PART 1 Background

CHAPTER 1

COMPUTER SYSTEM OVERVIEW

ISBN 1-256-52023-3

No artifact designed by man is so convenient for this kind of functional description as a digital computer. Almost the only ones of its properties that are detectable in its behavior are the organizational properties. Almost no interesting statement that one can make about on operating computer bears any particular relation to the specific nature of the hardware. A computer is an organization of elementary functional components in which, to a high approximation, only the function performed by those components is relevant to the behavior of the whole system.

THE SCIENCES OF THE ARTIFICIAL, HERBERT SIMON

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Describe the basic elements of a computer system and their interrelationship.
- Explain the steps taken by a processor to execute an instruction.
- Understand the concept of interrupts and how and why a processor uses interrupts.
- List and describe the levels of a typical computer memory hierarchy.
- Explain the basic characteristics of multiprocessor and multicore organizations.
- Discuss the concept of locality and analyze the performance of a multilevel memory hierarchy.
- Understand the operation of a stack and its use to support procedure call and return.

An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users. The OS also manages secondary memory and I/O (input/output) devices on behalf of its users. Accordingly, it is important to have some understanding of the underlying computer system hardware before we begin our examination of operating systems.

This chapter provides an overview of computer system hardware. In most areas, the survey is brief, as it is assumed that the reader is familiar with this subject. However, several areas are covered in some detail because of their importance to topics covered later in the book. Further topics are covered in Appendix C.

1.1 BASIC ELEMENTS

At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs. Thus, there are four main structural elements:

- **Processor:** Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the **central processing unit** (CPU).

ISBN 1-256-52023-3

- **Main memory:** Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. In contrast, the contents of disk memory are retained even when the computer system is shut down. Main memory is also referred to as *real memory* or *primary memory*.
- **I/O modules:** Move data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e.g., disks), communications equipment, and terminals.
- **System bus:** Provides for communication among processors, main memory, and I/O modules.

Figure 1.1 depicts these top-level components. One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal (to the processor) 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 which receives

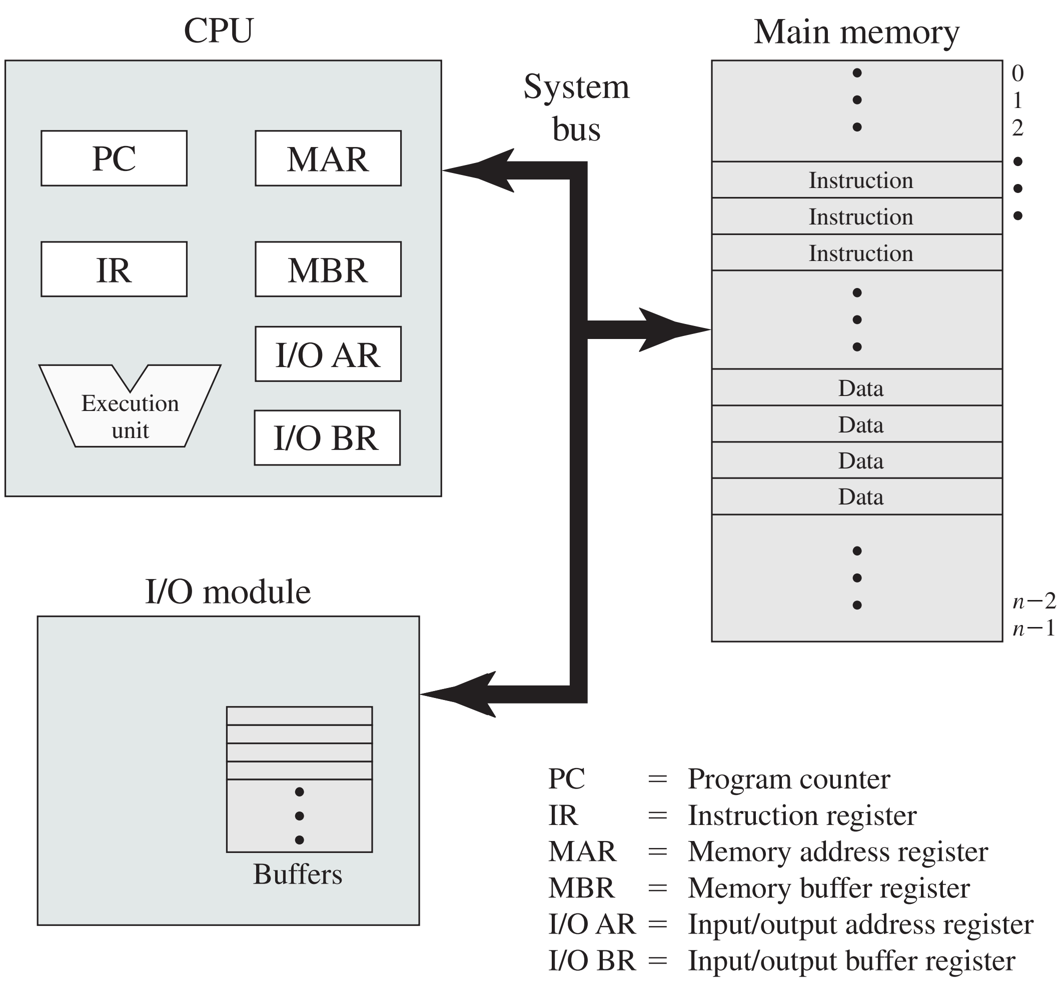


Figure 1.1 Computer Components: Top-Level View

ISBN 1-256-52023-3

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 processor.

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

1.2 EVOLUTION OF THE MICROPROCESSOR

The hardware revolution that brought about desktop and handheld computing was the invention of the microprocessor, which contained a processor on a single chip. Though originally much slower than multichip processors, microprocessors have continually evolved to the point that they are now much faster for most computations due to the physics involved in moving information around in sub-nanosecond timeframes.

Not only have microprocessors become the fastest general purpose processors available, they are now multiprocessors; each chip (called a socket) contains multiple processors (called cores), each with multiple levels of large memory caches, and multiple logical processors sharing the execution units of each core. As of 2010, it is not unusual for even a laptop to have 2 or 4 cores, each with 2 hardware threads, for a total of 4 or 8 logical processors.

Although processors provide very good performance for most forms of computing, there is increasing demand for numerical computation. Graphical Processing Units (GPUs) provide efficient computation on arrays of data using Single-Instruction Multiple Data (SIMD) techniques pioneered in supercomputers. GPUs are no longer used just for rendering advanced graphics, but they are also used for general numerical processing, such as physics simulations for games or computations on large spreadsheets. Simultaneously, the CPUs themselves are gaining the capability of operating on arrays of data—with increasingly powerful vector units integrated into the processor architecture of the x86 and AMD64 families.

Processors and GPUs are not the end of the computational story for the modern PC. Digital Signal Processors (DSPs) are also present, for dealing with streaming signals—such as audio or video. DSPs used to be embedded in I/O devices, like modems, but they are now becoming first-class computational devices, especially in handhelds. Other specialized computational devices (fixed function units) co-exist with the CPU to support other standard computations, such as encoding/decoding speech and video (codecs), or providing support for encryption and security.

To satisfy the requirements of handheld devices, the classic microprocessor is giving way to the System on a Chip (SoC), where not just the CPUs and caches are on the same chip, but also many of the other components of the system, such as DSPs, GPUs, I/O devices (such as radios and codecs), and main memory.

ISBN 1-256-52023-3

1.3 INSTRUCTION EXECUTION

A program to be executed by a processor consists of a set of instructions stored in memory. 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. Instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an *instruction cycle.* Using a simplified two-step description, the instruction cycle is depicted in Figure 1.2. The two steps are referred to as the *fetch stage* and the *execute stage.* Program execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Unless instructed 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). For example, consider a simplified computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. 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 subsequently.

The fetched instruction is loaded into 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

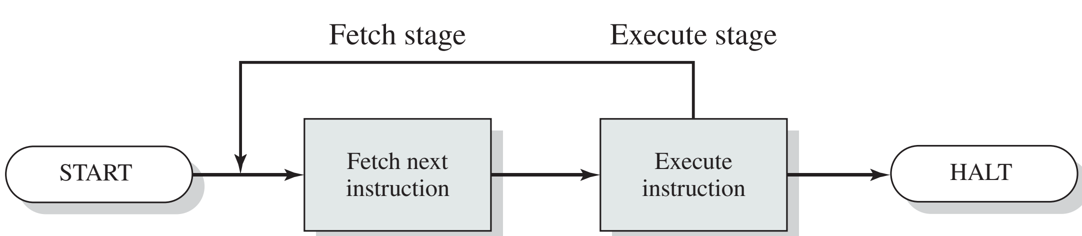


Figure 1.2 Basic Instruction Cycle

ISBN 1-256-52023-3

(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 1.3 Characteristics of a Hypothetical Machine

specifies that the next instruction will be from location 182. The processor sets the program counter to 182. Thus, on the next fetch stage, 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 processor that includes the characteristics listed in Figure 1.3. The processor contains a single data register, called the accumulator (AC). Both instructions and data are 16 bits long, and memory is organized as a sequence of 16-bit words. The instruction format provides 4 bits for the opcode, allowing as many as $2^4 = 16$ different opcodes (represented by a single hexadecimal[1] digit). The opcode defines the operation the processor is to perform. With the remaining 12 bits of the instruction format, up to $2^{12} = 4{,}096$ (4K) words of memory (denoted by three hexadecimal digits) can be directly addressed.

Figure 1.4 illustrates a partial program execution, showing the relevant portions of memory and processor registers. 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 stages, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR)

ISBN 1-256-52023-3

[1]A basic refresher on number systems (decimal, binary, hexadecimal) can be found at the Computer Science Student Resource Site at ComputerScienceStudent.com.

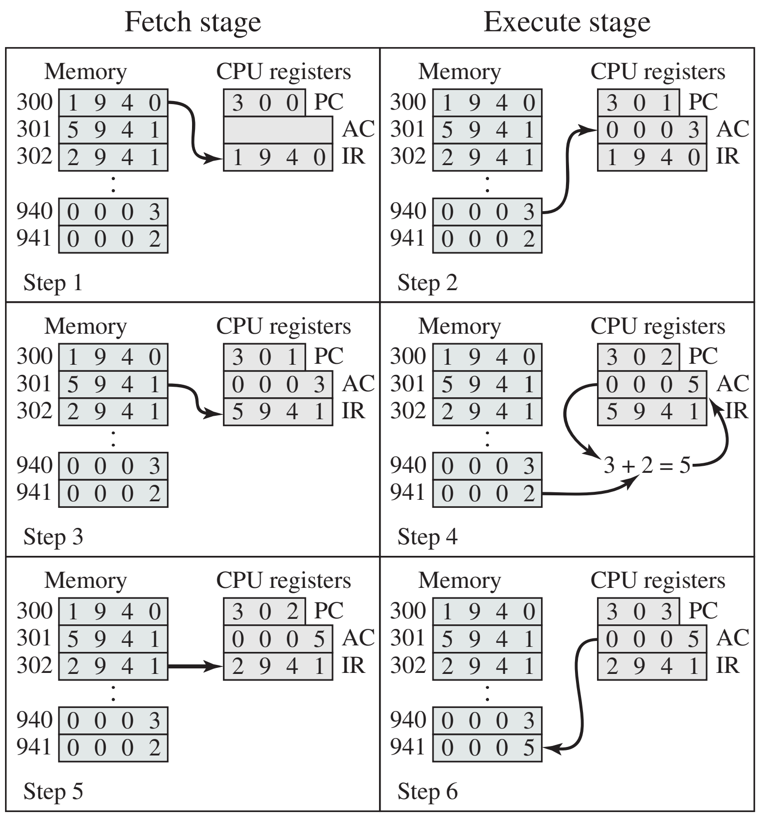


Figure 1.4 **Example of Program Execution** (contents of memory and registers in hexadecimal)

and a memory buffer register (MBR). For simplicity, these intermediate registers are not shown.

