

Computer Organization and Software Systems

Text Book(s) :

T1 - Stallings William, Computer Organization & Architecture, Pearson Education, 10thEd. 2013

T2 - A Silberschatz, Abraham and others, Operating Systems Concepts, Wiley Student Edition, 9thEd.

Reference Book(s) & other resources :

R1 - Patterson, David A & J L Hennenssy, Computer Organization and Design –The Hardware/Software Interface, Elsevier, Revised 4th Ed.

R2 - Randal E. Bryant, David R. O'Hallaron, Computer Systems –A Programmer's Perspective, Pearson, 2ndEd, 2016.

Introduction to Computer Systems

● Hardware Organization of a computer(T1: 3.1)

As discussed in Chapter 1, virtually all contemporary computer designs are based on concepts developed by John von Neumann at the Institute for Advanced Studies, Princeton. Such a design is referred to as the *von Neumann architecture* and is based on three key concepts:

- Data and instructions are stored in a single read–write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.
- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

The reasoning behind these concepts was discussed in Chapter 2 but is worth summarizing here. There is a small set of basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting “program” is in the form of hardware and is termed a *hardwired program*.

Now consider this alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (Figure 3.1a). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.

How shall control signals be supplied? The answer is simple but subtle. The entire program is actually a sequence of steps. At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code

and generate control signals (Figure 3.1b).

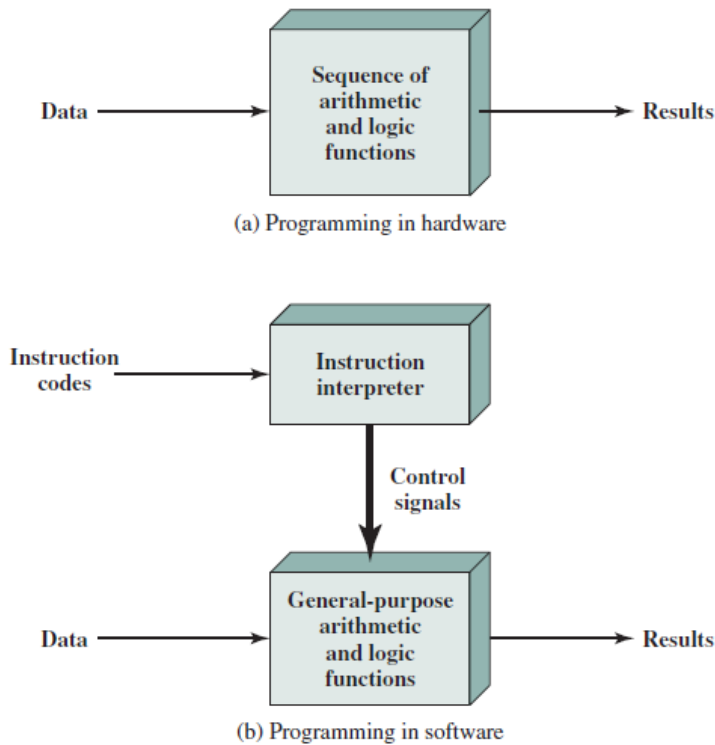


Figure 3.1 Hardware and Software Approaches

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called *software*.

Figure 3.1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU. Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as *I/O components*.

One more component is needed. An input device will bring instructions and data in sequentially. But a program is not invariably executed sequentially; it may jump around (e.g., the IAS jump instruction). Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence. Thus, there must be a place to temporarily store both instructions and data. That module is called *memory*, or *main memory*, to distinguish it from external storage or peripheral devices. Von Neumann pointed out that the same memory could be used to store both instructions and data.

Figure 3.2 illustrates these top-level components and suggests the interactions among them. The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a **memory address register (MAR)**, which specifies the address in memory for the next read or write, and a **memory buffer register (MBR)**, which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the CPU.

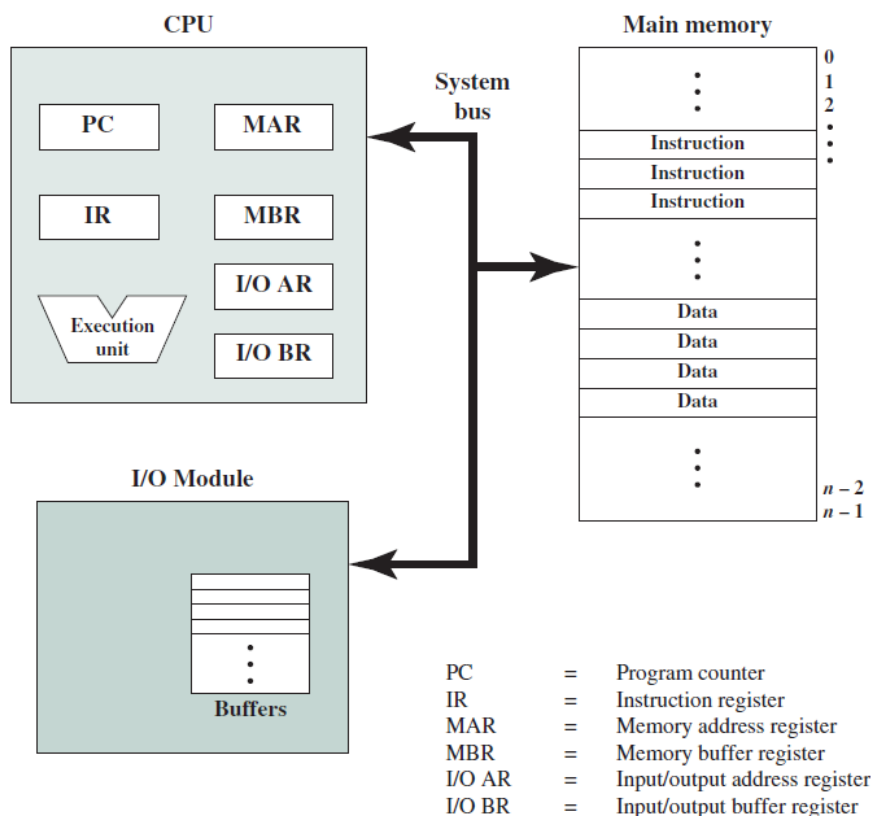


Figure 3.2 Computer Components: Top-Level View

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

Having looked briefly at these major components, we now turn to an overview of how these components function together to execute programs.

Computer Function :

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. This section provides an overview of the key elements of program execution. In its simplest form, instruction processing consists of two steps: The processor reads (*fetches*) instructions from memory one at a time and executes each instruction. Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction (see, for example, the lower portion of Figure 2.4).

• Running a Hello Program (R2: 1.2, 1.4-1.6)

The hello program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run hello.c on the system, the individual C statements must be

translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver*:

```
unix> gcc -o hello hello.c
```

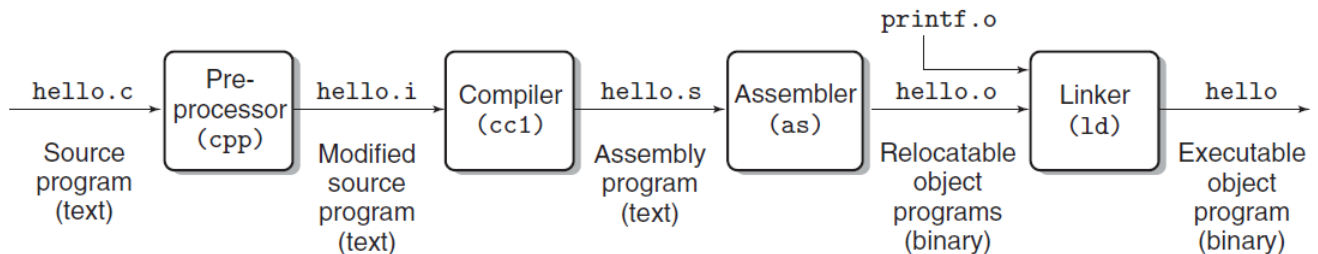


Figure 1.3 The compilation system.

Here, the gcc compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- . *Preprocessing phase*. The preprocessor (`cpp`) modifies the original C program according to directives that begin with the `#` character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.

- . *Compilation phase*. The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. Each statement in an assembly-language program exactly describes one low-level machine-language instruction in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- . *Assembly phase*. Next, the assembler (`as`) translates `hello.s` into machine language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. The `hello.o` file is a binary file whose bytes encode machine language instructions rather than characters. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

- . *Linking phase*. Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an *executable object file* (or simply *executable*) that is ready to be loaded into memory and executed by the system.

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell*:

```
unix> ./hello
hello, world
unix>
```

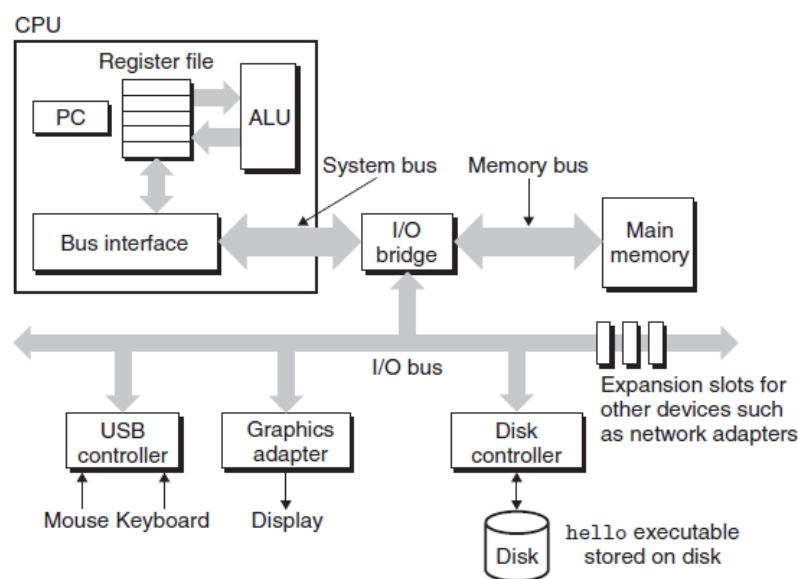
The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the hello program and then waits for it to terminate. The hello program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

Hardware Organization of a System

To understand what happens to our hello program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of Intel Pentium systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Figure 1.4

Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.



Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-sized chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). For the sake of our discussion here, we will assume a word size of 4 bytes, and we will assume that buses transfer only one word at a time.

I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable hello program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main

printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 10, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *dynamic random access memory* (DRAM) chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an IA32 machine running Linux, data of type short requires two bytes, types int, float, and long four bytes, and type double eight bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-sized storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory.

From the time that power is applied to the system, until the time that the power is shut off, a processor repeatedly executes the instruction pointed at by the program counter and updates the program counter to point to the next instruction. A processor *appears to* operate according to a very simple instruction execution model, defined by its *instruction set architecture*. In this model, instructions execute in strict sequence, and executing a single instruction involves performing a series of steps. The processor reads the instruction from memory pointed at by the program counter (PC), interprets the bits in the instruction, performs some simple operation dictated by the instruction, and then updates the PC to point to the next instruction, which may or may not be contiguous in memory to the instruction that was just executed.

There are only a few of these simple operations, and they revolve around main memory, the *register file*, and the *arithmetic/logic unit* (ALU). The register file is a small storage device that consists of a collection of word-sized registers, each with its own unique name. The ALU computes new data and address values.

Here are some examples of the simple operations that the CPU might carry out at the request of an instruction:

Load: Copy a byte or a word from main memory into a register, overwriting the previous contents of the register.

Store: Copy a byte or a word from a register to a location in main memory, overwriting the previous contents of that location.

. *Operate*: Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words, and store the result in a register, overwriting the previous contents of that register.

. *Jump*: Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC.

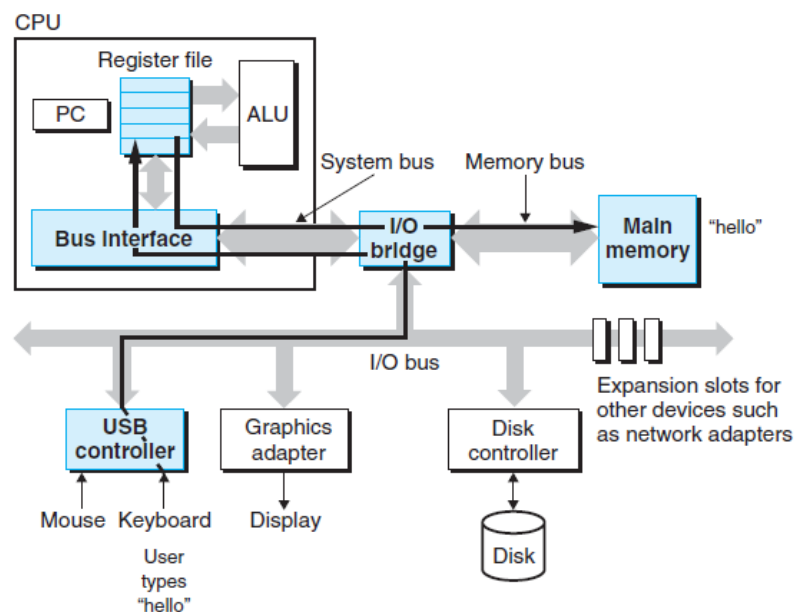
We say that a processor appears to be a simple implementation of its instruction set architecture, but in fact modern processors use far more complex mechanisms to speed up program execution. Thus, we can distinguish the processor's instruction set architecture, describing the effect of each machine-code instruction, from its *microarchitecture*, describing how the processor is actually implemented. When we study machine code in Chapter 3, we will consider the abstraction provided by the machine's instruction set architecture. Chapter 4 has more to say about how processors are actually implemented.

Running the hello Program

Given this simple view of a system's hardware organization and operation, we can begin to understand what happens when we run our example program. We must omit a lot of details here that will be filled in later, but for now we will be content with the big picture.

Initially, the shell program is executing its instructions, waiting for us to type a command. As we type the characters `./hello` at the keyboard, the shell program reads each one into a register, and then stores it in memory, as shown in Figure 1.5.

Figure 1.5
Reading the `hello`
command from the
keyboard.



When we hit the enter key on the keyboard, the shell knows that we have finished typing the command. The shell then loads the executable `hello` file by executing a sequence of instructions that copies the code and data in the `hello` object file from disk to main memory. The data include the string of characters `"hello, world\n"` that will eventually be printed out.

Using a technique known as *direct memory access* (DMA, discussed in Chapter 6), the data travels directly from disk to main memory, without passing through the processor. This step is shown in Figure 1.6.

Once the code and data in the `hello` object file are loaded into memory, the processor begins executing the machine-language instructions in the `hello` program's main routine. These instructions copy the bytes in the `"hello, world\n"` string from memory to the register file, and from there to the display device, where they are displayed on the screen. This step is shown in Figure 1.7.

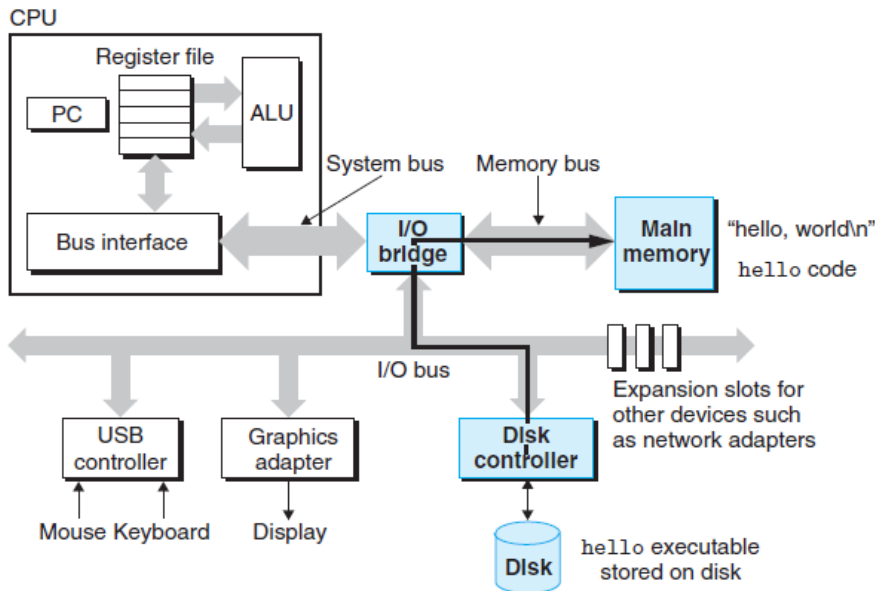
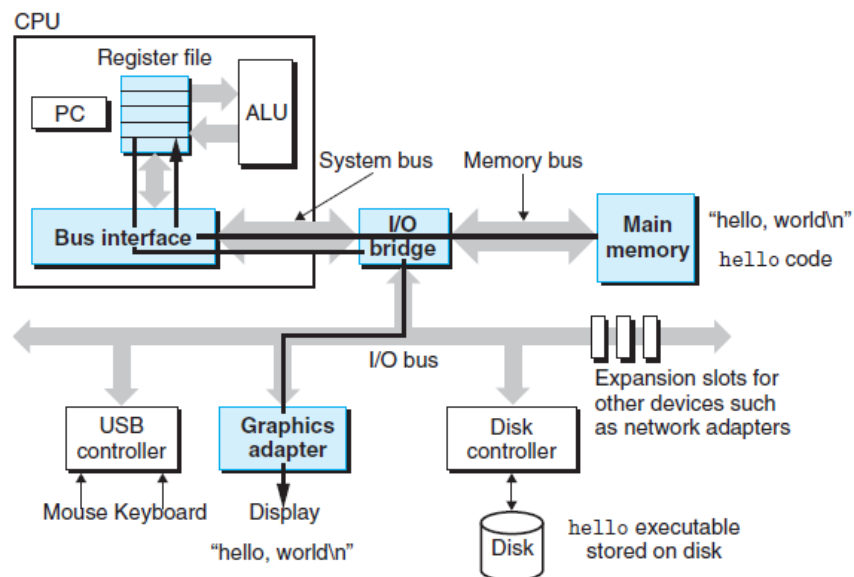


Figure 1.6 Loading the executable from disk into main memory.

Figure 1.7
Writing the output string
from memory to the
display.



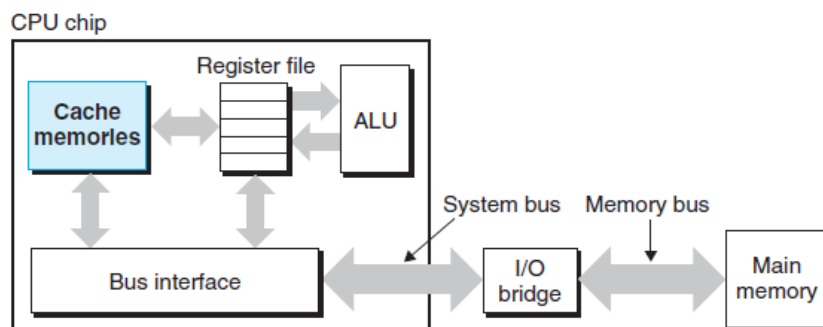
Caches Matter

An important lesson from this simple example is that a system spends a lot of time moving information from one place to another. The machine instructions in the hello program are originally stored on disk. When the program is loaded, they are copied to main memory. As the processor runs the program, instructions are copied from main memory into the processor. Similarly, the data string "hello,world\n", originally on disk, is copied to main memory, and then copied from main memory to the display device. From a programmer's perspective, much of this copying is overhead that slows down the "real work" of the program. Thus, a major goal for system designers is to make these copy operations run as fast as possible.

