the actual data from the disk. The control lines will be used to indicate what type of information is contained on the data lines of the bus at each point in the transfer. Some buses have two sets of signal lines to separately communicate both data and address in a single bus transmission. In either case, the control lines are used to indicate what the bus contains and to implement the bus protocol. And because the bus is shared, we also need a protocol to decide who uses it next; we will discuss this problem shortly.

Let's consider a typical *bus transaction*. A bus transaction includes two parts: sending the address and receiving or sending the data. Bus transactions are typically defined by what they do to memory. A *read* transaction transfers data *from* memory (to either the processor or an I/O device), and a *write* transaction writes data *to* the memory. Clearly, this terminology is confusing. To avoid this, we'll try to use the terms *input* and *output*, which are always defined from the perspective of the processor: an input operation is inputting data from the device to memory, where the processor can read it, and an output operation is outputting data to a device from memory where the processor wrote it. Figure 8.7 shows the steps in a typical output operation, in which data will be read from memory and sent to the device. Figure 8.8 shows the steps in an input operation where data is read from the device and written to memory. In both figures, the active portions of the bus and memory are shown in color, and a read or write is shown by shading the unit, as we did in Chapter 6. In these figures, we focus on how data is transferred between the I/O device and memory; in section 8.5, we will see how the I/O operation is initiated.

Types of Buses

Buses are traditionally classified as one of three types: *processor-memory buses*, *I/O buses*, or *backplane buses*. Processor-memory buses are short, generally high speed, and matched to the memory system so as to maximize memory-processor bandwidth. I/O buses, by contrast, can be lengthy, can have many types of devices connected to them, and often have a wide range in the data bandwidth of the devices connected to them. I/O buses do not typically interface directly to the memory but use either a processor-memory or a backplane bus to connect to memory. Backplane buses are designed to allow processors, memory, and I/O devices to coexist on a single bus; they balance the demands of processor-memory communication with the demands of I/O device-memory communication. Backplane buses received their name because they were often built into the *backplane*, an interconnection structure within the chassis; processor, memory, and I/O boards would then plug into the backplane using the bus for communication.

Processor-memory buses are often design-specific, while both I/O buses and backplane buses are frequently reused in different machines. In fact, backplane and I/O buses are often *standard buses* that are used by many different

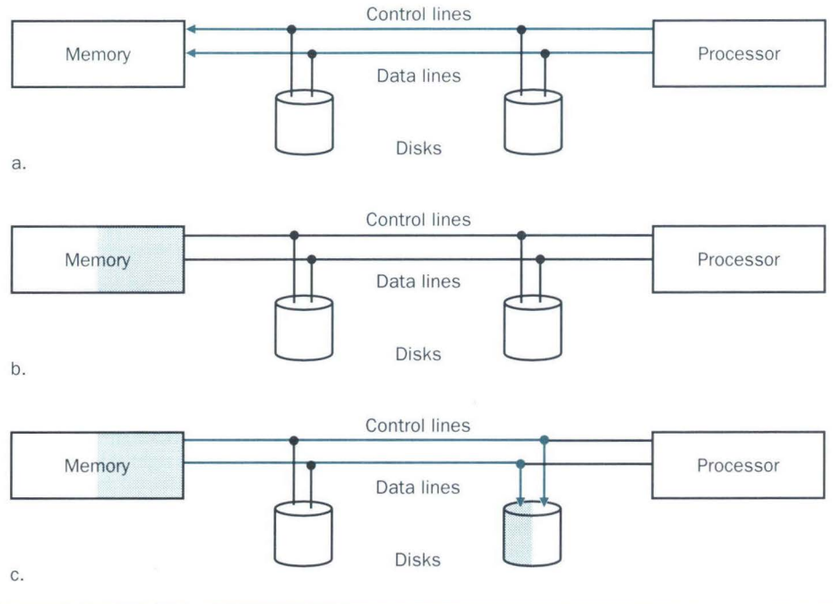


FIGURE 8.7 The three steps of an output operation. In each step, the active participants in the communication are shown in color, with the right side shaded if the device is doing a read and the left side shaded if the device is doing a write. Notice that the data lines of the bus can carry both an address (as in a) and data (as in c). (a) The first step in an output operation initiates a read from memory. The control lines signal a read request to memory, while the data lines contain the address. (b) During the second step in an output operation, memory is accessing the data. (c) In the third and final step in an output operation, memory transfers the data using the data lines of the bus and signals that the data is available to the I/O device using the control lines. The device stores the data as it appears on the bus.

computers manufactured by different companies. By comparison, processor-memory buses are often proprietary, although in many recent machines they may be the backplane bus, and the standard or I/O buses plug into the processor-memory bus. In many recent machines, the distinction among these bus types, especially between backplane buses and processor-memory buses, may be very minor.

During the design phase, the designer of a processor-memory bus knows all the types of devices that must connect to the bus, while the I/O or backplane bus designer must design the bus to handle unknown devices that vary in latency and bandwidth characteristics. Normally, an I/O bus presents a fairly simple and low-level interface to a device, requiring minimal additional electronics to interface to the bus. A backplane bus usually requires additional logic to interface between the bus and a device or between the backplane bus

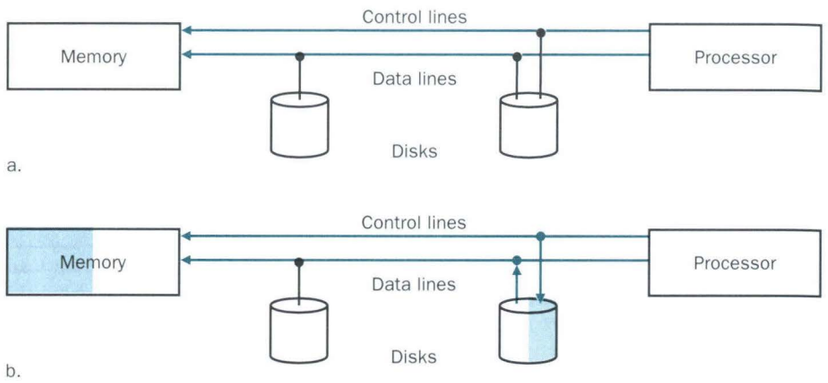


FIGURE 8.8 An input operation takes less active time because the device does not need to wait for memory to access data. As in the previous figure, the active participants in each step in the communication are shown in color, with the right side shaded if the device is doing a read and the left side shaded if the device is doing a write. (a) In the first step in an input operation, the control lines indicate a write request for memory, while the data lines contain the address. (b) The second step in an input operation occurs when the memory is ready and signals the device, which then transfers the data. Typically, the memory will store the data as it receives it. The device need not wait for the store to be completed. In the steps shown, we assume that the device had to wait for memory to indicate its readiness, but this will not be true in some systems that use buffering or have a fast memory system.

and a lower-level I/O bus. A backplane bus offers the cost advantage of a single bus. Figure 8.9 shows a system using a single backplane bus, a system using a processor-memory bus with attached I/O buses, and a system using all three types of buses. Machines with a separate processor-memory bus normally use a bus adapter to connect the I/O bus to the processor-memory bus. Some high-performance, expandable systems use an organization that combines the three buses: the processor-memory bus has one or more bus adapters that interface a standard backplane bus to the processor-memory bus. I/O buses, as well as device controllers, can plug into the backplane bus. The IBM RS/6000 and Silicon Graphics multiprocessors use this type of organization. This organization offers the advantage that the processor-memory bus can be made much faster than a backplane or I/O bus and that the I/O system can be expanded by plugging many I/O controllers or buses into the backplane bus, which will not affect the speed of the processor-memory bus.

Synchronous and Asynchronous Buses

The substantial differences between the circumstances under which a processor-memory bus and an I/O bus or backplane bus are designed lead to two different schemes for communication on the bus: *synchronous* and

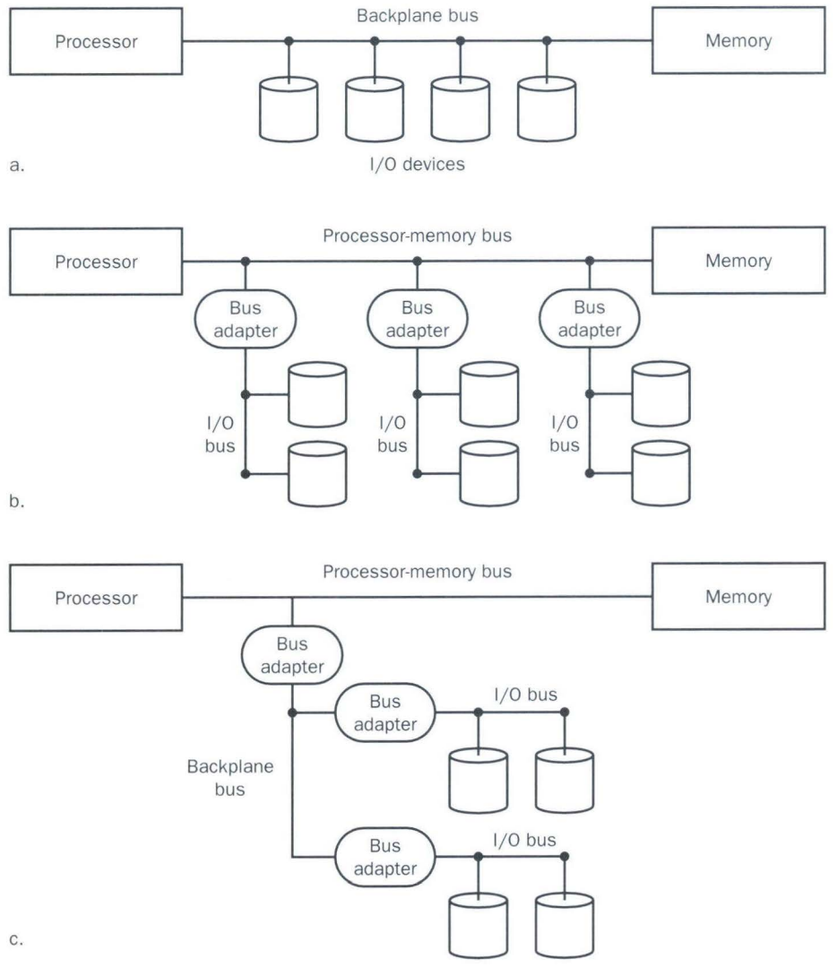


FIGURE 8.9 Many machines use a single backplane bus for both processor-memory and I/O traffic. Some high-performance machines use a separate processor-memory bus that I/O buses plug into. Some systems make use of all three types of buses, organized in a hierarchy. (a) A single bus used for processor-to-memory communication, as well as communication between I/O devices and memory. The bus used in older PCs has this structure. (b) A separate bus is used for processor–memory traffic. To communicate data between memory and I/O devices, the I/O buses interface to the processor-memory bus, using a bus adapter. The bus adapter provides speed matching between the buses. In many recent PCs, the processor-memory bus is a PCI bus (a backplane bus) that has I/O devices that interface directly as well as an I/O bus that plugs into the PCI bus; the latter is a SCSI bus. (c) A separate bus is used for processor-memory traffic. A small number of backplane buses tap into the processor-memory bus. The processor-memory buses interface to the backplane bus. This is usually done with a single-chip controller, such as a SCSI bus controller. An advantage of this organization is the small number of taps into the high-speed processor-memory bus.

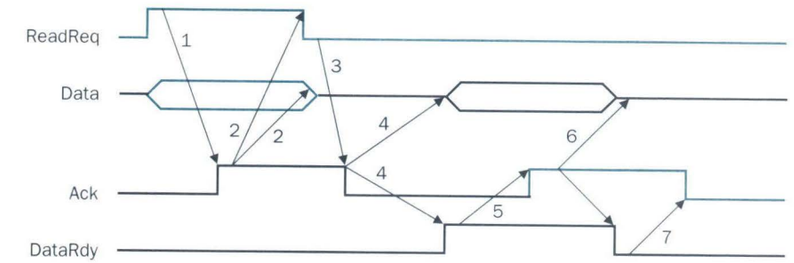
asynchronous. If a bus is synchronous, it includes a clock in the control lines and a fixed protocol for communicating that is relative to the clock. For example, for a processor-memory bus performing a read from memory, we might have a protocol that transmits the address and read command on the first clock cycle, using the control lines to indicate the type of request. The memory might then be required to respond with the data word on the fifth clock. This type of protocol can be implemented easily in a small finite state machine. Because the protocol is predetermined and involves little logic, the bus can run very fast and the interface logic will be small. Synchronous buses have two major disadvantages, however. First, every device on the bus must run at the same clock rate. Second, because of clock skew problems, synchronous buses cannot be long if they are fast (see Appendix B for a discussion of clock skew). Processor-memory buses are often synchronous because the devices communicating are close, small in number, and prepared to operate at high clock rates.

An asynchronous bus is not clocked. Because it is not clocked, an asynchronous bus can accommodate a wide variety of devices, and the bus can be lengthened without worrying about clock skew or synchronization problems. To coordinate the transmission of data between sender and receiver, an asynchronous bus uses a *handshaking protocol*. A handshaking protocol consists of a series of steps in which the sender and receiver proceed to the next step only when both parties agree. The protocol is implemented with an additional set of control lines.

A simple example will illustrate how asynchronous buses work. Let's consider a device requesting a word of data from the memory system. Assume that there are three control lines:

1. *ReadReq*: Used to indicate a read request for memory. The address is put on the data lines at the same time.
2. *DataRdy*: Used to indicate that the data word is now ready on the data lines. In an output transaction, the memory will assert this signal since it is providing the data. In an input transaction, an I/O device would assert this signal, since it would provide data. In either case, the data is placed on the data lines at the same time.
3. *Ack*: Used to acknowledge the ReadReq or the DataRdy signal of the other party.

In an asynchronous protocol, the control signals ReadReq and DataRdy are asserted until the other party (the memory or the device) indicates that the control lines have been seen and the data lines have been read; this indication is made by asserting the Ack line. This complete process is called *handshaking*. Figure 8.10 shows how such a protocol operates by depicting the steps in the communication.



The steps in the protocol begin immediately after the device signals a request by raising ReadReq and putting the address on the Data lines:

1. When memory sees the ReadReq line, it reads the address from the data bus and raises Ack to indicate it has been seen.
2. I/O device sees the Ack line high and releases the ReadReq and data lines.
3. Memory sees that ReadReq is low and drops the Ack line to acknowledge the Readreq signal.
4. This step starts when the memory has the data ready. It places the data from the read request on the data lines and raises DataRdy.
5. The I/O device sees DataRdy, reads the data from the bus, and signals that it has the data by raising Ack.
6. The memory sees the Ack signal, drops DataRdy, and releases the data lines.
7. Finally, the I/O device, seeing DataRdy go low, drops the Ack line, which indicates that the transmission is completed.

A new bus transaction can now begin.

FIGURE 8.10 The asynchronous handshaking protocol consists of seven steps to read a word from memory and receive it in an I/O device. The signals in color are those asserted by the I/O device, while the memory asserts the signals shown in black. The arrows label the seven steps and the event that triggers each step. The symbol showing two lines (high and low) at the same time on the data lines indicates that the data lines have valid data at this point. (The symbol indicates that the data is valid, but the value is not known.)

An asynchronous bus protocol works like a pair of finite state machines that are communicating in such a way that a machine does not proceed until it knows that another machine has reached a certain state; thus the two machines are coordinated.

The handshaking protocol does not solve all the problems of communicating between a sender and receiver that have different clocks. An additional problem arises when we sample an asynchronous signal (such as ReadReq). This problem, called a *synchronization failure*, can lead to unpredictable behavior; it can be overcome with devices called *synchronizers*, which are described in Appendix B.

FSM Control for I/O

Example Show how the control for an output transaction to an I/O device from memory (as in Figure 8.7) can be implemented as a pair of finite state machines.

Answer Figure 8.11 shows the two finite state machine controllers that implement the handshaking protocol of Figure 8.10.

If a synchronous bus can be used, it is usually faster than an asynchronous bus because of the overhead required to perform the handshaking. An example demonstrates this.

Performance Analysis of Synchronous versus Asynchronous Buses

Example We want to compare the maximum bandwidth for a synchronous and an asynchronous bus. The synchronous bus has a clock cycle time of 50 ns, and each bus transmission takes 1 clock cycle. The asynchronous bus requires 40 ns per handshake. The data portion of both buses is 32 bits wide. Find the bandwidth for each bus when performing one-word reads from a 200-ns memory.

Answer First, the synchronous bus, which has 50-ns bus cycles. The steps and times required for the synchronous bus are as follows:

1. Send the address to memory: 50 ns
2. Read the memory: 200 ns
3. Send the data to the device: 50 ns

Thus, the total time is 300 ns. This yields a maximum bus bandwidth of 4 bytes every 300 ns, or

$$\frac{4 \text{ bytes}}{300 \text{ ns}} = \frac{4 \text{ MB}}{0.3 \text{ seconds}} = 13.3 \frac{\text{MB}}{\text{second}}$$

At first glance, it might appear that the asynchronous bus will be *much* slower, since it will take seven steps, each at least 40 ns, and the step corresponding to the memory access will take 200 ns. If we look carefully at Figure 8.10, we realize that several of the steps can be overlapped with the memory access time. In particular, the memory receives the address at the end of step 1 and does not need to put the data on the bus until the beginning of step 5; steps 2, 3, and 4 can overlap with the memory access time. This leads to the following timing:

Step 1: 40 ns

Steps 2, 3, 4: maximum (3 x 40 ns, 200 ns) = 200 ns

Steps 5, 6, 7: 3 x 40 ns = 120 ns

Thus, the total time to perform the transfer is 360 ns, and the maximum bandwidth is

$$\frac{4 \text{ bytes}}{360 \text{ ns}} = \frac{4 \text{ MB}}{0.36 \text{ seconds}} = 11.1 \frac{\text{MB}}{\text{second}}$$

Accordingly, the synchronous bus is only about 20% faster. Of course, to sustain these rates, the device and memory system on the asynchronous bus will need to be fairly fast to accomplish each handshaking step in 40 ns.

Even though a synchronous bus may be faster, the choice between a synchronous and an asynchronous bus has implications not only for data bandwidth but also for an I/O system's capacity in terms of physical distance and the number of devices that can be connected to the bus. Asynchronous buses scale better with technology changes and can support a wider variety of device response speeds. It is for these reasons that I/O buses are often asynchronous, despite the increased overhead.

Increasing the Bus Bandwidth

Although much of the bandwidth of a bus is decided by the choice of a synchronous or asynchronous protocol and the timing characteristics of the bus, several other factors affect the bandwidth that can be attained by a single transfer. The most important of these are the following:

1. *Data bus width*: By increasing the width of the data bus, transfers of multiple words require fewer bus cycles.
2. *Separate versus multiplexed address and data lines*: Our example in Figure 8.8 used the same wires for address and data; including separate lines for addresses will make the performance of writes faster because the address and data can be transmitted in one bus cycle.

