that served only a few users. Fortunately, silicon fabrication companies could still produce faster, lower-power and less expensive chips for applications that needed millions of chips. But, for applications with only a few users, such as prototyping, low-volume design applications, and education, FPGAs remain a popular tool for building hardware.

Up until 1992, personal computers were either 8-bit, 16-bit, or 32-bit. Then DEC came out with the revolutionary 64-bit Alpha, a true 64-bit RISC machine that outperformed all other personal computers by a wide margin. It had a modest success, but almost a decade elapsed before 64-bit machines began to catch on in a big way, and then mostly as high-end servers.

Throughout the 1990s computing systems were getting faster and faster using a variety of microarchitectural optimizations, many of which we will examine in this book. Users of these systems were pampered by computer vendors, because each new system they bought would run their programs much faster than their old system. However, by the end of the 1990s this trend was beginning to wane because of two important obstacles in computer design: architects were running out of tricks to make programs faster, and the processors were getting too expensive to cool. Desperate to continue building faster processors, most computer companies began turning toward parallel architectures as a way to squeeze out more performance from their silicon. In 2001 IBM introduced the POWER4 dual-core architecture. This was the first time that a mainstream CPU incorporated two processors onto the same die. Today, most desktop and server class processors, and even some embedded processors, incorporate multiple processors on chip. The performance of these multiprocessors has unfortunately been less than stellar for the typical user, because (as we will see in later chapters) parallel machines require programmers to explicitly parallelize programs, which is a difficult and error-prone task.

1.2.6 The Fifth Generation—Low-Power and Invisible Computers

In 1981, the Japanese government announced that they were planning to spend $500 million to help Japanese companies develop fifth-generation computers, which would be based on artificial intelligence and represent a quantum leap over "dumb" fourth-generation computers. Having seen Japanese companies take over the market in many industries, from cameras to stereos to televisions, American and European computer makers went from 0 to full panic in a millisecond, demanding government subsidies and more. Despite lots of fanfare, the Japanese fifth-generation project basically failed and was quietly abandoned. In a sense, it was like Babbage's analytical engine—a visionary idea but so far ahead of its time that the technology for actually building it was nowhere in sight.

Nevertheless, what might be called the fifth generation did happen, but in an unexpected way: computers shrank. In 1989, Grid Systems released the first tablet computer, called the GridPad. It consisted of a small screen on which the users could write with a special pen to control the system. Systems such as the GridPad

showed that computers did not need to sit on a desk or in a server room, but instead, could be put into an easy-to-carry package with touchscreens and handwriting recognition to make them even more valuable.

The Apple Newton, released in 1993, showed that a computer could be built in a package no bigger than a portable audiocassette player. Like the GridPad, the Newton used handwriting for user input, which in this case proved to be a big stumbling block to its success. However, later machines of this class, now called **PDAs** (**Personal Digital Assistants**), have improved user interfaces and are very popular. They have now evolved into **smartphones**.

Eventually, the writing interface of the PDA was perfected by Jeff Hawkins, who had created a company called Palm to develop a low-cost PDA for the mass consumer market. Hawkins was an electrical engineer by training, but he had a keen interest in neuroscience, which is the study of the human brain. He realized that handwriting recognition could be made more reliable by training users to write in a manner that was more easily readable by computers, an input technique he called "Graffiti." It required a small amount of training for the user, but in the end it led to faster and more reliable writing, and the first Palm PDA, called the Palm Pilot, was a huge success. Graffiti is one of the great successes in computing, demonstrating the power of the human mind to take advantage of the power of the human mind.

Users of PDAs swore by the devices, religiously using them to manage their schedules and contacts. When cell phones started gaining popularity in the early 1990s, IBM jumped at the opportunity to integrate the cell phone with the PDA, creating the "smartphone." The first smartphone, called **Simon**, used a touchscreen for input, and it gave the user all of the capabilities of a PDA plus telephone, games, and email. Shrinking component sizes and cost eventually led to the wide use of smartphones, embodied in the popular Apple iPhone and Google Android platforms.

But even the PDAs and smartphones are not really revolutionary. Even more important are the "invisible" computers, which are embedded into appliances, watches, bank cards, and numerous other devices (Bechini et al., 2004). These processors allow increased functionality and lower cost in a wide variety of applications. Whether these chips form a true generation is debatable (they have been around since the 1970s), but they are revolutionizing how thousands of appliances and other devices work. They are already starting to have a major impact on the world and their influence will increase rapidly in the coming years. One unusual aspects of these embedded computers is that the hardware and software are often **codesigned** (Henkel et al., 2003). We will come back to them later in this book.

If we see the first generation as vacuum-tube machines (e.g. ENIAC), the second generation as transistor machines (e.g., the IBM 7094), the third generation as early integrated-circuit machines (e.g., the IBM 360), and the fourth generation as personal computers (e.g., the Intel CPUs), the real fifth generation is more a paradigm shift than a specific new architecture. In the future, computers will be

everywhere and embedded in everything—indeed, invisible. They will be part of the framework of daily life, opening doors, turning on lights, dispensing money, and doing thousands of other things. This model, devised by Mark Weiser, was originally called **ubiquitous computing**, but the term **pervasive computing** is also used frequently now (Weiser, 2002). It will change the world as profoundly as the industrial revolution did. We will not discuss it further in this book, but for more information about it, see Lyytinen and Yoo (2002), Saha and Mukherjee (2003), and Sakamura (2002).

1.3 THE COMPUTER ZOO

In the previous section, we gave a very brief history of computer systems. In this one we will look at the present and gaze toward the future. Although personal computers are the best known computers, there are other kinds of machines around these days, so it is worth taking a brief look at what else is out there.

1.3.1 Technological and Economic Forces

The computer industry is moving ahead like no other. The primary driving force is the ability of chip manufacturers to pack more and more transistors per chip every year. More transistors, which are tiny electronic switches, means larger memories and more powerful processors. Gordon Moore, co-founder and former chairman of Intel, once joked that if aviation technology had moved ahead as fast as computer technology, an airplane would cost $500 and circle the earth in 20 minutes on 5 gallons of fuel. However, it would be the size of a shoebox.

Specifically, while preparing a speech for an industry group, Moore noticed that each new generation of memory chips was being introduced 3 years after the previous one. Since each new generation had four times as much memory as its predecessor, he realized that the number of transistors on a chip was increasing at a constant rate and predicted this growth would continue for decades to come. This observation has become known as **Moore's law**. Today, Moore's law is often expressed as the doubling of the number of transistors every 18 months. Note that this is equivalent to about a 60 percent increase in transistor count per year. The sizes of the memory chips and their dates of introduction shown in Fig. 1-8 confirm that Moore's law has held for over four decades.

Of course, Moore's law is not a law at all, but simply an empirical observation about how fast solid-state physicists and process engineers are advancing the state of the art, and a prediction that they will continue at the same rate in the future. Some industry observers expect Moore's law to continue to hold for at least another decade, maybe longer. Other observers expect energy dissipation, current leakage, and other effects to kick in earlier and cause serious problems that need to be

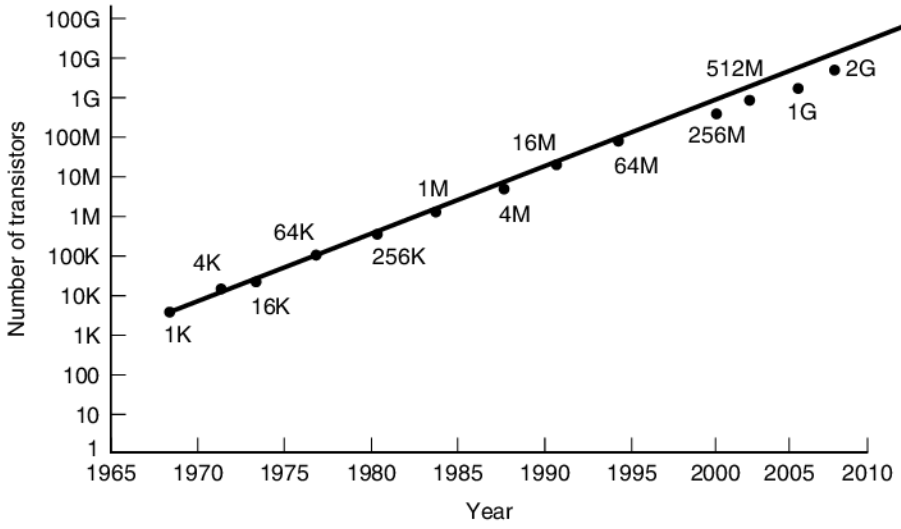

Figure 1-8. Moore's law predicts a 60 percent annual increase in the number of transistors that can be put on a chip. The data points given above and below the line are memory sizes, in bits.

solved (Bose, 2004, Kim et al., 2003). However, the reality of shrinking transistors is that the thickness of these devices is soon to be only a few atoms. At that point transistors will consist of too few atoms to be reliable, or we will simply reach a point where further size decreases will require subatomic building blocks. (As a matter of good advice, it is recommended that anyone working in a silicon fabrication plant take the day off on the day they decide to split the one-atom transistor!) Despite the many challenges in extending Moore's law trends, there are hopeful technologies on the horizon, including advances in quantum computing (Oskin et al., 2002) and carbon nanotubes (Heinze et al., 2002) that may create opportunities to scale electronics beyond the limits of silicon.

Moore's law has created what economists call a **virtuous circle**. Advances in technology (transistors/chip) lead to better products and lower prices. Lower prices lead to new applications (nobody was making video games for computers when computers cost $10 million each although when the price dropped to $120,000 M.I.T. students took up the challenge). New applications lead to new markets and new companies springing up to take advantage of them. The existence of all these companies leads to competition, which in turn creates economic demand for better technologies with which to beat the others. The circle is then round.

Another factor driving technological improvement is Nathan's first law of software (due to Nathan Myhrvold, a former top Microsoft executive). It states: "Software is a gas. It expands to fill the container holding it." Back in the 1980s, word

processing was done with programs like troff (still used for this book). Troff occupies kilobytes of memory. Modern word processors occupy many megabytes of memory. Future ones will no doubt require gigabytes of memory. (To a first approximation, the prefixes kilo, mega, giga, and tera mean thousand, million, billion, and trillion, respectively, but see Sec. 1.5 for details.) Software that continues to acquire features (not unlike boats that continue to acquire barnacles) creates a constant demand for faster processors, bigger memories, and more I/O capacity.

