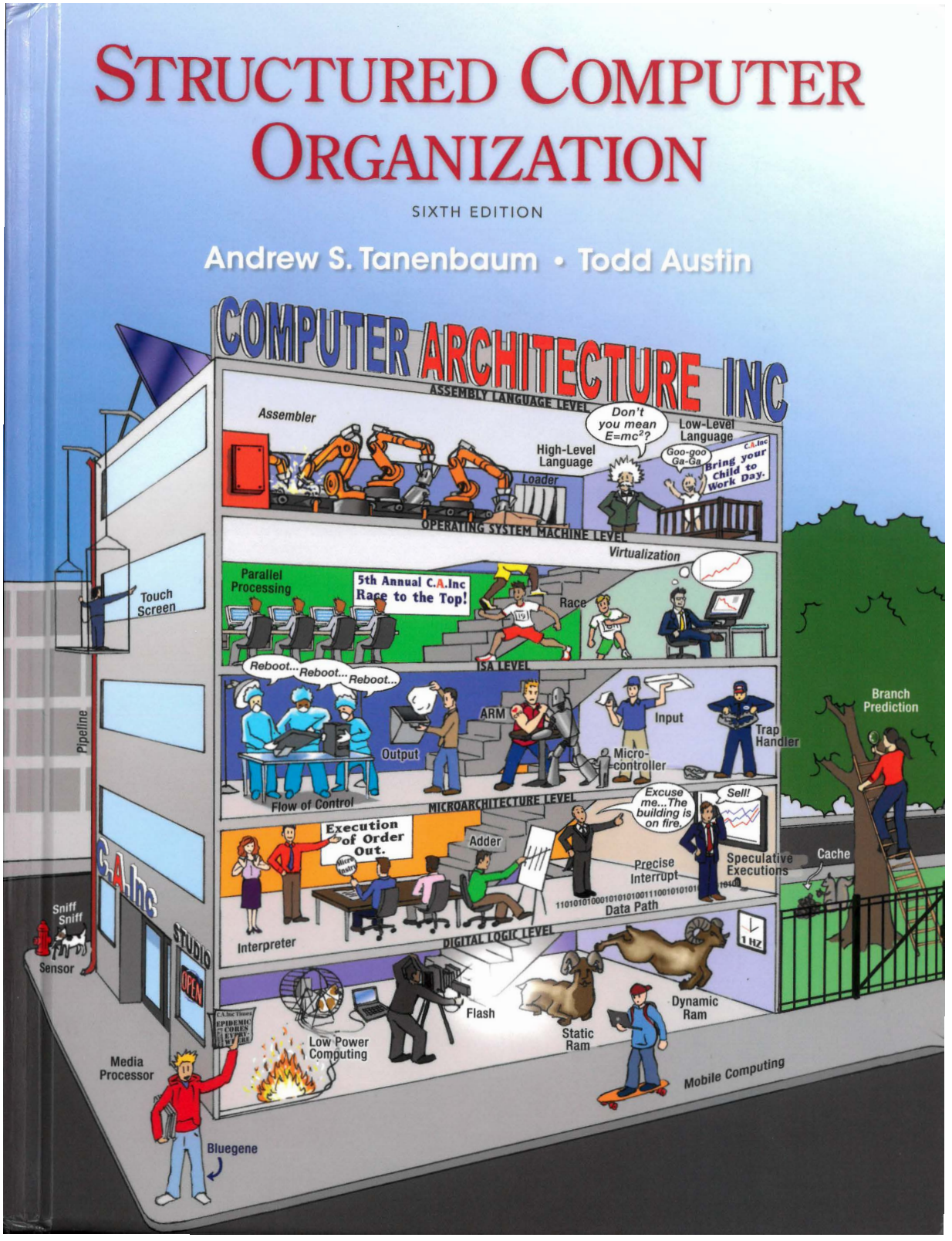
STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

Andrew S. Tanenbaum • Todd Austin



Spotify USA Inc. Exhibit 1015 Page 1

STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

Spotify USA Inc. Exhibit 1015 Page 2

STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

ANDREW S. TANENBAUM
Vrije Universiteit
Amsterdam, The Netherlands

TODD AUSTIN
University of Michigan
Ann Arbor, Michigan, United States

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Spotify USA Inc. Exhibit 1015 Page 3

Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson (Dunkelberger)
Associate Editor: Carole Snyder
Director of Marketing: Christy Lesko
Marketing Manager: Yez Alayan
Senior Marketing Coordinator: Kathryn Ferranti
Director of Production: Erin Gregg
Managing Editor: Jeff Holcomb
Associate Managing Editor: Robert Engelhardt
Manufacturing Buyer: Lisa McDowell
Art Director: Anthony Gemmellaro
Cover Illustrator: Jason Consalvo
Manager, Rights and Permissions: Michael Joyce
Media Editor: Daniel Sandin
Media Project Manager: Renata Butera
Printer/Binder: LSC Communications
Cover Printer: LSC Communications

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear in the Credits section in the end matter of this text.

Copyright © 2013, 2006, 1999 Pearson Education, Inc., publishing as Prentice Hall. All rights reserved. Printed in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to 201-236-3290.

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data
Tanenbaum, Andrew S.,
Structured computer organization / Andrew S. Tanenbaum, Todd Austin. -- 6th ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-13-291652-3
ISBN-10: 0-13-291652-5
1. Computer programming. 2. Computer organization. I. Austin, Todd. II. Title.

QA76.6.T38 2013
005.1--dc23

2012021627

5 2021

PEARSON

ISBN 10: 0-13-291652-5
ISBN 13: 978-0-13-291652-3

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

Spotify USA Inc. Exhibit 1015 Page 5

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.

Spotify USA Inc. Exhibit 1015 Page 6

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

Spotify USA Inc. Exhibit 1015 Page 7

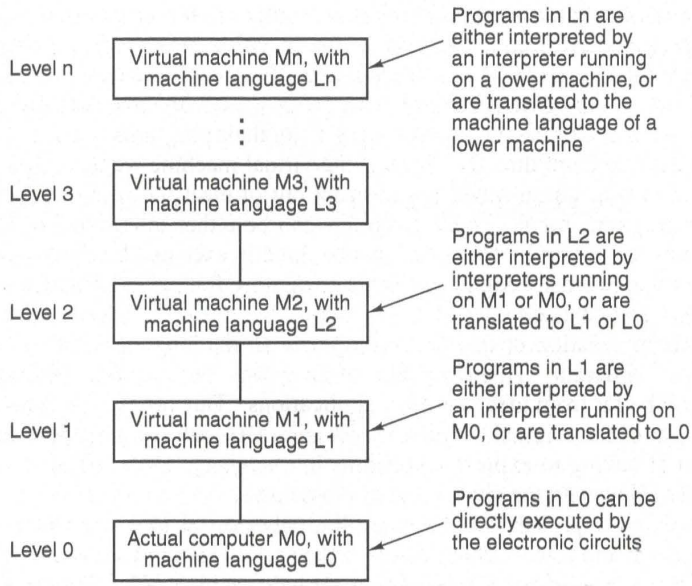


Figure 1-1. A multilevel machine.

"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.

Spotify USA Inc. Exhibit 1015 Page 8

1.1.2 Contemporary Multilevel Machines

Most modern computers consist of two or more levels. Machines with as many as six levels exist, as shown in Fig. 1-2. Level 0, at the bottom, is the machine's true hardware. Its circuits carry out the machine-language programs of level 1. For the sake of completeness, we should mention the existence of yet another level below our level 0. This level, not shown in Fig. 1-2 because it falls within the realm of electrical engineering (and is thus outside the scope of this book), is called the **device level**. At this level, the designer sees individual transistors, which are the lowest-level primitives for computer designers. If one asks how transistors work inside, that gets us into solid-state physics.

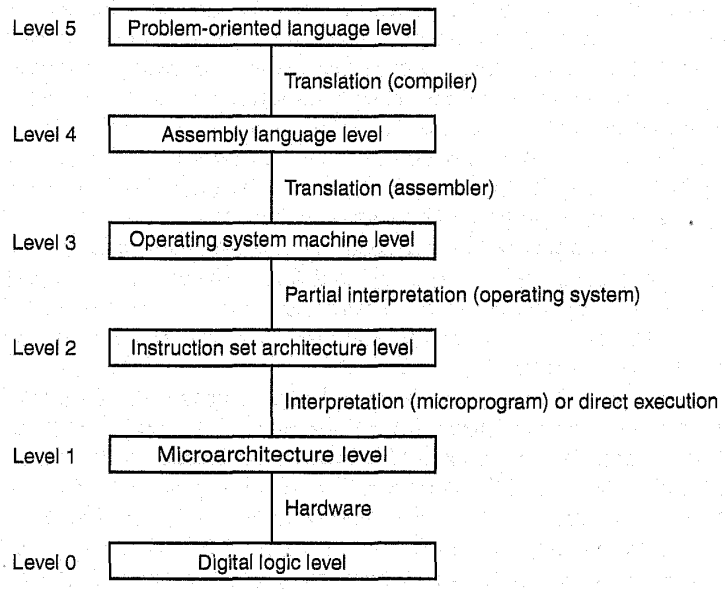


Figure 1-2. A six-level computer. The support method for each level is indicated below it (along with the name of the supporting program).