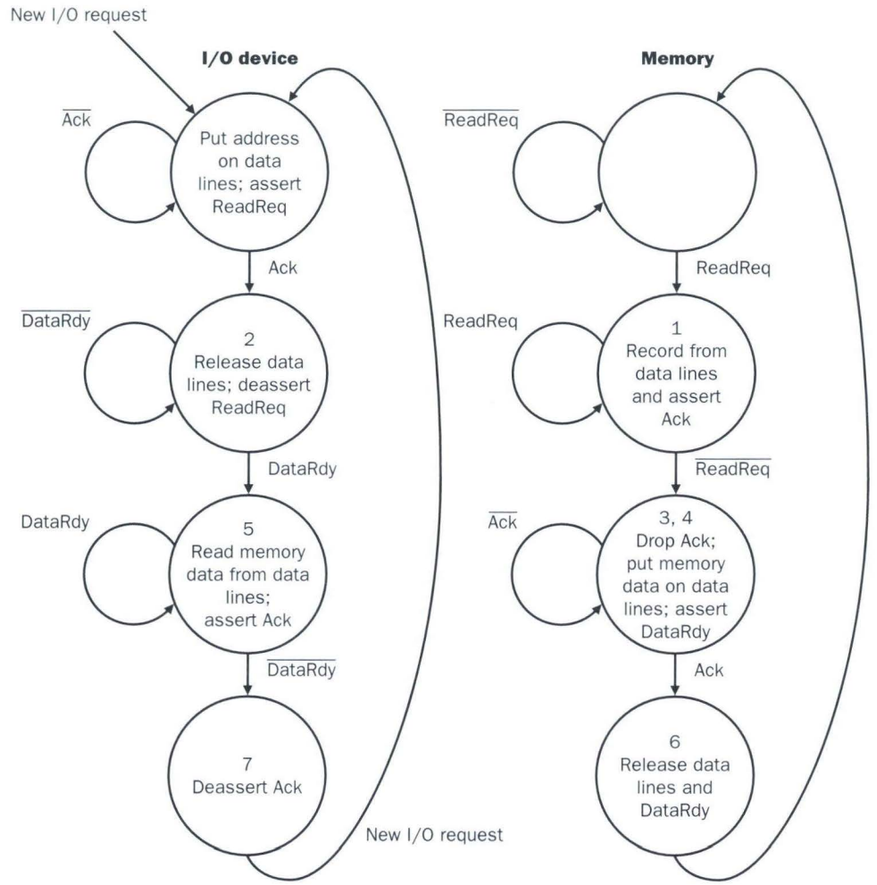


FIGURE 8.11 These finite state machines implement the control for the handshaking protocol illustrated in Figure 8.10. The numbers in each state correspond to the steps shown in Figure 8.10. The first state of the I/O device (upper-left corner) starts the protocol when a new I/O request is generated, just as in Figure 8.10. Each state in the finite state machine effectively records the state of both the device and memory. This is how they stay synchronized during the transaction. After completing a transaction, the I/O side can stay in the last state until a new request needs to be processed.

3. *Block transfers*: Allowing the bus to transfer multiple words in back-to-back bus cycles without sending an address or releasing the bus will reduce the time needed to transfer a large block.

Each of these design alternatives will increase the bus performance for a single bus transfer. The cost of implementing one of these enhancements is one or more of the following: more bus lines, increased complexity, or increased response time for requests that may need to wait while a long block transfer occurs.

Performance Analysis of Two Bus Schemes

Example

Suppose we have a system with the following characteristics:

1. A memory and bus system supporting block access of 4 to 16 32-bit words.
2. A 64-bit synchronous bus clocked at 200 MHz, with each 64-bit transfer taking 1 clock cycle, and 1 clock cycle required to send an address to memory.
3. Two clock cycles needed between each bus operation. (Assume the bus is idle before an access.)
4. A memory access time for the first four words of 200 ns; each additional set of four words can be read in 20 ns. Assume that a bus transfer of the most recently read data and a read of the next four words can be overlapped.

Find the sustained bandwidth and the latency for a read of 256 words for transfers that use 4-word blocks and for transfers that use 16-word blocks. Also compute the effective number of bus transactions per second for each case. Recall that a single bus transaction consists of an address transmission followed by data.

Answer

For the 4-word block transfers, each block takes

1. 1 clock cycle that is required to send the address to memory
2. $\frac{200 \text{ ns}}{5 \text{ ns/cycle}} = 40$ clock cycles to read memory
3. 2 clock cycles to send the data from the memory
4. 2 idle clock cycles between this transfer and the next

This is a total of 45 cycles, and 256/4 = 64 transactions are needed, so the entire transfer takes $45 \times 64 = 2880$ clock cycles. Thus the latency is 2880 cycles $\times$ 5 ns/cycle = 14,400 ns. The number of bus transactions per second is

$$64 \text{ transactions} \times \frac{1 \text{ second}}{14{,}400 \text{ ns}} = 4.44\text{M transactions/second}$$

The bus bandwidth is

$$(256 \times 4) \text{ bytes} \times \frac{1 \text{ second}}{14{,}400 \text{ ns}} = 71.11 \text{ MB/sec}$$

For the 16-word block transfers, the first block requires

1. 1 clock cycle to send an address to memory
2. 200 ns or 40 cycles to read the first four words in memory
3. 2 cycles to send the data of the block, during which time the read of the four words in the next block is started
4. 2 idle cycles between transfers and during which the read of the next block is completed

Each of the three remaining 16-word blocks requires repeating only the last two steps.

Thus, the total number of cycles for each 16-word block is $1 + 40 + 4 \times (2 + 2) = 57$ cycles, and $256/16 = 16$ transactions are needed, so the entire transfer takes, $57 \times 16 = 912$ cycles. Thus the latency is 912 cycles $\times$ 5 ns/cycle = 4560 ns, which is roughly one-third of the latency for the case with 4-word blocks. The number of bus transactions per second with 16-word blocks is

$$16 \text{ transactions} \times \frac{1 \text{ second}}{4560 \text{ ns}} = 3.51\text{M transactions/second}$$

which is lower than the case with 4-word blocks because each transaction takes longer (57 versus 45 cycles).

The bus bandwidth with 16-word blocks is

$$(256 \times 4) \text{ bytes} \times \frac{1 \text{ second}}{4560 \text{ ns}} = 224.56 \text{ MB/second}$$

which is 3.16 times higher than for the 4-word blocks. The advantage of using larger block transfers is clear.

Elaboration: Another method for increasing the effective bus bandwidth when multiple parties want to communicate on the bus is to release the bus when it is not being used for transmitting information. Consider the example of a memory read that we examined in Figure 8.10. What happens to the bus while the memory access is occurring? In this simple protocol, the device and memory continue to hold the bus during the memory access time when no actual transfer is taking place. An alternative protocol, which releases the bus, would operate like this:

1. The device signals the memory and transmits the request and address.
2. After the memory acknowledges the request, both the memory and device release all control lines.
3. The memory access occurs, and the bus is free for other uses during this period.

4. The memory signals the device on the bus to indicate that the data is available.
5. The device receives the data via the bus and signals that it has the data, so the memory system can release the bus.

For the synchronous bus with 16-word transfers in the example above, such a scheme would occupy the bus for only 272 of the 912 cycles required for the complete bus transaction.

This type of protocol is called a *split transaction protocol*. The advantage of such a protocol is that, by freeing the bus during the time data is not being transmitted, the protocol allows another requestor to use the bus. This can improve the effective bus bandwidth for the entire system, if the memory is sophisticated enough to handle multiple overlapping transactions.

With a split transaction, however, the time to complete one transfer is probably increased because the bus must be acquired twice. Split transaction protocols are also more expensive to implement, primarily because of the need to keep track of the other party in a communication. In a split transaction protocol, the memory system must contact the requestor to initiate the reply portion of the bus transaction, so the identity of the requestor must be transmitted and retained by the memory system.

Obtaining Access to the Bus

Now that we have reviewed some of the many design options for buses, we can deal with one of the most important issues in bus design: How is the bus reserved by a device that wishes to use it to communicate? We touched on this question in several of the above discussions, and it is crucial in designing large I/O systems that allow I/O to occur without the processor's continuous and low-level involvement.

Why is a scheme needed for controlling bus access? Without any control, multiple devices desiring to communicate could each try to assert the control and data lines for different transfers! Just as chaos reigns in a classroom when everyone tries to talk at once, multiple devices trying to use the bus simultaneously would result in confusion.

Chaos is avoided by introducing one or more *bus masters* into the system. A bus master controls access to the bus: it must initiate and control all bus requests. The processor must be able to initiate a bus request for memory and thus is always a bus master. The memory is usually a *slave*—since it will respond to read and write requests but never generate its own requests.

The simplest system possible has a single bus master: the processor. Having a single bus master is similar to what normally happens in a classroom—all communication requires the permission of the instructor. In a single-master system, all bus requests must be controlled by the processor. The steps involved in a bus transaction with a single-master bus are shown in Figure 8.12. The major drawback of this approach is that the processor must be involved in every bus transaction. A single sector read from a disk may require the processor to get involved hundreds to thousands of times, depending on the size of each transfer. Because devices have become faster and capable of transferring

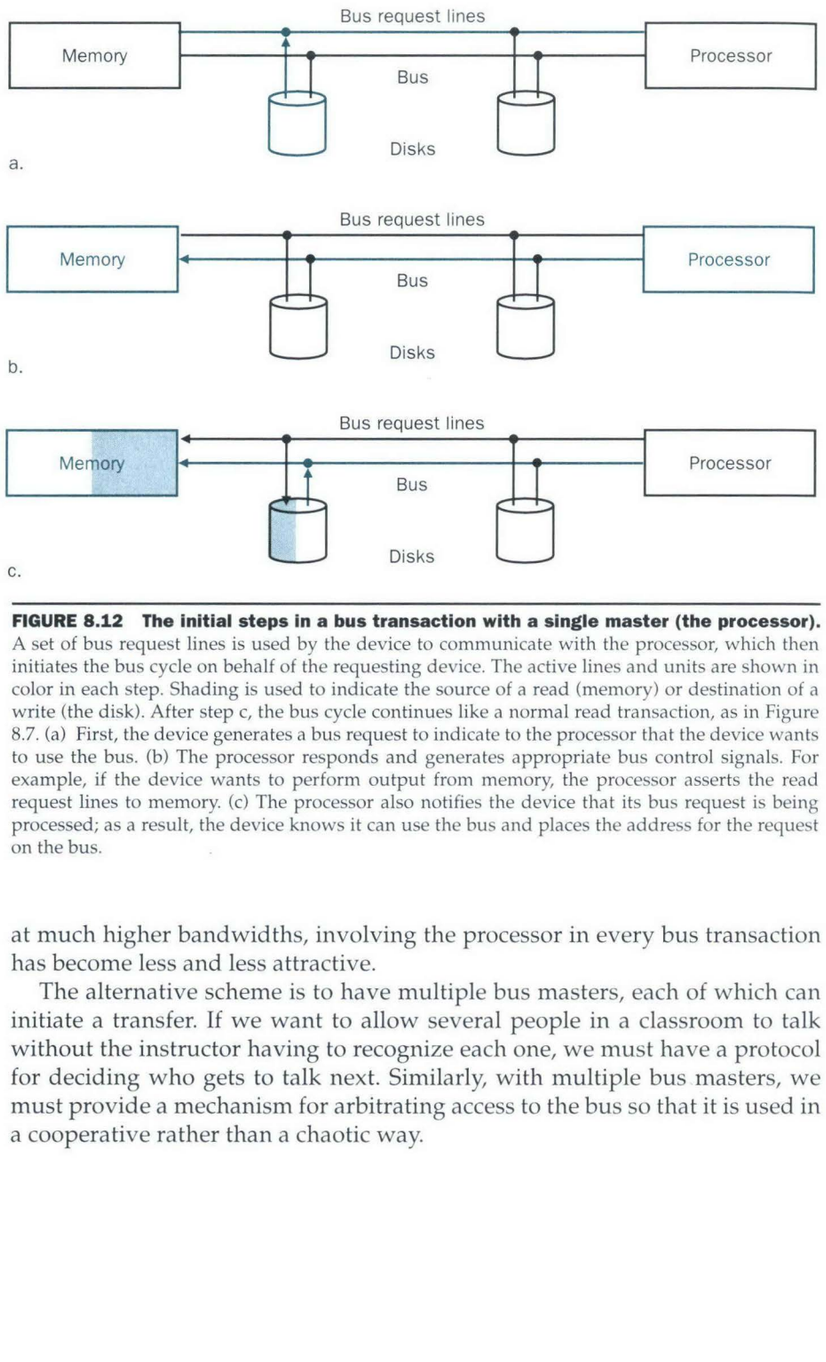


FIGURE 8.12 The initial steps in a bus transaction with a single master (the processor). A set of bus request lines is used by the device to communicate with the processor, which then initiates the bus cycle on behalf of the requesting device. The active lines and units are shown in color in each step. Shading is used to indicate the source of a read (memory) or destination of a write (the disk). After step c, the bus cycle continues like a normal read transaction, as in Figure 8.7. (a) First, the device generates a bus request to indicate to the processor that the device wants to use the bus. (b) The processor responds and generates appropriate bus control signals. For example, if the device wants to perform output from memory, the processor asserts the read request lines to memory. (c) The processor also notifies the device that its bus request is being processed; as a result, the device knows it can use the bus and places the address for the request on the bus.

at much higher bandwidths, involving the processor in every bus transaction has become less and less attractive.

The alternative scheme is to have multiple bus masters, each of which can initiate a transfer. If we want to allow several people in a classroom to talk without the instructor having to recognize each one, we must have a protocol for deciding who gets to talk next. Similarly, with multiple bus masters, we must provide a mechanism for arbitrating access to the bus so that it is used in a cooperative rather than a chaotic way.

Bus Arbitration

Deciding which bus master gets to use the bus next is called *bus arbitration*. There are a wide variety of schemes for bus arbitration; these may involve special hardware or extremely sophisticated bus protocols. In a bus arbitration scheme, a device (or the processor) wanting to use the bus signals a *bus request* and is later *granted* the bus. After a grant, the device can use the bus, later signaling to the arbiter that the bus is no longer required. The arbiter can then grant the bus to another device. Most multiple-master buses have a set of bus lines for performing requests and grants. A bus release line is also needed if each device does not have its own request line. Sometimes the signals used for bus arbitration have physically separate lines, while in other systems the data lines of the bus are used for this function (though this prevents overlapping of arbitration with transfer).

Arbitration schemes usually try to balance two factors in choosing which device to grant the bus. First, each device has a *bus priority*, and the highest-priority device should be serviced first. Second, we would prefer that any device, even one with low priority, never be completely locked out from the bus. This property, called *fairness*, ensures that every device that wants to use the bus is guaranteed to get it eventually. In addition to these factors, more sophisticated schemes aim at reducing the time needed to arbitrate for the bus. Because arbitration time is overhead, which increases the bus access time, it should be reduced and overlapped with bus transfers whenever possible.

Bus arbitration schemes can be divided into four broad classes:

- *Daisy chain arbitration*: In this scheme, the bus grant line is run through the devices from highest priority to lowest (the priorities are determined by the position on the bus). A high-priority device that desires bus access simply intercepts the bus grant signal, not allowing a lower-priority device to see the signal. Figure 8.13 shows how a daisy chain bus is organized. The advantage of a daisy chain bus is simplicity; the disadvantages are that it cannot assure fairness—a low-priority request may be locked out indefinitely—and the use of the daisy chain grant signal also limits the bus speed.
- *Centralized, parallel arbitration*: These schemes use multiple request lines, and the devices independently request the bus. A centralized arbiter chooses from among the devices requesting bus access and notifies the selected device that it is now bus master. The disadvantage of this scheme is that it requires a central arbiter, which may become the bottleneck for bus usage. PCI, a standard backplane bus, uses a central arbitration scheme.

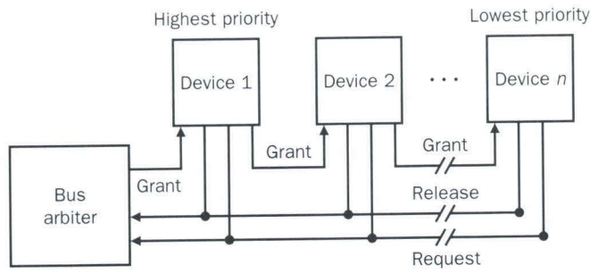


FIGURE 8.13 A daisy chain bus uses a bus grant line that chains through each device from highest to lowest priority. If the device has requested bus access, it uses the grant line to determine access has been given to it. Because the grant line is passed on only if a device does not want access, priority is built into the scheme. The name "daisy chain" arises from the structure of the grant line that chains from device to device. The detailed protocol used by a daisy chain is described in the elaboration below.

- *Distributed arbitration by self-selection*: These schemes also use multiple request lines, but the devices requesting bus access determine who will be granted access. Each device wanting bus access places a code indicating its identity on the bus. By examining the bus, the devices can determine the highest-priority device that has made a request. There is no need for a central arbiter; each device determines independently whether it is the high-priority requestor. This scheme, however, does require more lines for request signals. The NuBus, which is the backplane bus in Apple Macintosh IIs, uses this scheme.
- *Distributed arbitration by collision detection*: In this scheme, each device independently requests the bus. Multiple simultaneous requests result in a *collision*. The collision is detected and a scheme for selecting among the colliding parties is used. Ethernets, which use this scheme, are further described in Exercise 8.28 on page 708.

The suitability of different arbitration schemes is determined by a variety of factors, including how expandable the bus must be both in terms of the number of I/O devices and the bus length, how fast the arbitration should be, and what degree of fairness is needed.

Elaboration: The protocol followed by a device on a daisy chain bus is the following:

1. Signal the request line.
2. Wait for a transition on the grant line from low to high, indicating that the bus is being reassigned.

3. Intercept the grant signal, and do not allow lower-priority devices to see it. Stop asserting the request line.
4. Use the bus.
5. Signal that the bus is no longer required by asserting the release line.

By watching for a transition on the grant line, rather than just a level, we prevent the device from taking the bus away from a lower-priority device that believes it has been granted bus access. To improve fairness in a daisy chain scheme, we can simply make the rule that a device that has just used the bus cannot reacquire the bus until it sees the bus request line go low. Since a device will not release the request line until its request is satisfied, all devices will have an opportunity to use the bus before any single device uses it twice. Some bus systems—VME, for example—use multiple daisy chains with a separate set of request and grant lines for each daisy chain and a priority encoder to select from among the multiple requests.

