Chapter 3

Machine-Level Representation of Programs

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

Even though optimizing compilers are available, being able to read and understand assembly code is an important skill for serious programmers. By invoking the compiler with appropriate flags, the compiler will generate a file showing its output in assembly code. Assembly code is very close to the actual machine code that computers execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter 5, programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the run-time behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package, as covered in Chapter 13, it is important to know what type of storage is used to hold the different program variables. This information is visible at the assembly code level. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by optimizing compilers.

In this chapter, we will learn the details of a particular assembly language and see how C programs get compiled into this form of machine code. Reading the assembly code generated by a compiler involves a different set of skills than writing assembly code by hand. We must understand the transformations typical compilers make in converting the constructs of C into machine code. Relative to the computations expressed in the C code, optimizing compilers can rearrange execution order, eliminate unneeded computations, replace slow operations such as multiplication by shifts and adds, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can often be a challenge—much like putting together a puzzle having a slightly different design than the picture on the box. It is a form of *reverse engineering*—trying to understand the process by which a system was created by studying the system and working backward. In this case, the system is a machine-generated, assembly-language program, rather than something designed by a human. This simplifies the task of reverse engineering, because the generated code follows fairly regular patterns, and we can run experiments, having the compiler generate code for many different programs. In our presentation, we give many examples and provide a number of exercises illustrating different aspects of assembly language and compilers. This is a subject matter where mastering the details is a prerequisite to understanding the deeper and more fundamental concepts. Spending time studying the examples and working through the exercises will be well worthwhile.

We give a brief history of the Intel architecture. Intel processors have grown from rather primitive 16-bit processors in 1978 to the mainstream machines for today's desktop computers. The architecture has grown correspondingly with new features added and the 16-bit architecture transformed to support 32-bit data and addresses. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by GCC and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

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

We then move into material that is marked with an asterisk (*) and is intended for dedicated machine-language enthusiasts. We give a presentation of IA32 support for floating-point code. This is a particularly arcane feature of IA32, and so we advise that only people determined to work with floating-point code attempt to study this section. We give a brief presentation of GCC's support for embedding assembly code within C programs. In some applications, the programmer must drop down to assembly code to access low-level features of the machine. Embedded assembly is the best way to do this.

3.1 A Historical Perspective

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

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

8086: (1978, 29 K transistors). One of the first single-chip, 16-bit microprocessors. The 8088, a version of the 8086 with an 8-bit external bus, formed the heart of the original IBM personal computers. IBM contracted with then-tiny Microsoft to develop the MS-DOS operating system. The original models came with 32,768 bytes of memory and two floppy drives (no hard drive). Architecturally, the machines were limited to a 655,360-byte address space—addresses were only 20 bits long (1,048,576 bytes addressable), and the operating system reserved 393,216 bytes for its own use.

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

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

i486: (1989, 1.9 M transistors). Improved performance and integrated the floating-point unit onto the processor chip but did not change the instruction set.

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

PentiumPro: (1995, 6.5 M transistors). Introduced a radically new processor design, internally known as the *P6* microarchitecture. Added a class of "conditional move" instructions to the instruction set.

Pentium/MMX: (1997, 4.5 M transistors). Added new class of instructions to the Pentium processor for manipulating vectors of integers. Each datum can be 1, 2, or 4-bytes long. Each vector totals 64 bits.

Pentium II: (1997, 7 M transistors). Merged the previously separate PentiumPro and Pentium/MMX lines by implementing the MMX instructions within the P6 microarchitecture.

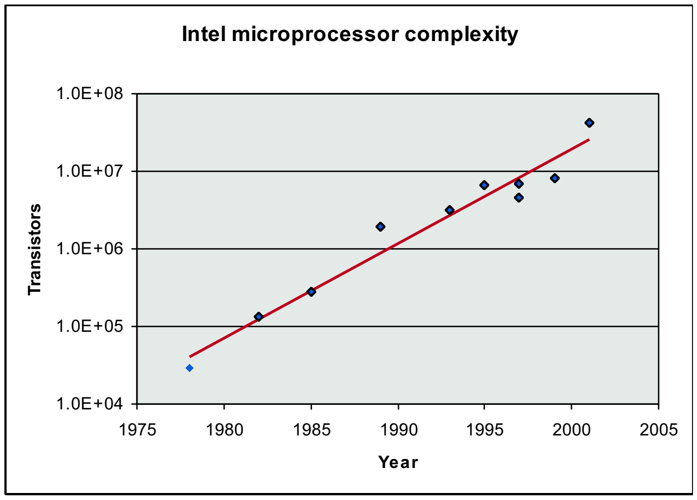
Pentium III: (1999, 8.2 M transistors). Introduced yet another class of instructions for manipulating vectors of integer or floating-point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits. Later versions of this chip went up to 24 M transistors, due to the incorporation of the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

