As illustrations, Figure 2.20 shows how complementing and incrementing affect the numeric values of several four-bit vectors.

2.3.4 Unsigned Multiplication

Integers x and y in the range $0 \leq x, y \leq 2^w - 1$ can be represented as w-bit unsigned numbers, but their product $x \cdot y$ can range between 0 and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$. This could require as many as $2w$ bits to represent. Instead, unsigned multiplication in C is defined to yield the w-bit value given by the low-order

$\vec{x}$		$\mathtt{\sim}\vec{x}$		$\mathit{incr}(\mathtt{\sim}\vec{x})$	
[0101]	5	[1010]	-6	[1011]	-5
[0111]	7	[1000]	-8	[1001]	-7
[1100]	-4	[0011]	3	[0100]	4
[0000]	0	[1111]	-1	[0000]	0
[1000]	-8	[0111]	7	[1000]	-8

Figure 2.20: **Examples of complementing and incrementing four-bit numbers.** The effect is to compute the two's value negation.

w bits of the $2w$-bit integer product. By Equation 2.7, this can be seen to be equivalent to computing the product modulo 2^w. Thus, the effect of the w-bit unsigned multiplication operation $*^{\mathtt{u}}_{w}$ is:

$$x *^{\mathtt{u}}_{w} y \quad = (x \cdot y) \bmod 2^w \tag{2.14}$$

It is well known that modular arithmetic forms a ring. We can therefore deduce that unsigned arithmetic over w-bit numbers forms a ring $\langle\{0, \ldots, 2^w - 1\}, +^{\mathtt{u}}_{w}, *^{\mathtt{u}}_{w}, -^{\mathtt{u}}_{w}, 0, 1\rangle$.

2.3.5 Two's-Complement Multiplication

Integers x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ can be represented as w-bit two's-complement numbers, but their product $x \cdot y$ can range between $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ and $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$. This could require as many as $2w$ bits to represent in two's-complement form—most cases would fit into $2w - 1$ bits, but the special case of 2^{2w-2} requires the full $2w$ bits (to include a sign bit of 0). Instead, signed multiplication in C generally is performed by truncating the $2w$-bit product to w bits. By Equation 2.8, the effect of the w-bit two's-complement multiplication operation $*^{\mathtt{t}}_{w}$ is

$$x *^{\mathtt{t}}_{w} y \quad = \quad \mathit{U2T}_w((x \cdot y) \bmod 2^w) \tag{2.15}$$

We claim that the bit-level representation of the product operation is identical for both unsigned and two's-complement multiplication. That is, given bit vectors $\vec{x}$ and $\vec{y}$ of length w, the bit-level representation of the unsigned product $\mathit{B2U}_w(\vec{x}) *^{\mathtt{u}}_{w} \mathit{B2U}_w(\vec{y})$ is identical to the bit-level representation of the two's-complement product $\mathit{B2T}_w(\vec{x}) *^{\mathtt{t}}_{w} \mathit{B2T}_w(\vec{x})$. This implies that the machine can use a single type of multiply instruction to multiply both signed and unsigned integers.

To see this, let $x = \mathit{B2T}_w(\vec{x})$ and $y = \mathit{B2T}_w(\vec{y})$ be the two's-complement values denoted by these bit patterns, and let $x' = \mathit{B2U}_w(\vec{x})$ and $y' = \mathit{B2U}_w(\vec{y})$ be the unsigned values. From Equation 2.3, we have $x' = x + x_{w-1}2^w$, and $y' = y + y_{w-1}2^w$. Computing the product of these values modulo 2^w gives the following:

$$\begin{aligned}
(x' \cdot y') \bmod 2^w \quad &= \quad [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w && (2.16)\\
&= \quad [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w && (2.17)\\
&= \quad (x \cdot y) \bmod 2^w && (2.18)
\end{aligned}$$

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
Unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
Two's Comp.	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [100]
Two's Comp.	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [001]
Two's Comp.	3 [011]	3 [011]	9 [001001]	1 [001]

Figure 2.21: **Three-bit unsigned and two's-complement multiplication examples.** Although the bit-level representations of the full products may differ, those of the truncated products are identical.

Thus, the low-order w bits of $x \cdot y$ and $x' \cdot y'$ are identical.

As illustrations, Figure 2.21 shows the results of multiplying different three-bit numbers. For each pair of bit-level operands, we perform both unsigned and two's-complement multiplication. Note that the unsigned truncated product always equals $x \cdot y \bmod 8$, and that the bit-level representations of both truncated products are identical.

Practice Problem 2.27:

Fill in the following table showing the results of multiplying different three-bit numbers, in the style of Figure 2.21:

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
Unsigned	[110]	[010]		
Two's Comp.	[110]	[010]		
Unsigned	[001]	[111]		
Two's Comp.	[001]	[111]		
Unsigned	[111]	[111]		
Two's Comp.	[111]	[111]		

We can see that unsigned arithmetic and two's-complement arithmetic over w-bit numbers are isomorphic—the operations $+^{\mathrm{u}}_{w}$, $-^{\mathrm{u}}_{w}$, and $*^{\mathrm{u}}_{w}$ have the exact same effect at the bit level as do $+^{\mathrm{t}}_{w}$, $-^{\mathrm{t}}_{w}$, and $*^{\mathrm{t}}_{w}$. From this, we can deduce that two's-complement arithmetic forms a ring $\langle\{-2^{w-1}, \ldots, 2^{w-1}-1\}, +^{\mathrm{t}}_{w}, *^{\mathrm{t}}_{w}, -^{\mathrm{t}}_{w}, 0, 1\rangle$.

2.3.6 Multiplying by Powers of Two

On most machines, the integer multiply instruction is fairly slow, requiring 12 or more clock cycles, whereas other integer operations—such as addition, subtraction, bit-level operations, and shifting—require only one clock cycle. As a consequence, one important optimization used by compilers is to attempt to replace multiplications by constant factors with combinations of shift and addition operations.

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, we claim the bit-level representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k 0s have been added to the right. This property can be derived using Equation 2.1:

$$
\begin{aligned}
B2U_{w+k}([x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\
&= \left[\sum_{i=0}^{w-1} x_i 2^i \right] \cdot 2^k \\
&= x2^k
\end{aligned}
$$

For $k < w$, we can truncate the shifted bit vector to be of length w, giving $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. By Equation 2.7, this bit-vector has numeric value $x2^k \bmod 2^w = x \ast^{\mathrm{u}}_{w} 2^k$. Thus, for unsigned variable `x`, the C expression `x << k` is equivalent to `x * pwr2k`, where `pwr2k` equals 2^{k}. In particular, we can compute `pwr2k` as `1U << k`.

By similar reasoning, we can show that for a two's-complement number x having bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and any k in the range $0 \leq k < w$, bit pattern $[x_{w-k-1}, \ldots, x_0, 0, \ldots, 0]$ will be the two's-complement representation of $x \ast^{\mathrm{t}}_{w} 2^k$. Therefore, for signed variable `x` , the C expression `x << k` is equivalent to `x * pwr2k`, where `pwr2k` equals 2^{k}.

Note that multiplying by a power of 2 can cause overflow with either unsigned or two's-complement arithmetic. Our result shows that even then we will get the same effect by shifting.

Practice Problem 2.28:

As we will see in Chapter 3, the `leal` instruction on an Intel-compatible processor can perform computations of the form `a<<k + b`, where `k` is either 0, 1, or 2, and `b` is either 0 or some program value. The compiler often uses this instruction to perform multiplications by constant factors. For example, we can compute `3*a` as `a<<1 + a`.

What multiples of `a` can be computed with this instruction?

2.3.7 Dividing by Powers of Two

Integer division on most machines is even slower than integer multiplication—requiring 30 or more clock cycles. Dividing by a power of 2 can also be performed using shift operations, but we use a right shift rather than a left shift. The two different shifts—logical and arithmetic—serve this purpose for unsigned and two's-complement numbers, respectively.

Integer division always rounds toward zero. For $x \geq 0$ and $y > 0$, the result should be $\lfloor x/y \rfloor$, where for any real number a, $\lfloor a \rfloor$ is defined to be the unique integer a' such that $a' \leq a < a' + 1$. As examples $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$, and $\lfloor 3 \rfloor = 3$.

Consider the effect of performing a logical right shift on an unsigned number. Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \leq k < w$. Let x' be the unsigned number with $w-k$-bit representation $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and x'' be the unsigned number with k-bit representation $[x_{k-1}, \ldots, x_0]$. We claim that $x' = \lfloor x/2^k \rfloor$. To see this, by Equation 2.1, we have

$x = \sum_{i=0}^{w-1} x_i 2^i$, $x' = \sum_{i=k}^{w-k-1} x_i 2^{i-k}$ and $x'' = \sum_{i=0}^{k-1} x_i 2^i$. We can therefore write x as $x = 2^k x' + x''$. Observe that $0 \leq x'' \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$, and hence $0 \leq x'' < 2^k$, implying that $\lfloor x''/2^k \rfloor = 0$. Therefore, $\lfloor x/2^k \rfloor = \lfloor x' + x''/2^k \rfloor = x' + \lfloor x''/2^k \rfloor = x'$.

Observe that performing a logical right shift of bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ by k yields the bit vector

$$[0, ..., 0, x_{w-1}, x_{w-2}, \ldots, x_k].$$

This bit vector has numeric value x'. That is, logically right shifting an unsigned number by k is equivalent to dividing it by 2^k. Therefore, for unsigned variable `x`, the C expression `x >> k` is equivalent to `x / pwr2k`, where `pwr2k` equals 2^{k}.

Now consider the effect of performing an arithmetic right shift on a two's-complement number. Let x be the two's-complement integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \leq k < w$. Let x' be the two's-complement number represented by the $w - k$ bits $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and x'' be the *unsigned* number represented by the low-order k bits $[x_{k-1}, \ldots, x_0]$. By a similar analysis as the unsigned case, we have $x = 2^k x' + x''$, and $0 \leq x'' < 2^k$, giving $x' = \lfloor x/2^k \rfloor$. Furthermore, observe that shifting bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ right *arithmetically* by k yields the bit vector

$$[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_k],$$

which is the sign extension from $w - k$ bits to w bits of $[x_{w-1}, x_{w-2}, \ldots, x_k]$. Thus, this shifted bit vector is the two's-complement representation of $\lfloor x/y \rfloor$.

For $x \geq 0$, our analysis shows that this shifted result is the desired value. For $x < 0$ and $y > 0$, however, the result of integer division should be $\lceil x/y \rceil$, where for any real number a, $\lceil a \rceil$ is defined to be the unique integer a' such that $a' - 1 < a \leq a'$. That is, integer division should round negative results upward toward zero. For example, the C expression `-5/2` yields `-2`. Thus, right shifting a negative number by k is not equivalent to dividing it by 2^k when rounding occurs. For example, the four-bit representation of -5 is $[1011]$. If we shift it right by one arithmetically we get $[1101]$, which is the two's-complement representation of -3.

We can correct for this improper rounding by "biasing" the value before shifting. This technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. Thus, for $x < 0$, if we first add $2^k - 1$ to x before right shifting, we will get a correctly rounded result. This analysis shows that for a two's-complement machine using arithmetic right shifts, the C expression `(x<0 ? (x + (1<<k)-1) : x) >> k` is equivalent to `x/pwr2k`, where `pwr2k` equals 2^{k}. For example, to divide -5 by 2, we first add bias $2 - 1 = 1$ giving bit pattern $[1100]$. Right shifting this by one arithmetically gives bit pattern $[1110]$, which is the two's-complement representation of -2.

Practice Problem 2.29:

In the following code, we have omitted the definitions of constants `M` and `N`:

```
#define M      /* Mystery number 1 */
#define N      /* Mystery number 2 */
int arith(int x, int y)
{
  int result = 0;
  result = x*M + y/N; /* M and N are mystery numbers. */
  return result;
}
```

We compiled this code for particular values of `M` and `N`. The compiler optimized the multiplication and division using the methods we have discussed. The following is a translation of the generated machine code back into C:

```
/* Translation of assembly code for arith */
int optarith(int x, int y)
{
  int t = x;
  x <<= 4;
  x -= t;
  if (y < 0) y += 3;
  y >>= 2;  /* Arithmetic shift */
  return x+y;
}
```

What are the values of `M` and `N`?

Practice Problem 2.30:

Assume we are running code on a 32-bit machine using two's-complement arithmetic for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

```
    int x = foo();   /* Arbitrary value */
    int y = bar();   /* Arbitrary value */

    unsigned ux = x;
    unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to 1) for all values of `x` and `y` or (2) give example values of `x` and `y` for which it is false (evaluates to 0):

A. `(x >= 0) || ((2*x) < 0)`

B. `(x & 7) != 7 || (x<<30 < 0)`

C. `(x * x) >= 0`

D. `x < 0 || -x <= 0`

E. `x > 0 || -x >= 0`

F. `x*y == ux*uy`

G. `~x*y + uy*ux == -y`

2.4 Floating Point

Floating-point representation encodes rational numbers of the form $V = x \times 2^y$. It is useful for performing computations involving very large numbers ($|V| \gg 0$), numbers very close to 0 ($|V| \ll 1$), and more generally as an approximation to real arithmetic.

Up until the 1980s, every computer manufacturer devised its own conventions for how floating-point numbers were represented and the details of the operations performed on them. In addition, they often did not worry too much about the accuracy of the operations, viewing speed and ease of implementation as being more critical than numerical precision.

All of this changed around 1985 with the advent of IEEE Standard 754, a carefully crafted standard for representing floating-point numbers and the operations performed on them. This effort started in 1976 under Intel's sponsorship with the design of the 8087, a chip that provided floating-point support for the 8086 processor. They hired William Kahan, a professor at the University of California, Berkeley, as a consultant to help design a floating-point standard for its future processors. They allowed Kahan to join forces with a committee generating an industry-wide standard under the auspices of the Institute of Electrical and Electronics Engineers (IEEE). The committee ultimately adopted a standard close to the one Kahan had devised for Intel. Nowadays virtually all computers support what has become known as *IEEE floating point*. This has greatly improved the portability of scientific application programs across different machines.

Aside: The IEEE.
The Institute of Electrical and Electronic Engineers (IEEE—pronounced "I-Triple-E") is a professional society that encompasses all of electronic and computer technology. It publishes journals, sponsors conferences, and sets up committees to define standards on topics ranging from power transmission to software engineering. **End Aside.**

In this section, we will see how numbers are represented in the IEEE floating-point format. We will also explore issues of *rounding*, when a number cannot be represented exactly in the format and hence must be adjusted upward or downward. We will then explore the mathematical properties of addition, multiplication, and relational operators. Many programmers consider floating point to be at best uninteresting and at worst arcane and incomprehensible. We will see that since the IEEE format is based on a small and consistent set of principles, it is really quite elegant and understandable.

2.4.1 Fractional Binary Numbers

A first step in understanding floating-point numbers is to consider binary numbers having fractional values. Let us first examine the more familiar decimal notation. Decimal notation uses a representation of the form: $d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$, where each decimal digit d_i ranges between 0 and 9. This notation represents a value d defined as

$$d \;=\; \sum_{i=-n}^{m} 10^i \times d_i$$

The weighting of the digits is defined relative to the decimal point symbol '.' meaning that digits to the left are weighted by positive powers of 10, giving integral values, while digits to the right are weighted by negative powers of 10, giving fractional values. For example, 12.34_{10} represents the number $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12\frac{34}{100}$.

By analogy, consider a notation of the form $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$, where each binary digit, or bit, b_i ranges between 0 and 1. This notation represents a number b defined as

$$b \;=\; \sum_{i=-n}^{m} 2^i \times b_i \tag{2.19}$$

The symbol '.' now becomes a *binary point*, with bits on the left being weighted by positive powers of two, and those on the right being weighted by negative powers of two. For example, 101.11_2 represents the number $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$,

One can readily see from Equation 2.19 that shifting the binary point one position to the left has the effect of dividing the number by two. For example, while 101.11_2 represents the number $5\frac{3}{4}$, 10.111_2 represents the number $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$. Similarly, shifting the binary point one position to the right has the effect of multiplying the number by two. For example, 1011.1_2 represents the number $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Note that numbers of the form $0.11 \cdots 1_2$ represent numbers just below 1. For example, 0.111111_2 represents $\frac{63}{64}$. We will use the shorthand notation $1.0 - \epsilon$ to represent such values.

Assuming we consider only finite-length encodings, decimal notation cannot represent numbers such as $\frac{1}{3}$ and $\frac{5}{7}$ exactly. Similarly, fractional binary notation can only represent numbers that can be written $x \times 2^y$. Other values can only be approximated. For example, although the number $\frac{1}{5}$ can be approximated with increasing accuracy by lengthening the binary representation, we cannot represent it exactly as a fractional binary number:

Representation	Value	Decimal
0.0_2	0	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

Practice Problem 2.31:

Fill in the missing information in the following table:

Fractional value	Binary representation	Decimal representation
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$		
$\frac{23}{16}$		
	10.1101	
	1.011	
		5.375
		3.0625

Practice Problem 2.32:

The imprecision of floating-point arithmetic can have disastrous effects. On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. The U. S. General Accounting Office (GAO) conducted a detailed analysis of the failure [53] and determined that the underlying cause was an imprecision in a numeric calculation. In this exercise, you will reproduce part of the GAO's analysis.

The Patriot system contains an internal clock, implemented as a counter that is incremented every 0.1 seconds. To determine the time in seconds, the program would multiply the value of this counter by a 24-bit quantity that was a fractional binary approximation to $\frac{1}{10}$. In particular, the binary representation of $\frac{1}{10}$ is the nonterminating sequence

$$0.000110011[0011]\cdots_2$$

where the portion in brackets is repeated indefinitely. The computer approximated 0.1 using just the leading bit plus the first 23 bits of this sequence to the right of the binary point. Let us call this number x.

A. What is the binary representation of $x - 0.1$?

B. What is the approximate decimal value of $x - 0.1$?

C. The clock starts at 0 when the system is first powered up and keeps counting up from there. In this case, the system had been running for around 100 hours. What was the difference between the time computed by the software and the actual time?

D. The system predicts where an incoming missile will appear based on its velocity and the time of the last radar detection. Given that a Scud travels at around 2,000 meters per second, how far off was its prediction?

Normally, a slight error in the absolute time reported by a clock reading would not affect a tracking computation. Instead, it should depend on the relative time between two successive readings. The problem was that the Patriot software had been upgraded to use a more accurate function for reading time, but not all of the function calls had been replaced by the new code. As a result, the tracking software used the accurate time for one reading and the inaccurate time for the other [73].

2.4.2 IEEE Floating-Point Representation

Positional notation such as considered in the previous section would not be efficient for representing very large numbers. For example, the representation of 5×2^{100} would consist of the bit pattern 101 followed by 100 zeros. Instead, we would like to represent numbers in a form $x \times 2^y$ by giving the values of x and y.

The IEEE floating-point standard represents a number in a form $V = (-1)^s \times M \times 2^E$:

- The *sign* s determines whether the number is negative ($s = 1$) or positive ($s = 0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The *significand* M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The *exponent* E weights the value by a (possibly negative) power of 2.

The bit representation of a floating-point number is divided into three fields to encode these values:

- The single sign bit `s` directly encodes the sign s.
- The k-bit exponent field `exp` $= e_{k-1} \cdots e_1 e_0$ encodes the exponent E.
- The n-bit fraction field `frac` $= f_{n-1} \cdots f_1 f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

In the single-precision floating-point format (a `float` in C), fields `s`, `exp`, and `frac` are 1, $k = 8$, and $n = 23$ bits each, yielding a 32-bit representation. In the double-precision floating-point format (a `double` in C), fields `s`, `exp`, and `frac` are 1, $k = 11$, and $n = 52$ bits each, yielding a 64-bit representation.

The value encoded by a given bit representation can be divided into three different cases, depending on the value of `exp`.

Normalized Values

This is the most common case. These kinds occur when the bit pattern of `exp` is neither all 0s (numeric value 0) nor all 1s (numeric value 255 for single precision, 2047 for double). In this case, the exponent field is interpreted as representing a signed integer in *biased* form. That is, the exponent value is $E = e - \mathit{Bias}$ where e is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$, and Bias is a bias value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double). This yields exponent ranges from -126 to $+127$ for single precision and -1022 to $+1023$ for double precision.

The fraction field `frac` is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the binary point to the left of the most significant bit. The significand is defined to be $M = 1 + f$. This is sometimes called an *implied leading 1* representation, because we can view M to be the number with binary representation $1.f_{n-1} f_{n-2} \cdots f_0$. This representation is a trick for getting an additional bit of precision for free, since we can always adjust the exponent E so that significand M is in the range $1 \leq M < 2$ (assuming there is no overflow). We therefore do not need to explicitly represent the leading bit, since it always equals 1.

Denormalized Values

When the exponent field is all 0s, the represented number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$, and the significand value is $M = f$, that is, the value of the fraction field without an implied leading 1.

Aside: Why set the bias this way for denormalized values?
Having the exponent value be $1 - Bias$ rather than simply $-Bias$ might seem counterintuitive. We will see shortly that it provides for smooth transition from denormalized to normalized values.**End Aside.**

Denormalized numbers serve two purposes. First, they provide a way to represent numeric value 0, since with a normalized number we must always have $M \geq 1$, and hence we cannot represent 0. In fact the floating-point representation of $+0.0$ has a bit pattern of all 0s: the sign bit is 0, the exponent field is all 0s (indicating a denormalized value), and the fraction field is all 0s, giving $M = f = 0$. Curiously, when the sign bit is 1, but the other fields are all 0s, we get the value -0.0. With IEEE floating-point format, the values -0.0 and $+0.0$ are considered different in some ways and the same in others.

A second function of denormalized numbers is to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* in which possible numeric values are spaced evenly near 0.0.

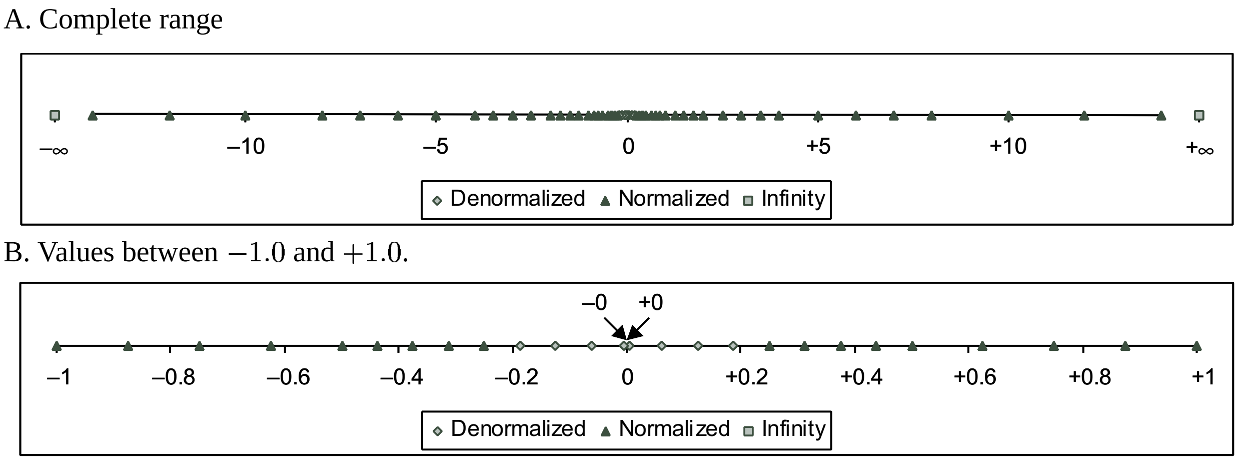
Special Values

A final category of values occurs when the exponent field is all 1s. When the fraction field is all 0s, the resulting values represent infinity, either $+\infty$ when $s = 0$, or $-\infty$ when $s = 1$. Infinity can represent results that *overflow*, as when we multiply two very large numbers, or when we divide by zero. When the fraction field is nonzero, the resulting value is called a "NaN," short for "Not a Number." Such values are returned as the result of an operation where the result cannot be given as a real number or as infinity, as when computing $\sqrt{-1}$ or $\infty - \infty$. They can also be useful in some applications for representing uninitialized data.

2.4.3 Example Numbers

Figure 2.22 shows the set of values that can be represented in a hypothetical 6-bit format having $k = 3$ exponent bits and $n = 2$ significand bits. The bias is $2^{3-1} - 1 = 3$. Part A of the figure shows all representable values (other than NaN). The two infinities are at the extreme ends. The normalized numbers with maximum magnitude are ± 14. The denormalized numbers are clustered around 0. These can be seen more clearly in part B of the figure, where we show just the numbers between -1.0 and $+1.0$. The two zeros are special cases of denormalized numbers. Observe that the representable numbers are not uniformly distributed—they are denser nearer the origin.

Figure 2.23 shows some examples for a hypothetical eight-bit floating-point format having $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is $2^{4-1} - 1 = 7$. The figure is divided into three regions representing the three classes of numbers. Closest to 0 are the denormalized numbers, starting with 0 itself. Denormalized numbers in this format have $E = 1 - 7 = -6$, giving a weight $2^E = \frac{1}{64}$. The fractions f range over the values $0, \frac{1}{8}, \ldots, \frac{7}{8}$, giving numbers V in the range 0 to $\frac{7}{8 \times 64} = \frac{7}{512}$.


A. Complete range

B. Values between -1.0 and $+1.0$.

Figure 2.22: **Representable values for six-bit floating-point format.** There are $k = 3$ exponent bits and $n = 2$ significand bits. The bias is 3.

Description	Bit representation	e	E	f	M	V
Zero	`0 0000 000`	0	-6	0	0	0
Smallest pos.	`0 0000 001`	0	-6	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$
	`0 0000 010`	0	-6	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$
	`0 0000 011`	0	-6	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$
	...					
	`0 0000 110`	0	-6	$\frac{6}{8}$	$\frac{6}{8}$	$\frac{6}{512}$
Largest denorm.	`0 0000 111`	0	-6	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$
Smallest norm.	`0 0001 000`	1	-6	0	$\frac{8}{8}$	$\frac{8}{512}$
	`0 0001 001`	1	-6	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$
	...					
	`0 0110 110`	6	-1	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$
	`0 0110 111`	6	-1	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$
One	`0 0111 000`	7	0	0	$\frac{8}{8}$	1
	`0 0111 001`	7	0	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	`0 0111 010`	7	0	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$
	...					
	`0 1110 110`	14	7	$\frac{6}{8}$	$\frac{14}{8}$	224
Largest norm.	`0 1110 111`	14	7	$\frac{7}{8}$	$\frac{15}{8}$	240
Infinity	`0 1111 000`	–	–	–	–	$+\infty$

Figure 2.23: **Example nonnegative values for eight-bit floating-point format.** There are $k = 4$ exponent bits and $n = 3$ significand bits. The bias is 7.

The smallest normalized numbers in this format also have $E = 1 - 7 = -6$, and the fractions also range over the values $0, \frac{1}{8}, \ldots \frac{7}{8}$. However, the significands then range from $1 + 0 = 1$ to $1 + \frac{7}{8} = \frac{15}{8}$, giving numbers V in the range $\frac{8}{512}$ to $\frac{15}{512}$.

Observe the smooth transition between the largest denormalized number $\frac{7}{512}$ and the smallest normalized number $\frac{8}{512}$. This smoothness is due to our definition of E for denormalized values. By making it $1 - Bias$ rather than $-Bias$, we compensate for the fact that the significand of a denormalized number does not have an implied leading 1.

As we increase the exponent, we get successively larger normalized values, passing through 1.0 and then to the largest normalized number. This number has exponent $E = 7$, giving a weight $2^E = 128$. The fraction equals $\frac{7}{8}$ giving a significand $M = \frac{15}{8}$. Thus, the numeric value is $V = 240$. Going beyond this overflows to $+\infty$.

One interesting property of this representation is that if we interpret the bit representations of the values in Figure 2.23 as unsigned integers, they occur in ascending order, as do the values they represent as floating-point numbers. This is no accident—the IEEE format was designed so that floating-point numbers could be sorted using an integer-sorting routine. A minor difficulty occurs when dealing with negative numbers, since they have a leading 1, and they occur in descending order, but this can be overcome without requiring floating-point operations to perform comparisons (see Problem 2.56).

Practice Problem 2.33:

Consider a five-bit floating-point representation based on the IEEE floating-point format, with one sign bit, two exponent bits ($k = 2$), and two fraction bits ($n = 2$). The exponent bias is $2^{2-1} - 1 = 1$.

The table that follows enumerates the entire nonnegative range for this five-bit floating-point representation. Fill in the blank table entries using the following directions:

e**:** The value represented by considering the exponent field to be an unsigned integer.

E**:** The value of the exponent after biasing.

f**:** The value of the fraction.

M**:** The value of the significand.

V**:** The numeric value represented.

Express the values of f, M and V as fractions of the form $\frac{x}{4}$. You need not fill in entries marked “—”.

Description	exp	frac	Single precision		Double precision	
			Value	Decimal	Value	Decimal
Zero	$00 \cdots 00$	$0 \cdots 00$	0	0.0	0	0.0
Smallest denorm.	$00 \cdots 00$	$0 \cdots 01$	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00 \cdots 00$	$1 \cdots 11$	$(1-\epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon) \times 2^{-1022}$	2.2×10^{-308}
Smallest norm.	$00 \cdots 01$	$0 \cdots 00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01 \cdots 11$	$0 \cdots 00$	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	$11 \cdots 10$	$1 \cdots 11$	$(2-\epsilon) \times 2^{127}$	3.4×10^{38}	$(2-\epsilon) \times 2^{1023}$	1.8×10^{308}

Figure 2.24: **Examples of nonnegative floating-point numbers.**

Bits	e	E	f	M	V
0 00 00					
0 00 01					
0 00 10					
0 00 11					
0 01 00					
0 01 01					
0 01 10					
0 01 11					
0 10 00	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
0 10 01					
0 10 10					
0 10 11					
0 11 00	—	—	—	—	$+\infty$
0 11 01	—	—	—	—	NaN
0 11 10	—	—	—	—	NaN
0 11 11	—	—	—	—	NaN

Figure 2.24 shows the representations and numeric values of some important single and double-precision floating-point numbers. As with the eight-bit format shown in Figure 2.23 we can see some general properties for a floating-point representation with a k-bit exponent and an n-bit fraction:

- The value $+0.0$ always has a bit representation of all 0s.
- The smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all 0s. It has a fraction (and significand) value $M = f = 2^{-n}$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-n-2^{k-1}+2}$.
- The largest denormalized value has a bit representation consisting of an exponent field of all 0s and a fraction field of all 1s. It has a fraction (and significand) value $M = f = 1 - 2^{-n}$ (which we have written $1 - \epsilon$) and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$, which is just slightly smaller than the smallest normalized value.

- The smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all 0s. It has a significand value $M = 1$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-2^{k-1}+2}$.

- The value 1.0 has a bit representation with all but the most significant bit of the exponent field equal to 1 and all other bits equal to 0. Its significand value is $M = 1$ and its exponent value is $E = 0$.

- The largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1. It has a fraction value of $f = 1 - 2^{-n}$, giving a significand $M = 2 - 2^{-n}$ (which we have written $2 - \epsilon$). It has an exponent value $E = 2^{k-1} - 1$, giving a numeric value $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

One useful exercise for understanding floating-point representations is to convert sample integer values into floating-point form. For example, we saw in Figure 2.10 that 12,345 has binary representation $[11000000111001]$. We create a normalized representation of this by shifting 13 positions to the right of a binary point, giving $12345 = 1.1000000111001_2 \times 2^{13}$. To encode this in IEEE single precision format, we construct the fraction field by dropping the leading 1 and adding 10 0s to the end, giving binary representation $[10000001110010000000000]$. To construct the exponent field, we add bias 127 to 13, giving 140, which has binary representation $[10001100]$. We combine this with a sign bit of 0 to get the floating-point representation in binary of $[01000110010000001110010000000000]$. Recall from Section 2.1.4 that we observed the following correlation in the bit-level representations of the integer value `12345` (`0x3039`) and the single-precision floating-point value `12345.0` (`0x4640E400`):

```
   0   0   0   0   3   0   3   9
00000000000000000011000000111001
                   *************
             4   6   4   0   E   4   0   0
          01000110010000001110010000000000
```

We can now see that the region of correlation corresponds to the low-order bits of the integer, stopping just before the most significant bit equal to 1 (this bit forms the implied leading 1), matching the high-order bits in the fraction part of the floating-point representation.

Practice Problem 2.34:

As mentioned in Practice problem 2.6, the integer 3490593 has hexadecimal representation `0x354321`, while the single-precision, floating-point number 3490593.0 has hexadecimal representation `0x4A550C84`. Derive this floating-point representation and explain the correlation between the bits of the integer and floating-point representations.

Practice Problem 2.35:

A. For a floating-point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an $n + 1$-bit fraction to be exact).

B. What is the numeric value of this integer for single-precision format ($k = 8$, $n = 23$)?

Mode	$1.40	$1.60	$1.50	$2.50	$–1.50
Round-to-even	$1	$2	$2	$2	$–2
Round-toward-zero	$1	$1	$1	$2	$–1
Round-down	$1	$1	$1	$2	$–2
Round-up	$2	$2	$2	$3	$–1

Figure 2.25: **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

2.4.4 Rounding

Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value x, we generally want a systematic method of finding the "closest" matching value x' that can be represented in the desired floating-point format. This is the task of the *rounding* operation. The key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have $1.50 and want to round it to the nearest dollar, should the result be $1 or $2? An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values x^- and x^+ such that the value x is guaranteed to lie between them: $x^- \leq x \leq x^+$. The IEEE floating-point format defines four different *rounding modes*. The default method finds a closest match, while the other three can be used for computing upper and lower bounds.

Figure 2.25 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even (also called round-to-nearest) is the default mode. It attempts to find a closest match. Thus, it rounds $1.40 to $1 and $1.60 to $2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-even mode adopts the convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both $1.50 and $2.50 to $2.

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications. Round-toward-zero mode rounds positive numbers downward and negative numbers upward, giving a value $\hat{x}$ such that $|\hat{x}| \leq |x|$. Round-down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since four is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern of the form $XX \cdots X.YY \cdots Y100 \cdots$, where X and Y denote arbitary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point). We would round 10.00011_2 ($2\frac{3}{32}$) down to 10.00_2 (2), and 10.00110_2 ($2\frac{3}{16}$) up to 10.01_2 ($2\frac{1}{4}$), because these values are not halfway between two possible values. We would round 10.11100_2 ($2\frac{7}{8}$) up to 11.00_2 (3) and 10.10100_2 down to 10.10_2 ($2\frac{1}{2}$), since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

2.4.5 Floating-Point Operations

The IEEE standard specifies a simple rule for determining the result of an arithmetic operation such as addition or multiplication. Viewing floating-point values x and y as real numbers, and some operation $\odot$ defined over real numbers, the computation should yield $Round(x \odot y)$, the result of applying rounding to the exact result of the real operation. In practice, there are clever tricks floating-point unit designers use to avoid performing this exact computation, since the computation need only be sufficiently precise to guarantee a correctly rounded result. When one of the arguments is a special value such as -0, ∞ or NaN, the standard specifies conventions that attempt to be reasonable. For example $1/-0$ is defined to yield $-\infty$, while $1/+0$ is defined to yield $+\infty$.

One strength of the IEEE standard's method of specifying the behavior of floating-point operations is that it is independent of any particular hardware or software realization. Thus, we can examine its abstract mathematical properties without considering how it is actually implemented.

We saw earlier that integer addition, both unsigned and two's-complement, forms an Abelian group. Addition over real numbers also forms an Abelian group, but we must consider what effect rounding has on these properties. Let us define $x +^{\mathrm{f}} y$ to be $Round(x + y)$. This operation is defined for all values of x and y, although it may yield infinity even when both x and y are real numbers due to overflow. The operation is commutative, with $x +^{\mathrm{f}} y = y +^{\mathrm{f}} x$ for all values of x and y. On the other hand, the operation is not associative. For example, with single-precision floating point the expression `(3.14+1e10)-1e10` would evaluate to `0.0`—the value 3.14 would be lost due to rounding. On the other hand, the expression `3.14+(1e10-1e10)` would evaluate to `3.14`. As with an Abelian group, most values have inverses under floating-point addition, that is, $x +^{\mathrm{f}} -x = 0$. The exceptions are infinities (since $+\infty - \infty = NaN$), and NaN's, since $NaN +^{\mathrm{f}} x = NaN$ for any x.

The lack of associativity in floating-point addition is the most important group property that is lacking. It has important implications for scientific programmers and compiler writers. For example, suppose a compiler is given the following code fragment:

```
x = a + b + c;
```

```
y = b + c + d;
```

The compiler might be tempted to save one floating-point addition by generating the following code:

```
t = b + c;
x = a + t;
y = t + d;
```

However, this computation might yield a different value for `x` than would the original, since it uses a different association of the addition operations. In most applications, the difference would be so small as to be inconsequential. Unfortunately, compilers have no way of knowing what trade-offs the user is willing to make between efficiency and faithfulness to the exact behavior of the original program. As a result, they tend to be very conservative, avoiding any optimizations that could have even the slightest effect on functionality.

On the other hand, floating-point addition satisfies the following monotonicity property: if $a \geq b$ then $x + a \geq x + b$ for any values of a, b, and x other than NaN. This property of real (and integer) addition is not obeyed by unsigned or two's-complement addition.

Floating-point multiplication also obeys many of the properties one normally associates with multiplication, namely those of a ring. Let us define $x \ {*}^{\mathrm{f}}\ y$ to be $Round(x \times y)$. This operation is closed under multiplication (although possibly yielding infinity or NaN), it is commutative, and it has 1.0 as a multiplicative identity. On the other hand, it is not associative due to the possibility of overflow or the loss of precision due to rounding. For example, with single-precision floating point, the expression `(1e20*1e20)*1e-20` will evaluate to $+\infty$, while `1e20*(1e20*1e-20)` will evaluate to `1e20`. In addition, floating-point multiplication does not distribute over addition. For example, with single-precision floating point, the expression `1e20*(1e20-1e20)` will evaluate to `0.0`, while `1e20*1e20-1e20*1e20` will evaluate to `NaN`.

On the other hand, floating-point multiplication satisfies the following monotonicity properties for any values of a, b, and c other than NaN:

$$\begin{array}{lcl} a \geq b \text{ and } c \geq 0 & \Rightarrow & a \ {*}^{\mathrm{f}}\ c \geq b \ {*}^{\mathrm{f}}\ c \\ a \geq b \text{ and } c \leq 0 & \Rightarrow & a \ {*}^{\mathrm{f}}\ c \leq b \ {*}^{\mathrm{f}}\ c \end{array}$$

In addition, we are also guaranteed that $a \ {*}^{\mathrm{f}}\ a \geq 0$, as long as $a \neq NaN$. As we saw earlier, none of these monotonicity properties hold for unsigned or two's-complement multiplication.

This lack of associativity and distributivity is of serious concern to scientific programmers and to compiler writers. Even such a seemingly simple task as writing code to determine whether two lines intersect in three-dimensional space can be a major challenge.

2.4.6 Floating Point in C

C provides two different floating-point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single- and double-precision floating point. In addition, the machines use the round-to-even rounding mode. Unfortunately, since the C standard does require the machine use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as -0, $+\infty$, $-\infty$, or NaN. Most systems provide a combination of include (‘.h’) files and procedure libraries to provide access to these features, but the details vary from one system to another. For example, the GNU compiler GCC defines macros INFINITY (for $+\infty$) and NAN (for NaN) when the following sequence occurs in the program file:

```
#define _GNU_SOURCE 1
#include <math.h>
```

Practice Problem 2.36:

Fill in the following macro definitions to generate the double-precision values $+\infty$, $-\infty$, and 0.