The Big Picture

The different bus characteristics allow the creation of buses optimized for a wide range of different devices, number of devices, and bandwidth demands. Figure 8.14 shows some of the design alternatives we have discussed and what choices might be made in low-cost versus high-performance systems. In general, higher-cost systems use wider and faster buses with more sophisticated protocols—typically a synchronous bus for the reasons we saw in the example on page 662. In contrast, a low-cost system favors a bus that is narrower and does not require intelligence among the devices (hence a single master), and is asynchronous so that low-speed devices can interface inexpensively.

Option	High performance	Low cost
Bus width	separate address and data lines	multiplex address and data lines
Data width	wider is faster (e.g., 32 bits)	narrower is cheaper (e.g., 8 bits)
Transfer size	multiple words require less bus overhead	single-word transfer is simpler
Bus masters	multiple masters (requires arbitration)	single master (no arbitration)
Clocking	synchronous	asynchronous

FIGURE 8.14 The I/O bus characteristics determine the performance of I/O transfers, the number of I/O devices that can be connected, and the cost of connecting devices. Shorter buses can be faster but will not be as expandable. Similarly, wider buses can have higher bandwidth but will be more expensive. Split transaction buses are another way to increase bandwidth at the expense of cost (see the elaboration on page 666).

Bus Standards

Most computers allow users to add additional and even new types of peripherals. The I/O bus serves as a way of expanding the machine and connecting new peripherals. To make this easier, the computer industry has developed several bus standards. The standards serve as a specification for the computer manufacturer and for the peripheral manufacturer. A bus standard ensures the computer designer that peripherals will be available for a new machine, and it ensures the peripheral builder that users will be able to hook up their new equipment.

Machines sometimes become so popular that their I/O buses become de facto standards, as is the case with the IBM PC-AT bus. Once a bus standard is heavily used by peripheral designers, other computer manufacturers incorporate that bus and offer a wide range of peripherals. Sometimes standards are created by groups that are trying to address a common problem. The small computer systems interface (SCSI) and Ethernet are examples of standards that arose from the cooperation of manufacturers. Sanctioning bodies like ANSI or IEEE also create and approve standards. The PCI standard was initiated by Intel and later developed by an industry committee.

Figure 8.15 summarizes the key characteristics of the two dominant bus standards: PCI (a general-purpose backplane bus) and SCSI (an I/O bus). A SCSI bus typically interfaces to a backplane bus or to a processor-memory bus. A SCSI controller coordinates transfers from a device on the I/O bus to the memory via the processor-memory bus. One emerging bus standard is Fibre Channel, proposed as a follow-on to SCSI and based on high-speed point-to-point links, which would be organized as a loop for multiple devices.

Bus bandwidth for a general-purpose bus is not simply a single number. Because of bus overhead, the size of the transfer affects bandwidth significantly. Since the bus usually transfers to or from memory, the speed of the memory also affects the bandwidth.

Buses provide the electrical interconnect among I/O devices, processors, and memory, and also define the lowest-level protocol for communication. Above this basic level, we must define hardware and software protocols for controlling data transfers between I/O devices and memory, and for the processor to specify commands to the I/O devices. These topics are covered in the next section.

Characteristic	PCI	SCSI
Bus type	backplane	I/O
Basic data bus width (signals)	32–64	8–32
Address/data multiplexed?	multiplexed	multiplexed
Number of bus masters	multiple	multiple
Arbitration	centralized, parallel arbitration	self-selection
Clocking	synchronous 33–66 MHz	asynchronous or synchronous (5–10 MHz)
Theoretical peak bandwidth	133–512 MB/sec	5–40 MB/sec
Estimated typical achievable bandwidth for basic bus	80 MB/sec	2.5–40.0 MB/sec (synchronous) or 1.5 MB/sec (asynchronous)
Maximum number of devices	1024 (with multiple bus segments; at most 32 devices/bus segment)	7–31 (bus width – 1)
Maximum bus length	0.5 meter	2.5 meters
Standard name	PCI	ANSI X3.131

FIGURE 8.15 Key characteristics of two dominant bus standards. Both PCI and SCSI bus standards have been significantly extended. PCI has a double-width version (64 bits vs. 32 bits) and a fast version (66 MHz vs. 33 MHz). The original SCSI bus was asynchronous. Faster, synchronous versions were developed, followed by extensions for a wider bus (16 and 32 bits versus 8, called wide SCSI) and a faster clock (10 MHz, called fast SCSI, vs. 5 MHz for the original synchronous SCSI). Fast, wide SCSI combines the higher clock rate and wider bus. In addition, a 20-MHz version of the SCSI bus (called Ultra) was developed and released in late 1996. The specifications for these standard buses become extremely complex. For example, the PCI standard is 282 pages long, while the SCSI-2 specification, which includes both the faster and wider versions, is over 600 pages long! The SCSI-2 specification, a good overview of SCSI and its development, and the PCI specification are available via links at *www.mkp.com/books_catalog/cod/links.htm*.

ENHANCED WEB

8.5 Interfacing I/O Devices to the Memory, Processor, and Operating System

A bus protocol defines how a word or block of data should be communicated on a set of wires. This still leaves several other tasks that must be performed to actually cause data to be transferred from a device and into the memory address space of some user program. This section focuses on these tasks and will answer such questions as the following:

- How is a user I/O request transformed into a device command and communicated to the device?
- How is data actually transferred to or from a memory location?
- What is the role of the operating system?

As we will see when we answer these questions, the operating system plays a major role in handling I/O, acting as the interface between the hardware and the program that requests I/O.

The responsibilities of the operating system arise from three characteristics of I/O systems:

1. The I/O system is shared by multiple programs using the processor.

2. I/O systems often use interrupts (externally generated exceptions) to communicate information about I/O operations. Because interrupts cause a transfer to kernel or supervisor mode, they must be handled by the operating system (OS).
3. The low-level control of an I/O device is complex because it requires managing a set of concurrent events and because the requirements for correct device control are often very detailed.

Hardware Software Interface

The three characteristics of I/O systems above lead to several different functions the OS must provide:

- The OS guarantees that a user's program accesses only the portions of an I/O device to which the user has rights. For example, the OS must not allow a program to read or write a file on disk if the owner of the file has not granted access to this program. In a system with shared I/O devices, protection could not be provided if user programs could perform I/O directly.
- The OS provides abstractions for accessing devices by supplying routines that handle low-level device operations.
- The OS handles the interrupts generated by I/O devices, just as it handles the exceptions generated by a program.
- The OS tries to provide equitable access to the shared I/O resources, as well as schedule accesses in order to enhance system throughput.

To perform these functions on behalf of user programs, the operating system must be able to communicate with the I/O devices and to prevent the user program from communicating with the I/O devices directly. Three types of communication are required:

1. The OS must be able to give commands to the I/O devices. These commands include not only operations like read and write, but other operations to be done on the device, such as a disk seek.
2. The device must be able to notify the OS when the I/O device has completed an operation or has encountered an error. For example, when a disk has completed a seek, it will notify the OS.
3. Data must be transferred between memory and an I/O device. For example, the block being read on a disk read must be moved from disk to memory.

In the next few sections, we will see how these communications are performed.

Giving Commands to I/O Devices

To give a command to an I/O device, the processor must be able to address the device and to supply one or more command words. Two methods are used to address the device: memory-mapped I/O and special I/O instructions. In memory-mapped I/O, portions of the address space are assigned to I/O devices. Reads and writes to those addresses are interpreted as commands to the I/O device.

For example, a write operation can be used to send data to an I/O device where the data will be interpreted as a command. When the processor places the address and data on the memory bus, the memory system ignores the operation because the address indicates a portion of the memory space used for I/O. The device controller, however, sees the operation, records the data, and transmits it to the device as a command. User programs are prevented from issuing I/O operations directly because the OS does not provide access to the address space assigned to the I/O devices and thus the addresses are protected by the address translation. Memory-mapped I/O can also be used to transmit data by writing or reading to select addresses. The device uses the address to determine the type of command, and the data may be provided by a write or obtained by a read. In any event, the address encodes both the device identity and the type of transmission between processor and device.

Actually performing a read or write of data to fulfill a program request usually requires several separate I/O operations. Furthermore, the processor may have to interrogate the status of the device between individual commands to determine whether the command completed successfully. For example, the DEC LP11 line printer has two I/O device registers—one for status information and one for data to be printed. The Status register contains a *done bit*, set by the printer when it has printed a character, and an *error bit*, indicating that the printer is jammed or out of paper. Each byte of data to be printed is put into the Data register. The processor must then wait until the printer sets the done bit before it can place another character in the buffer. The processor must also check the error bit to determine if a problem has occurred. Each of these operations requires a separate I/O device access.

Elaboration: The alternative to memory-mapped I/O is to use dedicated I/O instructions in the processor. These I/O instructions can specify both the device number and the command word (or the location of the command word in memory). The processor communicates the device address via a set of wires normally included as part of the I/O bus. The actual command can be transmitted over the data lines in the bus. Examples of computers with I/O instructions are the Intel 80x86 and the IBM 370 computers. By making the I/O instructions illegal to execute when not in kernel or supervisor mode, user programs can be prevented from accessing the devices directly.

Communicating with the Processor

The process of periodically checking status bits to see if it is time for the next I/O operation, as in the previous example, is called *polling*. Polling is the simplest way for an I/O device to communicate with the processor. The I/O device simply puts the information in a Status register, and the processor must come and get the information. The processor is totally in control and does all the work.

The disadvantage of polling is that it can waste a lot of processor time because processors are so much faster than I/O devices. The processor may read the Status register many times, only to find that the device has not yet completed a comparatively slow I/O operation, or that the mouse has not budged since the last time it was polled. When the device has completed an operation, we must still read the status to determine whether it was successful.

Polling can be used in several different ways, depending on the I/O device and whether the I/O device can initiate I/O independently. For example, a mouse is an input-only device that initiates I/O independently, when a user moves the mouse or clicks a button. Because a mouse has a low I/O rate, polling is often used to interface to a mouse. Many other I/O devices, such as a floppy disk or a printer, initiate I/O only under control of the operating system. Thus we need only poll such devices when the OS knows that the device is active. As we will see, this allows polling to be used even when the I/O rate is somewhat higher.

Overhead of Polling in an I/O System

Example

Let's determine the impact of polling overhead for three different devices. Assume that the number of clock cycles for a polling operation—including transferring to the polling routine, accessing the device, and restarting the user program—is 400 and that the processor executes with a 500-MHz clock.

Determine the fraction of CPU time consumed for the following three cases, assuming that you poll often enough so that no data is ever lost and assuming that the devices are potentially always busy:

1. The mouse must be polled 30 times per second to ensure that we do not miss any movement made by the user.
2. The floppy disk transfers data to the processor in 16-bit units and has a data rate of 50 KB/sec. No data transfer can be missed.
3. The hard disk transfers data in four-word chunks and can transfer at 4 MB/sec. Again, no transfer can be missed.

Answer

First the mouse:

$$\text{Clock cycles per second for polling} = 30 \times 400 = 12{,}000 \text{ cycles per second}$$

$$\text{Fraction of the processor clock cycles consumed} = \frac{12 \times 10^3}{500 \times 10^6} = 0.002\%$$

Polling can clearly be used for the mouse without much performance impact on the processor.

For the floppy disk, the rate at which we must poll is

$$\frac{50 \dfrac{\text{KB}}{\text{second}}}{2 \dfrac{\text{bytes}}{\text{polling access}}} = 25\text{K} \frac{\text{polling accesses}}{\text{second}}$$

Thus, we can compute the number of cycles (ignoring the base 2 versus base 10 discrepancy):

$$\text{Cycles per second for polling} = 25\text{K} \times 400$$

$$\text{Fraction of the processor consumed} = \frac{10 \times 10^6}{500 \times 10^6} = 2\%$$

This amount of overhead is significant, but might be tolerable in a low-end system with only a few I/O devices like this floppy disk.

In the case of the hard disk, we must poll at a rate equal to the data rate in four-word chunks, which is 250K times per second (4 MB per second/16 bytes per transfer). Thus,

$$\text{Cycles per second for polling} = 250\text{K} \times 400$$

Ignoring the discrepancy in bases,

$$\text{Fraction of the processor consumed} = \frac{100 \times 10^6}{500 \times 10^6} = 20\%$$

Thus one-fifth of the processor would be used in just polling the disk. Clearly, polling will probably be unacceptable for a hard disk on this machine.

If we knew that the floppy disk and hard disk were active only 25% of the time and we poll only when the device is active, then the average overhead for polling would be reduced to 0.5% and 5%, respectively. Although this reduces the overhead, notice that once the OS initiates an operation on the device, it must poll continuously since the OS does not know when the device will actually respond and want to initiate a transfer.

The overhead in a polling interface was recognized long ago, leading to the invention of interrupts to notify the processor when an I/O device requires attention from the processor. *Interrupt-driven I/O*, which is used by almost all systems for at least some devices, employs I/O interrupts to indicate to the processor that an I/O device needs attention. When a device wants to notify the processor that it has completed some operation or needs attention, it causes the processor to be interrupted.

An I/O interrupt is just like the exceptions we saw in Chapters 5, 6, and 7, with two important exceptions:

1. An I/O interrupt is asynchronous with respect to the instruction execution. That is, the interrupt is not associated with any instruction and does not prevent the instruction completion. This is very different from either page fault exceptions or exceptions such as arithmetic overflow. Our control unit need only check for a pending I/O interrupt at the time it starts a new instruction.

2. In addition to the fact that an I/O interrupt has occurred, we would like to convey further information such as the identity of the device generating the interrupt. Furthermore, the interrupts represent devices that may have different priorities and whose interrupt requests have different urgencies associated with them.

To communicate information to the processor, such as the identity of the device raising the interrupt, a system can use either vectored interrupts or an exception Cause register. When the interrupt is recognized by the processor, the device can send either the vector address or a status field to place in the Cause register. As a result, when the OS gets control, it knows the identity of the device that caused the interrupt and can immediately interrogate the device. An interrupt mechanism eliminates the need for the processor to poll the device and instead allows the processor to focus on executing programs.

Elaboration: To deal with the different priorities of the I/O devices, most interrupt mechanisms have several levels of priority. These priorities indicate the order in which the processor should process interrupts. Both internally generated exceptions and I/O interrupts have priorities; typically, I/O interrupts have lower priority than internal exceptions. There may be multiple I/O interrupt priorities, with high-speed devices associated with the higher priorities. If the exception mechanism is vectored (see section 5.6), the vector address for a fast device will correspond to the higher-priority interrupt. If a Cause register is used, then the register contents for a faster device are set for the higher-priority interrupt.

Transferring the Data between a Device and Memory

We have seen two different methods that enable a device to communicate with the processor. These two techniques, polling and I/O interrupts, form the basis for two methods of implementing the transfer of data between the I/O device and memory. Both these techniques work best with lower-bandwidth devices, where we are more interested in reducing the cost of the device controller and interface than in providing a high-bandwidth transfer. Both polling and interrupt-driven transfers put the burden of moving data and managing the transfer on the processor. After looking at these two schemes, we will examine a scheme more suitable for higher-performance devices or collections of devices.

We can use the processor to transfer data between a device and memory based on polling. Consider our mouse example. The processor can periodically read the mouse counter values and the position of the mouse buttons. If the position of the mouse or one of its buttons has changed, the operating system can notify the program associated with interpreting the mouse changes.

An alternative mechanism is to make the transfer of data interrupt driven. In this case, the OS would still transfer data in small numbers of bytes from or to the device. But because the I/O operation is interrupt driven, the OS simply works on other tasks while data is being read from or written to the device. When the OS recognizes an interrupt from the device, it reads the status to check for errors. If there are none, the OS can supply the next piece of data, for example, by a sequence of memory-mapped writes. When the last byte of an I/O request has been transmitted and the I/O operation is completed, the OS can inform the program. The processor and OS do all the work in this process, accessing the device and memory for each data item transferred. Let's see how an interrupt-driven I/O interface might work for the floppy disk.

Overhead of Interrupt-Driven I/O

Example Suppose we have the same hard disk and processor we used in the example on page 676, but we use interrupt-driven I/O. The overhead for each transfer, including the interrupt, is 500 clock cycles. Find the fraction of the processor consumed if the hard disk is only transferring data 5% of the time.

Answer The interrupt rate when the disk is busy is the same as the polling rate. Hence,

$$\begin{aligned}\text{Cycles per second for disk} &= 250\text{K} \times 500 \\ &= 125 \times 10^6 \text{ cycles per second}\end{aligned}$$

$$\text{Fraction of the processor consumed during a transfer} = \frac{125 \times 10^6}{500 \times 10^6} = 25\%$$

Assuming that the disk is only transferring data 5% of the time,

$$\text{Fraction of the processor consumed on average} = 25\% \times 5\% = 1.25\%$$

As we can see, the absence of overhead when an I/O device is not actually transferring is the major advantage of an interrupt-driven interface versus polling.

Interrupt-driven I/O relieves the processor from having to wait for every I/O event, although if we used this method for transferring data from or to a hard disk, the overhead could still be intolerable, since it would consume 25% of the processor when the disk was transferring. For high-bandwidth devices like hard disks, the transfers consist primarily of relatively large blocks of data (hundreds to thousands of bytes). So computer designers invented a mechanism for off-loading the processor and having the device controller transfer data directly to or from the memory without involving the processor. This mechanism is called *direct memory access* (DMA). The interrupt mechanism is still used by the device to communicate with the processor, but only on completion of the I/O transfer or when an error occurs.

DMA is implemented with a specialized controller that transfers data between an I/O device and memory independent of the processor. The DMA controller becomes the bus master and directs the reads or writes between itself and memory. There are three steps in a DMA transfer:

1. The processor sets up the DMA by supplying the identity of the device, the operation to perform on the device, the memory address that is the source or destination of the data to be transferred, and the number of bytes to transfer.
2. The DMA starts the operation on the device and arbitrates for the bus. When the data is available (from the device or memory), it transfers the data. The DMA device supplies the memory address for the read or write. If the request requires more than one transfer on the bus, the DMA unit generates the next memory address and initiates the next transfer. Using this mechanism, a DMA unit can complete an entire transfer, which may be thousands of bytes in length, without bothering the processor. Many DMA controllers contain some memory to allow them to deal flexibly with delays either in transfer or those incurred while waiting to become bus master.

3. Once the DMA transfer is complete, the controller interrupts the processor, which can then determine by interrogating the DMA device or examining memory whether the entire operation completed successfully.

There may be multiple DMA devices in a computer system. For example, in a system with a single processor-memory bus and multiple I/O buses, each I/O bus controller will often contain a DMA processor that handles any transfers between a device on the I/O bus and the memory. Let's see how much of the processor is consumed using DMA to handle our hard-disk example.

