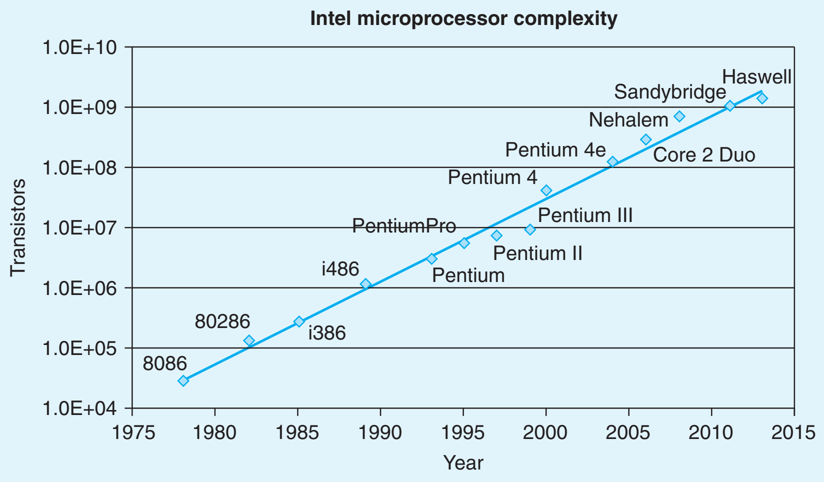
Aside Moore's Law



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 37%, 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 (by 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 prediction 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 50 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, including the storage capacities of magnetic disks and semiconductor memories. These remarkable growth rates have been the major driving forces of the computer revolution.

with the introduction of SSE2. Although we see vestiges of the historical evolution of x86 in x86-64 programs, many of the most arcane features of x86 do not appear.

3.2 Program Encodings

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

```
linux> gcc -Og -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 `-Og`[1] instructs the compiler to apply a level of optimization that yields machine code that follows the overall structure of the original C code. 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 `-Og` optimization as a learning tool and then see what happens as we increase the level of optimization. In practice, higher levels of optimization (e.g., specified with the option `-O1` or `-O2`) are considered a better choice in terms of the resulting program performance.

The `gcc` command invokes an entire 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` (as specified by the command-line directive `-o 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 1.9.3, 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 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

1. This optimization level was introduced in GCC version 4.8. Earlier versions of GCC, as well as non-GNU compilers, will not recognize this option. For these, using optimization level one (specified with the command-line flag `-O1`) is probably the best choice for generating code that follows the original program structure.

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.

The machine code for x86-64 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 `%rip` in x86-64) indicates the address in memory of the next instruction to be executed.
- The integer *register file* contains 16 named locations storing 64-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 arguments and local variables of a procedure, as well as 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 vector registers can each hold one or more integer or floating-point values.

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 (e.g., 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, x86-64 virtual addresses are represented by 64-bit words. In current implementations of these machines, the upper 16 bits must be set to zero, and so an address can potentially specify a byte over a range of 2^{48}, or 64 terabytes. More typical programs will only have access to a few megabytes, or perhaps several gigabytes. The operating system manages

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.

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 `mstore.c` containing the following function definition:

```
long mult2(long, long);

void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

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

```
linux> gcc -Og -S mstore.c
```

This will cause GCC to run the compiler, generating an assembly file `mstore.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 following set of lines:

```
multstore:
  pushq   %rbx
```

Aside How do I display the byte representation of a program?

To display the binary object code for a program (say, `mstore`), we use a *disassembler* (described below) to determine that the code for the procedure is 14 bytes long. Then we run the GNU debugging tool GDB on file `mstore.o` and give it the command

```
(gdb) x/14xb multstore
```

telling it to display (abbreviated '`x`') 14 hex-formatted (also '`x`') bytes ('`b`') starting at the address where function `multstore` is located. You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 3.10.2.

```
  movq    %rdx, %rbx
  call    mult2
  movq    %rax, (%rbx)
  popq    %rbx
  ret
```

Each indented line in the code corresponds to a single machine instruction. For example, the `pushq` instruction indicates that the contents of register `%rbx` should be pushed onto the program stack. All information about local variable names or data types has been stripped away.