2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory. The remaining 12 bits (three hexadecimal digits) specify the address, which is 940.
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.

In this example, three instruction cycles, each consisting of a fetch stage and an execute stage, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer instruction cycles would be needed. Most modern processors include instructions that contain more than one address. Thus the execution stage 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.

ISBN 1-256-52023-3

1.4 INTERRUPTS

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. Table 1.1 lists the most common classes of interrupts.

Interrupts are provided primarily as a way to improve processor utilization. For example, most I/O devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 1.2. 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 thousands or even millions of instruction cycles. Clearly, this is a very wasteful use of the processor.

To give a specific example, consider a PC that operates at 1 GHz, which would allow roughly 10^9 instructions per second.[2] A typical hard disk has a rotational speed of 7200 revolutions per minute for a half-track rotation time of 4 ms, which is 4 million times slower than the processor.

Figure 1.5a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. The solid vertical lines represent segments of code in a program. 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 routine that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

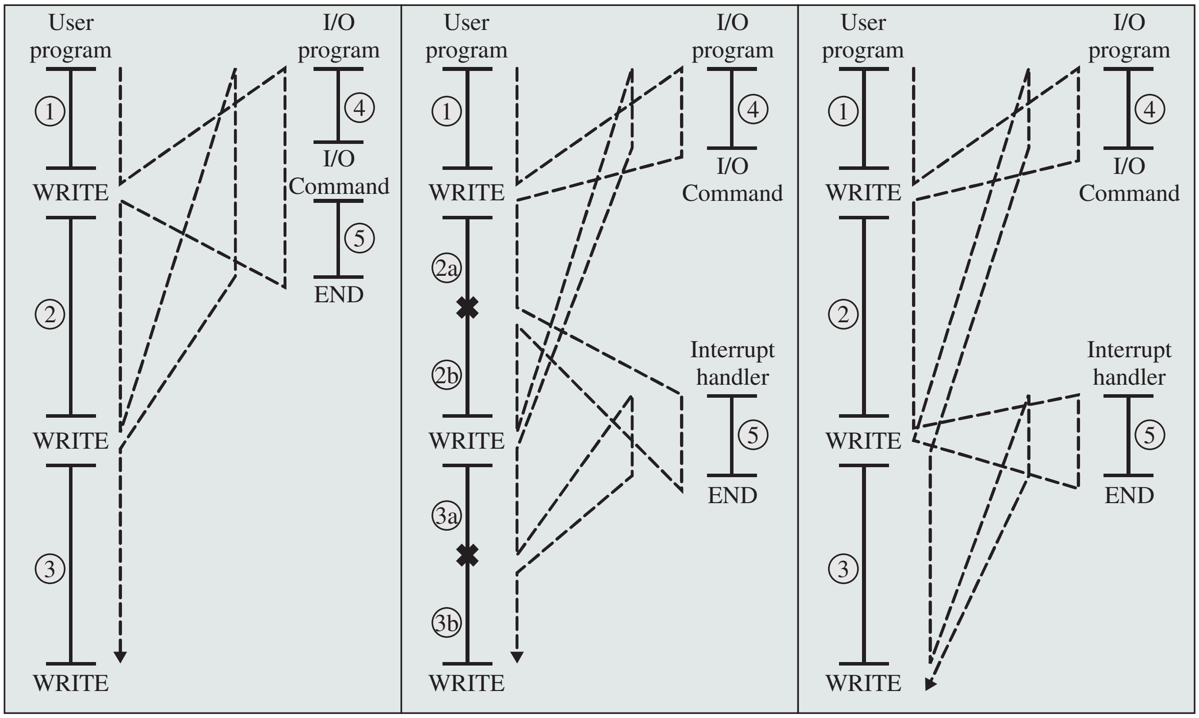
- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command. Without the use of interrupts, once this command is issued,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

Table 1.1 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, and 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 or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

[2]A discussion of the uses of numerical prefixes, such as giga and tera, is contained in a supporting document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

ISBN 1-256-52023-3



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

Figure 1.5 Program Flow of Control without and with Interrupts

The dashed line represents the path of execution followed by the processor; that is, this line shows the sequence in which instructions are executed. Thus, after the first WRITE instruction is encountered, the user program is interrupted and execution continues with the I/O program. After the I/O program execution is complete, execution resumes in the user program immediately following the WRITE instruction.

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 1.5b. 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 routine to service

ISBN 1-256-52023-3

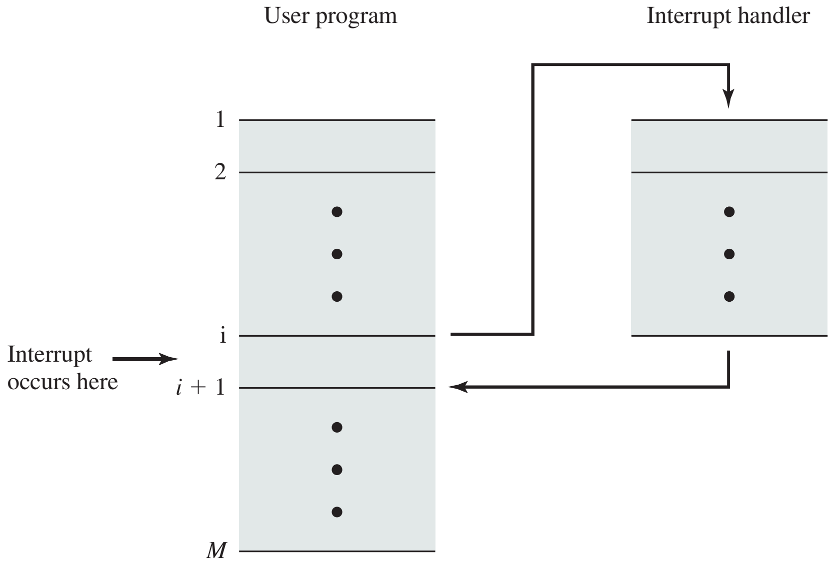


Figure 1.6 Transfer of Control via Interrupts

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 ✖ in Figure 1.5b. Note that an interrupt can occur at any point in the main program, not just at one specific instruction.

For the user program, an interrupt suspends the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 1.6). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the OS are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt stage* is added to the instruction cycle, as shown in Figure 1.7 (compare Figure 1.2). In the interrupt stage, 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 stage and fetches the next instruction of the current program. If an interrupt is pending,

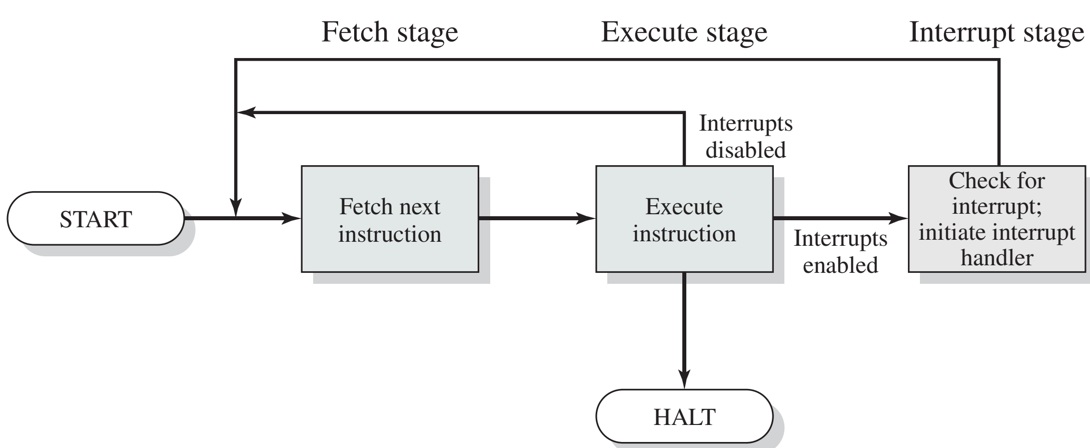


Figure 1.7 Instruction Cycle with Interrupts

ISBN 1-256-52023-3

the processor suspends execution of the current program and executes an *interrupt-handler* routine. The interrupt-handler routine is generally part of the OS. Typically, this routine 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 1.8, which is a timing diagram based on the flow of control in Figures 1.5a and 1.5b. Figures 1.5b and 1.8

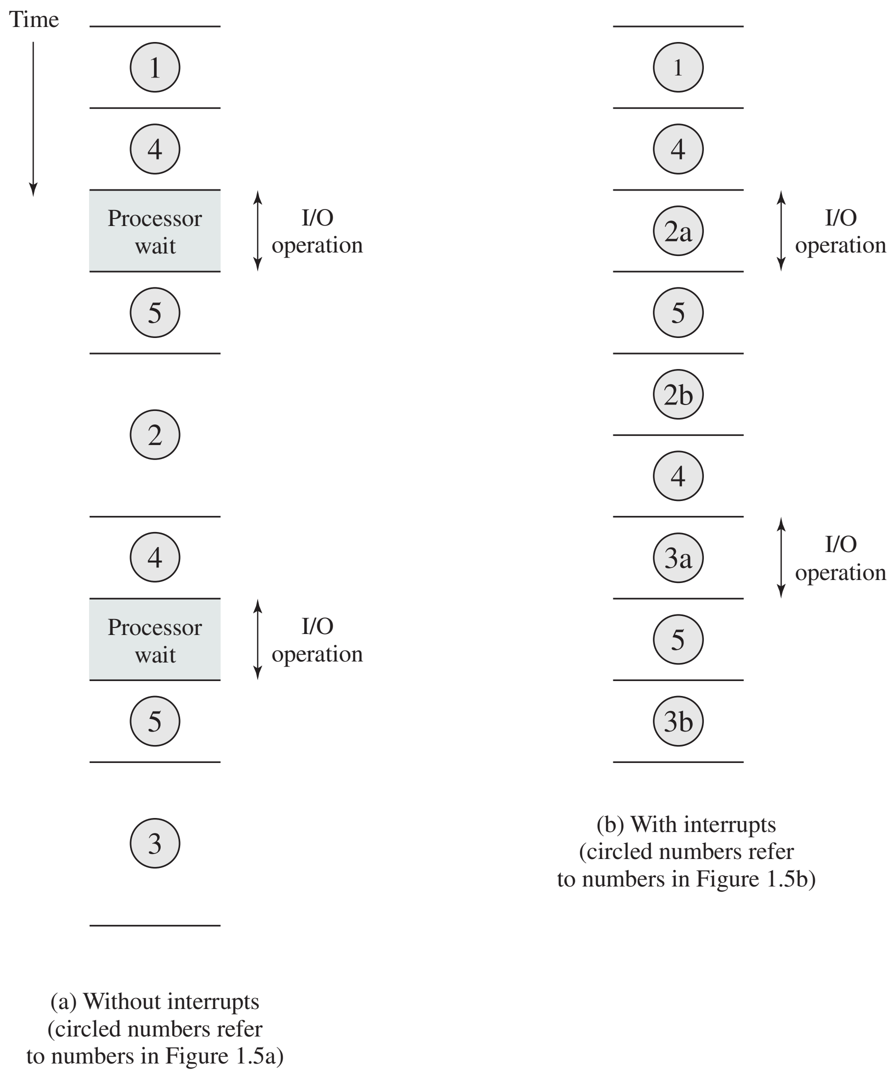


Figure 1.8 Program Timing: Short I/O Wait

ISBN 1-256-52023-3

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. 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 1.5c 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 1.9 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 underway overlaps with the execution of user instructions.