```
#define POS_INFINITY
#define NEG_INFINITY
#define NEG_ZERO
#endif
```

You cannot use any include files (such as math.h), but you can make use of the fact that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

When casting values between int, float, and double formats, the program changes the numeric values and the bit representations as follows (assuming a 32-bit int):

- From int to float, the number cannot overflow, but it may be rounded.
- From int or float to double, the exact numeric value can be preserved because double has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From double to float, the value can overflow to $+\infty$ or $-\infty$, since the range is smaller. Otherwise, it may be rounded, because the precision is smaller.
- From float or double to int the value will be truncated toward zero. For example, 1.999 will be converted to 1, while -1.999 will be converted to -1. Note that this behavior is very different from rounding. Furthermore, the value may overflow. The C standard does not specify a fixed result for this case, but on most machines the result will either be $TMax_w$ or $TMin_w$, where w is the number of bits in an int.

*Intel IA32 Floating-Point Arithmetic

In the next chapter, we will begin an in-depth study of Intel IA32 processors, the processor found in most of today’s personal computers. Here we highlight an idiosyncrasy of these machines that can seriously affect the behavior of programs operating on floating-point numbers when compiled with GCC.

IA32 processors, like most other processors, have special memory elements called *registers* for holding floating-point values as they are being computed and used. Values held in registers can be read and written more quickly than those held in the main memory. The unusual feature of IA32 is that the floating-point registers use a special 80-bit *extended-precision* format to provide a greater range and precision than the normal 32-bit single-precision and 64-bit double-precision formats used for values held in memory. As described in Homework Problem 2.58, the extended-precision representation is similar to an IEEE floating-point format with a 15-bit exponent (i.e., $k = 15$) and a 63-bit fraction (i.e., $n = 63$). All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single or double-precision format as they are stored in memory.

This extension to 80 bits for all register data and then contraction to a smaller format for all memory data has some undesirable consequences for programmers. It means that storing a value in memory and then retrieving it can change its value, due to rounding, underflow, or overflow. This storing and retrieving is not always visible to the C programmer, leading to some very peculiar results.

The following example illustrates this property:

code/data/fcomp.c

```
double recip(int denom)
{
  return 1.0/(double) denom;
}

void do_nothing() {} /* Just like the name says */

void test1(int denom)
{
  double r1, r2;
  int t1, t2;

  r1 = recip(denom);  /* Stored in memory              */
  r2 = recip(denom);  /* Stored in register            */
  t1 = r1 == r2;      /* Compares register to memory   */
  do_nothing();       /* Forces register save to memory */
  t2 = r1 == r2;      /* Compares memory to memory     */
  printf("test1 t1: r1 %f %c= r2 %f\n", r1, t1 ? ’=’ : ’!’, r2);
  printf("test1 t2: r1 %f %c= r2 %f\n", r1, t2 ? ’=’ : ’!’, r2);
}
```

code/data/fcomp.c

Variables `r1` and `r2` are computed by the same function with the same argument. One would expect them to be identical. Furthermore, both variables `t1` and `t2` are computing by evaluating the expression `r1 == r2`, and so we would expect them both to equal `1`. There are no apparent hidden side effects—function `recip` does a straightforward reciprocal computation, and, as the name suggests, function `do_nothing` does nothing. When the file is compiled with optimization flag ‘`-O2`’ and run with argument `10`, however, we get the following result:

```
test1 t1: r1 0.100000 != r2 0.100000
test1 t2: r1 0.100000 == r2 0.100000
```

The first test indicates the two reciprocals are different, while the second indicates they are the same! This is certainly not what we expect, nor what we want. Understanding all of the details of this example requires studying the machine-level floating-point code generated by GCC (see Section 3.14), but the comments in the code provide a clue as to why this outcome occurs. The value computed by function `recip` returns its result in a floating-point register. Whenever procedure `test1` calls some function, it must store any value currently in a floating-point register onto the main program stack, where local variables for a function are stored. In performing this store, the processor converts the extended-precision register values to double-precision memory values. Thus, before making the second call to `recip` (line 14), variable `r1` is converted and stored as a double-precision number. After the second call, variable `r2` has the extended-precision value returned by the function. In computing `t1` (line 15), the double-precision number `r1` is compared to the extended-precision number `r2`. Since 0.1 cannot be represented exactly in either format, the outcome of the test is false. Before calling function `do_nothing` (line 16), `r2` is converted and stored as a double-precision number. In computing `t2` (line 17), two double-precision numbers are compared, yielding true.

This example demonstrates a deficiency of GCC on IA32 machines (the same result occurs for both Linux and Microsoft Windows). The value associated with a variable changes due to operations that are not visible to the programmer, such as the saving and restoring of floating-point registers. Our experiments with the Microsoft Visual C++ compiler indicate that it does not have this problem.

Aside: Why should we be concerned about these inconsistencies?
As we will discuss in Chapter 5, one of the fundamental principles of optimizing compilers is that programs should produce the exact same results whether or not optimization is enabled. Unfortunately, GCC does not satisfy this requirement for floating-point code on IA32 machines. **End Aside.**

There are several ways to overcome this problem, although none are ideal. The simplest is to invoke GCC with the command-line option "`-ffloat-store`" indicating that the result of every floating-point computation should be stored to memory and read back before using, rather than simply held in a register. This will force every computed value to be converted to the lower-precision form. This slows down the program somewhat but makes the behavior more predictable. Unfortunately, we have found that GCC does not follow this write-then-read convention strictly, even when given the command-line option. For example, consider the following function:

code/data/fcomp.c

```
void test2(int denom)
{
  double r1;
  int t1;
  r1 = recip(denom);             /* Default: register, Forced store: memory */
  t1 = r1 == 1.0/(double) denom; /* Compares register or memory to register */
  printf("test2 t1: r1 %f %c= 1.0/10.0\n", r1, t1 ? ’=’ : ’!’);
}
```

code/data/fcomp.c

When compiled with just the "`-O2`" option, `t1` gets value 1—the comparison is made between two register values. When compiled with the "`-ffloat-store`" flag, `t1` gets value 0! Although the result of the call to `recip` is written to memory and read back into a register, the computed value `1.0/(double)` denom is kept in a register. Overall, we have found that seemingly minor changes in a program can cause these tests to succeed or fail in unpredictable ways.

As an alternative, we can have GCC use extended precision in all of its computations by declaring all of the variables to be long double as shown in the following code:

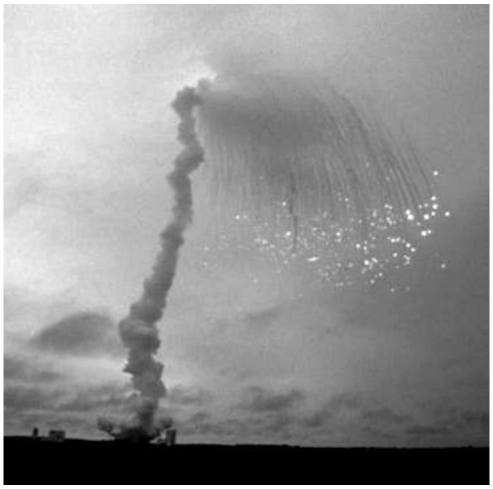
code/data/fcomp.c

```
long double recip_l(int denom)
{
  return 1.0/(long double) denom;
}

void test3(int denom)
{
  long double r1, r2;
  int t1, t2, t3;

  r1 = recip_l(denom); /* Stored in memory                */
  r2 = recip_l(denom); /* Stored in register              */
  t1 = r1 == r2;       /* Compares register to memory     */
  do_nothing();        /* Forces register save to memory */
  t2 = r1 == r2;       /* Compares memory to memory       */
  t3 = r1 == 1.0/(long double) denom; /* Compare memory to register */
  printf("test3 t1: r1 %f %c= r2 %f\n",
         (double) r1, t1 ? ’=’ : ’!’, (double) r2);
  printf("test3 t2: r1 %f %c= r2 %f\n",
         (double) r1, t2 ? ’=’ : ’!’, (double) r2);
  printf("test3 t3: r1 %f %c= 1.0/10.0\n",
         (double) r1, t2 ? ’=’ : ’!’);
}
```

code/data/fcomp.c

The declaration long double is allowed as part of the ANSI C standard, although for most machines and compilers, this declaration is equivalent to an ordinary double. For GCC on IA32 machines, however, it uses the extended-precision format for memory data as well as for floating point register data. This allows us to take full advantage of the wider range and greater precision provided by the extended-precision format while avoiding the anomalies we have seen in our earlier examples. Unfortunately, this solution comes at a price. GCC uses 12 bytes to store a long double, increasing memory consumption by 50%. (Although 10 bytes would suffice, it rounds this up to 12 to give a better memory performance. The same allocation is used on both Linux and Windows machines). Transferring these longer data between registers and memory takes more time, too. Still, this is the best option for programs that want to get the most accurate and predictable results.

Aside: Ariane 5: the high cost of floating-point overflow.



Converting large floating-point numbers to integers is a common source of programming errors. Such an error had disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after liftoff, the rocket veered off its flight path, broke up, and exploded. Communication satellites valued at $500 million were on board the rocket.

A later investigation [50] showed that the computer controlling the inertial navigation system had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that an overflow had occurred during the conversion of a 64-bit floating-point number to a 16-bit signed integer.

The value that overflowed measured the horizontal velocity of the rocket, which could be more than five times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based. **End Aside.**

Practice Problem 2.37:

Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or NaN. For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(float) x`

B. `x == (int)(double) x`

C. `f == (float)(double) f`

D. `d == (float) d`

E. `f == -(-f)`

F. `2/3 == 2/3.0`

G. `(d >= 0.0) || ((d*2) < 0.0)`

H. `(d+f)-d == f`

2.5 Summary

Computers encode information as bits, generally organized as sequences of bytes. Different encodings are used for representing integers, real numbers, and character strings. Different models of computers use different conventions for encoding numbers and for ordering the bytes within multibyte data.

The C language is designed to accomodate a wide range of different implementations in terms of word sizes and numeric encodings. Most current machines have 32-bit word sizes, although high-end machines increasingly have 64-bit words. Most machines use two's-complement encoding of integers and IEEE encoding of floating point. Understanding these encodings at the bit level, as well as understanding the mathematical characteristics of the arithmetic operations, is important for writing programs that operate correctly over the full range of numeric values.

The C standard dictates that when casting between signed and unsigned integers, the underlying bit pattern should not change. On a two's-complement machine, this behavior is characterized by functions $T2U_w$ and $U2T_w$, for a w-bit value. The implicit casting of C gives results that many programmers do not anticipate, often leading to program bugs.

Due to the finite lengths of the encodings, computer arithmetic has properties quite different from conventional integer and real arithmetic. The finite length can cause numbers to overflow, when they exceed the range of the representation. Floating-point values can also underflow, when they are so close to 0.0 that they are changed to zero.

The finite integer arithmetic implemented by C, as well as most other programming languages, has some peculiar properties compared to true integer arithmetic. For example, the expression `x*x` can evaluate to a negative number due to overflow. Nonetheless, both unsigned and two's-complement arithmetic satisfies the properties of a ring. This allows compilers to do many optimizations. For example, in replacing the expression `7*x` by `(x<<3)-x`, we make use of the associative, commutative and distributive properties, along with the relationship between shifting and multiplying by powers of two.

We have seen several clever ways to exploit combinations bit-level operations and arithmetic operations. For example, we saw that with two's-complement arithmetic, `~x+1` is equivalent to `-x`. As another example, suppose we want a bit pattern of the form $[0, \ldots, 0, 1, \ldots, 1]$, consisting of $w-k$ 0s followed by k 1s. Such bit patterns are useful for masking operations. This pattern can be generated by the C expression `(1<<k)-1`, exploiting the property that the desired bit pattern has numeric value $2^k - 1$. For example, the expression `(1<<8)-1` will generate the bit pattern `0xFF`.

Floating-point representations approximate real numbers by encoding numbers of the form $x \times 2^y$. The most common floating-point representation was defined by IEEE Standard 754. It provides for several different precisions, with the most common being single (32 bits) and double (64 bits). IEEE floating point also has representations for special values ∞ and not-a-number.

Floating-point arithmetic must be used very carefully, because it has only limited range and precision, and because it does not obey common mathematical properties such as associativity.

Bibliographic Notes

Reference books on C [41, 32] discuss properties of the different data types and operations. The C standard does not specify details such as precise word sizes or numeric encodings. Such details are intentionally omitted to make it possible to implement C on a wide range of different machines. Several books have been written giving advice to C programmers [42, 51] that warn about problems with overflow, implicit casting to unsigned, and some of the other pitfalls we have covered in this chapter. These books also provide helpful advice on variable naming, coding styles, and code testing. Books on Java (we recommend the one coauthored by James Gosling, the creator of the language [1]) describe the data formats and arithmetic operations supported by Java.

Most books on logic design [88, 39] have a section on encodings and arithmetic operations. Such books describe different ways of implementing arithmetic circuits. Overton's book on IEEE floating point [57] provides a detailed description of the format as well as the properties from the perspective of a numerical applications programmer.

Homework Problems

Homework Problem 2.38 [Category 1]:

Compile and run the sample code that uses `show_bytes` (file `show-bytes.c`) on different machines to which you have access. Determine the byte orderings used by these machines.

Homework Problem 2.39 [Category 1]:

Try running the code for `show_bytes` for different sample values.

Homework Problem 2.40 [Category 1]:

Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of C objects of types `short int`, `long int`, and `double`, respectively. Try these out on several machines.

Homework Problem 2.41 [Category 2]:

Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word size.

Homework Problem 2.42 [Category 2]:

Write a C expression that will yield a word consisting of the least significant byte of `x`, and the remaining bytes of `y`. For operands `x` = `0x89ABCDEF` and `y` = `0x76543210`, this would give `0x765432EF`.

Homework Problem 2.43 [Category 2]:

Using only bit-level and logical operations, write C expressions that yield 1 for the described condition and 0 otherwise. Your code should work on a machine with any word size. Assume `x` is an integer.

A. Any bit of `x` equals 1.

B. Any bit of `x` equals 0.

C. Any bit in the least significant byte of `x` equals 1.

D. Any bit in the least significant byte of `x` equals 0.

Homework Problem 2.44 [Category 3]:

Write a function `int_shifts_are_arithmetic()` that yields 1 when run on a machine that uses arithmetic right shifts for `int`'s and 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines. Write and test a procedure `unsigned_shifts_are_arithmetic()` that determines the form of shifts used for `unsigned int`'s.

Homework Problem 2.45 [Category 2]:

You are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. Here is a first attempt:

```
/* The following code does not run properly on some machines */
int bad_int_size_is_32()
{
    /* Set most significant bit (msb) of 32-bit machine */
    int set_msb = 1 << 31;
    /* Shift past msb of 32-bit word */
    int beyond_msb = 1 << 32;

    /* set_msb is nonzero when word size >= 32
       beyond_msb is zero when word size <= 32  */
    return set_msb && !beyond_msb;
}
```

When compiled and run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

A. In what way does our code fail to comply with the C standard?

B. Modify the code to run properly on any machine for which `int`'s are at least 32 bits.

C. Modify the code to run properly on any machine for which `int`'s are at least 16 bits.

Homework Problem 2.46 [Category 1]:

You just started working for a company that is implementing a set of procedures to operate on a data structure where four signed bytes are packed into a 32-bit `unsigned`. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two's-complement arithmetic and arithmetic right shifts with the following prototype:

```
/* Declaration of data type where 4 bytes are packed
   into an unsigned */
typedef unsigned packed_t;

/* Extract byte from word.  Return as signed integer */
int xbyte(packed_t word, int bytenum);
```

That is, the function will extract the designated byte and sign extend it to be a 32-bit `int`.

Your predecessor (who was fired for his incompetence) wrote the following code:

```
/* Failed attempt at xbyte */
int xbyte(packed_t word, int bytenum)
{
  return
    (word >> (bytenum << 3)) & 0xFF;
}
```

A. What is wrong with this code?

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

Homework Problem 2.47 [Category 1]:

Fill in the following table showing the effects of complementing and incrementing several five-bit vectors in the style of Figure 2.20. Show both the bit vectors and the numeric values.

$\vec{x}$	$\tilde{}\vec{x}$	$incr(\tilde{}\vec{x})$
[01101]		
[01111]		
[11000]		
[11111]		
[10000]		

Homework Problem 2.48 [Category 2]:

Show that first decrementing and then complementing is equivalent to complementing and then incrementing. That is, for any signed value `x`, the C expressions `-x`, `~x+1`, and `~(x-1)` yield identical results. What mathematical properties of two's-complement addition does your derivation rely on?

Homework Problem 2.49 [Category 3]:

Suppose we want to compute the complete $2w$-bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type `unsigned` is w bits. The low-order w bits of the product can be computed with the expression `x*y`, so we only require a procedure with prototype

```
    unsigned int unsigned_high_prod(unsigned x, unsigned y);
```

that computes the high-order w bits of $x \cdot y$ for unsigned variables.

We have access to a library function with prototype:

```
    int signed_high_prod(int x, int y);
```

that computes the high-order w bits of $x \cdot y$ for the case where x and y are in two's-complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.

[Hint:] Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

Homework Problem 2.50 [Category 2]:

Suppose we are given the task of generating code to multiply integer variable x by various different constant factors K. To be efficient we want to use only the operations +, -, and <<. For the following values of K, write C expressions to perform the multiplication using at most three operations per expression.

A. $K = 5$:

B. $K = 9$:

C. $K = 14$:

D. $K = -56$:

Homework Problem 2.51 [Category 2]:

Write C expressions to generate the bit patterns that follow, where a^k represents k repetitions of symbol a. Assume a w-bit data type. Your code may contain references to parameters j and k, representing the values of j and k, but not a parameter representing w.

A. $1^{w-k}0^k$.

B. $0^{w-k-j}1^k0^j$.

Homework Problem 2.52 [Category 2]:

Suppose we number the bytes in a w-bit word from 0 (least significant) to $w/8 - 1$ (most significant). Write code for the following C function, which will return an unsigned value in which byte i of argument x has been replaced by byte b:

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Here are some examples showing how the function should work:

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

Homework Problem 2.53 [Category 3]:

Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may assume that `int`'s are 32-bits long. The shift amount `k` can range from 0 to 31.

```
unsigned srl(unsigned x, int k)
{
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;

    /* ... */

}

int sra(int x, int k)
{
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;

    /* ... */

}
```

Homework Problem 2.54 [Category 1]:

We are running programs on a machine where values of type `int` are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type `unsigned` are also 32 bits.

We generate arbitrary values `x` and `y`, and convert them to other unsigned as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
/* Convert to unsigned */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

A. `(x<y) == (-x>-y)`.

B. `((x+y)<<4) + y-x == 17*y+15*x`.

C. `~x+~y == ~(x+y)`.

D. `(int) (ux-uy) == -(y-x)`.

E. `((x >> 1) << 1) <= x`.

Homework Problem 2.55 [Category 2]:

Consider numbers having a binary representation consisting of an infinite string of the form $0.y\,y\,y\,y\,y\,y\cdots$, where y is a k-bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101\cdots$ ($y = 01$), while the representation of $\frac{1}{5}$ is $0.001100110011\cdots$ ($y = 0011$).

A. Let $Y = B2U_k(y)$, that is, the number having binary representation y. Give a formula in terms of Y and k for the value represented by the infinite string. [Hint: Consider the effect of shifting the binary point k positions to the right.]

B. What is the numeric value of the string for the following values of y?

   (a) 001

   (b) 1001

   (c) 000111

Homework Problem 2.56 [Category 1]:

Fill in the return value for the following procedure that tests whether its first argument is greater than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is NaN. The two flavors of zero: $+0$ and -0 are considered equal.

```
int float_ge(float x, float y)
{
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;
    /* Give an expression using only ux, uy, sx, and sy */
    return /* ... */ ;
}
```

Homework Problem 2.57 [Category 1]:

Given a floating-point format with a k-bit exponent and an n-bit fraction, write formulas for the exponent E, significand M, the fraction f, and the value V for the quantities that follow. In addition, describe the bit representation.

A. The number 5.0.

B. The largest odd integer that can be represented exactly.

C. The reciprocal of the smallest positive normalized value.

Homework Problem 2.58 [Category 1]:

Intel-compatible processors also support an "extended precision" floating-point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single *integer* bit, and $n = 63$ fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

Description	Extended precision	
	Value	Decimal
Smallest denormalized		
Smallest normalized		
Largest normalized		

Homework Problem 2.59 [Category 1]:

Consider a 16-bit floating-point representation based on the IEEE floating-point format, with one sign bit, seven exponent bits ($k = 7$), and eight fraction bits ($n = 8$). The exponent bias is $2^{7-1} - 1 = 63$.

Fill in the table that follows for each of the numbers given, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include: 0, $\frac{67}{64}$, and $\frac{1}{256}$.

E: The integer value of the exponent.

V: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $\frac{7}{2}$, we would have $s = 0$, $M = \frac{7}{4}$, and $E = 1$. Our number would therefore have an exponent field of `0x40` (decimal value $63 + 1 = 64$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `40C0`.

You need not fill in entries marked "—".

Description	Hex	M	E	V
-0				—
Smallest value > 1				
256				—-
Largest denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

Homework Problem 2.60 [Category 1]:

We are running programs on a machine where values of type `int` have a 32-bit two's-complement representation. Values of type `float` use the 32-bit IEEE format, and values of type `double` use the 64-bit IEEE format.

We generate arbitrary integer values `x`, `y`, and `z`, and convert them to other `double` as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
/* Convert to double */
double   dx = (double) x;
double   dy = (double) y;
double   dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression *always* yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0. Note that you cannot use a IA32 machine running GCC to test your answers, since it would use the 80-bit extended-precision representation for both `float` and `double`.

A. `(double)(float) x == dx`.

B. `dx + dy == (double) (y+x)`.

C. `dx + dy + dz == dz + dy + dx`.

D. `dx * dy * dz == dz * dy * dx`.

E. `dx / dx == dy / dy`.

Homework Problem 2.61 [Category 1]:

You have been assigned the task of writing a C function to compute a floating-point representation of 2^x. You realize that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the code that follows to compute the correct result. Assume the function `u2f` returns a floating-point value having an identical bit representation as its unsigned argument.

```
float fpwr2(int x)


    /* Result exponent and significand */
    unsigned exp, sig;
    unsigned u;

    if (x < ______)
        /* Too small.  Return 0.0 */
        exp = ____________;
        sig = ____________;
     else if (x < ______)
        /* Denormalized result */
        exp = ____________;
        sig = ____________;
     else if (x < ______)
        /* Normalized result. */
        exp = ____________;
        sig = ____________;
     else
        /* Too big.  Return +oo */
        exp = ____________;
        sig = ____________;


    /* Pack exp and sig into 32 bits */
    u = exp << 23 | sig;
    /* Return as float */
    return u2f(u);
```

Homework Problem 2.62 [Category 1]:

Around 250 B.C., the Greek mathematician Archimedes proved that $\frac{223}{71} < \pi < \frac{22}{7}$. Had he had access to a computer and the standard library `<math.h>`, he would have been able to determine that the single-precision floating-point approximation of π has the hexadecimal representation `0x40490FDB`. Of course, all of these are just approximations, since π is not rational.

A. What is the fractional binary number denoted by this floating-point value?

B. What is the fractional binary representation of $\frac{22}{7}$? [Hint: See Problem 2.55].

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

Solutions to Practice Problems

Problem 2.1 Solution: [Pg. 28]

Understanding the relation between hexadecimal and binary formats will be important once we start looking at machine-level programs. The method for doing these conversions is in the text, but it takes a little practice for it to become familiar

A. `0x8F7A93` to binary:

Hexadecimal	8	F	7	A	9	3
Binary	1000	1111	0111	1010	1001	0011

B. Binary 1011011110011100 to hexadecimal:

Binary	1011	0111	1001	1100
Hexadecimal	B	7	9	C

C. `0xC4E5D` to binary:

Hexadecimal	C	4	E	5	D
Binary	1100	0100	1110	0101	1101

D. Binary 1101011011011111100110 to hexadecimal:

Binary	11	0101	0111	1110	0110
Hexadecimal	3	5	7	E	6

Problem 2.2 Solution: [Pg. 29]

This problem gives you a chance to think about powers of two and their hexadecimal representations.

n	2^n (Decimal)	2^n (Hexadecimal)
11	2048	`0x800`
7	128	`0x80`
13	8192	`0x200`
17	131072	`0x2000`
16	65536	`0x1000`
8	256	`0x100`
5	32	`0x20`

Problem 2.3 Solution: [Pg. 29]

This problem gives you a chance to try out conversions between hexadecimal and decimal representations for some smaller numbers. For larger ones, it becomes much more convenient and reliable to use a calculator or conversion program.

Decimal	Binary	Hexadecimal
0	00000000	`00`
$55 = 3 \cdot 16 + 7$	0011 0111	`37`
$136 = 8 \cdot 16 + 8$	1000 1000	`88`
$243 = 15 \cdot 16 + 3$	1111 0011	`F3`
$5 \cdot 16 + 2 = 82$	0101 0010	`52`
$10 \cdot 16 + 12 = 172$	1010 1100	`AC`
$14 \cdot 16 + 7 = 231$	1110 0111	`E7`
$10 \cdot 16 + 7 = 167$	1010 0111	`A7`
$3 \cdot 16 + 14 = 62$	0011 1110	`3E`
$11 \cdot 16 + 12 = 188$	1011 1100	`BC`

Problem 2.4 Solution: [Pg. 31]

When you begin debugging machine-level programs, you will find many cases where some simple hexadecimal arithmetic would be useful. You can always convert numbers to decimal, perform the arithmetic, and convert them back, but being able to work directly in hexadecimal is more efficient and informative.

A. `0x502c` + `0x8` = `0x5034`. Adding `8` to hex `c` gives `4` with a carry of `1`.

B. `0x502c` − `0x30` = `0x4ffc`. Subtracting `3` from `2` in the second digit position requires a borrow from the third. Since this digit is `0`, we must also borrow from the fourth position.

C. `0x502c` + 64 = `0x506c`. Decimal 64 (2^6) equals hexadecimal `0x40`.

D. `0x51da` − `0x502c` = `0xae`. To subtract hex `c` (decimal 12) from hex `a` (decimal 10), we borrow 16 from the second digit, giving hex `e` (decimal 14). In the second digit, we now subtract `2` from hex `c` (decimal 12), giving hex `a` (decimal 10).

Problem 2.5 Solution: [Pg. 38]

This problem tests your understanding of the byte representation of data and the two different byte orderings.

A. Little endian: `78` Big endian: `12`

B. Little endian: `78 56` Big endian: `12 34`

C. Little endian: `78 56 34` Big endian: `12 34 56`

Recall that `show_bytes` enumerates a series of bytes starting from the one with lowest address and working toward the one with highest address. On a little-endian machine it would list the bytes from least significant to most. On a big-endian machine, it would list bytes from the most significant byte to the least.

Problem 2.6 Solution: [Pg. 38]

This problem is another chance to practice hexadecimal to binary conversion. It also gets you thinking about integer and floating-point representations. We will explore these representations in more detail later in this chapter.

A. Using the notation of the example in the text, we write the two strings as follows:

```
       0   0   3   5   4   3   2   1
    00000000001101010100001100100001
               *********************
         4   A   5   5   0   C   8   4
      01001010010101010000110010000100
```

B. With the second word shifted two positions relative to the first, we find a sequence with 21 matching bits.

C. We find all bits of the integer embedded in the floating-point number, except for the most signficant bit having value 1. Such is the case for the example in the text as well. In addition, the floating-point number has some nonzero high-order bits that do not match those of the integer.

Problem 2.7 Solution: [Pg. 39]

It prints `41 42 43 44 45 46`. Recall also that the library routine `strlen` does not count the terminating null character, and so `show_bytes` printed only through the character '`F`.'

Problem 2.8 Solution: [Pg. 43]

This problem is a drill to help you become more familiar with Boolean operations.

Operation	Result
a	[01101001]
b	[01010101]
~a	[10010110]
~b	[10101010]
a & b	[01000001]
$a \mid b$	[01111101]
a ^ b	[00111100]

Problem 2.9 Solution: [Pg. 43]

This problem illustrates how Boolean algebra can be used to describe and reason about real-world systems. We can see that this color algebra is identical to the Boolean algebra over bit vectors of length 3.

A. Colors are complemented by complementing the values of R, G, and B. From this we can see that White is the complement of Black, Yellow is the complement of Blue, Magenta is the complement of Green, and Cyan is the complement of Red.

B. Black is 0, and White is 1.

C. We perform Boolean operations based on a bit-vector representation of the colors. From this we get the following:

Blue (001)	|	Red (100)	=	Magenta (101)
Magenta (101)	&	Cyan (011)	=	Blue (001)
Green (010)	^	White (111)	=	Magenta (101)

Problem 2.10 Solution: [Pg. 44]

This procedure relies on the fact that EXCLUSIVE-OR is commutative and associative, and that a ^ $a = 0$ for any a. We will see in Chapter 5 that the code does not work correctly when the two pointers `x` and `y` are equal, that is, they point to the same location.

Step	`*x`	`*y`
Initially	a	b
Step 1	a ^ b	b
Step 2	a ^ b	$(a$ ^ $b)$ ^ $b = (b$ ^ $b)$ ^ $a = a$
Step 3	$(a$ ^ $b)$ ^ $a = (a$ ^ $a)$ ^ $b = b$	a

Problem 2.11 Solution: [Pg. 45]

Here are the expressions:

A. `x | ~0xFF`

B. `x ^ 0xFF`

C. `x & ~0xFF`

These expressions are typical of the kind commonly found in performing low-level bit operations. The expression `~0xFF` creates a mask where the 8 least-significant bits equal 0 and the rest equal 1. Observe that such a mask will be generated regardless of the word size. By contrast, the expression `0xFFFFFF00` would only work on a 32-bit machine.

Problem 2.12 Solution: [Pg. 45]

These problems help you think about the relation between Boolean operations and typical masking operations. Here is the code:

```
/* Bit Set */
int bis(int x, int m)
{
  int result = x | m;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  int result = x & ~m;
  return result;
}
```

It is easy to see that `bis` is equivalent to Boolean OR—a bit is set in `z` if either this bit is set in `x` or it is set in `m`.

The `bic` operation is a bit more subtle. We want to set a bit of `z` to 0 if the corresponding bit of `m` equals 1. If we complement the mask giving `~m`, then we want to set a bit of `z` to 0 if the corresponding bit of the complemented mask equals 0. We can do this with the AND operation.

Problem 2.13 Solution: [Pg. 46]

This problem highlights the relation between bit-level Boolean operations and logic operations in C:

Expression	Value	Expression	Value
`x & y`	`0x02`	`x && y`	`0x01`
`x | y`	`0xF7`	`x || y`	`0x01`
`~x | ~y`	`0xFD`	`!x || !y`	`0x00`
`x & !y`	`0x00`	`x && ~y`	`0x01`

Problem 2.14 Solution: [Pg. 47]

The expression is `!(x ^ y)`.

That is `x^y` will be zero if and only if every bit of `x` matches the corresponding bit of `y`. We then exploit the ability of `!` to determine whether a word contains any nonzero bit.

There is no real reason to use this expression rather than simply writing `x == y`, but it demonstrates some of the nuances of bit-level and logical operations.

Problem 2.15 Solution: [Pg. 47]

This problem is a drill to help you understand the different shift operations.

`x`		`x << 3`		`x >> 2` (Logical)		`x >> 2` (Arithmetic)	
Hex	Binary	Binary	Hex	Binary	Hex	Binary	Hex
`0xF0`	[11110000]	[10000000]	`0x80`	[00111100]	`0x3C`	[11111100]	`0xFC`
`0x0F`	[00001111]	[01111000]	`0x78`	[00000011]	`0x03`	[00000011]	`0x03`
`0xCC`	[11001100]	[01100000]	`0x60`	[00110011]	`0x33`	[11110011]	`0xF3`
`0x55`	[01010101]	[10101000]	`0xA8`	[00010101]	`0x15`	[00010101]	`0x15`

Problem 2.16 Solution: [Pg. 49]

In general, working through examples for very small word sizes is a very good way to understand computer arithmetic.

The unsigned values correspond to those in Figure 2.1. For the two's-complement values, hex digits `0` through `7` have a most significant bit of 0, yielding nonnegative values, while while hex digits `8` through `F`, have a most significant bit of 1, yielding a negative value.

$\vec{x}$		$B2U_4(\vec{x})$	$B2T_4(\vec{x})$
Hexadecimal	Binary		
`A`	[1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
`0`	[0000]	0	0
`3`	[0011]	$2^1 + 2^0 = 3$	$2^1 + 2^0 = 3$
`8`	[1000]	$2^3 = 8$	$-2^3 = -8$
`C`	[1100]	$2^3 + 2^2 = 12$	$-2^3 + 2^2 = -4$
`F`	[1111]	$2^3 + 2^2 + 2^1 + 2^0 = 15$	$-2^3 + 2^2 + 2^1 + 2^0 = -1$

Problem 2.17 Solution: [Pg. 51]

For a 32-bit machine, any value consisting of eight hexadecimal digits beginning with one of the digits `8` through `f` represents a negative number. It is quite common to see numbers beginning with a string of `f`'s, since the leading bits of a negative number are all 1s. You must look carefully, though. For example, the number `0x80483b7` has only seven digits. Filling this out with a leading zero gives `0x080483b7`, a positive number.