While the gains in transistors per chip have been dramatic over the years, the gains in other computer technologies have been hardly less so. For example, the IBM PC/XT was introduced in 1982 with a 10-megabyte hard disk. Thirty years later, 1-TB hard disks are common on the PC/XT's successors. This improvement of five orders of magnitude in 30 years represents an annual capacity increase of nearly 50 percent. However, measuring disk improvement is trickier, since there are other parameters besides capacity, such as data rate, seek time, and price. Nevertheless, almost any metric will show that the price/performance ratio has increased since 1982 by about 50 percent per year. These enormous gains in disk performance, coupled with the fact that the dollar volume of disks shipped from Silicon Valley has exceeded that of CPU chips, led Al Hoagland to suggest that the place was named wrong: it should have been called Iron Oxide Valley (since this is the recording medium used on disks). Slowly this trend is shifting back in favor of silicon as silicon-based flash memories have begun to replace traditional spinning disks in many systems.

Another area that has seen spectacular gains has been telecommunication and networking. In less than two decades, we have gone from 300 bit/sec modems to analog modems at 56,000 bits/sec to fiber-optic networks at 10^{12} bits/sec. Fiber-optic transatlantic telephone cables, such as TAT-12/13, cost about $700 million, last for 10 years, and can carry 300,000 simultaneous calls, which comes to under 1 cent for a 10-minute intercontinental call. Optical communication systems running at 10^{12} bits/sec over distances exceeding 100 km without amplifiers have been proven feasible. The exponential growth of the Internet hardly needs comment here.

1.3.2 The Computer Spectrum

Richard Hamming, a former researcher at Bell Labs, once observed that a change of an order of magnitude in quantity causes a change in quality. Thus, a racing car that can go 1000 km/hour in the Nevada desert is a fundamentally different kind of machine than a normal car that goes 100 km/hour on a highway. Similarly, a 100-story skyscraper is not just a scaled up 10-story apartment building. And with computers, we are not talking about factors of 10, but over the course of four decades, factors of a million.

The gains afforded by Moore's law can be used by chip vendors in several different ways. One way is to build increasingly powerful computers at constant

price. Another approach is to build the same computer for less and less money every year. The computer industry has done both of these and more, so that a wide variety of computers are available now. A very rough categorization of current computers is given in Fig. 1-9.

Type	Price ($)	Example application
Disposable computer	0.5	Greeting cards
Microcontroller	5	Watches, cars, appliances
Mobile and game computers	50	Home video games and smartphones
Personal computer	500	Desktop or notebook computer
Server	5K	Network server
Mainframe	5M	Batch data processing in a bank

Figure 1-9. The current spectrum of computers available. The prices should be taken with a grain (or better yet, a metric ton) of salt.

In the following sections we will examine each of these categories and discuss their properties briefly.

1.3.3 Disposable Computers

At the bottom end, we find single chips glued to the inside of greeting cards for playing "Happy Birthday," "Here Comes the Bride," or some equally appalling ditty. The authors have not yet spotted a condolence card that plays a funeral dirge, but having now released this idea into the public domain, we expect it shortly. To anyone who grew up with multimillion-dollar mainframes, the idea of disposable computers makes about as much sense as disposable aircraft.

However, disposable computers are here to stay. Probably the most important development in the area of throwaway computers is the **RFID** (**Radio Frequency IDentification**) chip. It is now possible to manufacture, for a few cents, battery-less RFID chips smaller than 0.5 mm on edge that contain a tiny radio transponder and a built-in unique 128-bit number. When pulsed from an external antenna, they are powered by the incoming radio signal long enough to transmit their number back to the antenna. While the chips are tiny, their implications are certainly not.

Let us start with a mundane application: removing bar codes from products. Experimental trials have already been held in which products in stores have RFID chips (instead of bar codes) attached by the manufacturer. Customers select their products, put them in a shopping cart, and just wheel them out of the store, bypassing the checkout counter. At the store's exit, a reader with an antenna sends out a signal asking each product to identify itself, which it does by a short wireless transmission. Customers are also identified by chips on their debit or credit card. At the end of the month, the store sends each customer an itemized bill for this month's purchases. If the customer does not have a valid RFID bank or credit

card, an alarm is sounded. Not only does this system eliminate the need for cashiers and the corresponding wait in line, but it also serves as an antitheft system because hiding a product in a pocket or bag has no effect.

An interesting property of this system is that while bar codes identify the product type, they do not identify the specific item. With 128 bits available, RFID chips do. As a consequence, every package of, say, aspirins, on a supermarket shelf will have a different RFID code. This means that if a drug manufacturer discovers a manufacturing defect in a batch of aspirins after they have been shipped, supermarkets all over the world can be told to sound the alarm when a customer buys any package whose RFID number lies in the affected range, even if the purchase happens in a distant country months later. Aspirins not in the defective batch will not sound the alarm.

But labeling packages of aspirins, cookies, and dog biscuits is only the start. Why stop at labeling the dog biscuits when you can label the dog? Pet owners are already asking veterinarians to implant RFID chips in their animals, allowing them to be traced if they are stolen or lost. Farmers want their livestock tagged as well. The obvious next step is for nervous parents to ask their pediatrician to implant RFID chips in their children in case they get stolen or lost. While we are at it, why not have hospitals put them in all newborns to avoid mixups at the hospital? Governments and the police can no doubt think of many good reasons for tracking all citizens all the time. By now, the "implications" of RFID chips alluded to earlier may be getting a bit clearer.

Another (slightly less controversial) application of RFID chips is vehicle tracking. When a string of railroad cars with embedded RFID chips passes by a reader, the computer attached to the reader then has a list of which cars passed by. This system makes it easy to keep track of the location of all railroad cars, which helps suppliers, their customers, and the railroads. A similar scheme can be applied to trucks. For cars, the idea is already being used to collect tolls electronically (e.g., the E-Z Pass system).

Airline baggage systems and many other package transport systems can also use RFID chips. An experimental system tested at Heathrow airport in London allowed arriving passengers to remove the lugging from their luggage. Bags carried by passengers purchasing this service were tagged with RFID chips, routed separately within the airport, and delivered directly to the passengers' hotels. Other uses of RFID chips include having cars arriving at the painting station of the assembly line specify what color they are supposed to be, studying animal migrations, having clothes tell the washing machine what temperature to use, and many more. Some chips may be integrated with sensors so that the low-order bits may contain the current temperature, pressure, humidity or other environmental variable.

Advanced RFID chips also contain permanent storage. This capability led the European Central Bank to make a decision to put RFID chips in euro banknotes in the coming years. The chips would record where they have been. Not only would

this make counterfeiting euro notes virtually impossible, but it would make tracing kidnapping ransoms, the loot taken from robberies, and laundered money much easier to track and possibly remotely invalidate. When cash is no longer anonymous, standard police procedure in the future may be to check out where the suspect's money has been recently. Who needs to implant chips in people when their wallets are full of them? Again, when the public learns about what RFID chips can do, there is likely to be some public discussion about the matter.

The technology used in RFID chips is developing rapidly. The smallest ones are passive (not internally powered) and capable only of transmitting their unique numbers when queried. However, larger ones are active, can contain a small battery and a primitive computer, and are capable of doing some calculations. Smart cards used in financial transactions fall into this category.

RFID chips differ not only in being active or passive, but also in the range of radio frequencies they respond to. Those operating at low frequencies have a limited data rate but can be sensed at great distances from the antenna. Those operating at high frequencies have a higher data rate and a shorter range. The chips also differ in other ways and are being improved all the time. The Internet is full of information about RFID chips, with *www.rfid.org* being one good starting point.

1.3.4 Microcontrollers

Next up the ladder we have computers that are embedded inside devices that are not sold as computers. The embedded computers, sometimes called **microcontrollers**, manage the devices and handle the user interface. Microcontrollers are found in a large variety of different devices, including the following. Some examples of each category are given in parentheses.

1. Appliances (clock radio, washer, dryer, microwave, burglar alarm).
2. Communications gear (cordless phone, cell phone, fax, pager).
3. Computer peripherals (printer, scanner, modem, CD ROM-drive).
4. Entertainment devices (VCR, DVD, stereo, MP3 player, set-top box).
5. Imaging devices (TV, digital camera, camcorder, lens, photocopier).
6. Medical devices (X-ray, MRI, heart monitor, digital thermometer).
7. Military weapon systems (cruise missile, ICBM, torpedo).
8. Shopping devices (vending machine, ATM, cash register).
9. Toys (talking doll, game console, radio-controlled car or boat).

A car can easily contain 50 microcontrollers, running subsystems including the antilock brakes, fuel injection, radio, lights, and GPS. A jet plane can easily have 200 or more. A family might easily own several hundred computers without even

knowing it. Within a few years, practically everything that runs on electricity or batteries will contain a microcontroller. The number of microcontrollers sold every year dwarfs that of all other kinds of computers except disposable computers by orders of magnitude.

While RFID chips are minimal systems, microcontrollers are small, but complete, computers. Each microcontroller has a processor, memory, and I/O capability. The I/O capability usually includes sensing the device's buttons and switches and controlling the device's lights, display, sound, and motors. In most cases, the software is built into the chip in the form of a read-only memory created when the microcontroller is manufactured. Microcontrollers come in two general types: general purpose and special purpose. The former are just small, but ordinary computers; the latter have an architecture and instruction set tuned to some specific application, such as multimedia. Microcontrollers come in 4-bit, 8-bit, 16-bit, and 32-bit versions.

However, even the general-purpose microcontrollers differ from standard PCs in important ways. First, they are extremely cost sensitive. A company buying millions of units may make the choice based on a 1-cent price difference per unit. This constraint leads manufacturers to make architectural choices based much more on manufacturing costs, a criteria less dominant on chips costing hundreds of dollars. Microcontroller prices vary greatly depending on how many bits wide they are, how much and what kind of memory they have, and other factors. To get an idea, an 8-bit microcontroller purchased in large enough volume can probably be had for as little as 10 cents per unit. This price is what makes it possible to put a computer inside a $9.95 clock radio.

