Appendix

# Overview of Computer System and Architecture Concepts

In this appendix:

In this appendix, we give an overview of computer architecture concepts, with an emphasis on those concepts that are particularly relevant to OSs. Some readers will have already completed a course in computer organization or computer architecture, and hence will be familiar with these concepts. In this case, the appendix can provide a review of this material. For those who have not had a previous course in this topic, this appendix might be covered in detail, because the discussions of many OS concepts are based on the underlying computer architecture. The concepts presented here are needed throughout the presentation of OS concepts.

We start by giving a description of the major components of a typical computer system in Section A.1, and a discussion of the functions performed by each component. In Section A.2 we discuss the central processing unit and control concepts. Section A.3 outlines the ideas of memory and storage hierarchy, and Section A.4 describes the basic concepts of input/output systems. Section A.5 briefly discusses the role and characteristics of networks in modern computing. We then give a more detailed picture of typical computer system components in Section A.6. Finally, Section A.7 provides a summary.

## A.1 TYPICAL COMPUTER SYSTEM COMPONENTS

Computer systems vary widely, based on their functionality and expected use. They include the following types of systems:

**Personal** desktop and notebook computers that are typically utilized by a single user at a time.

Large **server** computers that provide services to hundreds or thousands of users each day. These include Internet **Web servers** that store Web documents, **database servers** that store large databases, **file servers** that store and manage files for a network of computers, and application servers that run some specific application that provides a remotely accessed service.

**Embedded** computer systems, which are used in automobiles, aircraft, telephones, calculators, appliances, media players, game consoles, computer network units, and many other such devices. As CPU chips have become cheaper and cheaper we see them in more and more places. In the future we will see them in places that might be hard to imagine today.

**Mobile** wearable devices, cell phones, and PDAs (personal digital assistants) that are used for keeping appointment calendars, email, phone directories, and other information. Today these units are becoming hard to distinguish from personal computers as they become more and more powerful.

Hence, it is difficult to decide what a typical computer system would look like. However, it is traditionally accepted that most computer systems have three major components, as illustrated in Figure A.1. These are the processor or central processing unit, the memory unit, and the input/output units.[1] In addition to the three major components, network devices connect computer systems together and allow sharing of information and programs. Let us briefly describe the main functionality of each of these units.

The **central processing unit** (or **CPU**) is the circuitry that performs the computation and control logic required by a computer system. The **memory** is the component that stores both the data required by a computation and the actual commands that perform the computation. Memory is often organized into several levels, leading to a **storage hierarchy** of different types of storage devices, as we describe in Section A.3. The class of **input/output** (or **I/O**) units include two broad subclasses of devices based on their major functionality: input and output. Some devices can also be used for both input and output. **Input devices** are used to load data and program instructions into the memory unit from devices such as CD-ROMs or disks. They are also used to process input commands from a user through devices such as a keyboard or pointing device (e.g., a mouse or touchpad). **Output devices** are used to display data and information to the user through devices such as printers or

[1] At a more detailed level, the CPU is sometimes separated into two components: the control unit and the data path unit, as we discuss in the next section. Similarly, the input/output unit is sometimes separated into input devices and output devices.

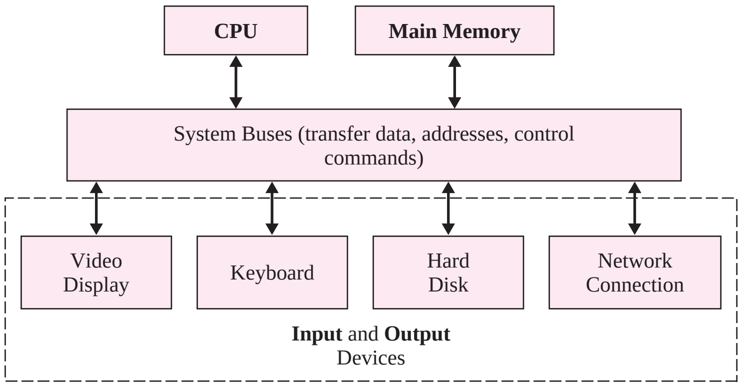


**FIGURE A.1**
Simplified diagram of the major computer system components.

video monitors, and to store data and programs on secondary storage devices such as various types of disks. Devices such as hard disk drives and CD-RW drives are used for *both input and output,* and hence are classified as **input/output devices.** Network devices can be considered as input/output devices but they are so special that they are best regarded as being something separate.

Disk devices in general (hard disk, floppy disk, CD, etc.) are considered as I/O devices if we consider a low-level hardware view of the computer system. If we take a more conceptual view of the roles they play, which is to store data and programs, then they can also be considered as part of the storage hierarchy of the computer system, as we discuss in Section A.3.

Another crucial component in many modern computer systems is the **network,** which is the hardware and software that allows the millions of computers and network devices in existence to communicate with one another. Networks can be formed from phone lines, fiber optic and other types of cables, satellites, wireless hubs, infrared devices, and other components. At the individual machine level, though, it is sometimes useful to consider the network as another type of input/output device, because its main functionality is to transfer data (such as files, text, pictures, commands, etc.) from one machine (as output) to another machine (as input). For computer users, the Internet is the most visible example of a network.

The following three sections discuss each of the three main computer system components—processor, memory, and input/output—in more detail. The network is discussed in Section A.5.

## A.2 THE PROCESSOR OR CENTRAL PROCESSING UNIT

As was said before, the central processing unit, or CPU, is the hardware circuitry that performs the various arithmetic and logical operations. Each processor will have a particular **instruction set** that defines the operations that can be performed by the processor. These typically include integer arithmetic operations, comparison operations, transfer operations, control operations, and so on. A processor usually has a set of *registers* that hold the operations that are being executed as well as some of the data values or *operands* needed by these operations.[2] Other operands can be accessed directly from memory locations, depending on the design of the instruction set. We further elaborate on the use of registers and the types of operands later in this section.

Instruction sets can vary widely. Some processors are designed based on the **RISC** (reduced instruction set computer) philosophy, where only a few basic instruction types are directly implemented in hardware. These instructions are usually similar to one another in their design. One of the advantages of RISC is to reduce hardware complexity by having a limited set of instruction types and hence increase the speed of execution of the instructions. The most common RISC microprocessors are the HP Alpha series (no longer being manufactured, but historically significant), ARM-embedded processors, MIPS, the PIC microcontroller family, the Apple/IBM/Motorola PowerPC and related designs, and the Sun Microsystems SPARC family.

Other processors have a much larger instruction set implemented directly in hardware, with a variety of instruction types included in the instruction set. This approach is known as **CISC** (complex instruction set computer). A RISC processor typically has between 30 and 100 different instructions with a fixed instruction format of 32 bits. A CISC processor typically has between 120 and 400 different instructions. Examples of CISC processors are the IBM System/360, DEC VAX, DEC PDP-11, the Motorola 68000 family, and Intel x86 architecture–based processors and compatible CPUs.

Most of today's processors are not completely RISC or completely CISC. The two are really design philosophies that have evolved toward each other so much that there is no longer a clear distinction between the approaches to increasing performance and efficiency. Chips that use various RISC instruction sets have added more instructions and complexity so that now they are as complex as their CISC counterparts and the debate is mostly among marketing departments.

### A.2.1 Instruction set architecture: The machine language

The instruction set architecture defines the **machine language** of the processor, which is the set of commands that the processor can directly execute. Each instruction is coded as a sequence of bits (a **bit string**) that can be decoded and executed by the processor. Instructions are stored in memory, and are typically executed in sequential order, except when a specific *transfer of control* is specified by some types of instructions. The instruction bit string is divided into several parts called **fields.** Although instruction formats can vary widely, some of the typical fields are the following:

The **opcode** (operation code) field specifies the particular operation to be executed.

[2] The registers that store data values can also be considered, at least conceptually, to be the top level of the storage hierarchy (see Section A.3), since they hold data and provide the fastest access time when accessed by the executing instructions. Physically they are part of the processor chip.

A **modifier field** is sometimes used to distinguish among different operations that have the same opcode and format—for example, integer addition and subtraction.

The **operand fields** specify the data values or addresses that are needed by each particular operation. Addresses can be either memory addresses or register addresses.