Because of physical laws, larger storage devices are slower than smaller storage devices. And faster devices are more expensive to build than their slower counterparts. For example, the disk drive on a typical system might be 1000 times larger than the main memory, but it might take the processor 10,000,000 times longer to read a word from disk than from memory.

Similarly, a typical register file stores only a few hundred bytes of information, as opposed to billions of bytes in the main memory. However, the processor can read data from the register file almost 100 times faster than from memory. Even more troublesome, as semiconductor technology progresses over the years, this *processor-memory gap* continues to increase. It is easier and cheaper to make processors run faster than it is to make main memory run faster.

Figure 1.8
Cache memories.



To deal with the processor-memory gap, system designers include smaller faster storage devices called *cache memories* (or simply caches) that serve as temporary staging areas for information that the processor is likely to need in the near future. Figure 1.8 shows the cache memories in a typical system. An *L1 cache* on the processor chip holds tens of thousands of bytes and can be accessed nearly as fast as the register file. A larger *L2 cache* with hundreds of thousands to millions of bytes is connected to the processor by a special bus. It might take 5 times longer for the process to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory. The L1 and L2 caches are implemented with a hardware technology known as *static random access memory* (SRAM). Newer and more powerful systems even have three levels of cache: L1, L2, and L3. The idea behind caching is that a system can get the effect of both a very large memory and a very fast one by exploiting *locality*, the tendency for programs to access data and code in localized regions. By setting up caches to hold data that is likely to be accessed often, we can perform most memory operations using the fast caches.

One of the most important lessons in this book is that application programmers who are aware of cache memories can exploit them to improve the performance of their programs by an order of magnitude. You will learn more about these important devices and how to exploit them in Chapter 6.

● Instruction Cycle State Diagram (T1: 3.2)

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to memory location 300, where the location address refers to a 16-bit word. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.

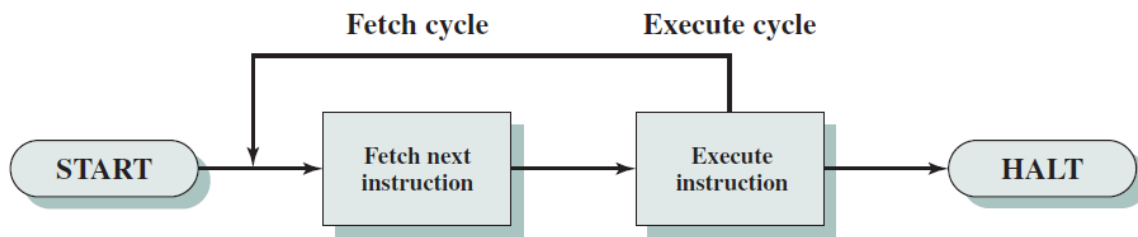


Figure 3.3 Basic Instruction Cycle

The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

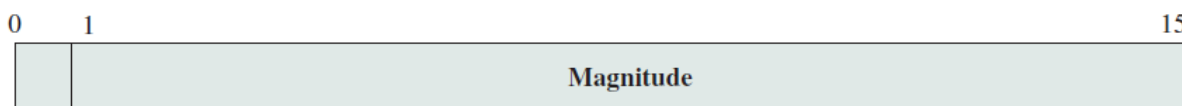
- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.

An instruction's execution may involve a combination of these actions.

Consider a simple example using a hypothetical machine that includes the characteristics listed in Figure 3.4. The processor contains a single data register, called an accumulator (AC). Both instructions and data are 16 bits long. Thus, it is convenient to organize memory using 16-bit words. The instruction format provides 4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.



(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
 Instruction register (IR) = Instruction being executed
 Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
 0010 = Store AC to memory
 0101 = Add to AC from memory

(d) Partial list of opcodes

Figure 3.4 Characteristics of a Hypothetical Machine

Figure 3.5 illustrates a partial program execution, showing the relevant portions of memory and processor registers.¹ The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register. For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301, and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302, and the PC is incremented.
6. The contents of the AC are stored in location 941.

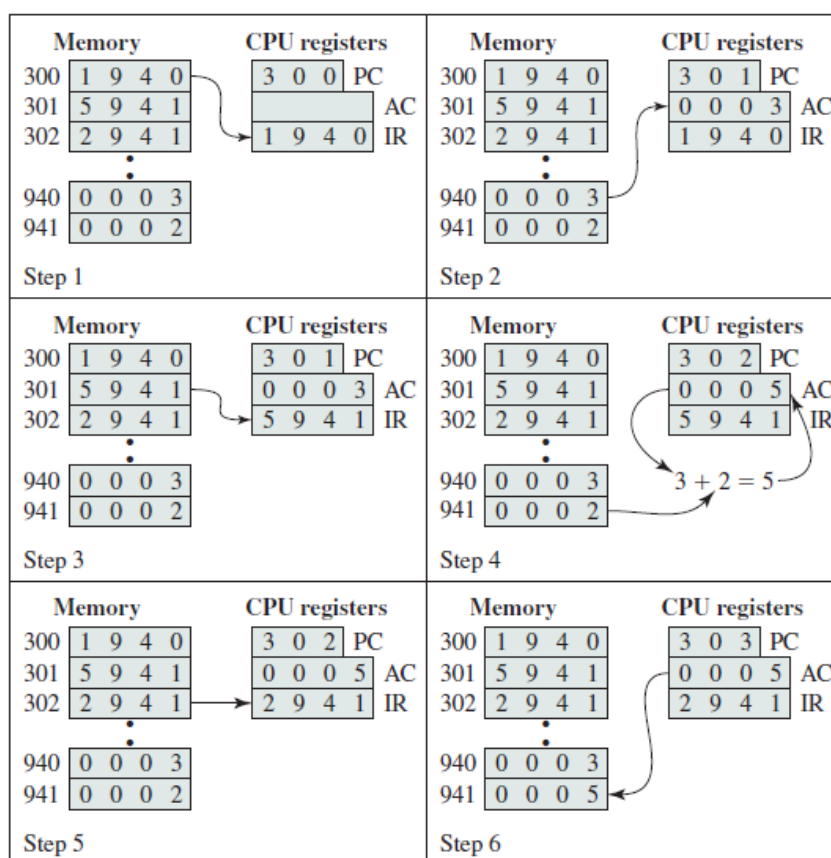


Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer cycles would be needed. Some older processors, for example, included instructions that contain more than one memory address. Thus, the execution cycle for a particular instruction on such processors could involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

For example, the PDP-11 processor includes an instruction, expressed symbolically as ADD B,A, that stores the sum of the contents of memory locations B and A into memory location A. A single instruction cycle with the following steps occurs:

- Fetch the ADD instruction.
- Read the contents of memory location A into the processor.
- Read the contents of memory location B into the processor. In order that the contents of A are not lost, the processor must have at least two registers for storing memory values, rather than a single accumulator.
- Add the two values.
- Write the result from the processor to memory location A.

Thus, the execution cycle for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations in mind, Figure 3.6 provides a more detailed look at the basic instruction cycle of Figure 3.3. The figure is in the form of a state diagram. For any given instruction cycle, some states may be null and others may be visited more than once. The states can be described as follows:

- **Instruction address calculation (iac):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
- **Instruction fetch (if):** Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
- **Data operation (do):** Perform the operation indicated in the instruction.
- **Operand store (os):** Write the result into memory or out to I/O.

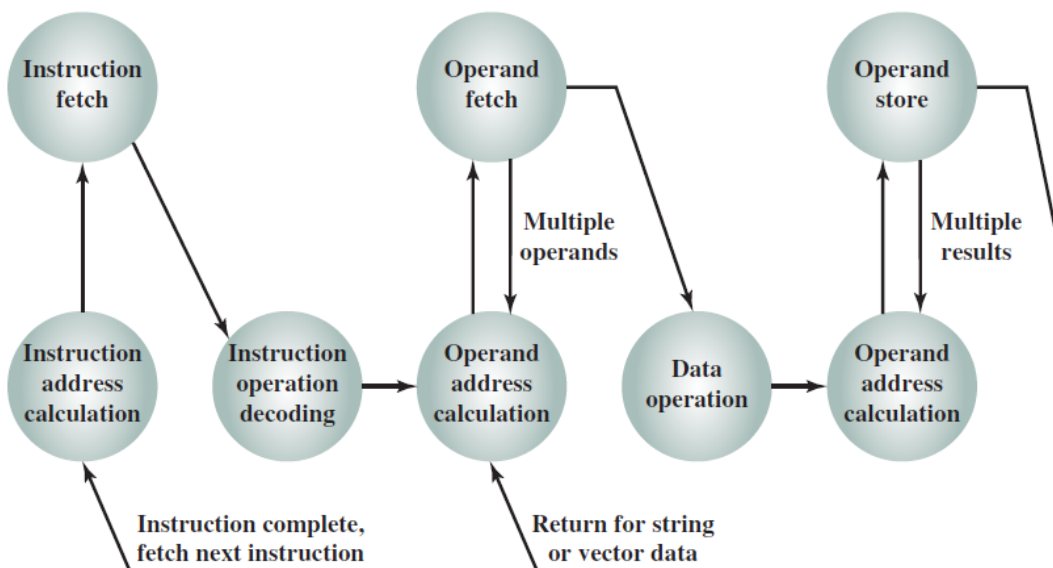


Figure 3.6 Instruction Cycle State Diagram

States in the upper part of Figure 3.6 involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations. The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed.

Also note that the diagram allows for multiple operands and multiple results, because some instructions on some machines require this. For example, the PDP-11 instruction ADD A,B results in the following sequence of states: iac, if, iod, oac, of, oac, of, do, oac, os.

Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) of characters. As Figure 3.6 indicates, this would involve repetitive operand fetch and/or store operations.

Interrupts

Virtually all computers provide a mechanism by which other modules (I/O, memory) may **interrupt** the normal processing of the processor. Table 3.1 lists the most common classes of interrupts. The specific nature of these interrupts is examined later in this book, especially in Chapters 7 and 14. However, we need to introduce the concept now to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure. The reader need not be concerned at this stage about the details of the generation and processing of interrupts, but only focus on the communication between modules that results from interrupts.

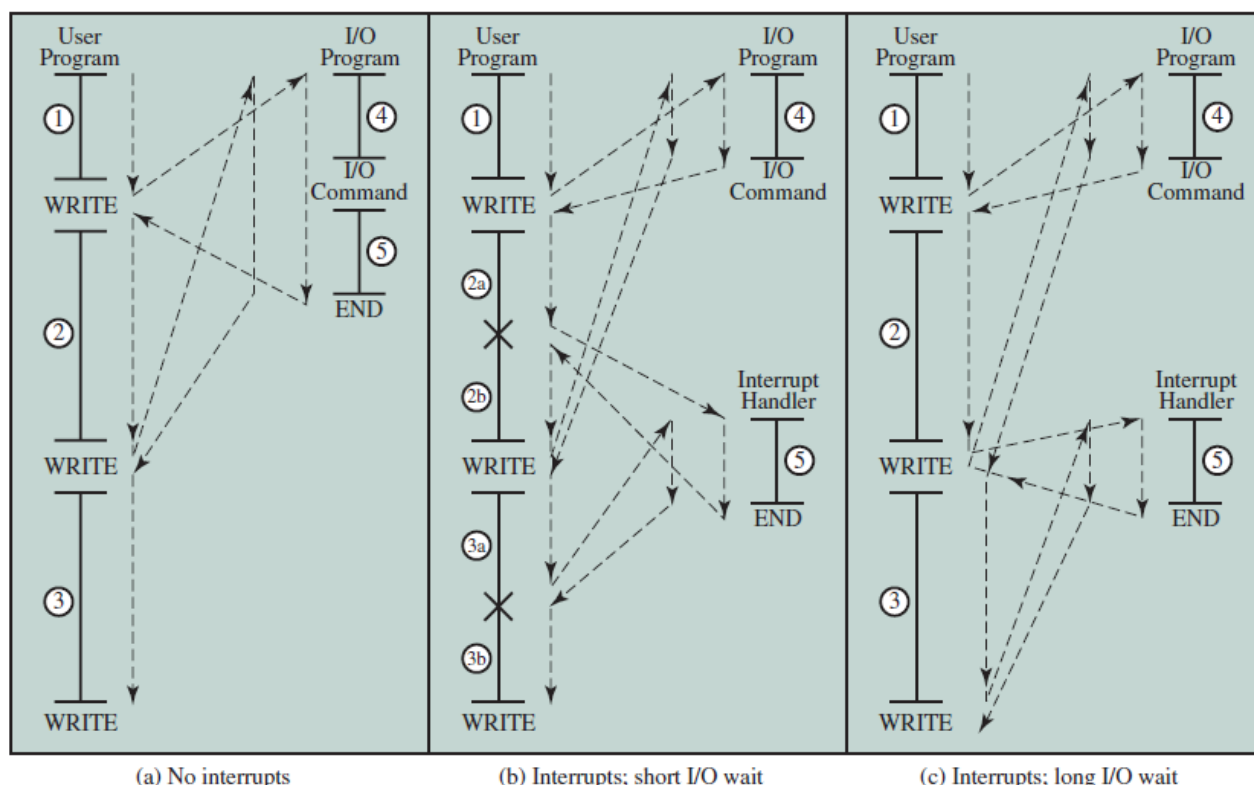
Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 3.3. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor.

Figure 3.7a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

- A sequence of instructions, labelled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labelled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Table 3.1 Classes of Interrupts

| | |
|-------------------------|--|
| Program | Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space. |
| Timer | Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis. |
| I/O | Generated by an I/O controller, to signal normal completion of an operation, request service from the processor, or to signal a variety of error conditions. |
| Hardware Failure | Generated by a failure such as power failure or memory parity error. |



✕ = interrupt occurs during course of execution of user program

Figure 3.7 Program Flow of Control without and with Interrupts

Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time.

Interrupts and the instruction cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 3.7b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced—that is, when it is ready to accept more data from the processor—the I/O module for that external device sends an *interrupt request* signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an **interrupt handler**, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk in Figure 3.7b.

Let us try to clarify what is happening in Figure 3.7. We have a user program that contains two WRITE commands. There is a segment of code at the beginning, then one WRITE command, then a second segment of code, then a second WRITE command, then a third and final segment of code. The WRITE command invokes the I/O program provided by the OS. Similarly, the I/O program consists of a segment of code, followed by an I/O command, followed by another segment of code. The I/O command invokes a hardware I/O operation.

USER PROGRAM

$\left. \begin{array}{l} \langle \text{statement} \rangle \\ \langle \text{statement} \rangle \\ \vdots \\ \langle \text{statement} \rangle \end{array} \right\}$ Code segment 1

WRITE

$\left. \begin{array}{l} \langle \text{statement} \rangle \\ \langle \text{statement} \rangle \\ \vdots \\ \langle \text{statement} \rangle \end{array} \right\}$ Code segment 2

WRITE

$\left. \begin{array}{l} \langle \text{statement} \rangle \\ \langle \text{statement} \rangle \\ \vdots \\ \langle \text{statement} \rangle \end{array} \right\}$ Code segment 3

I/O PROGRAM

$\left. \begin{array}{l} \langle \text{statement} \rangle \\ \langle \text{statement} \rangle \\ \vdots \\ \langle \text{statement} \rangle \end{array} \right\}$ Code segment 4

I/O command

$\left. \begin{array}{l} \langle \text{statement} \rangle \\ \langle \text{statement} \rangle \\ \vdots \\ \langle \text{statement} \rangle \end{array} \right\}$ Code segment 5

From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 3.8). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

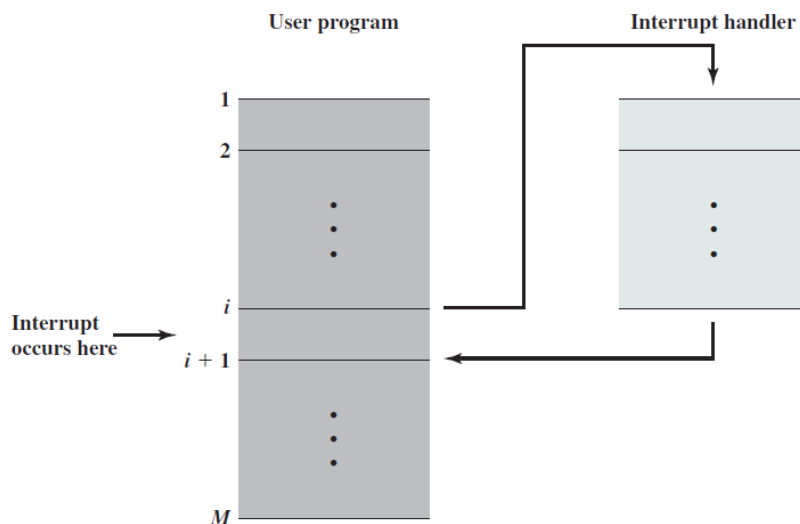


Figure 3.8 Transfer of Control via Interrupts

To accommodate interrupts, an *interrupt cycle* is added to the instruction cycle, as shown in Figure 3.9. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
- It sets the program counter to the starting address of an *interrupt handler* routine.

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt and to decide on the appropriate action. Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

To appreciate the gain in efficiency, consider Figure 3.10, which is a timing diagram based on the flow of control in Figures 3.7a and 3.7b. In this figure, user program code segments are shaded green, and I/O program code segments are shaded gray. Figure 3.10a shows the case in which interrupts are not used. The processor must wait while an I/O operation is performed.

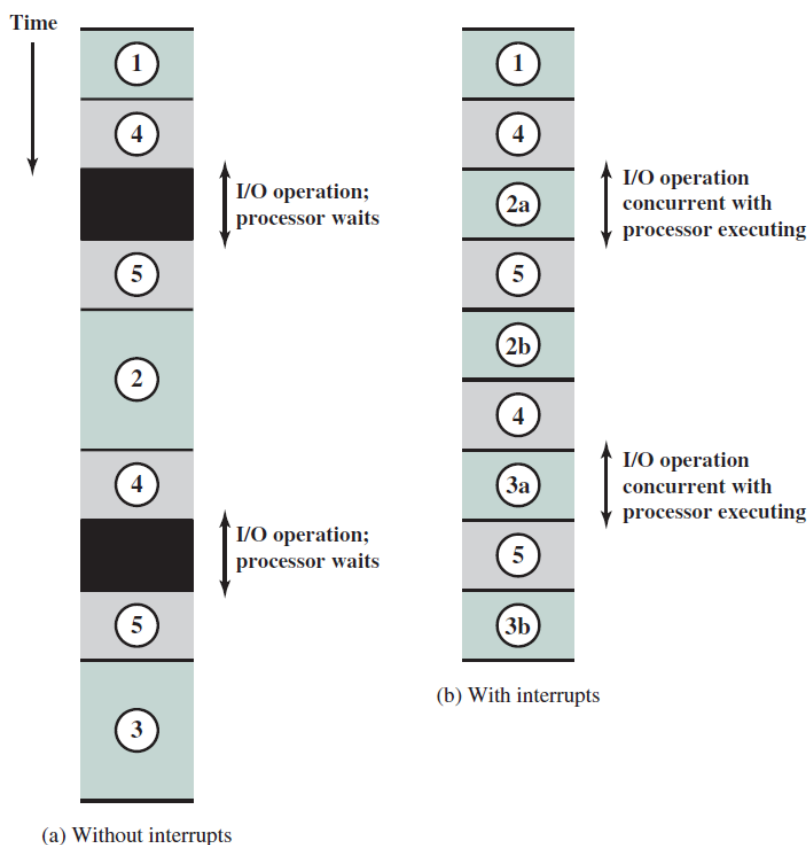


Figure 3.10 Program Timing: Short I/O Wait

Figures 3.7b and 3.10b assume that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program. In this case, the segment of code labelled code segment 2 is interrupted. A portion of the code (2a) executes (while the I/O operation is performed) and then the interrupt occurs (upon the completion of the I/O operation). After the interrupt is serviced, execution resumes with the remainder of code segment 2 (2b).