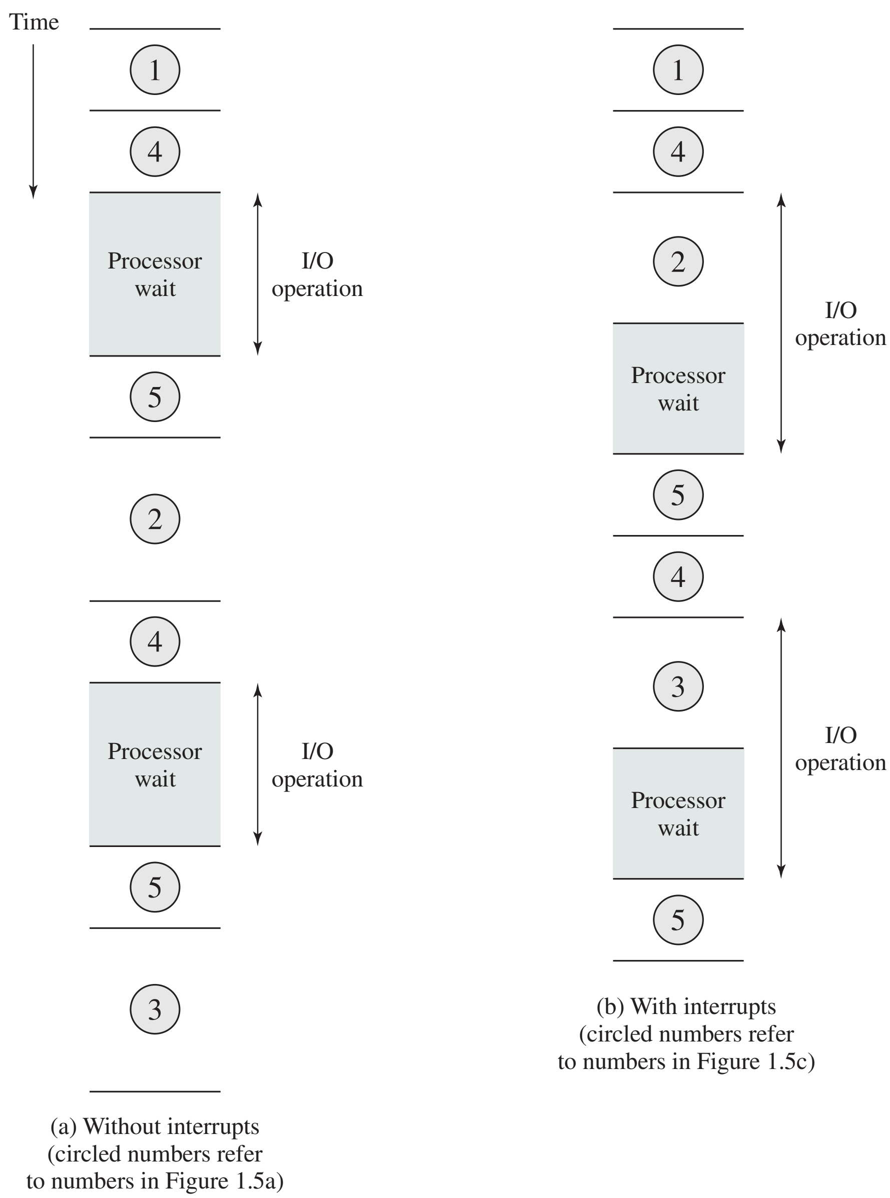


Figure 1.9 Program Timing: Long I/O Wait

ISBN 1-256-52023-3

Interrupt Processing

An interrupt triggers a number of events, both in the processor hardware and in software. Figure 1.10 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure 1.7.
3. The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor next needs to prepare to transfer control to the interrupt routine. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word[3] (PSW) and the location of the next instruction to be executed, which

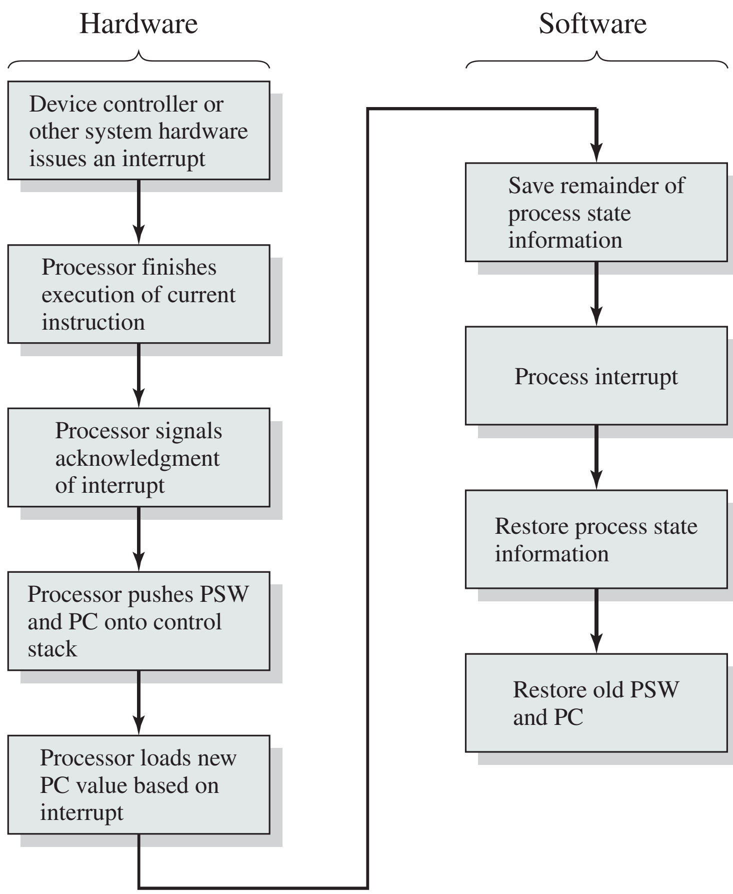


Figure 1.10 Simple Interrupt Processing

[3]The PSW contains status information about the currently running process, including memory usage information, condition codes, and other status information, such as an interrupt enable/disable bit and a kernel/user mode bit. See Appendix C for further discussion.

ISBN 1-256-52023-3

is contained in the program counter (PC). These can be pushed onto a control stack (see Appendix P).

5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt. Depending on the computer architecture and OS design, there may be a single program, one for each type of interrupt, or one for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack. However, there is other information that is considered part of the state of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. Other state information that must be saved is discussed in Chapter 3. Figure 1.11a shows a simple example. In this case, a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction ($N + 1$), a total of M words, are pushed onto the control stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.
7. The interrupt handler may now proceed to process the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 1.11b).
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

It is important to save all of the state information about the interrupted program for later resumption. This is because the interrupt is not a routine called from the program. Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program. Its occurrence is unpredictable.

ISBN 1-256-52023-3

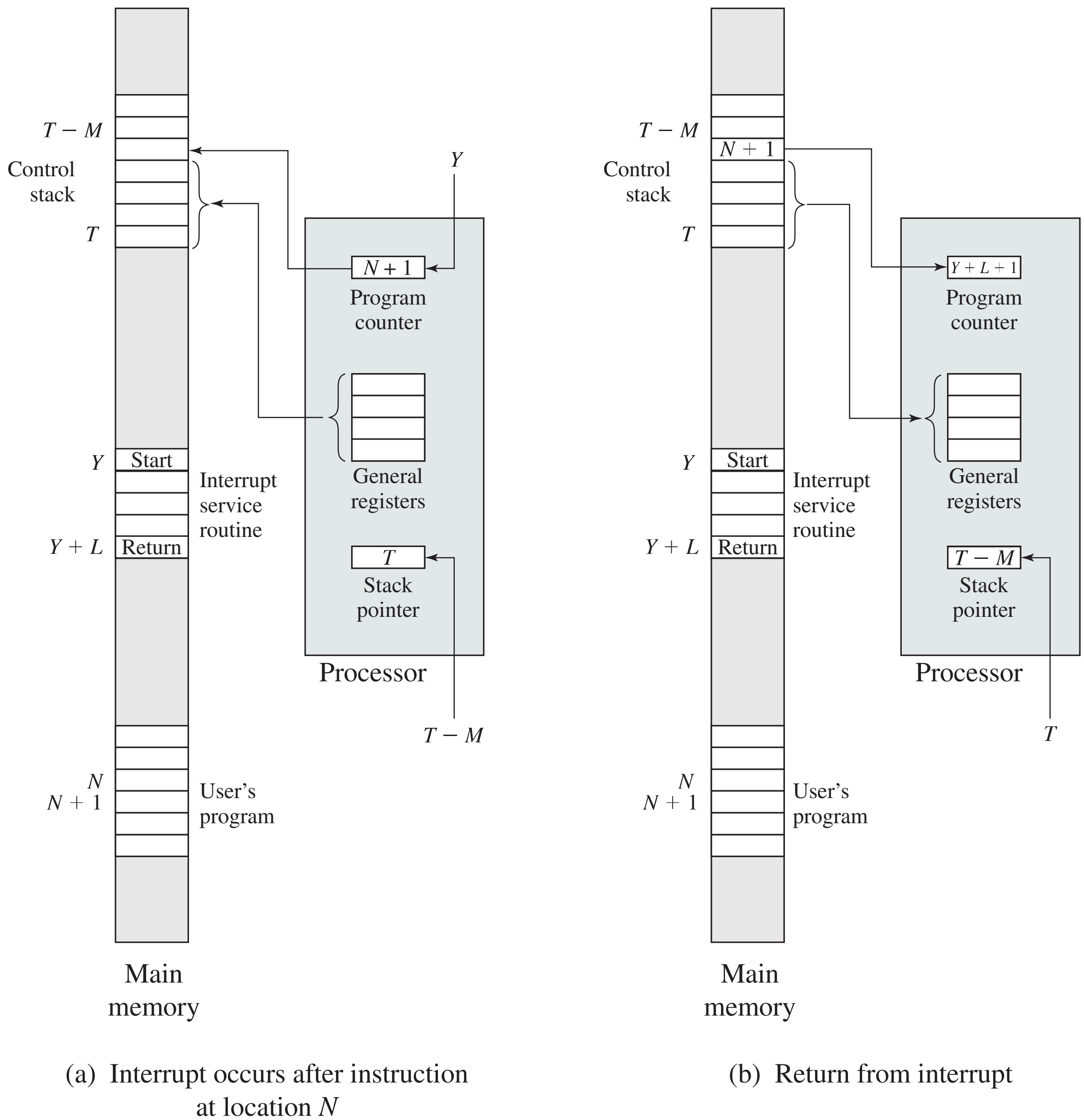


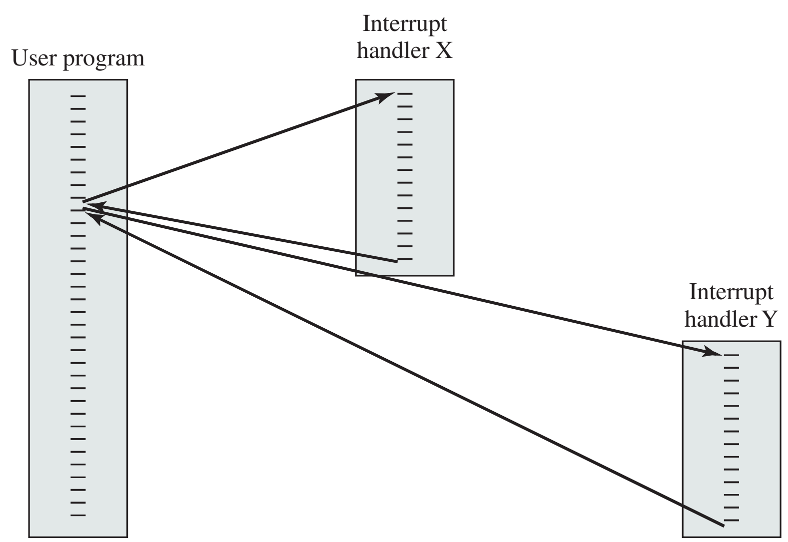
Figure 1.11 **Changes in Memory and Registers for an Interrupt**

