1

INTRODUCTION

A digital computer is a machine that can do work for people by carrying out instructions given to it. A sequence of instructions describing how to perform a certain task is called a **program**. The electronic circuits of each computer can recognize and directly execute a limited set of simple instructions into which all its programs must be converted before they can be executed. These basic instructions are rarely much more complicated than

Add two numbers.

Check a number to see if it is zero.

Copy a piece of data from one part of the computer's memory to another.

Together, a computer's primitive instructions form a language in which people can communicate with the computer. Such a language is called a **machine language**. The people designing a new computer must decide what instructions to include in its machine language. Usually, they try to make the primitive instructions as simple as possible consistent with the computer's intended use and performance requirements, in order to reduce the complexity and cost of the electronics needed. Because most machine languages are so simple, it is difficult and tedious for people to use them.

This simple observation has, over the course of time, led to a way of structuring computers as a sequence of abstractions, each abstraction building on the one below it. In this way, the complexity can be mastered and computer systems can be designed in a systematic, organized way. We call this approach **structured computer organization** and have named the book after it. In the next section we will describe what we mean by this term. After that we will look at some historical developments, the state of the art, and some important examples.

1.1 STRUCTURED COMPUTER ORGANIZATION

As mentioned above, there is a large gap between what is convenient for people and what is convenient for computers. People want to do *X*, but computers can only do *Y*. This leads to a problem. The goal of this book is to explain how this problem can be solved.

1.1.1 Languages, Levels, and Virtual Machines

The problem can be attacked in two ways: both involve designing a new set of instructions that is more convenient for people to use than the set of built-in machine instructions. Taken together, these new instructions also form a language, which we will call L1, just as the built-in machine instructions form a language, which we will call L0. The two approaches differ in the way programs written in L1 are executed by the computer, which, after all, can only execute programs written in its machine language, L0.

One method of executing a program written in L1 is first to replace each instruction in it by an equivalent sequence of instructions in L0. The resulting program consists entirely of L0 instructions. The computer then executes the new L0 program instead of the old L1 program. This technique is called **translation**.

The other technique is to write a program in L0 that takes programs in L1 as input data and carries them out by examining each instruction in turn and executing the equivalent sequence of L0 instructions directly. This technique does not require first generating a new program in L0. It is called **interpretation** and the program that carries it out is called an **interpreter**.

Translation and interpretation are similar. In both methods, the computer carries out instructions in L1 by executing equivalent sequences of instructions in L0. The difference is that, in translation, the entire L1 program is first converted to an L0 program, the L1 program is thrown away, and then the new L0 program is loaded into the computer's memory and executed. During execution, the newly generated L0 program is running and in control of the computer.

In interpretation, after each L1 instruction is examined and decoded, it is carried out immediately. No translated program is generated. Here, the interpreter is in control of the computer. To it, the L1 program is just data. Both methods, and increasingly, a combination of the two, are widely used.

Rather than thinking in terms of translation or interpretation, it is often simpler to imagine the existence of a hypothetical computer or **virtual machine** whose machine language is L1. Let us call this virtual machine M1 (and let us call the machine corresponding to L0, M0). If such a machine could be constructed cheaply enough, there would be no need for having language L0 or a machine that executed programs in L0 at all. People could simply write their programs in L1 and have the computer execute them directly. Even if the virtual machine whose language is L1 is too expensive or complicated to construct out of electronic circuits, people can still write programs for it. These programs can be either interpreted or translated by a program written in L0 that itself can be directly executed by the existing computer. In other words, people can write programs for virtual machines, just as though they really existed.

To make translation or interpretation practical, the languages L0 and L1 must not be "too" different. This constraint often means that L1, although better than L0, will still be far from ideal for most applications. This result is perhaps discouraging in light of the original purpose for creating L1—relieving the programmer of the burden of having to express algorithms in a language more suited to machines than people. However, the situation is not hopeless.

The obvious approach is to invent still another set of instructions that is more people-oriented and less machine-oriented than L1. This third set also forms a language, which we will call L2 (and with virtual machine M2). People can write programs in L2 just as though a virtual machine with L2 as its machine language really existed. Such programs can be either translated to L1 or executed by an interpreter written in L1.

The invention of a whole series of languages, each one more convenient than its predecessors, can go on indefinitely until a suitable one is finally achieved. Each language uses its predecessor as a basis, so we may view a computer using this technique as a series of **layers** or **levels**, one on top of another, as shown in Fig. 1-1. The bottommost language or level is the simplest and the topmost language or level is the most sophisticated.