The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions. Figure 3.7c indicates this state of affairs. In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O

operation is completed, this new WRITE call may be processed, and a new I/O operation may be started. Figure 3.11 shows the timing for this situation with and without the use of interrupts. We can see that there is still a gain in efficiency because part of the time during which the I/O operation is under way overlaps with the execution of user instructions.

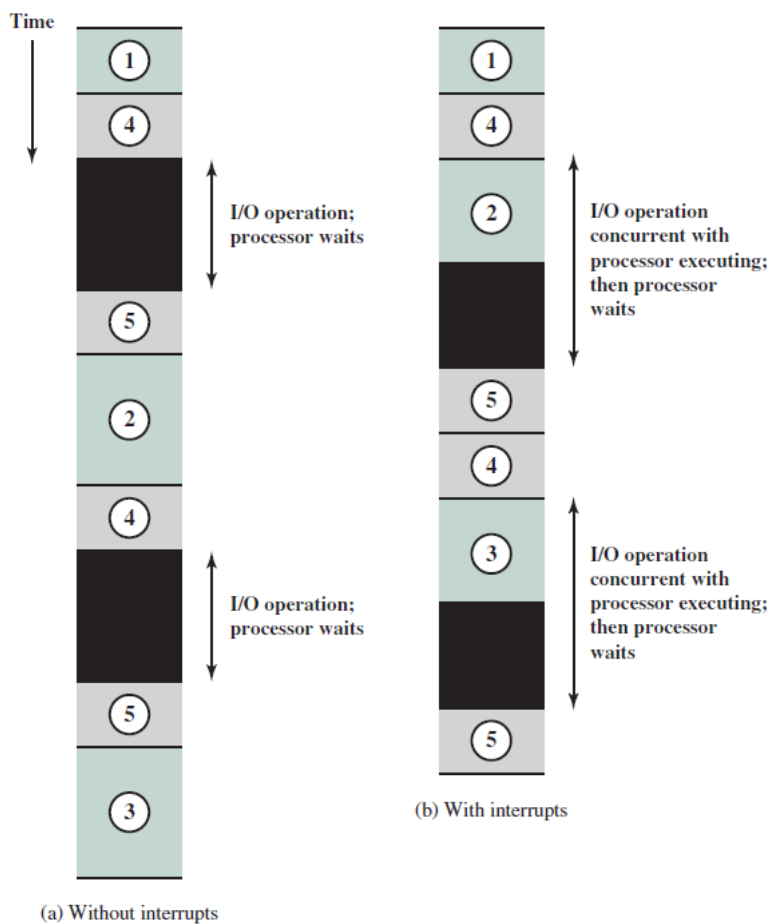


Figure 3.11 Program Timing: Long I/O Wait

Figure 3.12 shows a revised instruction cycle state diagram that includes interrupt cycle processing.

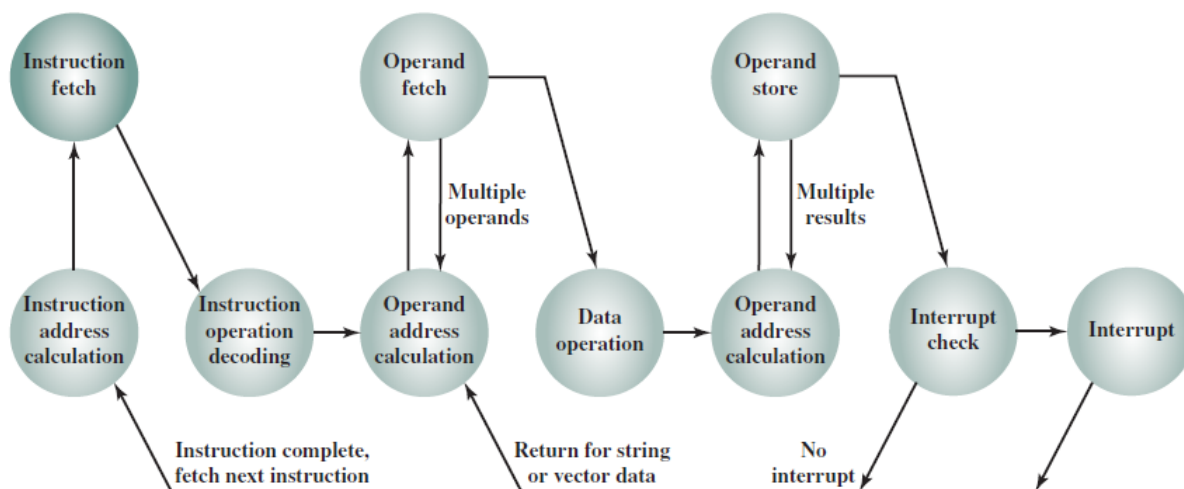
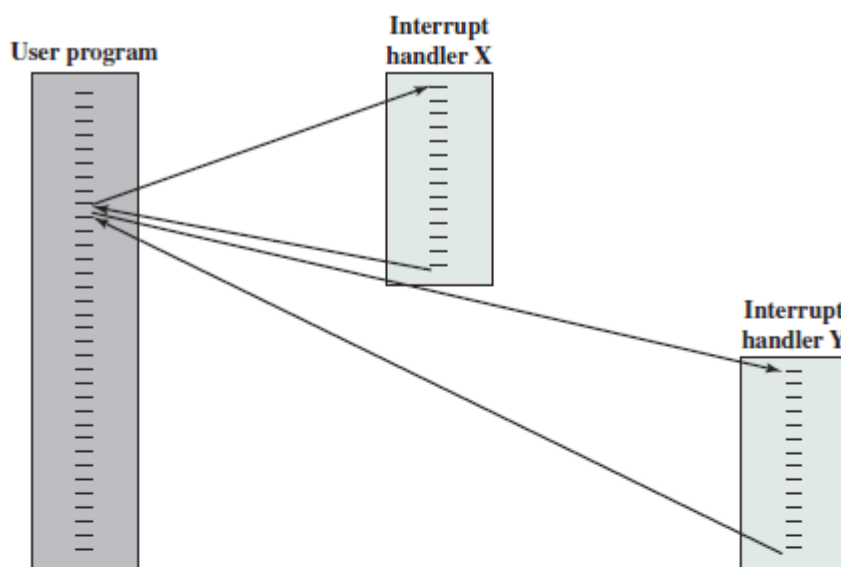


Figure 3.12 Instruction Cycle State Diagram, with Interrupts

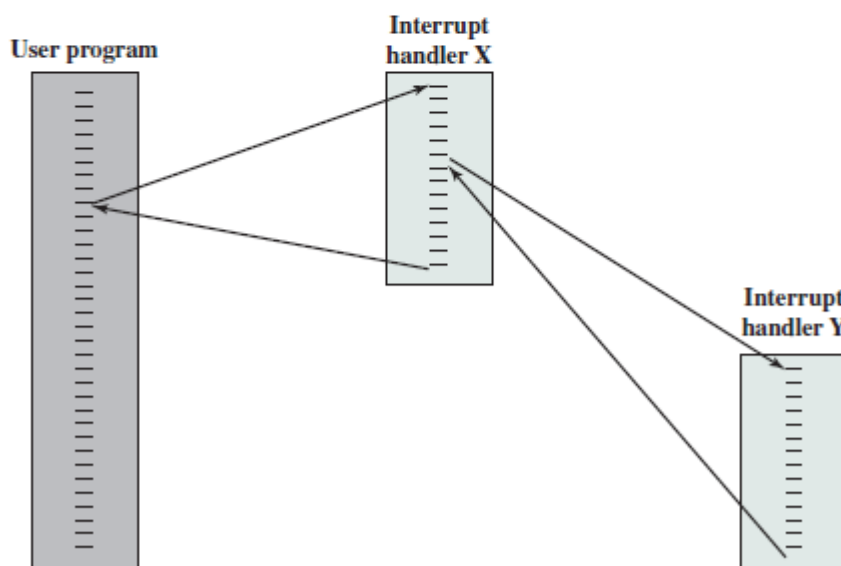
Multiple interrupts

The discussion so far has focused only on the occurrence of a single interrupt. Suppose, however, that multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every time it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives. The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A **disabled interrupt** simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is nice and simple, as interrupts are handled in strict sequential order (Figure 3.13a).



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 3.13 Transfer of Control with Multiple Interrupts

The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs. For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted (Figure 3.13b). As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. Figure 3.14 illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the system stack and execution continues at the printer **interrupt service routine (ISR)**. While this routine is still executing, at $t = 15$, a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs ($t = 20$). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion. When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and control transfers to the disk ISR. Only when that routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

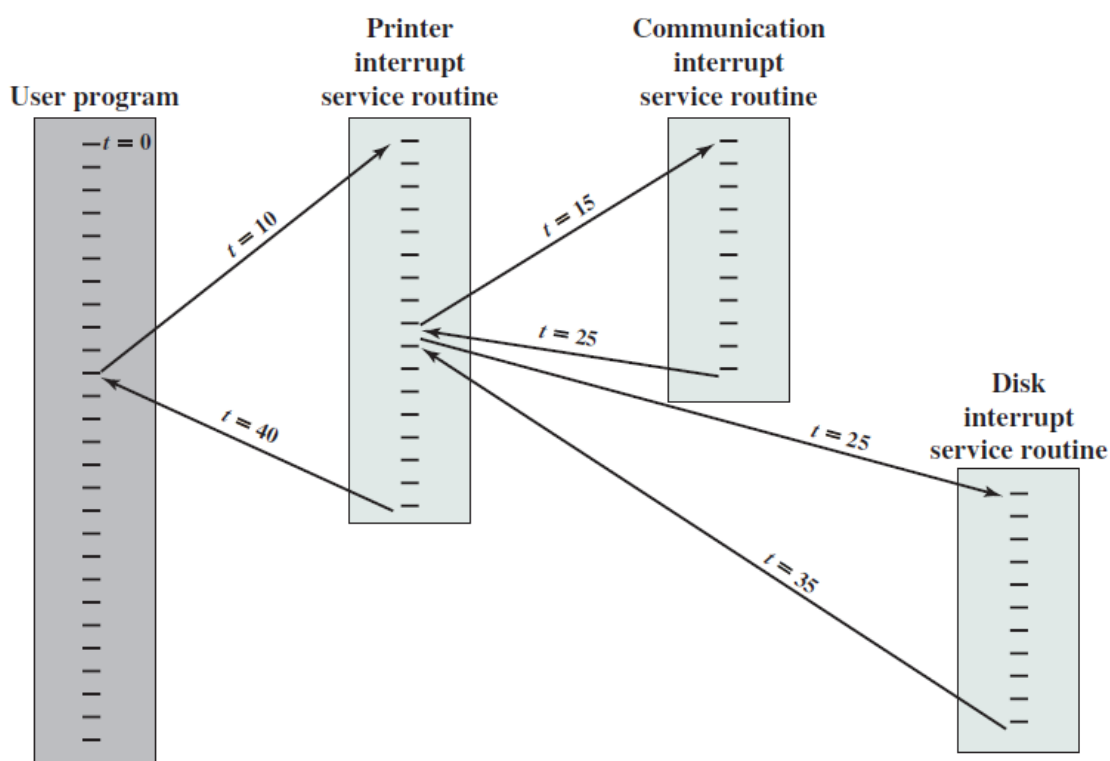


Figure 3.14 Example Time Sequence of Multiple Interrupts

When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and control transfers to the disk ISR. Only when that routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

I/O Function

Thus far, we have discussed the operation of the computer as controlled by the processor, and we have looked primarily at the interaction of processor and memory. The discussion has only alluded to the role of the I/O component. This role is discussed in detail in Chapter 7, but a brief summary is in order here.

An I/O module (e.g., a disk controller) can exchange data directly with the processor. Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read data from or write data to an I/O module. In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that of Figure 3.5 could occur, with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as direct memory access (DMA) and is examined in Chapter 7.

• Operating System role in Managing Hardware (T2: 1.1, 1.4, 1.5, R2: 1.7)

What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the *hardware*/the *operating system*, the *application programs* and the *users* (Figure 1.1).

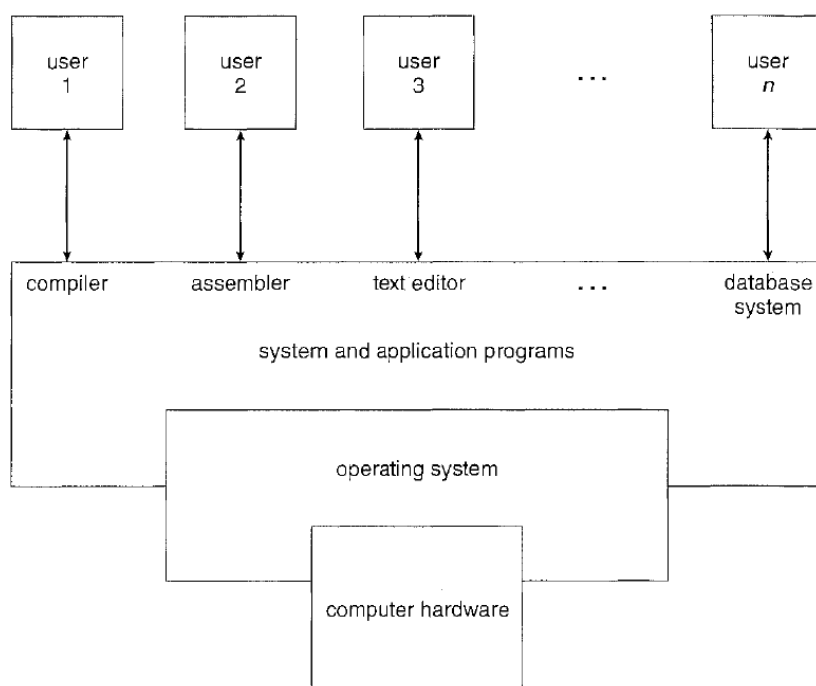


Figure 1.1 Abstract view of the components of a computer system.

The hardware – the central processing unit (CPU), the memory, and the input/output (I/O) devices – provides the basic computing resources for the system. The application programs – such as word processors/ spreadsheets/compilers, and Web browsers-define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware/ software and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a *government*. Like a government, it performs no useful function by

itself. It simply provides an *environment* within which other programs can do useful work.

To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case/ the operating system is designed mostly for ease of use with some attention paid to performance and none paid to resource utilization – how various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

In other cases, a user sits at a terminal connected to a mainframe or a minicomputer. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization - to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers-file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of handheld computers have come into fashion. Most of these devices are standalone units for individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems and networking. Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per unit of battery life is important as well.

Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocation. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

Operating System Structure

Now that we have discussed basic information about computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs(code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 1.9). Since, in general main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

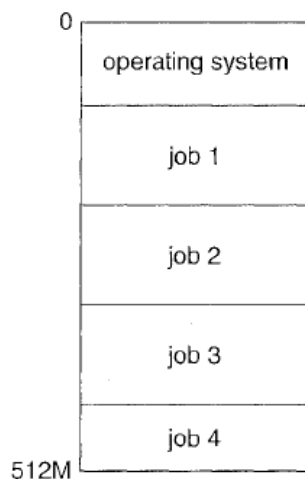


Figure 1.9 Memory layout for a multiprogramming system.

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multi-programmed system, the CPU would sit idle. In a multi-programmed system, the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU is switched to another job, and so on. Eventually the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multi-programmed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. Time sharing (or multitasking) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an interactive (or hands-on) computer system which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the response time should be short – typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user.

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is job scheduling which is discussed in Chapter 5. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in Chapters 8 and 9. In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is CPU scheduling, which is discussed in Chapter 5. Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. These considerations are discussed throughout the text.

In a time-sharing system, the operating system must ensure reasonable response time, which is sometimes accomplished through swapping where processes are swapped in and out of main memory to the disk. A more common method for achieving this goal is virtual memory, a technique that allows the execution of a process that is not completely in memory (Chapter 9). The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual physical memory. Further, it abstracts main memory into a large, uniform array of storage, separating logical as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Time-sharing systems must also provide a file system (Chapters 10 and 11). The file system resides on a collection of disks; hence, disk management must be provided (Chapter 12). Also, time-sharing systems provide a mechanism for protecting resources from inappropriate use (Chapter 14). To ensure orderly execution, the system must provide mechanisms for job synchronization and communication (Chapter 6), and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another (Chapter 7).

Operating System Operations

As mentioned earlier, modern operating systems are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A trap (or an exception) is a software generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multi-programming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

Dual-Mode Operation

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.

At the very least we need two separate modes of operation : user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.10. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well.

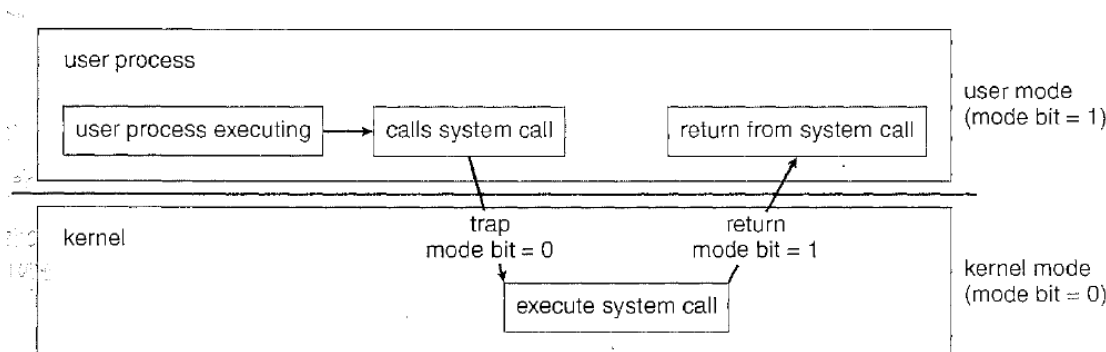


Figure 1.10 Transition from user to kernel mode.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users - and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control timer management and interrupt management. As we shall see throughout the text, there are many additional privileged instructions.

We can now see the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system. A system call usually takes the form of a trap to a specific location in the interrupt vector. This trap can be executed by a generic trap instruction, although some systems (such as the MIPS R2000 family) have a specific syscall instruction.

When a system call is executed, it is treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations passed in registers). The kernel verifies that the parameters are correct and legal executes the request, and returns control to the instruction following the system call. We describe system calls more fully in Section 2.3.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time, with potentially disastrous results. Recent versions of the Intel CPU do provide dual-mode operation. Accordingly, most contemporary operating systems such as Microsoft Vista and Windows XP, as well as Unix, Linux, and Solaris take advantage of this dual-mode feature and provide greater protection for the operating system.

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system. If a user program fails in some way-such as by making an attempt either to execute an illegal instruction or to access memory that is not in the user's address space-then the hardware traps to the operating system. The trap transfers control through the interrupt vector to the operating system, just as an interrupt does. When a program error occurs, the operating system must terminate the program abnormally. This situation is handled by the same code as a user-requested

abnormal termination. An appropriate error message is given, and the memory of the program may be dumped. The memory dump is usually written to a file so that the user or programmer can examine it and perhaps correct it and restart the program.

Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

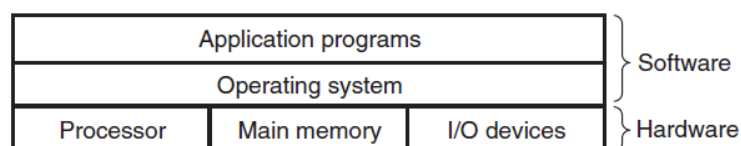
Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

Thus, we can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts and the counter is decremented by 1. As long as the counter is positive, control is returned to the user program. When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.