Example

Overhead of I/O Using DMA

Suppose we have the same processor and hard disk as our earlier example on page 676. Assume that the initial setup of a DMA transfer takes 1000 clock cycles for the processor, and assume the handling of the interrupt at DMA completion requires 500 clock cycles for the processor. The hard disk has a transfer rate of 4 MB/sec and uses DMA. If the average transfer from the disk is 8 KB, what fraction of the 500-MHz processor is consumed if the disk is actively transferring 100% of the time? Ignore any impact from bus contention between the processor and DMA controller.

Answer

Each DMA transfer takes

$$\frac{8 \text{ KB}}{4 \frac{\text{MB}}{\text{second}}} = 2 \times 10^{-3} \text{ seconds}$$

So if the disk is constantly transferring, it requires

$$\frac{1000 + 500 \frac{\text{cycles}}{\text{transfer}}}{2 \times 10^{-3} \frac{\text{seconds}}{\text{transfer}}} = 750 \times 10^3 \frac{\text{clock cycles}}{\text{second}}$$

Since the processor runs at 500 MHz,

$$\text{Fraction of processor consumed} = \frac{750 \times 10^3}{500 \times 10^6} = 1.5 \times 10^{-3} = 0.2\%$$

Unlike either polling or interrupt-driven I/O, DMA can be used to interface a hard disk without consuming all the processor cycles for a single I/O. In addition, the disk will not be actively transferring data most of the time, and this number will be considerably lower. Of course, if the processor is also contending for memory, it will be delayed when the memory is busy doing a DMA transfer. By using caches, the processor can avoid having to access memory most of the time, thereby leaving most of the memory bandwidth free for use by I/O devices.

Elaboration: To further reduce the need to interrupt the processor and occupy it in handling an I/O request that may involve doing several actual operations, the I/O controller can be made more intelligent. Intelligent controllers are often called *I/O processors* (as well as *I/O controllers* or *channel controllers*). These specialized processors basically execute a series of I/O operations, called an *I/O program*. The program may be stored in the I/O processor, or it may be stored in memory and fetched by the I/O processor. When using an I/O processor, the operating system typically sets up an I/O program that indicates the I/O operations to be done as well as the size and transfer address for any reads or writes. The I/O processor then takes the operations from the I/O program and interrupts the processor only when the entire program is completed. DMA processors are essentially special-purpose processors (usually single-chip and nonprogrammable), while I/O processors are often implemented with general-purpose microprocessors, which run a specialized I/O program.

Direct Memory Access and the Memory System

When DMA is incorporated into an I/O system, the relationship between the memory system and processor changes. Without DMA, all accesses to the memory system come from the processor and thus proceed through address translation and cache access as if the processor generated the references. With DMA, there is another path to the memory system—one that does not go through the address translation mechanism or the cache hierarchy. This difference generates some problems in both virtual memory systems and systems with caches. These problems are usually solved with a combination of hardware techniques and software support.

The difficulties in having DMA in a virtual memory system arise because pages have both a physical and a virtual address. DMA also creates problems for systems with caches because there can be two copies of a data item: one in the cache and one in memory. Because the DMA processor issues memory requests directly to the memory rather than through the cache, the value of a memory location seen by the DMA unit and the processor may differ. Consider a read from disk that the DMA unit places directly into memory. If some of the locations into which the DMA writes are in the cache, the processor will receive

Hardware Software Interface

In a system with virtual memory, should DMA work with virtual addresses or physical addresses? The obvious problem with virtual addresses is that the DMA unit will need to translate the virtual addresses to physical addresses. The major problem with the use of a physical address in a DMA transfer is that the transfer cannot easily cross a page boundary. If an I/O request crossed a page boundary, then the memory locations to which it was being transferred would not be contiguous in the physical memory—the memory locations would correspond to multiple virtual pages, each of which could be mapped to any physical page. Consequently, if we use physical addresses, we must constrain all DMA transfers to stay within one page.

One method to allow the system to initiate DMA transfers that cross page boundaries is to make the DMA work on virtual addresses. In such a system, the DMA unit has a small number of map entries that provide virtual-to-physical mapping for a transfer. The operating system provides the mapping when the I/O is initiated. By using this mapping, the DMA unit need not worry about the location of the virtual pages involved in the transfer.

Another technique is for the operating system to break the DMA transfer into a series of transfers, each confined within a single physical page. The transfers are then *chained* together and handed to an I/O processor or intelligent DMA unit that executes the entire sequence of transfers; alternatively, the operating system can individually request the transfers.

Whichever method is used, the operating system must still cooperate by not remapping pages while a DMA transfer involving that page is in progress.

the old value when it does a read. Similarly, if the cache is write-back, the DMA may read a value directly from memory when a newer value is in the cache, and the value has not been written back. This is called the *stale data problem* or *coherency problem*.

We have looked at three different methods for transferring data between an I/O device and memory. In moving from polling to an interrupt-driven to a DMA interface, we shift the burden for managing an I/O operation from the processor to a progressively more intelligent I/O controller. These methods have the advantage of freeing up processor cycles. Their disadvantage is that they increase the cost of the I/O system. Because of this, a given computer system can choose which point along this spectrum is appropriate for the I/O devices connected to it.

Hardware Software Interface

The coherency problem for I/O data is avoided by using one of three major techniques. One approach is to route the I/O activity through the cache. This ensures that reads see the latest value while writes update any data in the cache. Routing all I/O through the cache is expensive and potentially has a large negative performance impact on the processor, since the I/O data is rarely used immediately and may displace useful data that a running program needs. A second choice is to have the OS selectively invalidate the cache for an I/O read or force write-backs to occur for an I/O write (often called cache *flushing*). This approach requires some small amount of hardware support and is probably more efficient if the software can perform the function easily and efficiently. Because this flushing of large parts of the cache need only happen on DMA block accesses, it will be relatively infrequent. The third approach is to provide a hardware mechanism for selectively flushing (or invalidating) cache entries. Hardware invalidation to ensure cache coherence is typical in multiprocessor systems, and the same technique can be used for I/O; we discuss this topic in detail in Chapter 9.

8.6 Designing an I/O System

There are two primary types of specifications that designers encounter in I/O systems: latency constraints and bandwidth constraints. In both cases, knowledge of the traffic pattern affects the design and analysis.

Latency constraints involve ensuring that the latency to complete an I/O operation is bounded by a certain amount. In the simple case, the system may be unloaded, and the designer must ensure that some latency bound is met either because it is critical to the application or because the device must receive certain guaranteed service to prevent errors. Examples of the latter are similar to the analysis we looked at in the previous section. Likewise, determining the latency on an unloaded system is relatively easy, since it involves tracing the path of the I/O operation and summing the individual latencies.

Finding the average latency (or distribution of latency) under a load is a much more complex problem. Such problems are tackled either by queuing theory (when the behavior of the workload requests and I/O service times can be approximated by simple distributions) or by simulation (when the behavior of I/O events is complex). Both topics are beyond the limits of this text.

Designing an I/O system to meet a set of bandwidth constraints given a workload is the other typical problem designers face. Alternatively, the designer may be given a partially configured I/O system and be asked to balance the system to maintain the maximum bandwidth achievable as dictated by the preconfigured portion of the system. This latter design problem is a simplified version of the first.

The general approach to designing such a system is as follows:

1. Find the weakest link in the I/O system, which is the component in the I/O path that will constrain the design. Depending on the workload, this component can be anywhere, including the CPU, the memory system, the backplane bus, the I/O controllers, or the devices. Both the workload and configuration limits may dictate where the weakest link is located.
2. Configure this component to sustain the required bandwidth.
3. Determine the requirements for the rest of the system and configure them to support this bandwidth.

The easiest way to understand this methodology is with an example.

Example

I/O System Design

Consider the following computer system:

- A CPU that sustains 300 million instructions per second and averages 50,000 instructions in the operating system per I/O operation
- A memory backplane bus capable of sustaining a transfer rate of 100 MB/sec
- SCSI-2 controllers with a transfer rate of 20 MB/sec and accommodating up to seven disks
- Disk drives with a read/write bandwidth of 5 MB/sec and an average seek plus rotational latency of 10 ms

If the workload consists of 64-KB reads (where the block is sequential on a track) and the user program needs 100,000 instructions per I/O operation, find the maximum sustainable I/O rate and the number of disks and SCSI controllers required. Assume that the reads can always be done on an idle disk if one exists (i.e., ignore disk conflicts).

Answer

The two fixed components of the system are the memory bus and the CPU. Let's first find the I/O rate that these two components can sustain and determine which of these is the bottleneck. Each I/O takes 100,000 user instructions and 50,000 OS instructions, so

$$\text{Maximum I/O rate of CPU} = \frac{\text{Instruction execution rate}}{\text{Instructions per I/O}} = \frac{300 \times 10^6}{(50 + 100) \times 10^3} = 2000 \frac{\text{I/Os}}{\text{second}}$$

Each I/O transfers 64 KB, so

$$\text{Maximum I/O rate of bus} = \frac{\text{Bus bandwidth}}{\text{Bytes per I/O}} = \frac{100 \times 10^6}{64 \times 10^3} = 1562 \frac{\text{I/Os}}{\text{second}}$$

The bus is the bottleneck, so we can now configure the rest of the system to perform at the level dictated by the bus, 1562 I/Os per second.

Now, let's determine how many disks we need to be able to accommodate 1562 I/Os per second. To find the number of disks, we first find the time per I/O operation at the disk:

$$\text{Time per I/O at disk} = \text{Seek/rotational time} + \text{Transfer time}$$

$$= 10 \text{ ms} + \frac{64 \text{ KB}}{5 \text{ MB/sec}} = 22.8 \text{ ms}$$

This means each disk can complete 43.9 I/Os per second. To saturate the bus requires 1562 I/Os per second, or $1562/43.9 \approx 36$ disks.

To compute the number of SCSI buses, we need to know the average transfer rate per disk, which is given by

$$\text{Transfer rate} = \frac{\text{Transfer size}}{\text{Transfer time}} = \frac{64\text{KB}}{22.8 \text{ ms}} \approx 2.74 \text{ MB/sec}$$

Assuming the disk accesses are not clustered so that we can use all the bus bandwidth, we can place seven disks per SCSI bus and controller. This means we will need 36/7, or six buses and controllers.

Notice the significant number of simplifying assumptions that are needed to do this example. In practice, many of these simplifications might not hold for critical I/O-intensive applications (such as databases). For this reason, simulation is often the only realistic way to predict the I/O performance of a realistic workload.

8.7 Real Stuff: A Typical Desktop I/O System

The emergence of two dominant standards in the desktop personal computer market has led to an enormous degree of commonality among I/O systems. These two standards are PCI, as a backplane bus, and SCSI or SCSI-2, as an I/O bus. Although systems with older buses (ISA or IDE) continue to ship, such systems have rapidly been replaced on all but the least-expensive, lowest-performance machines. Interestingly, the benefits of a single bus standard, in terms of greater availability of devices and lower cost, have led to the adoption of backplane and I/O bus standards across both the IBM-compatible and Macintosh platforms, and a larger fraction of workstation vendors are also adhering to these standards.

Figure 8.16 shows the I/O system of the Macintosh 7200 series, which is typical of the I/O system of midrange to high-end desktop machines in 1997. PCI is used as the backplane bus, with slower devices sharing a lower-performance bus, such as SCSI.

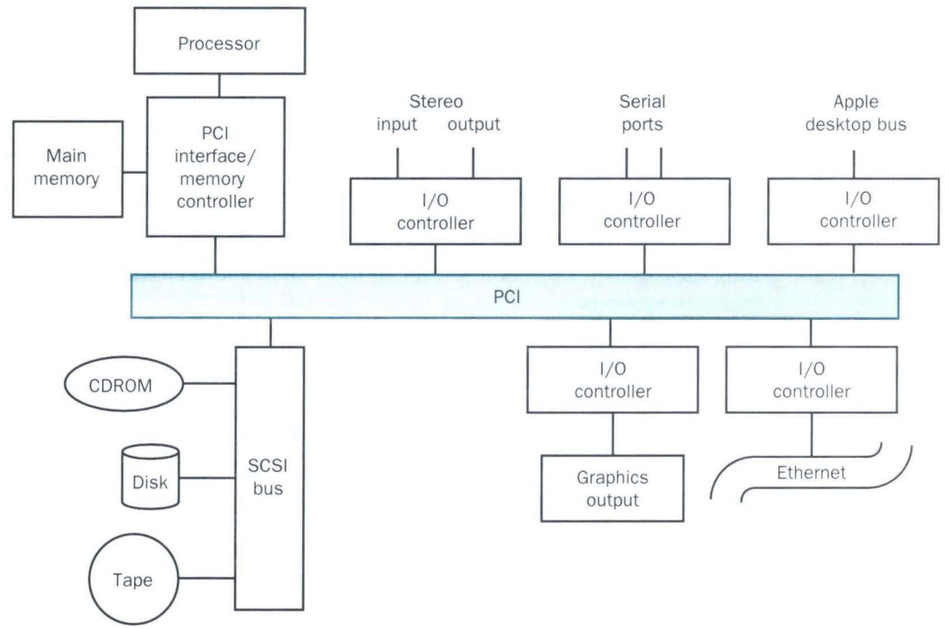


FIGURE 8.16 Organization of the I/O system on the Apple Macintosh 7200 series. The PCI backplane bus is used to interface all devices and interfaces to the processor and memory system. Serial ports provide for connections such as low-speed Appletalk network. The desktop bus provides support for keyboards and mice. In reality, several of the slow I/O devices (audio I/O, serial ports, and the desktop bus) share a single port onto the PCI bus, but we show them separately for simplicity.

8.8 Fallacies and Pitfalls

Fallacy: A 100-MB/sec bus can transfer 100 MB of data in 1 second.

Of course, this is only a fallacy when the definition of a megabyte of storage and a megabyte per second of bandwidth do not agree. As we discussed on page 642, I/O bandwidth measures are usually quoted in base 10 (i.e., 1 MB/sec = 10^6 bytes/sec), while 1 MB of data is typically a base 2 measure (i.e., 1 MB = 2^{20} bytes). How significant is this distinction? The time to transfer 100 MB of data on a 100-MB/sec bus is actually

$$\frac{100 \times 2^{20}}{100 \times 10^6} = \frac{1{,}048{,}576}{1{,}000{,}000} = 1.048576 \approx 1 \text{ second}$$

A similar, but smaller, error is introduced when we treat a kilobyte, meaning either 10^3 or 2^{10} bytes, as equivalent, while a larger error is introduced when we treat a gigabyte, meaning either 10^9 or 2^{30} bytes, as equivalent.

Pitfall: Using the peak transfer rate of a portion of the I/O system to make performance projections or performance comparisons.

Many of the components of an I/O system, from the devices to the controllers to the buses, are specified using their peak bandwidths. In practice, these peak bandwidth measurements are often based on unrealistic assumptions about the system or are unattainable because of other system limitations. For example, in quoting bus performance, the peak transfer rate is often specified using a memory system that is impossible to build.

A PCI bus has a peak bandwidth of about 133 MB/sec. In practice, even for long transfers, it is difficult to sustain more than about 80 MB/sec for realistic memory systems.

Amdahl's law also reminds us that the throughput of an I/O system will be limited by the lowest-performance component in the I/O path.

Fallacy: Magnetic storage is on its last legs and will be replaced shortly.

This is both a fallacy and a pitfall. Such claims have been made constantly for the past 20 years, though the string of failed alternatives in recent years seems to have reduced the level of claims for the death of magnetic storage. Among the unsuccessful candidates proposed to replace magnetic storage have been magnetic bubble memories, optical storage, and photographic storage. None of these systems has matched the combination of characteristics that favor magnetic disks: nonvolatility, low cost, reasonable access time, and high reliability. Magnetic storage technology continues to improve at the same or faster pace it has sustained over the past 25 years. In fact, the rate of density improvement has increased in the last 10 years, and rotational speeds and seek times have also improved significantly in the past few years.

Pitfall: Moving functions from the CPU to the I/O processor, expecting to improve performance without a careful analysis.

There are many examples of this pitfall trapping people, although I/O processors, when properly used, can certainly enhance performance. A frequent instance of this fallacy is the use of intelligent I/O interfaces, which, because of the higher overhead to set up an I/O, can turn out to have worse latency than a processor-directed I/O activity (although if the processor is freed up sufficiently, system throughput may still increase). Frequently, performance falls when the I/O processor has much lower performance than the main processor. Consequently, a small amount of main processor time is replaced with a larger amount of I/O processor time. Workstation designers have seen both these phenomena repeatedly.

A more serious problem can occur when the migration of an I/O feature changes the instruction set architecture or system architecture in a programmer-visible way. This forces all future machines to have to live with a decision that made sense in the past. If CPUs improve in cost/performance more rapidly than the I/O processor (and this will likely be the case), then moving the function may result in a slower machine in the next computer.

The most telling example comes from the IBM 360. It was decided that the performance of the ISAM system, an early database system, would improve if some of the record searching occurred in the disk controller itself. A key field was associated with each record, and the device searched each key as the disk rotated until it found a match. It would then transfer the desired record. This technique requires an extra large gap between records when a key is present.

The speed at which a track can be searched is limited by the speed of the disk and by the number of keys that can be packed on a track. On an IBM 3330 disk, the key is typically 10 characters; the gap is equivalent to 191 characters if there is a key, and 135 characters when no key is present. If we assume that the data is also 10 characters and that the track has nothing else on it, a 13,165-byte track can contain

$$\frac{13{,}165}{191 + 10 + 10} = 62 \text{ key-data records}$$

The time per key search is

$$\frac{16.7 \text{ ms (1 revolution)}}{62} = 0.27 \text{ ms/key search}$$

In place of this scheme, we could put several key-data pairs in a single block and have smaller interrecord gaps. Assuming that there are 15 key-data pairs per block and that the track has nothing else on it, then

$$\frac{13{,}165}{135 + 15 \times (10 + 10)} = \frac{13{,}165}{135 + 300} = 30 \text{ blocks of key-data pairs}$$

The revised performance is then

$$\frac{16.7 \text{ ms (1 revolution)}}{30 \times 15} \approx 0.04 \text{ ms/key search}$$

Of course, the disk-based search would look better if the keys were much longer.

As processors got faster, the CPU time for a search became trivial, while the time for a search using the hardware facility improved very little. While the strategy made early machines faster, programs that use the key search operation in the I/O processor run up to six times slower on today's machines!

8.9 Concluding Remarks

I/O systems are evaluated on several different characteristics: the variety of I/O devices supported; the maximum number of I/O devices; cost; and performance, measured both in latency and in throughput. These goals lead to widely varying schemes for interfacing I/O devices. In the low end, schemes like buffering and even DMA can be avoided to minimize cost. In midrange systems, buffered DMA is likely to be the dominant transfer mechanism. In the high end, latency and bandwidth may both be important, and cost may be secondary. Multiple paths to I/O devices with limited buffering often characterize high-end I/O systems. Increasing the bandwidth with both more and wider connections eliminates the need for buffering at an increase in cost. Typically, being able to access the data on an I/O device at any time (high availability) becomes more important as systems grow. As a result, redundancy and error correction mechanisms become more and more prevalent as we enlarge the system.