There are many different types of operands, and the way to interpret the meaning of each type of operand is called the **addressing mode.** We can distinguish between two main types of operands. The first type supplies a **data value** or the **address of a data value** needed by the operation. The second type provides the **address of an instruction,** and is used for changing the sequence of instruction execution by a **branch** or **jump** operation.

The most common addressing modes for **data** operands are the following:

**Register addressing:** The operand specifies a register location where the data that is needed or produced by the operation is stored.

**Immediate addressing:** The operand is a direct data value contained in one of the fields of the instruction bit string itself.

**Base register addressing:** The operand is stored in a memory location. The address of the memory location is calculated by adding the contents of a **base register** (which contains the address of a reference memory location) and a **displacement** or **offset.** The displacement can be a direct value in the instruction itself, or it could be the value in another register, called an **index register.**

**Indirect addressing:** The memory address of the data to be used as an operand is stored in a register or in another memory location. This is called *indirect addressing* because instead of pointing to the data to be used in an operation the instruction points to the address of the data, either in memory or in a register, and that address must first be accessed to get the actual data address needed.

The most common addressing modes for **instruction address** operands are the following:

**PC-relative addressing:** The memory address of the instruction is calculated by adding an offset to the contents of the **PC** (program counter) register, which holds the address of the next instruction to be executed. As in base register addressing, the offset can be a direct value in the instruction itself, or it could be the value in an index register.

**Indirect addressing:** The memory address of the instruction is stored in a register or in another memory location. As with indirect data addressing, instead of pointing to the address to be transferred to, the instruction points to the address of the address, either in memory or in a register, and that address must first be accessed to get the actual transfer address needed.

Some of these addressing modes are illustrated in Figure A.2.

The type of operation determines how to interpret the operands—whether as memory addresses or instruction addresses or direct data values or in some other way. RISC processors typically have a limited number of addressing modes, whereas CISC processors typically have a much larger variety of addressing modes.

We now illustrate some simple instruction formats and their addressing modes in Figure A.2.

**FIGURE A.2** Illustrating some addressing modes and instruction formats.

**FIGURE A.2(a)** Register addressing for add operation.

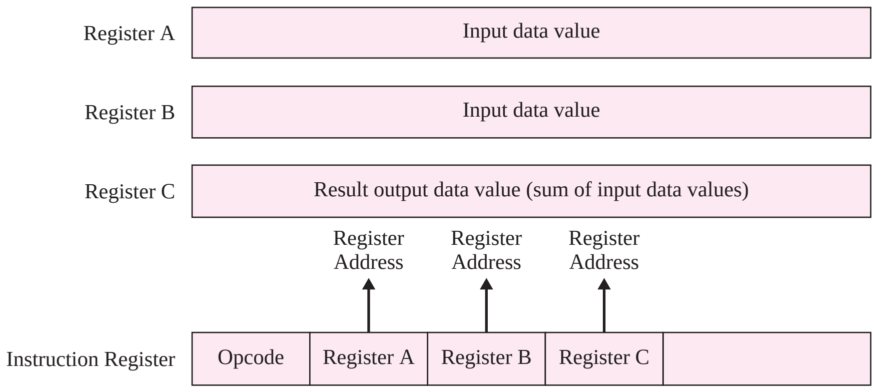


Figure A.2(a) illustrates an **add** operation which places the result of adding the contents of registers A and B into register C. Here the values to be added must first be loaded into registers A and B by previous instructions.

**FIGURE A.2(b)** Register and immediate addressing for add operation.

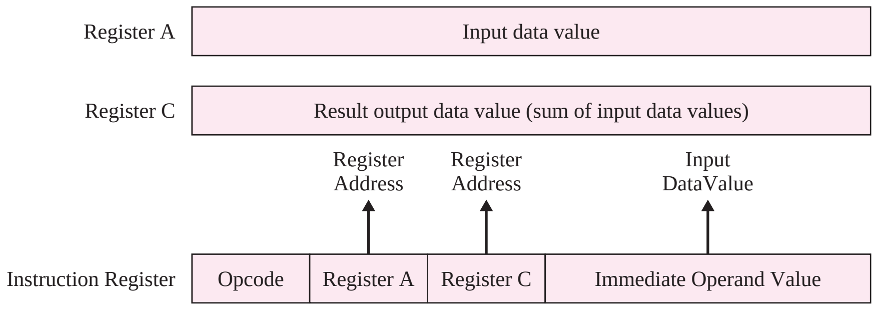


Figure A.2(b) shows an **add** operation where one of the operands is an immediate value stored in the instruction itself. This operation places the result of adding the contents of register A and the immediate operand value into register C.

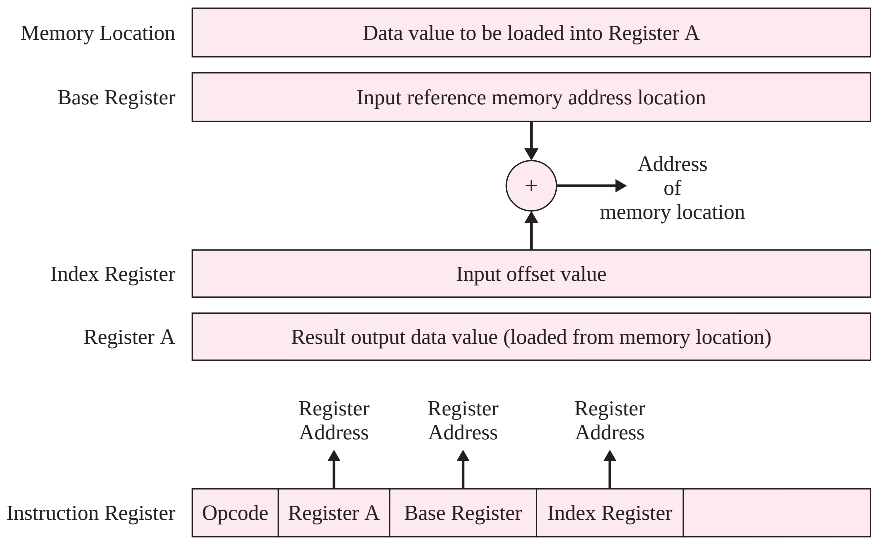


**FIGURE A.2(c)** Base and index registers for load operation.

Figure A.2(c) illustrates a **load** operation that places a value from memory into register A. This instruction uses base register addressing mode to calculate the memory address. The values in the base and index registers are added, and their result is used as the memory address whose contents are loaded into the result register A.

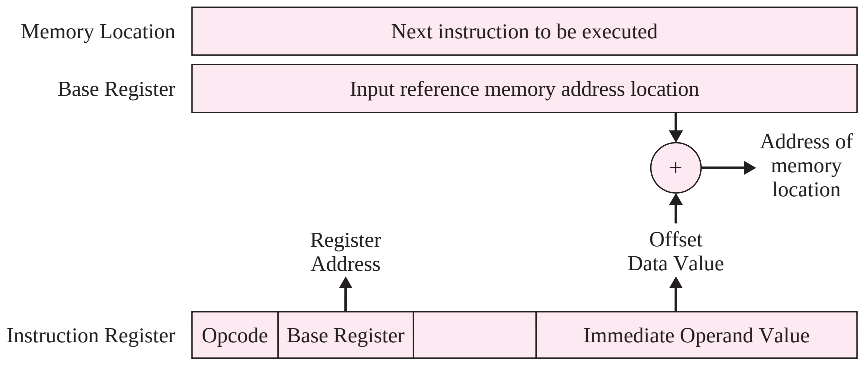


**FIGURE A.2(d)** Immediate relative address for jump operation.

Figure A.2(d) shows an unconditional **jump** operation, which transfers control to an instruction other than the next instruction. It calculates the memory address of the next instruction based on a base register and an immediate value. The next instruction to be executed is in the memory address calculated by adding the base register contents to the immediate index value stored in the instruction itself.