The Operating System Manages the Hardware

Back to our hello example. When the shell loaded and ran the hello program, and when the hello program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the *operating system*. We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10. All attempts by an application program to manipulate the hardware must go through the operating system.

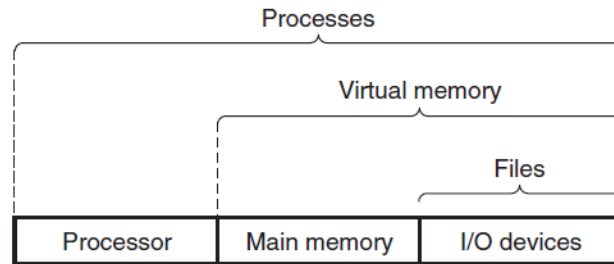
Figure 1.10
Layered view of a
computer system.



The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications, and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices. The operating system achieves both goals via the fundamental abstractions shown in Figure 1.11: *processes*, *virtual memory*, and *files*. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main

memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices. We will discuss each in turn.

Figure 1.11
Abstractions provided by
an operating system.



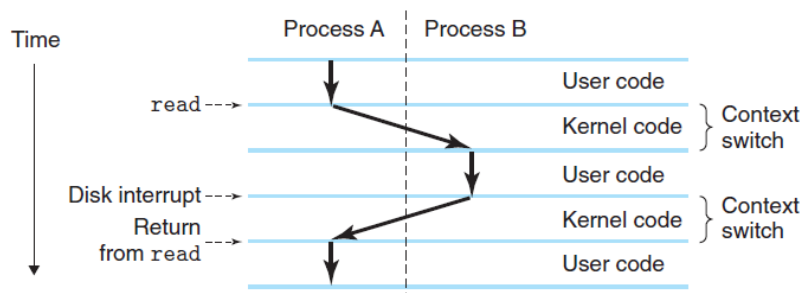
- **Processes**

When a program such as `hello` runs on a modern system, the operating system provides the illusion that the program is the only one running on the system. The program appears to have exclusive use of both the processor, main memory, and I/O devices. The processor appears to execute the instructions in the program, one after the other, without interruption. And the code and data of the program appear to be the only objects in the system’s memory. These illusions are provided by the notion of a process, one of the most important and successful ideas in computer science.

A *process* is the operating system’s abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware. By *concurrently*, we mean that the instructions of one process are interleaved with the instructions of another process. In most systems, there are more processes to run than there are CPUs to run them. Traditional systems could only execute one program at a time, while newer *multicore* processors can execute several programs simultaneously. In either case, a single CPU can appear to execute multiple processes concurrently by having the processor switch among them. The operating system performs this interleaving with a mechanism known as *context switching*. To simplify the rest of this discussion, we consider only a *uniprocessor system* containing a single CPU. We will return to the discussion of *multiprocessor* systems in Section 1.9.1.

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the *context*, includes information such as the current values of the PC, the register file, and the contents of main memory. At any point in time, a uniprocessor system can only execute the code for a single process. When the operating system decides to transfer control from the current process to some new process, it performs a *context switch* by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example `hello` scenario.

Figure 1.12
Process context
switching.



There are two concurrent processes in our example scenario: the shell process and the hello process. Initially, the shell process is running alone, waiting for input on the command line. When we ask it to run the hello program, the shell carries out our request by invoking a special function known as a *system call* that passes control to the operating system. The operating system saves the shell's context, creates a new hello process and its context, and then passes control to the new hello process. After hello terminates, the operating system restores the context of the shell process and passes control back to it, where it waits for the next command line input.

Implementing the process abstraction requires close cooperation between both the low-level hardware and the operating system software. We will explore how this works, and how applications can create and control their own processes, in Chapter 8.

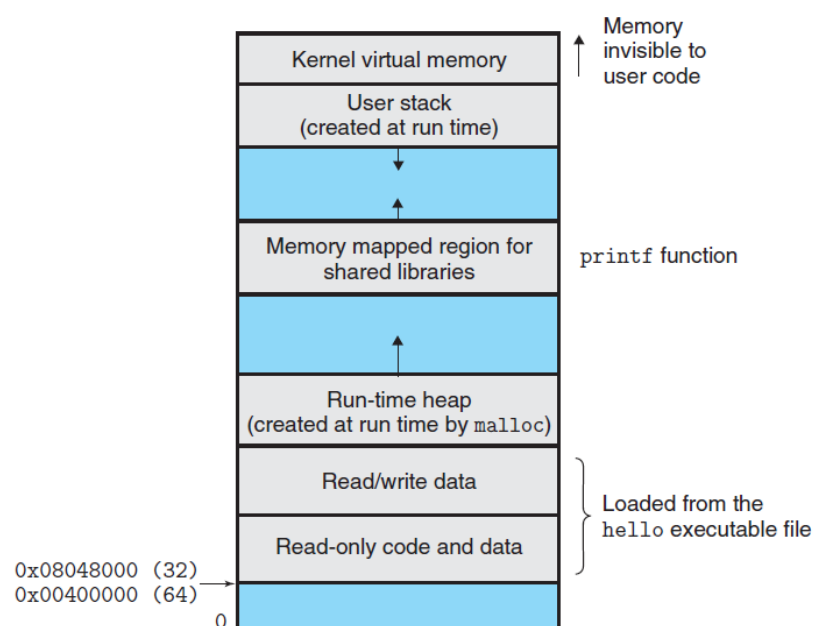
• Threads

Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called *threads*, each running in the context of the process and sharing the same code and global data. Threads are an increasingly important programming model because of the requirement for concurrency in network servers, because it is easier to share data between multiple threads than between multiple processes, and because threads are typically more efficient than processes. Multi-threading is also one way to make programs run faster when multiple processors are available, as we will discuss in Section 1.9.1. You will learn the basic concepts of concurrency, including how to write threaded programs, in Chapter 12.

• Virtual Memory

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has the same uniform view of memory, which is known as its *virtual address space*. The virtual address space for Linux processes is shown in Figure 1.13. (Other Unix systems use a similar layout.) In Linux, the topmost region of the address space is reserved for code and data in the operating system that is common to all processes. The lower region of the address space holds the code and data defined by the user's process. Note that addresses in the figure increase from the bottom to the top.

Figure 1.13
Process virtual address space.



The virtual address space seen by each process consists of a number of well defined areas, each with a specific purpose. You will learn more about these areas later in the book, but it will be helpful to look briefly at each, starting with the lowest addresses and working our way up:

- a. *Program code and data.* Code begins at the same fixed address for all processes, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file, in our case the hello executable. You will learn more about this part of the address space when we study linking and loading in Chapter 7.
- b. *Heap.* The code and data areas are followed immediately by the run-time *heap*. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at run time as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in Chapter 9.
- c. *Shared libraries.* Near the middle of the address space is an area that holds the code and data for *shared libraries* such as the C standard library and the math library. The notion of a shared library is a powerful but somewhat difficult concept. You will learn how they work when we study dynamic linking in Chapter 7.
- d. *Stack.* At the top of the user's virtual address space is the *user stack* that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. You will learn how the compiler uses the stack in Chapter 3.
- e. *Kernel virtual memory.* The *kernel* is the part of the operating system that is always resident in memory. The top region of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code.

For virtual memory to work, a sophisticated interaction is required between the hardware and the operating system software, including a hardware translation of every address generated by the processor. The basic idea is to store the contents of a process's virtual memory on disk, and then use the main memory as a cache for the disk. Chapter 9 explains how this works and why it is so important to the operation of modern systems.

- **Files**

A *file* is a sequence of bytes, nothing more and nothing less. Every I/O device, including disks, keyboards, displays, and even networks, is modelled as a file. All input and output in the system is performed by reading and writing files, using a small set of system calls known as *Unix I/O*.

This simple and elegant notion of a file is nonetheless very powerful because it provides applications with a uniform view of all of the varied I/O devices that might be contained in the system. For example, application programmers who manipulate the contents of a disk file are blissfully unaware of the specific disk technology. Further, the same program will run on different systems that use different disk technologies. You will learn about Unix I/O in Chapter 10.

Introduction to Computer Systems (Contd.)

•Performance Assessment(R1: 1.4, T1: 2.3)

Defining Performance

When we say one computer has better performance than another, what do we mean? If you were running a program on two different desktop computers, you'd say that the faster one is the desktop computer that gets the job done first. If you were running a datacenter that had several servers running jobs submitted by many users, you'd say that the faster computer was the one that completed the most jobs during a day. As an individual computer user, you are interested in reducing **response time**—the time between the start and completion of a task—also referred to as **execution time**. Datacenter managers are often interested in increasing **throughput** or **bandwidth**—the total amount of work done in a given time. Hence, in most cases, we will need different performance metrics as well as different sets of applications to benchmark personal mobile devices, which are more focused on response time, versus servers, which are more focused on throughput.

In discussing the performance of computers, we will be primarily concerned with response time for the first few chapters. To maximize performance, we want to minimize response time or execution time for some task. Thus, we can relate performance and execution time for a computer X:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

This means that for two computers X and Y, if the performance of X is greater than the performance of Y, we have

$$\begin{aligned}\text{Performance}_X &> \text{Performance}_Y \\ \frac{1}{\text{Execution time}_X} &> \frac{1}{\text{Execution time}_Y} \\ \text{Execution time}_Y &> \text{Execution time}_X\end{aligned}$$

That is, the execution time on Y is longer than that on X, if X is faster than Y.

In discussing a computer design, we often want to relate the performance of two different computers quantitatively. We will use the phrase “X is n times faster than Y”—or equivalently “X is n times as fast as Y”—to mean

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

If X is n times as fast as Y, then the execution time on Y is n times as long as it is on X:

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

For simplicity, we will normally use the terminology *as fast as* when we try to compare computers quantitatively. Because performance and execution time are reciprocals, increasing performance requires decreasing execution time. To avoid the potential confusion between the terms *increasing* and *decreasing*,

we usually say “improve performance” or “improve execution time” when we mean “increase performance” and “decrease execution time.”

Measuring Performance

Time is the measure of computer performance: the computer that performs the same amount of work in the least time is the fastest. Program *execution time* is measured in seconds per program. However, time can be defined in different ways, depending on what we count. The most straightforward definition of time is called *wall clock time*, *response time*, or *elapsed time*. These terms mean the total time to complete a task, including disk accesses, memory accesses, *input/output* (I/O) activities, operating system overhead—everything.

Computers are often shared, however, and a processor may work on several programs simultaneously. In such cases, the system may try to optimize throughput rather than attempt to minimize the elapsed time for one program. Hence, we often want to distinguish between the elapsed time and the time over which the processor is working on our behalf. **CPU execution time** or simply **CPU time**, which recognizes this distinction, is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**. Differentiating between system and user CPU time is difficult to do accurately, because it is often hard to assign responsibility for operating system activities to one user program rather than another and because of the functionality differences among operating systems.

For consistency, we maintain a distinction between performance based on elapsed time and that based on CPU execution time. We will use the term *system performance* to refer to elapsed time on an unloaded system and *CPU performance* to refer to user CPU time. We will focus on CPU performance in this chapter, although our discussions of how to summarize performance can be applied to either elapsed time or CPU time measurements.

Although as computer users we care about time, when we examine the details of a computer it's convenient to think about performance in other metrics. In particular, computer designers may want to think about a computer by using a measure that relates to how fast the hardware can perform basic functions. Almost all computers are constructed using a clock that determines when events take place in the hardware. These discrete time intervals are called **clock cycles** (or **ticks**, **clock ticks**, **clock periods**, **clocks**, **cycles**). Designers refer to the length of a **clock period** both as the time for a complete *clock cycle* (e.g., 250 picoseconds, or 250 ps) and as the *clock rate* (e.g., 4 gigahertz, or 4 GHz), which is the inverse of the clock period. In the next subsection, we will formalize the relationship between the clock cycles of the hardware designer and the seconds of the computer user.

CPU Performance and Its Factors

Users and designers often examine performance using different metrics. If we could relate these different metrics, we could determine the effect of a design change on the performance as experienced by the user. Since we are confining ourselves to CPU performance at this point, the bottom-line performance measure is CPU execution time. A simple formula relates the most basic metrics (clock cycles and clock cycle time) to CPU time:

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\text{CPU execution time for a program} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

This formula makes it clear that the hardware designer can improve performance by reducing the number of clock cycles required for a program or the length of the clock cycle. As we will see in later chapters, the designer often faces a trade-off between the number of clock cycles needed for a program and the length of each cycle. Many techniques that decrease the number of clock cycles may also increase the clock cycle time.

Instruction Performance

The performance equations above did not include any reference to the number of instructions needed for the program. However, since the compiler clearly generated instructions to execute, and the computer had to execute the instructions to run the program, the execution time must depend on the number of instructions in a program. One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction. Therefore, the number of clock cycles required for a program can be written as :

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \frac{\text{Average clock cycles}}{\text{per instruction}}$$

The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**. Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program. CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.

The Classic CPU Performance Equation

We can now write this basic performance equation in terms of **instruction count** (the number of instructions executed by the program), CPI, and clock cycle time:

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

or, since the clock rate is the inverse of clock cycle time:

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

These formulas are particularly useful because they separate the three key factors that affect performance. We can use these formulas to compare two different implementations or to evaluate a design alternative if we know its impact on these three parameters.

The BIG Picture

Figure 1.15 shows the basic measurements at different levels in the computer and what is being measured in each case. We can see how these factors are combined to yield execution time measured in seconds per program:

$$\text{Time} = \text{Seconds/Program} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Always bear in mind that the only complete and reliable measure of computer performance is time. For example, changing the instruction set to lower the instruction count may lead to an organization with a slower clock cycle time or higher CPI that offsets the improvement in instruction count. Similarly, because CPI depends on type of instructions executed, the code that executes the fewest number of instructions may not be the fastest.

| Components of performance | Units of measure |
|------------------------------------|--|
| CPU execution time for a program | Seconds for the program |
| Instruction count | Instructions executed for the program |
| Clock cycles per instruction (CPI) | Average number of clock cycles per instruction |
| Clock cycle time | Seconds per clock cycle |

FIGURE 1.15 The basic components of performance and how each is measured.

- MIPS Rate**

A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS), referred to as the **MIPS rate**. We can express the MIPS rate in terms of the clock rate and *CPI* as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} \quad (2.3)$$

EXAMPLE 2.2 Consider the execution of a program that results in the execution of 2 million instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the *CPI* for each instruction type are given below, based on the result of a program trace experiment:

| Instruction Type | <i>CPI</i> | Instruction Mix (%) |
|----------------------------------|------------|---------------------|
| Arithmetic and logic | 1 | 60 |
| Load/store with cache hit | 2 | 18 |
| Branch | 4 | 12 |
| Memory reference with cache miss | 8 | 10 |

The average *CPI* when the program is executed on a uniprocessor with the above trace results is $CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24$. The corresponding MIPS rate is $(400 \times 10^6) / (2.24 \times 10^6) \approx 178$.

- **Amdahl's Law**

Computer system designers look for ways to improve system performance by advances in technology or change in design. Examples include the use of parallel processors, the use of a memory cache hierarchy, and speedup in memory access time and I/O transfer rate due to technology improvements. In all of these cases, it is important to note that a speedup in one aspect of the technology or design does not result in a corresponding improvement in performance. This limitation is succinctly expressed by Amdahl's law.

Amdahl's law was first proposed by Gene Amdahl in 1967 ([AMDA67], [AMDA13]) and deals with the potential speedup of a program using multiple processors compared to a single processor. Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently sequential, and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead. Let T be the total execution time of the program using a single processor. Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\begin{aligned} \text{Speedup} &= \frac{\text{Time to execute program on a single processor}}{\text{Time to execute program on } N \text{ parallel processors}} \\ &= \frac{T(1 - f) + Tf}{T(1 - f) + \frac{Tf}{N}} = \frac{1}{(1 - f) + \frac{f}{N}} \end{aligned}$$

This equation is illustrated in Figures 2.3 and 2.4. Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by $1/(1 - f)$, so that there are diminishing returns for using more processors.

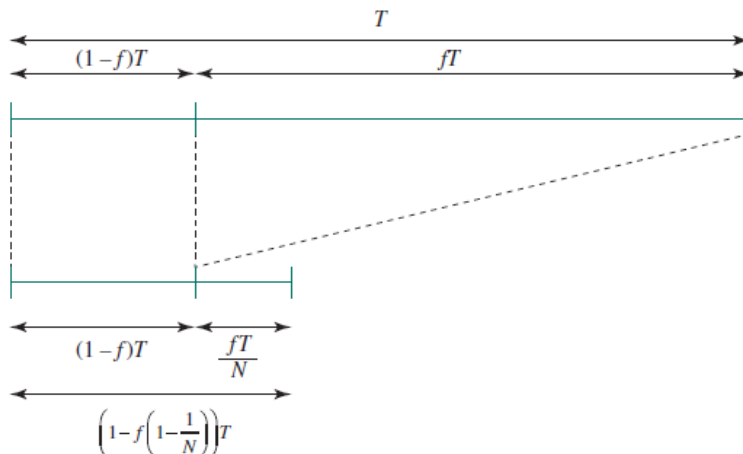


Figure 2.3 Illustration of Amdahl's Law

Nevertheless, Amdahl's law illustrates the problems facing industry in the development of multicore machines with an ever-growing number of cores: The software that runs on such machines must be adapted to a highly parallel execution environment to exploit the power of parallel processing.

Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system. Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as :

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}} \quad (2.1)$$

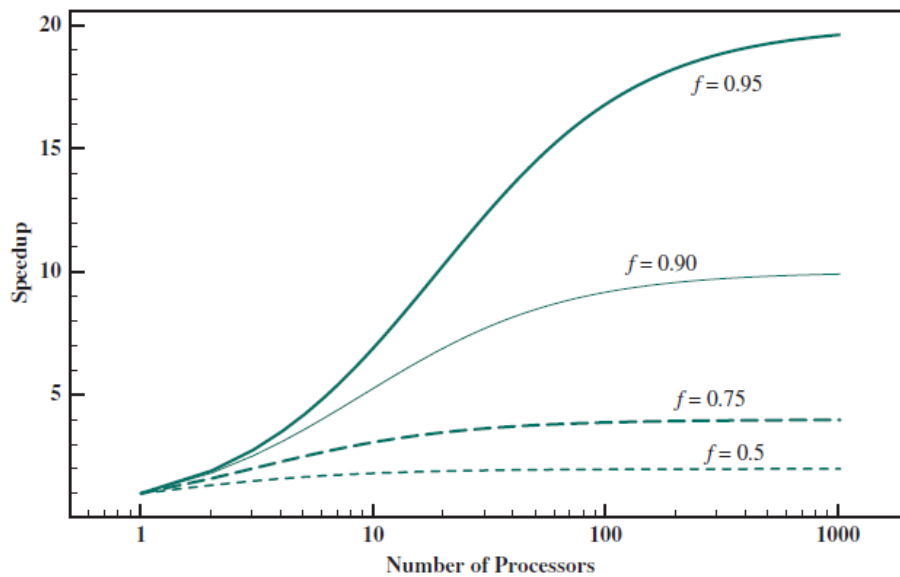


Figure 2.4 Amdahl's Law for Multiprocessors

Suppose that a feature of the system is used during execution a fraction of the time f , before enhancement, and that the speedup of that feature after enhancement is SU_f . Then the overall speedup of the system is :

$$\text{Speedup} = \frac{1}{(1 - f) + \frac{f}{SU_f}}$$

Memory Organization

• Storage Technologies(T1: 5.1, R2: 6.1)

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as *cores*. Hence, main memory was often referred to as *core*, a term that persists to this day. The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. Key aspects of this technology are explored in this section.