At the lowest level that we will study, the **digital logic level**, the interesting objects are called **gates**. Although built from analog components, such as transistors, gates can be accurately modeled as digital devices. Each gate has one or more digital inputs (signals representing 0 or 1) and computes as output some simple function of these inputs, such as AND or OR. Each gate is built up of at most a handful of transistors. A small number of gates can be combined to form a 1-bit memory, which can store a 0 or a 1. The 1-bit memories can be combined in groups of (for example) 16, 32, or 64 to form registers. Each **register** can hold a single binary

Spotify USA Inc. Exhibit 1015 Page 9

number up to some maximum. Gates can also be combined to form the main computing engine itself. We will examine gates and the digital logic level in detail in Chap. 3.

The next level up is the **microarchitecture level**. At this level we see a collection of (typically) 8 to 32 registers that form a local memory and a circuit called an **ALU** (**Arithmetic Logic Unit**), which is capable of performing simple arithmetic operations. The registers are connected to the ALU to form a **data path**, over which the data flow. The basic operation of the data path consists of selecting one or two registers, having the ALU operate on them (for example, adding them together), and storing the result back in some register.

On some machines the operation of the data path is controlled by a program called a **microprogram**. On other machines the data path is controlled directly by hardware. In early editions of this book, we called this level the "microprogramming level," because in the past it was nearly always a software interpreter. Since the data path is now often (partially) controlled directly by hardware, we changed the name to " microarchitecture level" to reflect this.

On machines with software control of the data path, the microprogram is an interpreter for the instructions at level 2. It fetches, examines, and executes instructions one by one, using the data path to do so. For example, for an ADD instruction, the instruction would be fetched, its operands located and brought into registers, the sum computed by the ALU, and finally the result routed back to the place it belongs. On a machine with hardwired control of the data path, similar steps would take place, but without an explicit stored program to control the interpretation of the level 2 instructions.

We will call level 2 the **Instruction Set Architecture level** (**ISA level**). Every computer manufacturer publishes a manual for each of the computers it sells, entitled "Machine Language Reference Manual," or "Principles of Operation of the Western Wombat Model 100X Computer," or something similar. These manuals are really about the ISA level, not the underlying levels. When they describe the machine's instruction set, they are in fact describing the instructions carried out interpretively by the microprogram or hardware execution circuits. If a computer manufacturer provides two interpreters for one of its machines, interpreting two different ISA levels, it will need to provide two "machine language" reference manuals, one for each interpreter.

The next level is usually a hybrid level. Most of the instructions in its language are also in the ISA level. (There is no reason why an instruction appearing at one level cannot be present at other levels as well.) In addition, there is a set of new instructions, a different memory organization, the ability to run two or more programs concurrently, and various other features. More variation exists between level 3 designs than between those at either level 1 or level 2.

The new facilities added at level 3 are carried out by an interpreter running at level 2, which, historically, has been called an operating system. Those level 3 instructions that are identical to level 2's are executed directly by the microprogram

Spotify USA Inc. Exhibit 1015 Page 10

(or hardwired control), not by the operating system. In other words, some of the level 3 instructions are interpreted by the operating system and some are interpreted directly by the microprogram (or hardwired control). This is what we mean by "hybrid" level. Throughout this book we will call this level the **operating system machine level.**

There is a fundamental break between levels 3 and 4. The lowest three levels are not designed for use by the average garden-variety programmer. Instead, they are intended primarily for running the interpreters and translators needed to support the higher levels. These interpreters and translators are written by people called **systems programmers** who specialize in designing and implementing new virtual machines. Levels 4 and above are intended for the applications programmer with a problem to solve.

Another change occurring at level 4 is the method by which the higher levels are supported. Levels 2 and 3 are always interpreted. Levels 4, 5, and above are usually, although not always, supported by translation.

Yet another difference between levels 1, 2, and 3, on the one hand, and levels 4, 5, and higher, on the other, is the nature of the language provided. The machine languages of levels 1, 2, and 3 are numeric. Programs in them consist of long series of numbers, which are fine for machines but bad for people. Starting at level 4, the languages contain words and abbreviations meaningful to people.

Level 4, the assembly language level, is really a symbolic form for one of the underlying languages. This level provides a method for people to write programs for levels 1, 2, and 3 in a form that is not as unpleasant as the virtual machine languages themselves. Programs in assembly language are first translated to level 1, 2, or 3 language and then interpreted by the appropriate virtual or actual machine. The program that performs the translation is called an **assembler**.

Level 5 usually consists of languages designed to be used by applications programmers with problems to solve. Such languages are often called **high-level languages**. Literally hundreds exist. A few of the better-known ones are C, C++, Java, Perl, Python, and PHP. Programs written in these languages are generally translated to level 3 or level 4 by translators known as **compilers**, although occasionally they are interpreted instead. Programs in Java, for example, are usually first translated to a an ISA-like language called Java byte code, which is then interpreted.

In some cases, level 5 consists of an interpreter for a specific application domain, such as symbolic mathematics. It provides data and operations for solving problems in this domain in terms that people knowledgeable in the domain can understand easily.

In summary, the key thing to remember is that computers are designed as a series of levels, each one built on its predecessors. Each level represents a distinct abstraction, with different objects and operations present. By designing and analyzing computers in this fashion, we are temporarily able to suppress irrelevant detail and thus reduce a complex subject to something easier to understand.

Spotify USA Inc. Exhibit 1015 Page 11

The set of data types, operations, and features of each level is called its **architecture**. The architecture deals with those aspects that are visible to the user of that level. Features that the programmer sees, such as how much memory is available, are part of the architecture. Implementation aspects, such as what kind of technology is used to implement the memory, are not part of the architecture. The study of how to design those parts of a computer system that are visible to the programmers is called **computer architecture**. In common practice, however, computer architecture and computer organization mean essentially the same thing.

1.1.3 Evolution of Multilevel Machines

To provide some perspective on multilevel machines, we will briefly examine their historical development, showing how the number and nature of the levels has evolved over the years. Programs written in a computer's true machine language (level 1) can be directly executed by the computer's electronic circuits (level 0), without any intervening interpreters or translators. These electronic circuits, along with the memory and input/output devices, form the computer's **hardware**. Hardware consists of tangible objects—integrated circuits, printed circuit boards, cables, power supplies, memories, and printers—rather than abstract ideas, algorithms, or instructions.

Software, in contrast, consists of **algorithms** (detailed instructions telling how to do something) and their computer representations—namely, programs. Programs can be stored on hard disk, CD-ROM, or other media, but the essence of software is the set of instructions that makes up the programs, not the physical media on which they are recorded.

In the very first computers, the boundary between hardware and software was crystal clear. Over time, however, it has blurred considerably, primarily due to the addition, removal, and merging of levels as computers have evolved. Nowadays, it is often hard to tell them apart (Vahid, 2003). In fact, a central theme of this book is

Hardware and software are logically equivalent.

Any operation performed by software can also be built directly into the hardware, preferably after it is sufficiently well understood. As Karen Panetta put it: "Hardware is just petrified software." Of course, the reverse is also true: any instruction executed by the hardware can also be simulated in software. The decision to put certain functions in hardware and others in software is based on such factors as cost, speed, reliability, and frequency of expected changes. There are few hard-and-fast rules to the effect that X must go into the hardware and Y must be programmed explicitly. These decisions change with trends in technology economics, demand, and computer usage.

Spotify USA Inc. Exhibit 1015 Page 12

The Invention of Microprogramming

The first digital computers, back in the 1940s, had only two levels: the ISA level, in which all the programming was done, and the digital logic level, which executed these programs. The digital logic level's circuits were complicated, difficult to understand and build, and unreliable.

In 1951, Maurice Wilkes, a researcher at the University of Cambridge, suggested designing a three-level computer in order to drastically simplify the hardware and thus reduce the number of (unreliable) vacuum tubes needed (Wilkes, 1951). This machine was to have a built-in, unchangeable interpreter (the microprogram), whose function was to execute ISA-level programs by interpretation. Because the hardware would now only have to execute microprograms, which have a limited instruction set, instead of ISA-level programs, which have a much larger instruction set, fewer electronic circuits would be needed. Because electronic circuits were then made from vacuum tubes, such a simplification promised to reduce tube count and hence enhance reliability (i.e., the number of crashes per day).