**FIGURE A.2(e)**
Program counter indexed addressing for conditional branch.

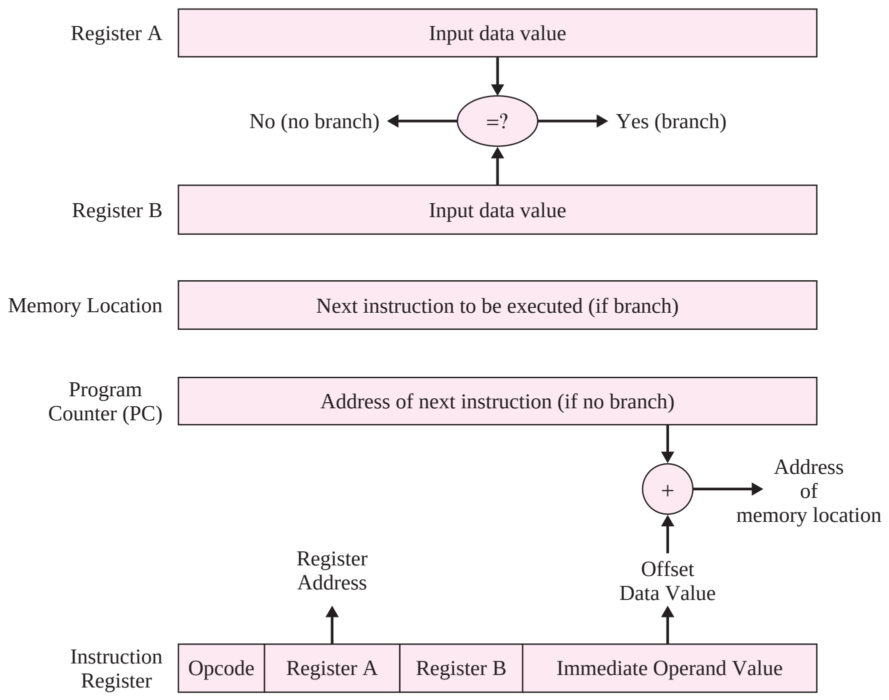


Figure A.2(e) shows a conditional **branch-on-equal** operation. The instruction first compares the values in registers A and B. If the values are equal, it transfers control to the instruction whose address is calculated by adding the contents of the program counter register (the next instruction address) to the immediate value in the instruction. Such an instruction can be used to control looping, for example.

In addition to different addressing modes—which determine how to interpret an operand location and value—many processors have two **execution modes. User mode** is used when a user or application program is executing. **Supervisory** (or **privileged**) **mode** is used when an OS kernel routine is executing. A special register in the processor determines which execution mode is being used. When in user mode, certain safeguards are incorporated during instruction execution. For example, memory protection is enabled in user mode to prohibit the program from accessing memory locations outside of the part of memory allocated to the user program. Certain privileged instructions are allowed to execute only when the system is in supervisory mode—for example, instructions that control I/O devices.

## A.2.2 Components of a CPU

Figure A.3 is a simplified diagram that shows the typical components of a CPU. The **integer ALU** (arithmetic and logic unit) and the **floating point unit** include the hardware circuitry that performs instruction set operations. Most regular instructions are handled by the integer ALU, whereas floating-point arithmetic instructions are handled by the floating point unit, since such operations require more complex, highly specialized circuitry. The **control unit** usually includes the processor registers, as well as circuitry for controlling the sequencing of instruction execution, the interpretation of instruction codes and operands, and the execution of instructions using the ALU or floating point unit circuitry.

The **processor cache** shown in Figure A.3 is a memory component that is part of the processor chip, and holds instructions and data from main memory that are being used by the processor. (There may be other cache memories outside the CPU chip itself.) The cache is connected to main memory via a separate **memory bus.** The main memory is also connected to a main **system bus.** The control unit is connected to the I/O devices through the system bus as well. Another component is a **DMA** (direct memory access) controller. It allows for transfer of data directly from I/O devices to memory. We discuss the idea of caching in some detail in Section A.3.2 and covered DMA in Chapter 14.

## A.2.3 Programs: Source, object, and executable

An **assembly language** is an alternate form of the machine language instructions that is easier to read (and write) by humans. In assembly language, each possible opcode is given a **mnemonic name**—a symbolic name to identify the instruction. The operands are shown as numbers or are also given names to identify program variables that are mapped to a memory address or register. A program that is written in a high-level programming language such as C++ (called the **source code**) is converted into a program in machine language (called the **object code**) by the programming language **compiler.** Such an object code program is then linked with other needed object code programs from program libraries and other program modules, creating an **executable code** program file. This is usually stored on disk as a binary file, and hence is sometimes called the **program binary** (or **bin**) file. The executable code is loaded into memory when needed, and the program instructions are executed in the desired sequence by the processor.

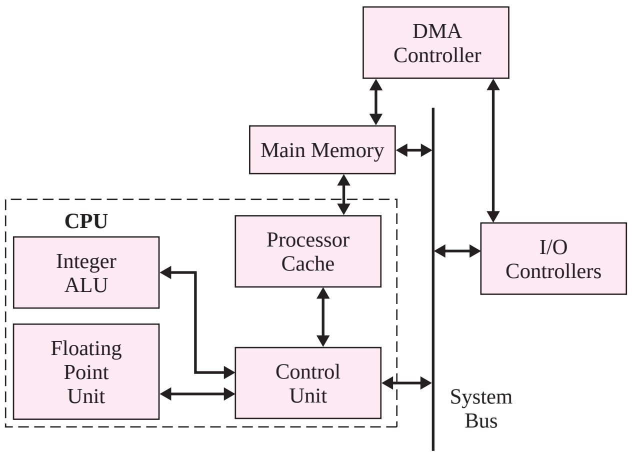


**FIGURE A.3**
Simplified diagram of typical CPU components.

Good programming language compilers should take advantage of the machine instruction set available when creating the object code. Hence, programmers who write compilers must study the instruction set architecture of each machine in detail in order to fully utilize its capabilities.

### A.2.4 Processor registers, data path, and control

There are several types of registers that are part of the CPU. They are used by the processor circuitry in various ways. Some processors use **general-purpose** registers, where the same physical register may be used in many or all of the ways discussed below. In other processor designs some or all of the registers are **special-purpose** registers and can only be utilized for specific functions. The following are the most common uses of registers:

**Instruction registers:** These registers are used to hold the instructions that are being executed. They are directly connected to the control circuitry that interprets the opcode and operands when executing an instruction.

**Program counter:** Also known as the **instruction counter,** this register holds the address of the next instruction to be executed. It is initialized to the address of the first program instruction when the program is loaded into memory and is to start execution. The length of the current instruction is normally added to this register as the instruction is executed in order to fetch the next sequential instruction. Of course, branching or subroutine calls or other transfer of control may alter that sequence.

**Data registers:** These registers hold operands. Some data registers may be dedicated to hold operands of a certain data type; for example, a floating-point register could only hold a floating-point operand. Small CPUs may have only one main data register, typically called an **accumulator.** In some such cases there will be an additional register used for larger operands or remainders of division operations and generically called an **accumulator extension** register.

**Address registers:** These hold values of main memory addresses where operands or instructions are stored. They may hold absolute memory addresses, or relative memory addresses (offsets) that are added to a value in a **base register** to calculate an absolute address. Registers that hold relative addresses are called **index registers.**

**Interrupt registers:** These hold information about interrupt events that may have occurred, as we discuss shortly.

**Program status registers:** These hold control information needed by the CPU. Different machines may have any number of status registers and the contents vary wildly. Examples of the sort of control information that they hold include the following:

- Results of the last comparison operation (i.e., $a > b$, $a = b$ or $a < b$)
- Processor status (i.e., whether it is in user or supervisory mode)
- Error status such as arithmetic overflow, divide by zero, etc.

**Clock:** The clock register is actually a timer that counts down to zero and causes an interrupt. This is known as a **clock** or **timer interrupt,** and can be set by the OS for various reasons. For example, in a multiuser system, the OS typically gives control to a user program for a limited amount of time known as the *time quantum.* By setting a timer interrupt, the OS can interrupt the user program if it is still running at the end of the time quantum and check to see if other programs are waiting to run on the processor. This interrupt may also be used to compute the actual date and time. A CPU usually has a privileged instruction that can only be executed by the OS to load a value into the clock register so that a user program cannot override the OS clock value.

Some registers can be set by user programs, and hence are known as **user-visible** registers. These usually include data, address, and instruction registers. Other registers can only be set by the processor or the OS kernel, such as status and interrupt registers. RISC processors typically have a large number of general-purpose registers because of their uniform instruction set design, whereas CISC processors often have both general-purpose and special-purpose registers. Some types of register use may require special-purpose registers; for example, interrupt registers, program status registers, and instruction registers.