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

Each successive processor has been designed to be backward compatible—able to run code compiled for any earlier version. As we will see, there are many strange artifacts in the instruction set due to this evolutionary heritage. Intel now calls its instruction set *IA32,* for "Intel Architecture 32-bit." The processor line is also referred to by the colloquial name "x86," reflecting the processor naming conventions up through the i486.

Aside: Why not the i586?
Intel discontinued their numeric naming convention, because they were not able to obtain trademark protection for their CPU numbers. The U. S. Trademark office does not allow numbers to be trademarked. Instead, they coined the name "Pentium" using the the Greek root word *penta* as an indication that this was their fifth-generation machine. Since then, they have used variants of this name, even though the PentiumPro is a sixth-generation machine (hence the internal name P6), and the Pentium 4 is a seventh-generation machine. Each new generation involves a major change in the processor design. **End Aside.**

Aside: Moore's Law.



If we plot the number of transistors in the different IA32 processors listed above versus the year of introduction, and use a logarithmic scale for the Y axis, we can see that the growth has been phenomenal. Fitting a line through the data, we see that the number of transistors increases at an annual rate of approximately 33%, meaning that the number of transistors doubles about every 30 months. This growth has been sustained over the roughly 25 year history of IA32.

In 1965, Gordon Moore, a founder of Intel Corporation extrapolated from the chip technology of the day, in which they could fabricate circuits with around 64 transistors on a single chip, to predict that the number of transistors per chip would double every year for the next 10 years. This predication became known as *Moore's Law*. As it turns out, his prediction was just a little bit optimistic, but also too short-sighted. Over its 40-year history the semiconductor industry has been able to double transistor counts on average every 18 months.

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

Over the years, several companies have produced processors that are compatible with Intel processors, capable of running the exact same machine-level programs. Chief among these is AMD. For years, AMD's strategy was to run just behind Intel in technology, producing processors that were less expensive although somewhat lower in performance. More recently, AMD has produced some of the highest performing processors for IA32. They were the first to the break the 1-gigahertz clock speed barrier for a commercially available microprocessor. Although we will talk about Intel processors, our presentation holds just as well for the compatible processors produced by Intel's rivals.

Much of the complexity of IA32 is not of concern to those interested in programs for the Linux operating system as generated by the GCC compiler. The memory model provided in the original 8086 and its extensions in the 80286 are obsolete. Instead, Linux uses what is referred to as *flat* addressing, where the entire memory space is viewed by the programmer as a large array of bytes.

As we can see in the list of developments, a number of formats and instructions have been added to IA32 for manipulating vectors of small integers and floating-point numbers. These features were added to allow improved performance on multimedia applications, such as image processing, audio and video encoding and decoding, and three-dimensional computer graphics. Unfortunately, current versions of GCC will not generate any code that uses these new features. In fact, in its default invocations GCC assumes it is generating code for an i386. The compiler makes no attempt to exploit the many extensions added to what is now considered a very old architecture.

3.2 Program Encodings

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

```
unix> gcc -O2 -o p p1.c p2.c
```

The command `gcc` indicates the GNU C compiler GCC. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The flag `-O2` instructs the compiler to apply level-two optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code. Level-two optimization is a good compromise between optimized performance and ease of use. All code in this book was compiled with this optimization level.

This command actually invokes a sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros. Second, the *compiler* generates assembly code versions of the two source files having names `p1.s` and `p2.s`. Next, the *assembler* converts the assembly code into binary object code files `p1.o` and `p2.o`. Finally, the *linker* merges these two object files along with code implementing standard Unix library functions (e.g., `printf`) and generates the final executable file. Linking is described in more detail in Chapter 7.

3.2.1 Machine-Level Code

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

The assembly programmer's view of the machine differs significantly from that of a C programmer. Parts of the processor state are visible that normally are hidden from the C programmer:

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

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

The program memory contains the object code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user, (for example, by using the `malloc` library procedure).