```
80483b7:  81 ec 84 01 00 00       sub    $0x184,%esp                A.  388
80483bd:  53                      push   %ebx
80483be:  8b 55 08                mov    0x8(%ebp),%edx             B.    8
80483c1:  8b 5d 0c                mov    0xc(%ebp),%ebx             C.   12
80483c4:  8b 4d 10                mov    0x10(%ebp),%ecx            D.   16
80483c7:  8b 85 94 fe ff ff       mov    0xfffffe94(%ebp),%eax      E. -364
80483cd:  01 cb                   add    %ecx,%ebx
80483cf:  03 42 10                add    0x10(%edx),%eax            F.   16
80483d2:  89 85 a0 fe ff ff       mov    %eax,0xfffffea0(%ebp)      G. -352
80483d8:  8b 85 10 ff ff ff       mov    0xffffff10(%ebp),%eax      H. -240
80483de:  89 42 1c                mov    %eax,0x1c(%edx)            I.   28
80483e1:  89 9d 7c ff ff ff       mov    %ebx,0xffffff7c(%ebp)      J. -132
80483e7:  8b 42 18                mov    0x18(%edx),%eax            K.   24
```

Problem 2.18 Solution: [Pg. 53]

The functions $T2U$ and $U2T$ are very peculiar from a mathematical perspective. It is important to understand how they behave.

We solve this problem by reordering the rows in the solution of Practice problem 2.16 according to the two's-complement value and then listing the unsigned value as the result of the function application. We show the hexadecimal values to make this process more concrete.

$\vec{x}$ (hex)	x	$T2U_4(x)$
`8`	-8	8
`A`	-6	10
`C`	-4	12
`F`	-1	15
`0`	0	0
`3`	3	3

Problem 2.19 Solution: [Pg. 54]

This exercise tests your understanding of Equation 2.4.

For the first four entries, the values of x are negative and $T2U_4(x) = x + 2^4$. For the remaining two entries, the values of x are nonnegative and $T2U_4(x) = x$.

Problem 2.20 Solution: [Pg. 56]

This problem reinforces your understanding of the relation between two's-complement and unsigned representations, and the effects of the C promotion rules. Recall that $TMin_{32}$ is -2147483648, and when cast to unsigned it becomes 2147483648. In addition, if either operand is unsigned, then the other operand will be cast to unsigned before comparing.

Expression	Type	Evaluation
`-2147483648 == 2147483648U`	unsigned	`1`
`-2147483648 < -21474836487`	signed	`1`
`(unsigned) -2147483648 < -21474836487`	unsigned	`1`
`-2147483648 < 21474836487`	signed	`1`
`(unsigned) -2147483648 < 21474836487`	unsigned	`0`

Problem 2.21 Solution: [Pg. 58]

The expressions in these functions are common program "idioms" for extracting values from a word in which multiple bit fields have been packed. They exploit the zero-filling and sign-extending properties of the different shift operations. Note carefully the ordering of the cast and shift operations. In `fun1`, the shifts are performed on unsigned `word` and hence are logical. In `fun2`, shifts are performed after casting `word` to `int` and hence are arithmetic.

A.

`w`	`fun1(w)`	`fun2(w)`
127	127	127
128	128	-128
255	255	-1
256	0	0

B. Function `fun1` extracts a value from the low-order 8 bits of the argument, giving an integer ranging between 0 and 255. Function `fun2` also extracts a value from the low-order 8 bits of the argument, but it also peforms sign extension. The result will be a number between -128 and 127.

Problem 2.22 Solution: [Pg. 60]

The effect of truncation is fairly intuitive for unsigned numbers, but not for two's-complement numbers. This exercise lets you explore its properties using very small word sizes.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
`0`	`0`	0	0	0	0
`3`	`3`	3	3	3	3
`8`	`0`	8	0	-8	0
`A`	`2`	10	2	-6	2
`F`	`7`	15	7	-1	-1

As Equation 2.7 states, the effect of this truncation on unsigned values is to simply to find their residue, modulo 8. The effect of the truncation on signed values is a bit more complex. According to Equation 2.8, we first compute the modulo 8 residue of the argument. This will give values 0–7 for arguments 0–7, and also for arguments -8–-1. Then we apply function $U2T_3$ to these residues, giving two repetitions of the sequences 0–3 and -4–-1.

Problem 2.23 Solution: [Pg. 60]

This problem was designed to demonstrate how easily bugs can arise due to the implicit casting from signed to unsigned. It seems quite natural to pass parameter `length` as an unsigned, since one would never want to use a negative length. The stopping criterion `i <= length-1` also seems quite natural. But combining these two yields an unexpected outcome!

Since parameter `length` is unsigned, the computation $0-1$ is performed using unsigned arithmetic, which is equivalent to modular addition. The result is then $UMax_{32}$ (assuming a 32-bit machine). The $\leq$ comparison is also performed using an unsigned comparison, and since any 32-bit number is less than or equal to $UMax_{32}$, the comparison always holds! Thus, the code attempts to access invalid elements of array `a`.

The code can be fixed by either declaring `length` to be an `int`, or by changing the test of the `for` loop to be `i < length`.

Problem 2.24 Solution: [Pg. 64]

This problem is a simple demonstration of arithmetic modulo 16. The easiest way to solve it is to convert the hex pattern into its unsigned decimal value. For nonzero values of x, we must have $(-^{\mathrm{u}}_{4}\, x) + x = 16$. Then we convert the complemented value back to hex.

x		$-^{\mathrm{u}}_{4}\, x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
3	3	13	D
8	8	8	8
A	10	6	6
F	15	1	1

Problem 2.25 Solution: [Pg. 66]

This problem is an exercise to make sure you understand two's-complement addition.

x	y	$x + y$	$x +^{\mathrm{t}}_{4}\, y$	Case
-16 $[10000]$	-11 $[10101]$	-27	5 $[00101]$	1
-16 $[10000]$	-16 $[10000]$	-32	0 $[00000]$	1
-8 $[11000]$	7 $[00111]$	-1	-1 $[11111]$	2
-2 $[11110]$	5 $[00101]$	3	3 $[00011]$	3
8 $[01000]$	8 $[01000]$	16	-16 $[10000]$	4

Problem 2.26 Solution: [Pg. 68]

This problem helps you understand two's-complement negation using a very small word size.

For $w = 4$, we have $TMin_4 = -8$. So -8 is its own additive inverse, while other values are negated by integer negation.

x		$-^{\mathrm{t}}_{4}\, x$	
Hex	Decimal	Decimal	Hex
0	0	0	0
3	3	-3	D
8	-8	-8	8
A	-6	6	6
F	-1	1	1

The bit patterns are the same as for unsigned negation.

Problem 2.27 Solution: [Pg. 71]

This problem is an exercise to make sure you understand two's-complement multiplication.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	6	[110]	2	[010]	12	[001100]	4	[100]
Two's Comp.	-2	[110]	2	[010]	-4	[111100]	-4	[100]
Unsigned	1	[001]	7	[111]	7	[000111]	7	[111]
Two's Comp.	1	[001]	-1	[111]	-1	[111111]	7	[111]
Unsigned	7	[111]	7	[111]	49	[110001]	1	[001]
Two's Comp.	-1	[111]	-1	[111]	1	[000001]	1	[001]

Problem 2.28 Solution: [Pg. 72]

In Chapter 3, we will see many examples of the `leal` instruction in action. The instruction is provided to support pointer arithmetic, but the C compiler often uses it as a way to perform multiplication by small constants.

For each value of k, we can compute two multiples: 2^k (when `b` is 0) and $2^k + 1$ (when `b` is `a`. Thus, we can compute multiples 1, 2, 3, 4, 5, 8, and 9.

Problem 2.29 Solution: [Pg. 73]

We have found that people have difficulty with this exercise when working directly with assembly code. It becomes more clear when put in the form shown in `optarith`.

We can see that `M` is 15; `x*M` is computed as `(x<<4)-x`.

We can see that `N` is 4; a bias value of 3 is added when `y` is negative, and the right shift is by 2.

Problem 2.30 Solution: [Pg. 74]

These "C puzzle" problems provide a clear demonstration that programmers must understand the properties of computer arithmetic:

A. `(x >= 0) || ((2*x) < 0)`.
False. Let `x` be -2147483648 (TMin_{32}). We will then have `2*x` equal to 0.

B. `(x & 7) != 7 || (x<<30 < 0)`.
True. If `(x & 7) != 7` evaluates to 0, then we must have bit x_2 equal to 1. When shifted left by 30, this will become the sign bit.

C. `(x * x) >= 0`.
False. When `x` is 65535 (`0xFFFF`), `x*x` is -131071 (`0xFFFE0001`).

D. `x < 0 || -x <= 0`.
True. If `x` is nonnegative, then `-x` is nonpositive.

E. `x > 0 || -x >= 0`.
False. Let `x` be -2147483648 (TMin_{32}). Then both `x` and `-x` are negative.

F. `x*y == ux*uy`.
True. two's-complement and unsigned multiplication have the same bit-level behavior.

G. `~x*y + uy*ux == -y`.
True. `~x` equals `-x-1`. `uy*ux` equals `x*y`. Thus, the left hand side is equivalent to `-x*y-y+x*y`.

Problem 2.31 Solution: [Pg. 76]

Understanding fractional binary representations is an important step to understanding floating-point encodings. This exercise lets you try out some simple examples.

Fractional value	Binary representation	Decimal representation
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$	0.011	0.375
$\frac{23}{16}$	1.0111	1.4375
$\frac{77}{32}$	10.1101	2.40625
$\frac{11}{8}$	1.011	1.375
$\frac{45}{8}$	101.101	5.625
$\frac{49}{16}$	11.0001	3.0625

One simple way to think about fractional binary representations is to represent a number as a fraction of the form $\frac{x}{2^k}$. We can write this in binary using the binary representation of x, with the binary point inserted k positions from the right. As an example, for $\frac{23}{16}$, we have $23_{10} = 10111_2$. We then put the binary point 4 positions from the right to get 1.0111_2.

Problem 2.32 Solution: [Pg. 77]

In most cases, the limited precision of floating-point numbers is not a major problem, because the *relative* error of the computation is still fairly low. In this example, however, the system was sensitive to the *absolute* error.

A. We can see that $x - 0.1$ has binary representation:

$$0.000000000000000000000001100[1100]\cdots_2$$

Comparing this to the binary representation of $\frac{1}{10}$, we can see that it is simply $2^{-20} \times \frac{1}{10}$, which is around 9.54×10^{-8}.

B. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343$.

C. $0.343 \times 2000 \approx 687$.

Problem 2.33 Solution: [Pg. 81]

Working through floating point representations for very small word sizes helps clarify how IEEE floating point works. Note especially the transition between denormalized and normalized values.

Bits	e	E	f	M	V
`0 00 00`	0	0	0	0	0
`0 00 01`	0	0	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
`0 00 10`	0	0	$\frac{2}{4}$	$\frac{2}{4}$	$\frac{2}{4}$
`0 00 11`	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$\frac{3}{4}$
`0 01 00`	1	0	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{4}{4}$
`0 01 01`	1	0	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{5}{4}$
`0 01 10`	1	0	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{6}{4}$
`0 01 11`	1	0	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{7}{4}$
`0 10 00`	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
`0 10 01`	2	1	$\frac{1}{4}$	$\frac{5}{4}$	$\frac{10}{4}$
`0 10 10`	2	1	$\frac{2}{4}$	$\frac{6}{4}$	$\frac{12}{4}$
`0 10 11`	2	1	$\frac{3}{4}$	$\frac{7}{4}$	$\frac{14}{4}$
`0 11 00`	—	—	—	—	$+\infty$
`0 11 01`	—	—	—	—	NaN
`0 11 10`	—	—	—	—	NaN
`0 11 11`	—	—	—	—	NaN

Problem 2.34 Solution: [Pg. 83]

Hexadecimal `0x354321` is equivalent to binary [1101010100001100100001]. Shifting this right 21 places gives $1.101010100001100100001_2 \times 2^{21}$. We form the fraction field by dropping the leading 1 and adding 2 0s, giving [10101010000110010000100]. The exponent is formed by adding bias 127 to 21, giving 148 (binary [10010100]). We combine this with a sign field of 0 to give a binary representation

$$[01001010010101010000110010000100].$$

We see that the correlation between the two representations correspond to the low-order bits of the integer, up to the most significant bit equal to 1 matching the high-order 21 bits of the fraction:

```
   0   0   3   5   4   3   2   1
00000000001101010100001100100001
           *********************

     4   A   5   5   0   C   8   4
  01001010010101010000110010000100
```

Problem 2.35 Solution: [Pg. 83]

This exercise helps you think about what numbers cannot be represented exactly in floating point.

The number has binary representation 1 followed by n 0's followed by 1, giving value $2^{n+1} + 1$.

When $n = 23$, the value is $2^{24} + 1 = 16,777,217$.

Problem 2.36 Solution: [Pg. 87]

In general it is better to use a library macro rather than inventing your own code. This code seems to work on a variety of machines, however.

We assume that the value `1e400` overflows to infinity.

code/data/ieee.c

```
#define POS_INFINITY 1e400
#define NEG_INFINITY (-POS_INFINITY)
#define NEG_ZERO (-1.0/POS_INFINITY)
```

code/data/ieee.c

Problem 2.37 Solution: [Pg. 91]

Exercises such as this one help you develop your ability to reason about floating point operations from a programmer's perspective. Make sure you understand each of the answers.

A. `x == (int)(float) x`
No. For example, when `x` is $TMax$.

B. `x == (int)(double) x`
Yes, since `double` has greater precision and range than `int`.

C. `f == (float)(double) f`
Yes, since `double` has greater precision and range than `float`.

D. `d == (float) d`
No. For example, when `d` is `1e40`, we will get $+\infty$ on the right.

E. `f == -(-f)`
Yes, since a floating-point number is negated by simply inverting its sign bit.

F. `2/3 == 2/3.0`
No, the left-hand value will be the integer value `0`, while the right-hand value will be the floating-point approximation of $\frac{2}{3}$.

G. `(d >= 0.0) || ((d*2) < 0.0)`
Yes, since multiplication is monotonic.

H. `(d+f)-d == f`
No, for example when `d` is $+\infty$ and `f` is 1, the left-hand side will be NaN, while the right-hand side will be 1.

Chapter 3

Machine-Level Representation of Programs

When programming in a high-level language such as C, we are shielded from the detailed, machine-level implementation of our program. In contrast, when writing programs in assembly code, a programmer must specify exactly how the program manages memory and the low-level instructions the program uses to carry out the 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.

Even though optimizing compilers are available, being able to read and understand assembly code is an important skill for serious programmers. By invoking the compiler with appropriate flags, the compiler will generate a file showing its output in assembly code. Assembly code is very close to the actual machine code that computers execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly 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 13, it is important to know what type of storage is used to hold the different program variables. This information is visible at the assembly code level. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by optimizing compilers.

In this chapter, we will learn the details of a 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 such as multiplication by shifts and adds, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can often be a challenge—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 matter where mastering the details is a prerequisite to understanding the deeper and more fundamental concepts. Spending time studying the examples and working through the exercises will be well worthwhile.

We give a brief history of the Intel architecture. Intel processors have grown from rather primitive 16-bit processors in 1978 to the mainstream machines for today's desktop computers. The architecture has grown correspondingly with new features added and the 16-bit architecture transformed to support 32-bit data and addresses. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by GCC and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

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

We then move into material that is marked with an asterisk (*) and is intended for dedicated machine-language enthusiasts. We give a presentation of IA32 support for floating-point code. This is a particularly arcane feature of IA32, and so we advise that only people determined to work with floating-point code attempt to study this section. We give a brief presentation of GCC's support for embedding assembly code within C programs. In some applications, the programmer must drop down to assembly code to access low-level features of the machine. Embedded assembly is the best way to do this.

3.1 A Historical Perspective

The Intel processor line has 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 the successive models of Intel processors, and some of their key features. We use the number of transistors required to implement the processors as an indication of how they have evolved in complexity (K denotes 1000, and M denotes 1,000,000).

8086: (1978, 29 K transistors). One of the first single-chip, 16-bit microprocessors. The 8088, a version 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.

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

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

i486: (1989, 1.9 M transistors). Improved performance and integrated the floating-point unit onto the processor chip but did not change the instruction set.

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

PentiumPro: (1995, 6.5 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). Merged the previously separate PentiumPro and Pentium/MMX lines by implementing the MMX instructions within the P6 microarchitecture.

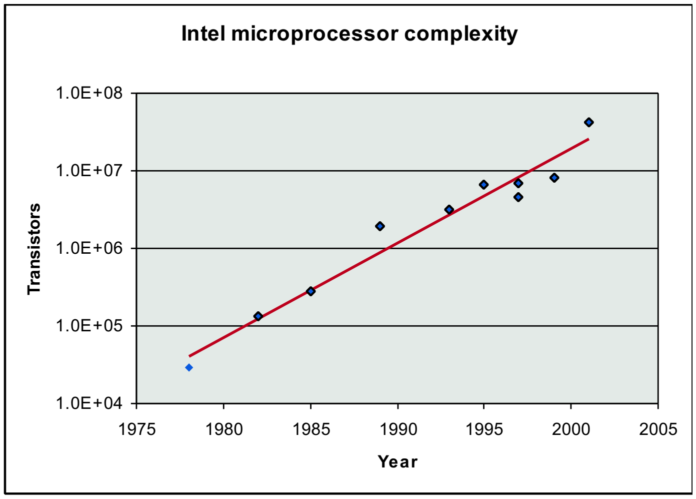
Pentium III: (1999, 8.2 M transistors). Introduced yet another class of instructions for manipulating vectors of integer or floating-point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits. Later versions of this chip went up to 24 M transistors, due to the incorporation of the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

Pentium 4: (2001, 42 M transistors). Added 8-byte integer and floating-point formats to the vector instructions, along with 144 new instructions for these formats. Intel shifted away from Roman numerals in their numbering convention.

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 now calls its instruction set *IA32,* for “Intel Architecture 32-bit.” The processor line is also referred to by the colloquial name “x86,” reflecting the processor naming conventions up through the i486.

Aside: Why not the i586?
Intel discontinued their numeric naming convention, because they were not able to obtain trademark protection for their CPU numbers. The U. S. Trademark office does not allow numbers to be trademarked. Instead, they coined the name "Pentium" using the the Greek root word *penta* as an indication that this was their fifth-generation machine. Since then, they have used variants of this name, even though the PentiumPro is a sixth-generation machine (hence the internal name P6), and the Pentium 4 is a seventh-generation machine. Each new generation involves a major change in the processor design. **End Aside.**

Aside: Moore's Law.



If we plot the number of transistors in the different IA32 processors listed above 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 33%, meaning that the number of transistors doubles about every 30 months. This growth has been sustained over the roughly 25 year history of IA32.

In 1965, Gordon Moore, a founder of Intel Corporation extrapolated from the chip technology of the day, in which they could fabricate circuits with around 64 transistors on a single chip, to predict that the number of transistors per chip would double every year for the next 10 years. This predication became known as *Moore's Law*. As it turns out, his prediction was just a little bit optimistic, but also too short-sighted. Over its 40-year history the semiconductor industry has been able to double transistor counts on average every 18 months.

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

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 AMD. For years, AMD's strategy was to run just behind Intel in technology, producing processors that were less expensive although somewhat lower in performance. More recently, AMD has produced some of the highest performing processors for IA32. They were the first to the break the 1-gigahertz clock speed barrier for a commercially available microprocessor. 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 IA32 is not of concern to those interested in programs for the Linux operating system as generated by the GCC compiler. The memory model provided in the original 8086 and its extensions in the 80286 are obsolete. Instead, Linux uses what is referred to as *flat* addressing, where the entire memory space is viewed by the programmer as a large array of bytes.

As we can see in the list of developments, a number of formats and instructions have been added to IA32 for manipulating vectors of small integers and floating-point numbers. These features were added to allow improved performance on multimedia applications, such as image processing, audio and video encoding and decoding, and three-dimensional computer graphics. Unfortunately, current versions of GCC will not generate any code that uses these new features. In fact, in its default invocations GCC assumes it is generating code for an i386. The compiler makes no attempt to exploit the many extensions added to what is now considered a very old architecture.

3.2 Program Encodings

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

```
unix> gcc -O2 -o p p1.c p2.c
```

The command `gcc` indicates the GNU C compiler GCC. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The flag `-O2` instructs the compiler to apply level-two optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code. Level-two optimization is a good compromise between optimized performance and ease of use. All code in this book was compiled with this optimization level.

This command actually invokes a sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros. 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`. Finally, the *linker* merges these two object files along with code implementing standard Unix library functions (e.g., `printf`) and generates the final executable file. Linking is described in more detail in Chapter 7.

3.2.1 Machine-Level Code

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 object 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 assembly programmer's view of the machine differs significantly from that of a C programmer. Parts of the processor state are visible that normally are hidden from the C programmer:

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

Whereas C provides a model in which objects of different data types can be declared and allocated in memory, assembly code views the memory as simply a large, byte-addressable array. Aggregate data types in C such as arrays and structures are represented in assembly 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 object code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user, (for example, by using the `malloc` library procedure).

The program memory is addressed using *virtual* addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the 32-bit addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

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

3.2.2 Code Examples

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

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

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

```
unix> gcc -O2 -S code.c
```

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

GCC generates assembly code in its own format, known as GAS (for “Gnu ASsembler”). We will base our presentation on this format, which differs significantly from the format used in Intel documentation and by Microsoft compilers. See the bibiliographic notes for advice on locating documentation of the different assembly code formats.

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

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

Each indented line in the above code corresponds to a single machine instruction. For example, the `pushl` instruction indicates that the contents of register `%ebp` should be pushed onto the program stack. All information about local variable names or data types has been stripped away. We still see a reference to the global variable `accum`, since the compiler has not yet determined where in memory this variable will be stored.

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

```
unix> gcc -O2 -c code.c
```

This will generate an object code file `code.o` that is in binary format and hence cannot be viewed directly. Embedded within the 852 bytes of the file `code.o` is a 19 byte sequence having hexadecimal representation:

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

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

Aside: How do I find the byte representation of a program?
First we used a disassembler (to be described shortly) to determine that the code for `sum` is 19 bytes long. Then we ran the GNU debugging tool GDB on file `code.o` and gave it the command:

```
(gdb)  x/19xb sum
```

telling it to examine (abbreviated 'x') 19 hex-formatted (also abbreviated 'x') bytes (abbreviated 'b'). You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 3.12. **End Aside.**

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

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

The result is (where we have added line numbers on the left and annotations on the right):

```
  Disassembly of function sum in file code.o
00000000 <sum>:
 Offset   Bytes                       Equivalent assembly language
   0:    55                            push   %ebp
   1:    89 e5                         mov    %esp,%ebp
   3:    8b 45 0c                      mov    0xc(%ebp),%eax
   6:    03 45 08                      add    0x8(%ebp),%eax
   9:    01 05 00 00 00 00             add    %eax,0x0
   f:    89 ec                         mov    %ebp,%esp
  11:    5d                            pop    %ebp
  12:    c3                            ret
  13:    90                            nop
```

On the left we see the 19 hexadecimal byte values listed in the byte sequence 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 are worth noting:

- IA32 instructions can range in length from 1 to 15 bytes. The instruction encoding is designed so that commonly used instructions and those with fewer operands require a smaller number of bytes than do less common ones or ones with more operands.

- The instruction format is designed in such a way that from a given starting position, there is a unique decoding of the bytes into machine instructions. For example, only the instruction `pushl %ebp` can start with byte value `55`.

- The disassembler determines the assembly code based purely on the byte sequences in the object 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 GAS. In our example, it has omitted the suffix '`l`' from many of the instructions.

- Compared with the assembly code in `code.s` we also see an additional `nop` instruction at the end. This instruction will never be executed (it comes after the procedure return instruction), nor would it have any effect if it were (hence the name `nop`, short for "no operation" and commonly spoken as "no op"). The compiler inserted this instruction as a way to pad the space used to store the procedure.

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:

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

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

```
unix> gcc -O2 -o prog code.o main.c
```

The file `prog` has grown to 11,667 bytes, since it contains not just the code for our two procedures but also information used to start and terminate the program as well as to interact with the operating system. We can also disassemble the file `prog`:

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

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

```
     Disassembly of function sum in executable file prog
080483b4 <sum>:
 80483b4:  55                      push   %ebp
 80483b5:  89 e5                   mov    %esp,%ebp
 80483b7:  8b 45 0c                mov    0xc(%ebp),%eax
 80483ba:  03 45 08                add    0x8(%ebp),%eax
 80483bd:  01 05 64 94 04 08       add    %eax,0x8049464
 80483c3:  89 ec                   mov    %ebp,%esp
 80483c5:  5d                      pop    %ebp
 80483c6:  c3                      ret
 80483c7:  90                      nop
```

Note that this code is almost identical to that generated by the disassembly of `code.c`. One main 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 finally determined the location for storing global variable `accum`. On line 5 of the disassembly for `code.o` the address of `accum` was still listed as `0`. In the disassembly of `prog`, the address has been set to `0x8049464`. This is shown in the assembly code rendition of the instruction. It can also be seen in the last four bytes of the instruction, listed from least-significant to most as `64 94 04 08`.

3.2.3 A Note on Formatting

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

```
int simple(int *xp, int y)
{
  int t = *xp + y;
  *xp = t;
  return t;
}
```

When GCC is run with the ‘-S’ flag, it generates the following file for `simple.s`.

```
  .file   "simple.c"
  .version        "01.01"
gcc2_compiled.:
.text
  .align 4
.globl simple
  .type    simple,@function
simple:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  movl (%eax),%edx
  addl 12(%ebp),%edx
  movl %edx,(%eax)
  movl %edx,%eax
  movl %ebp,%esp
  popl %ebp
  ret
.Lfe1:
  .size    simple,.Lfe1-simple
  .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

The file contains more information than we really require. 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 includes line numbers and explanatory annotations. For our example, an annotated version would appear as follows:

```
simple:
  pushl %ebp              Save frame pointer
  movl %esp,%ebp          Create new frame pointer
  movl 8(%ebp),%eax       Get xp
  movl (%eax),%edx        Retrieve *xp
  addl 12(%ebp),%edx      Add y to get t
  movl %edx,(%eax)        Store t at *xp
  movl %edx,%eax          Set t as return value
  movl %ebp,%esp          Reset stack pointer
  popl %ebp               Reset frame pointer
  ret                     Return
```

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

Figure 3.1: **Sizes of standard data types**

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.

3.3 Data Formats

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

Figure 3.1 shows the machine representations used for the primitive data types of C. Note that most of the common data types are stored as double words. This includes both regular and long `int`'s, whether or not they are signed. In addition, all pointers (shown here as `char *`) are stored as 4-byte double words. Bytes are commonly used when manipulating string data. Floating-point numbers come in three different forms: single-precision (4-byte) values, corresponding to C data type `float`; double-precision (8-byte) values, corresponding to C data type `double`; and extended-precision (10-byte) values. GCC uses the data type `long double` to refer to extended-precision floating-point values. It also stores them as 12-byte quantities to improve memory system performance, as will be discussed later. Although the ANSI C standard includes `long double` as a data type, they are implemented for most combinations of compiler and machine using the same 8-byte format as ordinary `double`. The support for extended precision is unique to the combination of GCC and IA32.

As the table indicates, every operation in GAS has a single-character suffix denoting the size of the operand. For example, the `mov` (move data) instruction has three variants: `movb` (move byte), `movw` (move word), and `movl` (move double word). The suffix '`l`' is used for double words, since on many machines 32-bit quantities are referred to as "long words," a holdover from an era when 16-bit word sizes were standard. Note that GAS uses the suffix '`l`' to denote both a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating point involves an entirely different set of

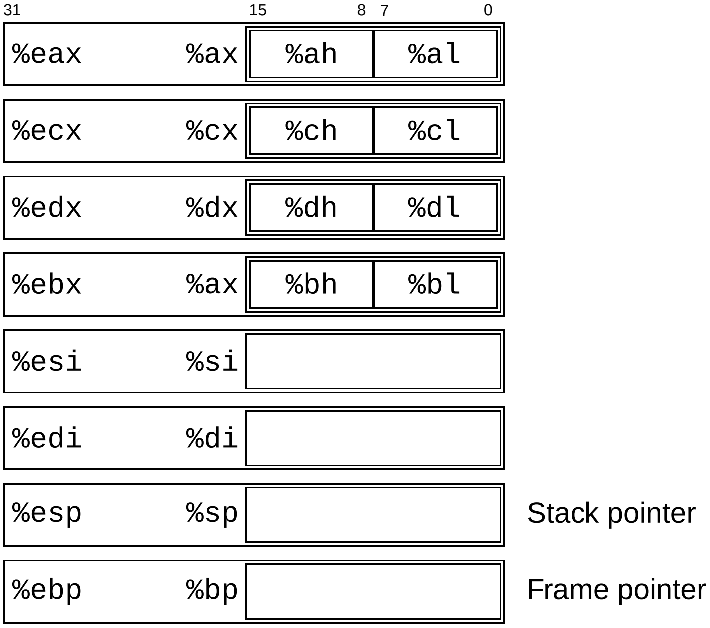


Figure 3.2: **Integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The two low-order bytes of the first four registers can be accessed independently.

instructions and registers.

3.4 Accessing Information

An IA32 central processing unit (CPU) contains a set of eight *registers* storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 3.2 diagrams the eight registers. Their names all begin with `%e`, but otherwise, they have peculiar names. With the original 8086, the registers were 16-bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first six registers can be considered general-purpose registers with no restrictions placed on their use. We said "for the most part," because some instructions use fixed registers as sources and/or destinations. In addition, within procedures there are different conventions for saving and restoring the first three registers (`%eax`, `%ecx`, and `%edx`), than for the next three (`%ebx`, `%edi`, and `%esi`). This will be discussed in Section 3.7. The final two registers (`%ebp` and `%esp`) contain pointers to important places in the program stack. They should only be altered according to the set of standard conventions for stack management.

As indicated in Figure 3.2, the low-order two bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining three bytes of the register do not change. Similarly, the low-order 16 bits of each register can be read or written by word operation instructions. This

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

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

feature stems from IA32's evolutionary heritage as a 16-bit microprocessor.

3.4.1 Operand Specifiers

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. IA32 supports a number of operand forms (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. With GAS, these are written with a '`$`' followed by an integer using standard C notation, such as, `$-577` or `$0x1F`. Any value that fits in a 32-bit word can be used, although the assembler will use one or two-byte encodings when possible. The second type, *register*, denotes the contents of one of the registers, either one of the eight 32-bit registers (e.g., `%eax`) for a double-word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In our figure, we use the notation $\mathtt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $\mathsf{R}[\mathtt{E}_a]$, viewing the set of registers as an array R indexed by register identifiers.

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

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

Instruction		Effect	Description
`movl`	S, D	$D \leftarrow S$	Move double word
`movw`	S, D	$D \leftarrow S$	Move word
`movb`	S, D	$D \leftarrow S$	Move byte
`movsbl`	S, D	$D \leftarrow \mathsf{SignExtend}(S)$	Move sign-extended byte
`movzbl`	S, D	$D \leftarrow \mathsf{ZeroExtend}(S)$	Move zero-extended byte
`pushl`	S	$\mathsf{R}[\texttt{\%esp}] \leftarrow \mathsf{R}[\texttt{\%esp}] - 4;$ $\mathsf{M}[\mathsf{R}[\texttt{\%esp}]] \leftarrow S$	Push
`popl`	D	$D \leftarrow \mathsf{M}[\mathsf{R}[\texttt{\%esp}]];$ $\mathsf{R}[\texttt{\%esp}] \leftarrow \mathsf{R}[\texttt{\%esp}] + 4$	Pop

Figure 3.4: **Data movement instructions.**

Practice Problem 3.1:

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

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

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

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

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

3.4.2 Data Movement Instructions

Among the most heavily used instructions are those that perform data movement. The generality of the operand notation allows a simple move instruction to perform what in many machines would require a number of instructions. Figure 3.4 lists the important data movement instructions. The most common is the `movl` instruction for moving double words. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination.