The design of I/O systems is complicated because the limiting factor in I/O system performance can be any of several critical resources in the I/O path, from the operating system to the device. Furthermore, independent requests from different programs interact in the I/O system, making the performance of an I/O request dependent on other activity that occurs at the same time. Lastly, design techniques that improve bandwidth often negatively impact latency, and vice versa. For example, adding buffering usually increases the system cost and also the system bandwidth. But it also increases latency by placing additional hardware between the device and memory. It is this combination of factors, including some that are unpredictable, that makes designing I/O systems and improving their performance challenging not only for architects but also for OS designers and even programmers building I/O-intensive applications.

The Big Picture

The performance of an I/O system, whether measured by bandwidth or latency, depends on all the elements in the path between the device and memory, including the operating system that generates the I/O commands. The bandwidth of the buses, the memory, and the device determine the maximum transfer rate from or to the device. Similarly, the latency depends on the device latency, together with any latency imposed by the memory system or buses. The effective bandwidth and response latency also depend on other I/O requests that may cause contention for some resource in the path. Finally, the operating system is a bottleneck. In some cases, the OS takes a long time to deliver an I/O request from a user program to an I/O device, leading to high latency. In other cases, the operating system effectively limits the I/O bandwidth because of limitations in the number of concurrent I/O operations it can support.

Future Directions in I/O Systems

What does the future hold for I/O systems? The rapidly increasing performance of processors strains I/O systems, whose physical components cannot improve in performance as fast as processors. To hide the growing gap between the speed of processors and the access time to secondary storage (primarily disks), main memory is used as a cache for secondary storage. These *file caches*, which rely on spatial and temporal locality in access to secondary storage, are maintained by the operating system. The use of file caches allows many file accesses to be handled from memory rather than from disk.

Magnetic disks are increasing in capacity quickly, but access time is improving only slowly. One reason for this is that the opportunities for magnetic disks are growing faster in the low end of the market than in the high end, and the low end is driven primarily by the demand for lower cost per megabyte. This market has helped shrink the size of the disk from the 14-inch platters of the mainframe disk to the 1.3-inch disks developed for laptop and palmtop computers. In fact, the dramatic demand for small disks has led to an accelerated rate of improvement in disk density, so that the density of magnetic disks has been growing faster since about 1990 than it ever did! What is surprising is that this period of growth came at a time when a number of people were predicting the end (or at least a reduction in the use) of magnetic disks!

In addition to increases in density, transfer rates have grown rapidly as disks increased in rotational speed and interfaces improved. In addition, virtually every high-performance disk manufactured today includes a track or sector buffer that caches sectors as the read head passes over them.

One major new disk organization that has emerged in the last few years is an array of small and inexpensive disks. The argument for arrays is that since price per megabyte is independent of disk size, potential throughput can be increased by having many disk drives and, hence, many disk heads. Simply spreading data over multiple disks automatically forces accesses to several disks. (While arrays improve throughput, latency is not necessarily reduced.) Adding redundant disks to the array offers the opportunity for the array to discover a failed disk and automatically recover the lost information. Arrays may thus enhance the reliability of a computer system as well as performance. This redundancy has inspired the acronym *RAID* for these arrays: *redundant arrays of inexpensive disks*. A number of computer companies offer RAIDs for their disk subsystems. For example, IBM has both a RAID offering (see the IBM link at *www.mkp.com/books_catalog/cod/links.htm*), as well as a disk subsystem built from the largest disks they manufacture. Ironically, RAID controllers turned out to be quite expensive, so the original meaning of the RAID acronym seemed inappropriate. The acronym was recast as "Redundant Arrays of Intelligent Disks."

The next level of the storage hierarchy below magnetic disks has also yielded extraordinary increases in capacity in the last several years. This increase has come partly from improvements in magnetic recording that also helped disks, but also from a different recording technology, the *helical scan tape*. Found in VCRs, camcorders, and digital audio tapes, helical scan tape records at an angle to the tape rather than parallel, as in longitudinally recorded tapes. The tape still moves at the same speed, but the fast-spinning tape head records bits much more densely—a factor of about 50 to 100 denser than longitudinally recorded tapes. And because the medium was created for consumer products, the improvement in cost per bit over time has been even greater than for traditional magnetic tapes used solely by the computer industry.

Advances in tape capacity are being enhanced by advances on two other fronts: compression and robots. Faster processors have enabled systems to begin using compression to multiply storage capacity. Factors of two to three are common, with compression of 20:1 possible for certain types of data such as images. The second enhancement that is changing the cost-effectiveness of very large online storage is the emergence of inexpensive robots to automatically load and store tapes, offering a new level in the hierarchy between *online* magnetic disks and *offline* magnetic disks on shelves. This "*robo-line*" storage means access to terabytes of information at the delay of tens of seconds, without the intervention of a human operator. Figure 8.17 is a photograph of a tape robot.

Computer networks are also making great strides. Both 100-Mbit Ethernet and switched Ethernet solutions are being used in new networks and in upgrading networks that cannot handle the tremendous explosion in bandwidth created by the use of multimedia and the growing importance of the World Wide Web. ATM represents another potential technology for expanding even further. To support the growth in traffic, the Internet backbones are being switched to optical fiber, which allows a significant increase in bandwidth for long-haul networks.

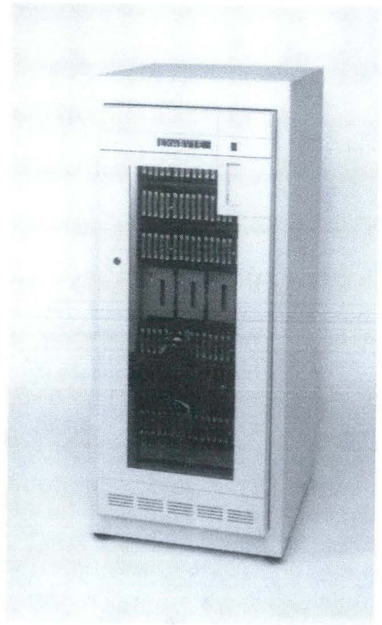


FIGURE 8.17 The Exabyte EXB-120 holds 116 8-mm helical scan tapes. Each tape holds 10 GB, yielding a total capacity of over a terabyte. Photo courtesy of the Exabyte Corporation.

One of the most interesting storage technologies being explored is holography. One research project under way hopes to demonstrate a storage device with terabyte capacity and with transfer rates of 1 Gbit/sec. This would represent about an order of magnitude improvement in both storage size and transfer rate versus the largest disks in 1997. See the pertinent IBM link at *www.mkp.com/books_catalog/cod/links.htm* for a description of this joint academic-industry research activity.

Such advances offer "computing science fiction" scenarios that would have seemed absurd just a few years ago. For example, if all the books in the Library of Congress were converted to ASCII, they would occupy just 10 terabytes (although the pictures might take even more, depending on their number and resolution). Helical scan tapes, tape robots, compression, and high-speed networks could be the building blocks of an electronic library. All the information on all the books in the world would be available at your fingertips for the cost of a large minicomputer. And parallel processing, discussed in the next chapter, will allow this information to be indexed so that all books could be searched by content rather than by title. Electronic libraries would change the lives of anyone with a library card, and the technology to create them is within our grasp.

8.10 Historical Perspective and Further Reading

The history of I/O systems is a fascinating one. Many of the most interesting artifacts of early computers are their I/O devices. Magnetic tape was the first low-cost magnetic storage and today persists as the lowest-cost storage medium. Early tape drives used reel-to-reel technologies and linear recording, which were eventually replaced by tape cartridges and helical recording. As disks became cheaper, tapes were relegated primarily to archival purposes, causing additional focus on density, as opposed to speed, and on large-scale archival technologies such as tape robots.

The earliest random access storage devices were drums and fixed-head disks. A drum had a cylindrical surface coated with a magnetic film. It used a large number of read/write heads positioned over each track on the drum (see Figure 8.18). Drums were relatively high-speed I/O devices often used for virtual memory paging or for creating a file cache to slower-speed devices. Drums, which had no seek time, survived into the 1970s in higher-speed applications, such as paging or use in high-end machines. Eventually improvements in disk speed and the significant cost advantage of disks eliminated drum technology. Large (2 to 3 feet in diameter) single-platter, fixed-head disks were also in use in the 1950s.

In 1956, IBM developed the first disk storage system with both moving heads and multiple disk surfaces in San Jose, helping to seed the development of the magnetic storage industry in the southern end of Silicon Valley. The IBM 305 RAMAC (Random Access Method of Accounting and Control) could store 5 million characters (5 MB) of data on 50 disks, each 24 inches in diameter. The RAMAC is shown in Figure 8.19.

Moving-head disks quickly became the dominant high-speed magnetic storage, though their high cost meant that magnetic tape continued to be used extensively until the 1970s. The next key development for hard disks was the removable hard disk drive developed by IBM in 1962; this made it possible to share the expensive drive electronics and helped disks overtake tapes as the preferred storage medium. Figure 8.20 shows a removable disk drive and the multiplatter disk used in the drive. IBM also invented the floppy disk drive in 1970, originally to hold microcode for the IBM 370 series. Floppy disks became popular with the PC about 10 years later.

The sealed Winchester disk, which was developed by IBM in 1973, completely dominates disk technology today. (All the disks shown in Figure 8.5 on page 649 are Winchester disks.) Winchester disks benefited from two related properties. First, reductions in the cost of the disk electronics made it unnecessary to share the electronics and thus made nonremovable disks economical.

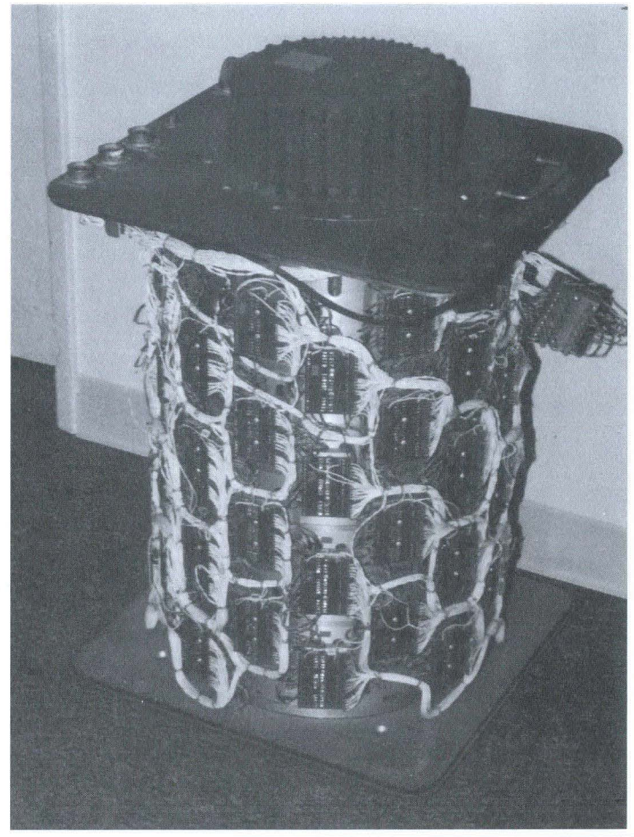


FIGURE 8.18 A magnetic drum made by Digital Development Corporation in the 1960s and used on a CDC machine. The electronics supporting the read/write heads can be seen on the outside of the drum. Photo courtesy of the Computer Museum of America.

Since the disk was fixed and could be in a sealed enclosure, both the environmental and control problems were greatly reduced, allowing significant gains in density. The first disk that IBM shipped had two spindles, each with a 30-MB disk; the moniker "30-30" for the disk led to the name Winchester. Winchester disks grew rapidly in popularity in the 1980s, completely replacing removable disks by the middle of that decade.

Recently, low-cost removable drives have been resurrected for use in backup and portable locations. These drives typically are available both in floppy

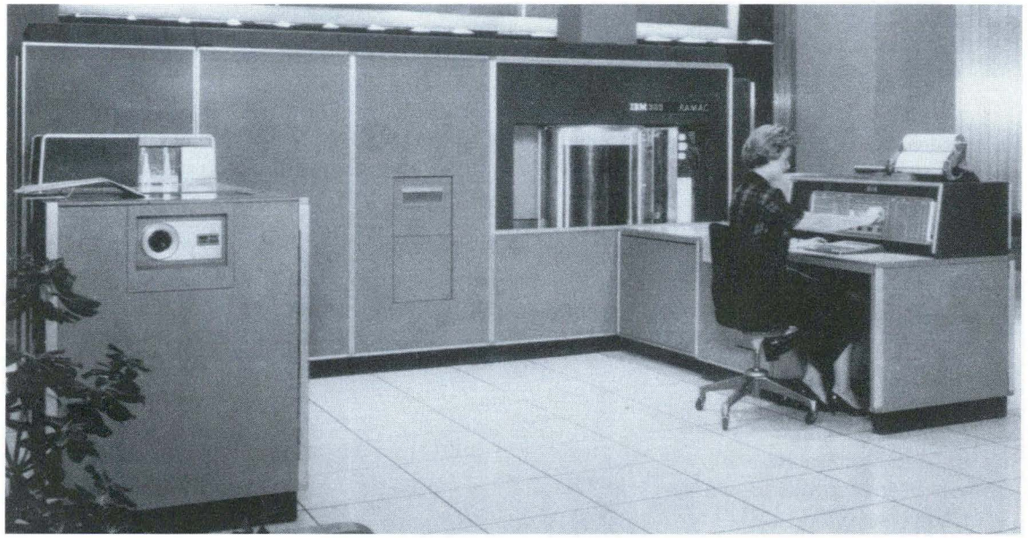


FIGURE 8.19 The RAMAC disk drive from IBM, made in 1956, was the first disk drive with a moving head and the first with multiple platters. The IBM storage technology Web site has a discussion of IBM's major contributions to storage technology. Find the link at *www.mkp.com/books_catalog/cod/links.htm*. Photo courtesy of IBM.

media, storing about 100 MB in 1997, and a removable hard disk format, storing 1–2 GB. These removable disks have lower density and are slower than nonremovable disks, but the removable media are attractive for certain environments.

The 1970s saw the invention of a number of remarkable I/O devices. Perhaps one of the most unusual was a film storage device that stored data optically on small strips of photographic film. These film storage devices could not only read and write film, but actually kept the filmstrips stored in the device (which was about 5 feet by 4 feet by 3 feet), retrieving them mechanically.

The early IBM 360s pioneered many of the ideas that we use in I/O systems today. The 360 was the first machine to make heavy use of DMA, and it introduced the notion of I/O programs that could be interpreted by the device. Chaining of I/O programs was a key feature. The concept of channels introduced in the 360 corresponds to the I/O bus of today.

The trend for high-end machines has been toward use of programmable I/O processors. The original machine to use this concept was the CDC 6600, which used I/O processors called *peripheral processors*.

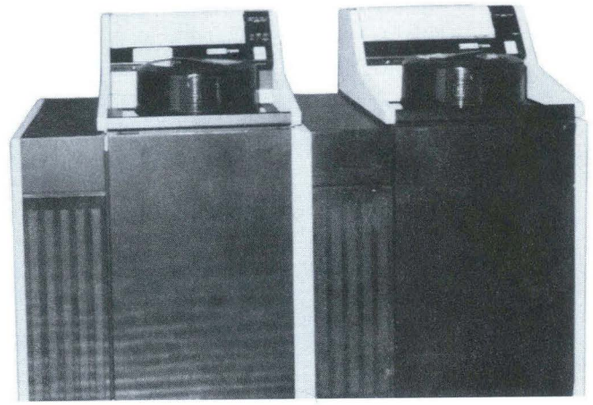


FIGURE 8.20 This is a DEC disk drive and the removable pack. These disks became popular starting in the mid-1960s and dominated disk technology until Winchester drives in the late 1970s. This drive was made in the mid-1970s; each disk pack in this drive could hold 80 MB. Photo courtesy of the Commercial Computing Museum.

The forerunner of today's workstations and personal computers was the Alto, developed at Xerox Palo Alto Research Center in 1973 [Thacker et al. 1982], shown in Figure 8.21. This machine integrated the needs of the I/O functions into the microcode of the processor. This included support for the bit-mapped graphics display, the disk, and the network. The network for the Alto was the first Ethernet [Metcalfe and Boggs 1976]. The Alto also supported the first laser printer, configured as a print server accessible over the Ethernet. Similarly, disk servers were also built. The mouse, invented earlier by Doug Engelbart of SRI, was a key part of the Alto. The 16-bit processor used a writable control store, which enabled researchers to program in support for the I/O devices. The single microprogrammed engine drove the graphics display, mouse, disks, network, and, when there was nothing else to do, ran the user's program.

While today we associate microprocessors with the personal computer revolution, they were originally developed to meet the demand for special-purpose controllers. Since the invention of the microprocessor, designers have developed many I/O controllers that adapt a microprocessor to a specific task. These include everything from DMA controllers to SCSI controllers to complete Ethernet controllers on a single chip.

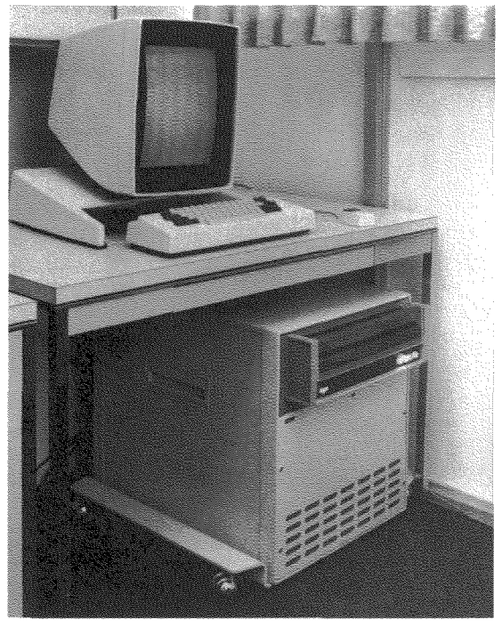


FIGURE 8.21 The Xerox Alto. Although never sold as a product, Xerox donated a number of these machines to several major universities as well as using them heavily internally. The use of a mouse, a local area network, and a personal graphics display with a window system were key characteristics of the Alto later broadly adopted by workstation and PC companies. Photo courtesy of the Computer History Center.

The first multivendor bus may have been the PDP-11 Unibus in 1970. DEC encouraged other companies to build devices that would plug into its bus, and many companies did. A more recent example is SCSI (small computer systems interface). This bus, originally called SASI, was invented by Shugart and was later standardized by the IEEE. This open system approach to buses contrasts with proprietary buses using patented interfaces, which companies adopt to forestall competition from plug-compatible vendors. The use of proprietary buses also raises the costs and lowers the availability of I/O devices that plug into proprietary buses because such devices must have an interface designed exclusively for that bus.