The program memory is addressed using *virtual* addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the 32-bit addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

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

3.2.2 Code Examples

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

```
int accum = 0;

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

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

```
unix> gcc -O2 -S code.c
```

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

GCC generates assembly code in its own format, known as GAS (for "Gnu ASsembler"). We will base our presentation on this format, which differs significantly from the format used in Intel documentation and by Microsoft compilers. See the bibiliographic notes for advice on locating documentation of the different assembly code formats.

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

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

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

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

```
unix> gcc -O2 -c code.c
```

This will generate an object code file `code.o` that is in binary format and hence cannot be viewed directly. Embedded within the 852 bytes of the file `code.o` is a 19 byte sequence having hexadecimal representation:

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

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

Aside: How do I find the byte representation of a program?
First we used a disassembler (to be described shortly) to determine that the code for `sum` is 19 bytes long. Then we ran the GNU debugging tool GDB on fi le `code.o` and gave it the command:

```
(gdb)  x/19xb sum
```

telling it to examine (abbreviated ‘x’) 19 hex-formatted (also abbreviated ‘x’) bytes (abbreviated ‘b’). You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 3.12. **End Aside.**

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

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

The result is (where we have added line numbers on the left and annotations on the right):

```
  Disassembly of function sum in file code.o
00000000 <sum>:
 Offset   Bytes                       Equivalent assembly language
   0:    55                           push   %ebp
   1:    89 e5                        mov    %esp,%ebp
   3:    8b 45 0c                     mov    0xc(%ebp),%eax
   6:    03 45 08                     add    0x8(%ebp),%eax
   9:    01 05 00 00 00 00            add    %eax,0x0
   f:    89 ec                        mov    %ebp,%esp
  11:    5d                           pop    %ebp
  12:    c3                           ret
  13:    90                           nop
```

On the left we see the 19 hexadecimal byte values listed in the byte sequence earlier, partitioned into groups of 1 to 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:

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

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

```
unix> gcc -O2 -o prog code.o main.c
```

The file `prog` has grown to 11,667 bytes, since it contains not just the code for our two procedures but also information used to start and terminate the program as well as to interact with the operating system. We can also disassemble the file `prog`:

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

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

```
     Disassembly of function sum in executable file prog
080483b4 <sum>:
 80483b4:  55                      push   %ebp
 80483b5:  89 e5                   mov    %esp,%ebp
 80483b7:  8b 45 0c                mov    0xc(%ebp),%eax
 80483ba:  03 45 08                add    0x8(%ebp),%eax
 80483bd:  01 05 64 94 04 08       add    %eax,0x8049464
 80483c3:  89 ec                   mov    %ebp,%esp
 80483c5:  5d                      pop    %ebp
 80483c6:  c3                      ret
 80483c7:  90                      nop
```

Note that this code is almost identical to that generated by the disassembly of `code.c`. One main difference is that the addresses listed along the left are different—the linker has shifted the location of this code to a different range of addresses. A second difference is that the linker has finally determined the location for storing global variable `accum`. On line 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:

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

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

```
  .file   "simple.c"
  .version        "01.01"
gcc2_compiled.:
.text
  .align 4
.globl simple
  .type    simple,@function
simple:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  movl (%eax),%edx
  addl 12(%ebp),%edx
  movl %edx,(%eax)
  movl %edx,%eax
  movl %ebp,%esp
  popl %ebp
  ret
.Lfe1:
  .size    simple,.Lfe1-simple
  .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

The file contains more information than we really require. All of the lines beginning with ‘.’ are directives to guide the assembler and linker. We can generally ignore these. On the other hand, there are no explanatory remarks about what the instructions do or how they relate to the source code.

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

```
 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
```

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

Figure 3.1: **Sizes of standard data types**

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

3.3 Data Formats

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

Figure 3.1 shows the machine representations used for the primitive data types of C. Note that most of the common data types are stored as double words. This includes both regular and long `int`'s, whether or not they are signed. In addition, all pointers (shown here as `char *`) are stored as 4-byte double words. Bytes are commonly used when manipulating string data. Floating-point numbers come in three different forms: single-precision (4-byte) values, corresponding to C data type `float`; double-precision (8-byte) values, corresponding to C data type `double`; and extended-precision (10-byte) values. GCC uses the data type `long double` to refer to extended-precision floating-point values. It also stores them as 12-byte quantities to improve memory system performance, as will be discussed later. Although the ANSI C standard includes `long double` as a data type, they are implemented for most combinations of compiler and machine using the same 8-byte format as ordinary `double`. The support for extended precision is unique to the combination of GCC and IA32.