Multiple Interrupts

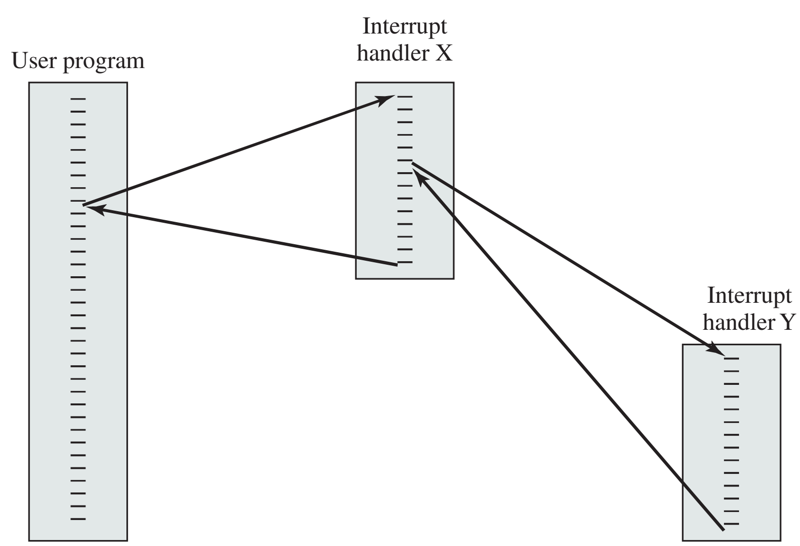
So far, we have discussed the occurrence of a single interrupt. Suppose, however, that one or more interrupts can occur while an interrupt is being processed. For example, a program may be receiving data from a communications line and printing results at the same time. The printer will generate an interrupt every time that 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.

ISBN 1-256-52023-3

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 ignores any new 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 reenabled interrupts. Thus, if an interrupt occurs when a user program is executing, then interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are reenabled before resuming the user program, and the processor checks to see if additional interrupts have occurred. This approach is simple, as interrupts are handled in strict sequential order (Figure 1.12a).



(a) Sequential interrupt processing



(b) Nested interrupt processing

Figure 1.12 Transfer of Control with Multiple Interrupts

ISBN 1-256-52023-3

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 because the buffer on the I/O device may fill and overflow.

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 interrupted (Figure 1.12b). 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 1.13, based on an example in [TANE06], illustrates a possible sequence. A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the control 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 request 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 transfers control 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.

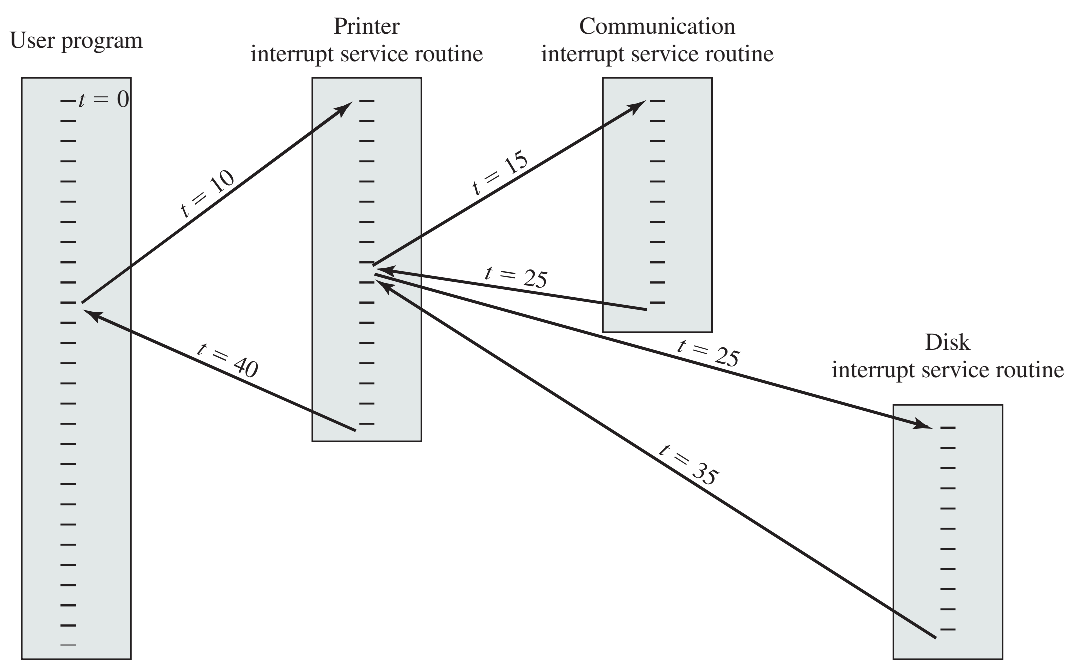


Figure 1.13 Example Time Sequence of Multiple Interrupts

ISBN 1-256-52023-3

1.5 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: namely, 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 speed

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 fast access times.

The way out of this dilemma is to not rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 1.14. 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 to the memory by the processor

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

Suppose that the processor has access to two levels of memory. Level 1 contains 1,000 bytes and has an access time of 0.1 μs; level 2 contains 100,000 bytes and has an access time of 1 μs. Assume that if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the byte is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the byte is in level 1 or level 2.

ISBN 1-256-52023-3

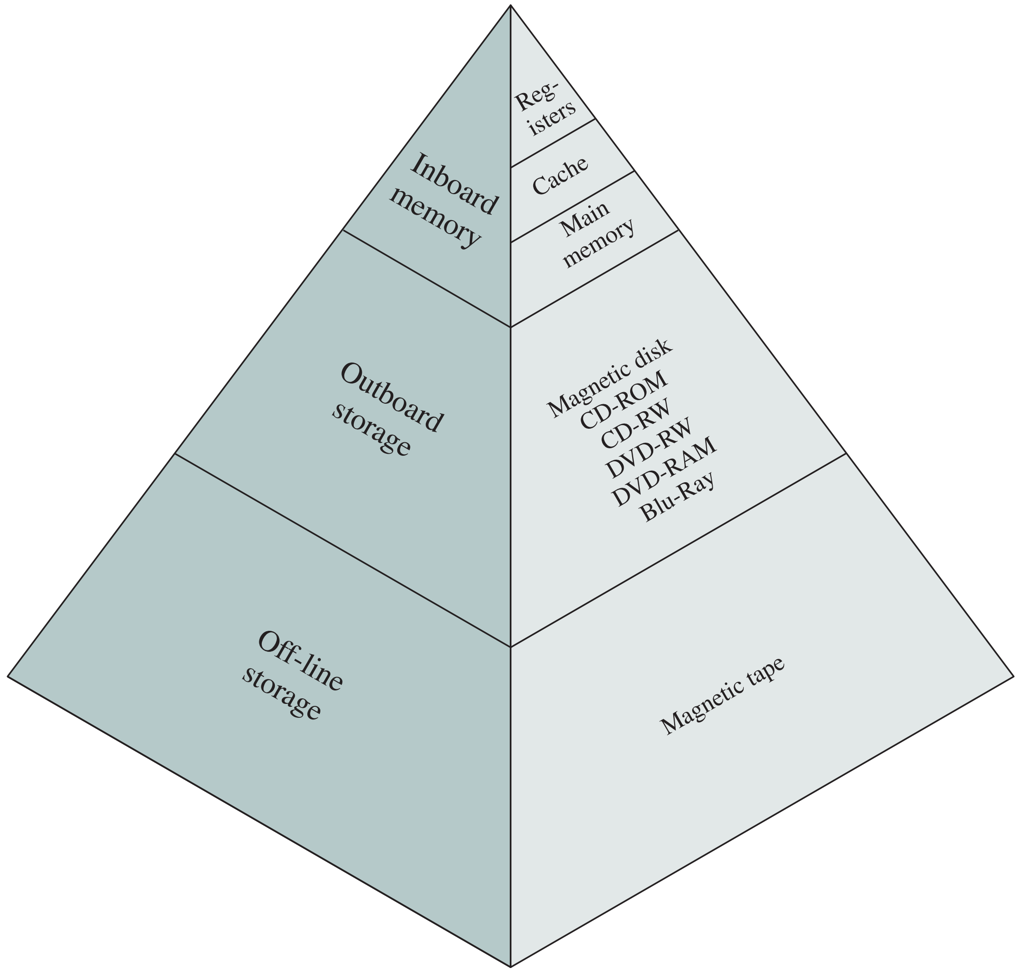


Figure 1.14 The Memory Hierarchy

Figure 1.15 shows the general shape of the curve that models this situation. The figure shows the average access time to a two-level memory as a function of the **hit ratio** H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[4] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in the cache (H = 0.95). Then the average time to access a byte can be expressed as

$$(0.95)\,(0.1\ \mu s) + (0.05)\,(0.1\ \mu s + 1\ \mu s) = 0.095 + 0.055 = 0.15\ \mu s$$

The result is close to the access time of the faster memory. So the strategy of using two memory levels works in principle, but only if conditions (a) through (d) in the preceding list apply. By employing a variety of technologies, a spectrum of

[4]If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

ISBN 1-256-52023-3

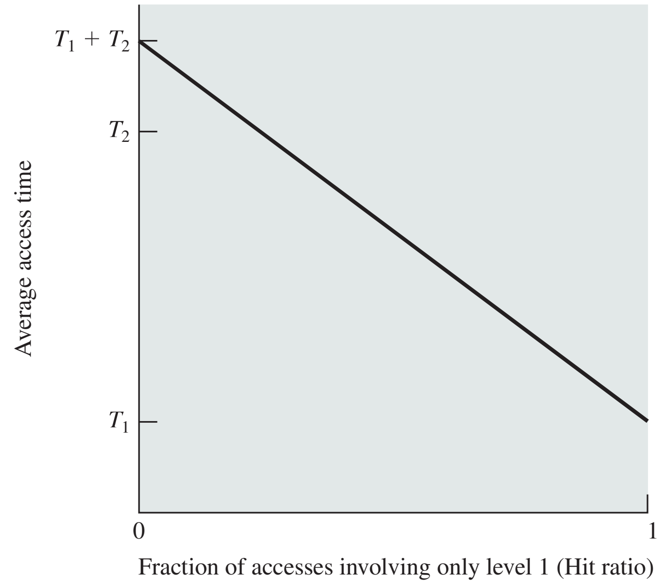


Figure 1.15 Performance of a Simple Two-Level Memory

memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference** [DENN68]. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data bytes. 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.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2 memory contain all program instructions and data. The current clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some processors contain hundreds of registers. Skipping down two levels, main memory is the principal internal memory system of the computer. Each location in main memory has a unique address, and most machine instructions refer to one or more main memory addresses. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

ISBN 1-256-52023-3

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files, and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. A hard disk is also used to provide an extension to main memory known as virtual memory, which is discussed in Chapter 8.

