3

Machine-Level Representation of Programs

3.1 A Historical Perspective 125

3.2 Program Encodings 128

3.3 Data Formats 135

3.4 Accessing Information 136

3.5 Arithmetic and Logical Operations 143

3.6 Control 148

3.7 Procedures 170

3.8 Array Allocation and Access 180

3.9 Heterogeneous Data Structures 191

3.10 Alignment 198

3.11 Putting it Together: Understanding Pointers 201

3.12 Life in the Real World: Using the GDB Debugger 204

3.13 Out-of-Bounds Memory References and Buffer Overflow 206

3.14 *Floating-Point Code 211

3.15 *Embedding Assembly Code in C Programs 223

3.16 Summary 230

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.

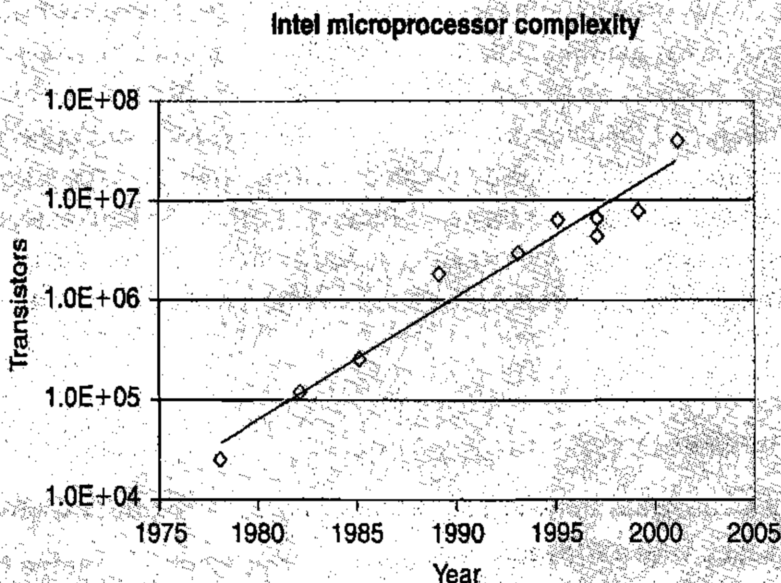
Pentium 4: (2001, 42 M transistors). Added 8-byte integer and floating-point formats to the vector instructions, along with 144 new instructions for these formats. Intel shifted away from Roman numerals in their numbering convention.

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 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.

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.

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:

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

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

```
unix> gcc -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 bibliographic 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.

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
1   00000000 <sum>:
    Offset  Bytes                       Equivalent assembly language
2      0:   55                              push   %ebp
3      1:   89 e5                           mov    %esp,%ebp
4      3:   8b 45 0c                        mov    0xc(%ebp),%eax
5      6:   03 45 08                        add    0x8(%ebp),%eax
6      9:   01 05 00 00 00 00               add    %eax,0x0
7      f:   89 ec                           mov    %ebp,%esp
8     11:   5d                              pop    %ebp
9     12:   c3                              ret
10    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 6 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:

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

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
1     080483b4 <sum>:
2      80483b4:  55                     push   %ebp
3      80483b5:  89 e5                  mov    %esp,%ebp
4      80483b7:  8b 45 0c               mov    0xc(%ebp),%eax
5      80483ba:  03 45 08               add    0x8(%ebp),%eax
6      80483bd:  01 05 64 94 04 08      add    %eax,0x8049464
7      80483c3:  89 ec                  mov    %ebp,%esp
8      80483c5:  5d                     pop    %ebp
9      80483c6:  c3                     ret
10     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 6 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:

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

When GCC is run with 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:

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

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

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.

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.

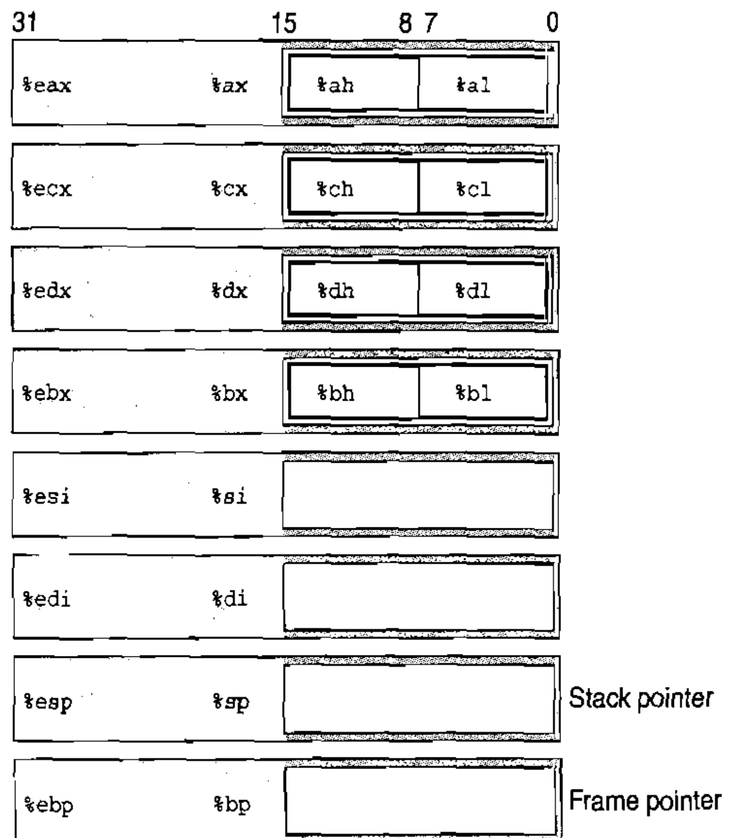
As the table indicates, every operation in GAS has a single-character suffix denoting the size of the operand. For example, the `mov` (move data) instruction has three variants: `movb` (move byte), `movw` (move word), and `movl` (move double word). The suffix '`l`' is used for double words, since on 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 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 feature stems from IA32's evolutionary heritage as a 16-bit microprocessor.

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.



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 E_a to denote an arbitrary register a, and indicate its value with the reference $R[E_a]$, 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 $M_b[Addr]$ to denote a reference to the b-byte value stored in memory starting at address $Addr$. To simplify things, we will generally drop the subscript b.

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

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

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

general form where some of the components are omitted. As we will see, the more complex addressing modes are useful when referencing array and structure elements.

Practice Problem 3.1

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

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

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

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

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

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:

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	$D \leftarrow$ SignExtend(S)	Move sign-extended byte
movzbl	S, D	$D \leftarrow$ ZeroExtend(S)	Move zero-extended byte
pushl	S	R[%esp] $\leftarrow$ R[%esp] $- 4$; M[R[%esp]] $\leftarrow S$	Push
popl	D	$D \leftarrow$ M[R[%esp]]; R[%esp] $\leftarrow$ R[%esp] $+ 4$	Pop

Figure 3.4 Data movement instructions.

```
1    movl $0x4050,%eax         Immediate--Register
2    movl %ebp,%esp            Register--Register
3    movl (%edi,%ecx),%eax     Memory--Register
4    movl $-17,(%esp)          Immediate--Memory
5    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
1    movb %dh,%al                     %eax = 9876548D
2    movsbl %dh,%eax                  %eax = FFFFFF8D
3    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.

