chapter

8

THE MEMORY SYSTEM

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Basic memory circuits
- Organization of the main memory
- Memory technology
- Direct memory access as an I/O mechanism
- Cache memory, which reduces the effective memory access time
- Virtual memory, which increases the apparent size of the main memory
- Magnetic and optical disks used for secondary storage

Programs and the data they operate on are held in the memory of the computer. In this chapter, we discuss how this vital part of the computer operates. By now, the reader appreciates that the execution speed of programs is highly dependent on the speed with which instructions and data can be transferred between the processor and the memory. It is also important to have sufficient memory to facilitate execution of large programs having large amounts of data.

Ideally, the memory would be fast, large, and inexpensive. Unfortunately, it is impossible to meet all three of these requirements simultaneously. Increased speed and size are achieved at increased cost. Much work has gone into developing structures that improve the effective speed and size of the memory, yet keep the cost reasonable.

The memory of a computer comprises a hierarchy, including a cache, the main memory, and secondary storage, as Chapter 1 explains. In this chapter, we describe the most common components and organizations used to implement these units. Direct memory access is introduced as a mechanism to transfer data between an I/O device, such as a disk, and the main memory, with minimal involvement from the processor. We examine memory speed and discuss how access times to memory data can be reduced by means of caches. Next, we present the virtual memory concept, which makes use of the large storage capacity of secondary storage devices to increase the effective size of the memory. We start with a presentation of some basic concepts, to extend the discussion in Chapters 1 and 2.

8.1 BASIC CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a computer that generates 16-bit addresses is capable of addressing up to $2^{16} = 64$K (kilo) memory locations. Machines whose instructions generate 32-bit addresses can utilize a memory that contains up to $2^{32} = 4$G (giga) locations, whereas machines with 64-bit addresses can access up to $2^{64} = 16$E (exa) $\approx 16 \times 10^{18}$ locations. The number of locations represents the size of the address space of the computer.

The memory is usually designed to store and retrieve data in word-length quantities. Consider, for example, a byte-addressable computer whose instructions generate 32-bit addresses. When a 32-bit address is sent from the processor to the memory unit, the high-order 30 bits determine which word will be accessed. If a byte quantity is specified, the low-order 2 bits of the address specify which byte location is involved.

The connection between the processor and its memory consists of address, data, and control lines, as shown in Figure 8.1. The processor uses the address lines to specify the memory location involved in a data transfer operation, and uses the data lines to transfer the data. At the same time, the control lines carry the command indicating a Read or a Write operation and whether a byte or a word is to be transferred. The control lines also provide the necessary timing information and are used by the memory to indicate when it has completed the requested operation. When the processor-memory interface receives the memory's response, it asserts the MFC signal shown in Figure 5.19. This is the processor's internal control signal that indicates that the requested memory operation has been completed. When asserted, the processor proceeds to the next step in its execution sequence.

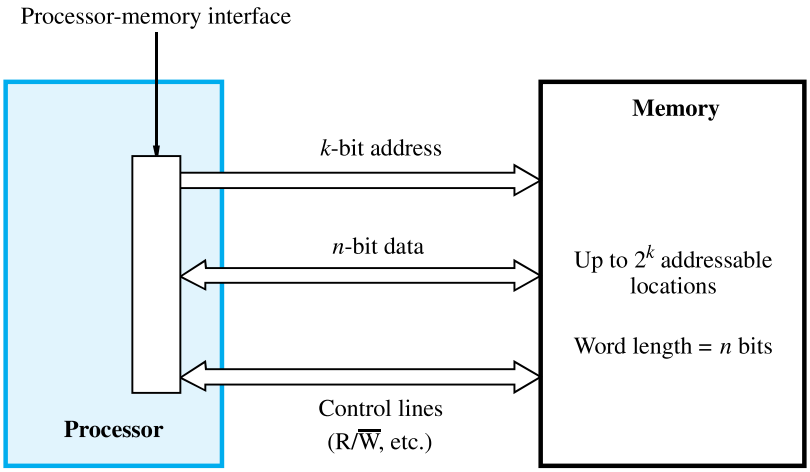

Figure 8.1 Connection of the memory to the processor.

A useful measure of the speed of memory units is the time that elapses between the initiation of an operation to transfer a word of data and the completion of that operation. This is referred to as the *memory access time.* Another important measure is the *memory cycle time,* which is the minimum time delay required between the initiation of two successive memory operations, for example, the time between two successive Read operations. The cycle time is usually slightly longer than the access time, depending on the implementation details of the memory unit.