Additional levels can be effectively added to the hierarchy in software. For example, a portion of main memory can be used as a buffer to temporarily hold data that are to be read out to disk. Such a technique, sometimes referred to as a disk cache (examined in detail in Chapter 11), improves performance in two ways:

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimizes processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

Appendix 1A examines the performance implications of multilevel memory structures.

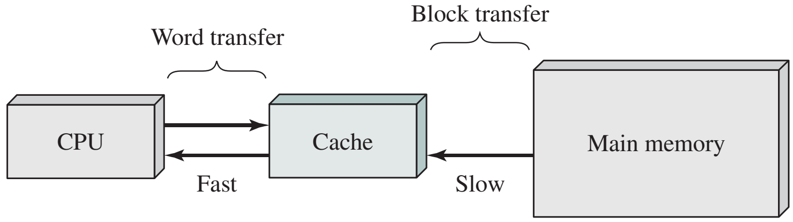
1.6 CACHE MEMORY

Although cache memory is invisible to the OS, it interacts with other memory management hardware. Furthermore, many of the principles used in virtual memory schemes (discussed in Chapter 8) are also applied in cache memory.

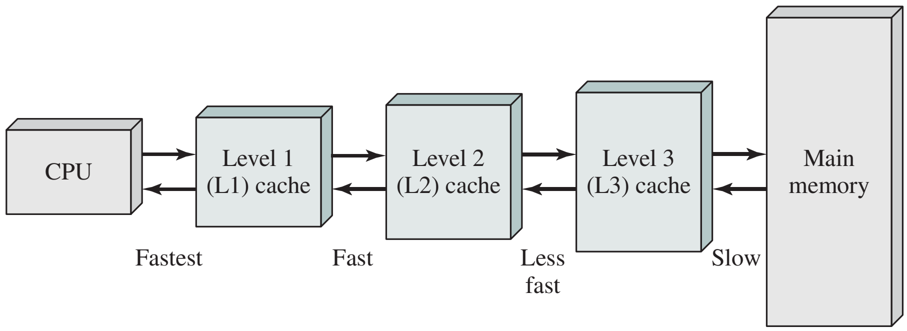
Motivation

On all instruction cycles, the processor accesses memory at least once, to fetch the instruction, and often one or more additional times, to fetch operands and/or store results. The rate at which the processor can execute instructions is clearly limited by the memory cycle time (the time it takes to read one word from or write one word to memory). This limitation has been a significant problem because of the persistent mismatch between processor and main memory speeds: Over the years, processor speed has consistently increased more rapidly than memory access speed. We are faced with a trade-off among speed, cost, and size. Ideally, main memory should be built with the same technology as that of the processor registers, giving memory cycle times comparable to processor cycle times. This has always been too expensive a strategy. The solution is to exploit the principle of locality by providing a small, fast memory between the processor and main memory, namely the cache.

ISBN 1-256-52023-3



(a) Single cache



(b) Three-level cache organization

Figure 1.16 Cache and Main Memory

Cache Principles

Cache memory is intended to provide memory access time approaching that of the fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories. The concept is illustrated in Figure 1.16a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of a portion of main memory. When the processor attempts to read a byte or word of memory, a check is made to determine if the byte or word is in the cache. If so, the byte or word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of bytes, is read into the cache and then the byte or 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 many of the near-future memory references will be to other bytes in the block.

Figure 1.16b 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.

Figure 1.17 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. Cache consists of C **slots** (also referred to as *lines*) of K words each, and the number of slots is considerably less than the number of main memory blocks ($C << M$).[5] Some subset of the blocks of main memory resides in the slots of the cache. If a word in a block

[5]The symbol << means *much less than.* Similarly, the symbol >> means *much greater than.*

ISBN 1-256-52023-3

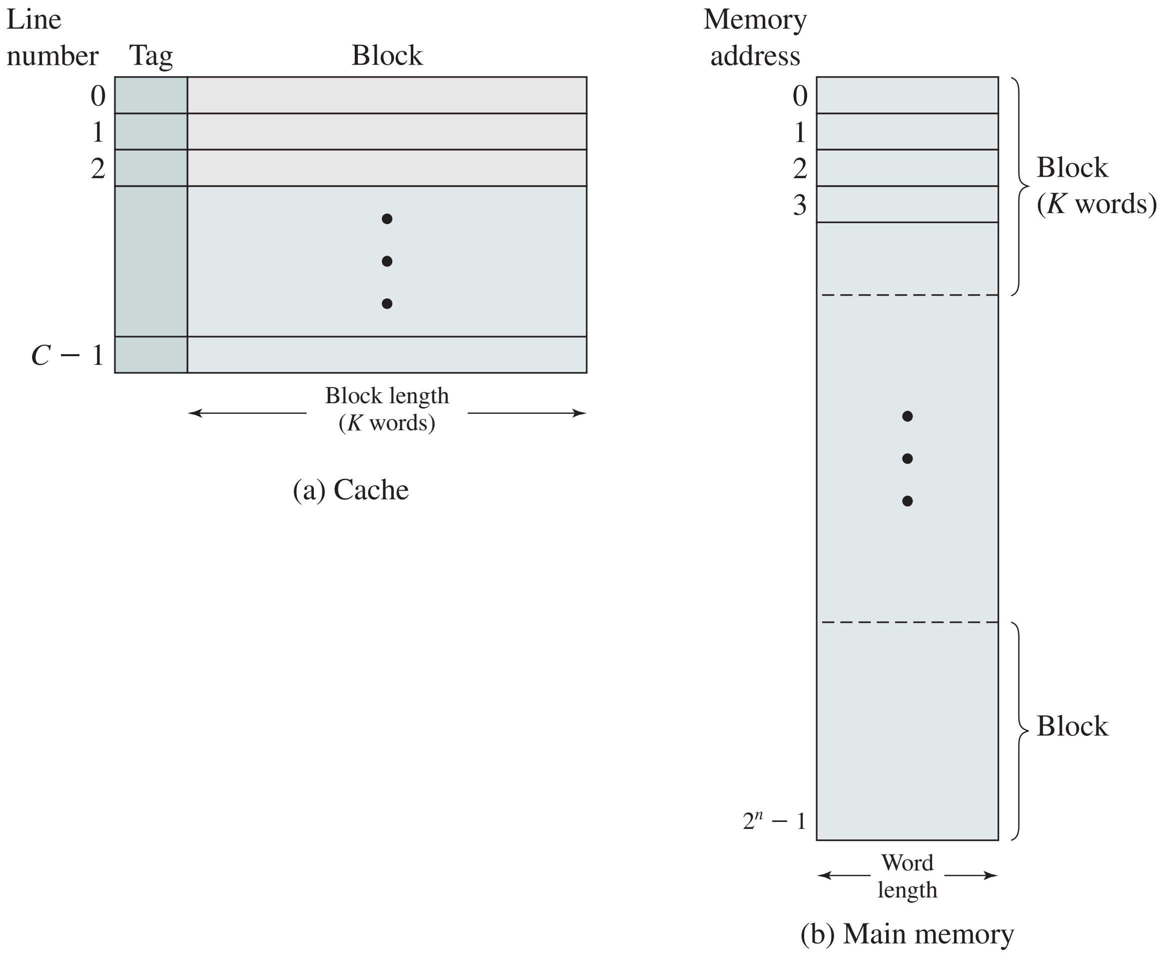


Figure 1.17 Cache/Main-Memory Structure

of memory that is not in the cache is read, that block is transferred to one of the slots of the cache. Because there are more blocks than slots, an individual slot cannot be uniquely and permanently dedicated to a particular block. Therefore, each slot includes a tag that identifies which particular block is currently being stored. The tag is usually some number of higher-order bits of the address and refers to all addresses that begin with that sequence of bits.

As a simple example, suppose that we have a 6-bit address and a 2-bit tag. The tag 01 refers to the block of locations with the following addresses: 010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110, 011111.

Figure 1.18 illustrates the read operation. The processor generates the 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.

Cache Design

A detailed discussion of cache design is beyond the scope of this book. Key elements are briefly summarized here. We will see that similar design issues must be

ISBN 1-256-52023-3

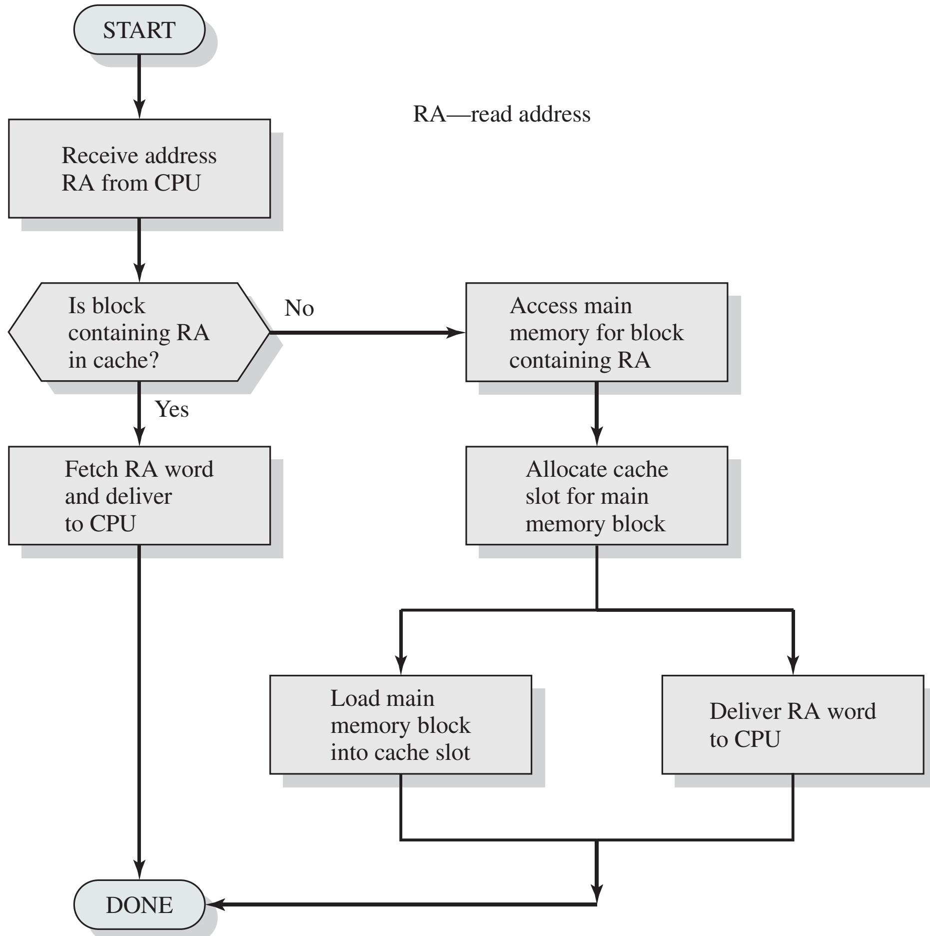


Figure 1.18 Cache Read Operation

addressed in dealing with virtual memory and disk cache design. They fall into the following categories:

- Cache size
- Block size
- Mapping function
- Replacement algorithm
- Write policy
- Number of cache levels

We have already dealt with the issue of **cache size**. It turns out that reasonably small caches can have a significant impact on performance. Another size issue is that of **block size**: the unit of data exchanged between cache and main memory. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality: the high probability that data in the vicinity of a referenced word are likely to be referenced in the near future. As the

ISBN 1-256-52023-3

block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched data becomes less than the probability of reusing the data that have to be moved out of the cache to make room for the new block.

