Chapter 3

Machine-Level Representation of Programs

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

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

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

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

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

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

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

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

3.1 A Historical Perspective

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

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

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

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

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

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

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

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

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

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

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

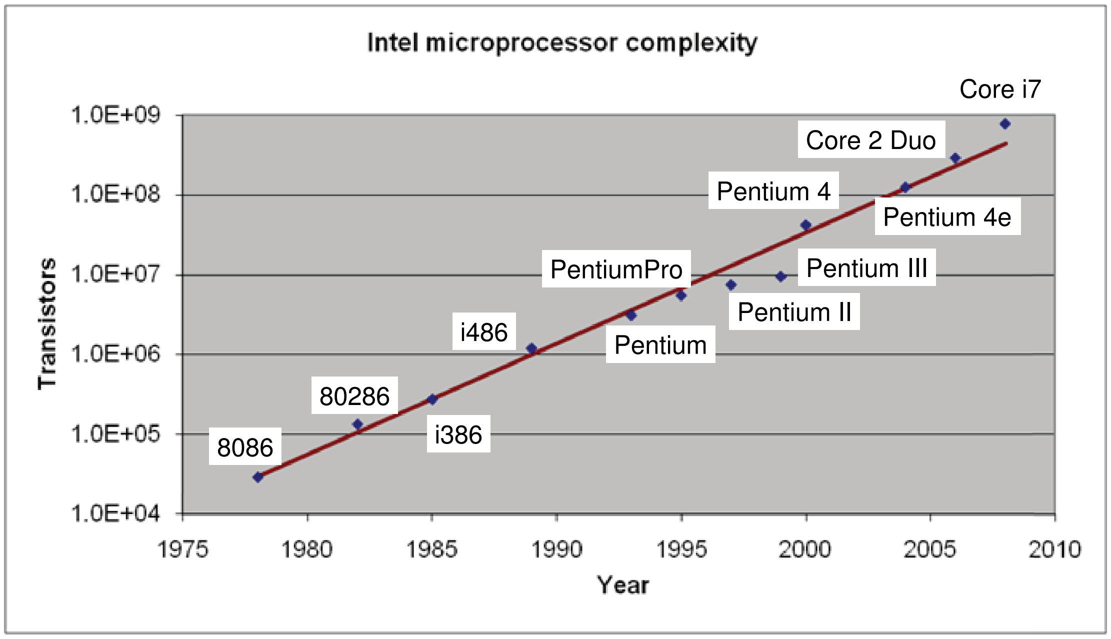
Pentium 4E: (2004, 125 M transistors). Added *hyperthreading*, a method to run two programs simultaneously on a single processor, as well as EM64T, Intel’s implementation of a 64-bit extension to IA32 developed by Advanced Micro Devices (AMD), which we refer to as x86-64.

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

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

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

Aside: Moore's Law.



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

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

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

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

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

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

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

3.2 Program Encodings

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

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

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

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

3.2.1 Machine-Level Code

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

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes. The assembly-code representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of machine code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

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

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

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

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

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

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

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

3.2.2 Code Examples

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

```
int accum = 0;

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

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

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

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

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

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

Each indented line in the above code corresponds to a single machine instruction. For example, the `pushl` instruction indicates that the contents of register `%ebp` should be pushed onto the program stack. All information about local variable names or data types has been stripped away. We still see a reference to the global variable `accum`, since the compiler has not yet determined where in memory this variable will be stored.

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

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

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

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

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

Aside: How do I find the byte representation of a program?
To generate these bytes, we used a *disassembler* (to be described shortly) to determine that the code for `sum` is 17 bytes long. Then we ran the GNU debugging tool GDB on file `code.o` and gave it the command

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
     Disassembly of function sum in executable file prog
08048394 <sum>:
   Offset   Bytes                          Equivalent assembly language
  8048394:  55                                 push   %ebp
  8048395:  89 e5                              mov    %esp,%ebp
  8048397:  8b 45 0c                           mov    0xc(%ebp),%eax
  804839a:  03 45 08                           add    0x8(%ebp),%eax
  804839d:  01 05 18 a0 04 08                  add    %eax,0x804a018
  80483a3:  5d                                 pop    %ebp
  80483a4:  c3                                 ret
```