The following `movl` 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
movl %ebp,%esp            Register--Register
movl (%edi,%ecx),%eax     Memory--Register
movl $-17,(%esp)          Immediate--Memory
movl %eax,-12(%ebp)       Register--Memory
```

The `movb` instruction is similar, except that it moves just a single byte. When one of the operands is a register, it must be one of the eight single-byte register elements illustrated in Figure 3.2. Similarly, the `movw` instruction moves two bytes. When one of its operands is a register, it must be one of the eight 2-byte register elements shown in Figure 3.2.

Both the `movsbl` and the `movzbl` instruction serve to copy a byte and to set the remaining bits in the destination. The `movsbl` instruction takes a single-byte source operand, performs a sign extension to 32 bits (i.e., it sets the high-order 24 bits to the most significant bit of the source byte), and copies this to a double-word destination. Similarly, the `movzbl` instruction takes a single-byte source operand, expands it to 32 bits by adding 24 leading zeros, and copies this to a double-word destination.

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

```
Assume initially that %dh = 8D, %eax = 98765432
movb %dh,%al                %eax = 9876548D
movsbl %dh,%eax             %eax = FFFFFF8D
movzbl %dh,%eax             %eax = 0000008D
```

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

The final two data movement operations are used to push data onto and pop data from the program stack. As we will see, the stack plays a vital role in the handling of procedure calls. Both the `pushl` and the `popl` instructions take a single operand—the data source for pushing and the data destination for popping. The program stack is stored in some region of memory. As illustrated in Figure 3.5, the stack grows downward such that the top element of the stack has the lowest address of all stack elements. (By convention, we draw stacks upside-down, with the stack "top" shown at the bottom of the figure). The stack pointer `%esp` holds the address of the top stack element. Pushing a double-word value onto the stack therefore involves first decrementing the stack pointer by 4 and then writing the value at the new top of stack address. Therefore, the behavior of the instruction `pushl %ebp` is equivalent to that of the following pair of instructions:

```
subl $4,%esp
movl %ebp,(%esp)
```

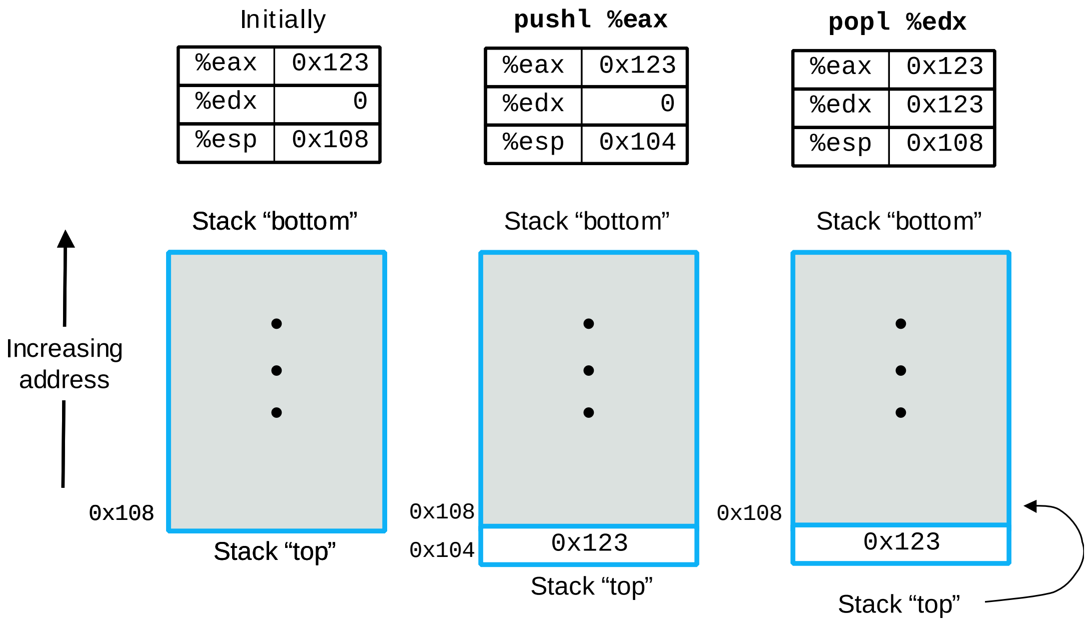


Figure 3.5: **Illustration of stack operation.** By convention, we draw stacks upside-down, so that the "top" of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register `%esp`) and storing to memory, while popping involves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

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

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

```
movl (%esp),%eax
addl $4,%esp
```

The third column of Figure 3.5 illustrates the effect of executing the instruction `popl %edx` immediately after executing the `pushl`. Value `0x123` would be read from memory and written to register `%edx`. Register `%esp` would be incremented back to `0x108`. As shown in the figure, the value `0x123` would remain at memory location `0x104` until it is overwritten by another push operation. However, the stack top is always considered to be the address indicated by `%esp`.

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

code/asm/exchange.c

```
int exchange(int *xp, int y)
{
    int x = *xp;

    *xp = y;
    return x;
}
```

code/asm/exchange.c

(a) C code

```
movl 8(%ebp),%eax      Get xp
movl 12(%ebp),%edx     Get y
movl (%eax),%ecx       Get x at *xp
movl %edx,(%eax)       Store y at *xp
movl %ecx,%eax         Set x as return value
```

(b) Assembly code

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

3.4.3 Data Movement Example

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

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

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

The statement

```
*xp = y;
```

does the reverse—it writes the value of parameter `y` at the location designated by `xp`. This 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 statement.

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

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

This code will print

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

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

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

When the body of the procedure starts execution, procedure parameters `xp` and `y` are stored at offsets 8 and 12 relative to the address in register `%ebp`. Instructions 1 and 2 then move these parameters into registers `%eax` and `%edx`. Instruction 3 dereferences `xp` and stores the value in register `%ecx`, corresponding to program value `x`. Instruction 4 stores `y` at `xp`. Instruction 5 moves `x` to register `%eax`. By convention, any function returning an integer or pointer value does so by placing the result in register `%eax`, and so this instruction implements line 6 of the C code. This example illustrates how the `movl` instruction can be used to read from memory to a register (instructions 1 to 3), to write from a register to memory (instruction 4), and to copy from one register to another (instruction 5).

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 putting that pointer in a register, and then using this register in an indirect 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 3.2:

You are given the following information. A function with prototype

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

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

```
movl 8(%ebp),%edi
movl 12(%ebp),%ebx
movl 16(%ebp),%esi
movl (%edi),%eax
movl (%ebx),%edx
movl (%esi),%ecx
movl %eax,(%ebx)
movl %edx,(%esi)
movl %ecx,(%edi)
```

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

Write C code for `decode1` that will have an effect equivalent to the assembly code above. You can test your answer by compiling your code with the `-S` switch. Your compiler may generate code that differs in the usage of registers or the ordering of memory references, but it should still be functionally equivalent.

Instruction	Effect	Description
`leal` S, D	$D \leftarrow$ &S	Load effective address
`incl` D	$D \leftarrow D$ + 1	Increment
`decl` D	$D \leftarrow D$ - 1	Decrement
`negl` D	$D \leftarrow$ -D	Negate
`notl` D	$D \leftarrow$ ~D	Complement
`addl` S, D	$D \leftarrow D$ + S	Add
`subl` S, D	$D \leftarrow D$ - S	Subtract
`imull` S, D	$D \leftarrow D$ * S	Multiply
`xorl` S, D	$D \leftarrow D$ ^ S	Exclusive-or
`orl` S, D	$D \leftarrow D \mid S$	Or
`andl` S, D	$D \leftarrow D$ & S	And
`sall` k, D	$D \leftarrow D$ << k	Left shift
`shll` k, D	$D \leftarrow D$ << k	Left shift (same as `sall`)
`sarl` k, D	$D \leftarrow D$ >> k	Arithmetic right shift
`shrl` k, D	$D \leftarrow D$ >> k	Logical right shift

Figure 3.7: **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. Note the nonintuitive ordering of the operands with GAS.

3.5 Arithmetic and Logical Operations

Figure 3.7 lists some of the double-word integer operations, divided into four groups. *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. With the exception of `leal`, each of these instructions has a counterpart that operates on words (16 bits) and on bytes. The suffix '`l`' is replaced by '`w`' for word operations and '`b`' for the byte operations. For example, `addl` becomes `addw` or `addb`.

3.5.1 Load Effective Address

The Load Effective Address `leal` instruction is actually a variant of the `movl` instruction. It has the form of an instruction that reads from memory to a register, but it does not reference memory at all. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 3.7 using the C address operator &S. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. The destination operand must be a register.

Practice Problem 3.3:

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

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

3.5.2 Unary and Binary Operations

Operations in the second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or a memory location. For example, the instruction `incl (%esp)` causes the 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 `+=`. Observe, however, that the source operand is given first and the destination second. This looks peculiar for noncommutative operations. For example, the instruction `subl %eax,%edx` decrements register `%edx` by the value in `%eax`. The first operand can be either an immediate value, a register, or a memory location. The second can be either a register or a memory location. As with the `movl` instruction, however, the two operands cannot both be memory locations.

Practice Problem 3.4:

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

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

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

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

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

3.5.3 Shift Operations

The final group consists of shift operations, where the shift amount is given first, and the value to shift is given second. Both arithmetic and logical right shifts are possible. The shift amount is encoded as a single byte, since only shifts amounts between 0 and 31 are allowed. The shift amount is given either as an immediate or in the single-byte register element `%cl`. As Figure 3.7 indicates, there are two names for the left shift instruction: `sall` and `shll`. Both have the same effect, filling from the right with 0s. The right shift instructions differ in that `sarl` performs an arithmetic shift (fill with copies of the sign bit), whereas `shrl` performs a logical shift (fill with 0s).

Practice Problem 3.5:

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

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

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

```
movl 12(%ebp),%ecx     Get n
movl 8(%ebp),%eax      Get x
_____________          x <<= 2
_____________          x >>= n
```

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

3.5.4 Discussion

With the exception of the right shift operations, none of the instructions distinguish between signed and unsigned operands. Two's complement arithmetic has the same bit-level behavior as unsigned arithmetic for all of the instructions listed.

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

Instruction 3 implements the expression `x+y`, getting one operand `y` from register `%eax` (which was fetched by instruction 1) and the other directly from memory. Instructions 4 and 5 perform the computation `z*48`, first using the `leal` instruction with a scaled-indexed addressing mode operand to compute $(z + 2z) = 3z$, *code/asm/arith.c*

```
int arith(int x,
          int y,
          int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    return t4;
}
```

code/asm/arith.c

(a) C code

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

(b) Assembly code

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

and then shifting this value left 4 bits to compute $2^4 \cdot 3z = 48z$. The C compiler often generates combinations of add and shift instructions to perform multiplications by constant factors, as was discussed in Section 2.3.6 (page 71). Instruction 6 performs the AND operation and instruction 7 performs the final multiplication. Then instruction 8 moves the return value into register `%eax`.

In the assembly code of Figure 3.8, the sequence of values in register `%eax` correspond to program values `y`, `t1`, `t3`, and `t4` (as the return value). In general, compilers generate code that uses individual registers for multiple program values and that move program values among the registers.

Practice Problem 3.6:

In the compilation of the loop

```
for (i = 0; i < n; i++)
    v += i;
```

we find the following assembly code line:

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

Explain why this instruction would be there, even though there are no EXCLUSIVE-OR operators in our C code. What operation in the C program does this instruction implement?

3.5.5 Special Arithmetic Operations

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

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

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

The `imull` instruction listed in Figure 3.7 is known as the "two-operand" multiply instruction. It generates a 32-bit product from two 32-bit operands, implementing the operations $*^{\mathrm{u}}_{32}$ and $*^{\mathrm{t}}_{32}$ described in Sections 2.3.4 and 2.3.5. Recall that when truncating the product to 32 bits, both unsigned multiply and two's complement multiply have the same bit-level behavior. IA32 also provides two different "one-operand" multiply instructions to compute the full 64-bit product of two 32-bit values—one for unsigned (`mull`), and one for two's complement (`imull`) multiplication. For both of these, one argument must be in register `%eax`, and the other is given as the instruction source operand. The product is then stored in registers `%edx` (high-order 32 bits) and `%eax` (low-order 32 bits). Note that although the name `imull` is used for two distinct multiplication operations, the assembler can tell which one is intended by counting the number of operands.

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

```
  x at %ebp+8, y at %ebp+12
  movl 8(%ebp),%eax         Put x in %eax
  imull 12(%ebp)            Multiply by y
  pushl %edx                Push high-order 32 bits
  pushl %eax                Push low-order 32 bits
```

Observe that the order in which we push the two registers is correct for a little-endian machine in which the stack grows toward lower addresses, (i.e., the low-order bytes of the product will have lower addresses than the high-order bytes).

Our earlier table of arithmetic operations (Figure 3.7) does not list any division or modulus operations. These operations are provided by the single-operand divide instructions similar to the single-operand multiply instructions. The signed division instruction `idivl` takes as dividend the 64-bit quantity in registers `%edx` (high-order 32 bits) and `%eax` (low-order 32 bits). The divisor is given as the instruction operand. The instructions store the quotient in register `%eax` and the remainder in register `%edx`. The `cltd`[1] instruction can be used to form the 64-bit dividend from a 32-bit value stored in register `%eax`. This instruction sign extends `%eax` into `%edx`.

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

As an example, suppose we have signed numbers x and y stored in positions 8 and 12 relative to %ebp, and we want to store values x/y and x%y on the stack. The code would proceed as follows:

```
 x at %ebp+8, y at %ebp+12
movl 8(%ebp),%eax          Put x in %eax
cltd                       Sign extend into %edx
idivl 12(%ebp)             Divide by y
pushl %eax                 Push x / y
pushl %edx                 Push x % y
```

The divlinstruction performs unsigned division. Typically register %edx is set to 0 beforehand.

3.6 Control

Up to this point, we have considered ways to access and operate on data. Another important part of program execution is to control the sequence of operations that are performed. The default for statements in C as well as for assembly code is to have control flow sequentially, with statements or instructions executed in the order they appear in the program. Some constructs in C, such as conditionals, loops, and switches, allow the control to flow in nonsequential order, with the exact sequence depending on the values of program data.

Assembly code provides lower-level mechanisms for implementing nonsequential control flow. The basic operation is to jump to a different part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon these low-level mechanisms to implement the control constructs of C.

In our presentation, we first cover the machine-level mechanisms and then show how the different control constructs of C are implemented with them.

3.6.1 Condition Codes

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

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

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

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

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

For example, suppose we used the addl instruction to perform the equivalent of the C expression t=a+b, where variables a, b, and t are of type int. Then the condition codes would be set according to the following C expressions:

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

The `leal` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 3.7 cause the condition codes to be set. For the logical operations, such as `xorl`, the carry and overflow flags are set to 0. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to 0.

In addition to the operations of Figure 3.7, the following table shows two operations (having 8, 16, and 32-bit forms) that set conditions codes without altering any other registers:

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

The `cmpb`, `cmpw`, and `cmpl` instructions set the condition codes according to the difference of their two operands. With GAS format, the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands.

The `testb`, `testw`, and `testl` instructions set the zero and negative flags based on the AND of their two operands. Typically, the same operand is repeated (e.g., `testl %eax,%eax` to see whether `%eax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

3.6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, the two most common methods of accessing them are to set an integer register or to perform a conditional branch based on some combination of condition codes. The different `set` instructions described in Figure 3.10 set a single byte to 0 or to 1 depending on some combination of the conditions codes. The destination operand is either one of the eight single-byte register elements (Figure 3.2) or a memory location where the single byte is to be stored. To generate a 32-bit result, we must also clear the high-order 24 bits. A typical instruction sequence for a C predicate (such as `a < b`) is therefore as follows:

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

The `movzbl` instruction is used to clear the high-order three bytes.

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")

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

refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic operations set the condition codes, the descriptions of the different `set` commands apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t = a - b`. For example, 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 a signed comparison with the `setl`, or "Set when less," instruction. When `a` and `b` are in two's complement form, then for `a` $<$ `b` we will have `a` $-$ `b` < 0 if the true difference were computed. When there is no overflow, this would be indicated by having the sign flag set. When there is positive overflow, because `a` $-$ `b` is a large positive number, however, we will have `t` < 0. When there is negative overflow, because `a` $-$ `b` is a small negative number, we will have `t` > 0. In either case, the sign flag will indicate the opposite of the sign of the true difference. Hence, 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, the carry flag will be set by the `cmpl` instruction when the integer difference `a` $-$ `b` of the unsigned arguments `a` and `b` would be negative, that is, when `(unsigned) a < (unsigned) b`. Thus, these tests use combinations of the carry and zero flags.

Practice Problem 3.7:

In the following C code, we have replaced some of the comparison operators with "__" and omitted the data types in the casts.

```
char ctest(int a, int b, int c)
{
  char t1 =         a __            b;
  char t2 =         b __ (     )    a;
  char t3 = (     ) c __ (     )    a;
  char t4 = (     ) a __ (     )    c;
  char t5 =         c __            b;
  char t6 =         a __            0;
  return t1 + t2 + t3 + t4 + t5 + t6;
}
```

For the original C code, GCC generates the following assembly code

```
  movl 8(%ebp),%ecx        Get a
  movl 12(%ebp),%esi       Get b
  cmpl %esi,%ecx           Compare a:b
  setl %al                 Compute t1
  cmpl %ecx,%esi           Compare b:a
  setb -1(%ebp)            Compute t2
  cmpw %cx,16(%ebp)        Compare c:a
  setge -2(%ebp)           Compute t3
  movb %cl,%dl
  cmpb 16(%ebp),%dl        Compare a:c
  setne %bl                Compute t4
  cmpl %esi,16(%ebp)       Compare c:b
  setg -3(%ebp)            Compute t5
  testl %ecx,%ecx          Test a
  setg %dl                 Compute t6
  addb -1(%ebp),%al        Add t2 to t1
  addb -2(%ebp),%al        Add t3 to t1
  addb %bl,%al             Add t4 to t1
  addb -3(%ebp),%al        Add t5 to t1
  addb %dl,%al             Add t6 to t1
  movsbl %al,%eax          Convert sum from char to int
```

Based on this assembly code, fill in the missing parts (the comparisons and the casts) in the C code.

3.6.3 Jump Instructions and their Encodings

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

```
  xorl %eax,%eax           Set %eax to 0
  jmp .L1                  Goto .L1
  movl (%eax),%edx         Null pointer dereference
.L1:
  popl %edx
```

The instruction `jmp .L1` will cause the program to skip over the `movl` instruction and instead resume execution with the `popl` instruction. In generating the object code file, the assembler determines the addresses

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 3.11: **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.

of all labeled instructions and encodes the *jump targets*(the addresses of the destination instructions) as part of the jump instructions.

The jmp instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly by giving a label as the jump target, e.g., the label ".L1" in the code above. Indirect jumps are written using '*' followed by an operand specifier using the same syntax as used for the movl instruction. As examples, the instruction

```
jmp *%eax
```

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

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

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

The other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the condition codes. Note that the names of these instructions and the conditions under which they jump match those of the set instructions. As with the set instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

Although we will not concern ourselves with the detailed format of object code, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter 7. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets.

There are several different encodings for jumps, but some of the most commonly used ones are *PC-relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using one, two, or four bytes. A second encoding method is to give an "absolute" address, using four bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

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

```
 1    jle .L4                     If <, goto dest2
 2    .p2align 4,,7               Aligns next instruction to multiple of 8
 3  .L5:                        dest1:
 4    movl %edx,%eax
 5    sarl $1,%eax
 6    subl %eax,%edx
 7    testl %edx,%edx
 8    jg .L5                      If >, goto dest1
 9  .L4:                        dest2:
10    movl %edx,%eax
```

Note that line 2 is a directive to the assembler that causes the address of the following instruction to begin on a multiple of 16, but leaving a maximum of 7 wasted bytes. This directive is intended to allow the processor to make optimal use of the instruction cache memory.

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

```
1    8:   7e 11                  jle    1b <silly+0x1b>    Target = dest2
2    a:   8d b6 00 00 00 00      lea    0x0(%esi),%esi     Added nops
3   10:   89 d0                  mov    %edx,%eax        dest1:
4   12:   c1 f8 01               sar    $0x1,%eax
5   15:   29 c2                  sub    %eax,%edx
6   17:   85 d2                  test   %edx,%edx
7   19:   7f f5                  jg     10 <silly+0x10>    Target = dest1
8   1b:   89 d0                  mov    %edx,%eax        dest2:
```

The "`lea 0x0(%esi),%esi`" instruction in line 2 has no real effect. It serves as a 6-byte `nop` so that the next instruction (line 3) has a starting address that is a multiple of 16.

In the annotations generated by the disassembler on the right, the jump targets are indicated explicitly as `0x1b` for instruction 1 and `0x10` for instruction 7. Looking at the byte encodings of the instructions, however, we see that the target of jump instruction 1 is encoded (in the second byte) as `0x11` (decimal 17). Adding this to `0xa` (decimal 10), the address of the following instruction, we get jump target address `0x1b` (decimal 27), the address of instruction 8.

Similarly, the target of jump instruction 7 is encoded as `0xf5` (decimal -11) using a single-byte, two's complement representation. Adding this to `0x1b` (decimal 27), the address of instruction 8, we get `0x10` (decimal 16), the address of instruction 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  80483c8:  7e 11                 jle    80483db <silly+0x1b>
2  80483ca:  8d b6 00 00 00 00     lea    0x0(%esi),%esi
3  80483d0:  89 d0                 mov    %edx,%eax
4  80483d2:  c1 f8 01              sar    $0x1,%eax
5  80483d5:  29 c2                 sub    %eax,%edx
6  80483d7:  85 d2                 test   %edx,%edx
7  80483d9:  7f f5                 jg     80483d0 <silly+0x10>
8  80483db:  89 d0                 mov    %edx,%eax
```

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

Practice Problem 3.8:

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 jbe instruction below?

```
8048d1c:  76 da          jbe    XXXXXXX
8048d1e:  eb 24          jmp    8048d44
```

B. What is the address of the mov instruction?

```
XXXXXXX:  eb 54          jmp    8048d44
XXXXXXX:  c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

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

```
8048902:  e9 cb 00 00 00  jmp    XXXXXXX
8048907:  90              nop
```

D. Explain the relation between the annotation on the right and the byte coding on the left. Both lines are part of the encoding of the jmp instruction.

```
80483f0:  ff 25 e0 a2 04  jmp    *0x804a2e0
80483f5:  08
```

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

code/asm/abs.c

```
int absdiff(int x, int y)
{
    if (x < y)
        return y - x;
    else
        return x - y;
}
```

code/asm/abs.c

(a) Original C code.

code/asm/abs.c

```
int gotodiff(int x, int y)
{
    int rval;

    if (x < y)
        goto less;
    rval = x - y;
    goto done;
 less:
    rval = y - x;
 done:
    return rval;
}
```

code/asm/abs.c

(b) Equivalent goto version of (a).

```
  movl 8(%ebp),%edx          Get x
  movl 12(%ebp),%eax         Get y
  cmpl %eax,%edx             Compare x:y
  jl .L3                     If <, goto less
  subl %eax,%edx             Compute x-y
  movl %edx,%eax             Set as return value
  jmp .L5                    Goto done
.L3:                       less:
  subl %edx,%eax             Compute y-x as return value
.L5:                       done: Begin completion code
```

(c) Generated assembly code.

Figure 3.12: **Compilation of conditional statements.** C procedure `absdiff` (a) contains an if-else statement. The generated assembly code is shown (c), along with a C procedure `gotodiff` (b) that mimics the control flow of the assembly code. The stack set-up and completion portions of the assembly code have been omitted

3.6.4 Translating Conditional Branches

Conditional statements in `C` are implemented using combinations of conditional and unconditional jumps. For example, Figure 3.12 shows the C code for a function that computes the absolute value of the difference of two numbers (a). GCC generates the assembly code shown as (c). We have created a version in C, called `gotodiff` (b), that more closely follows the control flow of this assembly code. It uses the `goto` statement in C, which is similar to the unconditional jump of assembly code. The statement `goto less` on line 6 causes a jump to the label `less` on line 9, skipping the statement on line 7. Note that using `goto` statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of assembly-code programs. We call such C programs "goto code."

The assembly code implementation first compares the two operands (line 3), setting the condition codes. If the comparison result indicates that `x` is less than `y`, it then jumps to a block of code that computes `y-x` (line 9). Otherwise it continues with the execution of code that computes `x-y` (lines 5 and 6). In both cases the computed result is stored in register `%eax`, and ends up at line 10, at which point it executes the stack completion code (not shown).

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

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

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

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

```
  t = test-expr;
  if (t)
    goto true;
  else-statement
  goto done;
true:
  then-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 3.9:

When given the C code

code/asm/simple-if.c

```
void cond(int a, int *p)
{
  if (p && a > 0)
    *p += a;
}
```

code/asm/simple-if.c

GCC generates the following assembly code:

```
  movl 8(%ebp),%edx
  movl 12(%ebp),%eax
  testl %eax,%eax
  je .L3
  testl %edx,%edx
  jle .L3
  addl %edx,(%eax)
.L3:
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.12(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.

3.6.5 Loops

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

Do-While Loops

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

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

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

Typically, the implementation of `do-while` has the following general form:

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

As an example, Figure 3.13 shows an implementation of a routine to compute the nth element in the Fibonacci sequence using a `do-while` loop. This sequence is defined by the following recurrence:

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad n \geq 3 \end{aligned}$$

For example, the first ten elements of the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, and 55. To implement this using a `do-while` loop, we have started the sequence with values $F_0 = 0$ and $F_1 = 1$, rather than with F_1 and F_2.

The assembly code implementing the loop is also shown, along with a table showing the correspondence between registers and program values. In this example, *body-statement* consists of lines 8 through 11, assigning values to `t`, `val`, and `nval`, along with the incrementing of `i`. These are implemented by lines 2 through 5 of the assembly code. The expression `i < n` comprises *test-expr*. This is implemented by line 6 and by the test condition of the jump instruction on line 7. Once the loop exits, `val` is copy to register `%eax` as the return value (line 8).

Creating a table of register usage, such as we have shown in Figure 3.13(b) is a very helpful step in analyzing an assembly language program, especially when loops are present.

Practice Problem 3.10:

For the C code

```
int dw_loop(int x, int y, int n)
{
  do {
    x += n;
    y *= n;
    n--;
  } while ((n > 0) & (y < n)); /* Note use of bitwise ’&’ */
  return x;
}
```

GCC generates the following assembly code:

code/asm/fib.c

```
int fib_dw(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;

    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i < n);

    return val;
}
```

code/asm/fib.c

(a) C code.

Register usage		
Register	Variable	Initially
`%ecx`	`i`	0
`%esi`	`n`	`n`
`%ebx`	`val`	0
`%edx`	`nval`	1
`%eax`	`t`	–

```
.L6:                              loop:
  leal (%edx,%ebx),%eax             Compute t = val + nval
  movl %edx,%ebx                    copy nval to val
  movl %eax,%edx                    Copy t to nval
  incl %ecx                         Increment i
  cmpl %esi,%ecx                    Compare i:n
  jl .L6                            If less, goto loop
  movl %ebx,%eax                    Set val as return value
```

(b) Corresponding assembly language code.

Figure 3.13: **C and assembly code for do-while version of Fibonacci program.** Only the code inside the loop is shown.

```
    Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
  movl 8(%ebp),%esi
  movl 12(%ebp),%ebx
  movl 16(%ebp),%ecx
  .p2align 4,,7       Inserted to optimize cache performance
.L6:
  imull %ecx,%ebx
  addl %ecx,%esi
  decl %ecx
  testl %ecx,%ecx
  setg %al
  cmpl %ecx,%ebx
  setl %dl
  andl %edx,%eax
  testb $1,%al
  jne .L6
```

A. Make a table of register usage, similar to the one shown in Figure 3.13(b).

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

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 3.13(b).

While Loops

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

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

It differs from `do-while` in that *test-expr* is evaluated and the loop is potentially terminated before the first execution of *body-statement*. A direct translation into a form using `goto` would be:

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

This translation requires two control statements within the inner loop—the part of the code that is executed the most. Instead, most C compilers transform the code into a `do-while` loop by using a conditional branch to skip the first execution of the body if needed:

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

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

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

As an example, Figure 3.14 shows an implementation of the Fibonacci sequence function using a `while` loop (a). Observe that this time we have started the recursion with elements F_1 (`val`) and F_2 (`nval`). The adjacent C function `fib_w_goto` (b) shows how this code has been translated into assembly. The assembly code in (c) closely follows the C code shown in `fib_w_goto`. The compiler has performed several interesting optimizations, as can be seen in the goto code (b). First, rather than using variable `i` as a loop variable and comparing it to `n` on each iteration, the compiler has introduced a new loop variable that we call "`nmi`", since relative to the original code, its value equals $n - i$. This allows the compiler to use only three registers for loop variables, compared to four otherwise. Second, it has optimized the initial test condition `(i < n)` into `(val < n)`, since the initial values of both `i` and `val` are 1. By this means, the compiler has totally eliminated variable `i`. Often the compiler can make use of the initial values of the variables to optimize the initial test. This can make deciphering the assembly code tricky. Third, for successive executions of the loop we are assured that $i \leq n$, and so the compiler can assume that `nmi` is nonnegative. As a result, it can test the loop condition as `nmi != 0` rather than `nmi >= 0`. This saves one instruction in the assembly code.

Practice Problem 3.11:

For the C code

```
int loop_while(int a, int b)
{
  int i = 0;
  int result = a;
  while (i < 256) {
```

code/asm/fib.c

```c
int fib_w(int n)
{
    int i = 1;
    int val = 1;
    int nval = 1;

    while (i < n) {
        int t = val+nval;
        val = nval;
        nval = t;
        i++;
    }

    return val;
}
```

code/asm/fib.c

(a) C code.

code/asm/fib.c

```c
int fib_w_goto(int n)
{
    int val = 1;
    int nval = 1;
    int nmi, t;

    if (val >= n)
        goto done;
    nmi = n-1;

 loop:
    t = val+nval;
    val = nval;
    nval = t;
    nmi--;
    if (nmi)
        goto loop;

 done:
    return val;
}
```

code/asm/fib.c

(b) Equivalent goto version of (a).

Register usage		
Register	Variable	Initially
`%edx`	`nmi`	`n-1`
`%ebx`	`val`	`1`
`%ecx`	`nval`	`1`

```
  movl 8(%ebp),%eax         Get n
  movl $1,%ebx              Set val to 1
  movl $1,%ecx              Set nval to 1
  cmpl %eax,%ebx            Compare val:n
  jge .L9                   If >= goto done:
  leal -1(%eax),%edx        nmi = n-1
.L10:                     loop:
  leal (%ecx,%ebx),%eax     Compute t = nval+val
  movl %ecx,%ebx            Set val to nval
  movl %eax,%ecx            Set nval to t
  decl %edx                 Decrement nmi
  jnz .L10                  if != 0, goto loop:
.L9:                      done:
```

(c) Corresponding assembly language code.

Figure 3.14: **C and assembly code for while version of Fibonacci.** The compiler has performed a number of optimizations, including replacing the value denoted by variable `i` with one we call `nmi`.

```
    result += a;
    a -= b;
    i += b;
  }
  return result;
}
```

GCC generates the following assembly code:

```
  Initially a and b are at offsets 8 and 12 from %ebp
  movl 8(%ebp),%eax
  movl 12(%ebp),%ebx
  xorl %ecx,%ecx
  movl %eax,%edx
  .p2align 4,,7
.L5:
  addl %eax,%edx
  subl %ebx,%eax
  addl %ebx,%ecx
  cmpl $255,%ecx
  jle .L5
```

A. Make a table of register usage within the loop body, similar to the one shown in Figure 3.14(c).

B. Identify *test-expr* and *body-statement* in the C code, and the corresponding lines in the assembly code. What optimizations has the C compiler performed on the initial test?

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

D. Write a goto version (in C) of the function that has similar structure to the assembly code, as was done in Figure 3.14(b).

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 that the behavior of such a loop is identical to the following code, which uses a while loop:

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

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

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

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

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

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

As an example, the following code shows an implementation of the Fibonacci function using a `for` loop:

code/asm/fib.c

```
int fib_f(int n)
{
    int i;
    int val = 1;
    int nval = 1;

    for (i = 1; i < n; i++) {
        int t = val+nval;
        val = nval;
        nval = t;
    }

    return val;
}
```

code/asm/fib.c

The transformation of this code into the while loop form gives code identical to that for the function `fib_w` shown in Figure 3.14. In fact, GCC generates identical assembly code for the two functions.

Practice Problem 3.12:

Consider the following assembly code:

```
    Initially x, y, and n are offsets 8, 12, and 16 from %ebp
  movl 8(%ebp),%ebx
  movl 16(%ebp),%edx
  xorl %eax,%eax
  decl %edx
  js .L4
  movl %ebx,%ecx
  imull 12(%ebp),%ecx
  .p2align 4,,7     Inserted to optimize cache performance
.L6:
  addl %ecx,%eax
  subl %ebx,%edx
  jns .L6
.L4:
```

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

```
int loop(int x, int y, int n)
{
  int result = 0;
  int i;
  for (i = ____; i ____ ; i = ___ ) {
    result += ______ ;
  }
  return result;
}
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%eax`. To solve this problem, you may need to do a bit of guessing about register usage and then see whether that guess makes sense.

A. Which registers hold program values `result` and `i`?

B. What is the initial value of `i`?

C. What is the test condition on `i`?

D. How does `i` get updated?

E. The C expression describing how to increment `result` in the loop body does not change value from one iteration of the loop to the next. The compiler detected this and moved its computation to before the loop. What is the expression?

F. Fill in all the missing parts of the C code.

3.6.6 Switch Statements

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

Figure 3.15(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 3.16 shows the assembly code generated when compiling `switch_eg`. The behavior of this code is shown using an extended form of C as the procedure `switch_eg_impl` in Figure 3.15(b). We say "extended" because C does not provide the necessary constructs to support this style of jump table, and hence our code is not legal C. The array `jt` contains 7 entries, each of which is the address of a block of code. We extend C with a data type `code` for this purpose.

Lines 1 to 4 set up the jump table access. To make sure that values of `x` that are either less than 100 or greater than 106 cause the computation specified by the `default` case, the code generates an unsigned value `xi` equal to `x-100`. For values of `x` between 100 and 106, `xi` will have values 0 through 6. All other values will be greater than 6, since negative values of `x-100` will wrap around to be very large unsigned numbers. The code therefore uses the `ja` (unsigned greater) instruction to jump to code for the default case when `xi` is greater than 6. Using `jt` to indicate the jump table, the code then performs a jump to the address at entry `xi` in this table. Note that this form of `goto` is not legal C. Instruction 4 implements the jump to an entry in the jump table. Since it is an indirect jump, the target is read from memory. The effective address of the read is determined by adding the base address specified by label `.L10` to the scaled (by 4 since each jump table entry is 4 bytes) value of variable `xi` (in register `%eax`).

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

```
.section .rodata
```

code/asm/switch.c

```c
int switch_eg(int x)
{
    int result = x;

    switch (x) {

    case 100:
        result *= 13;
        break;

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

    case 103:
        result += 11;
        break;

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

    default:
        result = 0;
    }

    return result;
}
```

code/asm/switch.c

(a) Switch statement.

code/asm/switch.c

```c
/* Next line is not legal C */
code *jt[7] = {
    loc_A, loc_def, loc_B, loc_C,
    loc_D, loc_def, loc_D
};

int switch_eg_impl(int x)
{
    unsigned xi = x - 100;
    int result = x;

    if (xi > 6)
        goto loc_def;

    /* Next goto is not legal C */
    goto jt[xi];

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

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

 loc_C:    /* Case 103 */
    result += 11;
    goto done;

 loc_D:    /* Cases 104, 106 */
    result *= result;
    goto done;

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

 done:
    return result;
}
```

code/asm/switch.c

(b) Translation into extended C.