When a new block of data is read into the cache, the **mapping function** determines which cache location the block will occupy. Two constraints affect the design of the mapping function. First, when one block is read in, another may have to be replaced. We would like to do this in such a way as to minimize the probability that we will replace a block that will be needed in the near future. The more flexible the mapping function, the more scope we have to design a replacement algorithm to maximize the hit ratio. Second, the more flexible the mapping function, the more complex is the circuitry required to search the cache to determine if a given block is in the cache.

The **replacement algorithm** chooses, within the constraints of the mapping function, which block to replace when a new block is to be loaded into the cache and the cache already has all slots filled with other blocks. We would like to replace the block that is least likely to be needed again in the near future. Although it is impossible to identify such a block, a reasonably effective strategy is to replace the block that has been in the cache longest with no reference to it. This policy is referred to as the least-recently-used (LRU) algorithm. Hardware mechanisms are needed to identify the least-recently-used block.

If the contents of a block in the cache are altered, then it is necessary to write it back to main memory before replacing it. The **write policy** dictates when the memory write operation takes place. At one extreme, the writing can occur every time that the block is updated. At the other extreme, the writing occurs only when the block is replaced. The latter policy minimizes memory write operations but leaves main memory in an obsolete state. This can interfere with multiple-processor operation and with direct memory access by I/O hardware modules.

Finally, it is now commonplace to have multiple levels of cache, labeled L1 (cache closest to the processor), L2, and in many cases a third level L3. A discussion of the performance benefits of multiple cache levels is beyond our scope; see [STAL10] for a discussion.

1.7 DIRECT MEMORY ACCESS

Three techniques are possible for I/O operations: programmed I/O, interrupt-driven I/O, and direct memory access (DMA). Before discussing DMA, we briefly define the other two techniques; see Appendix C for more detail.

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. In the case of **programmed I/O**, the I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed. For this purpose,

ISBN 1-256-52023-3

the processor periodically checks the status of the I/O module until it finds that the operation is complete.

With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the performance level of the entire system is severely degraded.

An alternative, known as **interrupt-driven I/O**, is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both of these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

When large volumes of data are to be moved, a more efficient technique is required: **direct memory access (DMA).** The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer.

The DMA module needs to take control of the bus to transfer data to and from memory. Because of this competition for bus usage, there may be times when the processor needs the bus and must wait for the DMA module. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle (the time it takes to transfer one word across the bus). The overall effect is to cause the processor to execute more slowly during a DMA transfer when processor access to the bus is required. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

ISBN 1-256-52023-3

1.8 MULTIPROCESSOR AND MULTICORE ORGANIZATION

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. A processor executes programs by executing machine instructions in sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel.

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability. In this book, we examine the three most popular approaches to providing parallelism by replicating processors: symmetric multiprocessors (SMPs), multicore computers, and clusters. SMPs and multicore computers are discussed in this section; clusters are examined in Chapter 16.

Symmetric Multiprocessors

Definition An SMP can be defined as a stand-alone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.
2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions (hence the term *symmetric*).
5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

Points 1 to 4 should be self-explanatory. Point 5 illustrates one of the contrasts with a loosely coupled multiprocessing system, such as a cluster. In the latter, the physical unit of interaction is usually a message or complete file. In an SMP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.

ISBN 1-256-52023-3

- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of tasks on individual processors and of synchronization among processors.

Organization Figure 1.19 illustrates the general organization of an SMP. There are multiple processors, each of which contains its own control unit, arithmetic-logic unit, and registers. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism; a shared bus is a common facility. The processors can communicate with each other through memory (messages and status information left in shared address spaces). It may

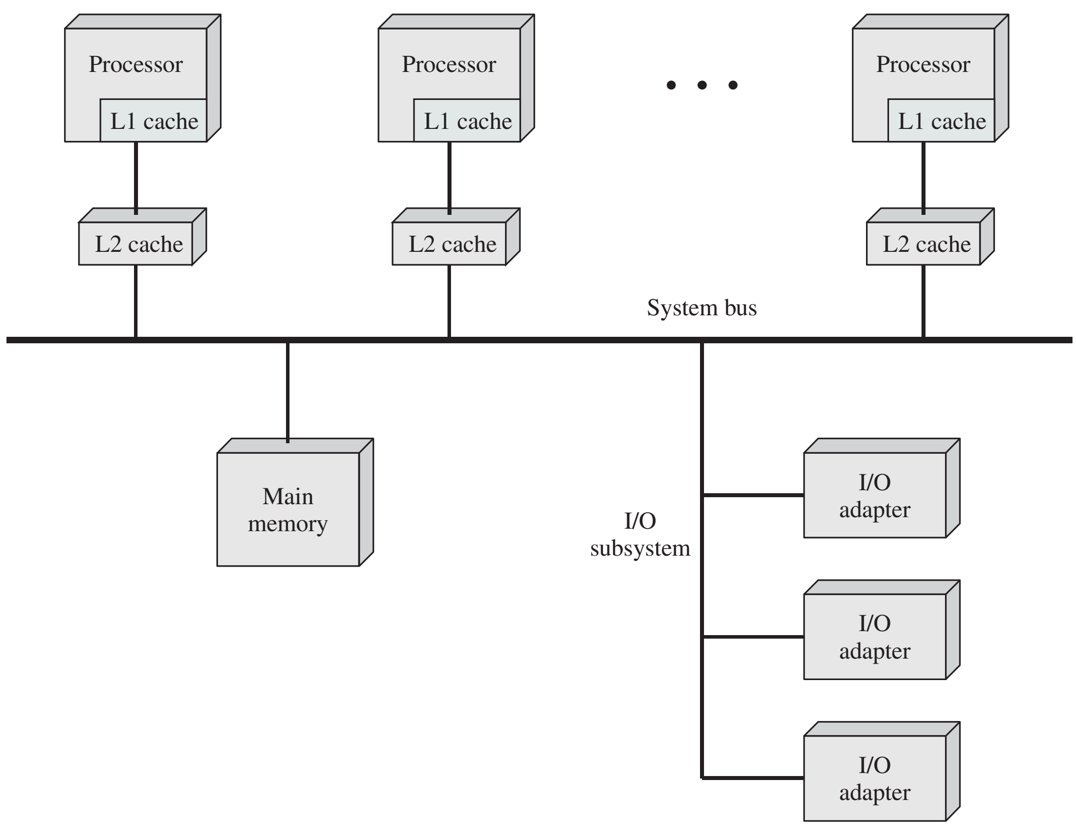


Figure 1.19 Symmetric Multiprocessor Organization

ISBN 1-256-52023-3

also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.

In modern computers, processors generally have at least one level of cache memory that is private to the processor. This use of cache introduces some new design considerations. Because each local cache contains an image of a portion of main memory, if a word is altered in one cache, it could conceivably invalidate a word in another cache. To prevent this, the other processors must be alerted that an update has taken place. This problem is known as the cache coherence problem and is typically addressed in hardware rather than by the OS.[6]

Multicore Computers

A **multicore** computer, also known as a **chip multiprocessor**, combines two or more processors (called cores) on a single piece of silicon (called a die). Typically, each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and, in some cases, L3 cache.

The motivation for the development of multicore computers can be summed up as follows. For decades, microprocessor systems have experienced a steady, usually exponential, increase in performance. This is partly due to hardware trends, such as an increase in clock frequency and the ability to put cache memory closer to the processor because of the increasing miniaturization of microcomputer components. Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access. In brief, designers have come up against practical limits in the ability to achieve greater performance by means of more complex processors. Designers have found that the best way to improve performance to take advantage of advances in hardware is to put multiple processors and a substantial amount of cache memory on a single chip. A detailed discussion of the rationale for this trend is beyond our scope, but is summarized in Appendix C.

An example of a multicore system is the Intel Core i7, which includes four x86 processors, each with a dedicated L2 cache, and with a shared L3 cache (Figure 1.20). One mechanism Intel uses to make its caches more effective is prefetching, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that's likely to be requested soon.

The Core i7 chip supports two forms of external communications to other chips. The **DDR3 memory controller** brings the memory controller for the DDR (double data rate) main memory onto the chip. The interface supports three channels that are 8 bytes wide for a total bus width of 192 bits, for an aggregate data rate of up to 32 GB/s. With the memory controller on the chip, the Front Side Bus is eliminated. The **QuickPath Interconnect (QPI)** is a point-to-point link electrical interconnect specification. It enables high-speed communications among connected processor chips. The QPI link operates at 6.4 GT/s (transfers per second).

[6] A description of hardware-based cache coherency schemes is provided in [STAL10].

ISBN 1-256-52023-3

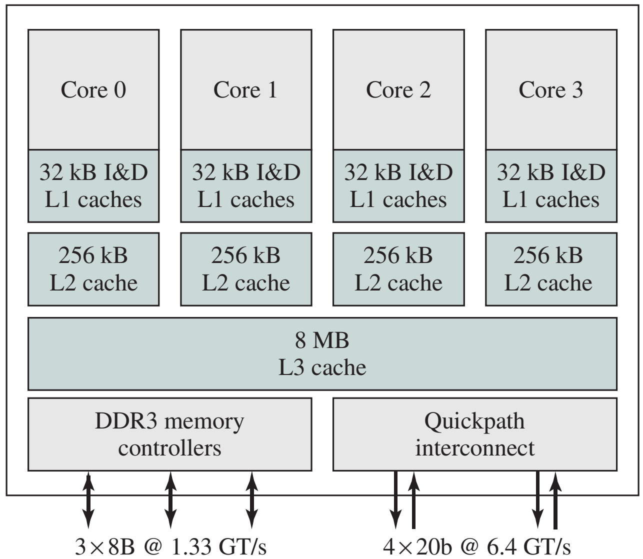


Figure 1.20 Intel Core i7 Block Diagram

At 16 bits per transfer, that adds up to 12.8 GB/s; and since QPI links involve dedicated bidirectional pairs, the total bandwidth is 25.6 GB/s.

1.9 RECOMMENDED READING AND WEB SITES

[STAL10] covers the topics of this chapter in detail. In addition, there are many other texts on computer organization and architecture. Among the more worthwhile texts are the following. [PATT09] is a comprehensive survey; [HENN07], by the same authors, is a more advanced text that emphasizes quantitative aspects of design.

[DENN05] looks at the history of the development and application of the locality principle, making for fascinating reading.

DENN05 Denning, P. "The Locality Principle." *Communications of the ACM*, July 2005.

HENN07 Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 2007.

PATT09 Patterson, D., and Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, CA: Morgan Kaufmann, 2009.

STAL10 Stallings, W. *Computer Organization and Architecture, 8th ed.* Upper Saddle River, NJ: Prentice Hall, 2010.