Second, virtually all microcontrollers operate in real time. They get a stimulus and are expected to give an instantaneous response. For example, when the user presses a button, often a light goes on, and there should not be any delay between the button being pressed and the light going on. The need to operate in real time often has impact on the architecture.

Third, embedded systems often have physical constraints in terms of size, weight, battery consumption, and other electrical and mechanical limits. The microcontrollers used in them have to be designed with these restrictions in mind.

One particularly fun application of microcontrollers is in the Arduino embedded control platform. Arduino was designed by Massimo Banzi and David Cuartielles in Ivrea, Italy. Their goal for the project was to produce a complete embedded computing platform that costs less than a large pizza with extra toppings, making it easily accessible to students and hobbyists. (This was a difficult task, because there is a glut of pizzas in Italy, so they are really cheap.) They achieved their goal well: a complete Arduino system costs less than 20 US dollars!

The Arduino system is an open-source hardware design, which means that all its details are published and free so that anyone can build (and even sell) an Arduino system. It is based on the Atmel AVR 8-bit RISC microcontroller, and most board designs also include basic I/O support. The board is programmed using

an embedded programming language called Wiring which has built-in all the bells and whistles required to control real-time devices. What makes the Arduino platform fun to use is its large and active development community. There are thousands of published projects using the Arduino, ranging from an electronic pollutant sniffer, to a biking jacket with turn signals, a moisture detector that sends email when a plant needs to be watered, and an unmanned autonomous airplane. To learn more about the Arduino and get your hands dirty on your own Arduino projects, go to *www.arduino.cc*.

1.3.5 Mobile and Game Computers

A step up are the mobile platforms and video game machines. They are normal computers, often with special graphics and sound capability but with limited software and little extensibility. They started out as low-end CPUs for simple phones and action games like ping pong on TV sets. Over the years they have evolved into far more powerful systems, rivaling or even outperforming personal computers in certain dimensions.

To get an idea of what is inside these systems, consider the specifications of three popular products. First, the Sony PlayStation 3. It contains a 3.2-GHz multicore proprietary CPU (called the Cell microprocessor), which is based on the IBM PowerPC RISC CPU, and seven 128-bit Synergistic Processing Elements (SPEs). The PlayStation 3 also contains 512 MB of RAM, a 550-MHz custom Nvidia graphics chip, and a Blu-ray player. Second, the Microsoft Xbox 360. It contains a 3.2-GHz IBM triple-core PowerPC CPU with 512 MB of RAM, a 500-MHz custom ATI graphics chip, a DVD player, and a hard disk. Third, the Samsung Galaxy Tablet (on which this book was proofread). It contains two 1-GHz ARM cores plus a graphics processing unit (integrated into the Nvidia Tegra 2 system-on-a-chip), 1 GB of RAM, dual cameras, a 3-axis gyroscope, and flash memory storage.

While these machines are not quite as powerful as high-end personal computers produced in the same time period, they are not that far behind, and in some ways they are ahead (e.g., the 128-bit SPE in the PlayStation 3 is wider than the CPU in any PC). The main difference between these machines and a PC is not so much the CPU as it is their being closed systems. Users may not expand them with plug-in cards, although USB or FireWire interfaces are sometimes provided. Also, and perhaps most important, these platforms are carefully optimized for a few application domains: highly interactive applications with 3D graphics and multimedia output. Everything else is secondary. These hardware and software restrictions, lack of extensibility, small memories, absence of a high-resolution monitor, and small (or sometime absent) hard disk make it possible to build and sell these machines more cheaply than personal computers. Despite these restrictions, millions of these devices have been sold and their numbers are growing all the time.

Mobile computers have the added requirement that they use as little energy as possible to perform their tasks. The less energy they use the longer their battery will last. This is a challenging design task because mobile platforms such as tablets and smartphones must be frugal in their energy use, but at the same time, users of these devices expect high-performance capabilities, such as 3D graphics, high-definition multimedia processing, and gaming.

1.3.6 Personal Computers

Next, we come to the personal computers that most people think of when they hear the term "computer." These include desktop and notebook models. They usually come with a few gigabytes of memory, a hard disk holding up to terabytes of data, a CD-ROM/DVD/Blu-ray drive, sound card, network interface, high-resolution monitor, and other peripherals. They have elaborate operating systems, many expansion options, and a huge range of available software.

The heart of every personal computer is a printed circuit board at the bottom or side of the case. It usually contains the CPU, memory, various I/O devices (such as a sound chip and possibly a modem), as well as interfaces to the keyboard, mouse, disk, network, etc., and some expansion slots. A photo of one of these circuit boards is given in Fig. 1-10.

Notebook computers are basically PCs in a smaller package. They use the same hardware components, but manufactured in smaller sizes. They also run the same software as desktop PCs. Since most readers are probably quite familiar with notebook and personal computers, additional introductory material is hardly needed.

Yet another variant on this theme is the tablet computer, such as the popular iPad. These devices are just normal PCs in a smaller package, with a solid-state disk instead of a rotating hard disk, a touch screen, and a different CPU than the x86. But from an architectural perspective, tablets are just notebooks with a different form factor.

1.3.7 Servers

Beefed-up personal computers or workstations are often used as network servers, both for local area networks (typically within a single company), and for the Internet. These come in single-processor and multiple-processor configurations, and have gigabytes of memory, terabytes of hard disk space, and high-speed networking capability. Some of them can handle thousands of transactions per second.

Architecturally, however, a single-processor server is not really very different from a single-processor personal computer. It is just faster, bigger, and has more disk space and possibly a faster network connection. Servers run the same operating systems as personal computers, typically some flavor of UNIX or Windows.

Figure 1-10. A printed circuit board is at the heart of every personal computer. This one is the Intel DQ67SW board. © 2011 Intel Corporation. Used by permission.

Clusters

Owing to almost continuous improvements in the price/performance ratio of servers, in recent years system designers have begun connecting large numbers of them together to form **clusters**. They consist of standard server-class systems connected by gigabit/sec networks and running special software that allow all the machines to work together on a single problem, often in business, science or engineering. Normally, they are what are called **COTS** (**Commodity Off The Shelf**) computers that anyone can buy from a PC vendor. The main addition is high-speed networking, but sometimes that is also a standard commercial network card, too.

Large clusters are typically housed in special-purpose rooms or buildings called **data centers**. Data centers can scale quite large, from a handful of machines to 100,000 or more of them. Usually, the amount of money available is the limiting factor. Owing to their low component price, individual companies can now own such machines for internal use. Many people use the terms "cluster" and "data center" interchangeably although technically the former is the collection of servers and the latter is the room or building.

A common use for a cluster is as an Internet Web server. When a Website expects thousands of requests per second for its pages, the most economical solution

is often to build a data center with hundreds, or even thousands, of servers. The incoming requests are then sprayed among the servers to allow them to be processed in parallel. For example, Google has data centers all over the world to service search requests, the largest one, in The Dalles, Oregon, is a facility that is as large as two (U.S.) football fields. The location was chosen because data centers require vast amounts of electric power and The Dalles is the site of a 2 GW hydroelectric dam on the Columbia River that can provide it. Altogether, Google is thought to have more than 1,000,000 servers in its data centers.

The computer business is a dynamic one, with things changing all the time. In the 1960s, computing was dominated by giant mainframe computers (see below) costing tens of millions of dollars to which users connected using small remote terminals. This was a very centralized model. Then in the 1980s personal computers arrived on the scene, millions of people bought one, and computing was decentralized.

With the advent of data centers, we are starting to relive the past in the form of **cloud computing**, which is mainframe computing V2.0. The idea here is that everyone will have one or more simple devices, including PCs, notebooks, tablets, and smartphones that are essentially user interfaces to the cloud (i.e, the data center) where all the user's photos, videos, music, and other data are stored. In this model, the data are accessible from different devices anywhere and at any time without the user having to keep track of where they are. Here, the data center full of servers has replaced the single large centralized computer, but the paradigm has reverted back to the old one: the users have simple terminals and data and computing power is centralized somewhere else.

Who knows how long this model will be popular? It could easily happen in 10 years that so many people have stored so many songs, photos, and videos in the cloud that the (wireless) infrastructure for communicating with it has become completely bogged down. This could lead to a new revolution: personal computers, where people store their own data on their own machines locally, thus bypassing the traffic jam over the air.

The take-home message here is that the model of computing popular in a given era depends a lot on the technology, economics, and applications available at the time and can change when these factors change.

1.3.8 Mainframes

Now we come to the mainframes: room-sized computers that hark back to the 1960s. These machines are the direct descendants of IBM 360 mainframes acquired decades ago. For the most part, they are not much faster than powerful servers, but they always have more I/O capacity and are often equipped with vast disk farms, often holding thousands of gigabytes of data. While expensive, they are often kept running due to the immense investment in software, data, operating procedures, and personnel that they represent. Many companies find it cheaper to just

pay a few million dollars once in a while for a new one, than to even contemplate the effort required to reprogram all their applications for smaller machines.

It is this class of computer that led to the now-infamous Year 2000 problem, which was caused by (mostly COBOL) programmers in the 1960s and 1970s representing the year as two decimal digits (in order to save memory). They never envisioned their software lasting three or four decades. While the predicted disaster never occurred due to a huge amount of work put into fixing the problem, many companies have repeated the same mistake by simply adding two more digits to the year. The authors hereby predict the end of civilization at midnight on Dec. 31, 9999, when 8000 years worth of old COBOL programs crash simultaneously.

In addition to their use for running 40-year-old legacy software, the Internet has breathed new life into mainframes. They have found a new niche as powerful Internet servers, for example, by handling massive numbers of e-commerce transactions per second, particularly in businesses with huge databases.

Up until recently, there was another category of computers even more powerful than mainframes: **supercomputers**. They had enormously fast CPUs, many gigabytes of main memory, and very fast disks and networks. They were used for massive scientific and engineering calculations such as simulating colliding galaxies, synthesizing new medicines, or modeling the flow of air around an airplane wing. However, in recent years, data centers constructed from commodity components have come to offer as much computing power at much lower prices, and the true supercomputers are now a dying breed.