A few of these three-level machines were constructed during the 1950s. More were constructed during the 1960s. By 1970 the idea of having the ISA level be interpreted by a microprogram, instead of directly by the electronics, was dominant. All the major machines of the day used it.

The Invention of the Operating System

In these early years, most computers were "open shop," which meant that the programmer had to operate the machine personally. Next to each machine was a sign-up sheet. A programmer wanting to run a program signed up for a block of time, say Wednesday morning 3 to 5 A.M. (many programmers liked to work when it was quiet in the machine room). When the time arrived, the programmer headed for the machine room with a deck of 80-column punched cards (an early input medium) in one hand and a sharpened pencil in the other. Upon arriving in the computer room, he or she gently nudged the previous programmer toward the door and took over the computer.

If the programmer wanted to run a FORTRAN program, the following steps were necessary:

1. He† went over to the cabinet where the program library was kept, took out the big green deck labeled FORTRAN compiler, put it in the card reader, and pushed the START button.
2. He put his FORTRAN program in the card reader and pushed the CONTINUE button. The program was read in.

† "He" should be read as "he or she" throughout this book.

Spotify USA Inc. Exhibit 1015 Page 13

3. When the computer stopped, he read his FORTRAN program in a second time. Although some compilers required only one pass over the input, many required two or more. For each pass, a large card deck had to be read in.
4. Finally, the translation neared completion. The programmer often became nervous near the end because if the compiler found an error in the program, he had to correct it and start the entire process all over again. If there were no errors, the compiler punched out the translated machine language program on cards.
5. The programmer then put the machine language program in the card reader along with the subroutine library deck and read them both in.
6. The program began executing. More often than not it did not work and unexpectedly stopped in the middle. Generally, the programmer fiddled with the console switches and looked at the console lights for a while. If lucky, he figured out the problem, corrected the error, and went back to the cabinet containing the big green FORTRAN compiler to start over again. If less fortunate, he made a printout of the contents of memory, called a **core dump**, and took it home to study.

This procedure, with minor variations, was normal at many computer centers for years. It forced the programmers to learn how to operate the machine and to know what to do when it broke down, which was often. The machine was frequently idle while people were carrying cards around the room or scratching their heads trying to find out why their programs were not working properly.

Around 1960 people tried to reduce the amount of wasted time by automating the operator's job. A program called an **operating system** was kept in the computer at all times. The programmer provided certain control cards along with the program that were read and carried out by the operating system. Figure 1-3 shows a sample job for one of the first widespread operating systems, FMS (FORTRAN Monitor System), on the IBM 709.

The operating system read the *JOB card and used the information on it for accounting purposes. (The asterisk was used to identify control cards, so they would not be confused with program and data cards.) Later, it read the *FORTRAN card, which was an instruction to load the FORTRAN compiler from a magnetic tape. The compiler then read in and compiled the FORTRAN program. When the compiler finished, it returned control back to the operating system, which then read the *DATA card. This was an instruction to execute the translated program, using the cards following the *DATA card as the data.

Although the operating system was designed to automate the operator's job (hence the name), it was also the first step in the development of a new virtual machine. The *FORTRAN card could be viewed as a virtual "compile program" instruction. Similarly, the *DATA card could be regarded as a virtual "execute

Spotify USA Inc. Exhibit 1015 Page 14

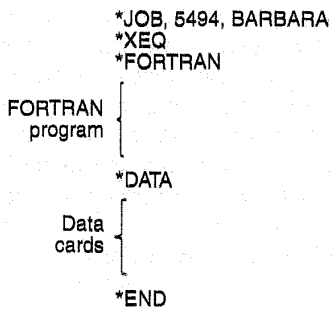


Figure 1-3. A sample job for the FMS operating system.

program" instruction. A level with only two instructions was not much of a level but it was a start in that direction.

In subsequent years, operating systems became more and more sophisticated. New instructions, facilities, and features were added to the ISA level until it began to take on the appearance of a new level. Some of this new level's instructions were identical to the ISA-level instructions, but others, particularly input/output instructions, were completely different. The new instructions were often known as **operating system macros** or **supervisor calls**. The usual term now is **system call**.

Operating systems developed in other ways as well. The early ones read card decks and printed output on the line printer. This organization was known as a **batch system**. Usually, there was a wait of several hours between the time a program was submitted and the time the results were ready. Developing software was difficult under those circumstances.

In the early 1960s researchers at Dartmouth College, M.I.T., and elsewhere developed operating systems that allowed (multiple) programmers to communicate directly with the computer. In these systems, remote terminals were connected to the central computer via telephone lines. The computer was shared among many users. A programmer could type in a program and get the results typed back almost immediately, in the office, in a garage at home, or wherever the terminal was located. These systems were called **timesharing systems**.

Our interest in operating systems is in those parts that interpret the instructions and features present in level 3 and not present in the ISA level rather than in the timesharing aspects. Although we will not emphasize it, you should keep in mind that operating systems do more than just interpret features added to the ISA level.

The Migration of Functionality to Microcode

Once microprogramming had become common (by 1970), designers realized that they could add new instructions by just extending the microprogram. In other words, they could add "hardware" (new machine instructions) by programming.

Spotify USA Inc. Exhibit 1015 Page 15

This revelation led to a virtual explosion in machine instruction sets, as designers competed with one another to produce bigger and better instruction sets. Many of these instructions were not essential in the sense that their effect could be easily achieved by existing instructions, but often they were slightly faster than a sequence of existing instructions. For example, many machines had an instruction INC (INCrement) that added 1 to a number. Since these machines also had a general ADD instruction, having a special instruction to add 1 (or to add 720, for that matter) was not necessary. However, the INC was usually a little faster than the ADD, so it got thrown in.

For the same reason, many other instructions were added to the microprogram. These often included

1. Instructions for integer multiplication and division.
2. Floating-point arithmetic instructions.
3. Instructions for calling and returning from procedures.
4. Instructions for speeding up looping.
5. Instructions for handling character strings.

Furthermore, once machine designers saw how easy it was to add new instructions, they began looking around for other features to add to their microprograms. A few examples of these additions include

1. Features to speed up computations involving arrays (indexing and indirect addressing).
2. Features to permit programs to be moved in memory after they have started running (relocation facilities).
3. Interrupt systems that signal the computer as soon as an input or output operation is completed.
4. The ability to suspend one program and start another in a small number of instructions (process switching).
5. Special instructions for processing audio, image, and multimedia files.

Numerous other features and facilities have been added over the years as well, usually for speeding up some particular activity.

The Elimination of Microprogramming

Microprograms grew fat during the golden years of microprogramming (1960s and 1970s). They also tended to get slower and slower as they acquired more bulk. Finally, some researchers realized that by eliminating the microprogram, vastly

Spotify USA Inc. Exhibit 1015 Page 16

reducing the instruction set, and having the remaining instructions be directly executed (i.e., hardware control of the data path), machines could be speeded up. In a certain sense, computer design had come full circle, back to the way it was before Wilkes invented microprogramming in the first place.

But the wheel is still turning. Modern processors still rely on microprogramming to translate complex instructions to internal microcode that can be executed directly on streamlined hardware.

The point of this discussion is to show that the boundary between hardware and software is arbitrary and constantly changing. Today's software may be tomorrow's hardware, and vice versa. Furthermore, the boundaries between the various levels are also fluid. From the programmer's point of view, how an instruction is actually implemented is unimportant (except perhaps for its speed). A person programming at the ISA level can use its multiply instruction as though it were a hardware instruction without having to worry about it, or even be aware of whether it really is a hardware instruction. One person's hardware is another person's software. We will come back to all these topics later in this book.

1.2 MILESTONES IN COMPUTER ARCHITECTURE

Hundreds of different kinds of computers have been designed and built during the evolution of the modern digital computer. Most have been long forgotten, but a few have had a significant impact on modern ideas. In this section we will give a brief sketch of some of the key historical developments in order to get a better understanding of how we got where we are now. Needless to say, this section only touches on the highlights and leaves many stones unturned. Figure 1-4 lists some of the milestone machines to be discussed in this section. Slater (1987) is a good place to look for additional historical material on the people who founded the computer age. For short biographies and beautiful color photographs by Louis Fabian Bachrach of some of the key people who founded the computer age, see Morgan's coffee-table book (1997).

1.2.1 The Zeroth Generation—Mechanical Computers (1642–1945)

The first person to build a working calculating machine was the French scientist Blaise Pascal (1623–1662), in whose honor the programming language Pascal is named. This device, built in 1642, when Pascal was only 19, was designed to help his father, a tax collector for the French government. It was entirely mechanical, using gears, and powered by a hand-operated crank.