Organization

The basic element of a **semiconductor memory** is the memory cell. Although a variety of electronic technologies are used, all semiconductor memory cells share certain properties:

- They exhibit two stable (or semistable) states, which can be used to represent binary 1 and 0.
- They are capable of being written into (at least once), to set the state.
- They are capable of being read to sense the state.

Figure 5.1 depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state. The details of the internal organization, functioning, and timing of the memory cell depend on the specific integrated circuit technology used and are beyond the scope of this book, except for a brief summary. For our purposes, we will take it as given that individual cells can be selected for reading and writing operations.

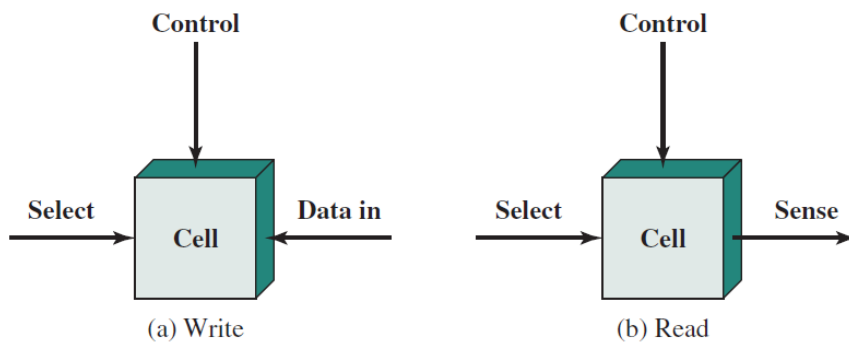


Figure 5.1 Memory Cell Operation

Random Access Memory

Random-access memory (RAM) comes in two varieties—*static* and *dynamic*. *Static RAM* (SRAM) is faster and significantly more expensive than *Dynamic RAM* (DRAM). SRAM is used for cache memories, both on and off the CPU chip. DRAM is used for the main memory plus the frame buffer of a graphics system. Typically, a desktop system will have no more than a few megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAM stores each bit in a *bistable* memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or *states*. Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable states.

Due to its bistable nature, an SRAM memory cell will retain its value indefinitely, as long as it is kept powered. Even when a disturbance, such as electrical noise, perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.

Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads, that is, 30×10^{-15} farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense—each cell consists of a capacitor and a single access transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

Various sources of leakage current cause a DRAM cell to lose its charge within a time period of around 10 to 100 milliseconds. Fortunately, for computers operating with clock cycle times measured in nanoseconds, this retention time is quite long. The memory system must periodically refresh every bit of memory by reading it out and then rewriting it. Some systems also use error-correcting codes, where the computer

words are encoded a few more bits (e.g., a 32-bit word might be encoded using 38 bits), such that circuitry can detect and correct any single erroneous bit within a word.

| | Transistors per bit | Relative access time | Persistent? | Sensitive? | Relative cost | Applications |
|------|------------------------|-------------------------|-------------|------------|------------------|-------------------------|
| SRAM | 6 | 1× | Yes | No | 100× | Cache memory |
| DRAM | 1 | 10× | No | Yes | 1× | Main mem, frame buffers |

Figure 6.2 Characteristics of DRAM and SRAM memory.

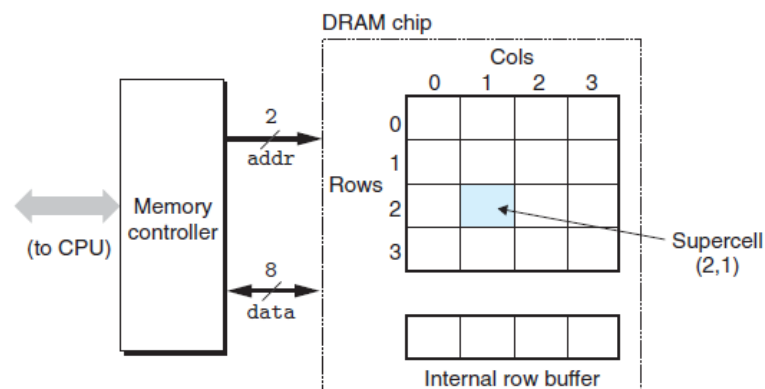
Figure 6.2 summarizes the characteristics of SRAM and DRAM memory. SRAM is persistent as long as power is applied. Unlike DRAM, no refresh is necessary. SRAM can be accessed faster than DRAM. SRAM is not sensitive to disturbances such as light and electrical noise. The trade-off is that SRAM cells use more transistors than DRAM cells, and thus have lower densities, are more expensive, and consume more power.

Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into d *supercells*, each consisting of w DRAM cells. A $d \times w$ DRAM stores a total of dw bits of information. The supercells are organized as a rectangular array with r rows and c columns, where $rc = d$. Each supercell has an address of the form (i, j) , where i denotes the row, and j denotes the column.

For example, Figure 6.3 shows the organization of a 16×8 DRAM chip with $d = 16$ supercells, $w = 8$ bits per supercell, $r = 4$ rows, and $c = 4$ columns. The shaded box denotes the supercell at address $(2, 1)$. Information flows in and out of the chip via external connectors called *pins*. Each pin carries a 1-bit signal. Figure 6.3 shows two of these sets of pins: eight data pins that can transfer 1 byte in or out of the chip, and two addr pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

Figure 6.3
High-level view of a 128-bit 16×8 DRAM chip.



Each DRAM chip is connected to some circuitry, known as the *memory controller*, that can transfer w bits at a time to and from each DRAM chip. To read the contents of supercell (i, j) , the memory controller sends the row address i to the DRAM, followed by the column address j . The DRAM responds by sending the contents of supercell (i, j) back to the controller. The row address i is called a *RAS (Row Access Strobe) request*. The column address j is called a *CAS (Column Access Strobe) request*. Notice that the RAS and CAS requests share the same DRAM address pins.

For example, to read supercell (2, 1) from the 16×8 DRAM in Figure 6.3, the memory controller sends row address 2, as shown in Figure 6.4(a). The DRAM responds by copying the entire contents of row 2 into an internal row buffer. Next, the memory controller sends column address 1, as shown in Figure 6.4(b). The DRAM responds by copying the 8 bits in supercell (2, 1) from the row buffer and sending them to the memory controller.

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to reduce the number of address pins on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address pins instead of two. The disadvantage of the two-dimensional array organization is that addresses must be sent in two distinct steps, which increases the access time.

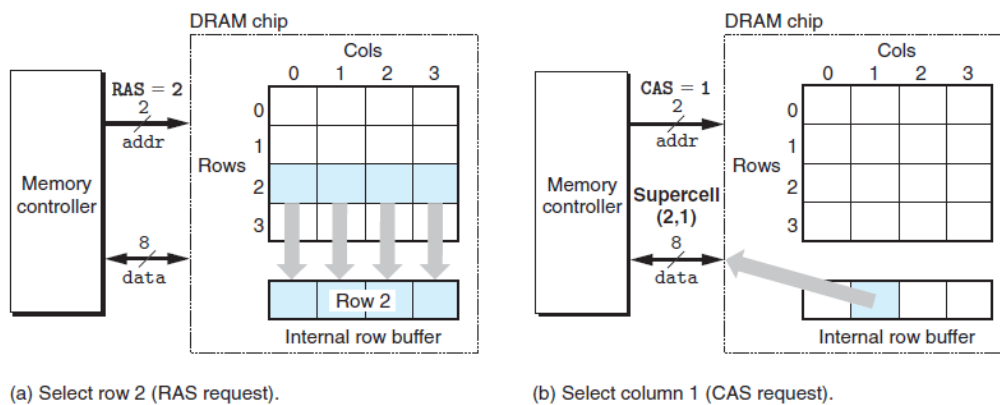


Figure 6.4 Reading the contents of a DRAM supercell.

Memory Modules

DRAM chips are packaged in *memory modules* that plug into expansion slots on the main system board (motherboard). Common packages include the 168-pin *dual inline memory module* (DIMM), which transfers data to and from the memory controller in 64-bit chunks, and the 72-pin *single inline memory module* (SIMM), which transfers data in 32-bit chunks.

Figure 6.5 shows the basic idea of a memory module. The example module stores a total of 64 MB (megabytes) using eight 64-Mbit $8M \times 8$ DRAM chips, numbered 0 to 7. Each supercell stores 1 byte of *main memory*, and each 64-bit doubleword¹ at byte address A in main memory is represented by the eight supercells whose corresponding supercell address is (i, j) . In the example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

To retrieve a 64-bit doubleword at memory address A , the memory controller converts A to a supercell address (i, j) and sends it to the memory module, which then broadcasts i and j to each DRAM. In response, each DRAM outputs the 8-bit contents of its (i, j) supercell. Circuitry in the module collects these outputs and forms them into a 64-bit doubleword, which it returns to the memory controller.

Main memory can be aggregated by connecting multiple memory modules to the memory controller. In this case, when the controller receives an address A , the controller selects the module k that contains A , converts A to its (i, j) form, and sends (i, j) to module k .

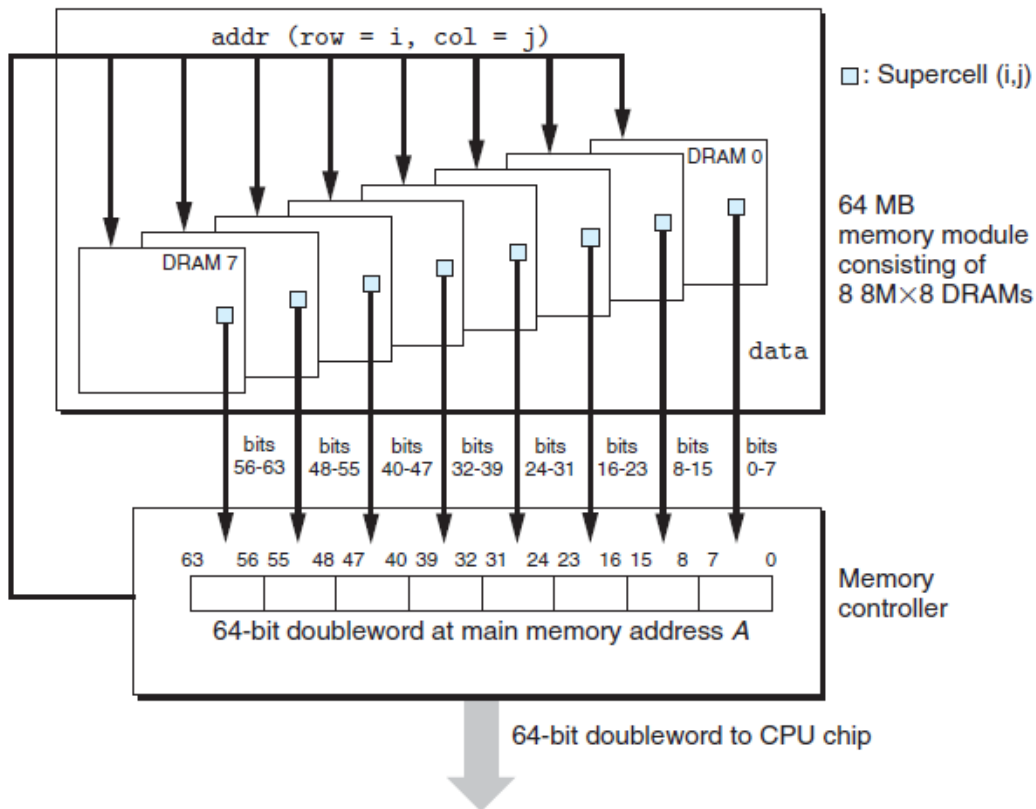


Figure 6.5 Reading the contents of a memory module.

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

- Fast page mode DRAM (FPM DRAM).** A conventional DRAM copies an entire row of supercells into its internal row buffer, uses one, and then discards the rest. FPM DRAM improves on this by allowing consecutive accesses to the same row to be served directly from the row buffer. For example, to read four supercells from row i of a conventional DRAM, the memory controller must send four RAS/CAS requests, even though the row address i is identical in each case. To read supercells from the same row of an FPM DRAM, the memory controller sends an initial RAS/CAS request, followed by three CAS requests. The initial RAS/CAS request copies row i into the row buffer and returns the supercell addressed by the CAS. The next three supercells are served directly from the row buffer, and thus more quickly than the initial supercell.

- Extended data out DRAM (EDODRAM).** An enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time.

- Synchronous DRAM (SDRAM).** Conventional, FPM, and EDO DRAMs are asynchronous in the sense that they communicate with the memory controller using a set of explicit control signals. SDRAM replaces many of these control signals with the rising edges of the same external clock signal that drives the memory controller. Without going into detail, the net effect is that an SDRAM can output the contents of its supercells at a faster rate than its asynchronous counterparts.

. *Double Data-Rate Synchronous DRAM (DDR SDRAM)*. DDR SDRAM is an enhancement of SDRAM that doubles the speed of the DRAM by using both clock edges as control signals. Different types of DDR SDRAMs are characterized by the size of a small prefetch buffer that increases the effective bandwidth: DDR (2 bits), DDR2 (4 bits), and DDR3 (8 bits).

. *Rambus DRAM (RDRAM)*. This is an alternative proprietary technology with a higher maximum bandwidth than DDR SDRAM.

. *Video RAM (VRAM)*. Used in the frame buffers of graphics systems. VRAM is similar in spirit to FPM DRAM. Two major differences are that (1) VRAM output is produced by shifting the entire contents of the internal buffer in sequence, and (2) VRAM allows concurrent reads and writes to the memory. Thus, the system can be painting the screen with the pixels in the frame buffer (reads) while concurrently writing new values for the next update (writes).

Nonvolatile Memory

DRAMs and SRAMs are *volatile* in the sense that they lose their information if the supply voltage is turned off. *Nonvolatile memories*, on the other hand, retain their information even when they are powered off. There are a variety of nonvolatile memories. For historical reasons, they are referred to collectively as *read-only memories* (ROMs), even though some types of ROMs can be written to as well as read. ROMs are distinguished by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them.

A *programmable ROM* (PROM) can be programmed exactly once. PROMs include a sort of fuse with each memory cell that can be blown once by zapping it with a high current.

An *erasable programmable ROM* (EPROM) has a transparent quartz window that permits light to reach the storage cells. The EPROM cells are cleared to zeros by shining ultraviolet light through the window. Programming an EPROM is done by using a special device to write ones into the EPROM. An EPROM can be erased and reprogrammed on the order of 1000 times.

An *electrically erasable PROM* (EEPROM) is akin to an EPROM, but does not require a physically separate programming device, and thus can be reprogrammed in-place on printed circuit cards. An EEPROM can be reprogrammed on the order of 10⁵ times before it wears out.

Flash memory is a type of nonvolatile memory, based on EEPROMs, that has become an important storage technology. Flash memories are everywhere, providing fast and durable nonvolatile storage for a slew of electronic devices, including digital cameras, cell phones, music players, PDAs, and laptop, desktop, and server computer systems. In Section 6.1.3, we will look in detail at a new form of flash-based disk drive, known as a *solid state disk* (SSD), that provides a faster, sturdier, and less power-hungry alternative to conventional rotating disks.

Programs stored in ROM devices are often referred to as *firmware*. When a computer system is powered up, it runs firmware stored in a ROM. Some systems provide a small set of primitive input and output functions in firmware, for example, a PC's BIOS (basic input/output system) routines. Complicated devices such as graphics cards and disk drive controllers also rely on firmware to translate I/O (input/output) requests from the CPU.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6
Example bus structure
that connects the CPU
and main memory.

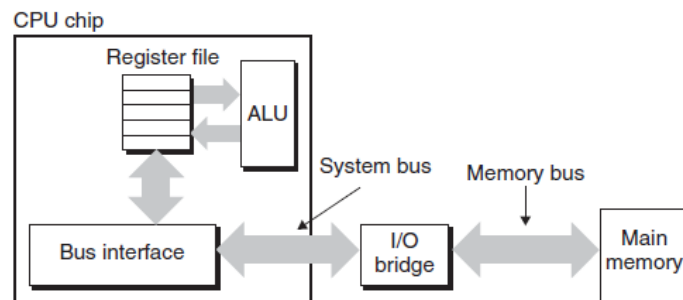


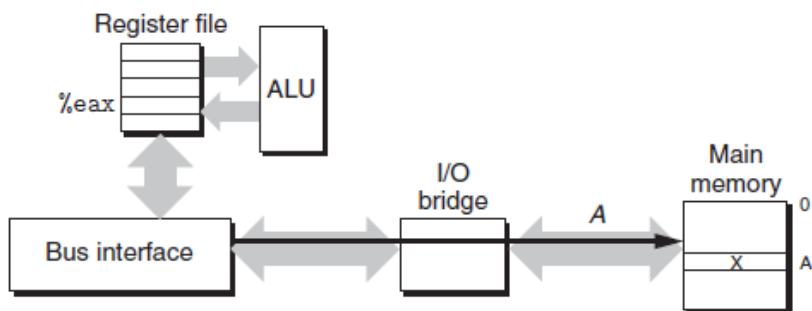
Figure 6.6 shows the configuration of an example computer system. The main components are the CPU chip, a chipset that we will call an *I/O bridge* (which includes the memory controller), and the DRAM memory modules that make up main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O bridge to the main memory.

The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

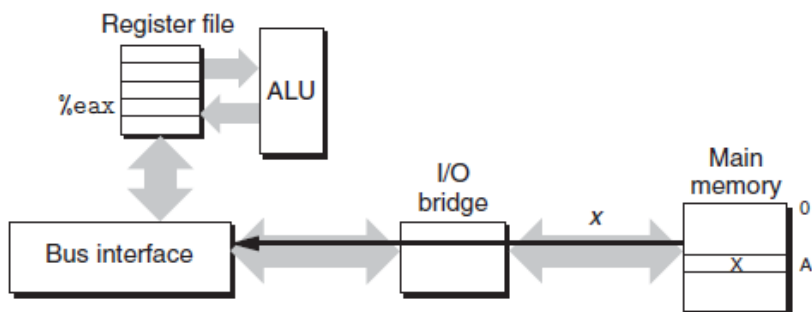
Consider what happens when the CPU performs a load operation such as

```
movl A,%eax
```

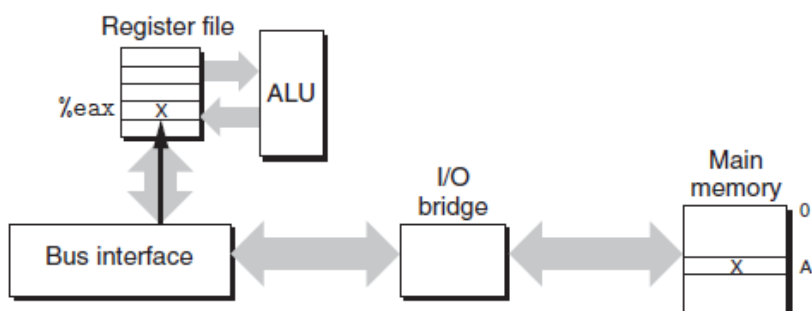
where the contents of address *A* are loaded into register *%eax*. Circuitry on the CPU chip called the *bus interface* initiates a read transaction on the bus. The read transaction consists of three steps. First, the CPU places the address *A* on the system bus. The I/O bridge passes the signal along to the memory bus (Figure 6.7(a)). Next, the main memory senses the address signal on the memory bus, reads the address from the memory bus, fetches the data word from the DRAM, and writes the data to the memory bus. The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads it from the bus, and copies it to register *%eax* (Figure 6.7(c)).