1.4 EXAMPLE COMPUTER FAMILIES

In this book we will focus on three popular instruction set architectures (ISAs): x86, ARM and AVR. The x86 architecture is found in nearly all personal computers (including Windows and Linux PCs and Macs) and server systems. Personal computers are of interest because every reader has undoubtedly used one. Servers are of interest because they run all the services on the Internet. The ARM architecture dominates the mobile market. For example, most smartphones and tablet computers are based on ARM processors. Finally, the AVR architecture is found in very low-cost microcontrollers found in many embedded computing applications. Embedded computers are invisible to their users but control cars, televisions, microwave ovens, washing machines, and practically every other electrical device costing more than $50. In this section, we will briefly introduce the three instruction set architectures that will be used as examples in the rest of the book.

1.4.1 Introduction to the x86 Architecture

In 1968, Robert Noyce, inventor of the silicon integrated circuit; Gordon Moore, of Moore's law fame; and Arthur Rock, a San Francisco venture capitalist, formed the Intel Corporation to make memory chips. In its first year of operation,

Intel sold only $3000 worth of chips, but business has picked up since then (Intel is now the world's largest CPU chip manufacturer).

In the late 1960s, calculators were large electromechanical machines the size of a modern laser printer and weighing 20 kg. In Sept. 1969, a Japanese company, Busicom, approached Intel with a request that it manufacture 12 custom chips for a proposed electronic calculator. The Intel engineer assigned to this project, Ted Hoff, looked at the plan and realized that he could put a 4-bit general-purpose CPU on a single chip that would do the same thing and be simpler and cheaper as well. Thus, in 1970, the first single-chip CPU, the 2300-transistor 4004, was born (Faggin et al., 1996).

It is worth noting that neither Intel nor Busicom had any idea what they had just done. When Intel decided that it might be worth a try to use the 4004 in other projects, it offered to buy back all the rights to the new chip from Busicom by returning the $60,000 Busicom had paid Intel to develop it. Intel's offer was quickly accepted, at which point it began working on an 8-bit version of the chip, the 8008, introduced in 1972. The Intel family, starting with the 4004 and 8008, is shown in Fig. 1-11, giving the introduction date, clock rate, transistor count, and memory.

Chip	Date	MHz	Trans.	Memory	Notes
4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions
Core Duo	1/2006	1600–3200	152M	2 GB	Dual cores on a single die
Core	7/2006	1200–3200	410M	64 GB	64-bit quad core architecture
Core i7	1/2011	1100–3300	1160M	24 GB	Integrated graphics processor

Figure 1-11. Key members of the Intel CPU family. Clock speeds are measured in MHz (megahertz), where 1 MHz is 1 million cycles/sec.

Intel did not expect much demand for the 8008, so it set up a low-volume production line. Much to everyone's amazement, there was an enormous amount of interest, so Intel set about designing a new CPU chip that got around the 8008's limit

of 16 kilobytes of memory (imposed by the number of pins on the chip). This design resulted in the 8080, a small, general-purpose CPU, introduced in 1974. Much like the PDP-8, this product took the industry by storm and instantly became a mass-market item. Only instead of selling thousands, as DEC had, Intel sold millions.

In 1978 came the 8086, a genuine 16-bit CPU on a single chip. The 8086 was designed to be similar to the 8080, but it was not completely compatible with the 8080. The 8086 was followed by the 8088, which had the same architecture as the 8086 and ran the same programs but had an 8-bit bus instead of a 16-bit bus, making it both slower and cheaper than the 8086. When IBM chose the 8088 as the CPU for the original IBM PC, this chip quickly became the personal computer industry standard.

Neither the 8088 nor the 8086 could address more than 1 megabyte of memory. By the early 1980s this had become a serious problem, so Intel designed the 80286, an upward compatible version of the 8086. The basic instruction set was essentially the same as that of the 8086 and 8088, but the memory organization was quite different, and rather awkward, due to the requirement of compatibility with the older chips. The 80286 was used in the IBM PC/AT and in the midrange PS/2 models. Like the 8088, it was a huge success, mostly because people viewed it as a faster 8088.

The next logical step was a true 32-bit CPU on a chip, the 80386, brought out in 1985. Like the 80286, this one was more or less compatible with everything back to the 8080. Being backward compatible was a boon to people for whom running old software was important, but a nuisance to people who would have preferred a simple, clean, modern architecture unencumbered by the mistakes and technology of the past.

Four years later the 80486 came out. It was essentially a faster version of the 80386 that also had a floating-point unit and 8 kilobytes of cache memory on chip. **Cache memory** is used to hold the most commonly used memory words inside or close to the CPU, in order to avoid (slow) accesses to main memory. The 80486 also had built-in multiprocessor support, allowing manufacturers to build systems containing multiple CPUs sharing a common memory.

At this point, Intel found out the hard way (by losing a trademark infringement lawsuit) that numbers (like 80486) cannot be trademarked, so the next generation got a name: **Pentium** (from the Greek word for five, *πεντε*). Unlike the 80486, which had one internal pipeline, the Pentium had two of them, which helped make it twice as fast (we will discuss pipelines in detail in Chap. 2).

Later in the production run, Intel added special **MMX** (**MultiMedia eXtension**) instructions. These instructions were intended to speed up computations required to process audio and video, making the addition of special multimedia coprocessors unnecessary.

When the next generation appeared, people who were hoping for the Sexium (*sex* is Latin for six) were sorely disappointed. The name Pentium was now so

well known that the marketing people wanted to keep it, and the new chip was called the Pentium Pro. Despite the small name change from its predecessor, this processor represented a major break with the past. Instead, of having two or more pipelines, the Pentium Pro had a very different internal organization and could execute up to five instructions at a time.

Another innovation found in the Pentium Pro was a two-level cache memory. The processor chip itself had 8 kilobytes of fast memory to hold commonly used instructions and another 8 kilobytes of fast memory to hold commonly used data. In the same cavity within the Pentium Pro package (but not on the chip itself) was a second cache memory of 256 kilobytes.

Although the Pentium Pro had a big cache, it lacked the MMX instructions (because Intel was unable to manufacture such a large chip with acceptable yields). When the technology improved enough to get both the MMX instructions and the cache on one chip, the combined product was released as the Pentium II. Next, yet more multimedia instructions, called **SSE** (**Streaming SIMD Extensions**), were added for enhanced 3D graphics (Raman et al., 2000). The new chip was dubbed the Pentium III, but internally it was essentially a Pentium II.

The next Pentium, released in Nov. 2000, was based on a different internal architecture but had the same instruction set as the earlier Pentiums. To celebrate this event, Intel switched from Roman numerals to Arabic numbers and called it the Pentium 4. As usual, the Pentium 4 was faster than all its predecessors. The 3.06-GHz version also introduced an intriguing new feature—hyperthreading. This feature allowed programs to split their work into two threads of control which the Pentium 4 could run in parallel, speeding up execution. In addition, another batch of SSE instructions was added to speed up audio and video processing even more.

In 2006, Intel changed the brand name from Pentium to Core and released a dual core chip, the **Core 2 duo**. When Intel decided it wanted a cheaper single-core version of the chip, it just sold Core 2 duos with one core disabled because wasting a little silicon on each chip manufacturered was ultimately cheaper than incurring the enormous expense of designing and testing a new chip from scratch. The Core series has continued to evolve, with the i3, i5, and i7 being popular variants for low-, medium-, and high-performance computers. No doubt more variants will follow. A photo of the i7 is presented in Fig. 1-12. There are actually eight cores on it, but except in the Xeon version, only six are enabled. This approach means that a chip with one or two defective cores can still be sold by disabling the defective one(s). Each core has its own level 1 and level 2 caches, but there is also a shared level 3 (L3) cache used by all the cores. We will discuss caches in detail later in this book.

In addition to the mainline desktop CPUs discussed so far, Intel has manufactured variants of some of the Pentium chips for special markets. In early 1998, Intel introduced a new product line called the **Celeron**, which was basically a low-price, low-performance version of the Pentium 2 intended for low-end PCs. Since

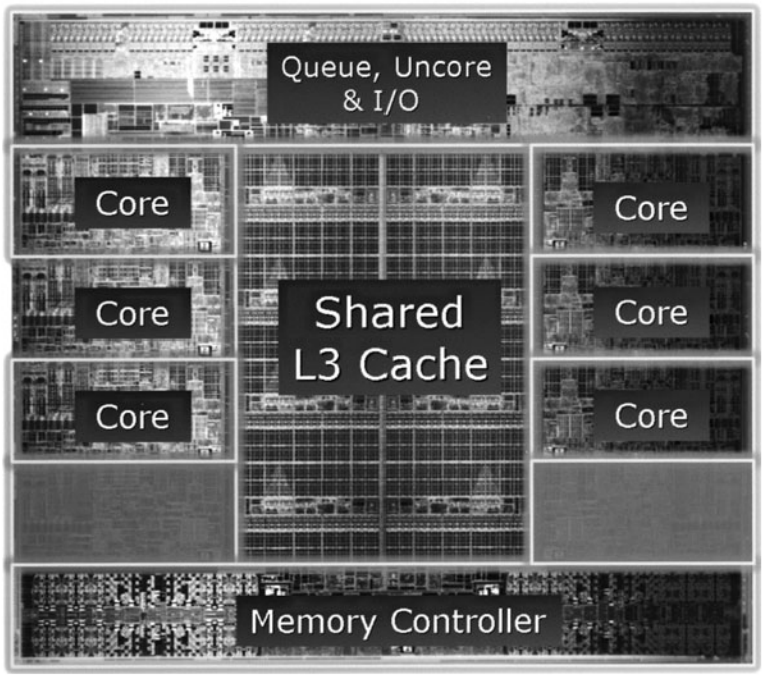

Figure 1-12. The Intel Core i7-3960X die. The die is 21 by 21 mm and has 2.27 billion transistors. © 2011 Intel Corporation. Used by permission.

the Celeron has the same architecture as the Pentium 2, we will not discuss it further in this book. In June 1998, Intel introduced a special version of the Pentium 2 for the upper end of the market. This processor, called the **Xeon**, had a larger cache, a faster bus, and better multiprocessor support but was otherwise a normal Pentium 2, so we will not discuss it separately either. The Pentium III also had a Xeon version as do more recent chips. On more recent chips, one feature of the Xeon is more cores.