Pascal's machine could do only addition and subtraction operations, but thirty years later the great German mathematician Baron Gottfried Wilhelm von Leibniz (1646–1716) built another mechanical machine that could multiply and divide as

Spotify USA Inc. Exhibit 1015 Page 17

2

COMPUTER SYSTEMS ORGANIZATION

A digital computer consists of an interconnected system of processors, memories, and input/output devices. This chapter is an introduction to these three components and to their interconnection, as background for a more detailed examination of the specific levels in the five subsequent chapters. Processors, memories, and input/output are key concepts and will be present at every level, so we will start our study of computer architecture by looking at all three in turn.

2.1 PROCESSORS

The organization of a simple bus-oriented computer is shown in Fig. 2-1. The **CPU** (**Central Processing Unit**) is the "brain" of the computer. Its function is to execute programs stored in the main memory by fetching their instructions, examining them, and then executing them one after another. The components are connected by a **bus**, which is a collection of parallel wires for transmitting address, data, and control signals. Buses can be external to the CPU, connecting it to memory and I/O devices, but also internal to the CPU, as we will see shortly. Modern computers have multiple buses.

The CPU is composed of several distinct parts. The control unit is responsible for fetching instructions from main memory and determining their type. The arithmetic logic unit performs operations such as addition and Boolean AND needed to carry out the instructions.

Spotify USA Inc. Exhibit 1015 Page 18

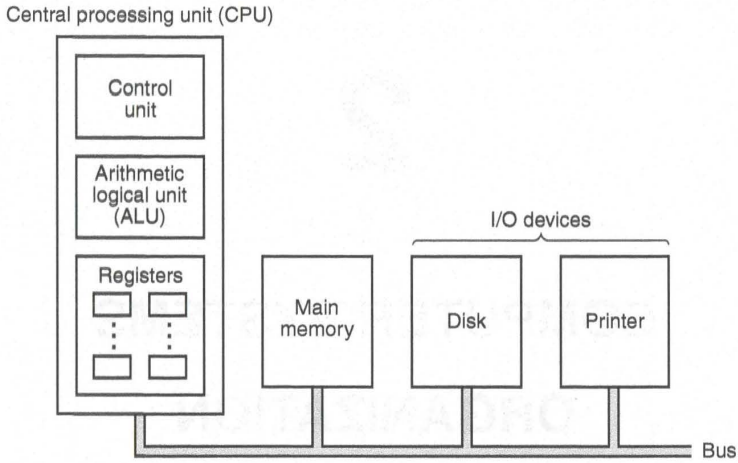


Figure 2-1. The organization of a simple computer with one CPU and two I/O devices.

The CPU also contains a small, high-speed memory used to store temporary results and certain control information. This memory is made up of a number of registers, each having has a certain size and function. Usually, all the registers have the same size. Each register can hold one number, up to some maximum determined by its size. Registers can be read and written at high speed since they are internal to the CPU.

The most important register is the **Program Counter** (**PC**), which points to the next instruction to be fetched for execution. (The name "program counter" is somewhat misleading because it has nothing to do with *counting* anything, but the term is universally used.) Also important is the **Instruction Register** (**IR**), which holds the instruction currently being executed. Most computers have numerous other registers as well, some of them general purpose as well as some for specific purposes. Yet other registers are used by the operating system to control the computer.

2.1.1 CPU Organization

The internal organization of part of a simple von Neumann CPU is shown in Fig. 2-2 in more detail. This part is called the **data path** and consists of the registers (typically 1 to 32), the **ALU** (**Arithmetic Logic Unit**), and several buses connecting the pieces. The registers feed into two ALU input registers, labeled *A* and *B* in the figure. These registers hold the ALU input while the ALU is performing

Spotify USA Inc. Exhibit 1015 Page 19

some computation. The data path is important in all machines and we will discuss it at great length throughout this book.

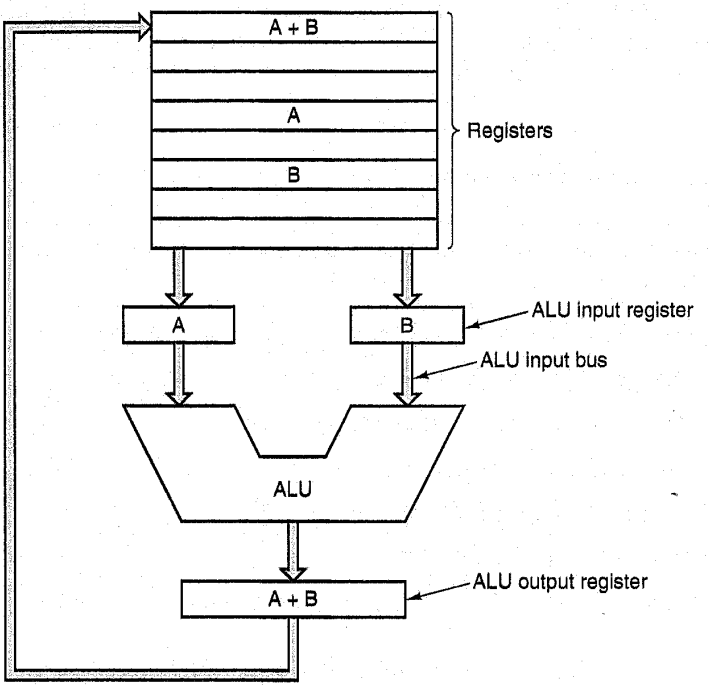


Figure 2-2. The data path of a typical von Neumann machine.

The ALU itself performs addition, subtraction, and other simple operations on its inputs, thus yielding a result in the output register. This output register can be stored back into a register. Later on, the register can be written (i.e., stored) into memory, if desired. Not all designs have the *A*, *B*, and output registers. In the example, addition is illustrated, but ALUs can also perform other operations.

Most instructions can be divided into one of two categories: register-memory or register-register. Register-memory instructions allow memory words to be fetched into registers, where, for example, they can be used as ALU inputs in subsequent instructions. ("Words" are the units of data moved between memory and registers. A word might be an integer. We will discuss memory organization later in this chapter.) Other register-memory instructions allow registers to be stored back into memory.

The other kind of instruction is register-register. A typical register-register instruction fetches two operands from the registers, brings them to the ALU input registers, performs some operation on them (such as addition or Boolean AND),

Spotify USA Inc. Exhibit 1015 Page 20

and stores the result back in one of the registers. The process of running two operands through the ALU and storing the result is called the **data path cycle** and is the heart of most CPUs. To a considerable extent, it defines what the machine can do. Modern computers have multiple ALUs operating in parallel and specialized for different functions. The faster the data path cycle is, the faster the machine runs.

2.1.2 Instruction Execution

The CPU executes each instruction in a series of small steps. Roughly speaking, the steps are as follows:

1. Fetch the next instruction from memory into the instruction register.
2. Change the program counter to point to the following instruction.
3. Determine the type of instruction just fetched.
4. If the instruction uses a word in memory, determine where it is.
5. Fetch the word, if needed, into a CPU register.
6. Execute the instruction.
7. Go to step 1 to begin executing the following instruction.

This sequence of steps is frequently referred to as the **fetch-decode-execute** cycle. It is central to the operation of all computers.

This description of how a CPU works closely resembles a program written in English. Figure 2-3 shows this informal program rewritten as a Java method (i.e., procedure) called *interpret*. The machine being interpreted has two registers visible to user programs: the program counter (PC), for keeping track of the address of the next instruction to be fetched, and the accumulator (AC), for accumulating arithmetic results. It also has internal registers for holding the current instruction during its execution (instr), the type of the current instruction (instr_type), the address of the instruction's operand (data_loc), and the current operand itself (data). Instructions are assumed to contain a single memory address. The memory location addressed contains the operand, for example, the data item to add to the accumulator.

The very fact that it is possible to write a program that can imitate the function of a CPU shows that a program need not be executed by a "hardware" CPU consisting of a box full of electronics. Instead, a program can be carried out by having another program fetch, examine, and execute its instructions. A program (such as the one in Fig. 2-3) that fetches, examines, and executes the instructions of another program is called an **interpreter**, as mentioned in Chap. 1.