A memory unit is called a *random-access memory* (RAM) if the access time to any location is the same, independent of the location's address. This distinguishes such memory units from serial, or partly serial, access storage devices such as magnetic and optical disks. Access time of the latter devices depends on the address or position of the data.

The technology for implementing computer memories uses semiconductor integrated circuits. The sections that follow present some basic facts about the internal structure and operation of such memories. We then discuss some of the techniques used to increase the effective speed and size of the memory.

Cache and Virtual Memory

The processor of a computer can usually process instructions and data faster than they can be fetched from the main memory. Hence, the memory access time is the bottleneck in the system. One way to reduce the memory access time is to use a *cache memory.* This is a small, fast memory inserted between the larger, slower main memory and the processor. It holds the currently active portions of a program and their data.

Virtual memory is another important concept related to memory organization. With this technique, only the active portions of a program are stored in the main memory, and the remainder is stored on the much larger secondary storage device. Sections of the program are transferred back and forth between the main memory and the secondary storage device

in a manner that is transparent to the application program. As a result, the application program sees a memory that is much larger than the computer's physical main memory.

Block Transfers

The discussion above shows that data move frequently between the main memory and the cache and between the main memory and the disk. These transfers do not occur one word at a time. Data are always transferred in contiguous blocks involving tens, hundreds, or thousands of words. Data transfers between the main memory and high-speed devices such as a graphic display or an Ethernet interface also involve large blocks of data. Hence, a critical parameter for the performance of the main memory is its ability to read or write blocks of data at high speed. This is an important consideration that we will encounter repeatedly as we discuss memory technology and the organization of the memory system.

8.2 SEMICONDUCTOR RAM MEMORIES

Semiconductor random-access memories (RAMs) are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. In this section, we discuss the main characteristics of these memories. We start by introducing the way that memory cells are organized inside a chip.

8.2.1 INTERNAL ORGANIZATION OF MEMORY CHIPS

Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. A possible organization is illustrated in Figure 8.2. Each row of cells constitutes a memory word, and all cells of a row are connected to a common line referred to as the *word line,* which is driven by the address decoder on the chip. The cells in each column are connected to a Sense/Write circuit by two *bit lines,* and the Sense/Write circuits are connected to the data input/output lines of the chip. During a Read operation, these circuits sense, or read, the information stored in the cells selected by a word line and place this information on the output data lines. During a Write operation, the Sense/Write circuits receive input data and store them in the cells of the selected word.

Figure 8.2 is an example of a very small memory circuit consisting of 16 words of 8 bits each. This is referred to as a 16 $\times$ 8 organization. The data input and the data output of each Sense/Write circuit are connected to a single bidirectional data line that can be connected to the data lines of a computer. Two control lines, R/$\overline{W}$ and CS, are provided. The R/$\overline{W}$ (Read/$\overline{\text{Write}}$) input specifies the required operation, and the CS (Chip Select) input selects a given chip in a multichip memory system.

The memory circuit in Figure 8.2 stores 128 bits and requires 14 external connections for address, data, and control lines. It also needs two lines for power supply and ground connections. Consider now a slightly larger memory circuit, one that has 1K (1024) memory cells. This circuit can be organized as a 128 $\times$ 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K $\times$ 1 format. In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 external