The circuitry to identify the particular instruction (from the opcode) and to execute the instruction using the operands is connected to the instruction register. Since instruction execution involves the transfer of information (opcode, operands, etc.) from registers and memory through the hardware circuitry, it is sometimes referred to as the **data path** component of the processor. On the other hand, the circuitry that controls the fetching of the next instruction and handling of other events such as interrupts (see below) is referred to as the **control** component of the processor.

## A.2.5 System timing

Another important component within each processor is the **system clock.** The operation of most logic circuits proceeds in synchronized steps. At the electronic level this is known as a system clock. (This should not be confused with the CPU register that is used by the OS for timing.) A system **clock cycle** is the fixed shortest time interval during which a processor action can occur. The speed of a processor is determined by how many cycles per second are generated by the system clock. A one-Gigacycle processor will have one billion clock cycles per second. The processor technology and the instruction set design are major factors that determine overall processor speed, because simple instructions take fewer clock cycles to complete than do complex instructions. That is considered one advantage of RISC machines, since the RISC instructions typically execute in a smaller number of clock cycles than will CISC instructions.

## A.2.6 Instruction execution cycle and pipelining

It is customary to divide a typical instruction execution cycle into the following five phases:

**Instruction Fetch:** The instruction is fetched from memory into an instruction register.

**Decode:** The opcode is decoded and the input operand locations are determined.

**Data Fetch:** The operands are fetched from memory if necessary.

**Execute:** The operation is executed.

**Write-back:** The operation output results are stored in the appropriate locations.

Note that the instruction or the operands may be in a cache memory instead of the primary memory. For many simple instructions, each phase typically takes one clock cycle, although this may differ depending on the CPU, the type of instruction, and the addressing modes for the operands. A simple instruction would thus take five clock cycles from start to finish. In order to speed up program execution, most modern processors employ a strategy called **pipelining,** where successive instructions overlap their execution phases. For example, while one instruction is in its write-back phase, the next instruction would be in its execute phase, the following one in its data fetch phase, and so forth. This would work as long as all instructions are executed in sequential order so that their order of execution is known in advance by the processor. A speedup of instruction processing by a factor of five would be realized in this case.

A pipelining processor would have to include provisions for instructions that change the order of execution, such as *branch* and *jump* instructions. A jump will terminate one execution pipeline and start another at a different instruction location. Instructions that have gone through some steps of their execution cycle may have to be cancelled (undone) if a branch is determined after their execution cycle is started. It is also sometimes necessary to delay the pipeline if an instruction needs as its input an operand that is being produced by the previous instruction. Hence, the speedup actually achieved by pipelining must be estimated by averaging the speedup achieved by many different programs.

### A.2.7 Interrupts

An important functionality included in the processor is the **interrupt.** This is particularly relevant to OSs, which use interrupts in various ways, as we see throughout this book. An interrupt is usually an **asynchronous event,** which is an event that can occur at any time, and is hence not synchronized with the system clock and with processor instruction execution cycle. The interrupt signals to the processor that it needs to handle a high-priority event. The processor hardware typically includes one or more **interrupt registers,** which are set by the interrupting event.

Whenever an instruction finishes executing, the control circuitry automatically checks to see whether any event has placed a value in an interrupt register. Hence, interrupts cannot be serviced *during instruction execution*—only between instructions.[3] If so, the **processor state**—which includes the contents of the program counter and any registers that will be used during interrupt processing—is saved into memory and a jump to execute the program code that handles interrupts is

[3] When pipelining is used, interrupts may be checked whenever an instruction completes its execution cycle. Provisions for undoing partially executed subsequent instructions by the processor would be needed.

performed. Once the interrupt handler is done, the system will normally restore the processor state and resume processing the user program from the point at which it was interrupted. The OS may switch to run another program if the interrupt caused the current program to be terminated or suspended.

While processing an interrupt, it is usually the case that lower priority or less important interrupts are disabled until interrupt handling is completed. The OS does this by setting an **interrupt disable** (or **interrupt mask**) register. Depending on the value in that register the system will not check for interrupts for lower priority interrupt levels. Hence, the OS can set this register before starting interrupt processing, and reset it back after completing the interrupt processing.

We can categorize the events that cause interrupts into hardware events and software events. In general, hardware interrupts are asynchronous and software interrupts are synchronous. Typical of the **hardware events** that can cause interrupts are the following:

Some I/O user action has occurred, such as mouse movement or mouse button click or keyboard input. The interrupt handler would retrieve the information about the I/O action, such as mouse coordinates or which character was input from the keyboard.

A disk I/O transfer was completed. The interrupt handler would check to see if other disk I/O operations were pending, and if so initiate the next disk I/O transfer to or from main memory.

A clock timer interrupt has occurred, which allows the OS to allocate the CPU to another program.

The **software events** that can cause interrupts may be further categorized into **traps,** which occur when a program error or violation happens, and **system calls,** which occur when a program requests services from the OS. (For historical reasons a system call interrupt is sometimes called a trap—somewhat confusing.) Some events that cause traps are the following:

A memory protection violation, for example, a program executing in user mode tries to access an area of memory outside of its allowed memory space.

An instruction protection violation, for example, a program executing in user mode attempts to execute an instruction reserved for supervisor mode.

An instruction error such as division by zero.

An arithmetic error such as a floating point overflow.

We discuss in detail how these events and other events that cause interrupts are handled by the OS throughout this book.

### A.2.8 Microprogramming

In some computers complex instructions are implemented as sequences of basic instructions, often using the concept of **microprogramming.** A microprogram is a sequence of basic operations that implement a more complex operation. This sequence is stored in a special microprogram memory in the processor, so that it can be invoked when the complex instruction is to be executed. The microprograms are sometimes referred to as **firmware.** Some CPU architectures, usually CISC, use microprogramming while others do not.

### A.2.9 Processor chip

Historically the CPU was built out of discrete components such as relays, tubes, transistors, or simple integrated circuits. In modern systems the whole processor is typically implemented as a single integrated circuit (chip). The **processor chip** includes the CPU, clock, registers, cache memory, and perhaps other circuitry depending on the particular processor design.

### A.2.10 Multicore chips

In the last few years the manufacturers of CPU integrated circuits have concluded that the demand for ever faster CPUs is slacking off somewhat. They have begun to use the extra space on the chips to provide multiple CPUs in the package. There are various alternative designs regarding placement of cache memories, etc. We talk about these caches in the next section. Although this would appear to be a fairly trivial change, we see in the chapters on memory that it is not at all trivial for the OS. At the present time chips with four CPU cores are fairly common. Predictions call for up to 128 cores in the next few years.

It is difficult to write a program that can effectively use multiple CPUs at the same time. But most users have many programs running at the same time and having multiple CPUs to run them on will mean that they will all run faster. Furthermore, most users use only a few programs, and they are ones that have been highly developed and are prepared to use the multiple CPUs. Such programs include most of the programs we use the most—word processors, spreadsheets, browsers, and so on.

## A.3 THE MEMORY UNIT AND STORAGE HIERARCHIES

### A.3.1 Storage units: Bits, bytes, and words

The memory unit is the hardware that stores the program instructions and operands that are needed by the processor. The basic physical storage unit is a single **bit,** which stores a binary zero (0) or one (1) value. In modern systems, bits are grouped into **bytes** (8 bits), and bytes are grouped into **words** (typically 4 bytes or 8 bytes, though CPUs designed for embedded systems may have 1- or 2-byte words). Normally, the basic unit that will be transferred between the memory unit and the processor is a word. Typically there will be instructions that will allow loading or storing of a single byte or half word. In most systems each byte has a unique **memory address.** Given a particular memory address, the memory circuitry can locate that particular byte in memory. The word containing this byte can then be transferred to or from the processor. Memory bytes or words can also be transferred to or from input/output devices. In many cases, blocks of multiple words are transferred directly.

The word size is usually the standard size for processor registers. A 32-bit processor thus will have standard data items of 32 bits, or 4 bytes. On the other hand, 16-bit processors would have 16-bit data formats as many older PC processors had. Some processors have a 64-bit "double word" data size. At one time this was mostly found in large mainframe computers. Most processors currently are of the 32-bit variety, but today's PCs are switching to a 64-bit format. The size of many operands is also one word size (4 bytes), although some operands can be a single byte or 2 bytes or 8 bytes. The particular opcode will determine the type and size of each operand.