As the table indicates, every operation in GAS has a single-character suffix denoting the size of the operand. For example, the `mov` (move data) instruction has three variants: `movb` (move byte), `movw` (move word), and `movl` (move double word). The suffix '`l`' is used for double words, since on many machines 32-bit quantities are referred to as "long words," a holdover from an era when 16-bit word sizes were standard. Note that GAS uses the suffix '`l`' to denote both a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating point involves an entirely different set of

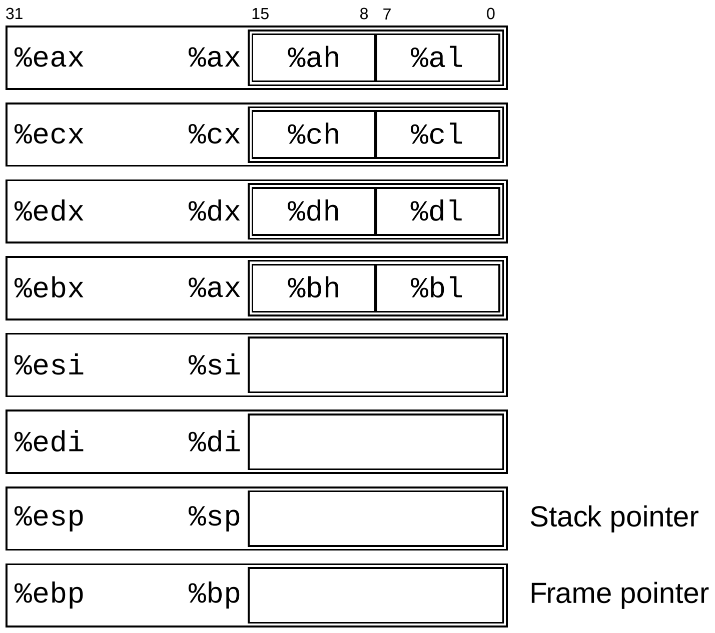


Figure 3.2: **Integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The two low-order bytes of the first four registers can be accessed independently.

instructions and registers.

3.4 Accessing Information

An IA32 central processing unit (CPU) contains a set of eight *registers* storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 3.2 diagrams the eight registers. Their names all begin with `%e`, but otherwise, they have peculiar names. With the original 8086, the registers were 16-bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first six registers can be considered general-purpose registers with no restrictions placed on their use. We said "for the most part," because some instructions use fixed registers as sources and/or destinations. In addition, within procedures there are different conventions for saving and restoring the first three registers (`%eax`, `%ecx`, and `%edx`), than for the next three (`%ebx`, `%edi`, and `%esi`). This will be discussed in Section 3.7. The final two registers (`%ebp` and `%esp`) contain pointers to important places in the program stack. They should only be altered according to the set of standard conventions for stack management.

As indicated in Figure 3.2, the low-order two bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining three bytes of the register do not change. Similarly, the low-order 16 bits of each register can be read or written by word operation instructions. This

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

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

feature stems from IA32's evolutionary heritage as a 16-bit microprocessor.

3.4.1 Operand Specifiers

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. IA32 supports a number of operand forms (Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. With GAS, these are written with a '`$`' followed by an integer using standard C notation, such as, `$-577` or `$0x1F`. Any value that fits in a 32-bit word can be used, although the assembler will use one or two-byte encodings when possible. The second type, *register*, denotes the contents of one of the registers, either one of the eight 32-bit registers (e.g., `%eax`) for a double-word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In our figure, we use the notation $\mathtt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $\mathsf{R}[\mathtt{E}_a]$, viewing the set of registers as an array R indexed by register identifiers.

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

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.**

Practice Problem 3.1:

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

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

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

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

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

3.4.2 Data Movement Instructions

Among the most heavily used instructions are those that perform data movement. The generality of the operand notation allows a simple move instruction to perform what in many machines would require a number of instructions. Figure 3.4 lists the important data movement instructions. The most common is the `movl` instruction for moving double words. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination.