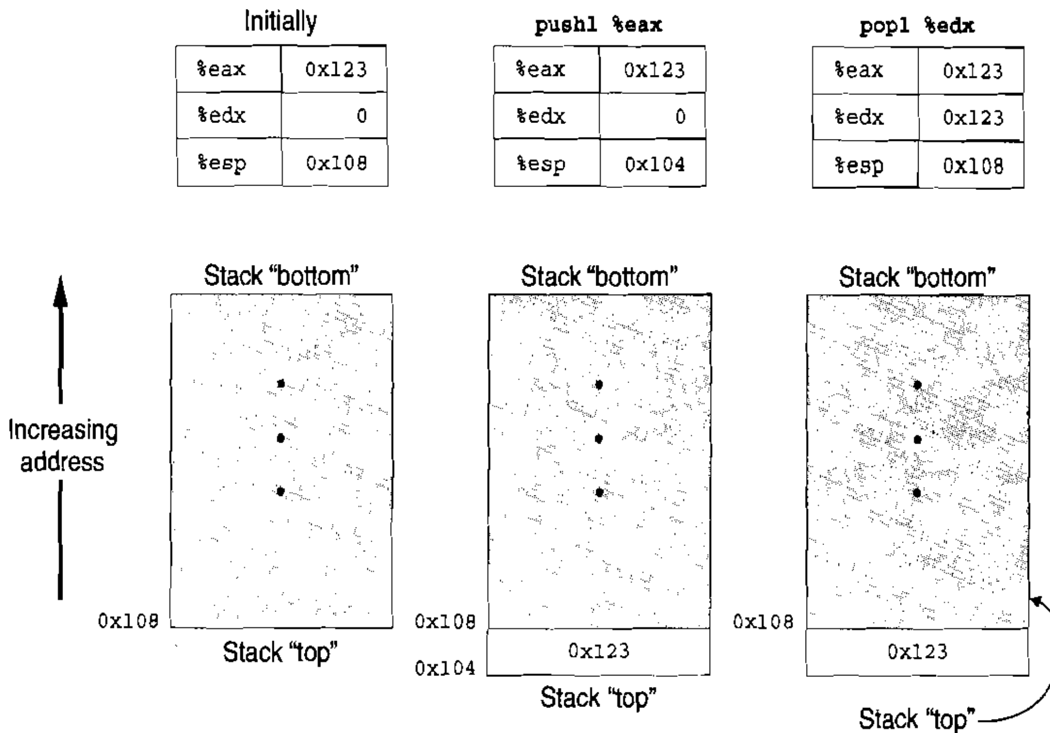


Figure 3.5 Illustration of stack operation. By convention, we draw stacks upside-down, so that the "top" of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register `%esp`) and storing to memory, while popping involves reading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

The 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)
```

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`.

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
```

code/asm/exchange.c

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

code/asm/exchange.c

(a) C code

```
1   movl 8(%ebp),%eax    Get xp
2   movl 12(%ebp),%edx   Get y
3   movl (%eax),%ecx     Get x at *xp
4   movl %edx,(%eax)     Store y at *xp
5   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.

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.

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:

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

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.

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.

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.

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

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

3.5.4 Discussion

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 $(z+2z) = 3z$, 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 76).

code/asm/arith.c

```
1  int arith(int x,
2            int y,
3            int z)
4  {
5      int t1 = x+y;
6      int t2 = z*48;
7      int t3 = t1 & 0xFFFF;
8      int t4 = t2 * t3;
9
10     return t4;
11 }
```

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.

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.

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 $*^{u}_{32}$ and $*^{t}_{32}$ described in Sections 2.3.4 and 2.3.5. Recall that when truncating the product to 32 bits, both unsigned multiply and two's complement multiply have the same bit-level behavior. IA32 also provides two different "one-operand" multiply instructions to compute the full 64-bit product of two 32-bit values—one for unsigned (mull), and one for two's complement

Instruction		Effect	Description
imull	S	R[%edx]:R[%eax] ← $S \times$ R[%eax]	Signed full multiply
mull	S	R[%edx]:R[%eax] ← $S \times$ R[%eax]	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.

(`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
1       movl 8(%ebp),%eax          Put x in %eax
2       imull 12(%ebp)             Multiply by y
3       pushl %edx                 Push high-order 32 bits
4       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`.

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
1       movl 8(%ebp),%eax          Put x in %eax
2       cltd                       Sign extend into %edx
3       idivl 12(%ebp)             Divide by y
4       pushl %eax                 Push x / y
5       pushl %edx                 Push x % y
```

The `divl` instruction 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

[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.

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.

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

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

The `movzbl` instruction 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") 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.

```
1   char ctest(int a, int b, int c)
2   {
3     char t1 =          a __          b;
4     char t2 =          b __ (     )  a;
5     char t3 = (     ) c __ (     )  a;
6     char t4 = (     ) a __ (     )  c;
7     char t5 =          c __          b;
8     char t6 =          a __          0;
9     return t1 + t2 + t3 + t4 + t5 + t6;
10  }
```

For the original C code, GCC generates the following assembly code

```
1     movl 8(%ebp),%ecx          Get a
2     movl 12(%ebp),%esi         Get b
3     cmpl %esi,%ecx             Compare a:b
4     setl %al                   Compute t1
5     cmpl %ecx,%esi             Compare b:a
6     setb -1(%ebp)              Compute t2
7     cmpw %cx,16(%ebp)          Compare c:a
8     setge -2(%ebp)             Compute t3
9     movb %cl,%dl
10    cmpb 16(%ebp),%dl          Compare a:c
11    setne %bl                  Compute t4
12    cmpl %esi,16(%ebp)         Compare c:b
13    setg -3(%ebp)              Compute t5
14    testl %ecx,%ecx            Test a
15    setg %dl                   Compute t6
16    addb -1(%ebp),%al          Add t2 to t1
17    addb -2(%ebp),%al          Add t3 to t1
18    addb %bl,%al               Add t4 to t1
19    addb -3(%ebp),%al          Add t5 to t1
20    addb %dl,%al               Add t6 to t1
21    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. (See Figure 3.11.) These jump destinations are generally indicated by a *label*. Consider the following assembly code sequence:

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

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

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`"

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.

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.

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

code/asm/abs.c

```
1  int absdiff(int x, int y)
2  {
3      if (x < y)
4          return y - x;
5      else
6          return x - y;
7  }
```

code/asm/abs.c

(a) Original C code.

code/asm/abs.c

```
1   int gotodiff(int x, int y)
2   {
3       int rval;
4
5       if (x < y)
6           goto less;
7       rval = x - y;
8       goto done;
9    less:
10      rval = y - x;
11   done:
12      return rval;
13  }
```

code/asm/abs.c

(b) Equivalent goto version of (a).