Spotify USA Inc. Exhibit 1015 Page 21

```
public class Interp {
  static int PC;                          // program counter holds address of next instr
  static int AC;                          // the accumulator, a register for doing arithmetic
  static int instr;                       // a holding register for the current instruction
  static int instr_type;                  // the instruction type (opcode)
  static int data_loc;                    // the address of the data, or -1 if none
  static int data;                        // holds the current operand
  static boolean run_bit = true;          // a bit that can be turned off to halt the machine

  public static void interpret(int memory[], int starting_address) {
    // This procedure interprets programs for a simple machine with instructions having
    // one memory operand. The machine has a register AC (accumulator), used for
    // arithmetic. The ADD instruction adds an integer in memory to the AC, for example.
    // The interpreter keeps running until the run bit is turned off by the HALT instruction.
    // The state of a process running on this machine consists of the memory, the
    // program counter, the run bit, and the AC. The input parameters consist of
    // the memory image and the starting address.

    PC = starting_address;
    while (run_bit) {
      instr = memory[PC];                 // fetch next instruction into instr
      PC = PC + 1;                        // increment program counter
      instr_type = get_instr_type(instr); // determine instruction type
      data_loc = find_data(instr, instr_type); // locate data (-1 if none)
      if (data_loc >= 0)                  // if data_loc is -1, there is no operand
        data = memory[data_loc];          // fetch the data
      execute(instr_type, data);          // execute instruction
    }

  }

  private static int get_instr_type(int addr) { ... }
  private static int find_data(int instr, int type) { ... }
  private static void execute(int type, int data) { ... }
}
```

Figure 2-3. An interpreter for a simple computer (written in Java).

This equivalence between hardware processors and interpreters has important implications for computer organization and the design of computer systems. After having specified the machine language, *L*, for a new computer, the design team can decide whether they want to build a hardware processor to execute programs in *L* directly or whether they want to write an interpreter to interpret programs in *L* instead. If they choose to write an interpreter, they must also provide some hardware machine to run the interpreter. Certain hybrid constructions are also possible, with some hardware execution as well as some software interpretation.

An interpreter breaks the instructions of its target machine into small steps. As a consequence, the machine on which the interpreter runs can be much simpler and less expensive than a hardware processor for the target machine would be. This

Spotify USA Inc. Exhibit 1015 Page 22

saving is especially significant if the target machine has a large number of instructions and they are fairly complicated, with many options. The saving comes essentially from the fact that hardware is being replaced by software (the interpreter) and it costs more to replicate hardware than software.

Early computers had small, simple sets of instructions. But the quest for more powerful computers led, among other things, to more powerful individual instructions. Very early on, it was discovered that more complex instructions often led to faster program execution even though individual instructions might take longer to execute. A floating-point instruction is an example of a more complex instruction. Direct support for accessing array elements is another. Sometimes it was as simple as observing that the same two instructions often occurred consecutively, so a single instruction could accomplish the work of both.

The more complex instructions were better because the execution of individual operations could sometimes be overlapped or otherwise executed in parallel using different hardware. For expensive, high-performance computers, the cost of this extra hardware could be readily justified. Thus expensive, high-performance computers came to have many more instructions than lower-cost ones. However, instruction compatibility requirements and the rising cost of software development created the need to implement complex instructions even on low-end computers where cost was more important than speed.

By the late 1950s, IBM (then the dominant computer company) had recognized that supporting a single family of machines, all of which executed the same instructions, had many advantages, both for IBM and for its customers. IBM introduced the term **architecture** to describe this level of compatibility. A new family of computers would have one architecture but many different implementations that could all execute the same program, differing only in price and speed. But how to build a low-cost computer that could execute all the complicated instructions of high-performance, expensive machines?

The answer lay in interpretation. This technique, first suggested by Maurice Wilkes (1951), permitted the design of simple, lower-cost computers that could nevertheless execute a large number of instructions. The result was the IBM System/360 architecture, a compatible family of computers, spanning nearly two orders of magnitude, in both price and capability. A direct hardware (i.e., not interpreted) implementation was used only on the most expensive models.

Simple computers with interpreted instructions also some had other benefits. Among the most important were

1. The ability to fix incorrectly implemented instructions in the field, or even make up for design deficiencies in the basic hardware.
2. The opportunity to add new instructions at minimal cost, even after delivery of the machine.
3. Structured design that permitted efficient development, testing, and documenting of complex instructions.

Spotify USA Inc. Exhibit 1015 Page 23

As the market for computers exploded dramatically in the 1970s and computing capabilities grew rapidly, the demand for low-cost computers favored designs of computers using interpreters. The ability to tailor the hardware and the interpreter for a particular set of instructions emerged as a highly cost-effective design for processors. As the underlying semiconductor technology advanced rapidly, the advantages of the cost outweighed the opportunities for higher performance, and interpreter-based architectures became the conventional way to design computers. Nearly all new computers designed in the 1970s, from minicomputers to mainframes, were based on interpretation.

By the late 70s, the use of simple processors running interpreters had become very widespread except among the most expensive, highest-performance models, such as the Cray-1 and the Control Data Cyber series. The use of an interpreter eliminated the inherent cost limitations of complex instructions so designers began to explore much more complex instructions, particularly the ways to specify the operands to be used.

This trend reached its zenith with Digital Equipment Corporation's VAX computer, which had several hundred instructions and more than 200 different ways of specifying the operands to be used in each instruction. Unfortunately, the VAX architecture was conceived from the beginning to be implemented with an interpreter, with little thought given to the implementation of a high-performance model. This mind set resulted in the inclusion of a very large number of instructions which were of marginal value and difficult to execute directly. This omission proved to be fatal to the VAX, and ultimately to DEC as well (Compaq bought DEC in 1998 and Hewlett-Packard bought Compaq in 2001).

Though the earliest 8-bit microprocessors were very simple machines with very simple instruction sets, by the late 70s, even microprocessors had switched to interpreter-based designs. During this period, one of the biggest challenges facing microprocessor designers was dealing with the growing complexity made possible by integrated circuits. A major advantage of the interpreter-based approach was the ability to design a simple processor, with the complexity largely confined to the memory holding the interpreter. Thus a complex hardware design could be turned into a complex software design.

The success of the Motorola 68000, which had a large interpreted instruction set, and the concurrent failure of the Zilog Z8000 (which had an equally large instruction set, but without an interpreter) demonstrated the advantages of an interpreter for bringing a new microprocessor to market quickly. This success was all the more surprising given Zilog's head start (the Z8000's predecessor, the Z80, was far more popular than the 68000's predecessor, the 6800). Of course, other factors were instrumental here, too, not the least of which was Motorola's long history as a chip manufacturer and Exxon's (Zilog's owner) long history of being an oil company, not a chip manufacturer.

Another factor working in favor of interpretation during that era was the existence of fast read-only memories, called **control stores**, to hold the interpreters.

Spotify USA Inc. Exhibit 1015 Page 24

Suppose that a typical interpreted instruction took the interpreter 10 instructions, called **microinstructions**, at 100 nsec each, and two references to main memory, at 500 nsec each. Total execution time was then 2000 nsec, only a factor-of-two worse than the best that direct execution could achieve. Had the control store not been available, the instruction would have taken 6000 nsec. A factor-of-six penalty is a lot harder to swallow than a factor-of-two penalty.

2.1.3 RISC versus CISC

During the late 70s there was experimentation with very complex instructions, made possible by the interpreter. Designers tried to close the "semantic gap" between what machines could do and what high-level programming languages required. Hardly anyone thought about designing simpler machines, just as now not a lot of research goes into designing less powerful spreadsheets, networks, Web servers, etc. (perhaps unfortunately).

One group that bucked the trend and tried to incorporate some of Seymour Cray's ideas in a high-performance minicomputer was led by John Cocke at IBM. This work led to an experimental minicomputer, named the **801**. Although IBM never marketed this machine and the results were not published until years later (Radin, 1982), word got out and other people began investigating similar architectures.

In 1980, a group at Berkeley led by David Patterson and Carlo Séquin began designing VLSI CPU chips that did not use interpretation (Patterson, 1985, Patterson and Séquin, 1982). They coined the term **RISC** for this concept and named their CPU chip the RISC I CPU, followed shortly by the RISC II. Slightly later, in 1981, across the San Francisco Bay at Stanford, John Hennessy designed and fabricated a somewhat different chip he called the **MIPS** (Hennessy, 1984). These chips evolved into commercially important products, the SPARC and the MIPS, respectively.

These new processors were significantly different than commercial processors of the day. Since they did not have to be backward compatible with existing products, their designers were free to choose new instruction sets that would maximize total system performance. While the initial emphasis was on simple instructions that could be executed quickly, it was soon realized that designing instructions that could be **issued** (started) quickly was the key to good performance. How long an instruction actually took mattered less than how many could be started per second.

At the time these simple processors were being first designed, the characteristic that caught everyone's attention was the relatively small number of instructions available, typically around 50. This number was far smaller than the 200 to 300 on established computers such as the DEC VAX and the large IBM mainframes. In fact, the acronym RISC stands for **Reduced Instruction Set Computer**, which was contrasted with CISC, which stands for **Complex Instruction Set Computer** (a thinly veiled reference to the VAX, which dominated university

Spotify USA Inc. Exhibit 1015 Page 25

Computer Science Departments at the time). Nowadays, few people think that the size of the instruction set is a major issue, but the name stuck.