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

```
linux> gcc -Og -c mstore.c
```

This will generate an object-code file `mstore.o` that is in binary format and hence cannot be viewed directly. Embedded within the 1,368 bytes of the file `mstore.o` is a 14-byte sequence with the hexadecimal representation

```
53 48 89 d3 e8 00 00 00 00 48 89 03 5b c3
```

This is the object code corresponding to the assembly instructions listed previously. A key lesson to learn from this is that the program 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.

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:

```
linux> objdump -d mstore.o
```

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

```
Disassembly of function sum in binary file mstore.o
0000000000000000 <multstore>:
Offset   Bytes                          Equivalent assembly language
   0:    53                             push   %rbx
   1:    48 89 d3                       mov    %rdx,%rbx
   4:    e8 00 00 00 00                 callq  9 <multstore+0x9>
   9:    48 89 03                       mov    %rax,(%rbx)
   c:    5b                             pop    %rbx
   d:    c3                             retq
```

On the left we see the 14 hexadecimal byte values, listed in the byte sequence shown earlier, partitioned into groups of 1 to 5 bytes each. Each of these groups is a single instruction, with the assembly-language equivalent shown on the right.

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

- x86-64 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 `pushq %rbx` can start with byte value `53`.
- 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 '`q`' from many of the instructions. These suffixes are size designators and can be omitted in most cases. Conversely, the disassembler adds the suffix '`q`' to the `call` and `ret` instructions. Again, these suffixes can safely be omitted.

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:

```
#include <stdio.h>

void multstore(long, long, long *);

int main() {
    long d;
    multstore(2, 3, &d);
    printf("2 * 3 --> %ld\n", d);
    return 0;
}
```

```
long mult2(long a, long b) {
    long s = a * b;
    return s;
 }
```

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

```
linux> gcc -Og -o prog main.c mstore.c
```

The file `prog` has grown to 8,655 bytes, since it contains not just the machine code for the procedures we provided but also code used to start and terminate the program as well as to interact with the operating system.

We can disassemble the file `prog`:

```
linux> objdump -d prog
```

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

```
     Disassembly of function sum in binary file prog
1    0000000000400540 <multstore>:
2      400540:  53                      push   %rbx
3      400541:  48 89 d3                mov    %rdx,%rbx
4      400544:  e8 42 00 00 00          callq  40058b <mult2>
5      400549:  48 89 03                mov    %rax,(%rbx)
6      40054c:  5b                      pop    %rbx
7      40054d:  c3                      retq
8      40054e:  90                      nop
9      40054f:  90                      nop
```

This code is almost identical to that generated by the disassembly of `mstore.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 filled in the address that the `callq` instruction should use in calling the function `mult2` (line 4 of the disassembly). One task for the linker is to match function calls with the locations of the executable code for those functions. A final difference is that we see two additional lines of code (lines 8–9). These instructions will have no effect on the program, since they occur after the return instruction (line 7). They have been inserted to grow the code for the function to 16 bytes, enabling a better placement of the next block of code in terms of memory system performance.

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 we give the command

```
linux> gcc -Og -S mstore.c
```

to generate the file `mstore.s`. The full content of the file is as follows:

```
        .file   "010-mstore.c"
        .text
        .globl  multstore
        .type   multstore, @function
multstore:
        pushq   %rbx
        movq    %rdx, %rbx
        call    mult2
        movq    %rax, (%rbx)
        popq    %rbx
        ret
        .size   multstore, .-multstore
        .ident  "GCC: (Ubuntu 4.8.1-2ubuntu1~12.04) 4.8.1"
        .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:

```
   void multstore(long x, long y, long *dest)
   x in %rdi, y in %rsi, dest in %rdx
1  multstore:
2    pushq   %rbx            Save %rbx
3    movq    %rdx, %rbx      Copy dest to %rbx
4    call    mult2           Call mult2(x, y)
5    movq    %rax, (%rbx)    Store result at *dest
6    popq    %rbx            Restore %rbx
7    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.

We also provide Web asides to cover material intended for dedicated machine-language enthusiasts. One Web aside describes IA32 machine code. Having a background in x86-64 makes learning IA32 fairly simple. 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

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