Figure 3.15: **Switch statement example with translation into extended C.** The translation shows the structure of jump table `jt` and how it is accessed. Such tables and accesses are not actually allowed in C.

```
    Set up the jump table access
    leal -100(%edx),%eax            Compute xi = x-100
    cmpl $6,%eax                    Compare xi:6
    ja .L9                          if >, goto loc_def
    jmp *.L10(,%eax,4)              Goto jt[xi]

    Case 100
.L4:                              loc_A:
    leal (%edx,%edx,2),%eax         Compute 3*x
    leal (%edx,%eax,4),%edx         Compute x+4*3*x
    jmp .L3                         Goto done

    Case 102
.L5:                              loc_B:
    addl $10,%edx                   result += 10, Fall through

    Case 103
.L6:                              loc_C:
    addl $11,%edx                   result += 11
    jmp .L3                         Goto done

    Cases 104, 106
.L8:                              loc_D:
    imull %edx,%edx                 result *= result
    jmp .L3                         Goto done

    Default case
.L9:                              loc_def:
    xorl %edx,%edx                  result = 0

    Return result
.L3:                              done:
    movl %edx,%eax                  Set result as return value
```

Figure 3.16: **Assembly code for switch statement example in Figure 3.15.**

```
  .align 4              Align address to multiple of 4
.L10:
  .long .L4             Case 100: loc_A
  .long .L9             Case 101: loc_def
  .long .L5             Case 102: loc_B
  .long .L6             Case 103: loc_C
  .long .L8             Case 104: loc_D
  .long .L9             Case 105: loc_def
  .long .L8             Case 106: loc_D
```

These declarations state that within the segment of the object code file called "`.rodata`" (for "Read-Only Data"), there should be a sequence of seven "long" (4-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly code labels (e.g., `.L4`). Label `.L10` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (instruction 4).

The code blocks starting with labels `loc_A` through `loc_D` and `loc_def` in `switch_eg_impl` (Figure 3.15(b)) implement the five different branches of the switch statement. Observe that the block of code labeled `loc_def` will be executed either when `x` is outside the range 100 to 106 (by the initial range checking) or when it equals either 101 or 105 (based on the jump table). Note how the code for the block labeled `loc_B` falls through to the block labeled `loc_C`.

Practice Problem 3.13:

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

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

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

```
 Setting up jump table access
  movl 8(%ebp),%eax    Retrieve x
  addl $2,%eax
  cmpl $6,%eax
  ja .L10
  jmp *.L11(,%eax,4)
```

```
 Jump table for switch2
.L11:
  .long .L4
  .long .L10
  .long .L5
  .long .L6
  .long .L8
  .long .L8
  .long .L9
```

Use the foregoing information to answer the following questions:

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

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

3.7 Procedures

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

3.7.1 Stack Frame Structure

IA32 programs make use of the program stack to support procedure calls. The stack is used to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage. The portion of the stack allocated for a single procedure call is called a *stack frame*. Figure 3.17 diagrams the general structure of a stack frame. The topmost stack frame is delimited by two pointers, with register `%ebp` serving as the *frame pointer*, and register `%esp` serving as the *stack pointer*. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

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

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

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

Finally, `Q` will use the stack frame for storing arguments to any procedures it calls.

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

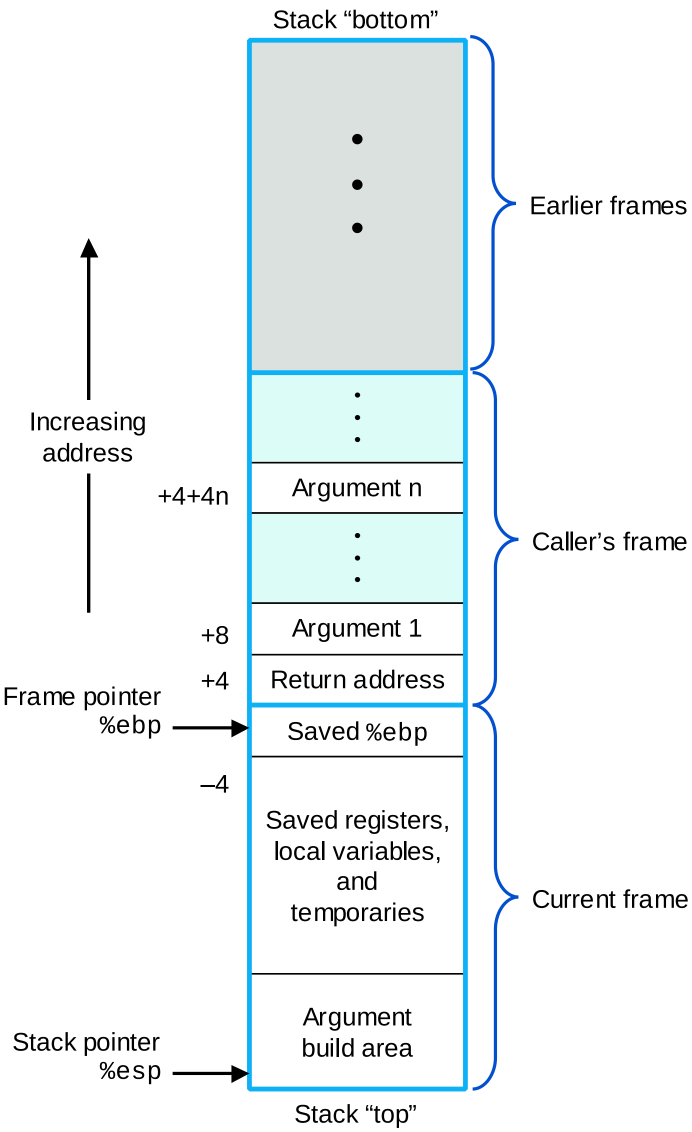


Figure 3.17: **Stack frame structure.** The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

3.7.2 Transferring Control

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

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

The `call` instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can either be direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by a `*` followed by an operand specifier having the same syntax as is used for the operands of the `movl` instruction (Figure 3.3).

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

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

Alternatively, this preparation can be performed by an explicit sequence of move and pop operations.

Register `%eax` is used for returning the value of any function that returns an integer or pointer.

Practice Problem 3.14:

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

```
  call next
next:
  popl %eax
```

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

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

C. What useful purpose does this code fragment serve?

3.7.3 Register Usage Conventions

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

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

Aside: Why the names "callee save" and "caller save?"
Consider the following scenario:

```
int P()
{
    int x = f();  /* Some computation */
    Q();
    return x;
}
```

Procedure `P` wants the value it has computed for `x` to remain valid across the call to `Q`. If `x` is in a *caller save* register, then `P` (the caller) must save the value before calling `P` and restore it after `Q` returns. If `x` is in a *callee save* register, and `Q` (the callee) wants to use this register, then `Q` must save the value before using the register and restore it before returning. In either case, saving involves pushing the register value onto the stack, while restoring involves popping from the stack back to the register. **End Aside.**

As an example, consider the following code:

```
int P(int x)
{
    int y = x*x;
    int z = Q(y);

    return y + z;
}
```

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

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

Most commonly, GCC uses the latter convention, since it tends to reduce the total number of stack writes and reads.

Practice Problem 3.15:

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

```
pushl %edi
pushl %esi
pushl %ebx
movl 24(%ebp),%eax
imull 16(%ebp),%eax
movl 24(%ebp),%ebx
leal 0(,%eax,4),%ecx
addl 8(%ebp),%ecx
movl %ebx,%edx
```

We see that just three registers (`%edi`, `%esi`, and `%ebx`) are saved on the stack. The program then modifies these and three other registers (`%eax`, `%ecx`, and `%edx`). At the end of the procedure, the values of registers `%edi`, `%esi`, and `%ebx` are restored using `popl` instructions, while the other three are left in their modified states.

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

3.7.4 Procedure Example

As an example, consider the C procedures defined in Figure 3.18. Figure 3.19 shows the stack frames for the two procedures. Observe that `swap_add` retrieves its arguments from the stack frame for `caller`. These locations are accessed relative to the frame pointer in register `%ebp`. The numbers along the left of the frames indicate the address offsets relative to the frame pointer.

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

```
  Calling code in caller
leal -4(%ebp),%eax          Compute &arg2
pushl %eax                  Push &arg2
leal -8(%ebp),%eax          Compute &arg1
pushl %eax                  Push &arg1
call swap_add               Call the swap_add function
```

Observe that this code computes the addresses of local variables `arg2` and `arg1` (using the `leal` instruction) and pushes them on the stack. It then calls `swap_add`.

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

code/asm/swapadd.c

```
int swap_add(int *xp, int *yp)
{
    int x = *xp;
    int y = *yp;

    *xp = y;
    *yp = x;
    return x + y;
}

int caller()
{
    int arg1 = 534;
    int arg2 = 1057;
    int sum = swap_add(&arg1, &arg2);
    int diff = arg1 - arg2;

    return sum * diff;
}
```

code/asm/swapadd.c

Figure 3.18: **Example of procedure definition and call.**

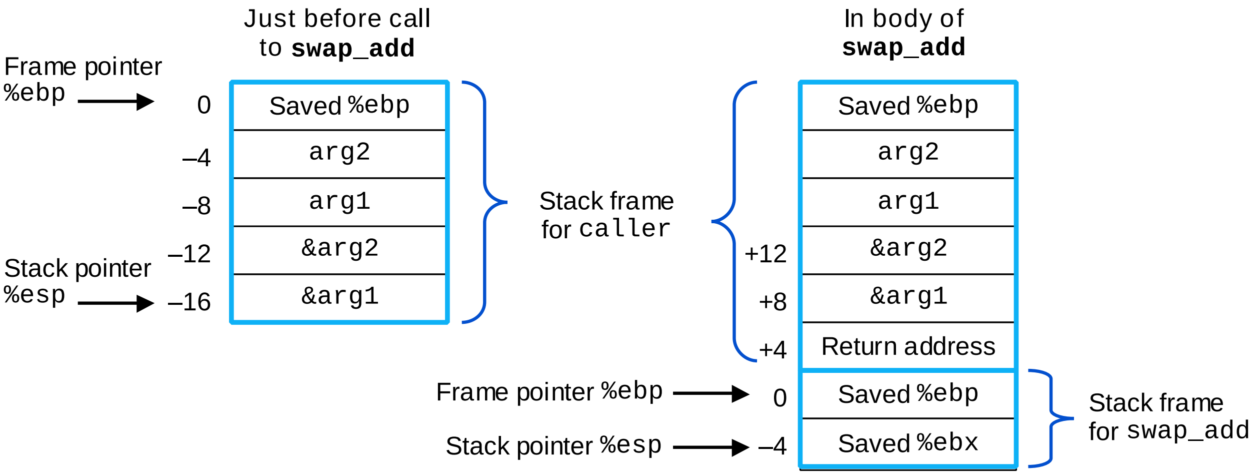


Figure 3.19: **Stack frames for `caller` and swap_add.** Procedure swap_add retrieves its arguments from the stack frame for `caller`.

The following is the setup code for `swap_add`. Recall that the `call` instruction will already push the return address on the stack.

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

Procedure `swap_add` requires register `%ebx` for temporary storage. Since this is a callee save register, it pushes the old value on the stack as part of the stack frame setup.

The following is the body code for `swap_add`:

```
        Body code in swap_add
 5    movl 8(%ebp),%edx           Get xp
 6    movl 12(%ebp),%ecx          Get yp
 7    movl (%edx),%ebx            Get x
 8    movl (%ecx),%eax            Get y
 9    movl %eax,(%edx)            Store y at *xp
10    movl %ebx,(%ecx)            Store x at *yp
11    addl %ebx,%eax              Set return value = x+y
```

This code retrieves its arguments from the stack frame for `caller`. Since the frame pointer has shifted, the locations of these arguments has shifted from positions -12 and -16 relative to the old value of `%ebp` to positions $+12$ and $+8$ relative to new value of `%ebp`. Observe that the sum of variables `x` and `y` is stored in register `%eax` to be passed as the returned value.

The following is the finishing code for `swap_add`:

```
        Finishing code in swap_add
12    popl %ebx                   Restore %ebx
13    movl %ebp,%esp              Restore %esp
14    popl %ebp                   Restore %ebp
15    ret                         Return to caller
```

This code simply restores the values of the three registers `%ebx`, `%esp`, and `%ebp`, and then executes the `ret` instruction. Note that instructions 13 and 14 could be replaced by a single `leave` instruction. Different versions of GCC seem to have different preferences in this regard.

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

```
 6    movl %eax,%edx              Resume here
```

Upon return from `swap_add`, procedure `caller` will resume execution with this instruction. Observe that this instruction copies the return value from `%eax` to a different register.

Practice Problem 3.16:

Given the C function

```
int proc(void)
{
  int x,y;
  scanf("%x %x", &y, &x);
  return x-y;
}
```

GCC generates the following assembly code:

```
 1  proc:
 2    pushl %ebp
 3    movl %esp,%ebp
 4    subl $24,%esp
 5    addl $-4,%esp
 6    leal -4(%ebp),%eax
 7    pushl %eax
 8    leal -8(%ebp),%eax
 9    pushl %eax
10    pushl $.LC0            Pointer to string "%x %x"
11    call scanf
    Diagram stack frame at this point
12    movl -8(%ebp),%eax
13    movl -4(%ebp),%edx
14    subl %eax,%edx
15    movl %edx,%eax
16    movl %ebp,%esp
17    popl %ebp
18    ret
```

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

Register	Value
`%esp`	`0x800040`
`%ebp`	`0x800060`

Suppose `proc` calls `scanf` (line 11), and that `scanf` reads values `0x46` and `0x53` from the standard input. Assume that the string `"%x %x"` is stored at memory location `0x300070`.

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

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

C. What is the value of `%esp` after line 10?

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

E. Indicate the regions of the stack frame that are not used by `proc` (these wasted areas are allocated to improve the cache performance).

code/asm/fib.c

```
int fib_rec(int n)
{
    int prev_val, val;

    if (n <= 2)
        return 1;
    prev_val = fib_rec(n-2);
    val = fib_rec(n-1);
    return prev_val + val;
}
```

code/asm/fib.c

Figure 3.20: **C code for recursive Fibonacci Program.**

3.7.5 Recursive Procedures

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

Figure 3.20 shows the C code for a recursive Fibonacci function. (Note that this code is very inefficient—we intend it to be an illustrative example, not a clever algorithm). The complete assembly code is shown as well in Figure 3.21.

Although there is a lot of code, it is worth studying closely. The set-up code (lines 2 to 6) creates a stack frame containing the old version of `%ebp`, 16 unused bytes,[2] and saved values for the callee save registers `%esi` and `%ebx`, as diagrammed on the left side of Figure 3.22. It then uses register `%ebx` to hold the procedure parameter `n` (line 7). In the event of a terminal condition, the code jumps to line 22, where the return value is set to 1.

For the nonterminal condition, instructions 10 to 12 set up the first recursive call. This involves allocating 12 bytes on the stack that are never used, and then pushing the computed value `n-2`. At this point, the stack frame will have the form shown on the right side of Figure 3.22. It then makes the recursive call, which will trigger a number of calls that allocate stack frames, perform operations on local storage, and so on. As each call returns, it deallocates any stack space and restores any modified callee save registers. Thus, when we return to the current call at line 14 we can assume that register `%eax` contains the value returned by the recursive call, and that register `%ebx` contains the value of function parameter `n`. The returned value (local variable `prev_val` in the C code) is stored in register `%esi` (line 14). By using a callee save register, we can be sure that this value will still be available after the second recursive call.

Instructions 15 to 17 set up the second recursive call. Again it allocates 12 bytes that are never used, and pushes the value of `n-1`. Following this call (line 18), the computed result will be in register `%eax`, and we can assume that the result of the previous call is in register `%esi`. These are added to give the return value

[2] It is unclear why the C compiler allocates so much unused storage on the stack for this function.

```
fib_rec:
  Setup code
  pushl %ebp                Save old %ebp
  movl %esp,%ebp            Set %ebp as frame pointer
  subl $16,%esp             Allocate 16 bytes on stack
  pushl %esi                Save %esi (offset -20)
  pushl %ebx                Save %ebx (offset -24)

  Body code
  movl 8(%ebp),%ebx         Get n
  cmpl $2,%ebx              Compare n:2
  jle .L24                  if <=, goto terminate
  addl $-12,%esp            Allocate 12 bytes on stack
  leal -2(%ebx),%eax        Compute n-2
  pushl %eax                Push as argument
  call fib_rec              Call fib_rec(n-2)
  movl %eax,%esi            Store result in %esi
  addl $-12,%esp            Allocate 12 bytes to stack
  leal -1(%ebx),%eax        Compute n-1
  pushl %eax                Push as argument
  call fib_rec              Call fib_rec(n-1)
  addl %esi,%eax            Compute val+nval
  jmp .L25                  Go to done

  Terminal condition
.L24:                     terminate:
  movl $1,%eax                Return value 1

  Finishing code
.L25:                     done:
  leal -24(%ebp),%esp         Set stack to offset -24
  popl %ebx                   Restore %ebx
  popl %esi                   Restore %esi
  movl %ebp,%esp              Restore stack pointer
  popl %ebp                   Restore %ebp
  ret                         Return
```

Figure 3.21: **Assembly code for the recursive Fibonacci program in Figure 3.20.**

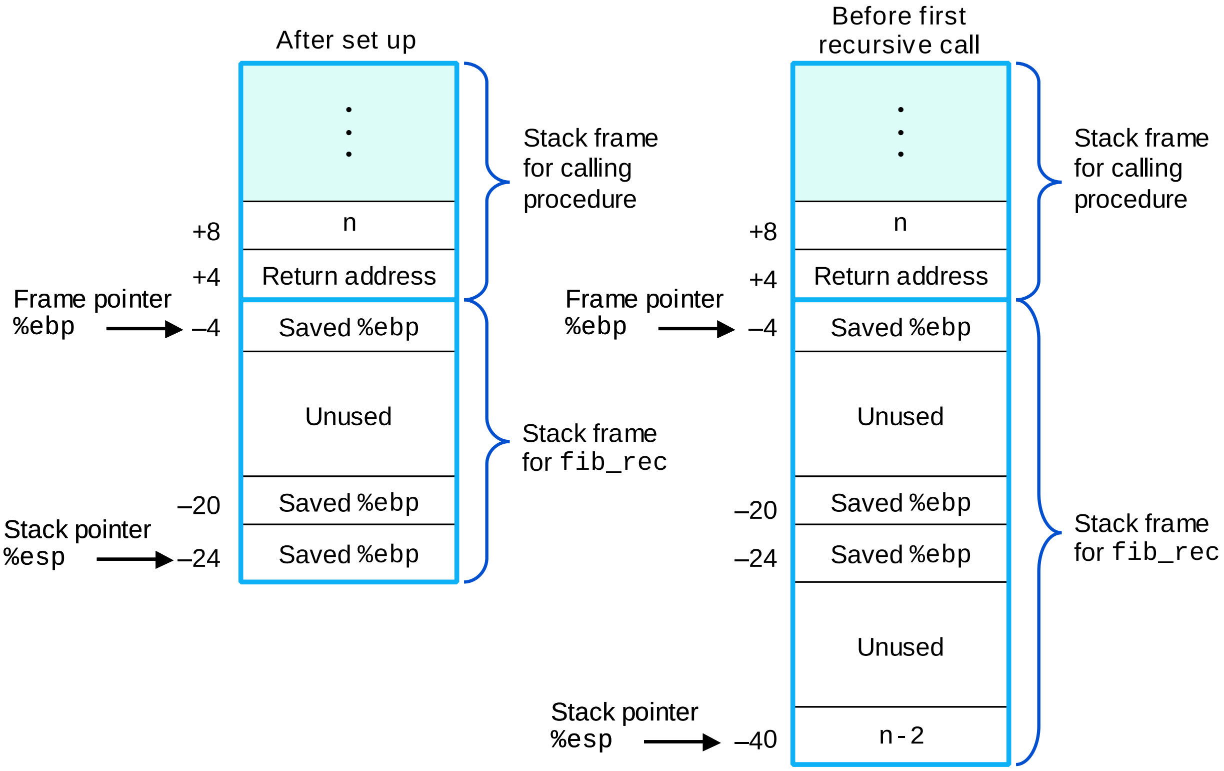


Figure 3.22: **Stack frame for recursive Fibonacci function.** State of frame is shown after initial set up (left), and just before the first recursive call (right).

(instruction 19).

The completion code restores the registers and deallocates the stack frame. It starts (line 24) by setting the stack frame to the location of the saved value of `%ebx`. Observe that by computing this stack position relative to the value of `%ebp`, the computation will be correct regardless of whether or not the terminal condition was reached.

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 one can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in assembly code.

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

3.8.1 Basic Principles

For data type T and integer constant N, the declaration

```
  T A[N];
```

has two effects. First, it allocates a contiguous region of $L \cdot N$ bytes in memory, where L is the size (in bytes) of data type T. Let us denote the starting location as x_{A}. Second, it introduces an identifier `A` that can be used as a pointer to the beginning of the array. The value of this pointer will be x_{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 $x_{\mathsf{A}} + L \cdot i$.

As examples, consider the following declarations:

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

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	x_{A}	$x_{\mathsf{A}} + i$
B	4	32	x_{B}	$x_{\mathsf{B}} + 4i$
C	8	48	x_{C}	$x_{\mathsf{C}} + 8i$
D	4	20	x_{D}	$x_{\mathsf{D}} + 4i$

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

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

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

will perform the address computation $x_{\mathtt{E}} + 4i$, read that memory location, and store the result in register %eax. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the primitive data types.

Practice Problem 3.17:

Consider the following declarations:

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

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

Array	Element size	Total size	Start address	Element i
S			$x_{\mathtt{S}}$	
T			$x_{\mathtt{T}}$	
U			$x_{\mathtt{U}}$	
V			$x_{\mathtt{V}}$	
W			$x_{\mathtt{W}}$	

3.8.2 Pointer Arithmetic

C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer. That is, if p is a pointer to data of type T, and the value of p is $x_{\mathtt{p}}$, 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 *Addr-Expr* denoting an address, **Addr-Expr* 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 %edx and %ecx, respectively. The following are some expressions involving E. We also show an assembly code implementation of each expression, with the result being stored in register %eax.

Expression	Type	Value	Assembly code
`E`	`int *`	x_{E}	`movl %edx,%eax`
`E[0]`	`int`	$\mathsf{M}[x_{\mathsf{E}}]$	`movl (%edx),%eax`
`E[i]`	`int`	$\mathsf{M}[x_{\mathsf{E}} + 4i]$	`movl (%edx,%ecx,4),%eax`
`&E[2]`	`int *`	$x_{\mathsf{E}} + 8$	`leal 8(%edx),%eax`
`E+i-1`	`int *`	$x_{\mathsf{E}} + 4i - 4$	`leal -4(%edx,%ecx,4),%eax`
`*(&E[i]+i)`	`int`	$\mathsf{M}[x_{\mathsf{E}} + 4i + 4i]$	`movl (%edx,%ecx,8),%eax`
`&E[i]-E`	`int`	i	`movl %ecx,%eax`

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

Practice Problem 3.18:

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

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

3.8.3 Arrays and Loops

Array references within loops often have very regular patterns that can be exploited by an optimizing compiler. For example, the function `decimal5` shown in Figure 3.23(a) computes the integer represented by an array of 5 decimal digits. In converting this to assembly code, the compiler generates code similar to that shown in Figure 3.23(b) as C function `decimal5_opt`. First, rather than using a loop index `i`, it uses pointer arithmetic to step through successive array elements. It computes the address of the final array element and uses a comparison to this address as the loop test. Finally, it can use a `do-while` loop since there will be at least one loop iteration.

The assembly code shown in Figure 3.23(c) shows a further optimization to avoid the use of an integer multiply instruction. In particular, it uses `leal` (line 5) to compute `5*val` as `val+4*val`. It then uses `leal` with a scaling factor of 2 (line 7) to scale to `10*val`.

Aside: Why avoid integer multiply?
In older models of the IA32 processor, the integer multiply instruction took as many as 30 clock cycles, and so compilers try to avoid it whenever possible. In the most recent models it requires only 3 clock cycles, and therefore these optimizations are not warranted. **End Aside.**

code/asm/decimal5.c

```
int decimal5(int *x)
{
    int i;
    int val = 0;

    for (i = 0; i < 5; i++)
        val = (10 * val) + x[i];

    return val;
}
```

code/asm/decimal5.c

(a) Original C code

code/asm/decimal5.c

```
int decimal5_opt(int *x)
{
    int val = 0;
    int *xend = x + 4;

    do {
        val = (10 * val) + *x;
        x++;
    } while (x <= xend);

    return val;
}
```

code/asm/decimal5.c

(b) Equivalent pointer code

```
    Body code
  movl 8(%ebp),%ecx              Get base addr of array x
  xorl %eax,%eax                 val = 0;
  leal 16(%ecx),%ebx             xend = x+4 (16 bytes = 4 double words)
.L12:                          loop:
  leal (%eax,%eax,4),%edx        Compute 5*val
  movl (%ecx),%eax               Compute *x
  leal (%eax,%edx,2),%eax        Compute *x + 2*(5*val)
  addl $4,%ecx                   x++
  cmpl %ebx,%ecx                 Compare x:xend
  jbe .L12                       if <=, goto loop:
```

(c) Corresponding assembly code.

Figure 3.23: **C and assembly code for array loop example.** The compiler generates code similar to the pointer code shown in `decimal5_opt`.

3.8.4 Nested Arrays

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

```
int A[4][3];
```

is equivalent to the declaration

```
typedef int row3_t[3];
row3_t A[4];
```

Data type `row3_t` is defined to be an array of three integers. Array `A` contains four such elements, each requiring 12 bytes to store the three integers. The total array size is then $4 \cdot 4 \cdot 3 = 48$ bytes.

Array `A` can also be viewed as a two-dimensional array with four rows and three columns, referenced as `A[0][0]` through `A[3][2]`. The array elements are ordered in memory in “row major” order, meaning all elements of row 0, followed by all elements of row 1, and so on.

Element	Address
`A[0][0]`	$x_{\mathtt{A}}$
`A[0][1]`	$x_{\mathtt{A}} + 4$
`A[0][2]`	$x_{\mathtt{A}} + 8$
`A[1][0]`	$x_{\mathtt{A}} + 12$
`A[1][1]`	$x_{\mathtt{A}} + 16$
`A[1][2]`	$x_{\mathtt{A}} + 20$
`A[2][0]`	$x_{\mathtt{A}} + 24$
`A[2][1]`	$x_{\mathtt{A}} + 28$
`A[2][2]`	$x_{\mathtt{A}} + 32$
`A[3][0]`	$x_{\mathtt{A}} + 36$
`A[3][1]`	$x_{\mathtt{A}} + 40$
`A[3][2]`	$x_{\mathtt{A}} + 44$

This ordering is a consequence of our nested declaration. Viewing `A` as an array of four elements, each of which is an array of three `int`’s, we first have `A[0]` (i.e., row 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 a `movl` instruction using the start of the array as the base address and the (possibly scaled) offset as an index. In general, for an array declared as

$$T\ \mathtt{D}[R][C];$$

array element `D[i][j]` is at memory address $x_{\mathtt{D}} + L(C \cdot i + j)$, where L is the size of data type T in bytes.

As an example, consider the 4×3 integer array `A` defined earlier. Suppose register `%eax` contains $x_{\mathtt{A}}$, `%edx` holds `i`, and `%ecx` holds `j`. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
  A in %eax, i in %edx, j in %ecx
 sall $2,%ecx                        j * 4
 leal (%edx,%edx,2),%edx             i * 3
 leal (%ecx,%edx,4),%edx             j * 4 + i * 12
 movl (%eax,%edx),%eax        Read M[xA + 4(3 · i + j)]
```

Practice Problem 3.19:

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

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
  return mat1[i][j] + mat2[j][i];
}
```

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

```
  movl 8(%ebp),%ecx
  movl 12(%ebp),%eax
  leal 0(,%eax,4),%ebx
  leal 0(,%ecx,8),%edx
  subl %ecx,%edx
  addl %ebx,%eax
  sall $2,%eax
  movl mat2(%eax,%ecx,4),%eax
  addl mat1(%ebx,%edx,4),%eax
```

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

3.8.5 Fixed Size Arrays

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

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

The code in Figure 3.24(a) computes element i, k of the product of matrices A and B. The C compiler generates code similar to that shown in Figure 3.24(b). This code contains a number of clever optimizations. It recognizes that the loop will access the elements of array A as A[i][0], A[i][1], . . . , A[i][15] in sequence. These elements occupy adjacent positions in memory starting with the address of array element A[i][0]. The program can therefore use a pointer variable Aptr to access these successive locations. The loop will access the elements of array B as B[0][k], B[1][k], . . . , B[15][k] in sequence. These elements occupy positions in memory starting with the address of array element `B[0][k]` and spaced 64 bytes apart. The program can therefore use a pointer variable `Bptr` to access these successive locations. In C, this pointer is shown as being incremented by 16, although in fact the actual pointer is incremented by $4 \cdot 16 = 64$. Finally, the code can use a simple counter to keep track of the number of iterations required.

We have shown the C code `fix_prod_ele_opt` to illustrate the optimizations made by the C compiler in generating the assembly. The following is the actual assembly code for the loop:

```
 Aptr is in %edx, Bptr in %ecx, result in %esi, cnt in %ebx
.L23:                          loop:
  movl (%edx),%eax               Compute t = *Aptr
  imull (%ecx),%eax              Compute v = *Bptr * t
  addl %eax,%esi                 Add v result
  addl $64,%ecx                  Add 64 to Bptr
  addl $4,%edx                   Add 4 to Aptr
  decl %ebx                      Decrement cnt
  jns .L23                       if >=, goto loop
```

Note that in the above code, all pointer increments are scaled by a factor of 4 relative to the C code.

Practice Problem 3.20:

The following C code sets the diagonal elements of a fixed-size array to `val`:

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

When compiled, GCC generates the following assembly code:

```
  movl 12(%ebp),%edx
  movl 8(%ebp),%eax
  movl $15,%ecx
  addl $1020,%eax
  .p2align 4,,7            Added to optimize cache performance
.L50:
  movl %edx,(%eax)
  addl $-68,%eax
  decl %ecx
  jns .L50
```

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 3.24(b).

code/asm/array.c

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

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

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

    return result;
}
```

code/asm/array.c

(a) Original C code

code/asm/array.c

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int cnt = N - 1;
    int result = 0;

    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    } while (cnt >= 0);

    return result;
}
```

code/asm/array.c

(b) Optimized C code.

Figure 3.24: **Original and optimized code to compute element i, k of matrix product for fixed length Arrays.** The compiler performs these optimizations automatically.

3.8.6 Dynamically Allocated Arrays

C only supports multidimensional arrays where the sizes (with the possible exception of the first dimension) are known at compile time. In many applications, we require code that will work for arbitrary size arrays that have been dynamically allocated. For these we must explicitly encode the mapping of multidimensional arrays into one-dimensional ones. We can define a data type `var_matrix` as simply an `int *`:

```
typedef int *var_matrix;
```

To allocate and initialize storage for an $n \times n$ array of integers, we use the Unix library function `calloc`:

```
var_matrix new_var_matrix(int n)
{
    return (var_matrix) calloc(sizeof(int), n * n);
}
```

The `calloc` function (documented as part of ANSI C [32, 41]) takes two arguments: the size of each array element and the number of array elements required. It attempts to allocate space for the entire array. If successful, it initializes the entire region of memory to 0s and returns a pointer to the first byte. If sufficient space is not available, it returns null.

New to C?: Dynamic memory allocation and deallocation in C, C++, and Java.
In C, storage on the heap (a pool of memory available for storing data structures) is allocated using the library function `malloc` or its cousin `calloc`. Their effect is similar to that of the `new` operation in C++ and Java. Both C and C++ require the program to explictly free allocated space using the `free` function. In Java, freeing is performed automatically by the run-time system via a process called *garbage collection*, as will be discussed in Chapter 10. **End.**

We can then use the indexing computation of row-major ordering to determine the position of element i, j of the matrix as $i \cdot n + j$:

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

This referencing translates into the following assembly code:

```
  movl 8(%ebp),%edx          Get A
  movl 12(%ebp),%eax         Get i
  imull 20(%ebp),%eax        Compute n*i
  addl 16(%ebp),%eax         Compute n*i + j
  movl (%edx,%eax,4),%eax    Get A[i*n + j]
```

Comparing this code with that used to index into a fixed-size array, we see that the dynamic version is somewhat more complex. It must use a multiply instruction to scale i by n, rather than a series of shifts and adds. In modern processors, this multiplication does not incur a significant performance penalty.

code/asm/array.c

```c
typedef int *var_matrix;

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

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

    return result;
}
```

code/asm/array.c

(a) Original C code

code/asm/array.c

```c
/* Compute i,k of variable matrix product */
int var_prod_ele_opt(var_matrix A, var_matrix B, int i, int k, int n)
{
    int *Aptr = &A[i*n];
    int nTjPk = n;
    int cnt = n;
    int result = 0;

    if (n <= 0)
        return result;

    do {
        result += (*Aptr) * B[nTjPk];
        Aptr += 1;
        nTjPk += n;
        cnt--;
    } while (cnt);

    return result;
}
```

code/asm/array.c

(b) Optimized C code

Figure 3.25: **Original and optimized code to compute element i,k of matrix product for variable length arrays.** The compiler performs these optimizations automatically.

In many cases, the compiler can simplify the indexing computations for variable-sized arrays using the same principles as we saw for fixed-size ones. For example, Figure 3.25(a) shows C code to compute element i, k of the product of two variable-sized matrices `A` and `B`. In Figure 3.25(b) we show an optimized version derived by reverse engineering the assembly code generated by compiling the original version. The compiler is able to eliminate the integer multiplications `i*n` and `j*n` by exploiting the sequential access pattern resulting from the loop structure. In this case, rather than generating a pointer variable `Bptr`, the compiler creates an integer variable we call `nTjPk`, for "`n` Times `j` Plus `k`," since its value equals `n*j+k` relative to the original code. Initially `nTjPk` equals `k`, and it is incremented by `n` on each iteration.

The compiler generates code for the loop, where register `%edx` holds `cnt`, `%ebx` holds `Aptr`, `%ecx` holds `nTjPk`, and `%esi` holds `result`. The code is as follows:

```
.L37:                              loop:
  movl 12(%ebp),%eax                 Get B
  movl (%ebx),%edi                   Get *Aptr
  addl $4,%ebx                       Increment Aptr
  imull (%eax,%ecx,4),%edi           Multiply by B[nTjPk]
  addl %edi,%esi                     Add to result
  addl 24(%ebp),%ecx                 Add n to nTjPk
  decl %edx                          Decrement cnt
  jnz .L37                           If cnt <> 0, goto loop
```

Observe that variables `B` and `n` must be retrieved from memory on each iteration. This is an example of *register spilling*. There are not enough registers to hold all of the needed temporary data, and hence the compiler must keep some local variables in memory. In this case the compiler chose to spill variables `B` and `n` because they are read only—they do not change value within the loop. Spilling is a common problem for IA32, since the processor has so few registers.

3.9 Heterogeneous Data Structures

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

3.9.1 Structures

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

New to C?: Representing an object as a `struct`.
The `struct` data type constructor is the closest thing C provides to the objects of C++ and Java. It allows the

programmer to keep information about some entity in a single data structure, and reference that information with names.