(a) CPU places address A on the memory bus.



(b) Main memory reads A from the bus, retrieves word x , and places it on the bus.



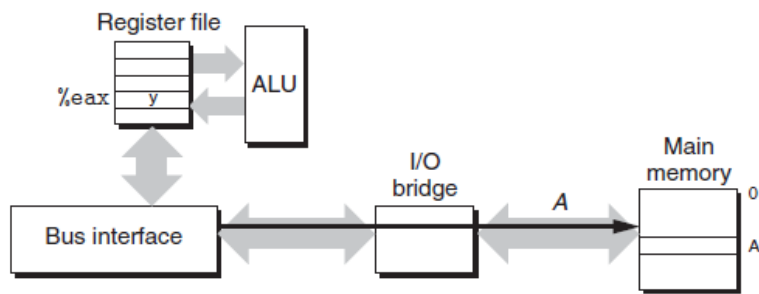
(c) CPU reads word x from the bus, and copies it into register $\%eax$.

Figure 6.7 Memory read transaction for a load operation: `movl A,%eax`.

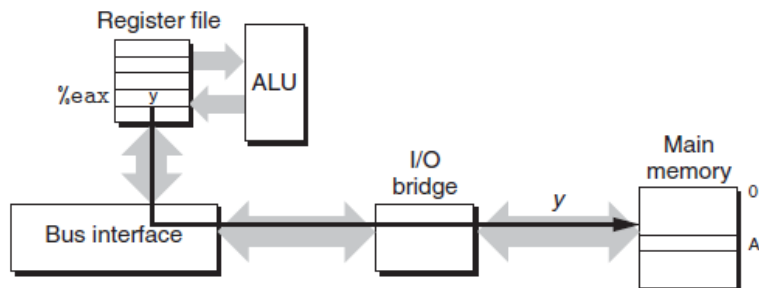
Conversely, when the CPU performs a store instruction such as

`movl %eax,A`

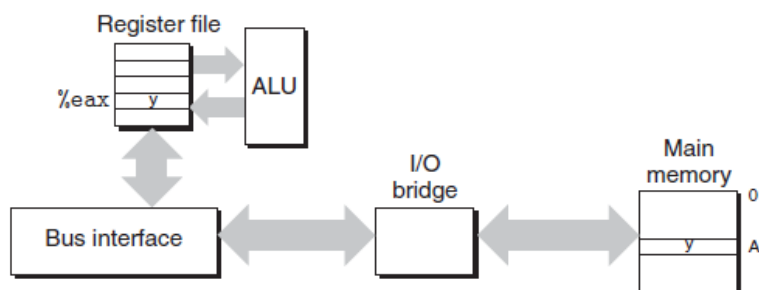
where the contents of register $\%eax$ are written to address A , the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data word in $\%eax$ to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).



(a) CPU places address `A` on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word `y` on the bus.



(c) Main memory reads data word `y` from the bus and stores it at address `A`.

Figure 6.8 Memory write transaction for a store operation: `movl %eax, A`.

Disk Storage

Disks are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory. However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.

Disk Geometry

Disks are constructed from *platters*. Each platter consists of two sides, or *surfaces*, that are coated with magnetic recording material. A rotating *spindle* in the center of the platter spins the platter at a fixed *rotational rate*, typically between 5400 and 15,000 *revolutions per minute* (RPM). A disk will typically contain one or more of these platters encased in a sealed container.

Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called *tracks*. Each track is partitioned into a collection of *sectors*. Each sector contains an equal number of data bits (typically 512 bytes) encoded in the magnetic material on the sector. Sectors are separated by *gaps* where no data bits are stored. Gaps store formatting bits that identify sectors.

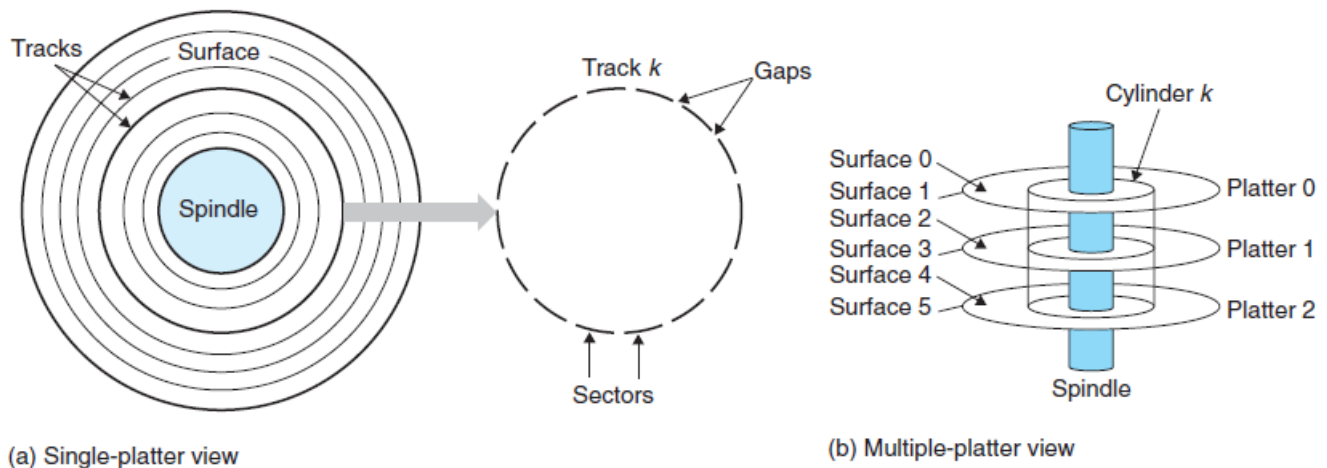


Figure 6.9 Disk geometry.

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a *disk drive*, although we will usually refer to it as simply a *disk*. We will sometime refer to disks as *rotating disks* to distinguish them from flash-based *solid state disks* (SSDs), which have no moving parts.

Disk manufacturers describe the geometry of multiple-platter drives in terms of *cylinders*, where a cylinder is the collection of tracks on all the surfaces that are equidistant from the center of the spindle. For example, if a drive has three platters and six surfaces, and the tracks on each surface are numbered consistently, then cylinder k is the collection of the six instances of track k .

Disk Capacity

The maximum number of bits that can be recorded by a disk is known as its *maximum capacity*, or simply *capacity*. Disk capacity is determined by the following technology factors:

- . *Recording density (bits/in)*: The number of bits that can be squeezed into a 1-inch segment of a track.
- . *Track density (tracks/in)*: The number of tracks that can be squeezed into a 1-inch segment of the radius extending from the center of the platter.
- . *Areal density (bits/in²)*: The product of the recording density and the track density.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every few years. The original disks, designed in an age of low areal density, partitioned every track into the same number of sectors, which was determined by the number of sectors that could be recorded on the innermost track. To maintain a fixed number of sectors per track, the sectors were spaced farther apart on the outer tracks. This was a reasonable approach when areal densities were relatively low. However, as areal densities increased, the gaps between sectors (where no data bits were stored) became unacceptably large. Thus, modern high-capacity disks use a technique known as *multiple zone recording*, where the set of cylinders is partitioned into disjoint subsets known as *recording zones*. Each zone consists of a contiguous collection of cylinders. Each track in each cylinder in a zone has the same number of sectors, which is determined by the number of sectors that can be packed into the innermost track of the zone. Note that diskettes (floppy disks) still use the old-fashioned approach, with a constant number of sectors per track.

The capacity of a disk is given by the following formula:

$$\text{Disk capacity} = \frac{\# \text{ bytes}}{\text{sector}} \times \frac{\text{average } \# \text{ sectors}}{\text{track}} \times \frac{\# \text{ tracks}}{\text{surface}} \times \frac{\# \text{ surfaces}}{\text{platter}} \times \frac{\# \text{ platters}}{\text{disk}}$$

For example, suppose we have a disk with 5 platters, 512 bytes per sector, 20,000 tracks per surface, and an average of 300 sectors per track. Then the capacity of the disk is:

$$\begin{aligned} \text{Disk capacity} &= \frac{512 \text{ bytes}}{\text{sector}} \times \frac{300 \text{ sectors}}{\text{track}} \times \frac{20,000 \text{ tracks}}{\text{surface}} \times \frac{2 \text{ surfaces}}{\text{platter}} \times \frac{5 \text{ platters}}{\text{disk}} \\ &= 30,720,000,000 \text{ bytes} \\ &= 30.72 \text{ GB.} \end{aligned}$$

Notice that manufacturers express disk capacity in units of gigabytes (GB), where $1 \text{ GB} = 10^9$ bytes.

Disk Operation

Disks read and write bits stored on the magnetic surface using a *read/write head* connected to the end of an *actuator arm*, as shown in Figure 6.10(a). By moving the arm back and forth along its radial axis, the drive can position the head over any track on the surface. This mechanical motion is known as a *seek*. Once the head is positioned over the desired track, then as each bit on the track passes underneath, the head can either sense the value of the bit (read the bit) or alter the value of the bit (write the bit). Disks with multiple platters have a separate read/write head for each surface, as shown in Figure 6.10(b). The heads are lined up vertically and move in unison. At any point in time, all heads are positioned on the same cylinder.

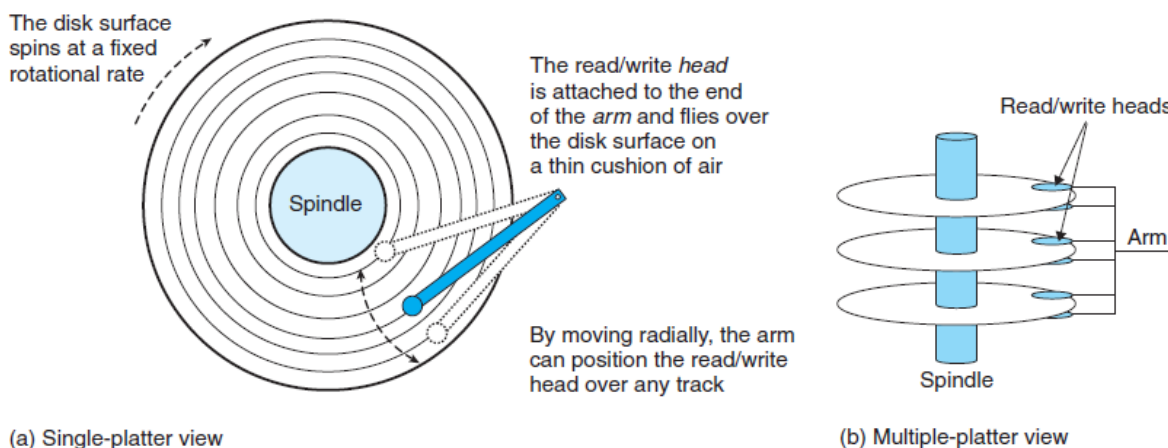


Figure 6.10 Disk dynamics.

The read/write head at the end of the arm flies (literally) on a thin cushion of air over the disk surface at a height of about 0.1 microns and a speed of about 80 km/h. This is analogous to placing the Sears Tower on its side and flying it around the world at a height of 2.5 cm (1 inch) above the ground, with each orbit of the earth taking only 8 seconds! At these tolerances, a tiny piece of dust on the surface is like a huge boulder. If the head were to strike one of these boulders, the head would cease flying and crash into the surface (a so-called *head crash*). For this reason, disks are always sealed in airtight packages. Disks read and write data in sector-sized blocks. The *access time* for a sector has three main components: *seek time*, *rotational latency*, and *transfer time*:

. **Seek time:** To read the contents of some target sector, the arm first positions the head over the track that contains the target sector. The time required to move the arm is called the *seek time*. The seek time, T_{seek} , depends on the previous position of the head and the speed that the arm moves across the surface. The average seek time in modern drives, $T_{avg\ seek}$, measured by taking the mean of several thousand seeks to random sectors, is typically on the order of 3 to 9 ms. The maximum time for a single seek, $T_{max\ seek}$, can be as high as 20 ms.

. **Rotational latency:** Once the head is in position over the track, the drive waits for the first bit of the target sector to pass under the head. The performance of this step depends on both the position of the surface when the head arrives at the target sector and the rotational speed of the disk. In the worst case, the head just misses the target sector and waits for the disk to make a full rotation. Thus, the maximum rotational latency, in seconds, is given by :

$$T_{max\ rotation} = \frac{1}{\text{RPM}} \times \frac{60\ \text{secs}}{1\ \text{min}}$$

The average rotational latency, $T_{avg\ rotation}$, is simply half of $T_{max\ rotation}$.

. **Transfer time:** When the first bit of the target sector is under the head, the drive can begin to read or write the contents of the sector. The transfer time for one sector depends on the rotational speed and the number of sectors per track. Thus, we can roughly estimate the average transfer time for one sector in seconds as :

$$T_{avg\ transfer} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{average \# sectors/track})} \times \frac{60\ \text{secs}}{1\ \text{min}}$$

We can estimate the average time to access the contents of a disk sector as the sum of the average seek time, the average rotational latency, and the average transfer time.

| Parameter | Value |
|-------------------------|----------|
| Rotational rate | 7200 RPM |
| $T_{avg\ seek}$ | 9 ms |
| Average # sectors/track | 400 |

For this disk, the average rotational latency (in ms) is

$$\begin{aligned} T_{avg\ rotation} &= 1/2 \times T_{max\ rotation} \\ &= 1/2 \times (60\ \text{secs} / 7200\ \text{RPM}) \times 1000\ \text{ms/sec} \\ &\approx 4\ \text{ms} \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\ transfer} &= 60 / 7200\ \text{RPM} \times 1 / 400\ \text{sectors/track} \times 1000\ \text{ms/sec} \\ &\approx 0.02\ \text{ms} \end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned} T_{access} &= T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer} \\ &= 9\ \text{ms} + 4\ \text{ms} + 0.02\ \text{ms} \\ &= 13.02\ \text{ms} \end{aligned}$$

This example illustrates some important points:

- . The time to access the 512 bytes in a disk sector is dominated by the seek time and the rotational latency. Accessing the first byte in the sector takes a long time, but the remaining bytes are essentially free.
- . Since the seek time and rotational latency are roughly the same, twice the seek time is a simple and reasonable rule for estimating disk access time.
- . The access time for a doubleword stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-sized block from memory is roughly 256 ns for SRAM and 4000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2500 times greater than DRAM. The difference in access times is even more dramatic if we compare the times to access a single word.

Logical Disk Blocks

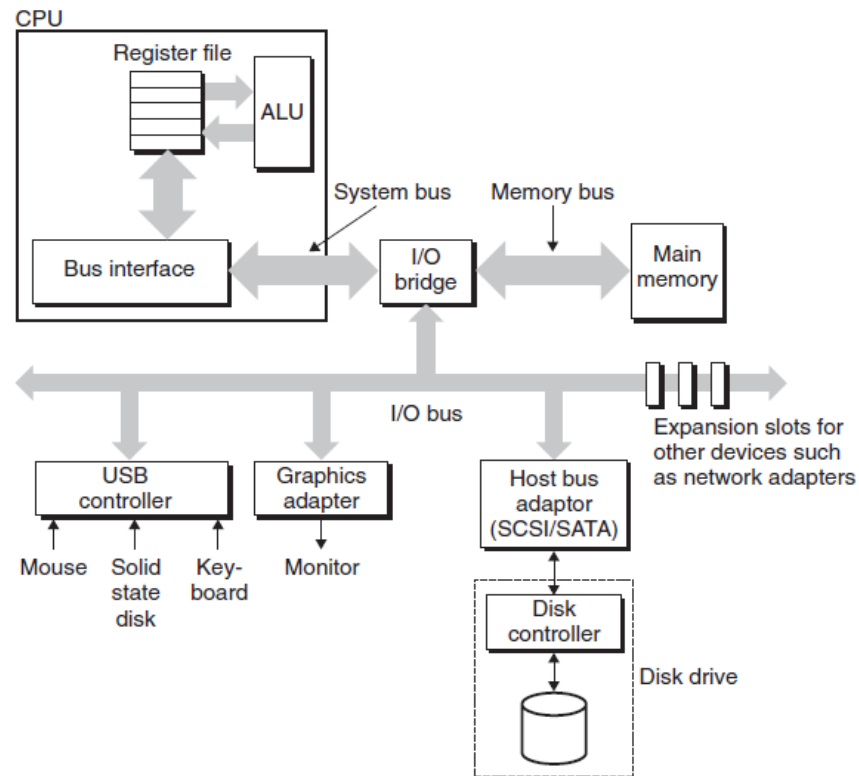
As we have seen, modern disks have complex geometries, with multiple surfaces and different recording zones on those surfaces. To hide this complexity from the operating system, modern disks present a simpler view of their geometry as a sequence of B sector-sized *logical blocks*, numbered $0, 1, \dots, B - 1$. A small hardware/firmware device in the disk package, called the *disk controller*, maintains the mapping between logical block numbers and actual (physical) disk sectors. When the operating system wants to perform an I/O operation such as reading a disk sector into main memory, it sends a command to the disk controller asking it to read a particular logical block number. Firmware on the controller performs a fast table lookup that translates the logical block number into a (*surface, track, sector*) triple that uniquely identifies the corresponding physical sector. Hardware on the controller interprets this triple to move the heads to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed by the head into a small memory buffer on the controller, and copies them into main memory.

Connecting I/O Devices

Input/output (I/O) devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an *I/O bus* such as Intel's *Peripheral Component Interconnect* (PCI) bus. Unlike the system bus and memory buses, which are CPU-specific, I/O buses such as PCI are designed to be independent of the underlying CPU. For example, PCs and Macs both incorporate the PCI bus. Figure 6.11 shows a typical I/O bus structure (modelled on PCI) that connects the CPU, main memory, and I/O devices.

Although the I/O bus is slower than the system and memory buses, it can accommodate a wide variety of third-party I/O devices. For example, the bus in Figure 6.11 has three different types of devices attached to it.

Figure 6.11
Example bus structure that connects the CPU, main memory, and I/O devices.



. A *Universal Serial Bus* (USB) controller is a conduit for devices attached to a USB bus, which is a wildly popular standard for connecting a variety of peripheral I/O devices, including keyboards, mice, modems, digital cameras, game controllers, printers, external disk drives, and solid state disks. USB 2.0 buses have a maximum bandwidth of 60 MB/s. USB 3.0 buses have a maximum bandwidth of 600 MB/s.

. A *graphics card* (or *adapter*) contains hardware and software logic that is responsible for painting the pixels on the display monitor on behalf of the CPU.

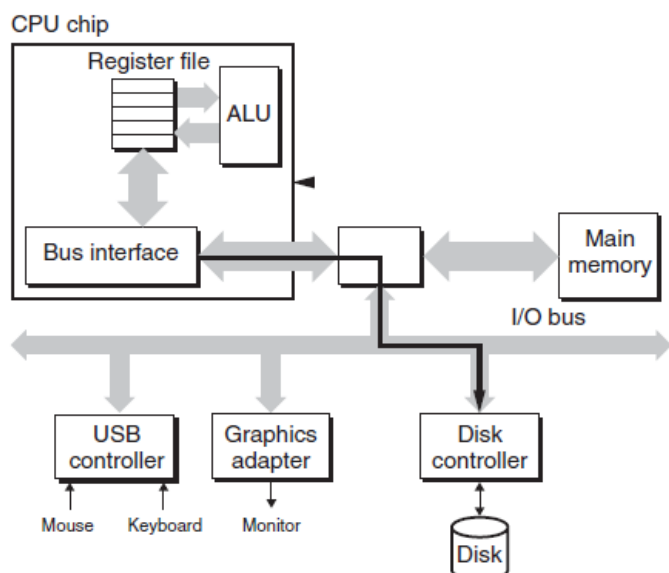
. A *host bus adaptor* that connects one or more disks to the I/O bus using a communication protocol defined by a particular *host bus interface*. The two most popular such interfaces for disks are *SCSI* (pronounced “scuzzy”) and *SATA* (pronounced “sat-uh”). SCSI disks are typically faster and more expensive than SATA drives. A SCSI host bus adaptor (often called a *SCSI controller*) can support multiple disk drives, as opposed to SATA adapters, which can only support one drive.