1n 2003, Intel introduced the Pentium M (as in Mobile), a chip designed for notebook computers. This chip was part of the Centrino architecture, whose goals were lower power consumption for longer battery lifetime; smaller, lighter, computers; and built-in wireless networking capability using the IEEE 802.11 (WiFi) standard. The Pentium M was very low power and much smaller than the Pentium 4, two characteristics that would soon allow it (and its successors) to subsume the Pentium 4 microarchitecture in future Intel products.

All the Intel chips are backward compatible with their predecessors as far back as the 8086. In other words, a Pentium 4 or Core can run old 8086 programs without modification. This compatibility has always been a design requirement for Intel, to allow users to maintain their existing investment in software. Of course,

the Core is four orders of magnitude more complex than the 8086, so it can do quite a few things that the 8086 could not do. These piecemeal extensions have resulted in an architecture that is not as elegant as it might have been had someone given the Pentium 4 architects 42 million transistors and instructions to start all over again.

It is interesting to note that although Moore's law has long been associated with the number of bits in a memory, it applies equally well to CPU chips. By plotting the transistor counts given in Fig. 1-8 against the date of introduction of each chip on a semilog scale, we see that Moore's law holds here too. This graph is given in Fig. 1-13.

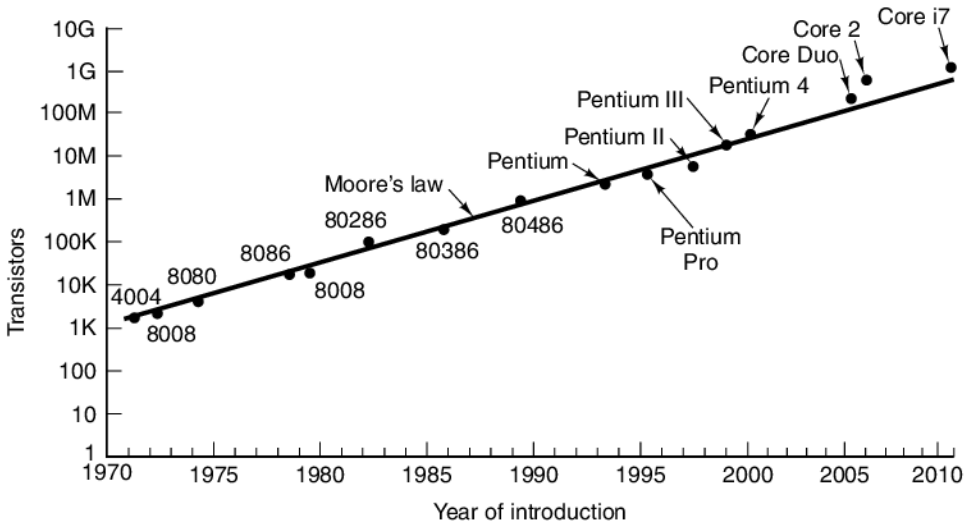

Figure 1-13. Moore's law for (Intel) CPU chips.

While Moore's law will probably continue to hold for some years to come, another problem is starting to overshadow it: heat dissipation. Smaller transistors make it possible to run at higher clock frequencies, which requires using a higher voltage. Power consumed and heat dissipated is proportional to the square of the voltage, so going faster means having more heat to get rid of. At 3.6 GHz, the Pentium 4 consumes 115 watts of power. That means it gets about as hot as a 100-watt light bulb. Speeding up the clock makes the problem worse.

In November 2004, Intel canceled the 4-GHz Pentium 4 due to problems dissipating the heat. Large fans can help but the noise they make is not popular with users, and water cooling, while used on large mainframes, is not an option for desktop machines (and even less so for notebook computers). As a consequence, the once-relentless march of the clock may have ended, at least until Intel's engineers figure out an efficient way to get rid of all the heat generated. Instead, Intel CPU designs now put two or more CPUs on a single chip, along with large shared

cache. Because of the way power consumption is related to voltage and clock speed, two CPUs on a chip consume far less power than one CPU at twice the speed. As a consequence, the gain offered by Moore's law may be increasingly exploited in the future to include more cores and larger on-chip caches, rather than higher and higher clock speeds. Taking advantage of these multiprocessors poses great challenges to programmers, because unlike the sophisticated uniprocessor microarchitectures of the past that could extract more performance from existing programs, multiprocessors require the programmer to explicitly orchestrate parallel execution, using threads, semaphores, shared memory and other headache- and bug-inducing technologies.

1.4.2 Introduction to the ARM Architecture

In the early 80s, the U.K.-based company Acorn Computer, flush with the success of their 8-bit BBC Micro personal computer, began working on a second machine with the hope of competing with the recently released IBM PC. The BBC Micro was based on the 8-bit 6502 processor, and Steve Furber and his colleagues at Acorn felt that the 6502 did not have the muscle to compete with the IBM PC's 16-bit 8086 processor. They began looking at the options in the marketplace, and decided that they were too limited.

Inspired by the Berkeley RISC project, in which a small team designed a remarkably fast processor (which eventually led to the SPARC architecture), they decided to build their own CPU for the project. They called their design the Acorn RISC Machine (or ARM, which would later be rechristened the Advanced RISC machine when ARM eventually split from Acorn). The design was completed in 1985. It included 32-bit instructions and data, and a 26-bit address space, and it was manufactured by VLSI Technology.

The first ARM architecture (called the ARM2) appeared in the Acorn Archimedes personal computer. The Archimedes was a very fast and inexpensive machine for its day, running up to 2 MIPS (millions of instructions per second) and costing only 899 British pounds at launch. The machine became very popular in the UK, Ireland, Australia and New Zealand, especially in schools.

Based on the success of the Archimedes, Apple approached Acorn to develop an ARM processor for their upcoming Apple Newton project, the first palmtop computer. To better focus on the project, the ARM architecture team left Acorn to create a new company called Advanced RISC Machines (ARM). Their new processor was called the ARM 610, which powered the Apple Newton when it was release in 1993. Unlike the original ARM design, this new ARM processor incorporated a 4-KB cache that significantly improved the design's performance. Although the Apple Newton was not a great success, the ARM 610 did see other successful applications including Acorn's RISC PC computer.

In the mid 1990s, ARM collaborated with Digital Equipment Corporation to develop a high-speed, low-power version of the ARM, intended for energy-frugal

mobile applications such as PDAs. They produced the StrongARM design, which from its first appearance sent waves through the industry due to its high speed (233 MHz) and ultralow power demands (1 watt). It gained efficiency through a simple, clean design that included two 16-KB caches for instructions and data. The StrongARM and its successors at DEC were moderately successful in the marketplace, finding their way into a number of PDAs, set-top boxes, media devices, and routers.

Perhaps the most venerable of the ARM architectures is the ARM7 design, first released by ARM in 1994 and still in wide use today. The design included separate instruction and data caches, and it also incorporated the 16-bit Thumb instruction set. The Thumb instruction set is a shorthand version of the full 32-bit ARM instruction set, allowing programmers to encode many of the most common operations into smaller 16-bit instructions, significantly reducing the amount of program memory needed. The processor has worked well for a wide range of low- to middle-end embedded applications such as toasters, engine control, and even the Nintendo Gameboy Advance hand-held gaming console.

Unlike many computer companies, ARM does not manufacture any microprocessors. Instead, it creates designs and ARM-based developer tools and libraries, and licenses them to system designers and chip manufacturers. For example, the CPU used in the Samsung Galaxy Tab Android-based tablet computer is an ARM-based processor. The Galaxy Tab contains the Tegra 2 system-on-chip processor, which includes two ARM Cortex-A9 processors and an Nvidia GeForce graphics processing unit. The Tegra 2 cores were designed by ARM, integrated into a system-on-a-chip design by Nvidia, and manufactured by Taiwan Semiconductor Manufacturing Company (TSMC). It's an impressive collaboration by companies in different countries in which all of the companies contributed value to the final design.

Figure 1-14 shows a die photo of the Nvidia's Tegra 2 system-on-a-chip. The design contains three ARM processors: two 1.2-GHz ARM Cortex-A9 cores plus an ARM7 core. The Cortex-A9 cores are dual-issue out-of-order cores with a 1-MB L2 cache and support for shared-memory multiprocessing. (That's a lot of buzzwords that we will get into in later chapters. For now, just know that these features make the design very fast!) The ARM7 core is an older and smaller ARM core used for system configuration and power management. The graphics core is a 333-MHz GeForce graphics processing unit (GPU) design optimized for low-power operation. Also included on the Tegra 2 are a video encoder/decoder, an audio processor and an HDMI video output interface.

The ARM architecture has found great success in the low-power, mobile and embedded markets. In January 2011, ARM announced that it had sold 15 billion ARM processors since its inception, and indicated that sales were continuing to grow. While tailored for lower-end markets, the ARM architecture does have the computational capability to perform in any market, and there are hints that it may be expanding its horizons. For example, in October 2011, a 64-bit ARM was

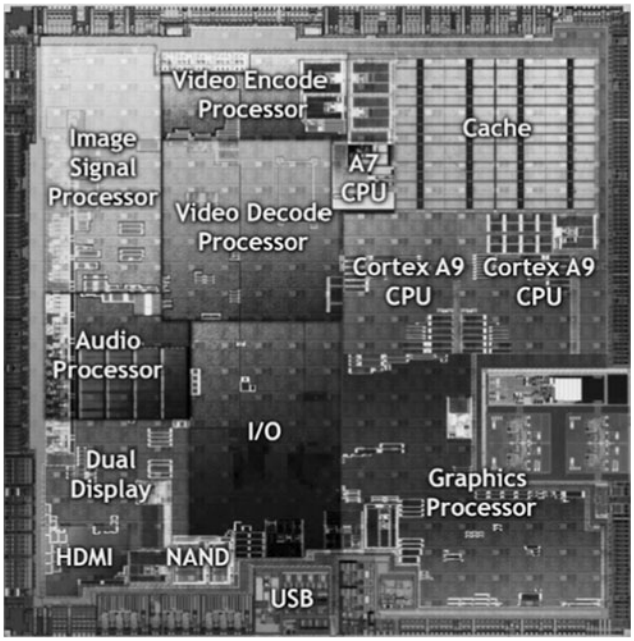

Figure 1-14. The Nvidia Tegra 2 system on a chip. © 2011 Nvidia Corporation. Used by permission.

announced. Also in January 2011, Nvidia announced "Project Denver," an ARM-based system-on-a-chip being developed for the server and other markets. The design will incorporate multiple 64-bit ARM processors plus a general-purpose GPU (GPGPU). The low-power aspects of the design will help to reduce the cooling requirements of server farms and data centers.