The following `movl` instruction examples show the five possible combinations of source and destination types. Recall that the source operand comes first and the destination second:

```
movl $0x4050,%eax         Immediate--Register
movl %ebp,%esp            Register--Register
movl (%edi,%ecx),%eax     Memory--Register
movl $-17,(%esp)          Immediate--Memory
movl %eax,-12(%ebp)       Register--Memory
```

The `movb` instruction is similar, except that it moves just a single byte. When one of the operands is a register, it must be one of the eight single-byte register elements illustrated in Figure 3.2. Similarly, the `movw` instruction moves two bytes. When one of its operands is a register, it must be one of the eight 2-byte register elements shown in Figure 3.2.

Both the `movsbl` and the `movzbl` instruction serve to copy a byte and to set the remaining bits in the destination. The `movsbl` instruction takes a single-byte source operand, performs a sign extension to 32 bits (i.e., it sets the high-order 24 bits to the most significant bit of the source byte), and copies this to a double-word destination. Similarly, the `movzbl` instruction takes a single-byte source operand, expands it to 32 bits by adding 24 leading zeros, and copies this to a double-word destination.

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

```
 Assume initially that %dh = 8D, %eax = 98765432
movb %dh,%al               %eax = 9876548D
movsbl %dh,%eax            %eax = FFFFFF8D
movzbl %dh,%eax            %eax = 0000008D
```

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

The final two data movement operations are used to push data onto and pop data from the program stack. As we will see, the stack plays a vital role in the handling of procedure calls. Both the `pushl` and the `popl` instructions take a single operand—the data source for pushing and the data destination for popping. The program stack is stored in some region of memory. As illustrated in Figure 3.5, the stack grows downward such that the top element of the stack has the lowest address of all stack elements. (By convention, we draw stacks upside-down, with the stack “top” shown at the bottom of the figure). The stack pointer `%esp` holds the address of the top stack element. Pushing a double-word value onto the stack therefore involves first decrementing the stack pointer by 4 and then writing the value at the new top of stack address. Therefore, the behavior of the instruction `pushl %ebp` is equivalent to that of the following pair of instructions:

```
subl $4,%esp
movl %ebp,(%esp)
```

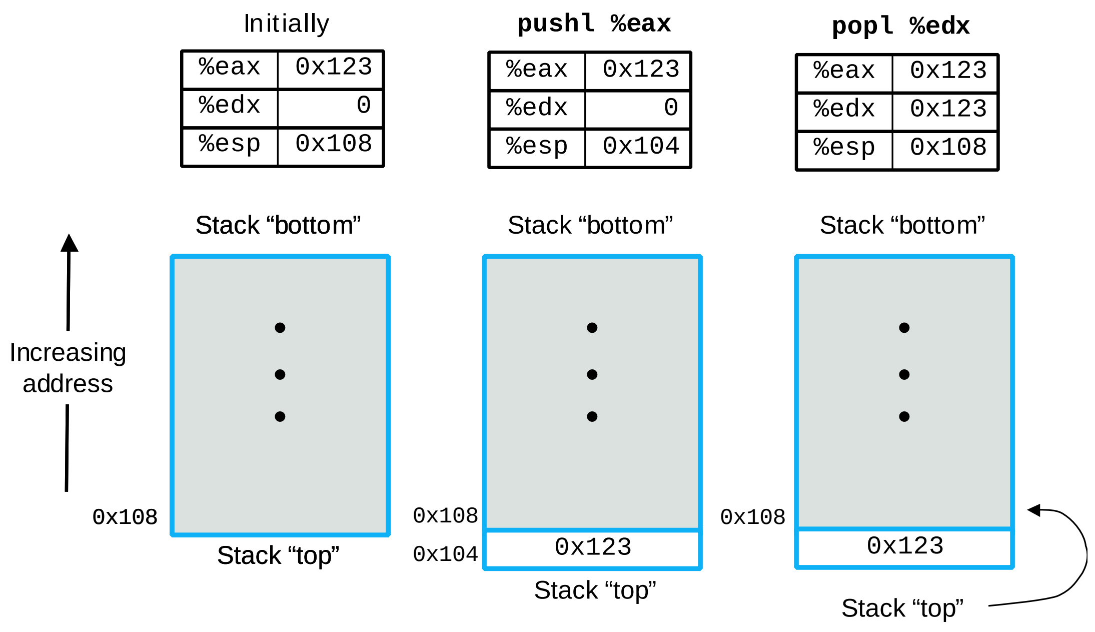


Figure 3.5: **Illustration of stack operation.** By convention, we draw stacks upside-down, so that the "top" of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register `%esp`) and storing to memory, while popping involves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

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

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

```
movl (%esp),%eax
addl $4,%esp
```

The third column of Figure 3.5 illustrates the effect of executing the instruction `popl %edx` immediately after executing the `pushl`. Value `0x123` would be read from memory and written to register `%edx`. Register `%esp` would be incremented back to `0x108`. As shown in the figure, the value `0x123` would remain at memory location `0x104` until it is overwritten by another push operation. However, the stack top is always considered to be the address indicated by `%esp`.