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

3.2.3 Notes on Formatting

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

```
int simple(int *xp, int y)
{
  int t = *xp + y;
  *xp = t;
  return t;
}
```

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

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

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

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

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

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

Aside: ATT versus Intel assembly-code formats

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

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

This gives the following assembly code:

Assembly code for simple in Intel format

```
1 simple:
2   push    ebp
3   mov     ebp, esp
4   mov     edx, DWORD PTR [ebp+8]
5   mov     eax, DWORD PTR [ebp+12]
6   add     eax, DWORD PTR [edx]
```

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

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

```
mov     DWORD PTR [edx], eax
pop     ebp
ret
```

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

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

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

3.3 Data Formats

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

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

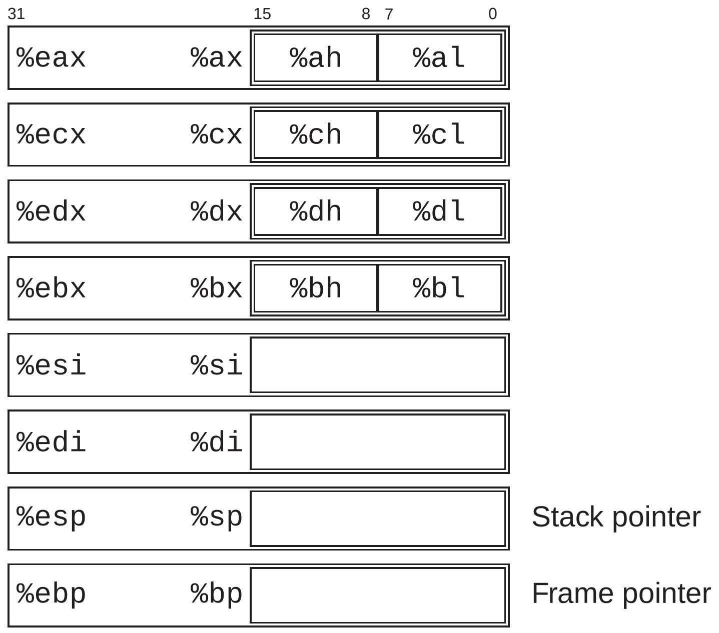


Figure 3.2: **IA32 integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The two low-order bytes of the first four registers can be accessed independently.

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

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

3.4 Accessing Information

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

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

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

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 and is also used when operating on integers with size designator `short`.

3.4.1 Operand Specifiers

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. IA32 supports a number of operand forms (see Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. In ATT-format assembly code, these are written with a '`$`' followed by an integer using standard C notation, for example, `$-577` or `$0x1F`. Any value that fits into a 32-bit word can be used, although the assembler will use 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, one of the eight 16-bit registers (e.g., `%ax`) for a word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In Figure

3.3, we use the notation $\mathtt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $\mathsf{R}[\mathtt{E}_a]$, viewing the set of registers as an array R indexed by register identifiers.

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

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

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 copy data from one location to another. The generality of the operand notation allows a simple data movement instruction to perform what in many machines would require a number of instructions. Figure 3.4 lists the important data movement instructions. As can be seen, we group the many different instructions into *instruction classes*, where the instructions in

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

Figure 3.4: **Data movement instructions.**

a class perform the same operation, but with different operand sizes. For example the MOV class consists of three instructions: `movb`, `movw`, and `movl`. All three of these instructions perform the same operation; they differ only in that they operate on data of size 1, 2, and 4 bytes, respectively.

The instructions in the MOV class copy their source values to their destinations. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination. Referring to Figure 3.2, the register operands for these instructions can be any of the 8 32-bit registers (`%eax`–`%ebp`) for `movl`, any of the 8 16-bit registers (`%ax`–`%bp`) for `movw`, and any of the single-byte register elements (`%ah`–`%bh`, `%al`–`%bl`) for `movb`. The following MOV instruction examples show the five possible combinations of source and destination types. Recall that the source operand comes first and the destination second:

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

Both the MOVS and the MOVZ instruction classes serve to copy a smaller amount of source data to a larger data location, filling in the upper bits by either sign expansion (MOVS) or by zero expansion (MOVZ). With sign expansion, the upper bits of the destination are filled in with copies of the most significant bit of the source value. With zero expansion, the upper bits are filled with zeros. As can be seen, there are three