As the basic data word size has increased from 16 to 64 bits, the instruction formats have also increased in size, mainly so that larger memory addresses can be used. Instructions in CISC machines tend to be variable length since it takes only a few bits to specify a register but many to specify a memory address. Depending on the addressing mode, instructions specify anywhere from none to three memory addresses, so the instruction lengths will vary accordingly.

## A.3.2 A storage hierarchy

Most current systems have several levels of storage, often referred to as the **storage hierarchy.** This is illustrated in Figure A.4. The traditional view of a storage hierarchy has three levels: primary, secondary, and tertiary storage. We discuss each of these next.

Primary storage consists of main memory and usually one or more cache memories. Even the processor registers are sometimes considered to be part of the main memory storage hierarchy. Hence, within primary storage, there can be several levels. If we consider the processor **registers** to be part of the memory hierarchy, they would be at the top level. At the next level is a high-speed low-capacity **cache memory,** which is usually included as part of the processor chip itself. There may be additional cache memories outside of the main CPU chip, each slower but larger than the previous level. At a still lower level, a lower-speed but higher-capacity **main memory** is included on one or more separate chips. The cache memory typically uses a

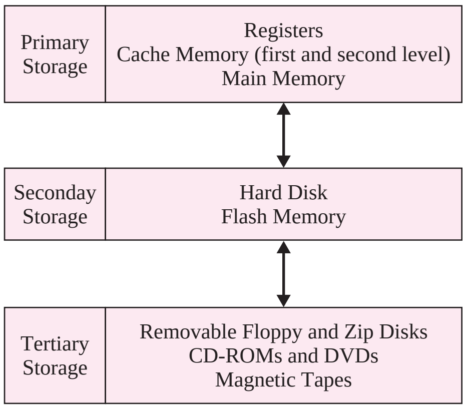


**FIGURE A.4**
A storage hierarchy.

hardware technology known as SRAM (static random access memory), whereas the main memory typically uses DRAM (dynamic random access memory) technology. SRAM technology is faster but more expensive than DRAM per unit of storage.[4]

Processor registers are faster to read or write than cache memory or main memory locations. For example, a register-to-register copy may take a single clock cycle in a RISC processor, whereas a register-to-cache transfer may take two clock cycles, and a register-to-memory transfer might take three or four clock cycles.

The cache memory is often divided into two parts: the **data cache** (for storing operands) and the **instruction cache** (for storing instructions). In some cases there are distinct cache parts for applications in user mode and the kernel in supervisor mode. Transfer of bytes between the cache and processor is several times faster than that between the main memory and the cache. Hence, the goal is to keep in the cache the data and instructions currently being used. This job is the responsibility of the cache management circuitry in the processor, but program design can affect the ability of the hardware to cache the needed instructions and data.

Memory capacity is usually measured in Kilobytes (KB or 1,024 bytes), Megabytes (MB or 1,048,576 bytes), Gigabytes (GB or 1,073,741,824 bytes), and even Terabytes (TB or 1,099,511,627,776 bytes). Since cache is more expensive than main memory it has a much smaller capacity. Many processors have two caches: a level-1 or **L1 cache** on the processor chip and an external level-2 or **L2 cache** outside the processor. A few processors have a third **L3 cache** that is also outside the CPU. The higher-level caches are faster than the lower-level caches but are more expensive and hold less information.

The **memory bus** is the hardware component that handles the transfer of data between main memory (on the memory chip) and cache memory (on the processor chip). Cache memory sizes often are in the 64-KB to several Megabyte range, whereas main memory capacity is typically in the 32-MB to 4-GB range. These numbers continue to grow rapidly, though.

### A.3.3 Secondary storage: Hard disk

The next level in the storage hierarchy is typically a **magnetic disk hard-drive** storage component or simply **hard disk,** which is slower than main memory but has a much higher capacity and lower cost per Megabyte. Hard disk capacity typically ranges between 10-GB to 1-TB or higher, but again these numbers continue to grow rapidly. A hard disk is a part of most standalone computer systems, but is often not included in embedded systems that are used in various devices such as PDAs, music players, telephones, cars, home appliances, and so on. Traditionally, the registers, cache memories, and main memory together are referred to as **primary storage,** whereas the hard disk is referred to as **secondary storage.** Every system must have a primary storage component.

An important distinction between primary and secondary storage is called **storage volatility.** In a **volatile memory,** memory content is lost when electric power

[4] Memory, processor, and disk technologies are always changing, so newer technologies may come in use at any time. We will not discuss further how different types of memories are actually built at the hardware level, since this is not directly relevant to our presentation.

is turned off. In **nonvolatile memory,** content is not lost when power is turned off. Most main memory systems are volatile, whereas most secondary storage systems are nonvolatile. Hence, the disk also serves as a backup storage medium in case of system crashes due to power failure.[5]

At the hardware level, transfer of data between primary and secondary storage involves an I/O device controller, which we discuss in Section A.4. Device controllers often have a storage component to hold data being transferred between the disks and main memory. This storage component is called the **disk cache** or **controller cache.**

This cache is needed because the controller typically has its own processor and clock that are not synchronized with the clock of the CPU. Once the CPU initiates a transfer operation, it leaves the actual control of the transfer to the I/O controller—while the CPU continues with program execution. Hence, main memory is being accessed by both the CPU and the device controllers. Because requests to access memory by the CPU are given higher priority, memory access by the controller may be delayed. The controller cache prevents the loss of data because of such delays by acting as a buffer storage when transferring data from disks and other secondary storage devices to main memory. Controller caches also exist in I/O controllers for some types of tertiary storage devices such as floppy disks and CDs, which we describe next. This type of data transfer between an I/O controller and main memory may make use of DMA technology (direct memory access), which we discussed in Chapter 14.

### A.3.4 Tertiary and offline storage: Removable discs and tapes

Additional levels of the storage hierarchy exist in many computer systems, such as various types of magnetic tape storage for backup, sometimes referred to as tertiary storage or offline storage. In addition, various types of rotating memories (floppy disk, CD-ROM, CD-RW, DVD, etc.)[6] are used as storage media to hold information. The information stored on removable media is generally either too large to fit on secondary storage or is not usually needed frequently or immediately, so it is not permanently kept on the hard disk. So this data is not usually available within the computer system as is the case with cache memory, main memory, and hard disk, which are referred to as **online storage** because they are available as soon as the computer system is turned on.

Removable media units can be automated so that the drive can select from among many individual media that are inserted into the drive. Examples include automated tape libraries or optical disc jukeboxes. In this case they are properly referred to as **tertiary storage.** Removable media storage units that are not automated are usually called **offline storage,** because the storage media (floppy disk, DVD, CD-ROM, tape) must be manually loaded before the data on the media can be accessed. Tertiary and offline storage devices can also be viewed as input/output devices (see Section A.4).

[5] Historically, main memories were not necessarily volatile. Magnetic core primary memory in particular would retain its contents even with the power turned off.

[6] CD-ROM stands for compact disc-read only memory; CD-RW stands for compact disc-read write; and DVD stands for digital video disc.

### A.3.5 Managing the storage hierarchy

Transfer between the various levels of the storage hierarchy is usually done in units of multiple bytes or **blocks** of bytes. The block size between main memory and cache memory is typically in the range of 16 bytes (four words) to 256 bytes (64 words), whereas the block size between hard disk and main memory is typically in the 4-KB to 16-KB range or even higher. The main reason for transferring blocks instead of single bytes or single words is to improve performance by reducing overall transfer time. Especially with tapes there is a large overhead to start and stop the tape movement. So transferring larger blocks with each read or write is much more efficient than transferring smaller blocks. Similarly, positioning a tape or disk to access the needed information is quite slow. Transferring more data at one time means that fewer such positioning operations are needed.

Performance is also improved by taking advantage of the *locality principle,* which states that programs tend to access a small portion of their instructions and operands in any short time interval. This locality characteristic has been shown to exist in most programs, and has two components:

**Temporal locality:** This characteristic states that a program that accesses certain memory addresses may soon access them again. An example is that instructions within a loop may be accessed repeatedly within a short period of time.