Recommended Web sites:

- **WWW Computer Architecture Home Page:** A comprehensive index to information relevant to computer architecture researchers, including architecture groups and projects, technical organizations, literature, employment, and commercial information
- **CPU Info Center:** Information on specific processors, including technical papers, product information, and latest announcements

ISBN 1-256-52023-3

1.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

address register	instruction register	program counter
cache memory	interrupt	programmed I/O
cache slot	interrupt-driven I/O	reentrant procedure
central processing unit	I/O module	register
data register	locality	secondary memory
direct memory access	main memory	spatial locality
hit ratio	multicore	stack
input/output	multiprocessor	system bus
instruction	processor	temporal locality
instruction cycle		

Review Questions

1.1. List and briefly define the four main elements of a computer.

1.2. Define the two main categories of processor registers.

1.3. In general terms, what are the four distinct actions that a machine instruction can specify?

1.4. What is an interrupt?

1.5. How are multiple interrupts dealt with?

1.6. What characteristics distinguish the various elements of a memory hierarchy?

1.7. What is cache memory?

1.8. What is the difference between a multiprocessor and a multicore system?

1.9. What is the distinction between spatial locality and temporal locality?

1.10. In general, what are the strategies for exploiting spatial locality and temporal locality?

Problems

1.1. Suppose the hypothetical processor of Figure 1.3 also has two I/O instructions:

0011 = Load AC from I/O

0111 = Store AC to I/O

In these cases, the 12-bit address identifies a particular external device. Show the program execution (using format of Figure 1.4) for the following program:

1. Load AC from device 5.
2. Add contents of memory location 940.
3. Store AC to device 6.

Assume that the next value retrieved from device 5 is 3 and that location 940 contains a value of 2.

1.2. The program execution of Figure 1.4 is described in the text using six steps. Expand this description to show the use of the MAR and MBR.

1.3. Consider a hypothetical 32-bit microprocessor having 32-bit instructions composed of two fields. The first byte contains the opcode and the remainder an immediate operand or an operand address.

ISBN 1-256-52023-3

- **a.** What is the maximum directly addressable memory capacity (in bytes)?
- **b.** Discuss the impact on the system speed if the microprocessor bus has
  - **1.** a 32-bit local address bus and a 16-bit local data bus, or
  - **2.** a 16-bit local address bus and a 16-bit local data bus.
- **c.** How many bits are needed for the program counter and the instruction register?

1.4. Consider a hypothetical microprocessor generating a 16-bit address (e.g., assume that the program counter and the address registers are 16 bits wide) and having a 16-bit data bus.

- **a.** What is the maximum memory address space that the processor can access directly if it is connected to a "16-bit memory"?
- **b.** What is the maximum memory address space that the processor can access directly if it is connected to an "8-bit memory"?
- **c.** What architectural features will allow this microprocessor to access a separate "I/O space"?
- **d.** If an input and an output instruction can specify an 8-bit I/O port number, how many 8-bit I/O ports can the microprocessor support? How many 16-bit I/O ports? Explain.

1.5. Consider a 32-bit microprocessor, with a 16-bit external data bus, driven by an 8-MHz input clock. Assume that this microprocessor has a bus cycle whose minimum duration equals four input clock cycles. What is the maximum data transfer rate across the bus that this microprocessor can sustain in bytes/s? To increase its performance, would it be better to make its external data bus 32 bits or to double the external clock frequency supplied to the microprocessor? State any other assumptions you make and explain. *Hint:* Determine the number of bytes that can be transferred per bus cycle.

1.6. Consider a computer system that contains an I/O module controlling a simple keyboard/printer Teletype. The following registers are contained in the CPU and connected directly to the system bus:

INPR: Input Register, 8 bits
OUTR: Output Register, 8 bits
FGI: Input Flag, 1 bit
FGO: Output Flag, 1 bit
IEN: Interrupt Enable, 1 bit

Keystroke input from the Teletype and output to the printer are controlled by the I/O module. The Teletype is able to encode an alphanumeric symbol to an 8-bit word and decode an 8-bit word into an alphanumeric symbol. The Input flag is set when an 8-bit word enters the input register from the Teletype. The Output flag is set when a word is printed.

- **a.** Describe how the CPU, using the first four registers listed in this problem, can achieve I/O with the Teletype.
- **b.** Describe how the function can be performed more efficiently by also employing IEN.

1.7. In virtually all systems that include DMA modules, DMA access to main memory is given higher priority than processor access to main memory. Why?

1.8. A DMA module is transferring characters to main memory from an external device transmitting at 9600 bits per second (bps). The processor can fetch instructions at the rate of 1 million instructions per second. By how much will the processor be slowed down due to the DMA activity?

1.9. A computer consists of a CPU and an I/O device D connected to main memory M via a shared bus with a data bus width of one word. The CPU can execute a maximum of 106 instructions per second. An average instruction requires five processor cycles, three of which use the memory bus. A memory read or write operation uses one processor cycle. Suppose that the CPU is continuously executing "background" programs that require 95% of its instruction execution rate but not any I/O instructions.

ISBN 1-256-52023-3

Assume that one processor cycle equals one bus cycle. Now suppose that very large blocks of data are to be transferred between *M* and *D*.

a. If programmed I/O is used and each one-word I/O transfer requires the CPU to execute two instructions, estimate the maximum I/O data transfer rate, in words per second, possible through *D*.

b. Estimate the same rate if DMA transfer is used.

1.10. Consider the following code:

```
for (i = 0; i < 20; i++)
    for (j = 0; j < 10; j++)
        a[i] = a[i] * j
```

a. Give one example of the spatial locality in the code.

b. Give one example of the temporal locality in the code.

1.11. Generalize Equations (1.1) and (1.2) in Appendix 1A to *n*-level memory hierarchies.

1.12. Consider a memory system with the following parameters:

$$Tc = 100 \text{ ns} \quad Cc = 0.01 \text{ cents/bit}$$
$$Tm = 1{,}200 \text{ ns} \quad Cm = 0.001 \text{ cents/bit}$$

a. What is the cost of 1 MByte of main memory?

b. What is the cost of 1 MByte of main memory using cache memory technology?

c. If the effective access time is 10% greater than the cache access time, what is the hit ratio *H*?

1.13. A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache (this includes the time to originally check the cache), and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main-memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?

1.14. Suppose a stack is to be used by the processor to manage procedure calls and returns. Can the program counter be eliminated by using the top of the stack as a program counter?

APPENDIX 1A PERFORMANCE CHARACTERISTICS OF TWO-LEVEL MEMORIES

In this chapter, reference is made to a cache that acts as a buffer between main memory and processor, creating a two-level internal memory. This two-level architecture exploits a property known as locality to provide improved performance over a comparable one-level memory.

The main memory cache mechanism is part of the computer architecture, implemented in hardware and typically invisible to the OS. Accordingly, this mechanism is not pursued in this book. However, there are two other instances of a two-level memory approach that also exploit the property of locality and that are, at least partially, implemented in the OS: virtual memory and the disk cache (Table 1.2). These two topics are explored in Chapters 8 and 11, respectively. In this appendix, we look at some of the performance characteristics of two-level memories that are common to all three approaches.

ISBN 1-256-52023-3

Table 1.2 Characteristics of Two-Level Memories

	Main Memory Cache	Virtual Memory (Paging)	Disk Cache
Typical access time ratios	5 : 1	10^6: 1	10^6: 1
Memory management system	Implemented by special hardware	Combination of hardware and system software	System software
Typical block size	4 to 128 bytes	64 to 4096 bytes	64 to 4096 bytes
Access of processor to second level	Direct access	Indirect access	Indirect access

Locality

The basis for the performance advantage of a two-level memory is the principle of locality, referred to in Section 1.5. 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.

This line of reasoning has been confirmed in many studies. With reference to point (1), a variety of studies have analyzed the behavior of high-level language programs. Table 1.3 includes key results, measuring the appearance of various statement types during execution, from the following studies. The earliest study of programming language behavior, performed by Knuth [KNUT71], examined a collection of FORTRAN programs used as student exercises. Tanenbaum [TANE78] published measurements collected from over 300 procedures used in OS programs and written in a language that supports structured programming (SAL). Patterson and Sequin [PATT82] analyzed a set of measurements taken from compilers and programs for typesetting, computer-aided design (CAD), sorting, and file

ISBN 1-256-52023-3

Table 1.3 Relative Dynamic Frequency of High-Level Language Operations

Study Language Workload	[HUCK83] Pascal Scientific	[KNUT71] FORTRAN Student	[PATT82] Pascal System	[PATT82] C System	[TANE78] SAL System
Assign	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Other	—	7	6	1	6

comparison. The programming languages C and Pascal were studied. Huck [HUCK83] analyzed four programs intended to represent a mix of general-purpose scientific computing, including fast Fourier transform and the integration of systems of differential equations. There is good agreement in the results of this mixture of languages and applications that branching and call instructions represent only a fraction of statements executed during the lifetime of a program. Thus, these studies confirm assertion (1), from the preceding list.

With respect to assertion (2), studies reported in [PATT85] provide confirmation. This is illustrated in Figure 1.21, which shows call-return behavior. Each call is represented by the line moving down and to the right, and each return by the line moving up and to the right. In the figure, a *window* with depth equal to 5 is defined. Only a sequence of calls and returns with a net movement of 6 in either direction causes the window to move. As can be seen, the executing program can remain within a stationary window for long periods of time. A study by the same analysts of C and Pascal programs showed that a window of depth 8 would only need to shift on less than 1% of the calls or returns [TAMI83].

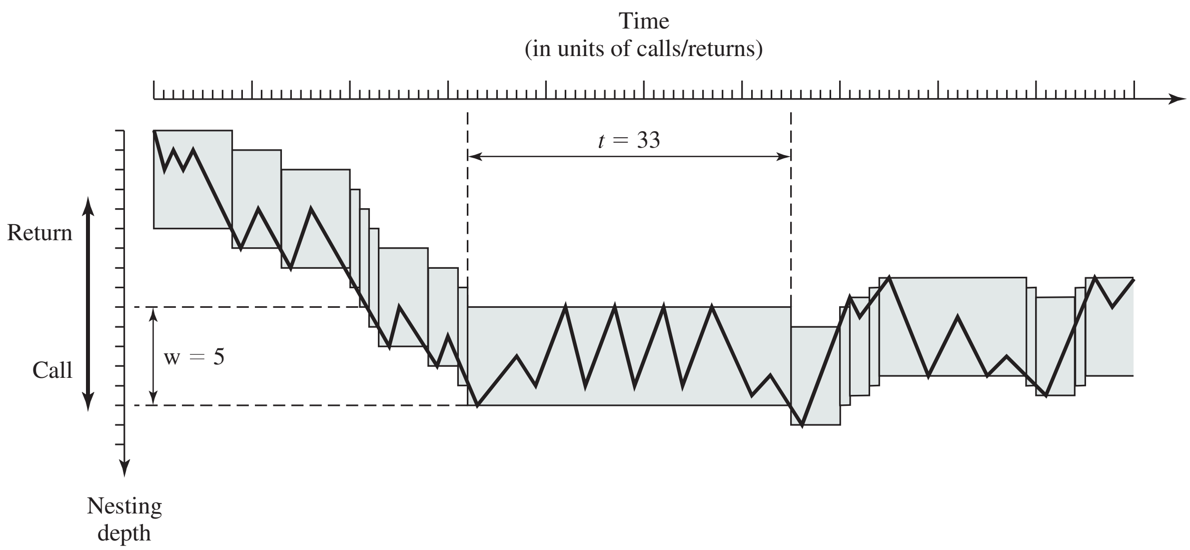


Figure 1.21 **Example Call-Return Behavior of a Program**

ISBN 1-256-52023-3

A distinction is made in the literature between spatial locality and temporal locality. **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. **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.

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 whose use is expected) 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.