To make a long story short, a great religious war ensued, with the RISC supporters attacking the established order (VAX, Intel, large IBM mainframes). They claimed that the best way to design a computer was to have a small number of simple instructions that execute in one cycle of the data path of Fig. 2-2 by fetching two registers, combining them somehow (e.g., adding or ANDing them), and storing the result back in a register. Their argument was that even if a RISC machine takes four or five instructions to do what a CISC machine does in one instruction, if the RISC instructions are 10 times as fast (because they are not interpreted), RISC wins. It is also worth pointing out that by this time the speed of main memories had caught up to the speed of read-only control stores, so the interpretation penalty had greatly increased, strongly favoring RISC machines.

One might think that given the performance advantages of RISC technology, RISC machines (such as the Sun UltraSPARC) would have mowed down CISC machines (such as the Intel Pentium) in the marketplace. Nothing like this has happened. Why not?

First of all, there is the issue of backward compatibility and the billions of dollars companies have invested in software for the Intel line. Second, surprisingly, Intel has been able to employ the same ideas even in a CISC architecture. Starting with the 486, the Intel CPUs contain a RISC core that executes the simplest (and typically most common) instructions in a single data path cycle, while interpreting the more complicated instructions in the usual CISC way. The net result is that common instructions are fast and less common instructions are slow. While this hybrid approach is not as fast as a pure RISC design, it gives competitive overall performance while still allowing old software to run unmodified.

2.1.4 Design Principles for Modern Computers

Now that more than two decades have passed since the first RISC machines were introduced, certain design principles have come to be accepted as a good way to design computers given the current state of the hardware technology. If a major change in technology occurs (e.g., a new manufacturing process suddenly makes memory cycle time 10 times faster than CPU cycle time), all bets are off. Thus machine designers should always keep an eye out for technological changes that may affect the balance among the components.

That said, there is a set of design principles, sometimes called the **RISC design principles**, that architects of new general-purpose CPUs do their best to follow. External constraints, such as the requirement of being backward compatible with some existing architecture, often require compromises from time to time, but these principles are goals that most designers strive to meet. Next we will discuss the major ones.

Spotify USA Inc. Exhibit 1015 Page 26

All Instructions Are Directly Executed by Hardware

All common instructions are directly executed by the hardware. They are not interpreted by microinstructions. Eliminating a level of interpretation provides high speed for most instructions. For computers that implement CISC instruction sets, the more complex instructions may be broken into separate parts, which can then be executed as a sequence of microinstructions. This extra step slows the machine down, but for less frequently occurring instructions it may be acceptable.

Maximize the Rate at Which Instructions Are Issued

Modern computers resort to many tricks to maximize their performance, chief among which is trying to start as many instructions per second as possible. After all, if you can issue 500 million instructions/sec, you have built a 500-MIPS processor, no matter how long the instructions actually take to complete. (**MIPS** stands for Millions of Instructions Per Second. The MIPS processor was so named as to be a pun on this acronym. Officially it stands for Microprocessor without Interlocked Pipeline Stages.) This principle suggests that parallelism can play a major role in improving performance, since issuing large numbers of slow instructions in a short time interval is possible only if multiple instructions can execute at once.

Although instructions are always encountered in program order, they are not always issued in program order (because some needed resource might be busy) and they need not finish in program order. Of course, if instruction 1 sets a register and instruction 2 uses that register, great care must be taken to make sure that instruction 2 does not read the register until it contains the correct value. Getting this right requires a lot of bookkeeping but has the potential for performance gains by executing multiple instructions at once.

Instructions Should Be Easy to Decode

A critical limit on the rate of issue of instructions is decoding individual instructions to determine what resources they need. Anything that can aid this process is useful. That includes making instructions regular, of fixed length, and with a small number of fields. The fewer different formats for instructions, the better.

Only Loads and Stores Should Reference Memory

One of the simplest ways to break operations into separate steps is to require that operands for most instructions come from—and return to—CPU registers. The operation of moving operands from memory into registers can be performed in separate instructions. Since access to memory can take a long time, and the delay is unpredictable, these instructions can best be overlapped with other instructions assuming they do nothing except move operands between registers and memory.

Spotify USA Inc. Exhibit 1015 Page 27

This observation means that only LOAD and STORE instructions should reference memory. All other instructions should operate only on registers.

Provide Plenty of Registers

Since accessing memory is relatively slow, many registers (at least 32) need to be provided, so that once a word is fetched, it can be kept in a register until it is no longer needed. Running out of registers and having to flush them back to memory only to later reload them is undesirable and should be avoided as much as possible. The best way to accomplish this is to have enough registers.

2.1.5 Instruction-Level Parallelism

Computer architects are constantly striving to improve performance of the machines they design. Making the chips run faster by increasing their clock speed is one way, but for every new design, there is a limit to what is possible by brute force at that moment in history. Consequently, most computer architects look to parallelism (doing two or more things at once) as a way to get even more performance for a given clock speed.

Parallelism comes in two general forms, namely, instruction-level parallelism and processor-level parallelism. In the former, parallelism is exploited within individual instructions to get more instructions/sec out of the machine. In the latter, multiple CPUs work together on the same problem. Each approach has its own merits. In this section we will look at instruction-level parallelism; in the next one, we will look at processor-level parallelism.

Pipelining

It has been known for years that the actual fetching of instructions from memory is a major bottleneck in instruction execution speed. To alleviate this problem, computers going back at least as far as the IBM Stretch (1959) have had the ability to fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a special set of registers called the **prefetch buffer**. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution into two parts: fetching and actual execution. The concept of a **pipeline** carries this strategy much further. Instead of being divided into only two parts, instruction execution is often divided into many (often a dozen or more) parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

Figure 2-4(a) illustrates a pipeline with five units, also called **stages**. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs.

Spotify USA Inc. Exhibit 1015 Page 28

Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path of Fig. 2-2. Finally, stage 5 writes the result back to the proper register.

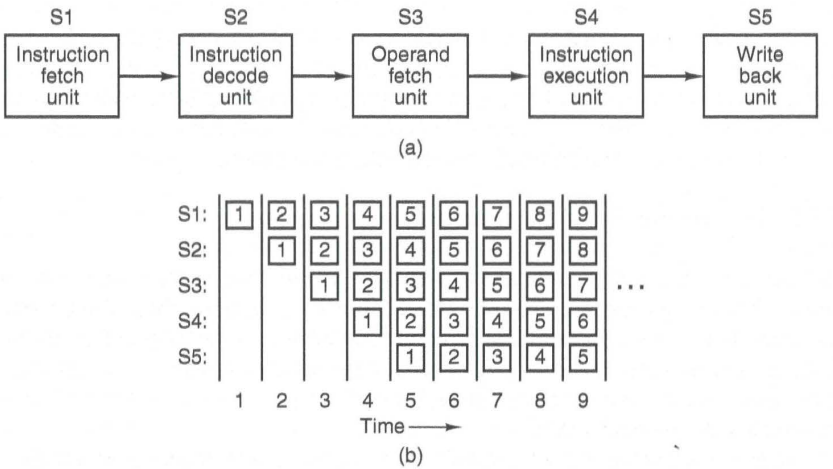


Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

In Fig. 2-4(b) we see how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction 2, and stage S1 fetches the third instruction. During cycle 4, stage S4 executes instruction 1, S3 fetches the operands for instruction 2, S2 decodes instruction 3, and S1 fetches instruction 4. Finally, in cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

Let us consider an analogy to clarify the concept of pipelining. Imagine a cake factory in which the baking of the cakes and the packaging of the cakes for shipment are separated. Suppose that the shipping department has a long conveyor belt with five workers (processing units) lined up along it. Every 10 sec (the clock cycle), worker 1 places an empty cake box on the belt. The box is carried down to worker 2, who places a cake in it. A little later, the box arrives at worker 3's station, where it is closed and sealed. Then it continues to worker 4, who puts a label on the box. Finally, worker 5 removes the box from the belt and puts it in a large container for later shipment to a supermarket. Basically, this is the way computer pipelining works, too: each instruction (cake) goes through several processing steps before emerging completed at the far end.

Spotify USA Inc. Exhibit 1015 Page 29

Getting back to our pipeline of Fig. 2-4, suppose that the cycle time of this machine is 2 nsec. Then it takes 10 nsec for an instruction to progress all the way through the five-stage pipeline. At first glance, with an instruction taking 10 nsec, it might appear that the machine can run at 100 MIPS, but in fact it does much better than this. At every clock cycle (2 nsec), one new instruction is completed, so the actual rate of processing is 500 MIPS, not 100 MIPS.