Ongoing development in the areas of tape robots (see Figure 8.17 on page 693), head-mounted displays, gloves for complete tactile feedback, and computer screens that you write on with pens are indications that the incredible developments in I/O technology are likely to continue in the future.

To Probe Further

Bashe, C. J., L. R. Johnson, J. H. Palmer, and E. W. Pugh [1986]. *IBM's Early Computers*, MIT Press, Cambridge, MA.

Describes the I/O system architecture and devices in IBM's early computers.

Borrill, P. L. [1986]. "32-bit buses: An objective comparison," *Proc. Buscon 1986 West*, San Jose, CA, 138–45.

A comparison of various 32-bit bus standards.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson [1994]. "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys* 26:2 (June) 145–88.

A tutorial covering disk arrays and the advantages of such an organization.

Gray, J., and A. Reuter [1993]. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco.

A description of transaction processing, including discussions of benchmarking and performance evaluation.

Hennessy, J., and D. Patterson [1995]. *Computer Architecture: A Quantitative Approach*, Second edition, Morgan Kaufmann Publishers, San Francisco, Chapters 6 and 7.

Chapter 6 focuses on I/O devices, including an extensive discussion of RAID technologies and more accurate I/O performance modeling. Chapter 7 focuses on interconnection technologies, including buses and an extensive discussion on networking.

Kahn, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November) 1397–1407.

A classic paper that describes the ARPANET.

Levy, J. V. [1978]. "Buses: The skeleton of computer structures," in *Computer Engineering: A DEC View of Hardware Systems Design*, C. G. Bell, J. C. Mudge, and J. E. McNamara, eds., Digital Press, Bedford, MA.

This is a good overview of key concepts in bus design with some examples from DEC machines.

Metcalfe, R. M., and D. R. Boggs [1976]. "Ethernet: Distributed packet switching for local computer networks," *Comm. ACM* 19:7 (July) 395–404.

Describes the Ethernet network.

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September) 5–15.

Describes the development of important ideas in I/O.

Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs [1982]. "Alto: A personal computer," in *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, eds., McGraw-Hill, New York, 549–72.

Describes the Alto—forerunner of workstations as well as the Apple Macintosh.

8.11 Key Terms

The wide variety of characteristics present in different I/O devices and the corresponding system techniques for adapting to those devices have introduced a number of new terms, summarized below.

asynchronous bus
backplane bus
bus arbitration
bus master
bus request
bus transaction
centralized, parallel arbitration
daisy chain arbitration
direct memory access (DMA)
distributed arbitration by collision detection
distributed arbitration by self-selection
Ethernet
fairness
handshaking protocol
I/O instruction
interrupt-driven I/O
memory-mapped I/O
polling
processor-memory buses
redundant arrays of inexpensive disks (RAID)
rotation latency or delay
sector
seek
slave
small computer systems interface (SCSI)
split transaction protocol
synchronous bus
track
transaction processing
transfer time

8.12 Exercises

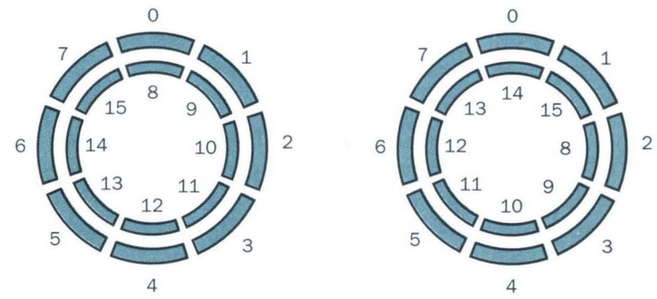
8.1 [10] <§§8.1–8.2> Here are two different I/O systems intended for use in transaction processing:

- System A can support 1000 I/O operations per second.
- System B can support 750 I/O operations per second.

The systems use the same processor that executes 50 million instructions per second. Assume that each transaction requires 5 I/O operations and that each I/O operation requires 10,000 instructions. Ignoring response time and assuming that transactions may be arbitrarily overlapped, what is the maximum transaction-per-second rate that each machine can sustain?

8.2 [15] <§§8.1–8.2> {Ex. 8.1} The latency of an I/O operation for the two systems in Exercise 8.1 differs. The latency for an I/O on system A is equal to 20 ms, while for system B the latency is 18 ms for the first 500 I/Os per second and 25 ms per I/O for each I/O between 500 and 750 I/Os per second. In the workload, every 10th transaction depends on the immediately preceding transaction and must wait for its completion. What is the maximum transaction rate that still allows every transaction to complete in 1 second and that does not exceed the I/O bandwidth of the machine? (For simplicity, assume that all transaction requests arrive at the beginning of a 1-second interval.)

8.3 [5] <§8.3> The following simplified diagram shows two potential ways of numbering the sectors of data on a disk (only two tracks are shown and each track has eight sectors). Assuming that typical reads are contiguous (e.g., all 16 sectors are read in order), which way of numbering the sectors will be likely to result in higher performance? Why?



8.4 [5] <§8.3> What size messages would result in ATM outperforming Ethernet by a factor of two, assuming latencies and bandwidths equivalent to those reported in the example on page 654?

8.5 [5] <§8.3> The speed of light is approximately 3×10^8 meters per second, and electrical signals travel at about 50% of this speed in a conductor. When the term *high speed* is applied to a network, it is the bandwidth that is higher, not necessarily the velocity of the electrical signals. How much of a factor is the actual "flight time" for the electrical signals? Consider two computers that are 100 meters apart and two computers that are 5000 kilometers apart. Compare your results to the latencies reported in the example on page 654.

8.6 [5] <§8.3> The number of bytes in transit on a network is defined as the flight time (described in Exercise 8.5) multiplied by the delivered bandwidth. Calculate the number of bytes in transit for the two networks described in Exercise 8.5, assuming a delivered bandwidth of 5 MB/sec.

8.7 [5] <§8.3> A secret government agency simultaneously monitors 100 cellular phone conversations and multiplexes the data onto a network with a bandwidth of 1 MB/sec and an overhead latency of 350 µs per 1-KB message. Calculate the transmission time per message and determine whether there is

sufficient bandwidth to support this application. Assume that the phone conversation data consists of 2 bytes sampled at a rate of 4 KHz.

8.8 [10] <§8.3> A program repeatedly performs a three-step process: It reads in a 4-KB block of data from disk, does some processing on that data, and then writes out the result as another 4-KB block elsewhere on the disk. Each block is contiguous and randomly located on a single track on the disk. The disk drive rotates at 7200 RPM, has an average seek time of 8 ms, and has a transfer rate of 20 MB/sec. The controller overhead is 2 ms. No other program is using the disk or processor, and there is no overlapping of disk operation with processing. The processing step takes 20 million clock cycles, and the clock rate is 400 MHz. What is the overall speed of the system in blocks processed per second?

8.9 [10] <§8.3> A transaction processing system utilizes a network and two different message sizes. The transaction request is quite small and consists of a 10-byte message. The transaction response is larger and consists of a 150-byte message. Assume that every transaction consists of a request and a response. Determine which of the two networks described in the example on page 654 would be better for this system.

8.10 [5] <§§8.3, 8.4> Assume that the bus and memory systems described in the example on page 665 are used to handle disk accesses from disks like the one described in the example on page 648. If the I/O is allowed to consume 100% of the bus and memory bandwidth, what is the maximum number of simultaneous disk transfers that can be sustained for the two block sizes?

8.11 [5] <§8.4> The example on page 665 assumed that the memory system took 200 ns to read the first four words, and each additional four words required 20 ns. Redo the example with the assumption that the memory system takes 150 ns to read the first four words and 30 ns to read each additional four words.

8.12 [5] <§8.4> The example on page 665 demonstrates that using larger block sizes results in an increase in the maximum sustained bandwidth that can be achieved. Under what conditions might a designer tend to favor smaller block sizes? Specifically, why would a designer choose a block size of 4 instead of 16 (assuming all of the characteristics are as identified in the example)?

8.13 [15] <§8.4> This question examines in more detail how increasing the block size for bus transactions decreases the total latency required and increases the maximum sustainable bandwidth. In the example on page 665, two different block sizes are considered (4 words and 16 words). Compute the total latency and the maximum bandwidth for all of the possible block sizes (between 4 and 16) and plot your results. Summarize what you learn by looking at your graph.

8.14 [15] <§8.4> This exercise is similar to Exercise 8.13. This time fix the block size at 4 and 16 (as in the example on page 665), but compute latencies and bandwidths for reads of different sizes. Specifically, consider reads of from 4 to 256 words, and use as many data points as you need to construct a meaningful graph. Use your graph to help determine at what point block sizes of 16 result in a reduced latency when compared with block sizes of 4.

8.15 [10] <§8.4> This exercise examines a design alternative to the example on page 665 that may improve the performance of writes. For writes, assume all of the characteristics reported in the example as well as the following:

5. The first four words are written 200 ns after the address is available, and each new write takes 20 ns. Assume a bus transfer of the most recent data to write, and a write of the previous four words can be overlapped.

The performance analysis reported in the example would thus remain unchanged for writes (in actuality, some minor changes might exist due to the need to compute error correction codes, etc., but we'll ignore this). An alternative bus scheme relies on separate 32-bit address and data lines. This will permit an address and data to be transmitted in the same cycle. For this bus alternative, what will the latency of the entire 256-word transfer be? What is the sustained bandwidth? Consider block sizes of four and eight words. When do you think the alternative scheme would be heavily favored?

8.16 <20> <§8.4> Consider an asynchronous bus used to interface an I/O device to the memory system described in the example on page 665. Each I/O request asks for 16 words of data from the memory, which, along with the I/O device, has a 4-word bus. Assume the same type of handshaking protocol as appears in Figure 8.10 on page 661 except that it is extended so that the memory can continue the transaction by sending additional blocks of data until the transaction is complete. Modify Figure 8.10 (both the steps and diagram) to indicate how such a transfer might take place. Assuming that each handshaking step takes 20 ns and memory access takes 60 ns, how long does it take to complete a transfer? What is the maximum sustained bandwidth for this asynchronous bus, and how does it compare to the synchronous bus in the example?

8.17 [15] <§§8.3–8.6> Redo the example on page 685, but instead assume that the reads are random 4-KB reads. You can assume that the reads are always to an idle disk, if one is available.

8.18 [20] <§§8.3–8.6> Here are a variety of building blocks used in an I/O system that has a synchronous processor-memory bus running at 200 MHz and one or more I/O adapters that interface I/O buses to the processor-memory bus.

- *Memory system:* The memory system has a 32-bit interface and handles four-word transfers. The memory system has separate address and data lines and, for writes to memory, accepts a word every clock cycle for 4 clock cycles and then takes an additional 4 clock cycles before the words have been stored and it can accept another transaction.
- *DMA interfaces:* The I/O adapters use DMA to transfer the data between the I/O buses and the processor-memory bus. The DMA unit arbitrates for the processor-memory bus and sends/receives four-word blocks from/to the memory system. The DMA controller can accommodate up to eight disks. Initiating a new I/O operation (including the seek and access) takes 1 ms, during which another I/O cannot be initiated by this controller (but outstanding operations can be handled).
- *I/O bus:* The I/O bus is a synchronous bus with a sustainable bandwidth of 10 MB/sec; each transfer is one word long.
- *Disks:* The disks have a measured average seek plus rotational latency of 12 ms. The disks have a read/write bandwidth of 5 MB/sec, when they are transferring.

Find the time required to read a 16-KB sector from a disk to memory, assuming that this is the only activity on the bus.

8.19 [15] <§§8.3–8.5> {Ex. 8.18} For the I/O system described in Exercise 8.18, find the maximum instantaneous bandwidth at which data can be transferred from disk to memory using as many disks as needed; how many disks and I/O buses (the minimum of each) do you need to achieve the bandwidth? Since you need only achieve this bandwidth for an instant, latencies need not be considered.

8.20 [20] <§§8.3–8.5> {Ex. 8.18, 8.19} Assume all accesses in the I/O system described in Exercise 8.18 are 4-KB block reads. If there are a total of six I/O buses, six DMA controllers, and 48 disks, find the maximum number of I/Os the system can sustain in steady state assuming that the reads are uniformly distributed to the disks. What is the sustained I/O bandwidth?

8.21 [15] <§§8.3–8.5> {Ex. 8.18, 8.19, 8.20} With the organization in Exercise 8.20, clearly it is possible to saturate the I/O buses because you have six of them at 10 MB/sec and 48 disks at 5 MB/sec. Compute the minimum block size (which should be a power of two) that will saturate the I/O buses. For this block size, how many I/O operations per second can the system perform and what is the I/O bandwidth?

8.22 [15] <§§7.3, 7.5, 8.4, 8.5> Consider a write-back cache used for a processor with a bus and memory system as described in the example on page 665

(assume that writes require the same amount of time as reads). The following performance measurements have been made:

- The cache miss rate is .05 misses per instruction for block sizes of 8 words.
- The cache miss rate is .03 misses per instruction for block sizes of 16 words.
- For either block size, 40% of the misses require a write-back operation, while the other 60% require only a read.

Assuming that the processor is stalled for the duration of a miss (including the write-back time if a write-back is needed), find the number of cycles per instruction that are spent handling cache misses for each block size. (Hint: First compute the miss penalty.)

8.23 [10] <§8.6> Write a paragraph identifying some of the simplifying assumptions that were made in the analysis described in the example on page 681.

8.24 [2 days–1 week] <§8.5, Appendix A> This assignment uses SPIM to build a simple set of I/O routines that will perform I/O to the terminal using polling. First, you need to build two I/O routines, whose C declarations and descriptions are shown below:

```
void print (char *string);
```

The procedure `print` takes a single argument, which is the address of a null-terminated ASCII string. All of the characters of the string except the null-terminating character should be output by `print`. It should print the characters one at a time, waiting for each character to be output before sending the next one. It should not return until all the characters have been output. The procedure `print` should work for strings of any length. This version of `print` should not use interrupts; just test the ready bit of the transmitter control register continuously until the device is ready.

```
char getchar();
```

The procedure `getchar` takes no arguments and returns a character result. If `getchar` waits until a character has been typed on the terminal, then it should return the character's value in `$v0` (the result register). Do not use interrupts; simply test the ready bit continuously until a character has arrived.

Write a main program that uses these two procedures to read a line from the terminal, which will be terminated by a carriage return. Then print the entire line to the terminal, including a carriage return and line feed. All your code should obey the conventions in Appendix A for procedure calling, stack usage, and register usage.

8.25 [3 days–1 week] <§8.5, Appendix A> Your assignment is to build an interrupt-driven mechanism for buffered I/O to and from the terminal. (This exercise handles output only; Exercise 8.26 handles input.)

For the output-only portion, there are three parts to the program:

1. A main program, which repeatedly calls procedure `print` to print the string "`I know what I am doing.`"
2. The procedure `print`, which stores the output characters in a buffer shared by it and the interrupt routine.
3. The interrupt routine, which copies characters from the output buffer to the transmitter.

You need to write all three routines. The routine `print` and the interrupt routine should communicate by using a shared circular buffer with space for 32 characters. The `print` procedure should take a string as argument and add the characters of the string to the output buffer one at a time, advancing as soon as there is space in the buffer. Keep in mind that `print` should not manipulate the terminal device registers directly, except to make sure that transmitter interrupts are enabled. Furthermore, `print` should contain additional code to deal with a full output buffer. The main program generates characters much faster than they can be output, so the buffer will quickly fill up. In a real system, if the output buffer fills up, the operating system will stop running the current user's process and switch to a different process. Your program doesn't need to support multiple users, so `print` can take a simpler approach: it just checks the buffer over and over again until eventually it isn't full anymore. The buffer is full when the next position in which `print` wants to insert a character has not been emptied by the interrupt routine.

After writing print, write the interrupt routine called by `print`. Here is a list of things the interrupt routine must do:

1. If the transmitter is not ready, then the interrupt routine should not do anything. (You shouldn't have received an interrupt in the first place if the transmitter isn't ready, but it's a good idea to check anyway.)
2. If the output buffer isn't empty, copy the next character from the output buffer to the Transmitter data register and adjust the buffer pointers.
3. If the output buffer is empty, turn off the interrupt-enable bit in the Transmitter control register. Otherwise, continuous interrupts will occur. Each time it deposits a character in the buffer, `print` will need to turn this bit on.
4. Don't forget that you must save and restore any registers that you use in the interrupt routine, even temporary registers such as register `$t0` and register `$t1`. This is necessary because interrupts can occur at any time and those registers may be in use at the time of the interrupt. You must save the registers on the stack. The only exceptions to this rule are registers `$k0` and `$k1`, which are reserved for use by interrupt routines; these registers need not be saved and restored. One of these registers can be used to return from the interrupt routine back to the code that was interrupted.

Test your code by writing the main routine that calls `print` to print the string. It should output lines continuously, with each line containing the characters "`I know what I am doing.`"

8.26 [3 days–1 week] <§8.5, Appendix A> {Ex. 8.25} Extend the code you've already written to be able to handle interrupt-driven input. This program should do input in the same way as the previous program did output: by using a buffer to communicate between the routine `getchar` and the interrupt routine. Be aware that `getchar` returns a character from the buffer, waiting in a loop if no characters are present. Similarly, the interrupt routine will add characters as they are typed, discarding characters if the buffer is full when they arrive. For this, an eight-entry buffer should work well.

Use these two routines to read characters from the terminal and to output them to the terminal. Try typing characters rapidly to make sure your program can handle the output or the input buffer filling up. For example, if you type two or three characters rapidly, the output buffer may fill up. However, no output should be lost: the print procedure will simply have to spin for a bit, during which time additional input characters will be buffered in the input buffer. If you type eight or ten characters very rapidly, then the input buffer will probably fill up. When this happens, your interrupt routine will have to discard characters: the program should continue to function, but there won't be any output of the discarded input characters you typed. Once the output catches up with the input, your program should accept input again just as if the input buffer had never filled up.

8.27 [1 day–1 week] <§§8.2–8.4> Take your favorite computer and write programs that achieve the following:

1. Maximum bandwidth from and to a single disk
2. Maximum bandwidth from and to multiple disks
3. The maximum number of 512-byte transactions from and to a single disk
4. The maximum number of 512-byte transactions from and to multiple disks

What is the percentage of the bandwidth that you achieve compared to what the I/O device manufacturer claims? Also, record processor utilization in each case for the programs that are running separately. Next, run all four

together and see what percentage of the maximum rates you can achieve. From this, can you determine where the system bottlenecks lie?