Operation of Two-Level Memory

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory but also the probability that a given reference can be found in M1. We have

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ & T_1 + (1 - H) \times T_2 \end{aligned} \qquad \textbf{(1.1)}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.g., cache, disk cache)
T_2 = access time of M2 (e.g., main memory, disk)
H = hit ratio (fraction of time reference is found in M1)

Figure 1.15 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

Performance

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \qquad \textbf{(1.2)}$$

ISBN 1-256-52023-3

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 >> C_2$, this requires $S_1 << S_2$. Figure 1.22 shows the relationship.[7]

Next, consider access time. For a two-level memory to provide a significant performance improvement, we need to have T_s approximately equal to T_1 $T_s \approx T_1$. Given that T_1 is much less than T_2 $T_s >> T_1$, a hit ratio of close to 1 is needed.

So we would like M1 to be small to hold down cost, and large to improve the hit ratio and therefore the performance. Is there a size of M1 that satisfies both requirements to a reasonable extent? We can answer this question with a series of subquestions:

- What value of hit ratio is needed to satisfy the performance requirement?
- What size of M1 will assure the needed hit ratio?
- Does this size satisfy the cost requirement?

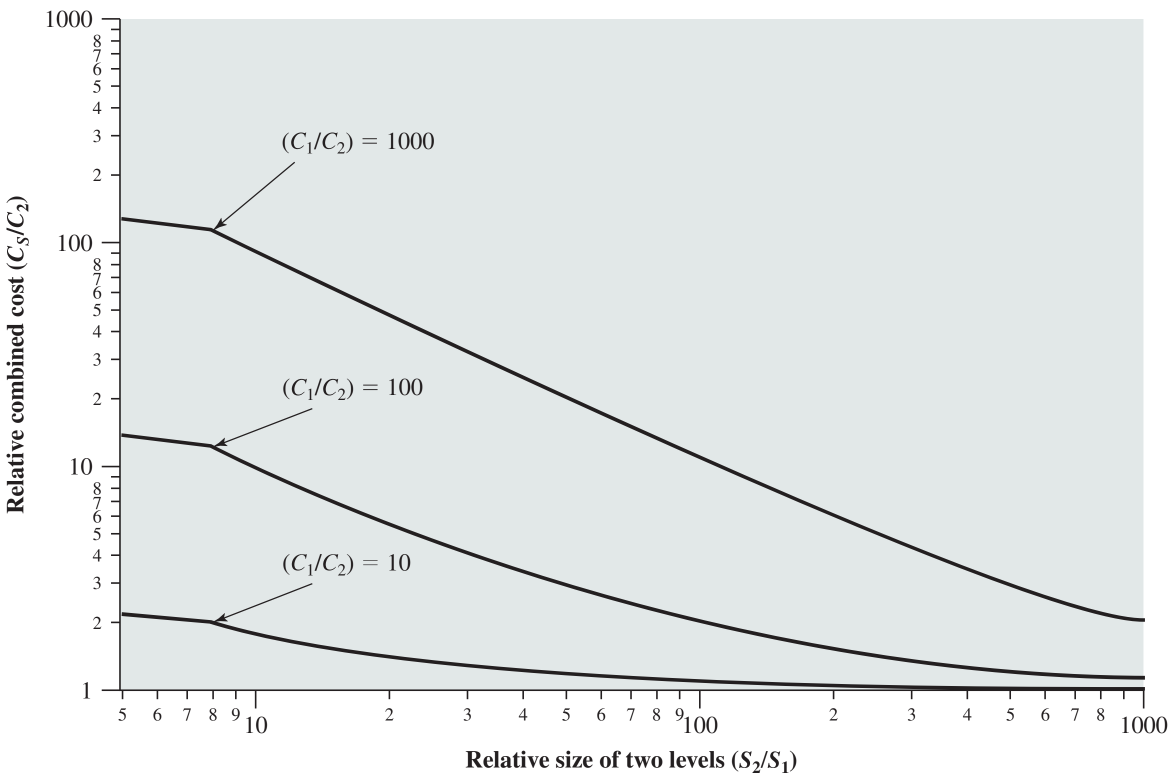


Figure 1.22 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

[7]Note that both axes use a log scale. A basic review of log scales is in the math refresher document at the Computer Science Student Resource Site at ComputerScienceStudent.com.

ISBN 1-256-52023-3

To get at this, consider the quantity T_1/T_s, which is referred to as the *access efficiency*. It is a measure of how close average access time (T_s) is to M1 access time (T_1). From Equation (1.1),

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\dfrac{T_2}{T_1}} \qquad \textbf{(1.3)}$$

In Figure 1.23, we plot T_1/T_s as a function of the hit ratio H, with the quantity T_2/T_1 as a parameter. A hit ratio in the range of 0.8 to 0.9 would seem to be needed to satisfy the performance requirement.

We can now phrase the question about relative memory size more exactly. Is a hit ratio of 0.8 or higher reasonable for $S_1 << S_2$? This will depend on a number of factors, including the nature of the software being executed and the details of the design of the two-level memory. The main determinant is, of course, the degree of locality. Figure 1.24 suggests the effect of locality on the hit ratio. Clearly, if M1 is the same size as M2, then the hit ratio will be 1.0: All of the items in M2 are always stored also in M1. Now suppose that there is no locality; that is, references are completely random. In that case the hit ratio should be a strictly linear function of the relative memory size. For example, if M1 is half the size of M2, then at any time half of the items from M2 are also in M1 and the hit ratio will be 0.5. In practice, however, there is some degree of locality in the references. The effects of moderate and strong locality are indicated in the figure.

So, if there is strong locality, it is possible to achieve high values of hit ratio even with relatively small upper-level memory size. For example, numerous studies

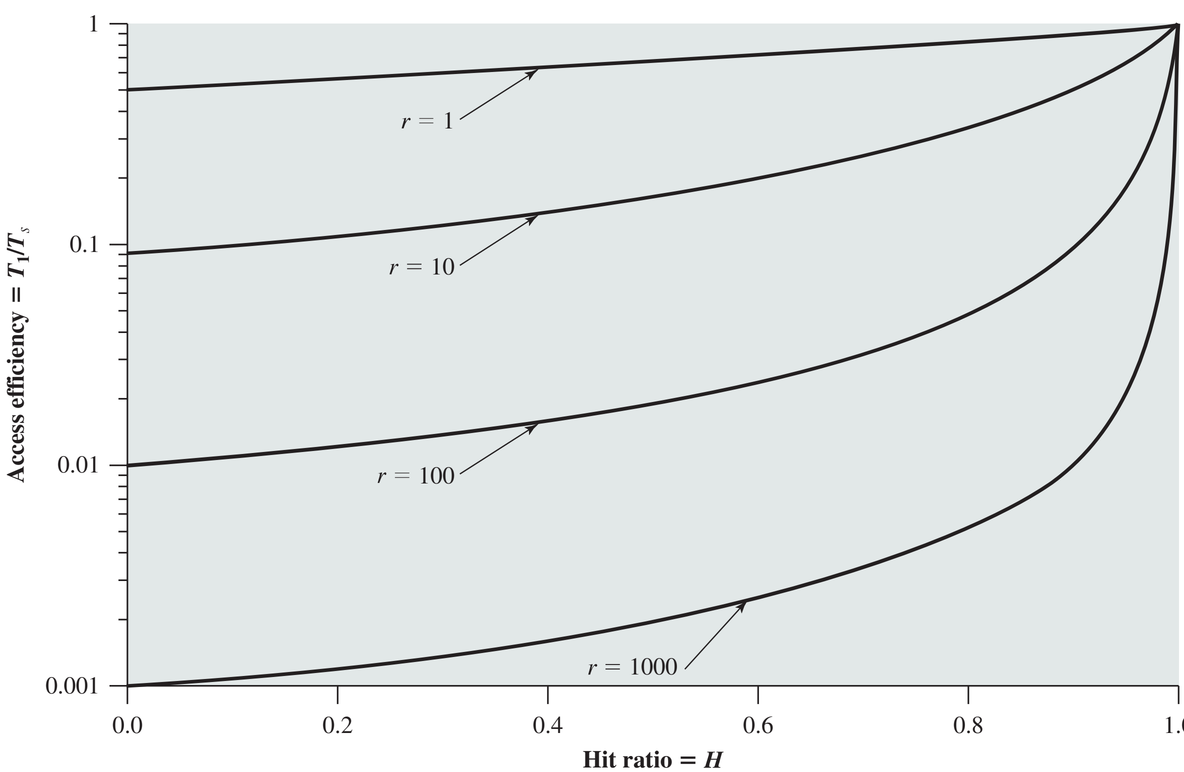


Figure 1.23 Access Efficiency as a Function of Hit Ratio ($r = T_2/T_1$)

ISBN 1-256-52023-3

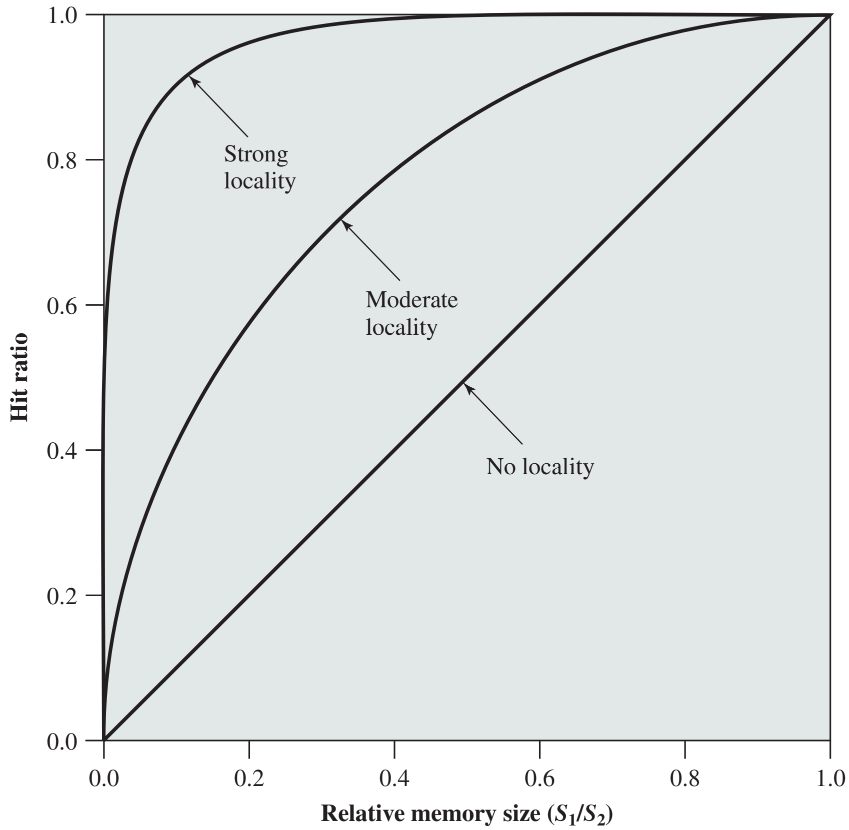


Figure 1.24 Hit Ratio as a Function of Relative Memory Size

have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* (e.g., [AGAR89], [PRZY88], [STRE83], and [SMIT82]). A cache in the range of 1K to 128K words is generally adequate, whereas main memory is now typically in the gigabyte range. When we consider virtual memory and disk cache, we will cite other studies that confirm the same phenomenon, namely that a relatively small M1 yields a high value of hit ratio because of locality.

This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

ISBN 1-256-52023-3