1.4.3 Introduction to the AVR Architecture

Our third example is very different from our first (the x86 architecture, used in personal computers and servers) and second (the ARM architecture, used in PDAs and smartphones). It is the AVR architecture, which is used in very low-end embedded systems. The AVR story starts in 1996 at the Norwegian Institute of Technology, where students Alf-Egil Bogen and Vegard Wollan designed an 8-bit RISC CPU called the AVR. It was reportedly given this name because it was "(A)lf and (V)egard's (R)ISC processor." Shortly after the design was completed, Atmel bought the design and started Atmel Norway, where the two architects continued to refine the AVR processor design. Atmel released their first AVR microcontroller, the AT90S1200, in 1997. To ease its adoption by system designers, they implemented the pinout to be exactly the same as that of the Intel 8051, which was one

of the most popular microcontrollers at the time. Today there is much interest in the AVR architecture because it is at the heart of the very popular open-source Arduino embedded controller platform.

The AVR architecture is implemented in three classes of microcontrollers, listed in Fig. 1-15. The lowest class, the tinyAVR is designed for the most area-, power- and cost-constrained applications. It includes an 8-bit CPU, basic digital I/O support, and analog input support (for example, reading temperature values off a thermistor). The tinyAVR is so small that its pins work double duty, such that they can be reprogrammed at run time to any of the digital or analog functions supported by the microcontroller. The megaAVR, which is found in the popular Arduino open-source embedded system, also adds serial I/O support, internal clocks, and programmable analog outputs. The top end of the bottom end is the AVR XMEGA microcontroller, which also incorporates an accelerator for cryptographic operations plus built-in support for USB interfaces.

Chip	Flash	EEPROM	RAM	Pins	Features
tinyAVR	0.5–16 KB	0–512 B	32–512 B	6–32	Tiny, digital I/O, analog input
megaAVR	8–256 KB	0.5–4 KB	0.25–8 KB	28-100	Many peripherals, analog out
AVR XMEGA	16–256 KB	1–4 KB	2–16 KB	44–100	Crypto acceleration, USB I/O

Figure 1-15. Microcontroller classes in the AVR family.

Along with various additional peripherals, each AVR processor class includes some additional memory resources. Microcontrollers typically have three types of memory on board: flash, EEPROM, and RAM. Flash memory is programmable using an external interface and high voltages, and this is where program code and data are stored. Flash RAM is nonvolatile, so even if the system is powered down, the flash memory will remember what was written to it. Like flash, EEPROM is also nonvolatile, but unlike flash RAM, it can be changed by the program while it is running. This is the storage in which an embedded system would keep user configuration information, such as whether your alarm clock displays time in 12- or 24-hour format. Finally, the RAM is where program variables will be stored as the program runs. This memory is volatile, so any value stored here will be lost once the system loses power. We study volatile and nonvolatile RAM types in detail in Chap. 2.

The recipe for success in the microcontroller business is to cram into the chip everything it may possibly need (and the kitchen sink, too, if it can be reduced to a square millimeter) and then put it into an inexpensive and small package with very few pins. By integrating lots of features into the microcontroller, it can work for many applications, and by making it cheap and small, it can serve many form factors. To get a sense of how many things get packed onto a modern microcontroller, let's take a look at the peripherals included in the Atmel megaAVR-168:

1. Three timers (two 8-bit timers and one 16-bit timer).
2. Real-time clock with oscillator.
3. Six pulse-width modulation channels used, for example, to control light intensity or motor speed.
4. Eight analog-to-digital conversion channels used to read voltage levels.
5. Universal serial receiver/transmitter.
6. I2C serial interface, a common standard for interfacing to sensors.
7. Programmable watchdog timer that detects when the system has locked up.
8. On-chip analog comparator that compares two input voltages.
9. Power brown-out detector that interrupts the system when power is failing.
10. Internal programmable clock oscillator to drive the CPU clock.

1.5 METRIC UNITS

To avoid any confusion, it is worth stating explicitly that in this book, as in computer science in general, metric units are used instead of traditional English units (the furlong-stone-fortnight system). The principal metric prefixes are listed in Fig. 1-16. The prefixes are typically abbreviated by their first letters, with the units greater than 1 capitalized (KB, MB, etc.). One exception (for historical reasons) is kbps for kilobits/sec. Thus, a 1-Mbps communication line transmits 10^6 bits/sec and a 100-psec (or 100-ps) clock ticks every 10^{-10} seconds. Since milli and micro both begin with the letter "m," a choice had to be made. Normally, "m" is for milli and "μ" (the Greek letter mu) is for micro.

It is also worth pointing out that in common industry practice for measuring memory, disk, file, and database sizes, the units have slightly different meanings. There, kilo means 2^{10} (1024) rather than 10^3 (1000) because memories are always a power of two. Thus, a 1-KB memory contains 1024 bytes, not 1000 bytes. Similarly, a 1-MB memory contains 2^{20} (1,048,576) bytes, a 1-GB memory contains 2^{30} (1,073,741,824) bytes, and a 1-TB database contains 2^{40} (1,099,511,627,776) bytes.

However, a 1-kbps communication line can transmit 1000 bits per second and a 10-Mbps LAN runs at 10,000,000 bits/sec because these speeds are not powers of two. Unfortunately, many people tend to mix up these two systems, especially for disk sizes.

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	yotta

Figure 1-16. The principal metric prefixes.

To avoid ambiguity, the standards organizations have introduced the new terms kibibyte for 2^{10} bytes, mebibyte for 2^{20} bytes, gibibyte for 2^{30} bytes, and tebibyte for 2^{40} bytes, but the industry has been slow to adopt them. We feel that until these new terms are in wider use, it is better to stick with the symbols KB, MB, GB, and TB for 2^{10}, 2^{20}, 2^{30}, and 2^{40} bytes, respectively, and the symbols kbps, Mbps, Gbps, and Tbps for 10^{3}, 10^{6}, 10^{9}, and 10^{12} bits/sec, respectively.

1.6 OUTLINE OF THIS BOOK

This book is about multilevel computers (which includes nearly all modern computers) and how they are organized. We will examine four levels in considerable detail—namely, the digital logic level, the microarchitecture level, the ISA level, and the operating system machine level. Some of the basic issues to be examined include the overall design of the level (and why it was designed that way), the kinds of instructions and data available, the memory organization and addressing, and the method by which the level is implemented. The study of these topics, and similar ones, is called computer organization or computer architecture.

We are primarily concerned with concepts rather than details or formal mathematics. For that reason, some of the examples given will be highly simplified, in order to emphasize the central ideas and not the details.

To provide some insight into how the principles presented in this book can be, and are, applied in practice, we will use the x86, ARM, and AVR architectures as running examples throughout the book. These three have been chosen for several reasons. First, all are widely used and the reader is likely to have access to at least one of them. Second, each one has its own unique architecture, which provides a basis for comparison and encourages a "what are the alternatives?" attitude. Books dealing with only one machine often leave the reader with a "true machine design revealed" feeling, which is absurd in light of the many compromises and arbitrary decisions that designers are forced to make. The reader is encouraged to

study these and all other computers with a critical eye and to try to understand why things are the way they are, as well as how they could have been done differently, rather than simply accepting them as given.

It should be made clear from the beginning that this is not a book about how to program the x86, ARM, or AVR architectures. These machines will be used for illustrative purposes where appropriate, but we make no pretense of being complete. Readers wishing a thorough introduction to one of them should consult the manufacturer's publications.

Chapter 2 is an introduction to the basic components of a computer—processors, memories, and input/output equipment. It is intended to provide an overview of the system architecture and an introduction to subsequent chapters.

Chapters 3, 4, 5, and 6 each deal with one specific level shown in Fig. 1-2. Our treatment is bottom-up, because machines have traditionally been designed that way. The design of level k is largely determined by the properties of level $k-1$, so it is hard to understand any level unless you already have a good grasp of the underlying level that motivated it. Also, it is educationally sound to proceed from the simpler lower levels to the more complex higher levels rather than vice versa.

Chapter 3 is about the digital logic level, the machine's true hardware. It discusses what gates are and how they can be combined into useful circuits. Boolean algebra, a tool for analyzing digital circuits, is also introduced. Computer buses are explained, especially the popular PCI bus. Numerous examples from industry are discussed in this chapter, including the three running examples mentioned above.

Chapter 4 introduces the architecture of the microarchitecture level and its control. Since the function of this level is to interpret the level 2 instructions in the layer above it, we will concentrate on this topic and illustrate it by means of examples. The chapter also contains discussions of the microarchitecture level of some real machines.

Chapter 5 discusses the ISA level, the one most computer vendors advertise as the machine language. We will look at our example machines here in detail.

Chapter 6 covers some of the instructions, memory organization, and control mechanisms present at the operating system machine level. The examples used here are the Windows operating system (popular on x86 based desktop systems) and UNIX, used on many x86 and ARM based systems.

Chapter 7 is about the assembly language level. It covers both assembly language and the assembly process. The subject of linking also comes up here.

Chapter 8 discusses parallel computers, an increasingly important topic nowadays. Some of these parallel computers have multiple CPUs that share a common memory. Others have multiple CPUs without common memory. Some are supercomputers; some are systems on a chip; others are clusters of computers.

Chapter 9 contains an alphabetical list of literature citations. Suggested readings are on the book's Website. See: *www.prenhall.com/tanenbaum*.

PROBLEMS

1. Explain each of the following terms in your own words:

a. Translator.
b. Interpreter.
c. Virtual machine.

2. Is it conceivable for a compiler to generate output for the microarchitecture level instead of for the ISA level? Discuss the pros and cons of this proposal.

3. Can you imagine any multilevel computer in which the device level and digital logic levels were not the lowest levels? Explain.

4. Consider a multilevel computer in which all the levels are different. Each level has instructions that are m times as powerful as those of the level below it; that is, one level r instruction can do the work of m level $r - 1$ instructions. If a level-1 program requires k seconds to run, how long would equivalent programs take at levels 2, 3, and 4, assuming n level r instructions are required to interpret a single $r + 1$ instruction?