In More Depth

Ethernet

An Ethernet is essentially a standard bus with multiple masters (each computer can be a master) and a distributed arbitration scheme using collision detection. Most Ethernets are implemented using coaxial cable as the medium. When a particular node wants to use the bus, it first checks to see whether some other node is using the bus; if not, it places a carrier signal on the bus, and then proceeds to transmit. A complication can arise because the control is distributed and the devices may be physically far apart. As a result, two nodes can both check the Ethernet, find it idle, and begin transmitting at once; this is called a *collision*. A node detects collisions by listening to the network when transmitting to see whether a collision has occurred. A collision is detected when the node finds that what it hears on the Ethernet differs from what it transmitted. When collisions occur, both nodes stop transmitting and delay a random time interval before trying to resume using the network—just as two polite people do when they both start talking at the same time. Consequently, the number of nodes on the network is limited—if too many collisions occur, the performance will be poor. In addition, constraints imposed by the requirement that collisions be detected by all nodes limit the length of the Ethernet and the number of connections to the network. Although this idea sounds like it might not work, it actually works amazingly well and has been central to the enormous growth in the use of local area networks.

8.28 [3 days–1 week] <§§8.3–8.4> Write a program that simulates an Ethernet. Assume the following network system characteristics:

- A transmission bandwidth of 10 Mbits/sec.
- A latency for a signal to travel the entire length of the network and return to its origin of 15 μs. This is also the time required to detect a collision.

Make the following assumptions about the 100 hosts on the network:

- The packet size is 1000 bytes.
- Each host tries to send a packet after T seconds of computation, where T is exponentially distributed with mean M. Note that the host begins its T seconds of computation only after successfully transmitting a packet.
- If a collision is detected, the host waits a random amount of time chosen from an exponential distribution with a mean of 60 μs.

Simulate and plot the sustained bandwidth of the network compared to the mean time between transmission attempts (M). Also, plot the average wait time between trying to initiate a transmission and succeeding in initiating it (compared to M).

Ethernets actually use an exponential back-off algorithm that increases the mean of the back-off time after successive collisions. Assume that the mean of the distribution from which the host chooses how much to delay is doubled on successive collisions. How well does this work? Is the bandwidth higher than when a single distribution is used? Can the initial mean be lower?

In More Depth

Disk Arrays

As mentioned in section 8.9, one method of organizing disk systems is to use arrays of smaller disks that provide more bandwidth through parallel access. In most disk arrays, all the spindles are synchronized—sector 0 in every disk rotates under the head at the exact same time—and the arms on all the disks are always over the same track. Furthermore, the data are "striped" across the disks in the array, so that consecutive sectors can be read in parallel. Let's explore how such a system might work.

8.29 [20] <§§8.3–8.5> Assume that we have the following two magnetic disk configurations: a single disk and an array of four disks. Each disk has 64 sectors per track, each sector holds 1000 bytes, and the disk revolves at 7200 RPM. Assume that the seek time is 6 ms. The delay of the disk controller is 1 ms per transaction, either for a single disk or for the array. Assume that the performance of the I/O system is limited only by the disks and the controller. Remember that the consecutive sectors on the single disk system will be spread one sector per disk in the array. Compare the performance in I/Os per second of these two disk organizations, assuming that the requests are random reads, half of which are 4 KB and half of which are 16 KB of data from sequential sectors. The sectors may be read in any order; for simplicity, assume that the rotational latency is one-half the revolution time for the single disk read of 16 sectors and the disk array read of 4 sectors. Challenge: Can you work out the actual average rotational latency in these two cases?

8.30 [10] <§§8.3–8.5> {Ex. 8.29} Using the same disk systems as in Exercise 8.29, with the same access patterns, determine the performance in megabytes per second for each system.

Glossary

absolute address A variable's or routine's actual address in memory.

abstraction A model that renders lower-level details of computer systems temporarily invisible in order to facilitate design of sophisticated systems.

activation record *See* procedure frame.

address A value used to delineate the location of a specific data element within a memory array.

address mapping *See* address translation.

address translation Also called address mapping. The process by which a virtual address is mapped to an address used to access memory.

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

aliasing A situation in which the same object is accessed by two addresses; can occur in virtual memory when there are two virtual addresses for the same physical page.

ALU *See* arithmetic logic unit (ALU).

Amdahl's law A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.

AND gate Hardware that performs the AND operation on input signals yielding a single signal result.

AND operation An operation that leaves a 1 in the result only if both bits of the operands are 1.

architecture *See* instruction set architecture.

arithmetic logic unit (ALU) Hardware that performs arithmetic and logical operations.

arithmetic mean The average of the execution times that is directly proportional to total execution time.

assembler A program that translates a symbolic version of an instruction into the binary version.

assembler directive An operation that tells the assembler how to translate a program but does not produce machine instructions; always begins with a period.

assembly language A symbolic language that can be translated into binary.

asserted signal A signal that is (logically) true, or 1.

asynchronous bus A bus that uses a handshaking protocol for coordinating usage rather than a clock; can accommodate a wide variety of devices of differing speeds.

atomic operation An operation in which the processor can both read a location and write it in the same bus operation, preventing any other processor or I/O device from reading or writing memory until it completes.

backpatching A method for translating from assembly language to machine instructions in which the assembler builds a (possibly incomplete) binary representation of every instruction in one pass over a program and then returns to fill in previously undefined labels.

backplane bus A bus that is designed to allow processors, memory, and I/O devices to coexist on a single bus.

barrier synchronization A synchronization scheme in which processors wait at the barrier and do not proceed until every processor has reached it.

base addressing Also called displacement addressing. An addressing regime in which the operand is at the memory location whose address is the sum of a register and an address in the instruction.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

biased notation A notation that represents the most negative value by $00\ldots000_{two}$ and the most positive value by $11\ldots11_{two}$, with 0 typically having the value $10\ldots00_{two}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

binary bit *See* binary digit.

binary digit Also called binary bit. One of the two numbers in base 2, 0 or 1, that are the components of information.

block The minimum unit of information that can be either present or not present in the two-level hierarchy.

Booth's algorithm An algorithm based on the observation that the ability to both add and subtract allows for multiple ways to compute a product, so that by looking at multiple bits we potentially save arithmetic operations.

branch delay slot The slot directly after a delayed branch instruction, which in the MIPS architecture is filled by an instruction that does not affect the branch.

branch hazard Also called control hazard. An occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

branch history table *See* branch prediction buffer.

branch not taken A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

branch prediction A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

branch prediction buffer Also called branch history table. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

branch taken A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional branches are taken branches.

branch target address The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

bubble *See* pipeline stall.

bus In logic design, a collection of data lines that is treated together as a single logical signal; also, a shared collection of lines with multiple sources and uses.

bus arbitration The process of deciding which bus master gets to use a bus next.

bus master A unit on the bus that can initiate bus requests.

bus request A signal on the bus requesting access to a bus.

bus transaction A sequence of bus operations that includes a request and may include a response, either of which may carry data. A transaction is initiated by a single request and may take many individual bus operations.

bypassing *See* forwarding.

cache coherency Consistency in the value of data between the versions in the caches of several processors.

cache memory A small, fast memory that acts as a buffer for a slower, larger memory.

cache miss A request for data from the cache that cannot be filled because the data is not present in the cache.

callee A procedure that executes a series of stored instructions based on parameters provided by the caller and then returns control to the caller.

callee-saved register A register saved by the routine making a procedure call.

caller The program that instigates a procedure and provides the necessary parameter values.

caller-saved register A register saved by the routine being called.

capacity miss A cache miss that occurs because the cache, even with full associativity, cannot contain all the block needed to satisfy the request.

central processor unit (CPU) Also called processor. The active part of the computer, which contains the datapath and control and which adds numbers, tests numbers, signals I/O devices to activate, and so on.

centralized, parallel arbitration A bus arbitration scheme that employs multiple request lines by which the devices independently request the bus and that uses a centralized arbiter to choose from the devices requesting bus access and to notify the selected device that it is now bus master.

chip *See* integrated circuit.

clock *See* clock cycle.

clock cycle Also called tick, clock tick, clock period, clock, cycle. The time for one clock period, usually of the processor clock, which runs at a constant rate. The clock cycle is often used to measure the speed at which hardware can perform basic functions.

clock cycles per instruction (CPI) Average number of clock cycles per instruction for a program or program fragment.

clock period *See* clock cycle.

clock rate The speed of the processor or system clock measured as the number of clock cycles per second and usually stated in megahertz or millions of clock cycles per second. The clock rate is the inverse of the clock period. Designers refer to the clock cycle time both as the duration of one clock period, measured as seconds per clock cycle (e.g., 2 ns) and as the clock rate, measured as clock cycles per second (e.g., 500 MHz).

clock skew The difference in absolute time between the times when two state elements see a clock edge.

clock tick *See* clock cycle.

clocking methodology The approach used to determine when data is valid and stable relative to the clock.

cluster A set of computers connected over a local area network (LAN) that function as a single large multiprocessor.

cold start miss *See* compulsory miss.

collision miss *See* conflict miss.

combinational logic A logic system whose blocks do not contain memory and hence compute the same output given the same input.

commit unit The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer-visible registers and memory.

compiler A program that translates high-level language statements into assembly language statements.

compulsory miss Also called cold start miss. A cache miss caused by the first access to a block that has never been in the cache.

computer generation A classification of computers often based on the implementation technology used in each generation, originally lasting eight to ten years.

conditional branch An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

conflict miss Also called collision miss. A cache miss that occurs in a set-associative or direct-mapped cache when multiple blocks compete for the same set and that are eliminated in a fully associative cache of the same size.

context switch A changing of the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.

control The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program.

control hazard *See* branch hazard.

control signal A signal used for multiplexor selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that is operated on by a functional unit.

control value *See* selector value.

CPI *See* clock cycles per instruction (CPI).

CPU *See* central processor unit (CPU).

CPU execution time Also called CPU time. The actual time the CPU spends computing for a specific task.

CPU time *See* CPU execution time.

crossbar network A network that allows any node to communicate with any other node in one pass through the network.

cycle *See* clock cycle.

D flip-flop A flip-flop with one data input that stores the value of that input signal in the internal memory when the clock edge occurs.

daisy chain arbitration A bus arbitration scheme in which the bus grant line is run through the devices from highest priority to lowest (the priorities are determined by the position on the bus) so that when the bus is requested the highest priority device sees the bus grant signal first.

data dependencies The need for specific data at a given point in a pipeline.

data hazard Also called pipeline data hazard. An occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

data parallelism Parallelism achieved by having massive data.

data segment The segment of a Unix object or executable file that contains a binary representation of the initialized data used by the program.

data transfer instruction A command that moves data between memory and registers.

datapath The component of the processor that performs arithmetic operations.

datapath element A functional unit used to operate on or hold data within a processor. In the MIPS implementation the datapath elements include the instruction and data memories, the register file, the arithmetic logic unit (ALU), and adders.

deasserted signal A signal that is (logically) false, or 0.

decoder A logic block that has an n-bit input and 2^n outputs where only one output is asserted for each input combination.

defect A microscopic flaw in a wafer or in patterning steps that can result in the failure of the die containing that defect.

delay *See* rotation latency.

delayed branch A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

delayed load A software format that requires load instructions to be followed by an instruction independent of the load.

die The individual rectangular sections that are cut from a wafer, more informally known as chips.

die area The size of a die.

direct-mapped cache A cache structure in which each memory location is mapped to exactly one location in the cache.

direct memory access (DMA) A mechanism that provides a device controller the ability to transfer data directly to or from the memory without involving the processor.

directory A repository for information on the state of every block in main memory, including which caches have copies of the block, whether it is dirty, and so on.

dispatch An operation in a microprogrammed control unit in which the next microinstruction is selected on the basis of one or more fields of a macroinstruction, usually by creating a table containing the addresses of the target microinstructions and indexing the table using a field of the macroinstruction. The dispatch tables are typically implemented in ROM or programmable logic array (PLA). The term *dispatch* is also used in dynamically scheduled processors to refer to the process of sending an instruction to a queue.

displacement addressing *See* base addressing.

distributed arbitration by collision detection A bus arbitration scheme that allows each device to independently request the bus and that uses a scheme for retrying the arbitration when multiple simultaneous requests occur.

distributed arbitration by self-selection A bus arbitration scheme that gives the devices requesting the bus the ability to determine which device gets the bus by having each requester detect whether it should receive the bus allocation.

distributed memory Physical memory that is divided into modules, with some placed near each processor in a multiprocessor.

distributed shared memory (DSM) A memory scheme that uses addresses to access remote data when demanded rather than retrieving the data in case it might be used.

divisor A number that the dividend is divided by; produces the dividend when multiplied by the quotient and added to the remainder.

DMA *See* direct memory access (DMA).

don't-care term An element of a logical function in which the output does not depend on the values of all the inputs. Don't-care terms may be specified in different ways.

double precision A floating-point value represented in two 32-bit words.

DRAM *See* dynamic random access memory (DRAM).

DSM *See* distributed shared memory (DSM).

dynamic pipeline scheduling A form of scheduling that goes past stalls in order to find later instructions to execute while waiting for the stalls to be resolved.

dynamic random access memory (DRAM) Memory that contains the instructions and data of a program while it is running, which allows faster access than accessing a magnetic disk.

edge-triggered clocking A clocking scheme in which all state changes occur on a clock edge.

Ethernet A computer network whose length is limited to about a kilometer. Originally capable of transferring up to 10 million bits per second, newer versions can run up to 100 million bits per second and even 1000 million bits per second. It treats the wire like a bus with multiple masters and uses collision detection and a back-off scheme for handling simultaneous accesses.

exception Also called interrupt. An unscheduled event that disrupts program execution; used to detect overflow.

exception enable Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

exclusive OR gate Hardware that performs the exclusive OR operation on input signals yielding a single signal result exclusive OR operation; also, an operation that leaves a 1 in the result only if two bits of the operands are unequal.

executable file A functional program in the format of an object file that contains no unresolved references, relocation information, symbol table, or debugging information.

execution time *See* response time.

exponent In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

external label Also called global label. A label referring to an object that can be referenced from files other than the one in which it is defined.

fairness A property of an allocation scheme, such as a bus arbitration protocol, that ensures that no device, even one with low priority, ever be completely locked out from the bus.

false sharing A sharing situation in which two unrelated shared variables are located in the same cache block and the full block is exchanged between processors even though the processors are accessing different variables.

finite state machine A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

firmware Microcode implemented in a memory structure, typically ROM or RAM.

flip-flop A memory element for which the output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.

floating point Computer arithmetic that represents numbers in which the binary point is not fixed.

floppy disk A portable form of secondary memory composed of a rotating mylar platter coated with a magnetic recording material.

flush (instructions) To discard instructions in a pipeline, usually due to an unexpected event.

formal parameter A variable that is the argument to a procedure or macro; replaced by that argument once the macro is expanded.

forward reference A label that is used before it is defined.

forwarding Also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

frame pointer A value denoting the location of the saved registers and local variables for a given procedure.

fully associative cache A cache structure in which a block can be placed in any location in the cache.

fully connected network A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

gate A device that implements basic logic functions, such as AND or OR.

general-purpose electronic computer A computer that has not been constructed for one specific function.

general-purpose register (GPR) A register that can be used for addresses or for data with virtually any instruction.

geometric mean $\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$ A formula useful for summarizing execution times that have been normalized.

gigabyte Traditionally 1,073,741,824 (2^{30}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000 (10^9) bytes.

global label *See* external label.

global miss rate The fraction of references that miss in all levels of a multilevel cache.

global pointer The register that is reserved for static data.

GPR *See* general-purpose register (GPR).

guard The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

handshaking protocol A series of steps used to coordinate asynchronous bus transfers in which the sender and receiver proceed to the next step only when both parties agree that the current step has been completed.

hard disk A form of secondary memory composed of rotating metal platters coated with a magnetic recording material.

hardwired control An implementation of finite state machine control typically using programmable logic arrays (PLAs) or collections of PLAs and random logic.

harmonic mean of rates $\text{HM} = \dfrac{n}{\sum_{i=1}^{n} \dfrac{1}{\text{Rate}_i}}$ A summary that tracks execution time when the data is given as rates rather than as a times.

hexadecimal Numbers in base 16.

high-level programming language A portable language such as C, Fortran, or Java composed of English words and algebraic notation that can be translated by a compiler into assembly language.

hit rate The fraction of memory accesses found in a cache.

hit time The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

hold time The minimum time during which the input must be valid after the clock edge.
horizontal microcode Use of microinstructions containing many fields that can control the datapath units in parallel and require little additional decoding. The use of many fields makes the microinstructions wider or more horizontal.
immediate addressing An addressing regime in which the operand is a constant within the instruction itself.
implementation Hardware that obeys the architecture abstraction.
imprecise exception *See* imprecise interrupt.
imprecise interrupt Also called imprecise exception. Interrupts or exceptions in pipelined computers that are not associated with the exact instruction that was the cause of the interrupt or exception.
in-order commit A commit in which the results of pipelined execution are written to the programmer-visible state in the same order that instructions are fetched.
in-order execution A conventional pipelined execution, in which all following instructions must wait when an instruction is blocked from executing.
input device A mechanism through which the computer is fed information, such as the keyboard or mouse.
instruction format A form of representation of an instruction composed of fields of binary numbers.
instruction latency The inherent execution time for an instruction.
instruction mix A measure of the dynamic frequency of instructions across one or many programs.
instruction set The vocabulary of commands understood by a given architecture.
instruction set architecture Also called architecture. An abstract interface between the hardware and the lowest level software of a machine that encompasses all the information necessary to write a machine language program that will run correctly, including instructions, registers, memory size, and so on.
integrated circuit Also called chip. A device combining dozens to millions of transistors.
interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)
interrupt-driven I/O An I/O scheme that employs interrupts to indicate to the processor that an I/O device needs attention.
interrupt enable *See* exception enable.
interrupt handler A piece of code that is run as a result of an exception or an interrupt.
I/O instruction A dedicated instruction that is used to give a command to an I/O device and that specifies both the device number and the command word (or the location of the command word in memory).
jump address table Also called jump table. A table of addresses of alternative instruction sequences.
jump-and-link instruction An instruction that jumps to an address and simultaneously saves the address of the following instruction in a register (`$ra` in MIPS).
jump table *See* jump address table.
kernel benchmark A small, time-intensive code fragment from a real program that is used for performance evaluation.
kernel mode Also called supervisor mode. A mode indicating that a running process is an operating system process.
kilobyte 1024 (2^{10}) bytes.
LAN *See* local area network (LAN).
latch A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.
latency (pipeline) The number of stages in a pipeline or the number of stages between two instructions during execution.
least recently used (LRU) A replacement scheme in which the block replaced is the one that has been unused for the longest time.
least significant bit The rightmost bit in a MIPS word.
level-sensitive clocking A timing methodology in which state changes occur at either high or low clock levels but are not instantaneous, as such changes are in edge-triggered designs.
link editor *See* linker.
linker Also called link editor. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.
load-store machine Also called register-register machine. An instruction set architecture in which all operations are between registers and data memory may only be accessed via loads or stores.
load-use data hazard A specific form of data hazard in which the data requested by a load instruction has not yet become available when it is requested.
loader A systems program that places an object program in main memory so that it is ready to execute.
local area network (LAN) A network designed to carry data within a geographically confined area, typically within a single building.
local label A label referring to an object that can be used only within the file in which it is defined.
local miss rate The fraction of references to one level of a cache that miss; used in multilevel hierarchies.
lock A synchronization device that allows access to data to only one processor at a time.