Pipelining allows a trade-off between **latency** (how long it takes to execute an instruction), and **processor bandwidth** (how many MIPS the CPU has). With a cycle time of T nsec, and n stages in the pipeline, the latency is nT nsec because each instruction passes through n stages, each of which takes T nsec.

Since one instruction completes every clock cycle and there are $10^9/T$ clock cycles/second, the number of instructions executed per second is $10^9/T$. For example, if T = 2 nsec, 500 million instructions are executed each second. To get the number of MIPS, we have to divide the instruction execution rate by 1 million to get $(10^9/T)/10^6 = 1000/T$ MIPS. Theoretically, we could measure instruction execution rate in BIPS instead of MIPS, but nobody does that, so we will not either.

Superscalar Architectures

If one pipeline is good, then surely two pipelines are better. One possible design for a dual pipeline CPU, based on Fig. 2-4, is shown in Fig. 2-5. Here a single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline, complete with its own ALU for parallel operation. To be able to run in parallel, the two instructions must not conflict over resource usage (e.g., registers), and neither must depend on the result of the other. As with a single pipeline, either the compiler must guarantee this situation to hold (i.e., the hardware does not check and gives incorrect results if the instructions are not compatible), or conflicts must be detected and eliminated during execution using extra hardware.

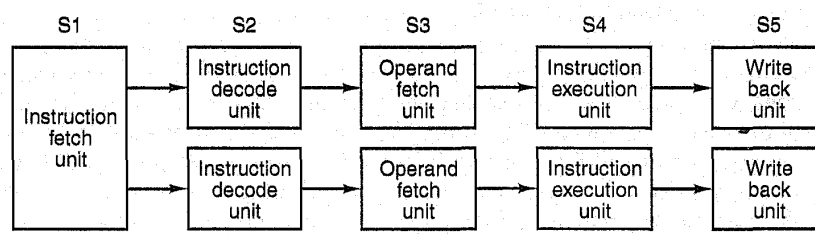


Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.

Although pipelines, single or double, were originally used on RISC machines (the 386 and its predecessors did not have any), starting with the 486 Intel began

Spotify USA Inc. Exhibit 1015 Page 30

introducing data pipelines into its CPUs. The 486 had one pipeline and the original Pentium had two five-stage pipelines roughly as in Fig. 2-5, although the exact division of work between stages 2 and 3 (called decode-1 and decode-2) was slightly different than in our example. The main pipeline, called the **u pipeline**, could execute an arbitrary Pentium instruction. The second pipeline, called the **v pipeline**, could execute only simple integer instructions (and also one simple floating-point instruction—FXCH).

Fixed rules determined whether a pair of instructions were compatible so they could be executed in parallel. If the instructions in a pair were not simple enough or incompatible, only the first one was executed (in the u pipeline). The second one was then held and paired with the instruction following it. Instructions were always executed in order. Thus Pentium-specific compilers that produced compatible pairs could produce faster-running programs than older compilers. Measurements showed that a Pentium running code optimized for it was exactly twice as fast on integer programs as a 486 running at the same clock rate (Pountain, 1993). This gain could be attributed entirely to the second pipeline.

Going to four pipelines is conceivable, but doing so duplicates too much hardware (computer scientists, unlike folklore specialists, do not believe in the number three). Instead, a different approach is used on high-end CPUs. The basic idea is to have just a single pipeline but give it multiple functional units, as shown in Fig. 2-6. For example, the Intel Core architecture has a structure similar to this figure. It will be discussed in Chap. 4. The term **superscalar architecture** was coined for this approach in 1987 (Agerwala and Cocke, 1987). Its roots, however, go back more than 40 years to the CDC 6600 computer. The 6600 fetched an instruction every 100 nsec and passed it off to one of 10 functional units for parallel execution while the CPU went off to get the next instruction.

The definition of "superscalar" has evolved somewhat over time. It is now used to describe processors that issue multiple instructions—often four or six—in a single clock cycle. Of course, a superscalar CPU must have multiple functional units to hand all these instructions to. Since superscalar processors generally have one pipeline, they tend to look like Fig. 2-6.

Using this definition, the 6600 was technically not superscalar because it issued only one instruction per cycle. However, the effect was almost the same: instructions were issued at a much higher rate than they could be executed. The conceptual difference between a CPU with a 100-nsec clock that issues one instruction every cycle to a group of functional units and a CPU with a 400-nsec clock that issues four instructions per cycle to the same group of functional units is very small. In both cases, the key idea is that the issue rate is much higher than the execution rate, with the workload being spread across a collection of functional units.

Implicit in the idea of a superscalar processor is that the S3 stage can issue instructions considerably faster than the S4 stage is able to execute them. If the S3 stage issued an instruction every 10 nsec and all the functional units could do their work in 10 nsec, no more than one would ever be busy at once, negating the whole

Spotify USA Inc. Exhibit 1015 Page 31

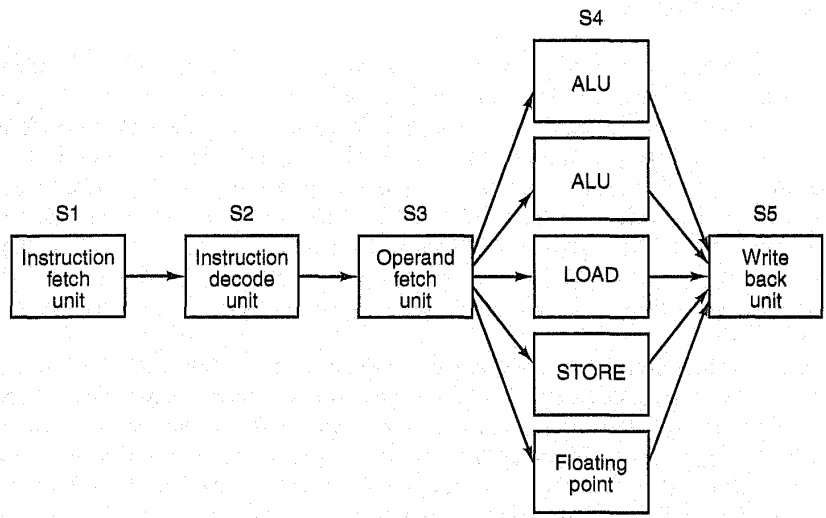


Figure 2-6. A superscalar processor with five functional units.

idea. In reality, most of the functional units in stage 4 take appreciably longer than one clock cycle to execute, certainly the ones that access memory or do floating-point arithmetic. As can be seen from the figure, it is possible to have multiple ALUs in stage S4.

2.1.6 Processor-Level Parallelism

The demand for ever faster computers seems to be insatiable. Astronomers want to simulate what happened in the first microsecond after the big bang, economists want to model the world economy, and teenagers want to play 3D interactive multimedia games over the Internet with their virtual friends. While CPUs keep getting faster, eventually they are going to run into the problems with the speed of light, which is likely to stay at 20 cm/nanosecond in copper wire or optical fiber, no matter how clever Intel's engineers are. Faster chips also produce more heat, whose dissipation is a huge problem. In fact, the difficulty of getting rid of the heat produced is the main reason CPU clock speeds have stagnated in the past decade.

Instruction-level parallelism helps a little, but pipelining and superscalar operation rarely win more than a factor of five or ten. To get gains of 50, 100, or more, the only way is to design computers with multiple CPUs, so we will now take a look at how some of these are organized.

Spotify USA Inc. Exhibit 1015 Page 32

Data Parallel Computers

A substantial number of problems in computational domains such as the physical sciences, engineering, and computer graphics involve loops and arrays, or otherwise have a highly regular structure. Often the same calculations are performed repeatedly on many different sets of data. The regularity and structure of these programs makes them especially easy targets for speed-up through parallel execution. Two primary methods have been used to execute these highly regular programs quickly and efficiently: SIMD processors and vector processors. While these two schemes are remarkably similar in most ways, ironically, the first is generally thought of as a parallel computer while the second is considered an ex extension to a single processor.

Data parallel computers have found many successful applications as a consequence of their remarkable efficiency. They are able to produce significant computational power with fewer transistors than alternative approaches. Gordon Moore (of Moore's law) famously noted that silicon costs about $1 billion per acre (4047 square meters). Thus, the more computational muscle that can be squeezed out of that acre of silicon, the more money a computer company can make selling silicon. Data parallel processors are one of the most efficient means to squeeze performance out of silicon. Because all of the processors are running the same instruction, the system needs only one "brain" controlling the computer. Consequently, the processor needs only one fetch stage, one decode stage, and one set of control logic. This is a huge saving in silicon that gives data parallel computers a big edge over other processors, as long as the software they are running is highly regular with lots of parallelism.