5. Some instructions at the operating system machine level are identical to ISA language instructions. These instructions are carried out directly by the microprogram or hardware rather than by the operating system. In light of your answer to the preceding problem, why do you think this is the case?

6. Consider a computer with identical interpreters at levels 1, 2, and 3. It takes an interpreter n instructions to fetch, examine, and execute one instruction. A level-1 instruction takes k nanoseconds to execute. How long does it take for an instruction at levels 2, 3, and 4?

7. In what sense are hardware and software equivalent? In what sense are they not equivalent?

8. Babbage's difference engine had a fixed program that could not be changed. Is this essentially the same thing as a modern CD-ROM that cannot be changed? Explain your answer.

9. One of the consequences of von Neumann's idea to store the program in memory is that programs can be modified, just like data. Can you think of an example where this facility might have been useful? (*Hint*: Think about doing arithmetic on arrays.)

10. The performance ratio of the 360 model 75 was 50 times that of the 360 model 30, yet the cycle time was only five times as fast. How do you account for this discrepancy?

11. Two system designs are shown in Fig. 1-5 and Fig. 1-6. Describe how input/output might occur in each system. Which one has the potential for better overall system performance?

12. Suppose that each of the 300 million people in the United States fully consumes two packages of goods a day bearing RFID tags. How many RFID tags have to be pro-

duced annually to meet that demand? At a penny a tag, what is the total cost of the tags? Given the size of GDP, is this amount of money going to be an obstacle to their use on every package offered for sale?

13. Name three appliances that are candidates for being run by an embedded CPU.

14. At a certain point in time, a transistor on a chip was 0.1 micron in diameter. According to Moore's law, how big would a transistor be on next year's model?

15. It has been shown that Moore's law not only applies to semiconductor density, but it also predicts the increase in (reasonable) simulation sizes, and the reduction in computational simulation run-times. First show for a fluid mechanics simulation that takes 4 hours to run on a machine today, that it should only take 1 hour to run on machines built 3 years from now, and only 15 minutes on machines built 6 years from now. Then show that for a large simulation that has an estimated run-time of 5 years that it would complete sooner if we waited 3 years to start the simulation.

16. In 1959, an IBM 7090 could execute about 500,000 instructions/sec, had a memory of 32,768 36-bit words, and cost $3 million. Compare this to a current computer and determine how much better the current one is by multiplying the ratio of memory sizes and speeds and then dividing this by the ratio of the prices. Now see what the same gains would have done to aviation in the same time period. The Boeing 707 was delivered to the airlines in substantial quantities in 1959. Its speed was 950 km/hr and its capacity was initially 180 passengers. It cost $4 million. What would the speed, capacity, and cost of an aircraft now be if it had the same gains as a computer? Clearly, state your assumptions about speed, memory size, and price.

17. Developments in the computer industry are often cyclic. Originally, instruction sets were hardwired, then they were microprogrammed, then RISC machines came along and they were hardwired again. Originally, computing was centralized on large mainframe computers. List two developments that demonstrate the cyclic behavior here as well.

18. The legal issue of who invented the computer was settled in April 1973 by Judge Earl Larson, who handled a patent infringement lawsuit filed by the Sperry Rand Corporation, which had acquired the ENIAC patents. Sperry Rand's position was that everybody making a computer owed them royalties because it owned the key patents. The case went to trial in June 1971 and over 30,000 exhibits were entered. The court transcript ran to over 20,000 pages. Study this case more carefully using the extensive information available on the Internet and write a report discussing the technical aspects of the case. What exactly did Eckert and Mauchley patent and why did the judge feel their system was based on Atanasoff's earlier work?

19. Pick the three people you think were most influential in creating modern computer hardware and write a short report describing their contributions and why you picked them.

20. Pick the three people you think were most influential in creating modern computer systems software and write a short report describing their contributions and why you picked them.

21. Pick the three people you think were most influential in creating modern Websites that draw a lot of traffic and write a short report describing their contributions and why you picked them.

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.

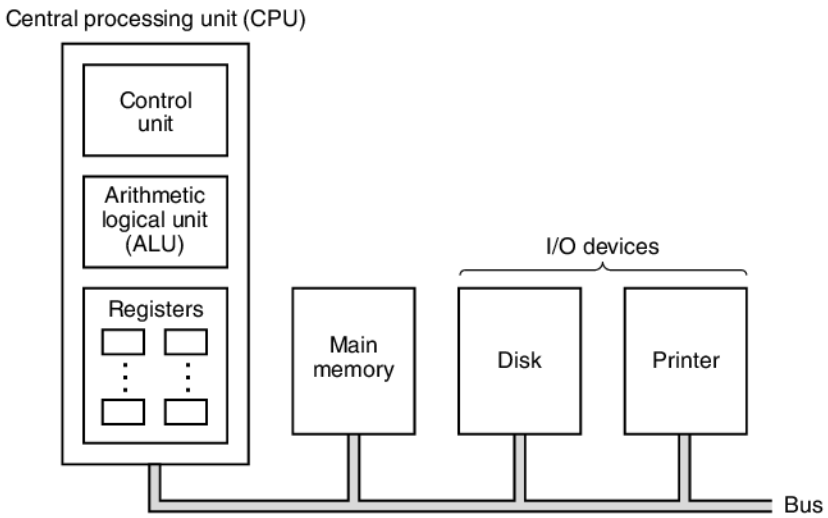

Figure 2-1. The organization of a simple computer with one CPU and two I/O devices.

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

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

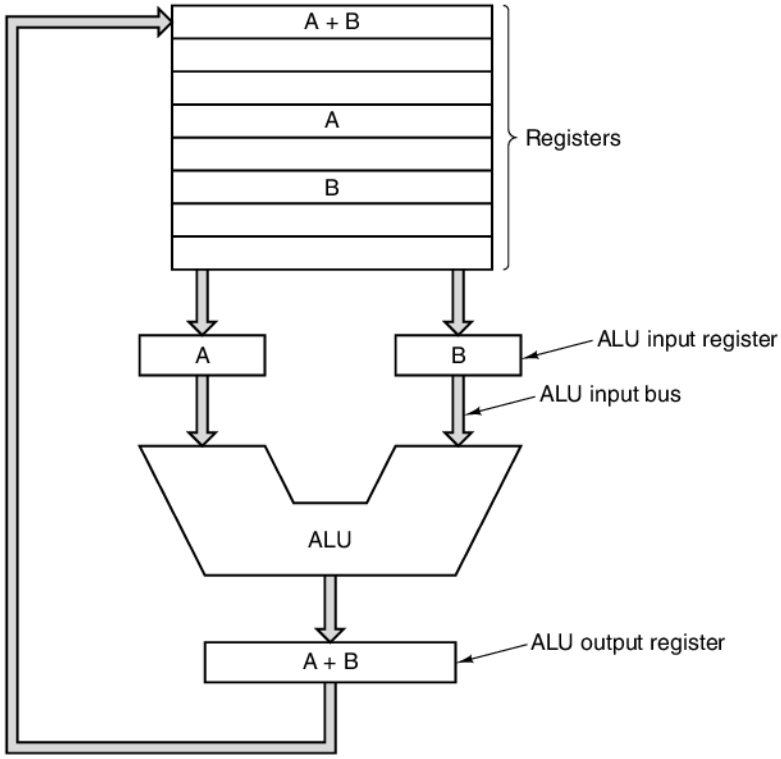

Figure 2-2. The data path of a typical von Neumann machine.