**Spatial locality:** This characteristic states that if a program accesses certain memory addresses, it may soon access other words that are stored nearby. For example, instructions are typically stored and accessed sequentially. Another example is that a program may process operands (data) that are stored consecutively—for example, accessing consecutive array elements or sequentially scanning through a block of text that is being edited.

If multiple words that are stored in spatial proximity in a block are loaded into cache memory, then access to subsequent words when needed will be quite fast since they will already be in the cache. These are known as **cache hits.** On the other hand, if these subsequent words are never accessed, the cost of loading them into the cache will be wasted. When instructions or operands are referenced that are not in cache memory, the system will try to locate them in main memory and transfer them to the cache. These are known as a **cache misses.**

If the words that caused a cache miss are not in main memory, they have to be located on hard disk and transferred to main memory, and the needed part is then transferred to cache. Hence, it is necessary to find an appropriate block size that reduces the access cost per unit of storage. Generally, the cost of transferring *n* consecutive bytes or words between one level and the next in a single transfer is much lower than transferring them using multiple transfers. This is particularly true for transfer between hard disk and main memory, and is also true to a lesser extent for transfer between main memory and cache memory. As we will see, a major part of the memory management component of an OS is to attempt to optimize these types of transfers. In general, the OS handles transfers between hard disk and primary memory, and the CPU hardware handles memory-to-cache transfers.

### A.3.6 Memory protection

Another aspect of main memory that is particularly relevant to OSs is the memory protection component. When an executing program references a memory location, the OS needs to make sure that this location is part of the **address space** for that program. It should not allow an application program to make references to memory locations that are being used by other programs or by the OS itself. This protects the OS and other user programs and data from being corrupted by an erroneous or malicious program.

One technique for memory protection is to use a pair of registers, the **base register** and **limit register.** This is illustrated in Figure A.5. Before a program starts execution, the OS sets those registers to delimit the addresses in memory that contain the program address space. Setting the contents of the base and limit registers are privileged instructions that can only be used when the CPU is in supervisory mode in the OS kernel. Once the OS sets the execution mode to user mode and transfers control to the user program, the base and limit registers cannot be changed. Any reference to memory locations outside this range causes a hardware interrupt that indicates an invalid memory reference. The OS will reset the base and limit registers whenever it transfers execution to another program.

In many modern systems a more complex scheme is used. Memory is divided into equal-sized **memory pages.** Typical memory page sizes range from 512 bytes to 4 KB. This technique uses **page tables,** which are data structures that refer to the particular memory pages that can be accessed by the executing user program. Only those memory locations referenced through the page table are accessible to the program. The page table is implemented through hardware support in the processor itself. The commands to load the contents of the page table would be privileged instructions that can only be executed by the OS in supervisory mode in the kernel. We discussed this and other memory protection techniques in detail in Chapters 10 and 11.

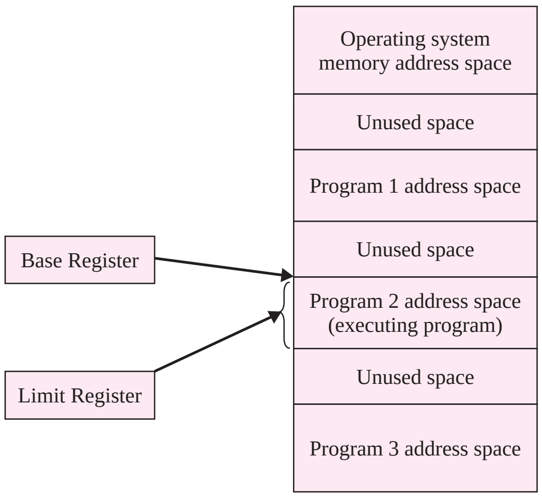


**FIGURE A.5**
A memory protection mechanism using base and limit registers.

## A.4 INPUT AND OUTPUT

The input and output systems are the components that connect the main memory and the processor to other devices. These are sometimes called **I/O devices** or **peripheral devices.**

### A.4.1 Types of I/O devices

I/O devices can be divided into four broad categories: user interface devices, storage devices, network devices, and devices that the computer is controlling.

**User interface I/O devices:** These are employed for user interaction with the computer system. Devices for direct interaction between a user and a system include keyboards, pointing devices (such as mouse, trackball, touch screen, or pad), joysticks, microphones (voice or sound input), other similar components for *input,* and video monitors, speakers (voice or sound output), and the like for *output.* Other I/O devices allow indirect interaction, such as digital cameras and scanners for video or image input, and printers and plotters for hard copy or film output.

**Storage I/O devices:** These are used for *storing information* and hence are considered as both input/output devices and as part of the storage hierarchy. They include magnetic disks (hard or floppy), optical discs/DVD, magnetic tape, flash memory chips, and so on.

**Network I/O devices:** These are devices that connect a computer system to a network, and include analog telephone modems, DSL (digital subscriber line) connections, cable modems, and wired cables. In addition, wireless connections such as infrared or Bluetooth are becoming quite common. They may use a wireless network card installed in a computer or device to connect to a wireless hub, which in turn connects to the network, or they may connect devices directly to a computer.

**Controlled devices:** Computers are often used to control noncomputing devices. Examples include motors, heating and air conditioning, light displays, and so on. Embedded computer systems also fit into this category.

As we can see, there are a wide variety of I/O devices, and new devices are frequently being introduced. To deal with this proliferation of I/O devices, efforts were undertaken to standardize single interfaces that can be used with different types of I/O devices. One such standard is the USB (Universal Serial Bus) 2.0 standard, which allows I/O transmission rates of 480 million bps (bits per second), and is hence suitable for connecting everything from keyboards to digital video cameras or external disk hard drives. Another standard is IEEE 1394, which also allows transmission rates of up to 400 million bps and is used for the same sorts of devices. This interface is also known by two proprietary names, FireWire™ from Apple and i.Link™ from Sony. FireWire is somewhat more efficient than USB for higher-speed devices and is commonly used for video cameras. It has also been selected as the standard connection interface for audio/visual component communication and control.

## A.4.2 Device controllers and device drivers

A **device controller** is a component that interfaces an I/O device to the computer processor and memory. Device controllers frequently contain their own processor, which has a specialized instruction set that is used by device manufacturers to write programs that control the I/O devices. A device controller will also have a **command set,** which is the set of commands that the OS can send to the controller across one of the system buses to control the I/O device. These commands are generally restricted to being used only by OS **device drivers,** and are usually not accessible to application or systems programmers. Many device controllers also have a memory component known as controller cache (see Section A.3.3).

Standard device controllers such as USB and FireWire can be used to connect to any type of I/O device that supports the standard. On the other hand, some specialized device controllers—such as disk controllers or graphics video controllers—can only connect to a single type of I/O device for which it was designed.[7] The controller handles the interfacing with the I/O device and may use its memory to either buffer or cache the data as it is being transferred from or to the computer primary memory. The command set of the controller will include commands that initiate input or output operations. For example, a hard disk controller would have commands to initiate a read-block command for a particular disk block address, while providing the address of the computer memory buffer that will hold the block. Figure A.6 is a simplified diagram to illustrate these concepts.

At the computer side, the OS typically handles all interactions with the device controllers. As was mentioned, the parts of the OS that interact with the device controllers and handle I/O are called the **device drivers.** Each device driver will be programmed to handle the low-level hardware commands and details of a particular device controller. The device driver will present an abstract and uniform view of the device to the rest of the OS.

[7] In some cases, a controller is limited further to a subset of a certain type of device; for example, an ATA controller only works with ATA disk drives rather than all types of disk drives. Sometimes the controller will only work with devices from a single manufacturer or even only with a specific model.

