Chapter 0
A Look Under the Hood

Norman Matloff
University of California at Davis

February 4, 2001

Throughout these chapters, we will use an analogy of computers to automobiles. A look under the hood of a car gives us a view of the engine, the source of power which we utilize when we drive the car. In the case of a computer, the analogy of "driving" is programming in a high-level language (HLL) such as C/C++, and there are two sources of power:

- the hardware, including the central processing unit which executes the computer's machine language,
- the low-level software, consisting of various services that the operating system makes available to programs.

When you write a program in an HLL, the compiler will translate your HLL statements into the machine language of that computer, and to calls to subroutines within the operating system. And of course, it is this translated version of your program that is actually executed, not your HLL program itself. Thus the hardware and operating system do indeed form the "engine" of the computer.

It is our goal to demystify this engine, by giving the reader an introductory "look under the hood." Note carefully that this engine consists of both hardware and software components, each of which is equally important.[1]

The skeptical reader will point out here, quite correctly, that one does not need to know about automobile engines in order to be a good driver. This raises the question as to whether one needs to know about a computer's hardware and operating system in order to be a good programmer. The answer to this question is that *such knowledge is actually vital to good programming*. Professional developers of software need much, much more than mere programming skills—they *do* need to know how the computer's "engine" works. And again, it is equally important to understand both the hardware and the software components of this engine.

For example, consider the following C code, which finds the sum of two variables X and Y of type **int**, storing the total in an **int** variable Sum:

```
Sum = X + Y;
```

[1]The term **hardware** in this book will refer to the major physical components of a computer. Our discussion will concern the *functions* of these components, i.e. what is known as **computer architecture**, but not the details of the electronic implementation, which are beyond our scope here. Thus no background on electronics is needed or used.

Suppose that you know that both X and Y are nonnegative. Then you would expect that the value of Sum, after the statement is executed, will be nonnegative. But actually, *it is entirely possible that Sum will turn out to have a negative value*. For example, suppose that X is 28,502 and Y is 12,344, and the program is run on a machine with 16-bit **word size** (do not worry about this term, which we will discuss later). Then value of Sum will be -24,689!

How could Sum become negative? After you learn how the hardware stores positive and negative integers, a negative value for Sum will be no mystery to you at all. But imagine how helpless you would be without this knowledge. Suppose your program were more complex than our simplified example above. Then you might spend hours or even days searching for the bug, a search which would be in vain, since the problem would be a *conceptual* error concerning the computer's engine, not a *programming* error.

Similarly, a C program which works just fine on one machine might fail on another, due to the order in which data are stored internally within the hardware. Some machines use **big-endian** data storage while others use **little-endian** storarge. We will explain these terms in Chapter 1, but the point now is that again even a high-level language programmer needs to know what happens "under the hood."

In another example, the author was once involved in a large software development project for a database application. After the development team finished the product and presented it to the client for testing, the client pronounced the product as being completely unacceptable. The programmers were shocked, since the client confirmed that the program was producing correct results. But the client, who had used similar products before, said that the program was too slow. The program was taking about 15 seconds to respond to his commands, while from experience with other programs of this type, he was expecting essentially instantaneous response. Upon looking closer at the program, the development team found a subtle problem which made the program's use of the operating system extremely inefficient, resulting in the 15-second delay which had been upsetting the client. After they changed the manner in which the program was making calls to operating system services, the response time did become instantaneous, and the client was pleased. So as in the last example, we again see that it is not enough to be just a good programmer—knowledge of the underlying engine can be crucial to the success of the program.

Another example in which this knowledge is important is the area of computer security, which is becoming of central interest in many organizations. Most attacks by "hackers" to invade computer systems are based on exploitation of "under the hood" aspects of those systems, so defense against those attacks requires equally thorough understanding of those aspects.

The examples above, which are just a few among the many which occur in real-world software development, show why an understanding of the computer's engine is important even when writing in an HLL. Furthermore, in some applications we must program in the computer's machine language directly, rather than indirectly through an HLL. The reason for this is that some applications require that our program access some special feature of our particular hardware. This is impossible in an HLL, since HLLs are by definition machine-independent languages, so we must write such applications directly in our computer's machine language.

This situation arises, for example, in **embedded applications**, in which a computer is used to control a machine, such as an automatic bank teller machine, a robot, and even common household items such as washing machines and autofocus cameras. Computers are in all these machines, and the programs which run on them do need to access machine-specific items.

On the other hand, machine language (ML) programming is inconvenient and hard for others to read. For this reason, the modern software engineering philosophy is that whenever a situation arises in which ML is needed, we minimize the usage of it by writing most of the program in an HLL and only the crucial section of the program in ML.

Finally, one must understand the differences among various computer engines when making decisions as to what computing equipment should be purchased for a given application—whether it involves recommending to an employer or client what workstations are most appropriate for his/her setting, or answering your Uncle Bill's question as to what kind of personal computer he should buy for home use.

For all these reasons, our goal will be to get a thorough "look under the hood" at modern computer systems. It is of course vital that we make the concepts concrete, by working with specific computer types, and that we look at both major philosophies common in computers used today:

- Complex Instruction Set Computers (CISC): The chips inside these computers are capable of many different fundamental operations, called **instructions**.
- Reduced Instruction Set Computers (RISC): The name here came from the fact that the chips inside these computers originally had only a few instruction types. That is not necessarily true today, but what is retained is the very primitive nature of their instructions. In many cases, in order to achieve the same effect that a CISC machine might accomplish in a single instruction, RISC machines may have to apply several instructions in concert. For example, virtually every CISC machine has instructions to do addition and multiplication, but some RISC machines can only do addition; in order to do a multiplication, such machines must set up a loop of addition instructions (or resort to some other such trick). But RISC proponents says that this is more than compensated by the fact that the lean instruction set of a RISC machine allows it to run faster—"lean and mean."

The main example of CISC today is the Intel family, such as the Pentium chip (and its ancestors, such as the i386) found in PCs.

Most newer chips are of the RISC type, including MIPS (used in Silicon Graphics workstations), SPARC (Sun workstations), Alpha (produced by Compaq) and PowerPC (Macintoshes).

We will be using public-domain simulators for DLX and Mac-1 which are available via the Internet (more details later).

In the course of working through these chapters, you will become proficient at writing machine language programs,[2] *But remember that this is not our goal at all. The goal is not to "learn yet another language," but rather to illustrate principles of computer systems. The programming is thus a means, not an end.*

[2]Actually, you will write in **assembly language**, which you will see is essentially the same as machine language.