There is an important relation between a language and a virtual machine. Each machine has a machine language, consisting of all the instructions that the machine can execute. In effect, a machine defines a language. Similarly, a language defines a machine—namely, the machine that can execute all programs written in the language. Of course, the machine defined by a certain language may be enormously complicated and expensive to construct directly out of electronic circuits but we can imagine it nevertheless. A machine with C or C++ or Java as its machine language would be complex indeed but could be built using today's technology. There is a good reason, however, for not building such a computer: it would not be cost effective compared to other techniques. Merely being doable is not good enough: a practical design must be cost effective as well.

In a certain sense, a computer with *n* levels can be regarded as *n* different virtual machines, each one with a different machine language. We will use the terms

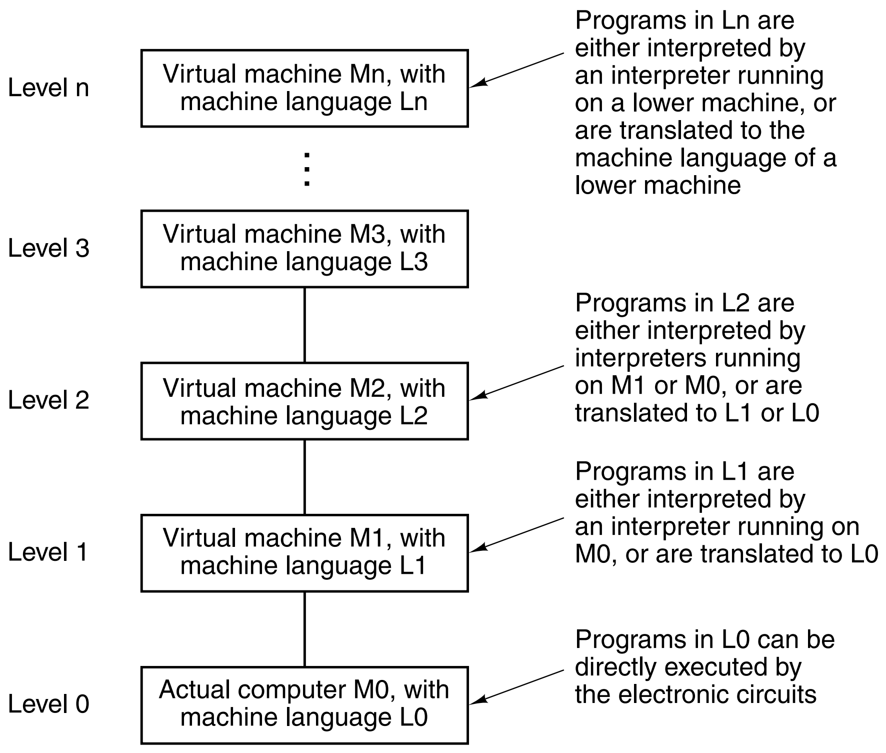
Level n
Virtual machine Mn, with machine language Ln
Programs in Ln are either interpreted by an interpreter running on a lower machine, or are translated to the machine language of a lower machine
Level 3
Virtual machine M3, with machine language L3
Level 2
Virtual machine M2, with machine language L2
Programs in L2 are either interpreted by interpreters running on M1 or M0, or are translated to L1 or L0
Level 1
Virtual machine M1, with machine language L1
Programs in L1 are either interpreted by an interpreter running on M0, or are translated to L0
Level 0
Actual computer M0, with machine language L0
Programs in L0 can be directly executed by the electronic circuits

Figure 1-1. A multilevel machine.

"level" and "virtual machine" interchangeably. However, please note that like many terms in computer science, "virtual machine" has other meanings as well. We will look at another one of these later on in this book. Only programs written in language L0 can be directly carried out by the electronic circuits, without the need for intervening translation or interpretation. Programs written in L1, L2, ... L*n* must be either interpreted by an interpreter running on a lower level or translated to another language corresponding to a lower level.

A person who writes programs for the level *n* virtual machine need not be aware of the underlying interpreters and translators. The machine structure ensures that these programs will somehow be executed. It is of no real interest whether they are carried out step by step by an interpreter which, in turn, is also carried out by another interpreter, or whether they are carried out by the electronic circuits directly. The same result appears in both cases: the programs are executed.

Most programmers using an *n*-level machine are interested only in the top level, the one least resembling the machine language at the very bottom. However, people interested in understanding how a computer really works must study all the levels. People who design new computers or new levels must also be familiar with levels other than the top one. The concepts and techniques of constructing machines as a series of levels and the details of the levels themselves form the main subject of this book.