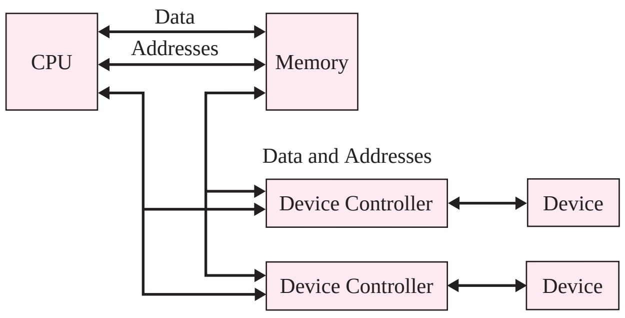


**FIGURE A.6** How I/O devices connect to memory and the CPU through device controllers.

### A.4.3 Other categorizations of I/O devices and connections

There are other ways to categorize I/O devices. One categorization is to divide them into groups based on the type of connection to the computer. I/O devices are typically connected to the memory and CPU at the hardware connection level using either serial or parallel physical connections (usually cables). A **serial** connection transfers bits serially over a single wire, whereas a **parallel** connection typically transfers 8 bits (or more) at a time in parallel over multiple wires. Interfaces to simple I/O devices such as keyboard, mouse, or modem typically use serial connections, whereas higher-speed devices such as some hard disk SCSI (small computer system interface) connections use parallel cables. USB and FireWire controllers use serial cables, but the cables are high grade and shielded, and this permits the high data transfer speeds of these controllers.

Another higher-level categorization of I/O devices is into **block devices** that transfer multiple bytes at a time versus **character devices** that transfer single characters or bytes. Disks are a good example of block devices, whereas a keyboard is an example of a character device.

A third categorization is whether the connection is wired through a cable or wireless. Wireless connections are being used increasingly to connect portable computers to the network or to output devices such as printers.

## A.5 THE NETWORK

Many computers are connected to some kind of network. At an abstract level, one may consider a network connection to be similar to the way that a computer's CPU and memory can be connected to I/O devices. However, the network allows the computer to be connected to other computers, as well as other devices connected to the network. This connectivity allows users to access functions and information on other computers and to use devices that their own computer does not have. It also allows for exchange of information among processes running on different computers.

### A.5.1 Client-server versus peer-to-peer versus multitier models

One common way to look at network interaction is through the **client–server model.** Here, one computer—typically where the user is located—is called the **client.** The client can access one or more **server** computers to access information or other functions that the server provides. Servers might include any of the following:

- database servers that contain large amounts of information
- Web servers that allow the client to access documents on the Internet
- printer servers that allow the user to print on various printers
- file servers that manage user files
- email servers for storing and forwarding email
- servers that support application such as word processing or spreadsheets

Another model for network interaction is the **peer-to-peer model** in which the computers are considered to be equals. For example, the computers could be cooperating toward solving a large computing problem that has been designed to run in a distributed manner over multiple computers on the network.

As distributed systems have evolved it has become necessary to have more complex models than these. Large applications are frequently designed in **multiple tiers.** In a typical three-tier design there will be a front-end that is responsible for the user interface, a middle tier that contains the main logic for the application—often called the business rules—and a database tier that is responsible for all the data storage for the application. In Chapter 17 we discussed the reasons why these more complex architectures have evolved. These models are discussed in greater length in Chapter 15 on networking and Chapters 7 and 17 on distributed processing systems.

## A.5.2 Network controllers, routers, and name servers

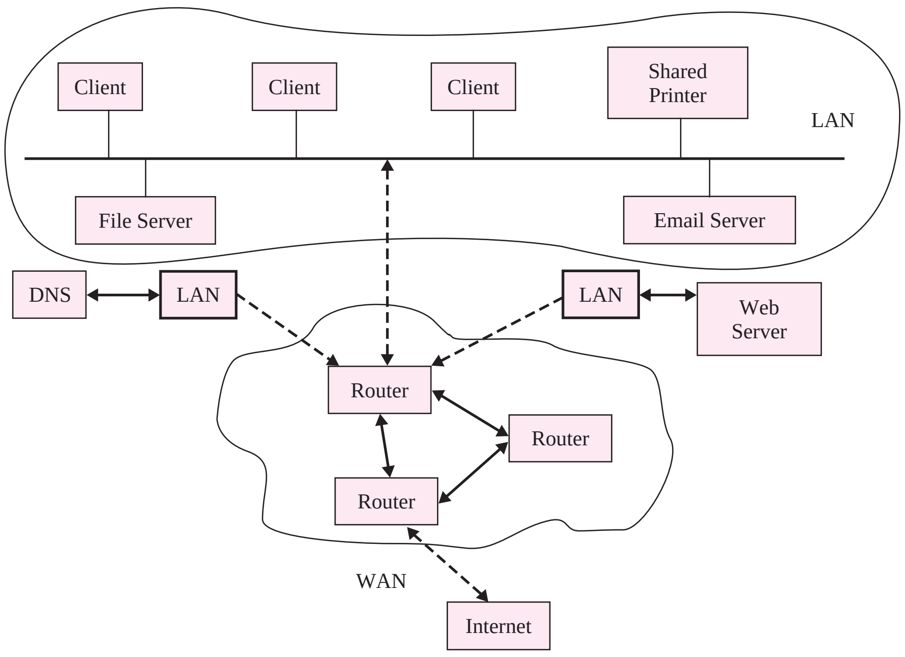
Similar to the manner in which a computer interacts with a device controller that controls an I/O device, the CPU and memory connect to a network through a **network interface controller,** or **NIC.** At the hardware level there are various types of network connections of varying speed, and new technologies for connections are being introduced all the time. Some of the common hardware devices and technologies that connect computers to a network are modems, Ethernet, DSL, cable modems, and several wireless techniques.

At the Physical level, it is useful to distinguish between two types of connections used to build a network: wired and wireless. Hardware for wired networks includes cables or optical fibers of different types, network gateways, routers, switches, hubs, and other similar components. Wireless network components include satellites, base stations for wireless connections, wireless hubs, infrared and Bluetooth ports, and so on.

The network can route a message from its source to its destination through the use of **bridges, switching devices,** or **routers.** To manage the complexity it is common to divide a network within an organization into subnetworks, each connecting a small number of computers via a local area network (LAN). These subnetworks are connected to one another through local routers, which then connect to a regional router, which then connects to the rest of the global network through one or more additional Internet routers.

In the case of the Internet, every computer on the network has a numeric IP (Internet protocol) address (such as 192.168.2.1), which uniquely identifies that computer, and allows the network to route messages addressed to that IP address. Computers also have unique names, such as ourserver.example.com. Specialized servers called **domain name servers** (DNS) have databases that can find a computer's numeric IP address when given its name. The other specialized computers that connect the network, namely the routers and switching devices, can then find a path through the network to deliver a message to the destination computer based on the numeric IP address or the **media access control** (**MAC**) address of the destination. These devices use specific network protocols at various levels to physically deliver the message. Figure A.7 shows a simplified diagram to illustrate these concepts. The techniques for doing this routing and switching are covered in Chapter 15.

**FIGURE A.7**
How a network connects various computers.



## A.5.3 Types of networks

We conclude this brief introduction to networks with a traditional characterization of the types of networks.[8]

**Local area networks** (**LAN**s) are networks that normally connect computers within a limited geographical area, say a group of offices or one building or a number of adjacent buildings within an organization. These networks are primarily built of cables that run through and between the buildings, possibly with switches or routers connecting, say, the various networks on each floor or in each cluster of adjacent offices. Increasingly, wireless access points are being used that allow the connection of a computer with a wireless network card to the local area network.

**Wide area networks** (**WAN**s), on the other hand, generally refer to networks that connect computers over a large geographical area. These use phone lines, fiber optic cables, satellites, and other connections to connect the thousands of local area networks to one another, and hence to allow global connectivity of computers.

**Mobile networks** are made up of thousands of telecommunications towers and control systems that operate as fixed base stations, which are then connected to local or wide area networks. Mobile devices such as cellular phones or handheld computers or PDAs can connect to a nearby base station, which connects it to the rest of the network, and to other parts of the global network.

[8] The technical distinction between a LAN and a WAN is somewhat different. See Chapter 15 for details.

## A.6 A MORE DETAILED PICTURE

We conclude this appendix with Figure A.8, which presents a more detailed picture to illustrate how various system components that we discussed throughout this appendix are connected to one another.