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

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

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

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

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

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

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

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

```
void rotate_left(struct rect *rp)
{
    /* Exchange width and height */
    int t      = rp->height;
    rp->height = rp->width;
    rp->width  = t;
}
```

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

As an example, consider the following structure declaration:

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

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

Offset	0	4	8			20
Contents	`i`	`j`	`a[0]`	`a[1]`	`a[2]`	`p`

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

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

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

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

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

```
 r in %eax, i in %edx
leal 8(%eax,%edx,4),%ecx   %ecx = &r->a[i]
```

As a final example, the following code implements the statement:

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

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

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

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

Practice Problem 3.21:

Consider the following structure declaration:

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

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

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

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

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

p:
s.x:
s.y:
next:

B. How many total bytes does the structure require?

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

```
movl 8(%ebp),%eax
movl 8(%eax),%edx
movl %edx,4(%eax)
leal 4(%eax),%edx
movl %edx,(%eax)
movl %eax,12(%eax)
```

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

3.9.2 Unions

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

Consider the following declarations:

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

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

The offsets of the fields, as well as the total size of data types S3 and U3, are shown in the following table:

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

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

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

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

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

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

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

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

With this encoding, however, there is no way to determine whether a given node is a leaf or an internal node. A common method is to introduce an additional tag field:

```
struct NODE {
    int is_leaf;
    union {
        struct {
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data;
    } info;
};
```

where the field is_leaf is 1 for a leaf node and is 0 for an internal node. This structure requires a total of 12 bytes: 4 for is_leaf, and either 4 each for info.internal.left and info.internal.right, or 8 for info.data. In this case, the savings gain of using a union is small relative to the awkwardness of the resulting code. For data structures with more fields, the savings can be more compelling.

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

```
unsigned float2bit(float f)
{
    union {
        float f;
        unsigned u;
    } temp;
    temp.f = f;
    return temp.u;
};
```

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

```
unsigned copy(unsigned u)
{
    return u;
}
```

The body of both procedures is just a single instruction:

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

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

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

```
double bit2double(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 IA32, argument `word0` will become the low-order four bytes of `d`, while `word1` will become the high-order four bytes. On a big-endian machine, the role of the two arguments will be reversed.

Practice Problem 3.22:

Consider the following union declaration.

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

This declaration illustrates that structures can be embedded within unions.

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

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

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

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

B. How many total bytes would the structure require?

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

```
movl 8(%ebp),%eax
movl 4(%eax),%edx
movl (%edx),%ecx
movl %ebp,%esp
movl (%eax),%eax
movl (%ecx),%ecx
subl %eax,%ecx
movl %ecx,4(%edx)
```

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

3.10 Alignment

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

The IA32 hardware will work correctly regardless of the alignment of data. However, Intel recommends that data be aligned to improve memory system performance. Linux follows an alignment policy where 2-byte data types (e.g., `short`) must have an address that is a multiple of 2, while any larger data types (e.g., `int`, `int *`, `float`, and `double`) must have an address that is a multiple of 4. Note that this requirement means that the least significant bit of the address of an object of type `short` must equal 0. Similarly, any object of type `int`, or any pointer, must be at an address having the low-order two bits equal to 0.

Aside: Alignment with Microsoft Windows.
Microsoft Windows imposes a stronger alignment requirement—any k-byte (primitive) object must have an address that is a multiple of k. In particular, it requires that the address of a `double` be a multiple of 8. This requirement enhances the memory performance at the expense of some wasted space. The design decision made in Linux was probably good for the i386, back when memory was scarce and memory buses were only 4 bytes wide. With modern processors, Microsoft's alignment is a better design decision.

The command line flag `-malign-double` causes GCC on Linux to use 8-byte alignment for data of type `double`. This will lead to improved memory performance, but it can cause incompatibilities when linking with library code that has been compiled assuming a 4-byte alignment. **End Aside.**

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 159 contains the following directive on line 2:

```
.align 4
```

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

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

For example, consider the following structure declaration:

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

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

Offset	0	4	5
Contents	i	c	j

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 "XXX") between fields `c` and `j`:

Offset	0	4	5	8
Contents	i	c	XXX	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 $x_{\mathtt{p}} + 4$) 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 will allocate 12 bytes for structure S1, with the final 3 bytes being wasted space:

Offset	0	4	8	9
Contents	i	j	c	XXX

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 3.23:

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

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

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

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

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

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

3.11 Putting it Together: Understanding Pointers

Pointers are a central feature of the C programming language. They provide a uniform way to provide remote access to data structures. Pointers are a source of confusion for novice programmers, but the underlying concepts are fairly simple. The code in Figure 3.26 lets us illustrate a number of these concepts.

- *Every pointer has a type.* This type indicates what kind of object the pointer points to. In our example code, we see the following pointer types:

Pointer type	**Object type**	**Pointers**
`int *`	`int`	`xp`, `ip[0]`, `ip[1]`
`union uni *`	`union uni`	`up`

  Note in the preceding table, that we indicate the type of the pointer itself, as well as the type of the object it points to. 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 a cast (line 21).

- *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. We will see the values of our pointers shortly.

```
struct str {  /* Example Structure */
    int t;
    char v;
};

union uni {   /* Example Union */
    int t;
    char v;
} u;

int g = 15;

void fun(int* xp)
{
    void (*f)(int*) = fun;   /* f is a function pointer */

    /* Allocate structure on stack */
    struct str s = {1,’a’}; /* Initialize structure */

    /* Allocate union from heap */
    union uni *up = (union uni *) malloc(sizeof(union uni));

    /* Locally declared array */
    int *ip[2] = {xp, &g};

    up->v = s.v+1;

    printf("ip     = %p, *ip    = %p, **ip   = %d\n",
           ip, *ip, **ip);
    printf("ip+1   = %p, ip[1]  =  %p, *ip[1] = %d\n",
           ip+1, ip[1], *ip[1]);
    printf("&s.v   = %p, s.v    = ’%c’\n", &s.v, s.v);
    printf("&up->v =  %p, up->v  = ’%c’\n", &up->v, up->v);
    printf("f      =  %p\n", f);
    if (--(*xp) > 0)
        f(xp);                /* Recursive call of fun */
}

int test()
{
    int x = 2;
    fun(&x);
    return x;
}
```

Figure 3.26: **Code illustrating use of pointers in C.** In C, pointers can be generated to any data type.

- *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. In our example code, we see this operator being applied to global variable `g` (line 24), to structure element `s.v` (line 32), to union element `up->v` (line 33), and to local variable `x` (line 42).

- *Pointers are dereferenced with the* `*` *operator*. The result is a value having the type associated with the pointer. We see dereferencing applied to both `ip` and `*ip` (line 29), to `ip[1]` (line 31), and `xp` (line 35). In addition, the expression `up->v` (line 33) both derefences pointer `up` and selects field `v`.

- *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)`). We can see this in line 29, where we print the pointer value of array `ip`, and reference its first (element 0) entry as `*ip`.

- *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. We see this with variable `f` (line 15), which is declared to be a variable that points to a function taking an `int *` as argument and returning `void`. The assignment makes `f` point to `fun`. When we later apply `f` (line 36), we are making a recursive call.

New to C?: Function pointers.

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

```
void (*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 `void`.

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

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

would be read as

```
(void *) 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 a `void *`.

Kernighan & Ritchie [41, Sect. 5.12] present a helpful tutorial on reading C declarations. **End.**

Our code contains a number of calls to `printf`, printing some of the pointers (using directive `%p`) and values. When executed, it generates the following output:

```
ip     = 0xbfffefa8, *ip    = 0xbfffefe4, **ip   = 2    ip[0] = xp. *xp = x = 2
ip+1   = 0xbfffefac, ip[1]  =  0x804965c, *ip[1] = 15   ip[1] = &g. g = 15
&s.v   = 0xbfffefb4, s.v    = ’a’                 s in stack frame
&up->v =  0x8049760, up->v  = ’b’                 up points to area in heap
f      =  0x8048414                               f points to code for fun
ip     = 0xbfffef68, *ip    = 0xbfffefe4, **ip   = 1    ip in new frame, x = 1
ip+1   = 0xbfffef6c, ip[1]  =  0x804965c, *ip[1] = 15   ip[1] same as before
&s.v   = 0xbfffef74, s.v    = ’a’                 s in new frame
&up->v =  0x8049770, up->v  = ’b’                 up points to new area in heap
f      =  0x8048414                               f points to code for fun
```

We see that the function is executed twice—first by the direct call from `test` (line 42), and second by the indirect, recursive call (line 36). We can see that the printed values of the pointers all correspond to addresses. Those starting with `0xbfffef` point to locations on the stack, while the rest are part of the global storage (`0x804965c`), part of the executable code (`0x8048414`), or locations on the heap (`0x8049760` and `0x8049770`).

Array `ip` is instantiated twice—once for each call to `fun`. The second value (`0xbfffef68`) is smaller than the first (`0xbfffefa8`), because the stack grows downward. The contents of the array, however, are the same in both cases. Element 0 (`*ip`) is a pointer to variable `x` in the stack frame for `test`. Element 1 is a pointer to global variable `g`.

We can see that structure `s` is instantiated twice, both times on the stack, while the union pointed to by variable `up` is allocated on the heap.

Finally, variable `f` is a pointer to function `fun`. In the disassembled code, we find the following as the initial code for `fun`:

```
08048414 <fun>:
 8048414:  55                      push   %ebp
 8048415:  89 e5                   mov    %esp,%ebp
 8048417:  83 ec 1c                sub    $0x1c,%esp
 804841a:  57                      push   %edi
```

The value `0x8048414` printed for pointer `f` is exactly the address of the first instruction in the code for `fun`.

New to C?: Passing parameters to a function.
Other languages, such as Pascal, provide two different ways to pass parameters to procedures—by *value* (identified in Pascal by keyword `var`), where the caller provides the actual parameter value, and by *reference*, where the caller provides a pointer to the value. In C, all parameters are passed by value, but we can simulate the effect of a reference parameter by explicitly generating a pointer to a value and passing this pointer to a procedure. We saw this in function `fun` (Figure 3.26) with the parameter `xp`. With the initial call `fun(&x)` (line 42), the function is given a reference to local variable `x` in `test`. This variable is decremented by each call to `fun` (line 35), causing the recursion to stop after two calls.

C++ reintroduced the concept of a reference parameter, but many feel this was a mistake. **End.**

3.12 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 3.27 shows examples of some GDB commands that help when working with machine-level, IA32 programs. It is very helpful to first run OBJDUMP to get a disassembled version of the program. Our examples are based on running GDB on the file `prog`, described and disassembled on page 123. We start GDB with the following command line:

```
unix> gdb prog
```

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

As our examples suggests, GDB has an obscure command syntax, but the online help information (invoked within GDB with the `help` command) overcomes this shortcoming.

3.13 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 register values and return pointers. 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;
    while ((c = getchar()) != ’\n’ && c != EOF)
        *dest++ = c;
    *dest++ = ’\0’; /* Terminate String */
    if (c == EOF)
        return NULL;
    return s;
}
```

Command	**Effect**
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command line arguments here)
`kill`	Stop your program
Breakpoints	
`break sum`	Set breakpoint at entry to function `sum`
`break *0x80483c3`	Set breakpoint at address `0x80483c3`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas sum`	Disassemble function `sum`
`disas 0x80483b7`	Disassemble function around address `0x80483b7`
`disas 0x80483b7 0x80483c7`	Disassemble code within specified address range
`print /x $eip`	Print program counter in hex
Examining data	
`print $eax`	Print contents of `%eax` in decimal
`print /x $eax`	Print contents of `%eax` in hex
`print /t $eax`	Print contents of `%eax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($ebp+8)`	Print contents of `%ebp` plus 8 in hex
`print *(int *) 0xbffff890`	Print integer at address `0xbffff890`
`print *(int *) ($ebp+8)`	Print integer at address `%ebp` + 8
`x/2w 0xbffff890`	Examine two (4-byte) words starting at address `0xbffff890`
`x/20b sum`	Examine first 20 bytes of function `sum`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.27: **Example GDB commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

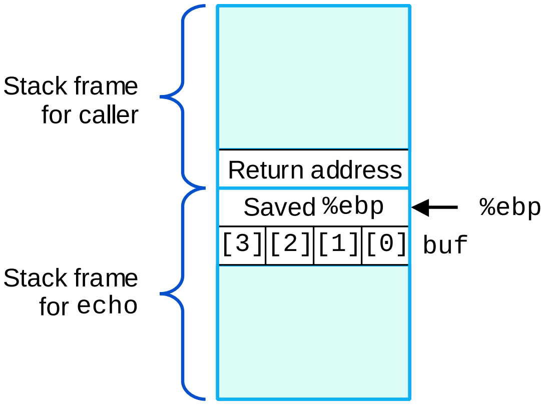


Figure 3.28: **Stack organization for echo function.** Character array `buf` is just below part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

```

/* Read input line and write it back */
void echo()
{
    char buf[4];  /* 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 four characters long. Any string longer than three characters will cause an out-of-bounds write.

Examining a portion of the assembly code for `echo` shows how the stack is organized.

```
1 echo:
2   pushl %ebp                  Save %ebp on stack
3   movl %esp,%ebp
4   subl $20,%esp               Allocate space on stack
5   pushl %ebx                  Save %ebx
6   addl $-12,%esp              Allocate more space on stack
7   leal -4(%ebp),%ebx          Compute buf as %ebp-4
8   pushl %ebx                  Push buf on stack
9   call gets                   Call gets
```

We can see in this example that the program allocates a total of 32 bytes (lines 4 and 6) for local storage. However, the location of character array `buf` is computed as just four bytes below `%ebp` (line 7). Figure

3.28 shows the resulting stack structure. As can be seen, any write to `buf[4]` through `buf[7]` will cause the saved value of `%ebp` to be corrupted. When the program later attempts to restore this as the frame pointer, all subsequent stack references will be invalid. Any write to `buf[8]` through `buf[11]` will cause the return address to be corrupted. When the `ret` instruction is executed at the end of the function, the program will "return" to the wrong address. As this example illustrates, buffer overflow can cause a program to seriously misbehave.

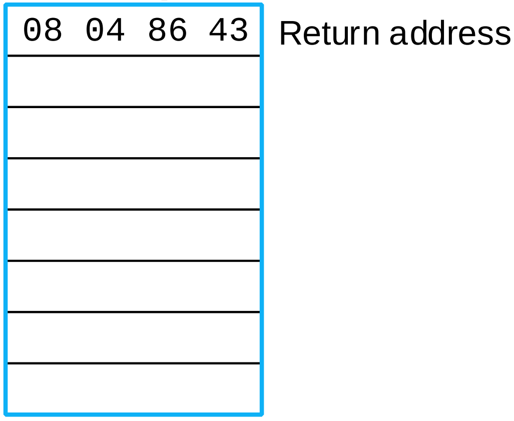
Our code for `echo` is simple but sloppy. A better version involves using the function `fgets`, which includes as an argument a count on the maximum number bytes to read. Homework problem 3.37 asks you to write an echo function that can handle an input string of arbitrary length. In general, using `gets` or any function that can overflow storage is considered a bad programming practice. The C compiler even produces the following error message when compiling a file containing a call to `gets`: "the `gets` function is dangerous and should not be used."

Practice Problem 3.24:

Figure 3.29 shows a (low quality) implementation of a function that reads a line from standard input, copies the string to newly allocated storage, and returns a pointer to the result.

Consider the following scenario. Procedure `getline` is called with the return address equal to `0x8048643`, register `%ebp` equal to `0xbffffc94`, register `%esi` equal to `0x1`, and register `%ebx` equal to `0x2`. You type in the string " `012345678901`." The program terminates with a segmentation fault. You run GDB and determine that the error occurs during the execution of the `ret` instruction of `getline`.

A. Fill in the diagram that follows, indicating as much as you can about the stack just after executing the instruction at line 6 in the disassembly. Label the quantities stored on the stack (e.g., " `Return Address`") on the right, and their hexadecimal values (if known) within the box. Each box represents 4 bytes. Indicate the position of `%ebp`.



B. Modify your diagram to show the effect of the call to `gets` (line 10).

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

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

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

code/asm/bufovf.c

```
/* This is very low quality code.
   It is intended to illustrate bad programming practices.
   See Practice Problem 3.24. */
char *getline()
{
    char buf[8];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return(result);
}
```

code/asm/bufovf.c

C Code

```
08048524 <getline>:
 8048524:  55                 push   %ebp
 8048525:  89 e5              mov    %esp,%ebp
 8048527:  83 ec 10           sub    $0x10,%esp
 804852a:  56                 push   %esi
 804852b:  53                 push   %ebx
  Diagram stack at this point
 804852c:  83 c4 f4           add    $0xfffffff4,%esp
 804852f:  8d 5d f8           lea    0xfffffff8(%ebp),%ebx
 8048532:  53                 push   %ebx
 8048533:  e8 74 fe ff ff     call   80483ac <_init+0x50>      gets
  Modify diagram to show values at this point
```

Disassembly up through call to `gets`

Figure 3.29: **C and disassembled code for Problem 3.24.**

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 pointer with a pointer to the code in the buffer. 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 three years probation, 400 hours of community service, and a $10,500 fine. Even to this day, however, people continue to find security leaks in systems that leave them vulnerable to buffer overflow attacks. This highlights the need for careful programming. Any interface to the external environment should be made "bullet proof" so that no behavior by an external agent can cause the system to misbehave.

Aside: Worms and viruses.
Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [75], 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." **End Aside.**

In Problem 3.38, you can gain first-hand experience at mounting a buffer overflow attack. Note that we do not condone using this or any other method to gain unauthorized access to a system. Breaking into computer systems is like breaking into a building—it is a criminal act even when the perpetrator does not have malicious intent. We give this problem for two reasons. First, it requires a deep understanding of machine-language programming, combining such issues as stack organization, byte ordering, and instruction encoding. Second, by demonstrating how buffer overflow attacks work, we hope you will learn the importance of writing code that does not permit such attacks.

Aside: Battling Microsoft via buffer overflow.
In July, 1999, Microsoft introduced an instant messaging (IM) system whose clients were compatible with the popular America Online (AOL) IM servers. This allowed Microsoft IM users to chat with AOL IM users. However, one month later, Microsoft IM users were suddenly and mysteriously unable to chat with AOL users. Microsoft released updated clients that restored service to the AOL IM system, but within days these clients no longer worked either. Somehow AOL was able to determine whether a user was running the AOL version of the IM client despite Microsoft's repeated attempts to have its client exactly mimic the AOL IM protocol.

The AOL client code was vulnerable to a buffer overflow attack. Most likely this was an inadvertent "feature" in the AOL code. AOL exploited this bug in its own code to detect imposters by attacking the client when the user logged in. The AOL exploit code sampled a small number of locations in the memory image of the client, packed them into a network packet, and sent them back to the server. If the server did not receive such a packet, or if the packet it received did not match the expected "footprint" of the AOL client, then the server assumed the client was not an AOL client and denied it access. So if other IM clients, such as Microsoft's, wanted access to the AOL IM servers, they would not only have to incorporate the buffer overflow bug that existed in AOL's clients, but they would also have to have identical binary code and data in the appropriate memory locations. But as soon as they matched these locations and distributed new versions of their client programs to customers, AOL could simply change its exploit code to sample different locations in the client's memory image. This was clearly a war that the non-AOL clients could never win!

The entire episode had a number of unusual twists and turns. Information about the client bug and AOL's exploitation of it was first divulged when someone posing to be an independent consultant by the name of Phil Bucking sent a description via e-mail to Richard Smith, a noted security expert. Smith did some tracing and determined that the e-mail actually originated from within Microsoft. Later Microsoft admitted that one of its employees had sent the e-mail [52]. On the other side of the controversy, AOL never admitted to the bug nor their exploitation of it, even though conclusive evidence was made public by Geoff Chapell of Australia.

So, who violated which code of conduct in this incident? First, AOL had no obligation to open its IM system to non-AOL clients, so they were justified in blocking Microsoft. On the other hand, using buffer overflows is a tricky business. A small bug would have crashed the client computers, and it made the systems more vulnerable to attacks by external agents (although there is no evidence that this occurred). Microsoft would have done well to publicly announce AOL's intentional use of buffer overflow. However, their Phil Bucking subterfuge was clearly the wrong way to spread this information, from both an ethical and a public relations point of view. **End Aside.**

3.14 *Floating-Point Code

The set of instructions for manipulating floating-point values is one of the least elegant features of the IA32 architecture. In the original Intel machines, floating point was performed by a separate *coprocessor*, a unit with its own registers and processing capabilities that executes a subset of the instructions. This coprocessor was implemented as a separate chip named the 8087, 80287, and i387, to accompany the processor chips 8086, 80286, and i386, respectively. During these product generations, chip capacity was insufficient to include both the main processor and the floating-point coprocessor on a single chip. In addition, lower-budget machines would omit floating-point hardware and simply perform the floating-point operations (very slowly!) in software. Since the i486, floating point has been included as part of the IA32 CPU chip.

The original 8087 coprocessor was introduced to great acclaim in 1980. It was the first single-chip floating-point unit (FPU), and the first implementation of what is now known as IEEE floating point. Operating as a coprocessor, the FPU would take over the execution of floating-point instructions after they were fetched by the main processor. There was minimal connection between the FPU and the main processor. Communicating data from one processor to the other required the sending processor to write to memory and the receiving one to read it. Artifacts of that design remain in the IA32 floating-point instruction set today. In addition, the compiler technology of 1980 was much less sophisticated than it is today. Many features of IA32 floating point make it a difficult target for optimizing compilers.

3.14.1 Floating-Point Registers

The floating-point unit contains eight floating-point registers, but unlike normal registers, these are treated as a shallow stack. The registers are identified as `%st(0)`, `%st(1)`, and so on, up to `%st(7)`, with `%st(0)` being the top of the stack. When more than eight values are pushed onto the stack, the ones at the bottom simply disappear.

Rather than directly indexing the registers, most of the arithmetic instructions pop their source operands from the stack, compute a result, and then push the result onto the stack. Stack architectures were considered a clever idea in the 1970s, since they provide a simple mechanism for evaluating arithmetic instructions, and they allow a very dense coding of the instructions. With advances in compiler technology and with the memory required to encode instructions no longer considered a critical resource, these properties are no longer important. Compiler writers would be much happier with a larger, conventional set of floating-point registers.

> **Aside: Other stack-based languages.**
> Stack-based interpreters are still commonly used as an intermediate representation between a high-level language and its mapping onto an actual machine. Other examples of stack-based evaluators include Java byte code, the intermediate format generated by Java compilers, and the Postscript page formatting language. **End Aside.**

Having the floating-point registers organized as a bounded stack makes it difficult for compilers to use these registers for storing the local variables of a procedure that calls other procedures. For storing local integer variables, we have seen that some of the general purpose registers can be designated as callee saved and hence be used to hold local variables across a procedure call. Such a designation is not possible for an IA32 floating-point register, since its identity changes as values are pushed onto and popped from the stack. For a push operation causes the value in `%st(0)` to now be in `%st(1)`.

On the other hand, it might be tempting to treat the floating-point registers as a true stack, with each procedure call pushing its local values onto it. Unfortunately, this approach would quickly lead to a stack overflow, since there is room for only eight values. Instead, compilers generate code that saves every local floating-point value on the main program stack before calling another procedure and then retrieves them on return. This generates memory traffic that can degrade program performance.

As noted in Section 2.4.6, the IA32 floating-point registers are all 80 bits wide. They encode numbers in an *extended-precision* format as described in Homework Problem 2.58. All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single- or double-precision format as they are stored in memory.

3.14.2 Stack Evaluation of Expressions

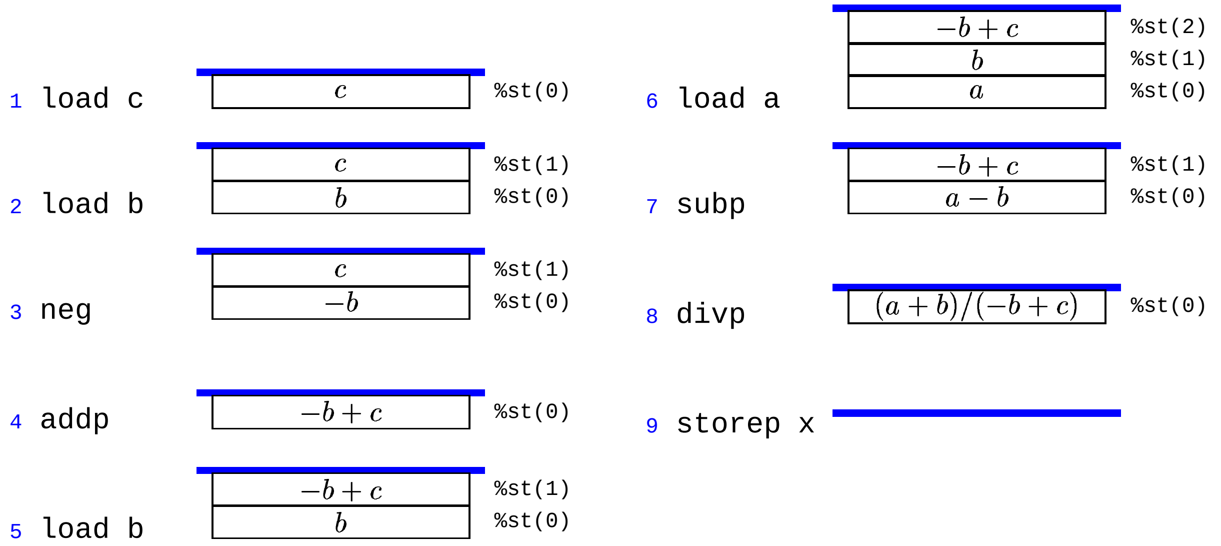
To understand how IA32 uses its floating-point registers as a stack, let us consider a more abstract version of stack-based evaluation. Assume we have an arithmetic unit that uses a stack to hold intermediate results, having the instruction set illustrated in Figure 3.30. For example, so-called RPN (for Reverse Polish Notation) pocket calculators provide this feature. In addition to the stack, this unit has a memory that can hold values we will refer to by names such as `a`, `b`, and `x`. As Figure 3.30 indicates, we can push memory

Instruction	Effect
`load` *S*	Push value at *S* onto stack
`storep` *D*	Pop top stack element and store at *D*
`neg`	Negate top stack element
`addp`	Pop top two stack elements; Push their sum
`subp`	Pop top two stack elements; Push their difference
`multp`	Pop top two stack elements; Push their product
`divp`	Pop top two stack elements; Push their ratio

Figure 3.30: **Hypothetical stack instruction set.** These instructions are used to illustrate stack-based expression evaluation

values onto this stack with the `load` instruction. The `storep` operation pops the top element from the stack and stores the result in memory. A unary operation such as `neg` (negation) uses the top stack element as its argument and overwrites this element with the result. Binary operations such as `addp` and `multp` use the top two elements of the stack as their arguments. They pop both arguments off the stack and then push the result back onto the stack. We use the suffix '`p`' with the store, add, subtract, multiply, and divide instructions to emphasize the fact that these instructions pop their operands.

As an example, suppose we wish to evaluate the expression `x = (a-b)/(-b+c)`. We could translate this expression into the code that follows. Alongside each line of code, we show the contents of the floating-point register stack. In keeping with our earlier convention, we show the stack as growing downward, so the "top" of the stack is really at the bottom.



```
load c
load b
neg
addp
load b
load a
subp
divp
storep x
```

As this example shows, there is a natural recursive procedure for converting an arithmetic expression into stack code. Our expression notation has four types of expressions having the following translation rules:

1. A variable reference of the form *Var*. This is implemented with the instruction `load` *Var*.

2. A unary operation of the form `-` *Expr*. This is implemented by first generating the code for *Expr*

followed by a `neg` instruction.

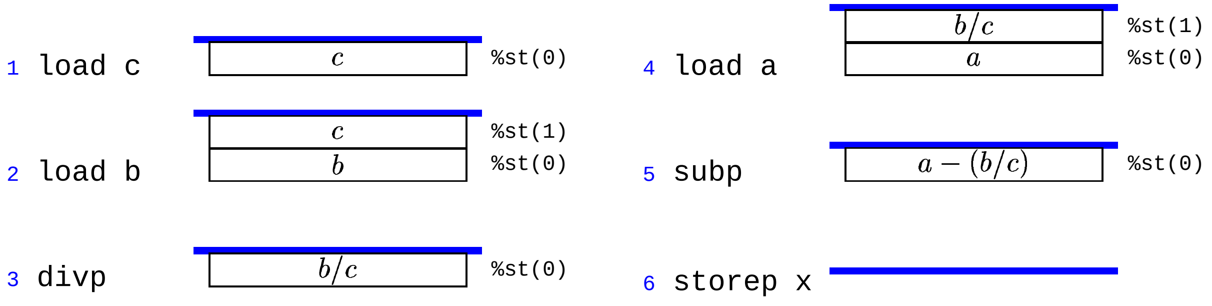
3. A binary operation of the form $Expr_1$ `+` $Expr_2$, $Expr_1$ `-` $Expr_2$, $Expr_1$ `*` $Expr_2$, or $Expr_1$ `/` $Expr_2$. This is implemented by generating the code for $Expr_2$, followed by the code for $Expr_1$, followed by an `addp`, `subp`, `multp`, or `divp` instruction.

4. An assignment of the form Var `=` $Expr$. This is implemented by first generating the code for $Expr$, followed by the `storep` Var instruction.

As an example, consider the expression `x = a-b/c`. Since division has precedence over subtraction, this expression can be parenthesized as `x = a-(b/c)`. The recursive procedure would therefore proceed as follows:

1. Generate code for $Expr \doteq$ `a-(b/c)`:
   (a) Generate code for $Expr_2 \doteq$ `b/c`:
      i. Generate code for $Expr_2 \doteq$ `c` using the instruction `load c`.
      ii. Generate code for $Expr_1 \doteq$ `b`, using the instruction `load b`.
      iii. Generate instruction `divp`.
   (b) Generate code for $Expr_1 \doteq$ `a`, using the instruction `load a`.
   (c) Generate instruction `subp`.
2. Generate instruction `storep x`.

The overall effect is to generate the following stack code:

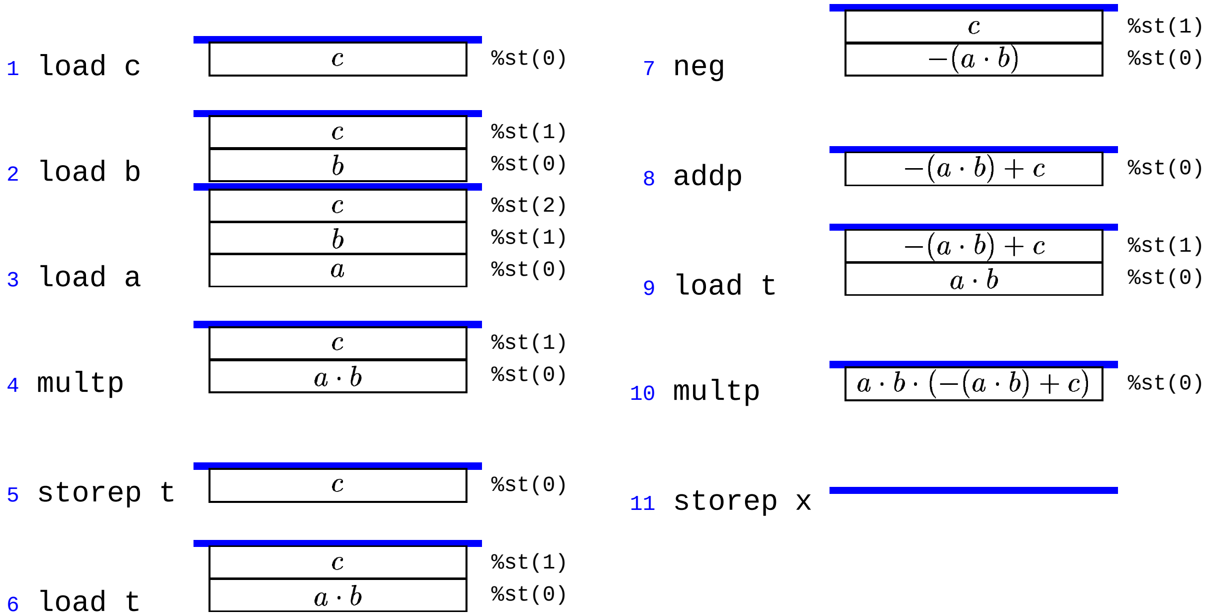


```
1 load c
2 load b
3 divp
4 load a
5 subp
6 storep x
```

Practice Problem 3.25:

Generate stack code for the expression `x = a*b/c * -(a+b*c)`. Diagram the contents of the stack for each step of your code. Remember to follow the C rules for precedence and associativity.

Stack evaluation becomes more complex when we wish to use the result of some computation multiple times. For example, consider the expression `x = (a*b)*(-(a*b)+c)`. For efficiency, we would like to compute `a*b` only once, but our stack instructions do not provide a way to keep a value on the stack once it has been used. With the set of instructions listed in Figure 3.30, we would therefore need to store the intermediate result `a+b` in some memory location, say `t`, and retrieve this value for each use. This gives the following code:



```
load c
load b
load a
multp
storep t
load t
neg
addp
load t
multp
storep x
```

This approach has the disadvantage of generating additional memory traffic, even though the register stack has sufficient capacity to hold its intermediate results. The IA32 floating-point unit avoids this inefficiency by introducing variants of the arithmetic instructions that leave their second operand on the stack, and that can use an arbitrary stack value as their second operand. In addition, it provides an instruction that can swap the top stack element with any other element. Although these extensions can be used to generate more efficient code, the simple and elegant algorithm for translating arithmetic expressions into stack code is lost.

3.14.3 Floating-Point Data Movement and Conversion Operations

Floating-point registers are referenced with the notation `%st(`i`)`, where i denotes the position relative to the top of the stack. The value i can range between 0 and 7. Register `%st(0)` is the top stack element, `%st(1)` is the second element, and so on. The top stack element can also be referenced as `%st`. When a new value is pushed onto the stack, the value in register `%st(7)` is lost. When the stack is popped, the new value in `%st(7)` is not predictable. Compilers must generate code that works within the limited capacity of the register stack.

Figure 3.31 shows the set of instructions used to push values onto the floating-point register stack. The first group of these read from a memory location, where the argument $Addr$ is a memory address given in one of the memory operand formats listed in Figure 3.3. These instructions differ by the presumed format of the source operand and hence the number of bytes that must be read from memory. Recall that the notation $\mathsf{M}_b[Addr]$ indicates an access of b bytes with starting address $Addr$. All of these instructions convert the operand to extended-precision format before pushing it onto the stack. The final load instruction `fld` is used to duplicate a stack value. That is, it pushes a copy of floating-point register `%st(`i`)` onto the stack. For example, the instruction `fld %st(0)` pushes a copy of the top stack element onto the stack.

Instruction		Source format	Source location
`flds`	$Addr$	Single	$\mathsf{M}_4[Addr]$
`fldl`	$Addr$	double	$\mathsf{M}_8[Addr]$
`fldt`	$Addr$	extended	$\mathsf{M}_{10}[Addr]$
`fildl`	$Addr$	integer	$\mathsf{M}_4[Addr]$
`fld`	`%st(`i`)`	extended	`%st(`i`)`

Figure 3.31: **Floating-point load instructions.** All convert the operand to extended-precision format and push it onto the register stack.

Instruction		Pop (Y/N)	Destination format	Destination location
`fsts`	$Addr$	N	Single	$\mathsf{M}_4[Addr]$
`fstps`	$Addr$	Y	Single	$\mathsf{M}_4[Addr]$
`fstl`	$Addr$	N	Double	$\mathsf{M}_8[Addr]$
`fstpl`	$Addr$	Y	Double	$\mathsf{M}_8[Addr]$
`fstt`	$Addr$	N	Extended	$\mathsf{M}_{10}[Addr]$
`fstpt`	$Addr$	Y	Extended	$\mathsf{M}_{10}[Addr]$
`fistl`	$Addr$	N	integer	$\mathsf{M}_4[Addr]$
`fistpl`	$Addr$	Y	integer	$\mathsf{M}_4[Addr]$
`fst`	`%st(`i`)`	N	Extended	`%st(`i`)`
`fstp`	`%st(`i`)`	Y	Extended	`%st(`i`)`

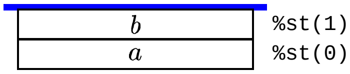
Figure 3.32: **Floating-point store instructions.** All convert from extended-precision format to the destination format. Instructions with suffix ‘p’ pop the top element off the stack.

Figure 3.32 shows the instructions that store the top stack element either in memory or in another floating-point register. There are both "popping" versions that pop the top element off the stack (similar to the `storep` instruction for our hypothetical stack evaluator), as well as nonpopping versions that leave the source value on the top of the stack. As with the floating-point load instructions, different variants of the instruction generate different formats for the result and therefore store different numbers of bytes. The first group of these store the result in memory. The address is specified using any of the memory operand formats listed in Figure 3.3. The second group copies the top stack element to some other floating-point register.

Practice Problem 3.26:

Assume for the following code fragment that register `%eax` contains an integer variable `x` and that the top two stack elements correspond to variables `a` and `b`, respectively. Fill in the boxes to diagram the stack contents after each instruction