Additional devices such as *network adapters* can be attached to the I/O bus by plugging the adapter into empty *expansion slots* on the motherboard that provide a direct electrical connection to the bus.

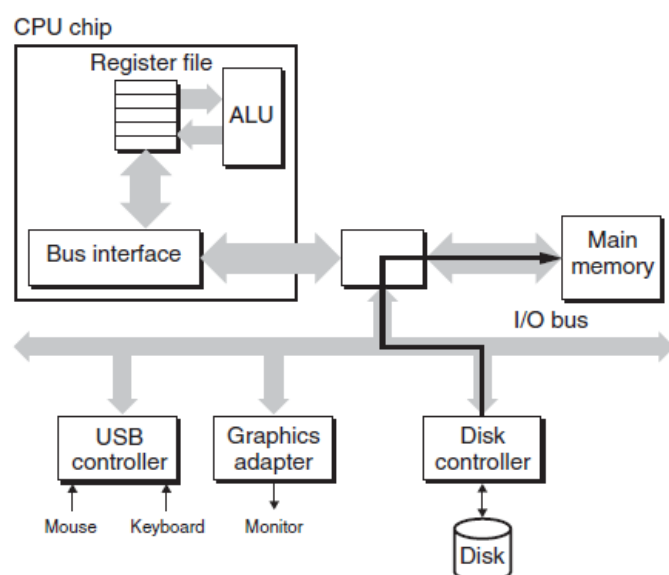
Accessing Disks

While a detailed description of how I/O devices work and how they are programmed is outside our scope here, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.

The CPU issues commands to I/O devices using a technique called *memory-mapped I/O* (Figure 6.12(a)). In a system with memory-mapped I/O, a block of addresses in the address space is reserved for communicating with I/O devices. Each of these addresses is known as an *I/O port*. Each device is associated with (or mapped to) one or more ports when it is attached to the bus.



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



(b) The disk controller reads the sector and performs a DMA transfer into main memory.

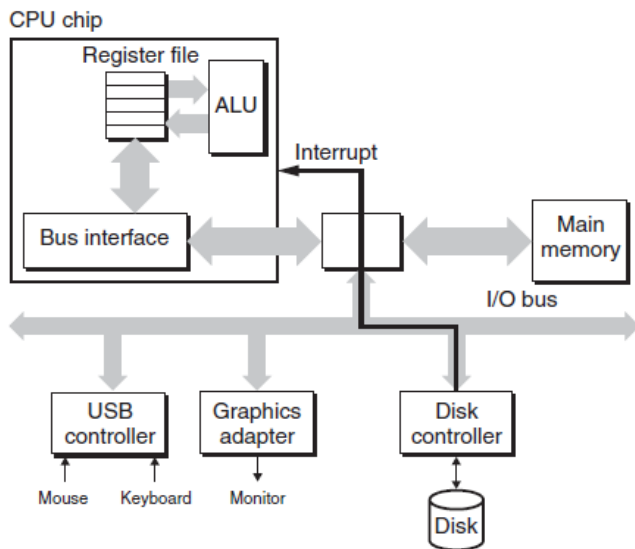
Figure 6.12 Reading a disk sector.

As a simple example, suppose that the disk controller is mapped to port 0xa0. Then the CPU might initiate a disk read by executing three store instructions to address 0xa0: The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished. (We will discuss interrupts in Section 8.1.) The second instruction indicates the logical block number that should be read. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is taking place would be enormously wasteful.

After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU (Figure 6.12(b)). This process, whereby a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as *direct memory access* (DMA). The transfer of data is known as a *DMA transfer*.

After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is currently working on and jump to an operating system routine. The routine records the fact that the I/O has finished and then returns control to the point where the CPU was interrupted.



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

Figure 6.12 (continued) Reading a disk sector.

Solid State Disks

A *solid state disk* (SSD) is a storage technology, based on flash memory (Section 6.1.1), that in some situations is an attractive alternative to the conventional rotating disk. Figure 6.15 shows the basic idea. An SSD package plugs into a standard disk slot on the I/O bus (typically USB or SATA) and behaves like any other disk, processing requests from the CPU to read and write logical disk blocks. An SSD package consists of one or more flash memory chips, which replace the mechanical drive in a conventional rotating disk, and a *flash translation layer*, which is a hardware/firmware device that plays the same role as a disk controller, translating requests for logical blocks into accesses of the underlying physical device.

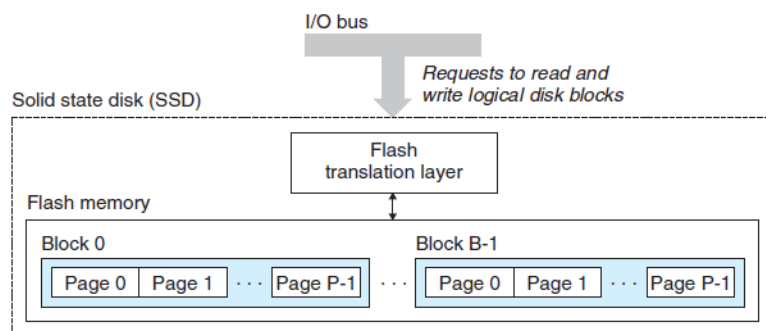


Figure 6.15 Solid state disk (SSD).

| Reads | | Writes | |
|----------------------------|------------|-----------------------------|-------------|
| Sequential read throughput | 250 MB/s | Sequential write throughput | 170 MB/s |
| Random read throughput | 140 MB/s | Random write throughput | 14 MB/s |
| Random read access time | 30 μ s | Random write access time | 300 μ s |

Figure 6.16 Performance characteristics of a typical solid state disk. Source: Intel X25-E SATA solid state drive product manual.

SSDs have different performance characteristics than rotating disks. As shown in Figure 6.16, sequential reads and writes (where the CPU accesses logical disk blocks in sequential order) have comparable performance, with sequential reading somewhat faster than sequential writing. However, when logical blocks are accessed in random order, writing is an order of magnitude slower than reading.

The difference between random reading and writing performance is caused by a fundamental property of the underlying flash memory. As shown in Figure 6.15, a flash memory consists of a sequence of B blocks, where each block consists of P pages. Typically, pages are 512–4KB in size, and a block consists of 32–128 pages, with total block sizes ranging from 16 KB to 512 KB. Data is read and written in units of pages. A page can be written only after the entire block to which it belongs has been *erased* (typically this means that all bits in the block are set to 1). However, once a block is erased, each page in the block can be written once with no further erasing. A block wears out after roughly 100,000 repeated writes. Once a block wears out it can no longer be used.

Random writes are slow for two reasons. First, erasing a block takes a relatively long time, on the order of 1 ms, which is more than an order of magnitude longer than it takes to access a page. Second, if a write operation attempts to modify a page p that contains existing data (i.e., not all ones), then any pages in the same block with useful data must be copied to a new (erased) block before the write to page p can occur. Manufacturers have developed sophisticated logic in the flash translation layer that attempts to amortize the high cost of erasing blocks and to minimize the number of internal copies on writes, but it is unlikely that random writing will ever perform as well as reading.

SSDs have a number of advantages over rotating disks. They are built of semiconductor memory, with no moving parts, and thus have much faster random access times than rotating disks, use less power, and are more rugged. However, there are some disadvantages. First, because flash blocks wear out after repeated writes, SSDs have the potential to wear out as well. *Wear leveling* logic in the flash translation layer attempts to maximize the lifetime of each block by spreading erasures evenly across all blocks, but the fundamental limit remains. Second, SSDs are about 100 times more expensive per byte than rotating disks, and thus the typical storage capacities are 100 times less than rotating disks. However, SSD prices are decreasing rapidly as they become more popular, and the gap between the two appears to be decreasing. SSDs have completely replaced rotating disks in portable music devices, are popular as disk replacements in laptops, and have even begun to appear in desktops and servers. While rotating disks are here to stay, it is clear that SSDs are an important new storage technology.

Storage Technology Trends

Different storage technologies have different price and performance trade-offs. SRAM is somewhat faster than DRAM, and DRAM is much faster than disk. On the other hand, fast storage is always more expensive than slower storage. SRAM costs more per byte than DRAM. DRAM costs much more than disk. SSDs split the difference between DRAM and rotating disk.

The price and performance properties of different storage technologies are changing at dramatically different rates. Figure 6.17 summarizes the price and performance properties of storage technologies since 1980, when the first PCs were introduced. The numbers were culled from back issues of trade magazines and the Web. Although they were collected in an informal survey, the numbers reveal some interesting trends.

| Metric | 1980 | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2010:1980 |
|-------------|--------|------|------|------|------|------|------|-----------|
| \$/MB | 19,200 | 2900 | 320 | 256 | 100 | 75 | 60 | 320 |
| Access (ns) | 300 | 150 | 35 | 15 | 3 | 2 | 1.5 | 200 |

(a) SRAM trends

| Metric | 1980 | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2010:1980 |
|-------------------|-------|-------|------|------|------|------|-------|-----------|
| \$/MB | 8000 | 880 | 100 | 30 | 1 | .1 | 0.06 | 130,000 |
| Access (ns) | 375 | 200 | 100 | 70 | 60 | 50 | 40 | 9 |
| Typical size (MB) | 0.064 | 0.256 | 4 | 16 | 64 | 2000 | 8,000 | 125,000 |

(b) DRAM trends

| Metric | 1980 | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2010:1980 |
|-------------------|------|------|------|------|--------|---------|-----------|-----------|
| \$/MB | 500 | 100 | 8 | 0.30 | 0.01 | 0.005 | 0.0003 | 1,600,000 |
| Seek time (ms) | 87 | 75 | 28 | 10 | 8 | 5 | 3 | 29 |
| Typical size (MB) | 1 | 10 | 160 | 1000 | 20,000 | 160,000 | 1,500,000 | 1,500,000 |

(c) Rotating disk trends

| Metric | 1980 | 1985 | 1990 | 1995 | 2000 | 2003 | 2005 | 2010 | 2010:1980 |
|----------------------|------|-------|-------|-------|-------|---------|--------|---------|-----------|
| Intel CPU | 8080 | 80286 | 80386 | Pent. | P-III | Pent. 4 | Core 2 | Core i7 | — |
| Clock rate (MHz) | 1 | 6 | 20 | 150 | 600 | 3300 | 2000 | 2500 | 2500 |
| Cycle time (ns) | 1000 | 166 | 50 | 6 | 1.6 | 0.30 | 0.50 | 0.4 | 2500 |
| Cores | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 4 | 4 |
| Eff. cycle time (ns) | 1000 | 166 | 50 | 6 | 1.6 | 0.30 | 0.25 | 0.10 | 10,000 |

(d) CPU trends

Figure 6.17 Storage and processing technology trends.

Memory Organization(Contd..)

• Locality (Class Notes)

Well-written computer programs tend to exhibit good *locality*. That is, they tend to reference data items that are near other recently referenced data items, or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

The basis for the performance advantage of a two-level memory is a principle known as **locality of reference**. This principle states that memory references tend to cluster. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Intuitively, the principle of locality makes sense. Consider the following line of reasoning:

1. Except for branch and call instructions, which constitute only a small fraction of all program instructions, program execution is sequential. Hence, in most cases, the next instruction to be fetched immediately follows the last instruction fetched.

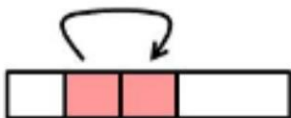
2. It is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, a program remains confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time references to instructions tend to be localized to a few procedures.
3. Most iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of the iteration, computation is therefore confined to a small contiguous portion of a program.
4. In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures will be to closely located data items.

Locality is typically described as having two distinct forms: *temporal locality* and *spatial locality*.

Temporal locality refers to the tendency for a processor to access memory locations that have been used recently. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly.



Spatial locality refers to the tendency of execution to involve a number of memory locations that are clustered. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data.



In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Traditionally, temporal locality is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. Recently, there has been considerable research on refining these techniques to achieve greater performance, but the basic strategies remain the same.

Programmers should understand the principle of locality because, in general, *programs with good locality run faster than programs with poor locality*. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality. At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as *cache memories* that hold blocks of the most recently referenced instructions and data items. At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system. The principle of locality also plays a crucial role in the design of application programs. For example, Web

browsers exploit temporal locality by caching recently referenced documents on a local disk. High-volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

• Memory Hierarchy(T1: 4.1)

Previous sections described some fundamental and enduring properties of storage technology and computer software:

. *Storage technology*: Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.

. *Computer software*: Well-written programs tend to exhibit good locality.

In one of the happier coincidences of computing, these fundamental properties of hardware and software complement each other beautifully. Their complementary nature suggests an approach for organizing memory systems, known as the *memory hierarchy*, that is used in all modern computer systems. Figure 6.23 shows a typical memory hierarchy. In general, the storage devices get slower, cheaper, and larger as we move from higher to lower *levels*. At the highest level (L0) are a small number of fast CPU registers that the CPU can access in a single clock cycle. Next are one or more small to moderate-sized SRAM-based cache memories that can be accessed in a few CPU clock cycles. These are followed by a large DRAM based main memory that can be accessed in tens to hundreds of clock cycles. Next are slow but enormous local disks. Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network. For example, distributed file systems such as the Andrew File System (AFS) or the Network File System (NFS) allow a program to access files that are stored on remote network-connected servers. Similarly, the World Wide Web allows programs to access remote files stored on Web servers anywhere in the world.

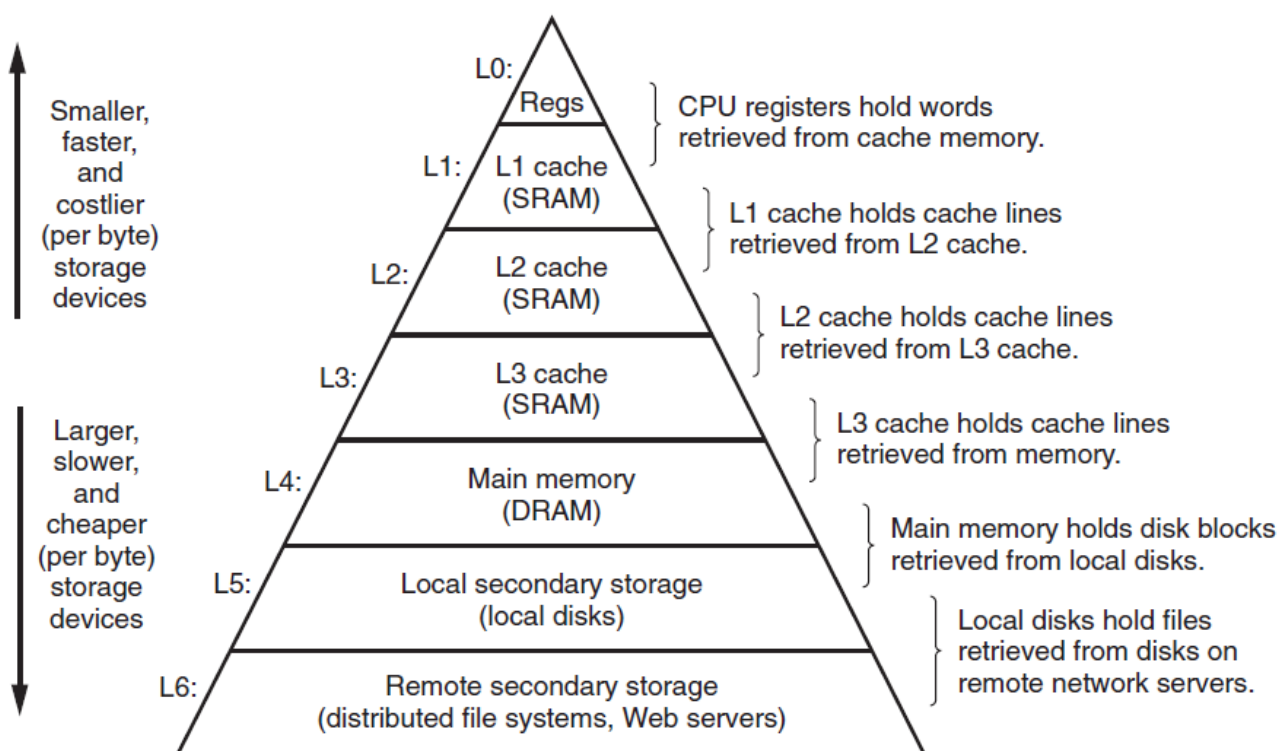


Figure 6.23 The memory hierarchy.

The design constraints on a computer's memory can be summed up by three questions:

How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components. As might be expected, there is a trade-off among the three key characteristics of memory: capacity, access time, and cost.

A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit;
- Greater capacity, smaller cost per bit;
- Greater capacity, slower access time.

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large- capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower- capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:

- a. Decreasing cost per bit;
- b. Increasing capacity;
- c. Increasing access time;
- d. Decreasing frequency of access of the memory by the processor.

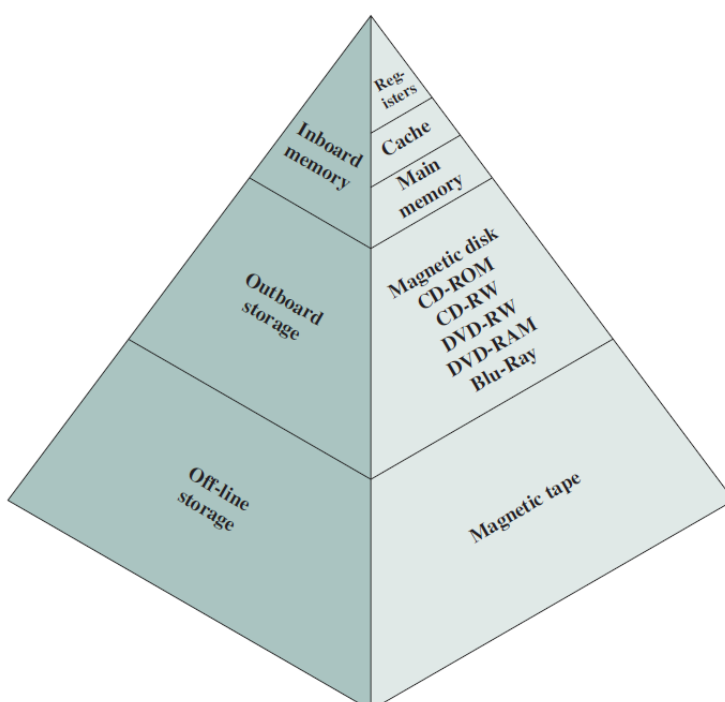


Figure 4.1 The Memory Hierarchy

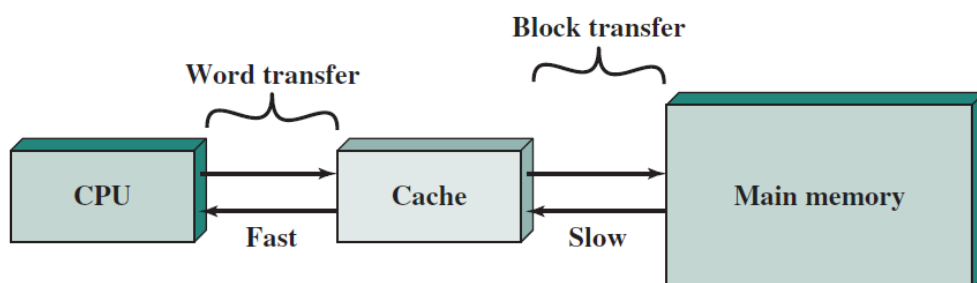
Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

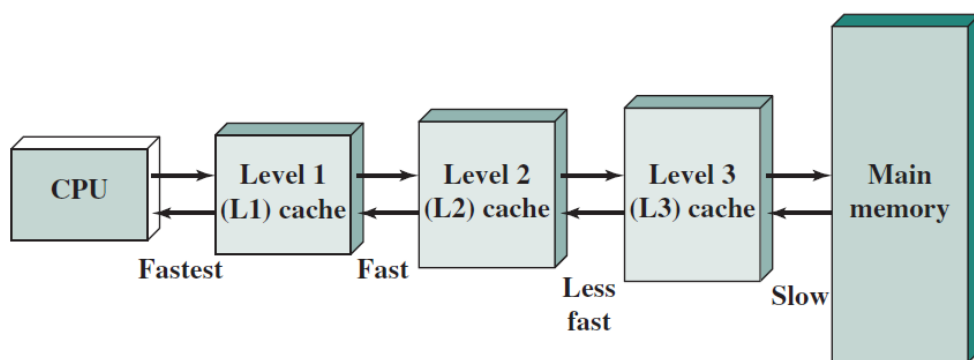
• Cache Memories (T1: 4.2-4.3)

Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory. The concept is illustrated in Figure 4.3a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

Figure 4.3b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.