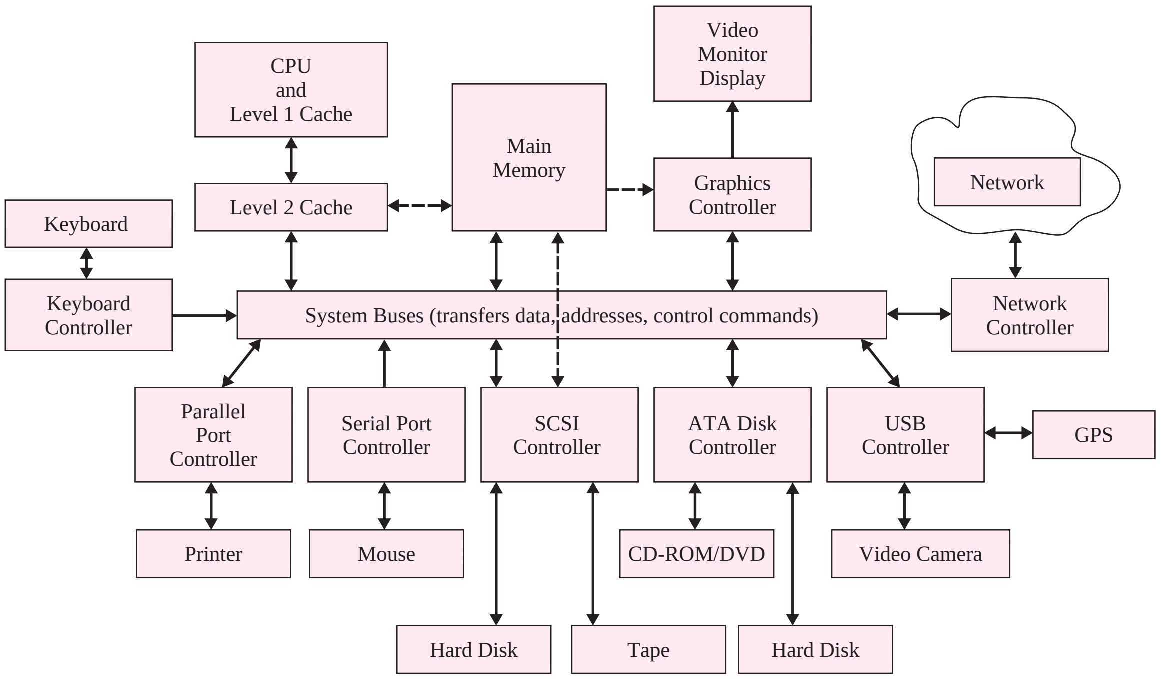


FIGURE A.8 A diagram to illustrate a computer system in some additional detail.

## A.7 SUMMARY

In this appendix, we gave an overview of the basic components of a computer system. We started with a simple overview and a diagram of typical computer system components, and concluded with a more detailed—though still simplified—diagram. In between, we devoted one section to each of the main components of modern-day computers: the processor or CPU, memory and storage hierarchy, I/O devices, and the network. From the discussion, it should be clear that there is overlap between these categories. For example, hard disks can be considered as both an I/O device or as part of the storage hierarchy, and the network interface to a computer can also be abstracted to look like I/O devices. However, the traditional division is useful for structuring our discussion and presentation of computer systems and OSs.

## BIBLIOGRAPHY

Belady, L. A., R. P. Parmelee, and C. A. Scalzi, “The IBM History of Memory Management Technology,” *IBM Journal of Research and Development,* Vol. 25, No. 5, September 1981, pp. 491–503.

Brown, G. E., et al., “Operating System Enhancement through Firmware,” *SIGMICRO Newsletter,* Vol. 8, September 1977, pp. 119–133.

Bucci, G., G. Neri, and F. Baldassarri, “MP80: A Microprogrammed CPU with a Microcoded Operating System Kernel,” *Computer,* October 1981, pp. 81–90.

Chow, F., S. Correll, M. Himelstein, E. Killian, and L. Weber, “How Many Addressing Modes Are Enough?” *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, CA, October 5–8, 1987, pp. 117–122.

Davidson, S., and B. D. Shriver, “An Overview of Firmware Engineering,” *Computer,* May 1978, pp. 21–31.

DeRosa, J., R. Glackemeyer, and T. Knight, “Design and Implementation of the VAX 8600 Pipeline,” *Computer,* Vol. 18, No. 5, May 1985, pp. 38–50.

Elmer-DeWitt, P., and L. Mondi, “Hardware, Software, Vaporware,” *Time,* February 3, 1986, p. 51.

Fenner, J. N., J. A. Schmidt, H. A. Halabi, and D. P. Agrawal, “MASCO: The Design of a Microprogrammed Processor,” *Computer,* Vol. 18, No. 3, March 1985, pp. 41–53.

Foley, J. D., “Interfaces for Advanced Computing,” *Scientific American,* Vol. 257, No. 4, October 1987, pp. 126–135.

Foster, C. C., and T. Iberall, *Computer Architecture,* 3rd ed., New York: Van Nostrand Reinhold, 1985.

Fox, E. R., K. J. Kiefer, R. F. Vangen, and S. P. Whalen, “Reduced Instruction Set Architecture for a GaAs Microprocessor System,” *Computer,* Vol. 19, Issue 10, October 1986, pp. 71–81.

Hunt, J. G., “Interrupts,” *Software—Practice and Experience,* Vol. 10, No. 7, July 1980, pp. 523–530.

Leonard, T. E., ed., *VAX Architecture Reference Manual.* Bedford, MA: Digital Press, 1987.

Lilja, D. J., “Reducing the Branch Penalty in Pipelined Processors,” *Computer,* Vol. 21, No. 7, July 1988, pp. 47–53.

Mallach, E. G., “Emulator Architecture,” *Computer,* Vol. 8, August 1975, pp. 24–32.

Patterson, D. A., “Reduced Instruction Set Computers,” *Communications of the ACM,* Vol. 28, No. 1, January 1985, pp. 8–21.

Patterson, D., and J. Hennessy, *Computer Organization and Design,* 3rd ed., San Francisco, CA: Morgan Kaufmann, 2004.

Patterson, D. A., and R. S. Piepho, “Assessing RISCs in High- Level Language Support,” *IEEE Micro,* Vol. 2, No. 4, November 1982, pp. 9–19.

Patton, C. P., “Microprocessors: Architecture and Applications,” *IEEE Computer,* Vol. 18, No. 6, June 1985, pp. 29–40.

Pohm, A. V., and T. A. Smay, “Computer Memory Systems,” *Computer,* October 1981, pp. 93–110.

Rauscher, T. G., and P. N. Adams, “Microprogramming: A Tutorial and Survey of Recent Developments,” *IEEE Transactions on Computers,* Vol. C-29, No. 1, January 1980, pp. 2–20.

Smith, A. J.; “Cache Memories,” *ACM Computing Surveys,* Vol. 14, No. 3, September 1982, pp. 473–530.

## WEB RESOURCES

http://books.elsevier.com/companions/1558606041/ (Hennessy and Patterson)

http://en.wikipedia.org/wiki/Cache

## REVIEW QUESTIONS

**A.1** What are the two major classes of CPU design?

**A.2** What is the importance of the instruction set architecture to a discussion of the design and development of OSs?

**A.3** Why is a system hardware timer important to an OS?

**A.4** What is the purpose of an interrupt?

**A.5** What is the significance of multicore CPU chips?

**A.6** True or false? Primary storage in computers is always made up of electronic memory circuits.

**A.7** It is hard to overemphasize the importance of caching to the performance of an OS.

**a.** What is the purpose of a cache?

**b.** What theory underlies its function?

**A.8** In theory we could make the cache between secondary storage and primary storage as big as the secondary storage. This would have the advantage of having much smaller latency. Why do we not do this?

**A.9** What is the purpose of memory protection?

**A.10** What is the purpose of having device controllers?

**A.11** In order to help us discuss and understand large complex topics such as I/O devices, we can view the subject as a space with many dimensions. We first discussed a broad division of I/O devices according to the purpose of the device. What were the three broad purposes that were discussed? Give some examples of each class.

**A.12** We also divided the I/O device space into those interfaces that were general-purpose interfaces and those that were for specific device types. Give some examples of each class.

**A.13** DMA controllers cause many fewer interrupts per block transferred to or from a device than do controllers, which do not use DMA. Other than obviously freeing up the CPU to do other things, why do we need controllers that use DMA?

**A.14** What is the function of a device driver and how do we configure OSs with the correct drivers?

**A.15** What facility is used to translate computer names such as omega.example.com to IP addresses for use in the Internet?