```
    testl %eax,%eax
    jne L11
```



```
    fstp %st(0)
    jmp L9
L11:
```

```
    fstp %st(1)
L9:
```

Write a C expression describing the contents of the top stack element at the end of this code sequence in terms of `x`, `a` and `b`.

A final floating-point data movement operation allows the contents of two floating-point registers to be swapped. The instruction `fxch %st(`i`)` exchanges the contents of floating-point registers `%st(0)` and `%st(`i`)`. The notation `fxch` written with no argument is equivalent to `fxch %st(1)`, that is, swap the top two stack elements.

3.14.4 Floating-Point Arithmetic Instructions

Figure 3.33 documents some of the most common floating-point arithmetic operations. Instructions in the first group have no operands. They push the floating-point representation of some numerical constant onto the stack. There are similar instructions for such constants as π, e, and $\log_2 10$. Instructions in the second group have a single operand. The operand is always the top stack element, similar to the `neg` operation of the hypothetical stack evaluator. They replace this element with the computed result. Instructions in the third group have two operands. For each of these instructions, there are many different variants for how the operands are specified, as will be discussed shortly. For noncommutative operations such as subtraction and

Instruction	Computation
fldz	0
fld1	1
fabs	$\|Op\|$
fchs	$-Op$
fcos	$\cos Op$
fsin	$\sin Op$
fsqrt	$\sqrt{Op}$
fadd	$Op_1 + Op_2$
fsub	$Op_1 - Op_2$
fsubr	$Op_2 - Op_1$
fdiv	Op_1 / Op_2
fdivr	Op_2 / Op_1
fmul	$Op_1 \cdot Op_2$

Figure 3.33: **Floating-point arithmetic operations.** Each of the binary operations has many variants.

Instruction		Operand 1	Operand 2	(Format)	Destination	Pop %st(0) (Y/N)
fsubs	*Addr*	%st(0)	$\mathrm{M}_4[Addr]$	Single	%st(0)	N
fsubl	*Addr*	%st(0)	$\mathrm{M}_8[Addr]$	Double	%st(0)	N
fsubt	*Addr*	%st(0)	$\mathrm{M}_{10}[Addr]$	Extended	%st(0)	N
fisubl	*Addr*	%st(0)	$\mathrm{M}_4[Addr]$	integer	%st(0)	N
fsub	%st(i),%st	%st(i)	%st(0)	Extended	%st(0)	N
fsub	%st,%st(i)	%st(0)	%st(i)	Extended	%st(i)	N
fsubp	%st,%st(i)	%st(0)	%st(i)	Extended	%st(i)	Y
fsubp		%st(0)	%st(1)	Extended	%st(1)	Y

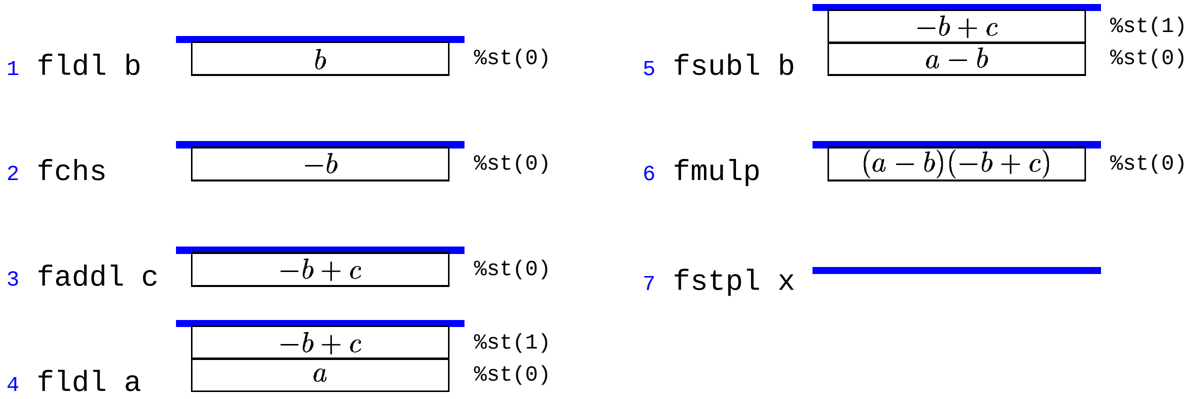
Figure 3.34: **Floating-point subtraction instructions.** All store their results into a floating-point register in extended-precision format. Instructions with suffix ‘p’ pop the top element off the stack.

division there is both a forward (e.g., `fsub`) and a reverse (e.g., `fsubr`) version, so that the arguments can be used in either order.

In Figure 3.33 we show just a single form of the subtraction operation `fsub`. In fact, this operation comes in many different variants, as shown in Figure 3.34. All compute the difference of two operands: $Op_1 - Op_2$ and store the result in some floating-point register. Beyond the simple `subp` instruction we considered for the hypothetical stack evaluator, IA32 has instructions that read their second operand from memory or from some floating-point register other than `%st(1)`. In addition, there are both popping and nonpopping variants. The first group of instructions reads the second operand from memory, either in single-precision, double-precision, or integer format. It then converts this to extended-precision format, subtracts it from the top stack element, and overwrites the top stack element. These can be seen as a combination of a floating-point load following by a stack-based subtraction operation.

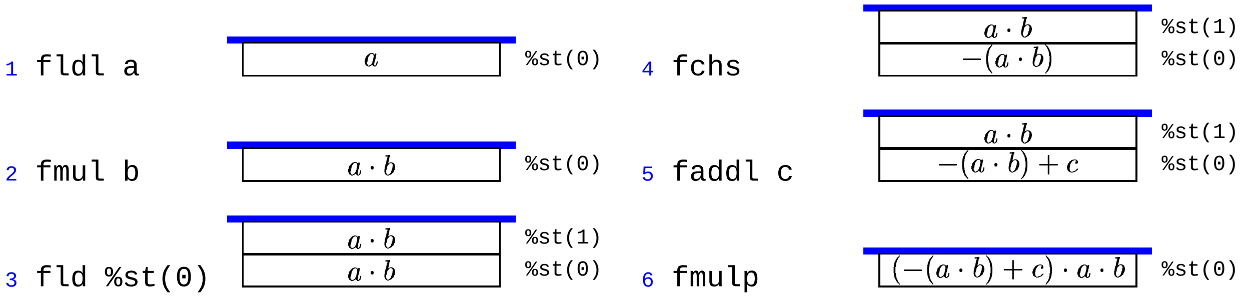
The second group of subtraction instructions use the top stack element as one argument and some other stack element as the other, but they vary in the argument ordering, the result destination, and whether or not they pop the top stack element. Observe that the assembly code line `fsubp` is shorthand for `fsubp %st,%st(1)`. This line corresponds to the `subp` instruction of our hypothetical stack evaluator. That is, it computes the difference between the top two stack elements, storing the result in `%st(1)`, and then popping `%st(0)` so that the computed value ends up on the top of the stack.

All of the binary operations listed in Figure 3.33 come in all of the variants listed for `fsub` in Figure 3.34. As an example, we can rewrite the code for the expression `x = (a-b)*(-b+c)` using the IA32 instructions. For exposition purposes we will still use symbolic names for memory locations and we assume these are double-precision values.



```
1 fldl b
2 fchs
3 faddl c
4 fldl a
5 fsubl b
6 fmulp
7 fstpl x
```

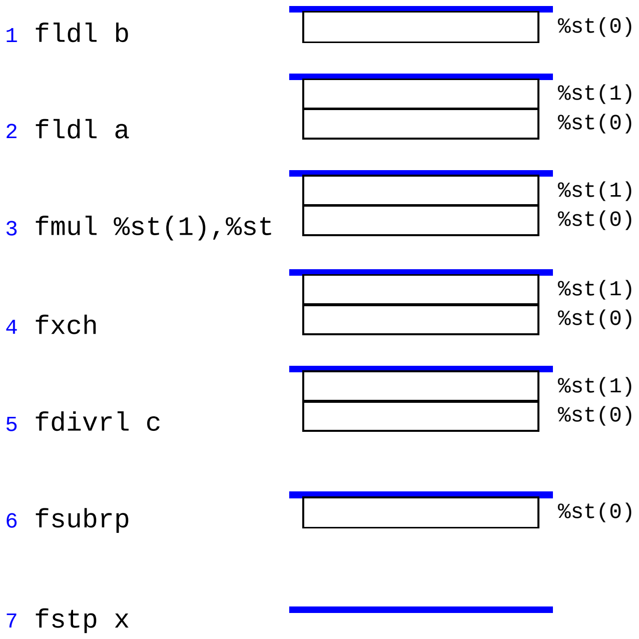
As another example, we can write the code for the expression `x = (a*b)+(-(a*b)+c)` as follows. Observe how the instruction `fld %st(0)` is used to create two copies of `a*b` on the stack, avoiding the need to save the value in a temporary memory location.



```
fldl a
fmul b
fld %st(0)
fchs
faddl c
fmulp
```

Practice Problem 3.27:

Diagram the stack contents after each step of the following code:



```
fldl b
fldl a
fmul %st(1),%st
fxch
fdivrl c
fsubrp
fstp x
```

Give an expression describing this computation.

3.14.5 Using Floating Point in Procedures

Floating-point arguments are passed to a calling procedure on the stack, just as are integer arguments. Each parameter of type `float` requires 4 bytes of stack space, while each parameter of type `double` requires 8. For functions whose return values are of type `float` or `double`, the result is returned on the top of the floating-point register stack in extended-precision format.

As an example, consider the following function

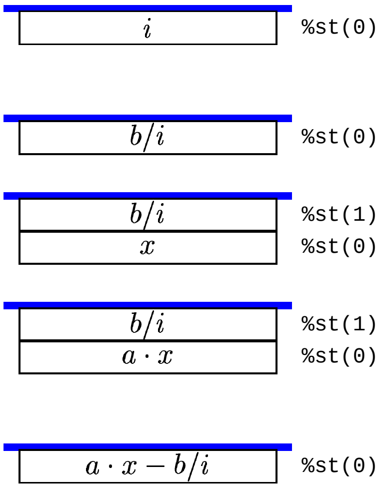
```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

Arguments a, x, b, and i will be at byte offsets 8, 16, 20, and 28 relative to %ebp, respectively, as follows:

Offset	8	16	20	28
Contents	a	x	b	i

The body of the generated code, and the resulting stack values are as follows:

```
fildl 28(%ebp)

fdivrl 20(%ebp)

flds 16(%ebp)

fmull 8(%ebp)

fsubp %st,%st(1)
```



Practice Problem 3.28:

For a function funct2 with arguments a, x, b, and i (and a different declaration than that of funct, the compiler generates the following code for the function body:

```
movl 8(%ebp),%eax
fldl 12(%ebp)
flds 20(%ebp)
movl %eax,-4(%ebp)
fildl -4(%ebp)
fxch %st(2)
faddp %st,%st(1)
fdivrp %st,%st(1)
fld1
flds 24(%ebp)
faddp %st,%st(1)
```

The returned value is of type double. Write C code for funct2. Be sure to correctly declare the argument types.

Ordered		Unordered		Op_2	Type	Number of pops
fcoms	*Addr*	fucoms	*Addr*	$\mathtt{M}_4[Addr]$	Single	0
fcoml	*Addr*	fucoml	*Addr*	$\mathtt{M}_8[Addr]$	Double	0
fcom	%st(i)	fucom	%st(i)	%st(i)	Extended	0
fcom		fucom		%st(1)	Extended	0
fcomps	*Addr*	fucomps	*Addr*	$\mathtt{M}_4[Addr]$	Single	1
fcompl	*Addr*	fucompl	*Addr*	$\mathtt{M}_8[Addr]$	Double	1
fcomp	%st(i)	fucomp	%st(i)	%st(i)	Extended	1
fcomp		fucomp		%st(1)	Extended	1
fcompp		fucompp		%st(1)	Extended	2

Figure 3.35: **Floating-point comparison instructions.** Ordered vs. unordered comparisons differ in their treatment of *NaN*'s.

3.14.6 Testing and Comparing Floating-Point Values

Similar to the integer case, determining the relative values of two floating-point numbers involves using a comparison instruction to set condition codes and then testing these condition codes. For floating point, however, the condition codes are part of the *floating-point status word*, a 16-bit register that contains various flags about the floating-point unit. This status word must be transferred to an integer word, and then the particular bits must be tested.

There are a number of different floating-point comparison instructions as documented in Figure 3.35. All of them perform a comparison between operands Op_1 and Op_2, where Op_1 is the top stack element. Each line of the table documents two different comparison types: an *ordered* comparison used for comparisons such as $<$ and $\leq$, and an *unordered* comparison used for equality comparisons. The two comparisons differ only in their treatment of *NaN* values, since there is no relative ordering between *NaN*'s and other values. For example, if variable x is a *NaN* and variable y is some other value, then both expressions x < y and x >= y should yield 0.

The various forms of comparison instructions also differ in the location of operand Op_2, analogous to the different forms of floating-point load and floating-point arithmetic instructions. Finally, the various forms differ in the number of elements popped off the stack after the comparison is completed. Instructions in the first group shown in the table do not change the stack at all. Even for the case where one of the arguments is in memory, this value is not on the stack at the end. Operations in the second group pop element Op_1 off the stack. The final operation pops both Op_1 and Op_2 off the stack.

The floating-point status word is transferred to an integer register with the fnstsw instruction. The operand for this instruction is one of the 16-bit register identifiers shown in Figure 3.2, for example, %ax. The bits in the status word encoding the comparison results are in bit positions 0, 2, and 6 of the high-order byte of the status word. For example, if we use instruction fnstw %ax to transfer the status word, then the relevant bits will be in %ah. A typical code sequence to select these bits is then:

```
fnstsw %ax        Store floating point status word in %ax
andb $69,%ah      Mask all but bits 0, 2, and 6
```

Note that 69_{10} has bit representation [00100101], that is, it has 1s in the three relevant bit positions. Figure

$Op_1 : Op_2$	Binary	Decimal
$>$	[00000000]	0
$<$	[00000001]	1
$=$	[00100000]	64
Unordered	[00100101]	69

Figure 3.36: **Encoded results from floating-point comparison.** The results are encoded in the high-order byte of the floating-point status word after masking out all but bits 0, 2, and 6.

3.36 shows the possible values of byte `%ah` that would result from this code sequence. Observe that there are only four possible outcomes for comparing operands Op_1 and Op_2: the first is either greater, less, equal, or incomparable to the second, where the latter outcome only occurs when one of the values is a NaN.

As an example, consider the following procedure:

```
int less(double x, double y)
{
    return x < y;
}
```

The compiled code for the function body is as follows:

```
  fldl 16(%ebp)      Push y
  fcompl 8(%ebp)     Compare y:x
  fnstsw %ax         Store floating point status word in %ax
  andb $69,%ah       Mask all but bits 0, 2, and 6
  sete %al           Test for comparison outcome of 0 (>)
  movzbl %al,%eax    Copy low order byte to result, and set rest to 0
```

Practice Problem 3.29:

Show how, by inserting a single line of assembly code into the preceding code sequence, you can implement the following function:

```
int greater(double x, double y)
{
    return x > y;
}
```

This completes our coverage of assembly-level, floating-point programming with IA32. Even experienced programmers find this code arcane and difficult to read. The stack-based operations, the awkwardness of getting status results from the FPU to the main processor, and the many subtleties of floating-point computations combine to make the machine code lengthy and obscure. It is remarkable that the modern processors manufactured by Intel and its competitors can achieve respectable performance on numeric programs given the form in which they are encoded.

3.15 *Embedding Assembly Code in C Programs

In the early days of computing, most programs were written in assembly code. Even large-scale operating systems were written without the help of high-level languages. This becomes unmanageable for programs of significant complexity. Since assembly code does not provide any form of type checking, it is very easy to make basic mistakes, such as using a pointer as an integer rather than dereferencing the pointer. Even worse, writing in assembly code locks the entire program into a particular class of machine. Rewriting an assembly language program to run on a different machine can be as difficult as writing the entire program from scratch.

> **Aside: Writing large programs in assembly code.**
> Frederick Brooks, Jr., a pioneer in computer systems wrote a fascinating account of the development of OS/360, an early operating system for IBM machines [5] that still provides important object lessons today. He became a devoted believer in high-level languages for systems programming as a result of this effort. Surprisingly, however, there is an active group of programmers who take great pleasure in writing assembly code for IA32. They communicate with one another via the Internet news group `comp.lang.asm.x86`. Most of them write computer games for the DOS operating system. **End Aside.**

Early compilers for higher-level programming languages did not generate very efficient code and did not provide access to the low-level object representations, as is often required by systems programmers. Programs requiring maximum performance or requiring access to object representations were still often written in assembly code. Nowadays, however, optimizing compilers have largely removed performance optimization as a reason for writing in assembly code. Code generated by a high quality compiler is generally as good or even better than what can be achieved manually. The C language has largely eliminated machine access as a reason for writing in assembly code. The ability to access low-level data representations through unions and pointer arithmetic, along with the ability to operate on bit-level data representations, provide sufficient access to the machine for most programmers. For example, almost every part of a modern operating system such as Linux is written in C.

Nonetheless, there are times when writing in assembly code is the only option. This is especially true when implementing an operating system. For example, there are a number of special registers storing process state information that the operating system must access. There are either special instructions or special memory locations for performing input and output operations. Even for application programmers, there are some machine features, such as the values of the condition codes, that cannot be accessed directly in C.

The challenge then is to integrate code consisting mainly of C with a small amount written in assembly language. One method is to write a few key functions in assembly code, using the same conventions for argument passing and register usage as are followed by the C compiler. The assembly functions are kept in a separate file, and the compiled C code is combined with the assembled assembly code by the linker. For example, if file `p1.c` contains C code and file `p2.s` contains assembly code, then the compilation command

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

will cause file `p1.c` to be compiled, file `p2.s` to be assembled, and the resulting object code to be linked to form an executable program `p`.

3.15.1 Basic Inline Assembly

With GCC, it is also possible to mix assembly with C code. Inline assembly allows the user to insert assembly code directly into the code sequence generated by the compiler. Features are provided to specify instruction operands and to indicate to the compiler which registers are being overwritten by the assembly instructions. The resulting code is, of course, highly machine-dependent, since different types of machines do not have compatible machine instructions. The `asm` directive is also specific to GCC, creating an incompatibility with many other compilers. Nonetheless, this can be a useful way to keep the amount of machine-dependent code to an absolute minimum.

Inline assembly is documented as part of the GCC information archive. Executing the command `info gcc` on any machine with GCC installed will give a hierarchical document reader. Inline assembly is documented by first following the link titled “C Extensions” and then the link titled “Extended Asm.” Unfortunately, the documentation is somewhat incomplete and imprecise.

The basic form of inline assembly is to write code that looks like a procedure call:

`asm(` *code-string* `);`

The term *code-string* denotes an assembly code sequence given as a quoted string. The compiler will insert this string verbatim into the assembly code being generated, and hence the compiler-supplied and the user-supplied assembly will be combined. The compiler does not check the string for errors, and so the first indication of a problem might be an error report from the assembler.

We illustrate the use of `asm` by an example where having access to the condition codes can be useful. Consider functions with the following prototypes:

```
int ok_smul(int x, int y, int *dest);

int ok_umul(unsigned x, unsigned y, unsigned *dest);
```

Each is supposed to compute the product of arguments `x` and `y` and store the result in the memory location specified by argument `dest`. As return values, they should return 0 when the multiplication overflows and 1 when it does not. We have separate functions for signed and unsigned multiplication, since they overflow under different circumstances.

Examining the documentation for the IA32 multiply instructions `mul` and `imul`, we see that both set the carry flag `CF` when they overflow. Examining Figure 3.10, we see that the instruction `setae` can be used to set the low-order byte of a register to 0 when this flag is set and to 1 otherwise. Thus, we wish to insert this instruction into the sequence generated by the compiler.

In an attempt to use the least amount of both assembly code and detailed analysis, we attempt to implement `ok_smul` with the following code:

code/asm/okmul.c

```
/* First attempt.  Does not work */
int ok_smul1(int x, int y, int *dest)
{
    int result = 0;

    *dest = x*y;
    asm("setae %al");
    return result;
}
```

code/asm/okmul.c

The strategy here is to exploit the fact that register `%eax` is used to store the return value. Assuming the compiler uses this register for variable `result`, the first line will set the register to 0. The inline assembly will insert code that sets the low-order byte of this register appropriately, and the register will be used as the return value.

Unfortunately, GCC has its own ideas of code generation. Instead of setting register `%eax` to 0 at the beginning of the function, the generated code does so at the very end, and so the function always returns 0. The fundamental problem is that the compiler has no way to know what the programmer's intentions are, and how the assembly statement should interact with the rest of the generated code.

By a process of trial and error (we will develop more systematic approaches shortly), we were able to generate code that works, but that also is less than ideal:

code/asm/okmul.c

```
/* Second attempt.  Works in limited contexts */
int dummy = 0;

int ok_smul2(int x, int y, int *dest)
{
    int result;

    *dest = x*y;
    result = dummy;
    asm("setae %al");
    return result;
}
```

code/asm/okmul.c

This code uses the same strategy as before, but it reads a global variable `dummy` to initialize `result` to 0. Compilers are typically more conservative about generating code involving global variables, and therefore less likely to rearrange the ordering of the computations.

The preceding code depends on quirks of the compiler to get proper behavior. In fact, it only works when compiled with optimization enabled (command line flag `-O`). When compiled without optimization, it stores `result` on the stack and retrieves its value just before returning, overwriting the value set by the `setae` instruction. The compiler has no way of knowing how the inserted assembly language relates to the rest of the code, because we provided the compiler no such information.

3.15.2 Extended Form of `asm`

GCC provides an extended version of the `asm` that allows the programmer to specify which program values are to be used as operands to an assembly code sequence and which registers are overwritten by the assembly code. With this information the compiler can generate code that will correctly set up the required source values, execute the assembly instructions, and make use of the computed results. It will also have information it requires about register usage so that important program values are not overwritten by the assembly code instructions.

The general syntax of an extended assembly sequence is

`asm(` *code-string* [`:` *output-list* [`:` *input-list* [`:` *overwrite-list*]]] `);`

where the square brackets denote optional arguments. The declaration contains a string describing the assembly code sequence, followed by optional lists of outputs (i.e., results generated by the assembly code), inputs (i.e., source values for the assembly code), and registers that are overwritten by the assembly code. These lists are separated by the colon (‘`:`’) character. As the square brackets show, we only include lists up to the last nonempty list.

The syntax for the code string is reminiscent of that for the format string in a `printf` statement. It consists of a sequence of assembly code instructions separated by the semicolon (‘`;`’) character. Input and output operands are denoted by references `%0`, `%1`, and so on, up to possibly `%9`. Operands are numbered, according to their ordering first in the output list and then in the input list. Register names such as “`%eax`” must be written with an extra ‘`%`’ symbol, such as “`%%eax`.”

The following is a better implementation of `ok_smul` using the extended assembly statement to indicate to the compiler that the assembly code generates the value for the variable `result`:

code/asm/okmul.c

```c
/* Uses the extended assembly statement to get reliable code */
int ok_smul3(int x, int y, int *dest)
{
    int result;

    *dest = x*y;

    /* Insert the following assembly code:
       setae %bl            # Set low-order byte
       movzbl %bl, result   # Zero extend to be result
    */
    asm("setae %%bl; movzbl %%bl,%0"
        : "=r" (result)  /* Output    */
        :                /* No inputs */
        : "%ebx"         /* Overwrites */
        );

    return result;
}
```

code/asm/okmul.c

The first assembly instruction stores the test result in the single-byte register `%bl`. The second instruction then zero-extends and copies the value to whatever register the compiler chooses to hold `result`, indicated by operand `%0`. The output list consists of pairs of values separated by spaces. (In this example there is only a single pair). The first element of the pair is a string indicating the operand type, where '`r`' indicates an integer register and '`=`' indicates that the assembly code assigns a value to this operand. The second element of the pair is the operand enclosed in parentheses. It can be any assignable value (known in C as an *lvalue*). The compiler will generate the necessary code sequence to perform the assignment. The input list has the same general format, where the operand can be any C expression. The compiler will generate the necessary code to evaluate the expression. The overwrite list simply gives the names of the registers (as quoted strings) that are overwritten.

The preceding code works regardless of the compilation flags. As this example illustrates, it may take a little creative thinking to write assembly code that will allow the operands to be described in the required form. For example, there are no direct ways to specify a program value to use as the destination operand for the `setae` instruction, since the operand must be a single byte. Instead, we write a code sequence based on a specific register and then use an extra data movement instruction to copy the resulting value to some part of the program state.

Practice Problem 3.30:

GCC provides a facility for extended-precision arithmetic. This can be used to implement function `ok_smul`, with the advantage that it is portable across machines. A variable declared as type "`long long`" will have twice the size of normal `long` variable. Thus, the statement

```
long long prod = (long long) x * y;
```

will compute the full 64-bit product of `x` and `y`. Using this facility, write a version of `ok_smul` that does not use any `asm` statements.

One would expect the same code sequence could be used for `ok_umul`, but GCC uses the `imull` (signed multiply) instruction for both signed and unsigned multiplication. This generates the correct value for either product, but it sets the carry flag according to the rules for signed multiplication. We therefore need to include an assembly-code sequence that explicitly performs unsigned multiplication using the `mull` instruction as documented in Figure 3.9, as follows:

code/asm/okmul.c

```
/* Uses the extended assembly statement */
int ok_umul(unsigned x, unsigned y, unsigned *dest)
{
    int result;

    /* Insert the following assembly code:
       movl  x,%eax         # Get x
       mull  y              # Unsigned multiply by y
       movl  %eax, *dest    # Store low-order 4 bytes at dest
```

```
        setae %dl            # Set low-order byte
        movzbl %dl, result   # Zero extend to be result
     */
     asm("movl %2,%%eax; mull %3; movl %%eax,%0;
          setae %%dl; movzbl %%dl,%1"
         : "=r" (*dest), "=r" (result) /* Outputs    */
         : "r"  (x),     "r"  (y)      /* Inputs     */
         : "%eax", "%edx"              /* Overwrites */
         );

     return result;
}
```

code/asm/okmul.c

Recall that the `mull` instruction requires one of its arguments to be in register `%eax` and is given the second argument as an operand. We indicate this in the `asm` statement by using a `movl` to move program value `x` to `%eax` and indicating that program value `y` should be the argument for the `mull` instruction. The instruction then stores the 8-byte product in two registers with `%eax` holding the low-order 4 bytes and `%edx` holding the high-order bytes. We then use register `%edx` to construct the return value. As this example illustrates, comma (‘ `,` ’) characters are used to separate pairs of operands in the input and output lists, and register names in the overwrite list. Note that we were able to specify `*dest` as an output of the second `movl` instruction, since this is an assignable value. The compiler then generates the correct machine code to store the value in `%eax` at this memory location.

To see how the compiler generates code in connection with an `asm` statement, here is the code generated for `ok_umul`:

```
     Set up asm inputs
1     movl 8(%ebp),%ecx          Load x into %ecx
2     movl 12(%ebp),%ebx         Load y into %ebx
3     movl 16(%ebp),%esi         Load dest into %esi
     The following instruction was generated by asm.
       Input registers: %ecx for x, %ebx for y
       Output registers: %ecx for product, %ebx for result
4     movl %ecx,%eax; mull %ebx; movl %eax,%ecx;
5     setae %dl; movzbl %dl,%ebx
     Process asm outputs
6     movl %ecx,(%esi)           Store product at dest
7     movl %ebx,%eax             Set result as return value
```

Lines 1–3 of this code fetch the procedure arguments and store them in registers. Note that it does not use registers `%eax` or `%edx`, since we have declared that these will be overwritten. Our inline assembly statement appears as lines 4 and 5, but with register names substituted for the arguments. In particular, it will use registers `%ecx` for argument `%2` (`x`), and `%ebx` for argument `%3` (`y`). The product will be held temporarily in `%ecx`, while it uses register `%ebx` for argument `%1` (`result`). Line 6 then stores the product at `dest`, completing the processing of argument `%0` (`*dest`). Line 7 copies `result` to register `%eax` as the return value. Thus, the compiler generated not only the code indicated by our `asm` statement, but code to set up the statement inputs (lines 1–3) and to make use of the outputs (lines 6–7).

Although the syntax of the `asm` statement is somewhat arcane, and its use makes the code less portable, this statement can be very useful for writing programs that accesses machine-level features using a minimal amount of assembly code. We have found that a certain amount of trial and error is required to get code that works. The best strategy is to compile the code with the `-S` switch and then examine the generated assembly code to see if it will have the desired effect. The code should be tested with different settings of switches such as with and without the `-O` flag.

3.16 Summary

In this chapter, we have peered beneath the layer of abstraction provided by a high-level 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 will have efficient mappings onto the machine. We have also seen examples where the high-level language abstraction hides important details about the operation of a program. For example, the behavior of floating-point code can depend on whether values are held in registers or in memory. In Chapter 7, we will see many examples where we need to know whether a program variable is on the run-time stack, in some dynamically allocated data structure, or in some global storage locations. Understanding how programs map onto machines makes it easier to understand the difference between these kinds of storage.

Assembly language is very different from C code. In assembly language 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 the that the lack of bounds checking in C makes many programs prone to buffer overflows, and this has made many systems vulnerable to attacks by malicious intruders.

We have only examined the mapping of C onto IA32, 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++ simply 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. The advantage of this approach is that the same Java byte code can be executed on many different machines, whereas the machine code we have considered runs only under IA32.

Bibliographic Notes

The best references on IA32 are from Intel. Two useful references are part of their series on software development. The basic architecture manual [18] gives an overview of the architecture from the perspective of an assembly-language programmer, and the instruction set reference manual [19] gives detailed descriptions of the different instructions. These references contain far more information than is required to understand Linux code. In particular, with flat mode addressing, all of the complexities of the segmented addressing scheme can be ignored.

The GAS format used by the Linux assembler is very different from the standard format used in Intel documentation and by other compilers (particularly those produced by Microsoft). One main distinction is that the source and destination operands are given in the opposite order

On a Linux machine, running the command `info as` will display information about the assembler. One of the subsections documents machine-specific information, including a comparison of GAS with the more standard Intel notation. Note that GCC refers to these machines as "`i386`"—it generates code that could even run on a 1985 vintage machine.

Muchnick's book on compiler design [56] is considered the most comprehensive reference on code optimization techniques. It covers many of the techniques we discuss here, such as register usage conventions and the advantages of generating code for loops based on their `do-while` form.

Much has been written about the use of buffer overflow to attack systems over the Internet. Detailed analyses of the 1988 Internet worm have been published by Spafford [75] as well as by members of the team at MIT who helped stop its spread [26]. Since then, a number of papers and projects have generated about both creating and preventing buffer overflow attacks, such as [20].

Homework Problems

Homework Problem 3.31 [Category 1]:

You are given the information that follows. A function with prototype

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

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

```
movl 16(%ebp),%eax
movl 12(%ebp),%edx
subl %eax,%edx
movl %edx,%eax
imull 8(%ebp),%edx
sall $31,%eax
sarl $31,%eax
xorl %edx,%eax
```

Parameters `x`, `y`, and `z` are stored at memory locations with offsets 8, 12, and 16 relative to the address in register `%ebp`. The code stores the return value in register `%eax`.

Write C code for `decode2` that will have an effect equivalent to our assembly code. You can test your solution by compiling your code with the `-S` switch. Your compiler may not generate identical code, but it should be functionally equivalent.

Homework Problem 3.32 [Category 2]:

The following C code is almost identical to that in Figure 3.12:

```
int absdiff2(int x, int y)
{
    int result;

    if (x < y)
        result = y-x;
    else
        result = x-y;
    return result;
}
```

When compiled, however, it gives a different form of assembly code:

```
  movl 8(%ebp),%edx
  movl 12(%ebp),%ecx
  movl %edx,%eax
  subl %ecx,%eax
  cmpl %ecx,%edx
  jge .L3
  movl %ecx,%eax
  subl %edx,%eax
.L3:
```

A. What subtractions are performed when $x < y$? When $x \geq y$?

B. In what way does this code deviate from the standard implementation of if-else described previously?

C. Using C syntax (including goto's), show the general form of this translation.

D. What restrictions must be imposed on the use of this translation to guarantee that it has the behavior specified by the C code?

Homework Problem 3.33 [Category 2]:

The code that follows shows an example of branching on an enumerated type value in a switch statement. Recall that enumerated types in C are simply a way to introduce a set of names having associated integer values. By default, the values assigned to the names go from 0 upward. In our code, the actions associated with the different case labels have been omitted.

```
/* Enumerated type creates set of constants numbered 0 and upward */
```

```
 The jump targets
 Arguments p1 and p2 are in registers %ebx and %ecx.
.L15:                 MODE_A
  movl (%ecx),%edx
  movl (%ebx),%eax
  movl %eax,(%ecx)
  jmp .L14
  .p2align 4,,7       Inserted to optimize cache performance
.L16:                 MODE_B
  movl (%ecx),%eax
  addl (%ebx),%eax
  movl %eax,(%ebx)
  movl %eax,%edx
  jmp .L14
  .p2align 4,,7       Inserted to optimize cache performance
.L17:                 MODE_C
  movl $15,(%ebx)
  movl (%ecx),%edx
  jmp .L14
  .p2align 4,,7       Inserted to optimize cache performance
.L18:                 MODE_D
  movl (%ecx),%eax
  movl %eax,(%ebx)
.L19:                 MODE_E
  movl $17,%edx
  jmp .L14
  .p2align 4,,7       Inserted to optimize cache performance
.L20:
  movl $-1,%edx
.L14:                 default
  movl %edx,%eax      Set return value
```

Figure 3.37: **Assembly code for Problem 3.33.** This code implements the different branches of a `switch` statement.

```
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

int switch3(int *p1, int *p2, mode_t action)
{
  int result = 0;
  switch(action) {
  case MODE_A:

  case MODE_B:

  case MODE_C:

  case MODE_D:

  case MODE_E:

  default:

  }
  return result;
}
```

The part of the generated assembly code implementing the different actions is shown shown in Figure 3.37. The annotations indicate the values stored in the registers and the case labels for the different jump destinations.