```
1      movl 8(%ebp),%edx       Get x
2      movl 12(%ebp),%eax      Get y
3      cmpl %eax,%edx          Compare x:y
4      jl .L3                  If <, goto less
5      subl %eax,%edx          Compute x-y
6      movl %edx,%eax          Set as return value
7      jmp .L5                 Goto done
8    .L3:                    less:
9      subl %edx,%eax            Compute y-x as return value
10   .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.

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

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

code/asm/simple-if.c

GCC generates the following assembly code:

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

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 3.12(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:

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}, n \geq 3$$

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.

code/asm/fib.c

```
1  int fib_dw(int n)
2  {
3      int i = 0;
4      int val = 0;
5      int nval = 1;
6
7      do {
8          int t = val + nval;
9          val = nval;
10         nval = t;
11         i++;
12     } while (i < n);
13
14     return val;
15 }
```

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	–

```
1  .L6:                              loop:
2    leal (%edx,%ebx),%eax             Compute t = val + nval
3    movl %edx,%ebx                    copy nval to val
4    movl %eax,%edx                    Copy t to nval
5    incl %ecx                         Increment i
6    cmpl %esi,%ecx                    Compare i:n
7    jl .L6                            If less, goto loop
8    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.

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

```
1  int dw_loop(int x, int y, int n)
2  {
3    do {
4      x += n;
5      y *= n;
6      n--;
7    } while ((n > 0) & (y < n)); /* Note use of bitwise '&' */
8    return x;
9  }
```

GCC generates the following assembly code:

```
     Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
1      movl 8(%ebp),%esi
2      movl 12(%ebp),%ebx
3      movl 16(%ebp),%ecx
4      .p2align 4,,7          Inserted to optimize cache performance
5    .L6:
6      imull %ecx,%ebx
7      addl %ecx,%esi
8      decl %ecx
9      testl %ecx,%ecx
10     setg %al
11     cmpl %ecx,%ebx
12     setl %dl
13     andl %edx,%eax
14     testb $1,%al
15     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

code/asm/fib.c

```
1   int fib_w(int n)
2   {
3       int i = 1;
4       int val = 1;
5       int nval = 1;
6
7       while (i < n) {
8           int t = val+nval;
9           val = nval;
10          nval = t;
11          i++;
12      }
13
14      return val;
15  }
```

code/asm/fib.c

(a) C code.

code/asm/fib.c

```
1   int fib_w_goto(int n)
2   {
3       int val = 1;
4       int nval = 1;
5       int nmi, t;
6
7       if (val >= n)
8           goto done;
9       nmi = n-1;
10
11   loop:
12      t = val+nval;
13      val = nval;
14      nval = t;
15      nmi--;
16      if (nmi)
17          goto loop;
18
19   done:
20      return val;
21  }
```

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

```
1     movl 8(%ebp),%eax        Get n
2     movl $1,%ebx             Set val to 1
3     movl $1,%ecx             Set nval to 1
4     cmpl %eax,%ebx           Compare val:n
5     jge .L9                  If >= goto done
6     leal -1(%eax),%edx       nmi = n-1
7   .L10:                     loop:
8     leal (%ecx,%ebx),%eax    Compute t = nval+val
9     movl %ecx,%ebx           Set val to nval
10    movl %eax,%ecx           Set nval to t
11    decl %edx                Decrement nmi
12    jnz .L10                 if != 0, goto loop
13  .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.

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 following C code:

```
1  int loop_while(int a, int b)
2  {
3    int i = 0;
4    int result = a;
5    while (i < 256) {
6      result += a;
7      a -= b;
8      i += b;
9    }
10   return result;
11 }
```

GCC generates the following assembly code:

```
     Initially a and b are at offsets 8 and 12 from %ebp
1      movl 8(%ebp),%eax
2      movl 12(%ebp),%ebx
3      xorl %ecx,%ecx
4      movl %eax,%edx
5      .p2align 4,,7
6    .L5:
7      addl %eax,%edx
8      subl %ebx,%eax
9      addl %ebx,%ecx
10     cmpl $255,%ecx
11     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

```
1  int fib_f(int n)
2  {
3      int i;
4      int val = 1;
5      int nval = 1;
6
7      for (i = 1; i < n; i++) {
8          int t = val+nval;
9          val = nval;
10         nval = t;
11     }
12
13     return val;
14 }
```

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
1      movl 8(%ebp),%ebx
2      movl 16(%ebp),%edx
3      xorl %eax,%eax
4      decl %edx
5      js .L4
6      movl %ebx,%ecx
7      imull 12(%ebp),%ecx
8      .p2align 4,,7     Inserted to optimize cache performance
9    .L6:
10     addl %ecx,%eax
11     subl %ebx,%edx
12     jns .L6
13   .L4:
```

The preceding code was generated by compiling C code that had the following overall form:

```
1  int loop(int x, int y, int n)
2  {
3    int result = 0;
4    int i;
5    for (i = ____; i ____ ; i = ___ ) {
6      result += _____ ;
7    }
8    return result;
9  }
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%eax`. 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 multiway branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, 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.

code/asm/switch.c

```
1  int switch_eg(int x)
2  {
3      int result = x;
4
5      switch (x) {
6
7      case 100:
8          result *= 13;
9          break;
10
11     case 102:
12         result += 10;
13         /* Fall through */
14
15     case 103:
16         result += 11;
17         break;
18
19     case 104:
20     case 106:
21         result *= result;
22         break;
23
24     default:
25         result = 0;
26     }
27
28     return result;
29 }
```

code/asm/switch.c

(a) Switch statement.

code/asm/switch.c

```
1  /* Next line is not legal C */
2  code *jt[7] = {
3      loc_A, loc_def, loc_B, loc_C,
4      loc_D, loc_def, loc_D
5  };
6
7  int switch_eg_impl(int x)
8  {
9      unsigned xi = x - 100;
10     int result = x;
11
12     if (xi > 6)
13         goto loc_def;
14
15     /* Next goto is not legal C */
16     goto jt[xi];
17
18  loc_A:       /* Case 100 */
19     result *= 13;
20     goto done;
21
22  loc_B:       /* Case 102 */
23     result += 10;
24     /* Fall through */
25
26  loc_C:     /* Case 103 */
27     result += 11;
28     goto done;
29
30  loc_D:     /* Cases 104, 106 */
31     result *= result;
32     goto done;
33
34  loc_def:  /* Default case */
35     result = 0;
36
37  done:
38     return result;
39 }
```

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.

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

```
      Set up the jump table access
1     leal -100(%edx),%eax          Compute xi = x-100
2     cmpl $6,%eax                  Compare xi:6
3     ja .L9                        if >, goto loc_def
4     jmp *.L10(,%eax,4)            Goto jt[xi]

      Case 100
5   .L4:                          loc_A:
6     leal (%edx,%edx,2),%eax       Compute 3*x
7     leal (%edx,%eax,4),%edx       Compute x+4*3*x
8     jmp .L3                       Goto done

      Case 102
9   .L5:                          loc_B:
10    addl $10,%edx                 result += 10, Fall through

      Case 103
11  .L6:                          loc_C:
12    addl $11,%edx                 result += 11
13    jmp .L3                       Goto done

      Cases 104, 106
14  .L8:                          loc_D:
15    imull %edx,%edx               result *= result
16    jmp .L3                       Goto done

      Default case
17  .L9:                          loc_def:
18    xorl %edx,%edx                result = 0

      Return result
19  .L3:                          done:
20    movl %edx,%eax                Set result as return value
```

Figure 3.16 Assembly code for switch statement example in Figure 3.15.

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:

```
1    .section .rodata
2      .align 4        Align address to multiple of 4
3    .L10:
4      .long .L4       Case 100:  loc_A
5      .long .L9       Case 101:  loc_def
6      .long .L5       Case 102:  loc_B
7      .long .L6       Case 103:  loc_C
8      .long .L8       Case 104:  loc_D
9      .long .L9       Case 105:  loc_def
10     .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

```
1   movl 8(%ebp),%eax     Retrieve x
2   addl $2,%eax
3   cmpl $6,%eax
4   ja .L10
5   jmp *.L11(,%eax,4)
```

Jump table for switch2

```
1   .L11:
2     .long .L4
3     .long .L10
4     .long .L5
5     .long .L6
6     .long .L8
7     .long .L8
8     .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.