A **Single Instruction-stream Multiple Data-stream** or **SIMD processor** consists of a large number of identical processors that perform the same sequence of instructions on different sets of data. The world's first SIMD processor was the University of Illinois ILLIAC IV computer (Bouknight et al., 1972). The original ILLIAC IV design consisted of four quadrants, each quadrant having an 8×8 square grid of processor/memory elements. A single control unit per quadrant broadcast a single instruction to all processors, which was executed by all processors in lockstep each using its own data from its own memory. Owing to funding constraints only one 50 megaflops (million floating-point operations per second) quadrant was ever built; had the entire 1-gigaflop machine been completed, it would have doubled the computing power of the entire world.

Modern graphics processing units (GPUs) heavily rely on SIMD processing to provide massive computational power with few transistors. Graphics processing lends itself to SIMD processors because most of the algorithms are highly regular, with repeated operations on pixels, vertices, textures, and edges. Fig. 2-7 shows the SIMD processor at the core of the Nvidia Fermi GPU. A Fermi GPU contains up to 16 SIMD stream multiprocessors (SM), with each SM containing 32 SIMD processors. Each cycle, the scheduler selects two threads to execute on the SIMD

Spotify USA Inc. Exhibit 1015 Page 33

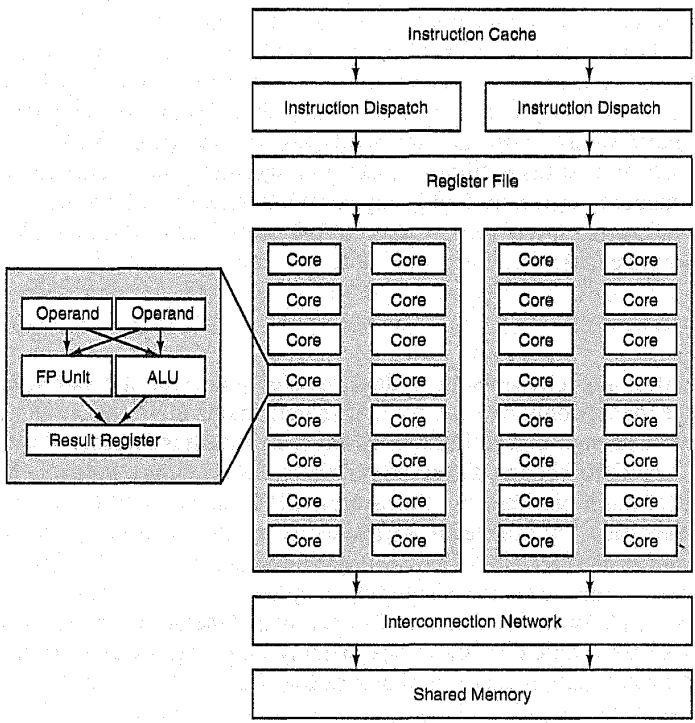


Figure 2-7. The SIMD core of the Fermi graphics processing unit.

processor. The next instruction from each thread then executes on up to 16 SIMD processors, although possibly fewer if there is not enough data parallelism. If each thread is able to perform 16 operations per cycle, a fully loaded Fermi GPU core with 32 SMs will perform a whopping 512 operations per cycle. This is an impressive feat considering that a similar-sized general purpose quad-core CPU would struggle to achieve 1/32 as much processing.

A **vector processor** appears to the programmer very much like a SIMD processor. Like a SIMD processor, it is very efficient at executing a sequence of operations on pairs of data elements. But unlike a SIMD processor, all of the operations are performed in a single, heavily pipelined functional unit. The company Seymour Cray founded, Cray Research, produced many vector processors, starting with the Cray-1 back in 1974 and continuing through current models.

Both SIMD processors and vector processors work on arrays of data. Both execute single instructions that, for example, add the elements together pairwise for two vectors. But while the SIMD processor does it by having as many adders as elements in the vector, the vector processor has the concept of a **vector register**,

Spotify USA Inc. Exhibit 1015 Page 34

which consists of a set of conventional registers that can be loaded from memory in a single instruction, which actually loads them from memory serially. Then a vector addition instruction performs the pairwise addition of the elements of two such vectors by feeding them to a pipelined adder from the two vector registers. The result from the adder is another vector, which can either be stored into a vector register or used directly as an operand for another vector operation. The SSE (Streaming SIMD Extension) instructions available on the Intel Core architecture use this execution model to speed up highly regular computation, such as multimedia and scientific software. In this respect, the Intel Core architecture has the ILLIAC IV as one of its ancestors.

Multiprocessors

The processing elements in a data parallel processor are not independent CPUs, since there is only one control unit shared among all of them. Our first parallel system with multiple full-blown CPUs is the **multiprocessor**, a system with more than one CPU sharing a common memory, like a group of people in a room sharing a common blackboard. Since each CPU can read or write any part of memory, they must coordinate (in software) to avoid getting in each other's way. When two or more CPUs have the ability to interact closely, as is the case with multiprocessors, they are said to be tightly coupled.

Various implementation schemes are possible. The simplest one is to have a single bus with multiple CPUs and one memory all plugged into it. A diagram of such a bus-based multiprocessor is shown in Fig. 2-8(a).

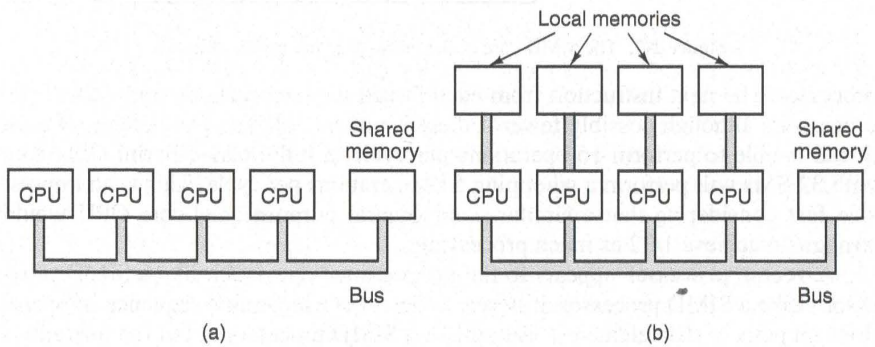


Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

It does not take much imagination to realize that with a large number of fast processors constantly trying to access memory over the same bus, conflicts will result. Multiprocessor designers have come up with various schemes to reduce this

Spotify USA Inc. Exhibit 1015 Page 35

contention and improve performance. One design, shown in Fig. 2-8(b), gives each processor some local memory of its own, not accessible to the others. This memory can be used for program code and those data items that need not be shared. Access to this private memory does not use the main bus, greatly reducing bus traffic. Other schemes (e.g., caching—see below) are also possible.

Multiprocessors have the advantage over other kinds of parallel computers that the programming model of a single shared memory is easy to work with. For example, imagine a program looking for cancer cells in a photograph of some tissue taken through a microscope. The digitized photograph could be kept in the common memory, with each processor assigned some region of the photograph to hunt in. Since each processor has access to the entire memory, studying a cell that starts in its assigned region but straddles the boundary into the next region is no problem.

Multicomputers

Although multiprocessors with a modest number of processors (≤ 256) are relatively easy to build, large ones are surprisingly difficult to construct. The difficulty is in connecting so many the processors to the memory. To get around these problems, many designers have simply abandoned the idea of having a shared memory and just build systems consisting of large numbers of interconnected computers, each having its own private memory, but no common memory. These systems are called **multicomputers**. The CPUs in a multicomputer are said to be **loosely coupled**, to contrast them with the **tightly coupled** multiprocessor CPUs.

The CPUs in a multicomputer communicate by sending each other messages, something like email, but much faster. For large systems, having every computer connected to every other computer is impractical, so topologies such as 2D and 3D grids, trees, and rings are used. As a result, messages from one computer to another often must pass through one or more intermediate computers or switches to get from the source to the destination. Nevertheless, message-passing times on the order of a few microseconds can be achieved without much difficulty. Multicomputers with over 250,000 CPUs, such as IBM's Blue Gene/P, have been built.

Since multiprocessors are easier to program and multicomputers are easier to build, there is much research on designing hybrid systems that combine the good properties of each. Such computers try to present the illusion of shared memory without going to the expense of actually constructing it. We will go into multiprocessors and multicomputers in detail in Chap. 8.

2.2 PRIMARY MEMORY

The **memory** is that part of the computer where programs and data are stored. Some computer scientists (especially British ones) use the term **store** or **storage** rather than memory, although more and more, the term "storage" is used to refer

Spotify USA Inc. Exhibit 1015 Page 36