logic minimization A technique for reducing the number of gates needed to implement a set of logic functions.
loop unrolling A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.
LRU *See* least recently used (LRU).
machine language Binary representation used for communication within a computer system.
macro A pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions.
macroinstruction An instruction in the instruction set architecture being implemented, used to distinguish the instructions visible to the programmer from the microinstructions of a microprogrammed control unit.
magnetic disk A form of nonvolatile secondary memory composed of rotating platters coated with a magnetic recording material.
main memory *See* primary memory.
main-memory coherence Consistency in the value of data in memory in a network-connected multiprocessor.
massively parallel A computer with at least 100 processors.
maximally encoded Use of encoded forms of control that require multiple levels of decode; vertical microcode is maximally encoded.
megabyte Traditionally 1,048,576 (2^{20}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000 (10^6) bytes.
megaFLOPS *See* million floating-point operations per second (MFLOPS).
memory The storage area in which programs are kept when they are running and that contains the data needed by the running programs.
memory hierarchy A structure that uses multiple levels of memories; as the distance from the CPU increases, the size of the memories and the access time both increase.
memory-mapped I/O An I/O scheme in which portions of address space are assigned to I/O devices and reads and writes to those addresses are interpreted as commands to the I/O device.
MESI cache coherency protocol A write-invalidate protocol whose name is an acronym for the four states of the protocol: Modified, Exclusive, Shared, Invalid.
message passing Communicating between multiple processors by explicitly sending and receiving information.
metastability A situation that occurs if a signal is sampled when it is not stable for the required set-up and hold times, possibly causing the sampled value to fall in the indeterminate region between a high and low value.
MFLOPS *See* million floating-point operations per second (MFLOPS).
microcode The set of microinstructions that control a processor.
microcode assembler A program that translates microprograms into microinstructions that can be implemented in a ROM or PLA.
microinstruction A representation of control using low-level instructions, each of which asserts a set of control signals that are active on a given clock cycle as well as specifies what microinstruction to execute next.
microprogram A symbolic representation of control in the form of instructions, called microinstructions, that are executed on a simple micromachine.
microprogrammed control A method of specifying control that uses microcode rather than a finite state representation.
million floating-point operations per second (MFLOPS) Also called megaFLOPS. A measurement of program execution speed based on the number of millions of floating-point operations executed per second. MFLOPS is computed as the number of floating-point operations in a program divided by the product of the execution time and 10^6.
million instructions per second (MIPS) A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6.
MIMD *See* multiple instruction streams, multiple data streams (MIMD).
minimally encoded Use of an unencoded control format that can directly control a datapath; horizontal microcode is minimally encoded.
minterms Also called product terms. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the programmable logic array (PLA).
MIPS *See* million instructions per second (MIPS).
miss penalty The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, and insert it in the level that experienced the miss.
miss rate The fraction of memory accesses not found in a level of the memory hierarchy.

most significant bit The leftmost bit in a MIPS word.
motherboard A plastic board containing packages of integrated circuits or chips, including processor, cache, memory, and connectors for I/O devices such as networks and disks.
multicomputer Parallel processors with multiple private addresses.
multicycle implementation Also called multiple clock cycle implementation. An implementation in which an instruction is executed in multiple clock cycles.
multilevel cache A memory hierarchy with multiple levels of caches, rather than just a cache and main memory.
multiple clock cycle implementation *See* multicycle implementation.
multiple-instruction issue A procedure in which the instruction fetch unit can send multiple instructions to the next pipeline stage in a single clock cycle.
multiple instruction streams, multiple data streams (MIMD) A computer classification in Flynn's taxonomy referring to computers that use multiple instruction streams and multiple data streams.
multiprocessor Parallel processors with a single shared address.
multistage network A network that supplies a small switch at each node.
NAND gate An inverted AND gate.
network bandwidth Informally, the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network.
next-state counter A counter that supplies the sequential next state.
next-state function A combinational function that, given the inputs and the current state, determines the next state of a finite state machine.
next-state output An output of the combinational logic that specifies the next-state number.
nonblocking cache A cache that allows the processor to make references to the cache while the cache is handling an earlier miss.
nonuniform memory access (NUMA) A type of single-address space multiprocessor in which some memory accesses are faster than others depending which processor asks for which word.
nonvolatile memory A form of memory that retains data even in the absence of a power source and that is used to store programs between runs. Magnetic disk is nonvolatile and DRAM is not.
nop An instruction that does no operation to change state.
NOR gate An inverted OR gate.
normalized A number in floating-point notation that has no leading 0s.
NUMA *See* nonuniform memory access (NUMA).
object program A combination of machine language instructions, data, and information needed to place them properly in memory.
opcode The field that denotes the operation and format of an instruction.
operating system Supervising program that manages the resources of a computer for the benefit of the programs that run on that machine.
out-of-order commit A commit in which the results of pipelined execution need not be written to the programmer visible state in the same order that instructions are fetched.
out-of-order execution A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.
output device A mechanism that conveys the result of a computation to the user.
overflow (floating-point) A situation in which a positive exponent becomes too large to fit in the exponent field.
page fault An event that occurs when an accessed page is not present in main memory.
page mode A mechanism in DRAM that provides the ability to access multiple bits of a row by changing the column address only and, hence, is faster than a normal access cycle that changes row and column addresses.
page table The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.
parallel processing program A single program that runs on multiple processors simultaneously.
PC *See* program counter (PC).
PC-relative addressing An addressing regime in which the address is the sum of the program counter (PC) and a constant in the instruction.
personal computer A general-purpose computer designed to be manufactured in high volume and at a cost affordable enough to allow for use in the home.
physical address An address in main memory.
physically addressed cache A cache that is addressed by a physical address.
pipeline data hazard *See* data hazard.

pipeline stall Also called bubble. A stall initiated in order to resolve a hazard.
pipelining An implementation technique in which multiple instructions are overlapped in execution, much like to an assembly line.
pipelining stage A step in executing an instruction that occurs simultaneously with other steps in other instructions and typically lasts one clock cycle.
pixel The smallest individual picture element. Screen are composed of hundreds of thousands to millions of pixels, organized in a matrix.
PLA *See* programmable logic array (PLA).
polling The process of periodically checking the status of an I/O device to determine the need to service the device.
precise exception *See* precise interrupt.
precise interrupt Also called precise exception. An interrupt or exception that is always associated with the correct instruction in pipelined computers.
prefetching A technique in which data blocks needed in the future are brought into the cache early by the use of special instructions that specify the address of the block.
primary memory Also called main memory. Volatile memory used to hold programs while they are running; typically consists of DRAM in today's computers.
procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.
procedure call convention *See* register-use convention.
procedure call frame A block of memory that is used to hold values passed to a procedure as arguments, to save registers that a procedure may modify but that the procedure's caller does not want changed, and to provide space for variables local to a procedure.
procedure frame Also called activation record. The segment of the stack containing a procedure's saved registers and local variables.
processor-memory bus A bus that connects processor and memory and that is short, generally high speed, and matched to the memory system so as to maximize memory-processor bandwidth.
product terms *See* minterms.
program counter (PC) The register containing the address of the instruction in the program being executed
programmable logic array (PLA) A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic: the first generating product terms of the inputs and input complements and the second generating sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.
programmable ROM (PROM) A form of read-only memory that can be programmed when a designer knows its contents.
PROM *See* programmable ROM (PROM).
propagation time The time required for an input to a flip-flop to propagate to the outputs of the flip-flop.
protection A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the operating system from a user process.
pseudoinstruction A common variation of assembly language instructions often treated as if it were an instruction in its own right.
quotient The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.
RAID *See* redundant arrays of inexpensive disks (RAID).
raster cathode ray tube (CRT) display A display, such as a television set, that scans an image one line at a time, 30 to 75 times per second.
read-only memory (ROM) A memory whose contents are designated at creation time, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.
receive message routine A routine used by a processor in machines with private memories to accept a message from another processor.
recursive procedures Procedures that call themselves either directly or indirectly through a chain of calls.
redundant arrays of inexpensive disks (RAID) An organization of disks that uses an array of small and inexpensive disks so as to increase both performance and reliability.
reference bit Also called use bit. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.
register addressing A mode of addressing in which the operand is a register.
register file A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
register-register machine *See* load-store machine.
register use *See* register-use convention.
register-use convention Also called procedure call convention. A software protocol governing the use of registers by procedures.
relocation information The segment of a Unix object file that identifies instructions and data words that depend on absolute addresses.

remainder The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

rename buffer Also called rename register. An extra internal register within processors that is used to hold results while waiting for the commit unit to commit the result to one of the real registers.

rename register *See* rename buffer.

reorder buffer A register that holds instructions in a dynamic pipelined machine whose results have not yet been committed to programmer-visible registers or memory; machines with out-of-order execution and in-order commit will retire an instruction from the reorder buffer only when the instruction has finished execution and all instructions ahead of it have been completed.

reservation station A buffer within a functional unit that holds the operands and the operation.

response time Also called execution time. The total time required for the computer to complete a task, including disk accesses, memory accesses, I/O activities, operating system overhead, CPU execution time, and so on.

restartable instruction An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction.

return address A link to the calling site that allows a procedure to return to the proper address; in MIPS it is stored in register `$ra`.

ROM *See* read-only memory (ROM).

rotation latency Also called delay. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

round Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

scientific notation A notation that renders numbers with a single digit to the left of the decimal point.

SCSI *See* small computer systems interface (SCSI).

secondary memory Nonvolatile memory used to store programs and data between runs; typically consists of magnetic disks in today's computers.

sector One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

seek The process of positioning a read/write head over the proper track on a disk.

segmentation A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

selector value Also called control value. The control signal that is used to select one of the input values of a multiplexor as the output of the multiplexor.

semiconductor A substance that does not conduct electricity well.

send message routine A routine used by a processor in machines with private memories to pass to another processor.

separate compilation Splitting a program across many files, each of which can be compiled without knowledge of what is in the other files.

sequential access memory Memory whose access time differs depending on the location of the data being retrieved because data is stored sequentially so that all data must be passed over to access the final bit of information; contrasts with random access memory, in which any bit may be accessed in the same time.

sequential logic A group of logic elements that contain memory and hence whose value depends on the inputs as well as the current contents of the memory.

set-associative cache A cache that has a fixed number of locations (at least two) where each block can be placed.

set-up time The minimum time that the input to a memory device must be valid before the clock edge.

shared memory A memory for a parallel processor with a single address space, implying implicit communication with loads and stores.

sign-extend To increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

significand In the numerical representation system of floating-point arithmetic, the value in that is placed in the significand field.

silicon A substance found in sand that does not conduct electricity well.

silicon crystal ingot A rod composed of silicon crystal that is between 6 and 12 inches in diameter and about 12 to 24 inches long.

SIMD *See* single instruction stream, multiple data streams (SIMD).

SIMM *See* single in-line memory module (SIMM).

single clock cycle implementation *See* single-cycle implementation.

single-cycle implementation Also called single clock cycle implementation. An implementation in which an instruction is executed in one clock cycle.

single in-line memory module (SIMM) A small printed circuit board containing 4 to 24 DRAM integrated circuits. Today's computers use SIMMs to allow main memory to be upgraded and expanded over time by the customer.

single instruction stream, multiple data streams (SIMD) A computer classification in Flynn's taxonomy that refers to computers with single instruction streams but multiple data streams and in which a single instruction operates on many data elements at the same time.

single instruction stream, single data stream (SISD) A computer classification in Flynn's taxonomy that refers to computers with single instruction streams and single data streams. (SISD is the conventional processor covered in the first eight chapters.)

single precision A floating-point value represented in a single 32-bit word.

SISD *See* single instruction stream, single data stream (SISD).

slave A device that responds to read and write requests but does not generate them and hence cannot be a bus master.

small computer systems interface (SCSI) A bus used as a standard for I/O devices.

SMP *See* symmetric multiprocessor (SMP).

snooping cache coherency A method for maintaining cache coherency in which all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of the desired block.

source language The high-level language in which a program is originally written.

spatial locality The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

SPEC benchmark *See* system performance evaluation cooperative (SPEC) benchmark.

speculative execution A pipelining technique that combines dynamic scheduling with branch prediction.

speedup The measure of how a machine performs relative to how it previously performed before an enhancement was implemented. Speedup is equal to the ratio of execution time before the enhancement to execution time after the enhancement.

split cache A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other with one handling instructions and one handling data.

split transaction protocol A protocol in which the bus is released during a bus transaction while the requester is waiting for the data to be transmitted, which frees the bus for access by another requester.

SRAM *See* static random access memory (SRAM).

stack A data structure for spilling registers organized as a last-in-first-out queue.

stack frame *See* procedure call frame.

stack pointer A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found.

stack segment The portion of memory used by a program to hold procedure call frames.

state assignment A control optimization that works by attempting to choose the state numbers such that the resulting logic equations contain more redundancy and can thus be simplified.

state element A memory element.

state input An input to the combinational logic that specifies the current state.

static data The portion of memory that contains data whose size is known to the compiler and whose lifetime is the program's entire execution.

static random access memory (SRAM) A memory where data is stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

sticky bit A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

stored-program computer A computer whose instructions are represented as numbers, allowing the same memory to contain instructions and data and thus allowing programs to produce programs.

stored-program concept The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

structural hazard An occurrence in which a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the combination of instructions that are set to execute in the given clock cycle.

subroutine library A collection of commonly used programs.

sum of products A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

supercomputer The fastest and most expensive computer, typically used for scientific computation. Supercomputers generally cost between $1 and $30 million.

superpipelining A technique that increases processor speed by lengthening pipelines.

superscalar An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle.

superscalar pipelining A technique that replicates internal components of the computer in order to launch and execute multiple instructions in every pipeline stage.

supervisor mode *See* kernel mode.

symbol table A table that matches names of labels to the addresses of the memory words that instructions occupy.

symmetric multiprocessor (SMP) Also called UMA machine. A multiprocessor in which accesses to main memory take the same amount of time no matter which processor requests the access and no matter which word is asked.

synchronization The process of coordinating the behavior of two or more processes, which may be running on different processors.

synchronizer failure A situation in which a flip-flop enters a metastable state and where some logic blocks reading the output of the flip-flop see a 0 while others see a 1.

synchronous bus A bus that includes a clock in the control lines and a fixed protocol for communicating that is relative to the clock.

synchronous system A memory system that employs clocks and where data signals are read only when the clock indicates that the signal values are stable.

system call A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

system CPU time The CPU time spent in the operating system performing tasks on behalf of the program.

system performance evaluation cooperative (SPEC) benchmark A set of standard CPU-intensive, integer and floating point benchmarks based on real programs.

systems software Software that provides services that are commonly useful, including operating systems, compilers, and assemblers.

tag A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

temporal locality The principle stating that if a data location is referenced then it will tend to be referenced again soon.

terabyte Originally 1,099,511,627,776 (2^{40}) bytes, although some communications and secondary storage systems have redefined it to mean 1,000,000,000,000 (10^{12}) bytes.

text segment The segment of a Unix object file that contains the machine language code for routines in the source file.

three Cs model A cache model in which all cache misses are classified into one of three categories: compulsory misses, capacity misses, and conflict misses.

tick *See* clock cycle.

TLB *See* translation-lookaside buffer (TLB).

track One of 1000 to 5000 concentric circles that makes up the surface of a magnetic disk.

transaction processing A type of application that involves handling small short operations (called transactions) that typically require both I/O and computation. Transaction processing applications typically have both response time requirements and a performance measurement based on the throughput of transactions.

transfer time The time required to transfer a block of bits, typically a sector, during disk access.

transistor An on/off switch controlled by electricity.

translation-lookaside buffer (TLB) A cache that keeps track of recently used address mappings to avoid an access to the page table.

ulp *See* units in the last place (ulp).

UMA *See* uniform memory access (UMA).

UMA machine *See* symmetric multiprocessor (SMP).

underflow (floating-point) A situation in which a negative exponent becomes too large to fit in the exponent field.

uniform memory access (UMA) Memory access that takes the same amount of time no matter which processor requests the access and no matter which word is asked for.

units in the last place (ulp) The number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented.

unresolved reference A reference that requires more information from an outside source in order to be complete.

use bit *See* reference bit.

user CPU time The CPU time spent in a program itself.

vacuum tube An electronic component, predecessor of the transistor, that consists of a hollow glass tube about 5 to 10 cm long from which as much air has been removed as possible.

valid bit A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

vector processor An architecture and compiler model that was popularized by supercomputers in which high-level operations work on linear arrays of numbers.

vector supercomputer A supercomputer whose instructions operate on vectors of numbers, typically 64 floating-point numbers at a time.

vectored interrupt An interrupt for which the address to which control is transferred is determined by the cause of the exception.

vertical microcode Use of microinstructions containing many fewer fields that require additional decoding before being used to control the datapath units. The use of fewer fields makes the microinstructions narrower or more vertical.

very large scale integrated (VLSI) circuit A device containing tens of thousands to millions of transistors.

virtual address An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

virtual machine A virtual computer that appears to have nondelayed branches and loads and a richer instruction set than the actual hardware.

virtual memory A technique that uses main memory as a "cache" for secondary storage.

virtually addressed cache A cache that is accessed with a virtual address rather than a physical address.

VLSI circuit *See* very large scale integrated (VLSI) circuit.

volatile memory Storage, such as DRAM, that only retains data if it is receiving power.

wafer A slice from a silicon ingot no more than 0.1 inch thick, used to create chips.

weighted arithmetic mean A summary that tracks the execution time of a workload with weighting factors designed to reflect the presence of the programs in a workload; computed as the sum of the products of weighting factors and execution times.

wide area network A network extended over hundreds of kilometers which can span a continent.

word The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

workload A set of programs run on a computer that is either the actual collection of applications run by a user or is constructed from real programs to approximate such a mix. A typical workload specifies both the programs as well as the relative frequencies.

write-back A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

write-broadcast A snooping protocol scheme in which the writing processor disseminates the new data over the bus, allowing all copies to be updated with the new value.

write buffer A queue that holds data while the data are waiting to be written to memory.

write-invalidate A type of snooping protocol in which the writing processor causes all copies in other caches to be invalidated before changing its local copy, which allows it to update the local data until another processor asks for it.

write-through A scheme in which writes always update both the cache and the memory, ensuring that data is always consistent between the two.

yield The percentage of good dies from the total number of dies on the wafer.