(a) Single cache



(b) Three-level cache organization

Figure 4.3 Cache and Main Memory

Figure 4.4 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n -bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = 2^n/K$ blocks in main memory. The cache consists of m blocks, called **lines**. Each line contains K words, plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each “word” being a single byte; in this case the line size is 4 bytes. The number of lines is considerably less than the number of main memory blocks ($m \ll M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

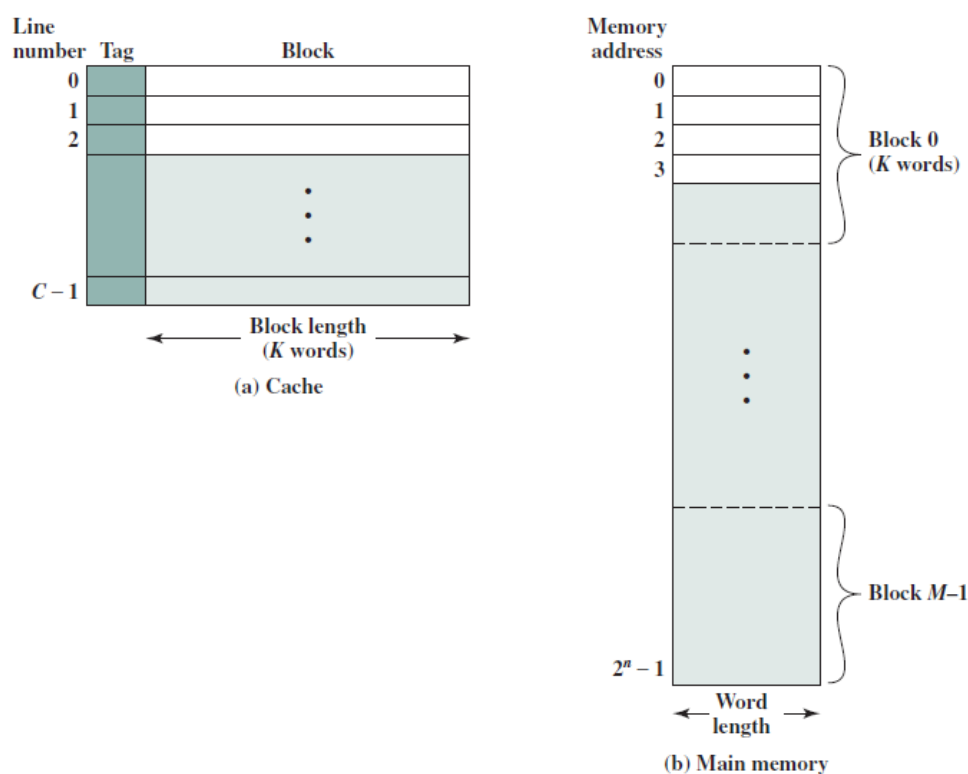


Figure 4.4 Cache/Main Memory Structure

Figure 4.5 illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor. Figure 4.5 shows these last two operations occurring in parallel and reflects the organization shown in Figure 4.6, which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor. In other organizations, the cache is physically interposed between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

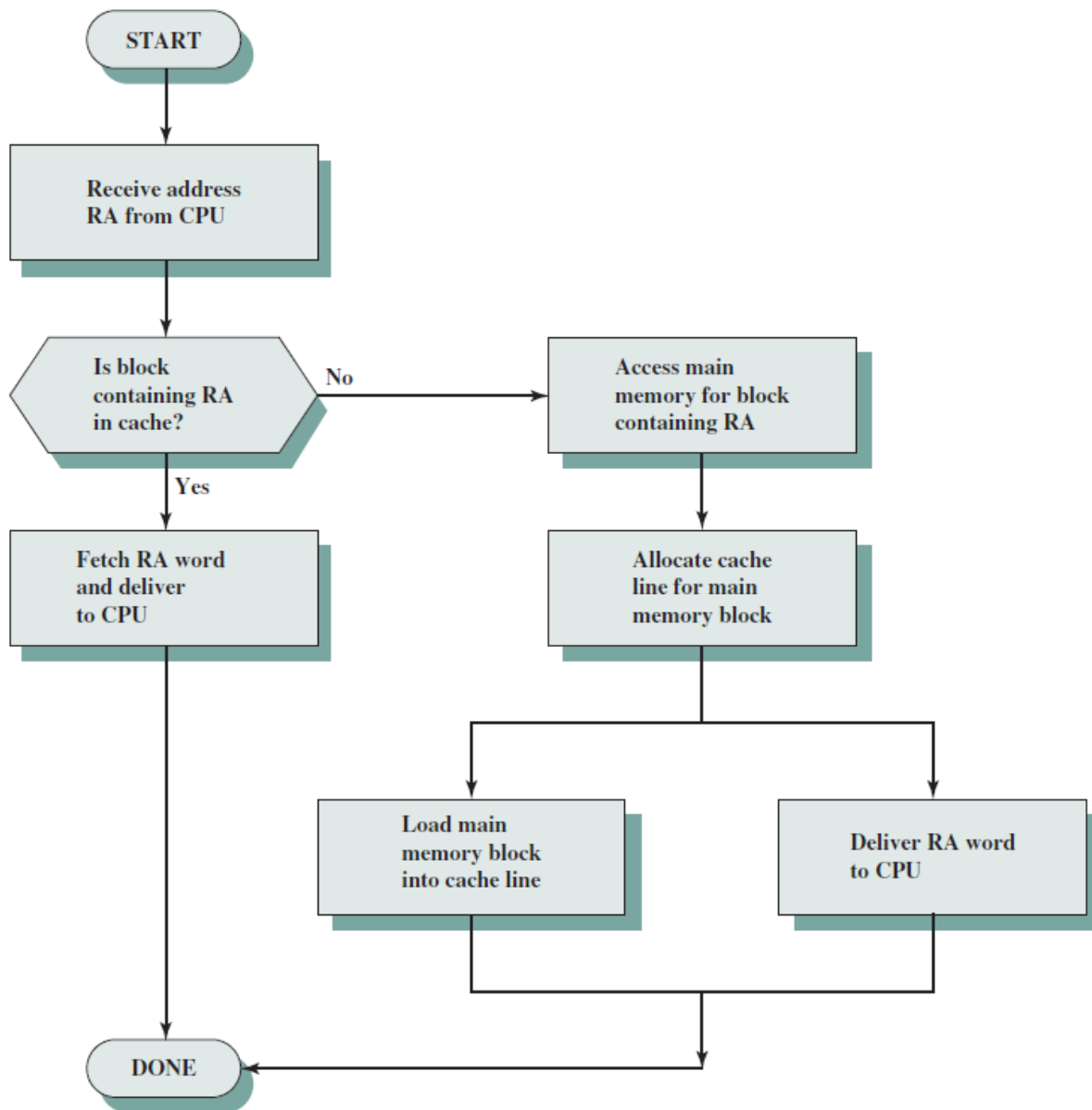


Figure 4.5 Cache Read Operation

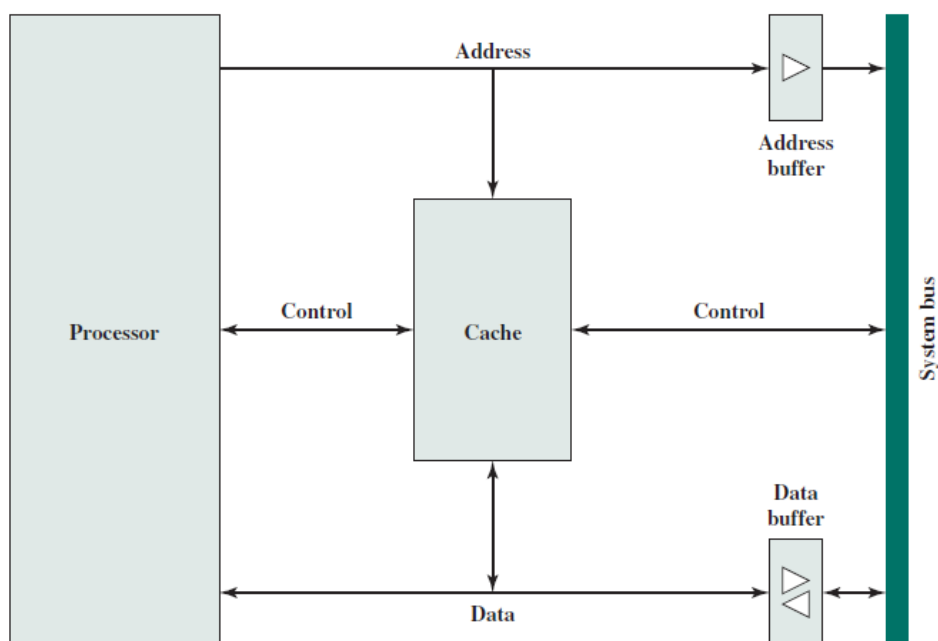


Figure 4.6 Typical Cache Organization

Mapping Function

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set-associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example.

Direct-Mapped Caches

DIRECT MAPPING The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

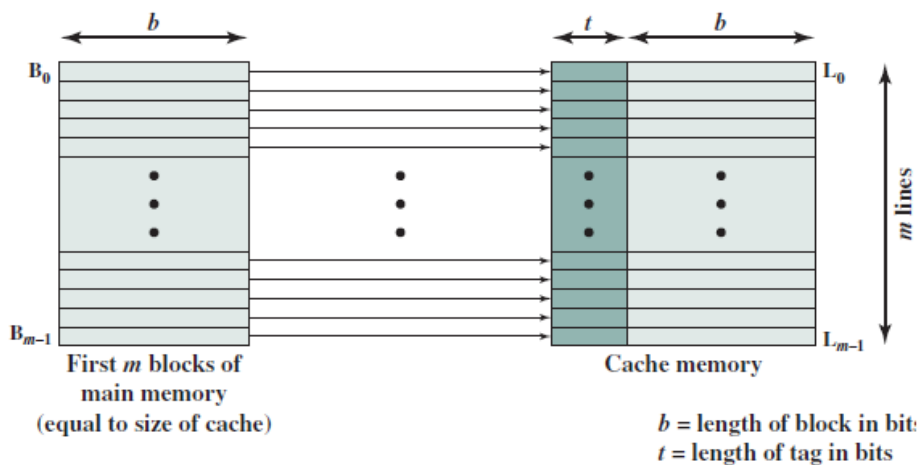
where

i = cache line number

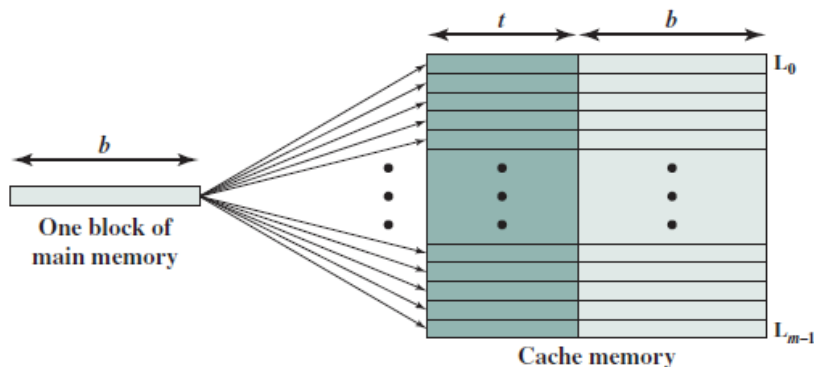
j = main memory block number

m = number of lines in the cache

Figure 4.8a shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1 , and so on.



(a) Direct mapping



(b) Associative mapping

Figure 4.8 Mapping from Main Memory to Cache: Direct and Associative

The mapping function is easily implemented using the main memory address. Figure 4.9 illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of $s - r$ bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = $m = 2^r$
- Size of cache = 2^{r+w} words or bytes
- Size of tag = $(s - r)$ bits

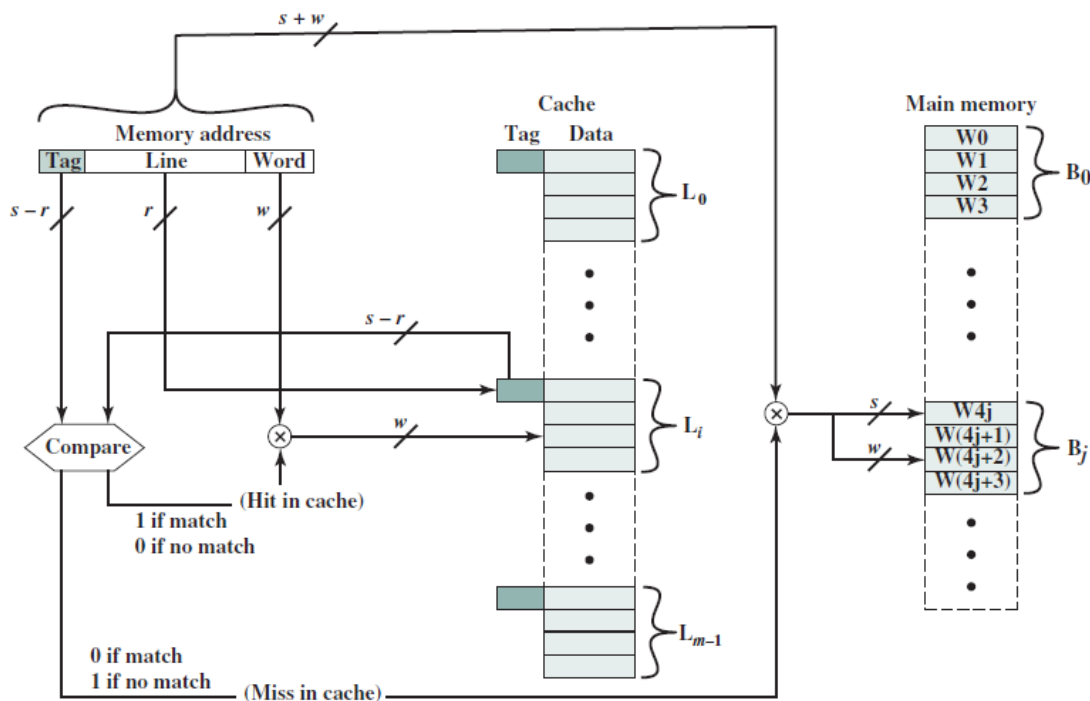


Figure 4.9 Direct-Mapping Cache Organization

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

| Cache line | Main memory blocks assigned |
|------------|---|
| 0 | $0, m, 2m, \dots, 2^s - m$ |
| 1 | $1, m + 1, 2m + 1, \dots, 2^s - m + 1$ |
| \vdots | \vdots |
| $m - 1$ | $m - 1, 2m - 1, 3m - 1, \dots, 2^s - 1$ |

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).

One approach to lower the **miss** penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a victim cache. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory.

- oFullyAssociative Caches

Memory Organization(Contd..)

●Cache Memories(Contd..)

- oSet Associative Caches(T1: 4.3)

- oIssues with Writes(T1: 4.3)

- oPerformance Impact of Cache Parameters(T1: 4.3)

- oWriting Cache friendly Codes(R2: 6.5)

- oReplacement Algorithms(Class Notes)

Instruction Set Architecture -CISC Vs RISC

●CISC Instruction Set(Intel x86as an example)(T1: 12.1 –12.5, 13.1 –13.4)

- oMachine Instruction Characteristics

- oTypes of Operands

- oTypes of Operations

- oAddressing Modes

- oInstruction Formats

Instruction Set Architecture -CISC Vs RISC (Contd...)

•RISC Instruction Architecture (MIPS as an Example)(R1: Selected topics from Chapter 2 and Chapter 4, and Class Notes)

- oMachine Instruction Characteristics
- oTypes of Operands
- oTypes of Operations
- oAddressing Modes
- oInstruction Formats
- oSingle cycle implementation

Instruction Set Architecture -CISC Vs RISC (Contd...) **(Selected topics from T1, R1: Chapter 4 and Class Notes)**

•RISC Instruction Architecture (MIPS as an Example) (Contd..)

- oMulticycle Implementation

•Control Unit

- oMicroprogrammed control unit
- oHardwired Control Unit (MIPS as an example)

Instruction Set Architecture -CISC Vs RISC (Contd...)

•Pipeline(T1: 14.4)

- oOverview of pipeline
- oResource Hazard
- oData Hazard : Forwarding versus Stalling
- oControl Hazard

Process Management(T2: 3.1-3.3 and 4.1-4.3)

•Concept of Process

•Process State Diagram

•Operations on Processes: Process creation and termination examples

•Process vs. Threads

•Multithreading Models

Process Management(Contd...)(T2: 6.1- 6.3)

•Process Scheduling criteria

•Process Scheduling Algorithms -FCFS, SJF, Priority, RR, Multilevel Queue, Multilevel Feedback Queue

Process Coordination(T2: 5.1-5.6, 7.1, 7.2)

•TheCritical section problem

•Peterson's Solution

•Synchronization Hardware

•Semaphores

•Deadlock:

- oSystem Model

- oDeadlock Characterization

Process Coordination(Contd...)

•Methods of Handling Deadlocks(T2: 7.3-7.7)

- oDeadlock Prevention

- oDeadlock Avoidance: Banker's Algorithm

- oDeadlock Detection

- oRecovery from Deadlock

Memory Management(T2: 8.1-8.6)

•Memory-Management Strategies

•Swapping

•Partitioning

•Paging

•Segmentation

Memory Management(Contd...)(T2: 9.1-9.4)

•Virtual-Memory

•Demand Paging

•Page Replacement Algorithms: FIFO, Optimal, LRU andLFU

Optimizing Program Performance(R2: 5.1:5.5)

•Capabilities and Limitations of Optimizing Compilers

•Expressing Program Performance

•Eliminating Loop Inefficiencies

•Reducing Procedure Calls

Optimizing Program Performance(R2: 5.6:5.9)

•Eliminating Unneeded Memory References

•Understanding Modern Processors

•Loop Unrolling

•Enhancing Parallelism