Aside ATT versus Intel assembly-code formats

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

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

This gives the following assembly code:

```
multstore:
  push    rbx
  mov     rbx, rdx
  call    mult2
  mov     QWORD PTR [rbx], rax
  pop     rbx
  ret
```

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

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

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

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

3.3 Data Formats

Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term "word" to refer to a 16-bit data type. Based on this, they refer to 32-bit quantities as "double words," and 64-bit quantities as "quad words." Figure 3.1 shows the x86-64 representations used for the primitive data types of C. Standard `int` values are stored as double words (32 bits). Pointers (shown here as `char *`) are stored as 8-byte quad words, as would be expected in a 64-bit machine. With x86-64, data type `long` is implemented with 64 bits, allowing a very wide range of values. Most of our code examples in this chapter use pointers and `long` data

Web Aside ASM:EASM Combining assembly code with C programs

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

There are two ways to incorporate assembly code into C programs. First, we can write an entire function as a separate assembly-code file and let the assembler and linker combine this with code we have written in C. Second, we can use the *inline assembly* feature of GCC, where brief sections of assembly code can be incorporated into a C program using the `asm` directive. This approach has the advantage that it minimizes the amount of machine-specific code.

Of course, including assembly code in a C program makes the code specific to a particular class of machines (such as x86-64), and so it should only be used when the desired feature can only be accessed in this way.

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

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

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

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

not implemented with the same high-performance hardware as is the case for single- and double-precision arithmetic.

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

3.4 Accessing Information

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

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

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

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