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

code/asm/exchange.c

```
int exchange(int *xp, int y)
{
    int x = *xp;

    *xp = y;
    return x;
}
```

code/asm/exchange.c

(a) C code

```
movl 8(%ebp),%eax     Get xp
movl 12(%ebp),%edx    Get y
movl (%eax),%ecx      Get x at *xp
movl %edx,(%eax)      Store y at *xp
movl %ecx,%eax        Set x as return value
```

(b) Assembly code

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

3.4.3 Data Movement Example

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

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

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

The statement

```
*xp = y;
```

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

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

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

This code will print

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

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

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

When the body of the procedure starts execution, procedure parameters `xp` and `y` are stored at offsets 8 and 12 relative to the address in register `%ebp`. Instructions 1 and 2 then move these parameters into registers `%eax` and `%edx`. Instruction 3 dereferences `xp` and stores the value in register `%ecx`, corresponding to program value `x`. Instruction 4 stores `y` at `xp`. Instruction 5 moves `x` to register `%eax`. By convention, any function returning an integer or pointer value does so by placing the result in register `%eax`, and so this instruction implements line 6 of the C code. This example illustrates how the `movl` instruction can be used to read from memory to a register (instructions 1 to 3), to write from a register to memory (instruction 4), and to copy from one register to another (instruction 5).

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

Practice Problem 3.2:

You are given the following information. A function with prototype

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

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

```
movl 8(%ebp),%edi
movl 12(%ebp),%ebx
movl 16(%ebp),%esi
movl (%edi),%eax
movl (%ebx),%edx
movl (%esi),%ecx
movl %eax,(%ebx)
movl %edx,(%esi)
movl %ecx,(%edi)
```

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

Write C code for `decode1` that will have an effect equivalent to the assembly code above. You can test your answer by compiling your code with the `-S` switch. Your compiler may generate code that differs in the usage of registers or the ordering of memory references, but it should still be functionally equivalent.

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow$ &S	Load effective address
`incl`	D	$D \leftarrow D$ + 1	Increment
`decl`	D	$D \leftarrow D$ - 1	Decrement
`negl`	D	$D \leftarrow$ -D	Negate
`notl`	D	$D \leftarrow$ ~D	Complement
`addl`	S, D	$D \leftarrow D$ + S	Add
`subl`	S, D	$D \leftarrow D$ - S	Subtract
`imull`	S, D	$D \leftarrow D$ * S	Multiply
`xorl`	S, D	$D \leftarrow D$ ^ S	Exclusive-or
`orl`	S, D	$D \leftarrow D$ | S	Or
`andl`	S, D	$D \leftarrow D$ & S	And
`sall`	k, D	$D \leftarrow D$ << k	Left shift
`shll`	k, D	$D \leftarrow D$ << k	Left shift (same as `sall`)
`sarl`	k, D	$D \leftarrow D$ >> k	Arithmetic right shift
`shrl`	k, D	$D \leftarrow D$ >> k	Logical right shift

Figure 3.7: **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. Note the nonintuitive ordering of the operands with GAS.

3.5 Arithmetic and Logical Operations

Figure 3.7 lists some of the double-word integer operations, divided into four groups. *Binary* operations have two operands, while *unary* operations have one operand. These operands are specified using the same notation as described in Section 3.4. With the exception of `leal`, each of these instructions has a counterpart that operates on words (16 bits) and on bytes. The suffix '`l`' is replaced by '`w`' for word operations and '`b`' for the byte operations. For example, `addl` becomes `addw` or `addb`.

3.5.1 Load Effective Address

The Load Effective Address `leal` instruction is actually a variant of the `movl` instruction. It has the form of an instruction that reads from memory to a register, but it does not reference memory at all. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 3.7 using the C address operator &S. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. The destination operand must be a register.

Practice Problem 3.3:

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