A. What register corresponds to program variable `result`?

B. Fill in the missing parts of the C code. Watch out for cases that fall through.

Homework Problem 3.34 [Category 2]:

Switch statements are particularly challenging to reverse engineer from the object code. In the following procedure, the body of the switch statement has been removed.

```
int switch_prob(int x)
{
    int result = x;

    switch(x) {

        /* Fill in code here */
    }

    return result;
}
```

```
 1 080483c0 <switch_prob>:
 2  80483c0:  55                      push   %ebp
 3  80483c1:  89 e5                   mov    %esp,%ebp
 4  80483c3:  8b 45 08                mov    0x8(%ebp),%eax
 5  80483c6:  8d 50 ce                lea    0xffffffce(%eax),%edx
 6  80483c9:  83 fa 05                cmp    $0x5,%edx
 7  80483cc:  77 1d                   ja     80483eb <switch_prob+0x2b>
 8  80483ce:  ff 24 95 68 84 04 08    jmp    *0x8048468(,%edx,4)
 9  80483d5:  c1 e0 02                shl    $0x2,%eax
10  80483d8:  eb 14                   jmp    80483ee <switch_prob+0x2e>
11  80483da:  8d b6 00 00 00 00       lea    0x0(%esi),%esi
12  80483e0:  c1 f8 02                sar    $0x2,%eax
13  80483e3:  eb 09                   jmp    80483ee <switch_prob+0x2e>
14  80483e5:  8d 04 40                lea    (%eax,%eax,2),%eax
15  80483e8:  0f af c0                imul   %eax,%eax
16  80483eb:  83 c0 0a                add    $0xa,%eax
17  80483ee:  89 ec                   mov    %ebp,%esp
18  80483f0:  5d                      pop    %ebp
19  80483f1:  c3                      ret
20  80483f2:  89 f6                   mov    %esi,%esi
```

Figure 3.38: **Disassembled code for Problem 3.34.**

Figure 3.38 shows the disassembled object code for the procedure. We are only interested in the part of code shown on lines 4 through 16. We can see on line 4 that parameter `x` (at offset 8 relative to `%ebp`) is loaded into register `%eax`, corresponding to program variable `result`. The "`lea 0x0(%esi),%esi`" instruction on line 11 is a nop instruction inserted to make the instruction on line 12 start on an address that is a multiple of 16.

The jump table resides in a different area of memory. Using the debugger GDB we can examine the six 4-byte words of memory starting at address `0x8048468` with the command `x/6w 0x8048468`. GDB prints the following:

```
(gdb) x/6w 0x8048468
0x8048468:  0x080483d5      0x080483eb      0x080483d5      0x080483e0
0x8048478:  0x080483e5      0x080483e8
(gdb)
```

Fill in the body of the switch statement with C code that will have the same behavior as the object code.

Homework Problem 3.35 [Category 2]:

The code generated by the C compiler for `var_prod_ele` (Figure 3.25(b)) is not optimal. Write code for this function based on a hybrid of procedures `fix_prod_ele_opt` (Figure 3.24) and `var_prod_ele_opt` (Figure 3.25) that is correct for all values of `n`, but compiles into code that can keep all of its temporary data in registers.

Recall that the processor only has six registers available to hold temporary data, since registers `%ebp` and `%esp` cannot be used for this purpose. One of these registers must be used to hold the result of the multiply instruction. Hence, you must reduce the number of local variables in the loop from six (result, Aptr, B, nTjPk, n, and cnt) to five.

Homework Problem 3.36 [Category 2]:

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

im/code/asm/structprob-ans.c

```
typedef struct {
    int left;
    a_struct a[CNT];
    int right;
} b_struct;

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

im/code/asm/structprob-ans.c

Unfortunately, the '.h' file defining the compile-time constant CNT and the structure a_struct are in files for which you do not have access privileges. Fortunately, you have access to a '.o' version of code, which you are able to disassemble with the objdump program, yielding the disassembly shown in Figure 3.39.

Using your reverse engineering skills, deduce the following:

A. The value of CNT.

B. A complete declaration of structure a_struct. Assume that the only fields in this structure are idx and x.

Homework Problem 3.37 [Category 1]:

Write a function good_echo that reads a line from standard input and writes it to standard output. Your implementation should work for an input line of arbitrary length. You may use the library function fgets, but you must make sure your function works correctly even when the input line requires more space than you have allocated for your buffer. Your code should also check for error conditions and return when one is encounted. You should refer to the definitions of the standard I/O functions for documentation [32, 41].

Homework Problem 3.38 [Category 3]:

In this problem, you will mount a buffer overflow attack on your own program. As stated earlier, we do not condone using this or any other form of attack to gain unauthorized access to a system, but by doing this exercise, you will learn a lot about machine-level programming.

Download the file bufbomb.c from the CS:APP website and compile it to create an executable program. In bufbomb.c, you will find the following functions:

```
int getbuf()
{
    char buf[12];
    getxs(buf);
    return 1;
}

void test()
{
  int val;
  printf("Type Hex string:");
  val = getbuf();
  printf("getbuf returned 0x%x\n", val);
}
```

The function `getxs` (also in `bufbomb.c`) is similar to the library `gets`, except that it reads characters encoded as pairs of hex digits. For example, to give it a string "`0123`," the user would type in the string "`30 31 32 33`." The function ignores blank characters. Recall that decimal digit x has ASCII representation `0x3`x.

A typical execution of the program is as follows:

```
unix> ./bufbomb
Type Hex string: 30 31 32 33
getbuf returned 0x1
```

Looking at the code for the `getbuf` function, it seems quite apparent that it will return value 1 whenever it is called. It appears as if the call to `getxs` has no effect. Your task is to make `getbuf` return -559038737 (`0xdeadbeef`) to `test`, simply by typing an appropriate hexadecimal string to the prompt.

The following suggestions may help you solve the problem:

- Use OBJDUMP to create a disassembled version of `bufbomb`. Study this closely to determine how the stack frame for `getbuf` is organized and how overflowing the buffer will alter the saved program state.

- Run your program under GDB. Set a breakpoint within `getbuf` and run to this breakpoint. Determine such parameters as the value of `%ebp` and the saved value of any state that will be overwritten when you overflow the buffer.

- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with GCC and disassemble it with OBJDUMP. You should be able to get the exact byte sequence that you will type at the prompt. OBJDUMP will produce some pretty strange looking assembly instructions when it tries to disassemble the data in your file, but the hexadecimal byte sequence should be correct.

```
00000000 <test>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   8b 45 08                mov    0x8(%ebp),%eax
   7:   8b 4d 0c                mov    0xc(%ebp),%ecx
   a:   8d 04 80                lea    (%eax,%eax,4),%eax
   d:   8d 44 81 04             lea    0x4(%ecx,%eax,4),%eax
  11:   8b 10                   mov    (%eax),%edx
  13:   c1 e2 02                shl    $0x2,%edx
  16:   8b 99 b8 00 00 00       mov    0xb8(%ecx),%ebx
  1c:   03 19                   add    (%ecx),%ebx
  1e:   89 5c 02 04             mov    %ebx,0x4(%edx,%eax,1)
  22:   5b                      pop    %ebx
  23:   89 ec                   mov    %ebp,%esp
  25:   5d                      pop    %ebp
  26:   c3                      ret
```

Figure 3.39: **Disassembled code for Problem 3.36.**

Keep in mind that your attack is very machine and compiler specific. You may need to alter your string when running on a different machine or with a different version of GCC.

Homework Problem 3.39 [Category 2]:

Use the `asm` statement to implement a function with the prototype

```
void full_umul(unsigned x, unsigned y, unsigned dest[]);
```

This function should compute the full 64-bit product of its arguments and store the results in the destination array, with `dest[0]` having the low-order 4 bytes and `dest[1]` having the high-order 4 bytes.

Homework Problem 3.40 [Category 2]:

The `fscale` instruction computes the function $x \cdot 2^{\mathsf{RTZ}(y)}$ for floating-point values x and y, where RTZ denotes the round-toward-zero function, rounding positive numbers downward and negative numbers upward. The arguments to `fscale` come from the floating-point register stack, with x in `%st(0)` and y in `%st(1)`. It writes the computed value written `%st(0)` without popping the second argument. (The actual implementation of this instruction works by adding $\mathsf{RTZ}(y)$ to the exponent of x).

Using an `asm` statement, implement a function with the prototype

```
double scale(double x, int n, double *dest);
```

which computes $x \cdot 2^n$ using the `fscale` instruction and stores the result at the location designated by pointer `dest`. Extended `asm` does not provide very good support for IA32 floating point. In this case, however, you can access the arguments from the program stack.

Solutions to Practice Problems

Problem 3.1 Solution: [Pg. 128]

This exercise gives you practice with the different operand forms.

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

Problem 3.2 Solution: [Pg. 132]

Reverse engineering is a good way to understand systems. In this case, we want to reverse the effect of the C compiler to determine what C code gave rise to this assembly code. The best way is to run a "simulation," starting with values `x`, `y`, and `z` at the locations designated by pointers `xp`, `yp`, and `zp`, respectively. We would then get the following behavior:

```
movl 8(%ebp),%edi      xp
movl 12(%ebp),%ebx     yp
movl 16(%ebp),%esi     zp
movl (%edi),%eax       x
movl (%ebx),%edx       y
movl (%esi),%ecx       z
movl %eax,(%ebx)       *yp = x
movl %edx,(%esi)       *zp = y
movl %ecx,(%edi)       *xp = z
```

From this we can generate the following C code:

code/asm/decode1-ans.c

```
void decode1(int *xp, int *yp, int *zp)
{
    int tx = *xp;
    int ty = *yp;
    int tz = *zp;

    *yp = tx;
    *zp = ty;
    *xp = tz;
}
```

code/asm/decode1-ans.c

Problem 3.3 Solution: [Pg. 133]

This exercise demonstrates the versatility of the `leal` instruction and gives you more practice in deciphering the different operand forms. Note that although the operand forms are classified as type “Memory” in Figure 3.3, no memory access occurs.

Expression	Result
`leal 6(%eax), %edx`	$6 + x$
`leal (%eax,%ecx), %edx`	$x + y$
`leal (%eax,%ecx,4), %edx`	$x + 4y$
`leal 7(%eax,%eax,8), %edx`	$7 + 9x$
`leal 0xA(,$ecx,4), %edx`	$10 + 4y$
`leal 9(%eax,%ecx,2), %edx`	$9 + x + 2y$

Problem 3.4 Solution: [Pg. 134]

This problem gives you a chance to test your understanding of operands and the arithmetic instructions.

Instruction	Destination	Value
`addl %ecx,(%eax)`	`0x100`	`0x100`
`subl %edx,4(%eax)`	`0x104`	`0xA8`
`imull $16,(%eax,%edx,4)`	`0x10C`	`0x110`
`incl 8(%eax)`	`0x108`	`0x14`
`decl %ecx`	`%ecx`	`0x0`
`subl %edx,%eax`	`%eax`	`0xFD`

Problem 3.5 Solution: [Pg. 135]

This exercise gives you a chance to generate a little bit of assembly code. The solution code was generated by GCC. By loading parameter `n` in register `%ecx`, it can then use byte register `%cl` to specify the shift amount for the `sarl` instruction:

```
movl 12(%ebp),%ecx    Get n
movl 8(%ebp),%eax     Get x
sall $2,%eax          x <<= 2
sarl %cl,%eax         x >>= n
```

Problem 3.6 Solution: [Pg. 136]

This instruction is used to set register `%edx` to 0, exploiting the property that $x \hat{}\ x = 0$ for any x. It corresponds to the C statement `i = 0`.

This is an example of an assembly language *idiom*—a fragment of code that is often generated to fulfill a special purpose. Recognizing such idioms is one step in becoming proficient at reading assembly code.

Problem 3.7 Solution: [Pg. 140]

This example requires you to think about the different comparison and set instructions. A key point to note is that by casting the value on one side of a comparison to `unsigned`, the comparison is performed as if both sides are unsigned, due to implicit casting.

```
char ctest(int a, int b, int c)
{
  char t1 =         a <             b;
  char t2 =         b <  (unsigned)    a;
  char t3 = (short) c >= (short)    a;
  char t4 = (char) a != (char)    c;
  char t5 =         c >             b;
  char t6 =         a >             0;
  return t1 + t2 + t3 + t4 + t5 + t6;
}
```

Problem 3.8 Solution: [Pg. 144]

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `jbe` instruction has as target `0x8048d1c` + `0xda`. As the original disassembled code shows, this is `0x8048cf8`.

```
8048d1c:  76 da           jbe    8048cf8
8048d1e:  eb 24           jmp    8048d44
```

B. According to the annotation produced by the disassembler, the jump target is at absolute address `0x8048d44`. According to the byte encoding, this must be at an address `0x54` bytes beyond that of the `mov` instruction. Subtracting these gives address `0x8048cf0`, as confirmed by the disassembled code:

```
8048cee:  eb 54           jmp    8048d44
8048cf0:  c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

C. The target is at offset `000000cb` relative to `0x8048907` (the address of the `nop` instruction). Summing these gives address `0x80489d2`.

```
8048902:  e9 cb 00 00 00  jmp    80489d2
8048907:  90              nop
```

D. An indirect jump is denoted by instruction code `ff 25`. The address from which the jump target is to be read is encoded explicitly by the following 4 bytes. Since the machine is little endian, these are given in reverse order as `e0 a2 04 08`.

```
80483f0:  ff 25 e0 a2 04  jmp    *0x804a2e0
80483f5:  08
```

Problem 3.9 Solution: [Pg. 146]

Annotating assembly code and writing C code that mimics its control flow are good first steps in understanding assembly language programs. This problem gives you practice for an example with simple control flow. It also gives you a chance to examine the implementation of logical operations.

A. *code/asm/simple-if.c*

```
void cond(int a, int *p)
{
  if (p == 0)
    goto done;
  if (a <= 0)
    goto done;
  *p += a;
 done:
}
```

code/asm/simple-if.c

B. The first conditional branch is part of the implementation of the `||` expression. If the test for `p` being nonnull fails, the code will skip the test of `a > 0`.

Problem 3.10 Solution: [Pg. 148]

The code generated when compiling loops can be tricky to analyze, because the compiler can perform many different optimizations on loop code, and because it can be difficult to match program variables with registers. We start practicing this skill with a fairly simple loop.

A. The register usage can be determined by simply looking at how the arguments get fetched.

Register usage		
Register	Variable	Initially
`%esi`	`x`	`x`
`%ebx`	`y`	`y`
`%ecx`	`n`	`n`

B. The *body-statement* portion consists of lines 4 through 6 in the C code and lines 6 through 8 in the assembly code. The *test-expr* portion is on line 7 in the C code. In the assembly code, it is implemented by the instructions on lines 9 through 14, as well as by the branch condition on line 15.

C. The annotated code is as follows:

```
   Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
1    movl 8(%ebp),%esi       Put x in %esi
2    movl 12(%ebp),%ebx      Put y in %ebx
3    movl 16(%ebp),%ecx      Put n in %ecx
```

```
  .p2align 4,,7
.L6:                     loop:
  imull %ecx,%ebx          y *= n
  addl %ecx,%esi           x += n
  decl %ecx                n--
  testl %ecx,%ecx          Test n
  setg %al                 n > 0
  cmpl %ecx,%ebx           Compare y:n
  setl %dl                 y < n
  andl %edx,%eax           (n > 0) & (y < n)
  testb $1,%al             Test least significant bit
  jne .L6                   If != 0, goto loop
```

Note the somewhat strange implementation of the test expression. Apparently, the compiler recognizes that the two predicates `(n > 0)` and `(y < n)` can only evaluate to 0 or 1, and hence the branch condition need only test the least significant byte of their AND. The compiler could have been more clever and used the `testb` instruction to perform the AND operation.

Problem 3.11 Solution: [Pg. 151]

This problem offers another chance to practice deciphering loop code. The C compiler has done some interesting optimizations.

A. The register usage can be determined by looking at how the arguments get fetched, and how registers are initialized.

Register usage		
Register	Variable	Initially
`%eax`	`a`	`a`
`%ebx`	`b`	`b`
`%ecx`	`i`	0
`%edx`	`result`	`a`

B. The *test-expr* occurs on line 5 of the C code and on line 10 and the jump condition of line 11 in the assembly code. The *body-statement* occurs on lines 6 through 8 of the C code and on lines 7 through 9 of the assembly code. The compiler has detected that the initial test of the while loop will always be true, since `i` is initialized to 0, which is clearly less than 256.

C. The annotated code is as follows

```
1   movl 8(%ebp),%eax        Put a in %eax
2   movl 12(%ebp),%ebx       Put b in %ebx
3   xorl %ecx,%ecx           i = 0
4   movl %eax,%edx           result = a
5   .p2align 4,,7
  a in %eax, b in %ebx, i in %ecx, result in %edx
```

```
.L5:                        loop:
  addl %eax,%edx              result += a
  subl %ebx,%eax              a -= b
  addl %ebx,%ecx              i += b
  cmpl $255,%ecx              Compare i:255
  jle .L5                     If <= goto loop
  movl %edx,%eax              Set result as return value
```

D. The equivalent `goto` code is as follows

```
int loop_while_goto(int a, int b)
{
  int i = 0;
  int result = a;
 loop:
  result += a;
  a -= b;
  i += b;
  if (i <= 255)
    goto loop;
  return result;
}
```

Problem 3.12 Solution: [Pg. 155]

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces you to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops. It also demonstrates an optimization technique known as *code motion*, where a computation is moved out of a loop when it can be determined that its result will not change within the loop.

A. We can see that `result` must be in register `%eax`. It gets set to 0 initially and it is left in `%eax` at the end of the loop as a return value. We can see that `i` is held in register `%edx`, since this register is used as the basis for two conditional tests.

B. The instructions on lines 2 and 4 set `%edx` to `n-1`.

C. The tests on lines 5 and 12 require `i` to be nonnegative.

D. Variable `i` gets decremented by instruction 4.

E. Instructions 1, 6, and 7 cause `x*y` to be stored in register `%ecx`.

F. Here is the original code:

```
int loop(int x, int y, int n)
{
  int result = 0;
  int i;
  for (i = n-1; i >= 0; i = i-x) {
    result += y * x;
  }
  return result;
}
```

Problem 3.13 Solution: [Pg. 159]

This problem gives you a chance to reason about the control flow of a switch statement. Answering the questions requires you to combine information from several places in the assembly code:

1. Line 2 of the assembly code adds 2 to `x` to set the lower range of the cases to 0. That means that the minimum case label is -2.
2. Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 6. This implies that the maximum case label is $-2 + 6 = 4$.
3. In the jump table, we see that the second entry (case label -1) has the same destination (`.L10`) as the jump instruction on line 4, indicating the default case behavior. Thus, case label -1 is missing in the switch statement body.
4. In the jump table, we see that the fifth and sixth entries have the same destination. These correspond to case labels 2 and 3.

From this reasoning, we draw the following two conclusions:

A. The case labels in the switch statement body had values -2, 0, 1, 2, 3, and 4.

B. The case with destination `.L8` had labels 2 and 3.

Problem 3.14 Solution: [Pg. 162]

This is another example of an assembly code idiom. At first it seems quite peculiar—a `call` instruction with no matching `ret`. Then we realize that it is not really a procedure call after all.

A. `%eax` is set to the address of the `popl` instruction.

B. This is not a true subroutine call, since the control follows the same ordering as the instructions and the return address is popped from the stack.

C. This is the only way in IA32 to get the value of the program counter into an integer register.

Problem 3.15 Solution: [Pg. 164]

This problem makes concrete the discussion of register usage conventions. Registers `%edi`, `%esi`, and `%ebx` are callee save. The procedure must save them on the stack before altering their values and restore them before returning. The other three registers are caller save. They can be altered without affecting the behavior of the caller.

Problem 3.16 Solution: [Pg. 166]

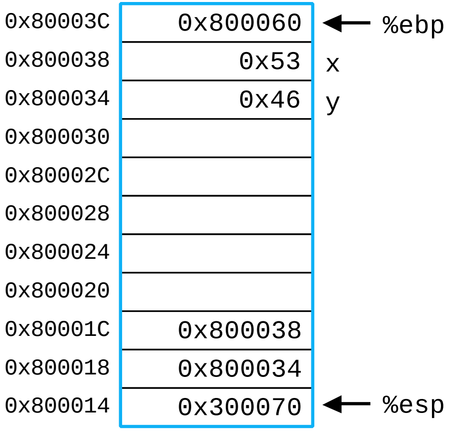
Being able to reason about how functions use the stack is a critical part of understanding compiler-generated code. As this example illustrates, the compiler allocates a significant amount of space that never gets used.

A. We started with `%esp` having value `0x800040`. Line 2 decrements this by 4, giving `0x80003C`, and this becomes the new value of `%ebp`.

B. We can see how the two `leal` instructions compute the arguments to pass to `scanf`. Since arguments are pushed in reverse order, we can see that `x` is at offset -4 relative to `%ebp` and `y` is at offset -8. The addresses are therefore `0x800038` and `0x800034`.

C. Starting with the original value of `0x800040`, line 2 decremented the stack pointer by 4. Line 4 decremented it by 24, and line 5 decremented it by 4. The three pushes decremented it by 12, giving an overall change of 44. Thus, after line 10 `%esp` equals `0x800014`.

D. The stack frame has the following structure and contents:



E. Byte addresses `0x800020` through `0x800033` are unused.

Problem 3.17 Solution: [Pg. 172]

This exercise tests your understanding of data sizes and array indexing. Observe that a pointer of any kind is 4 bytes long. The GCC implementation of `long double` uses 12 bytes to store each value, even though the actual format requires only 10 bytes.

Array	Element size	Total size	Start address	Element i
S	2	28	$x_{\mathtt{S}}$	$x_{\mathtt{S}} + 2i$
T	4	12	$x_{\mathtt{T}}$	$x_{\mathtt{T}} + 4i$
U	4	24	$x_{\mathtt{U}}$	$x_{\mathtt{U}} + 4i$
V	12	96	$x_{\mathtt{V}}$	$x_{\mathtt{V}} + 12i$
W	4	16	$x_{\mathtt{W}}$	$x_{\mathtt{W}} + 4i$

Problem 3.18 Solution: [Pg. 173]

This problem is a variant of the one shown for integer array `E`. It is important to understand the difference between a pointer and the object being pointed to. Since data type `short` requires two bytes, all of the array indices are scaled by a factor of two. Rather than using `movl`, as before, we now use `movw`.

Expression	Type	Value	Assembly
`S+1`	`short *`	$x_{\mathtt{S}} + 2$	`leal 2(%edx),%eax`
`S[3]`	`short`	$\mathtt{M}[x_{\mathtt{S}} + 6]$	`movw 6(%edx),%ax`
`&S[i]`	`short *`	$x_{\mathtt{S}} + 2i$	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	$\mathtt{M}[x_{\mathtt{S}} + 8i + 2]$	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short *`	$x_{\mathtt{S}} + 2i - 10$	`leal -10(%edx,%ecx,2),%eax`

Problem 3.19 Solution: [Pg. 176]

This problem requires you to work through the scaling operations to determine the address computations, and to apply the formula for row-major indexing. The first step is to annotate the assembly to determine how the address references are computed:

```
movl 8(%ebp),%ecx                Get i
movl 12(%ebp),%eax               Get j
leal 0(,%eax,4),%ebx             4*j
leal 0(,%ecx,8),%edx             8*i
subl %ecx,%edx                   7*i
addl %ebx,%eax                   5*j
sall $2,%eax                     20*j
movl mat2(%eax,%ecx,4),%eax      mat2[(20*j + 4*i)/4]
addl mat1(%ebx,%edx,4),%eax      + mat1[(4*j + 28*i)/4]
```

From this we can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $\mathtt{M} = 5$ and $\mathtt{N} = 7$.

Problem 3.20 Solution: [Pg. 177]

This exercise requires you to study assembly code to understand how it has been optimized. This is an important skill for improving program performance. By adjusting your source code, you can have an effect on the efficiency of the generated machine code.

The following is an optimized version of the C code:

```
/* Set all diagonal elements to val */
void fix_set_diag_opt(fix_matrix A, int val)
{
  int *Aptr = &A[0][0] + 255;
  int cnt = N-1;
  do {
    *Aptr = val;
    Aptr -= (N+1);
    cnt--;
  } while (cnt >= 0);
}
```

The relation to the assembly code can be seen via the following annotations:

```
  movl 12(%ebp),%edx    Get val
  movl 8(%ebp),%eax     Get A
  movl $15,%ecx         i = 0
  addl $1020,%eax       Aptr = &A[0][0] + 1020/4
  .p2align 4,,7
.L50:                  loop:
  movl %edx,(%eax)      *Aptr = val
  addl $-68,%eax        Aptr -= 68/4
  decl %ecx             i--
  jns .L50              if i >= 0 goto loop
```

Observe how the assembly code program starts at the end of the array and works backward. It decrements the pointer by 68 (= $17 \cdot 4$), since array elements `A[i-1][i-1]` and `A[i][i]` are spaced `N+1` elements apart.

Problem 3.21 Solution: [Pg. 183]

This problem gets you to think about structure layout and the code used to access structure fields. The structure declaration is a variant of the example shown in the text. It shows that nested structures are allocated by embedding the inner structures within the outer ones.

A. The layout of the structure is as follows:

Offset	0	4	8	12
Contents	`p`	`s.x`	`s.y`	`next`

B. It uses 16 bytes.

C. As always, we start by annotating the assembly code:

```
movl 8(%ebp),%eax       Get sp
movl 8(%eax),%edx       Get sp->s.y
movl %edx,4(%eax)       Copy to sp->s.x
leal 4(%eax),%edx       Get &(sp->s.x)
movl %edx,(%eax)        Copy to sp->p
movl %eax,12(%eax)      sp->next = p
```

From this, we can generate C code as follows:

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

Problem 3.22 Solution: [Pg. 187]

This is a very tricky problem. It raises the need for puzzle-solving skills as part of reverse engineering to new heights. It shows very clearly that unions are simply a way to associate multiple names (and types) with a single storage location.

A. The layout of the union is shown in the table that follows. As the table illustrates, the union can have either its "`e1`" interpretation (having fields `e1.p` and `e1.y`), or it can have its "`e2`" interpretation (having fields `e2.x` and `e2.next`).

Offset	0	4
Contents	`e1.p`	`e1.y`
	`e2.x`	`e2.next`

B. It uses 8 bytes.

C. As always, we start by annotating the assembly code. In our annotations, we show multiple possible interpretations for some of the instructions, and then indicate which interpretation later gets discarded. For example, line 2 could be interpreted as either getting element `e1.y` or `e2.next`. In line 3, we see that the value gets used in an indirect memory reference, for which only the second interpretation of line 2 is possible.

```
1   movl 8(%ebp),%eax        Get up
2   movl 4(%eax),%edx        up->e1.y (no) or up->e2.next
3   movl (%edx),%ecx         up->e2.next->e1.p or up->e2.next->e2.x (no)
4   movl (%eax),%eax         up->e1.p (no) or up->e2.x
5   movl (%ecx),%ecx         *(up->e2.next->e1.p)
6   subl %eax,%ecx           *(up->e2.next->e1.p) - up->e2.x
7   movl %ecx,4(%edx)        Store in up->e2.next->e1.y
```

From this, we can generate C code as follows:

```
void proc (union ele *up)
{
    up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
}
```

Problem 3.23 Solution: [Pg. 190]

Understanding structure layout and alignment is very important for understanding how much storage different data structures require and for understanding the code generated by the compiler for accessing structures. This problem lets you work out the details of some example structures.

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

i	c	j	d	Total	Alignment
0	4	8	12	16	4

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

i	c	d	j	Total	Alignment
0	4	5	8	12	4

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

w	c	Total	Alignment
0	6	10	2

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

w	c	Total	Alignment
0	8	20	4

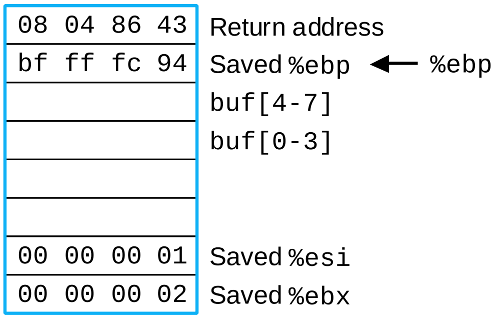
E. `struct P3 { struct P1 a[2]; struct P2 *p };`

a	p	Total	Alignment
0	32	36	4

Problem 3.24 Solution: [Pg. 197]

This problem covers a wide range of topics, such as stack frames, string representations, ASCII code, and byte ordering. It demonstrates the dangers of out-of-bounds memory references and the basic ideas behind buffer overflow.

A. Stack at line 7.



B. Stack after line 10 (showing only words that are modified).

`08 04 86 00`	Return address	
`31 30 39 38`	Saved `%ebp`	← `%ebp`
`37 36 35 34`	`buf[4-7]`	
`33 32 31 30`	`buf[0-3]`	

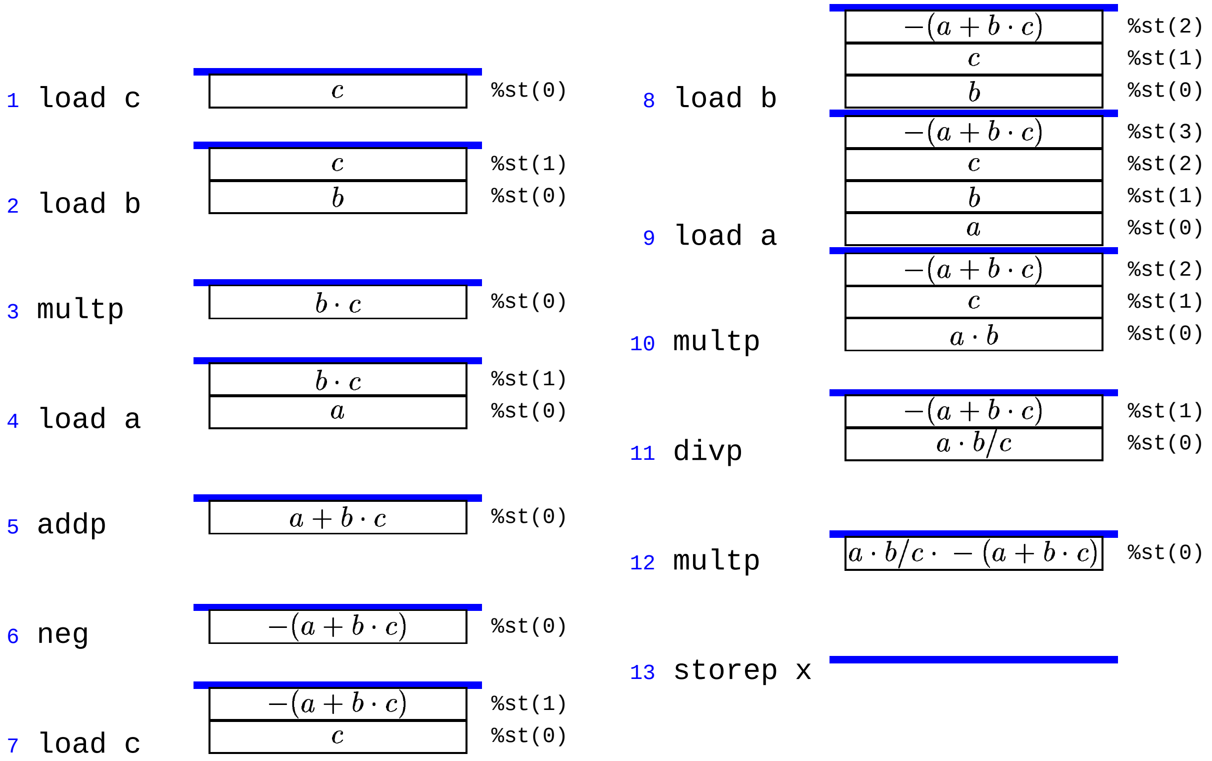
C. The program is attempting to return to address `0x08048600`. The low-order byte was overwritten by the terminating null character.

D. The saved value of register `%ebp` was changed to `0x31303938`, and this will be loaded into the register before `getline` returns. The other saved registers are not affected, since they are saved on the stack at lower addresses than `buf`.

E. The call to `malloc` should have had `strlen(buf)+1` as its argument, and it should also check that the returned value is non-null.

Problem 3.25 Solution: [Pg. 203]

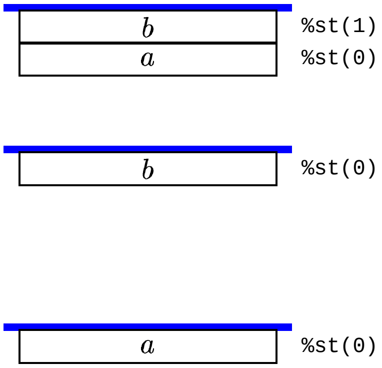
This problem gives you a chance to try out the recursive procedure described in Section 3.14.2.



Problem 3.26 Solution: [Pg. 206]

The following code is similar to that generated by the compiler for selecting between two values based on the outcome of a test:

```
    test %eax,%eax
    jne L11
    fstp %st(0)
    jmp L9
L11:
    fstp %st(1)
L9:
```

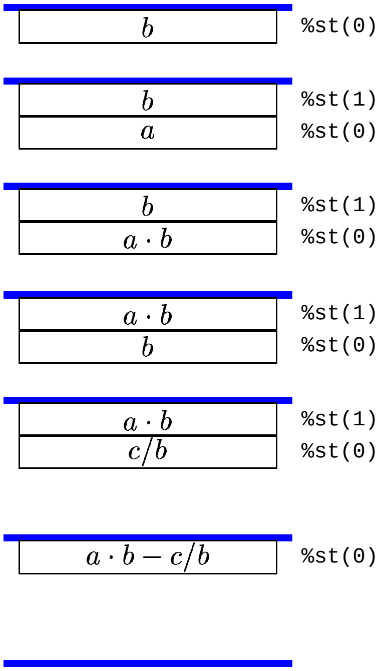


The resulting top of stack value is `x ? a : b`.

Problem 3.27 Solution: [Pg. 209]

Floating-point code is tricky, with its different conventions about popping operands, the order of the arguments, etc. This problem gives you a chance to work through some specific cases in complete detail.

```
fldl b
fldl a
fmul %st(1),%st
fxch
fdivrl c
fsubrp
fstp x
```



This code computes the expression `x = a*b - c/b`.

Problem 3.28 Solution: [Pg. 210]

This problem requires you to think about the different operand types and sizes in floating-point code.

code/asm/fpfunct2-ans.c

```
double funct2(int a, double x, float b, float i)
{
    return a/(x+b) - (i+1);
}
```

code/asm/fpfunct2-ans.c

Problem 3.29 Solution: [Pg. 212]

Insert the following code between lines 4 and 5:

```
  cmpb $1,%ah       Test if comparison outcome is <
```

Problem 3.30 Solution: [Pg. 217]

```
int ok_smul(int x, int y, int *dest)
{
    long long prod = (long long) x * y;
    int trunc = (int) prod;

    *dest = trunc;
    return (trunc == prod);
}
```