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),

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.

```
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

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.

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.

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

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.

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.

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.

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.

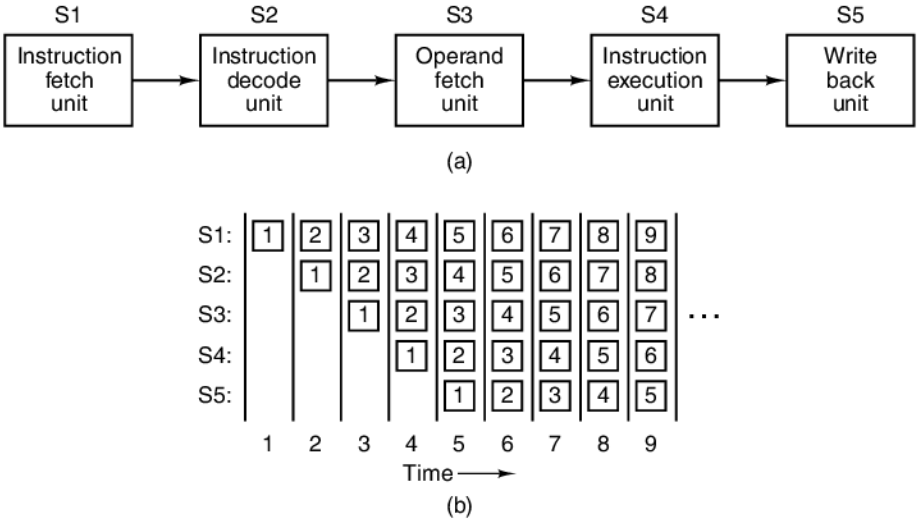

Figure 2-4. (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

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.

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.

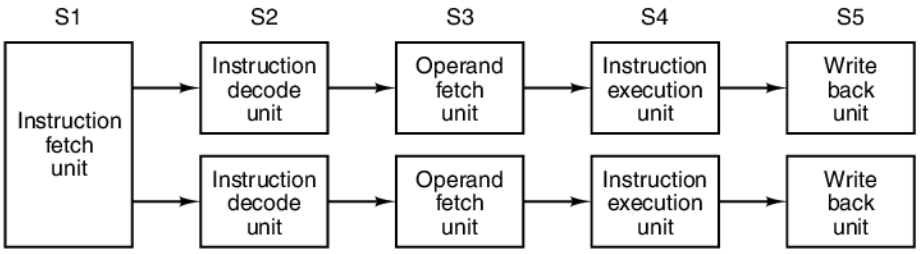

Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.

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

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

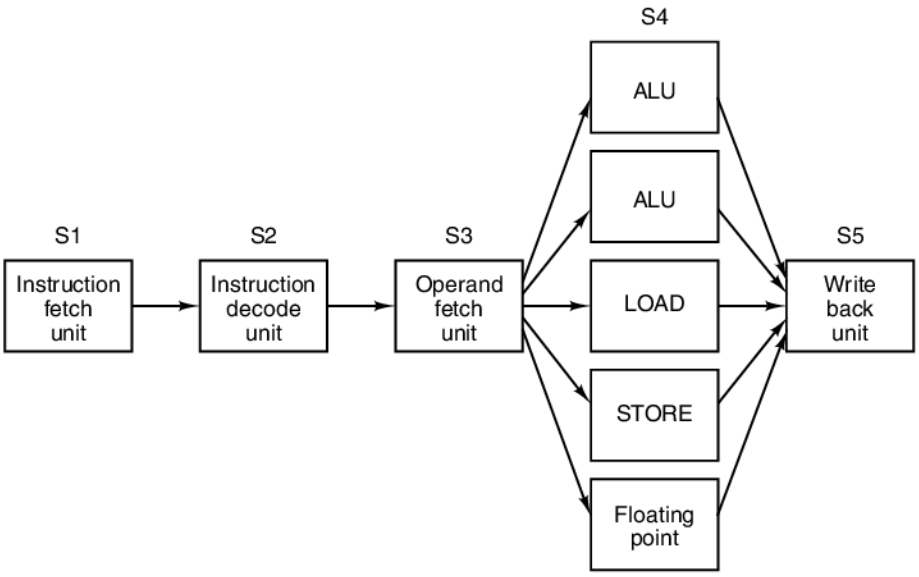

Figure 2-6. A superscalar processor with five functional units.

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.

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

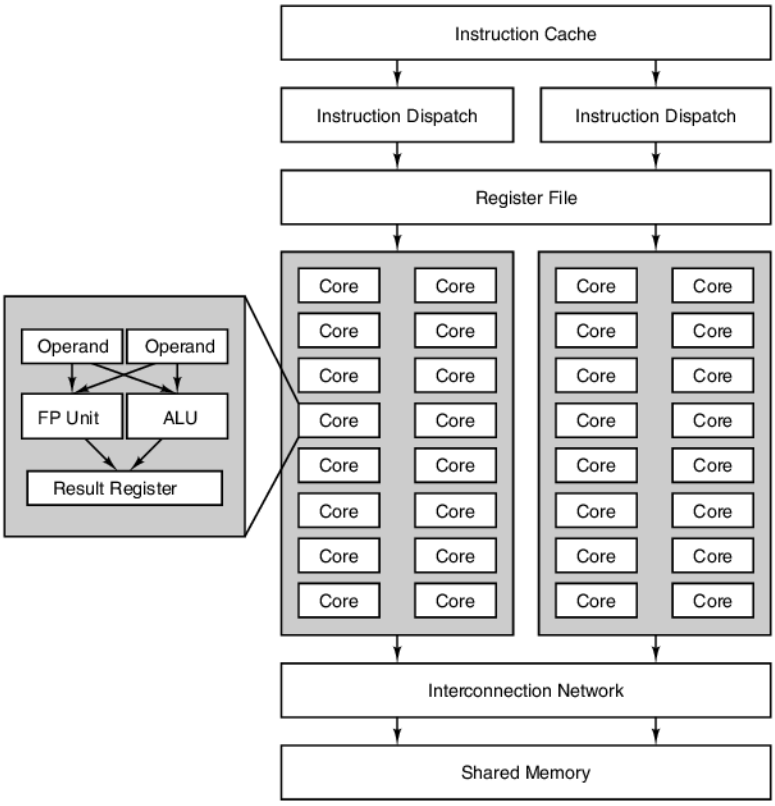

Figure 2-7. The SIMD core of the Fermi graphics processing unit.

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**,

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).

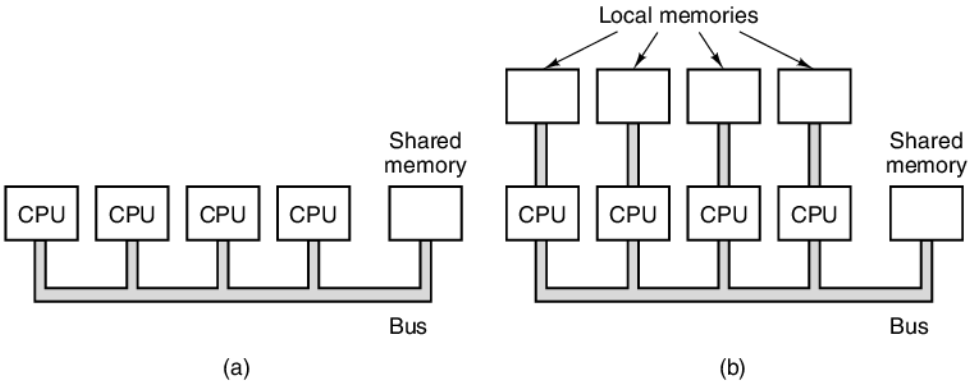

Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

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

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

to disk storage. Without a memory from which the processors can read and write information, there would be no stored-program digital computers.

2.2.1 Bits

The basic unit of memory is the binary digit, called a **bit**. A bit may contain a 0 or a 1. It is the simplest possible unit. (A device capable of storing only zeros could hardly form the basis of a memory system; at least two values are needed.)

People often say that computers use binary arithmetic because it is "efficient." What they mean (although they rarely realize it) is that digital information can be stored by distinguishing between different values of some continuous physical quantity, such as voltage or current. The more values that must be distinguished, the less separation between adjacent values, and the less reliable the memory. The binary number system requires only two values to be distinguished. Consequently, it is the most reliable method for encoding digital information. If you are not familiar with binary numbers, see Appendix A.

Some computers, such as the large IBM mainframes, are advertised as having decimal as well as binary arithmetic. This trick is accomplished by using 4 bits to store one decimal digit using a code called **BCD** (**Binary Coded Decimal**). Four bits provide 16 combinations, used for the 10 digits 0 through 9, with six combinations not used. The number 1944 is shown below encoded in decimal and in pure binary, using 16 bits in each example:

decimal: 0001 1001 0100 0100 binary: 0000011110011000

Sixteen bits in the decimal format can store the numbers from 0 to 9999, giving only 10,000 combinations, whereas a 16-bit pure binary number can store 65,536 different combinations. For this reason, people say that binary is more efficient.

Consider, however, what would happen if some brilliant young electrical engineer invented a highly reliable electronic device that could directly store the digits 0 to 9 by dividing the region from 0 to 10 volts into 10 intervals. Four of these devices could store any decimal number from 0 to 9999. Four such devices would provide 10,000 combinations. They could also be used to store binary numbers, by only using 0 and 1, in which case, four of them could store only 16 combinations. With such devices, the decimal system would obviously be more efficient.

2.2.2 Memory Addresses

Memories consist of a number of **cells** (or **locations**), each of which can store a piece of information. Each cell has a number, called its **address**, by which programs can refer to it. If a memory has n cells, they will have addresses 0 to $n - 1$. All cells in a memory contain the same number of bits. If a cell consists of k bits,

it can hold any one of 2^k different bit combinations. Figure 2-9 shows three different organizations for a 96-bit memory. Note that adjacent cells have consecutive addresses (by definition).

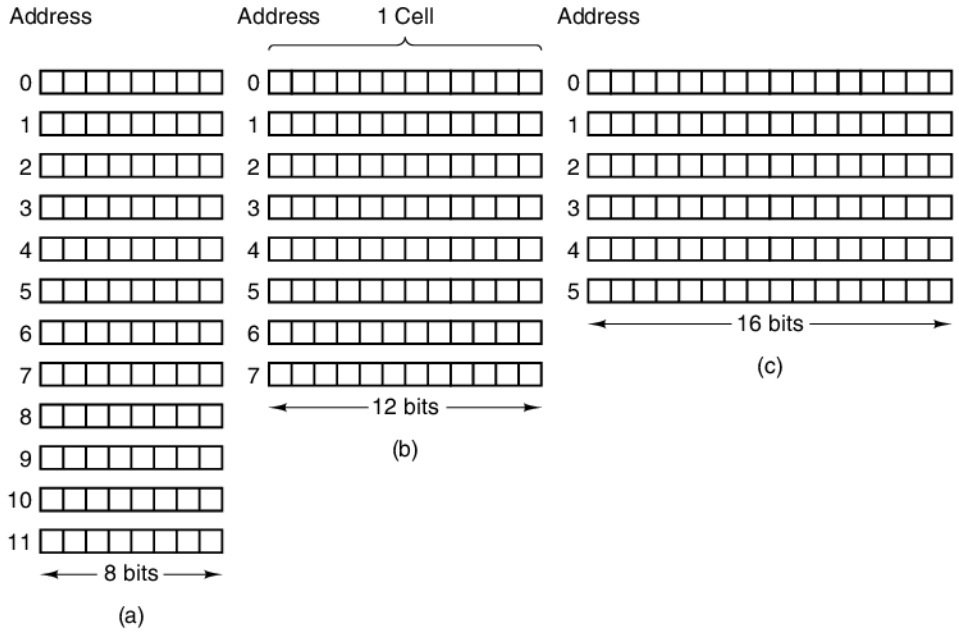

Figure 2-9. Three ways of organizing a 96-bit memory.

Computers that use the binary number system (including octal and hexadecimal notation for binary numbers) express memory addresses as binary numbers. If an address has m bits, the maximum number of cells addressable is 2^m. For example, an address used to reference the memory of Fig. 2-9(a) needs at least 4 bits in order to express all the numbers from 0 to 11. A 3-bit address is sufficient for Fig. 2-9(b) and (c), however. The number of bits in the address determines the maximum number of directly addressable cells in the memory and is independent of the number of bits per cell. A memory with 2^{12} cells of 8 bits each and a memory with 2^{12} cells of 64 bits each need 12-bit addresses.

The number of bits per cell for some computers that have been sold commercially is listed in Fig. 2-10.

The significance of the cell is that it is the smallest addressable unit. In recent years, nearly all computer manufacturers have standardized on an 8-bit cell, which is called a **byte**. The term **octet** is also used. Bytes are grouped into **words**. A computer with a 32-bit word has 4 bytes/word, whereas a computer with a 64-bit word has 8 bytes/word. The significance of a word is that most instructions operate on entire words, for example, adding two words together. Thus a 32-bit machine will have 32-bit registers and instructions for manipulating 32-bit words,