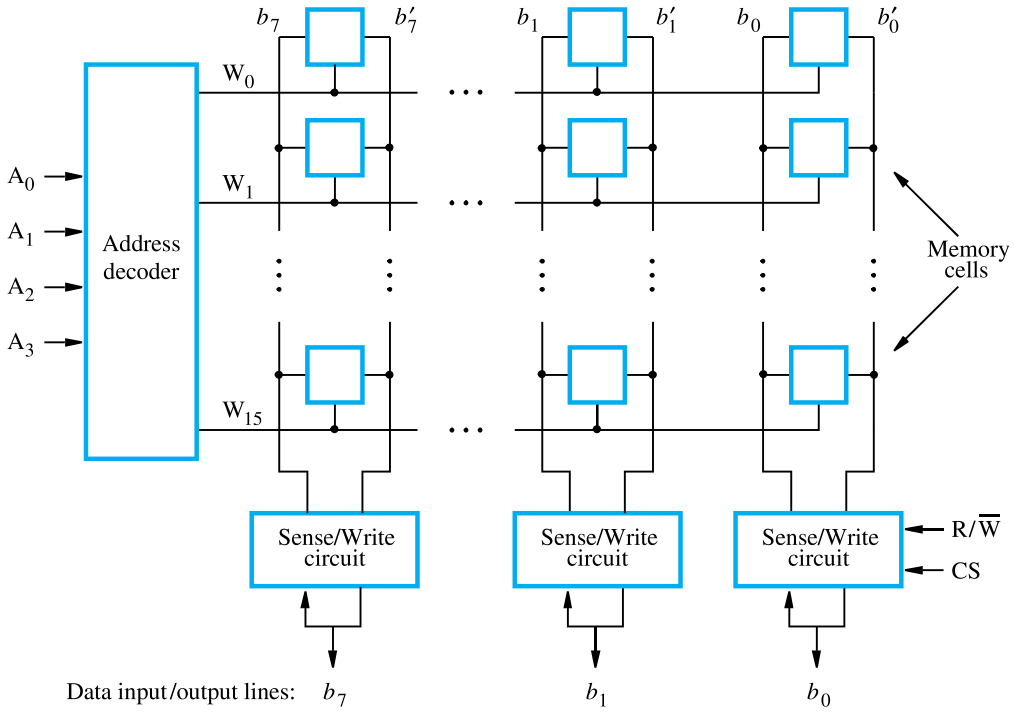

Figure 8.2 Organization of bit cells in a memory chip.

connections. Figure 8.3 shows such an organization. The required 10-bit address is divided into two groups of 5 bits each to form the row and column addresses for the cell array. A row address selects a row of 32 cells, all of which are accessed in parallel. But, only one of these cells is connected to the external data line, based on the column address.

Commercially available memory chips contain a much larger number of memory cells than the examples shown in Figures 8.2 and 8.3. We use small examples to make the figures easy to understand. Large chips have essentially the same organization as Figure 8.3, but use a larger memory cell array and have more external connections. For example, a 1G-bit chip may have a 256M $\times$ 4 organization, in which case a 28-bit address is needed and 4 bits are transferred to or from the chip.

8.2.2 STATIC MEMORIES

Memories that consist of circuits capable of retaining their state as long as power is applied are known as *static memories*. Figure 8.4 illustrates how a *static RAM* (SRAM) cell may be implemented. Two inverters are cross-connected to form a latch. The latch is connected to two bit lines by transistors T_1 and T_2. These transistors act as switches that can be opened or

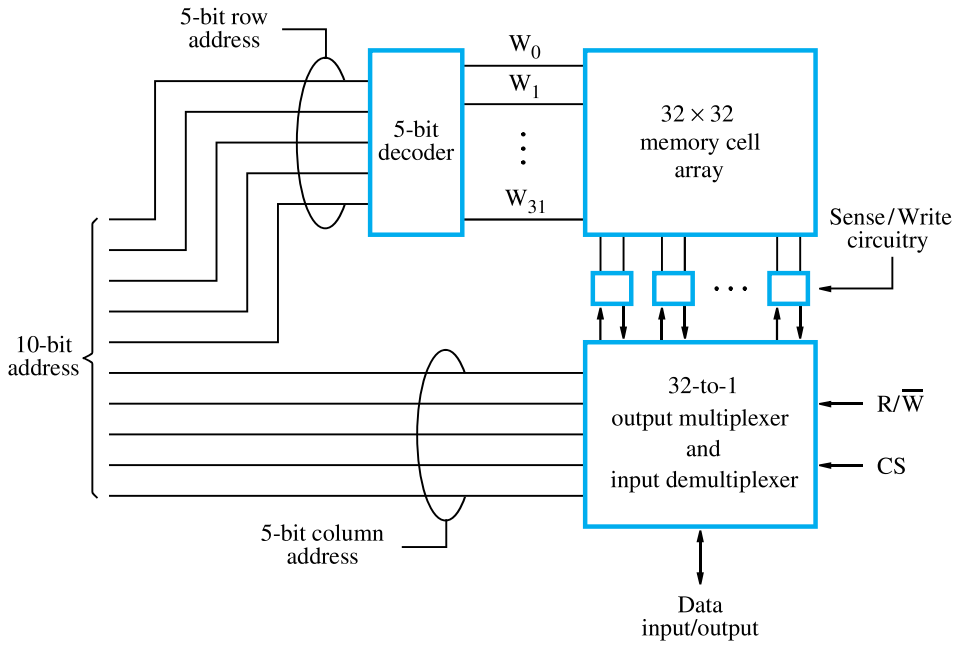

Figure 8.3 Organization of a 1K × 1 memory chip.

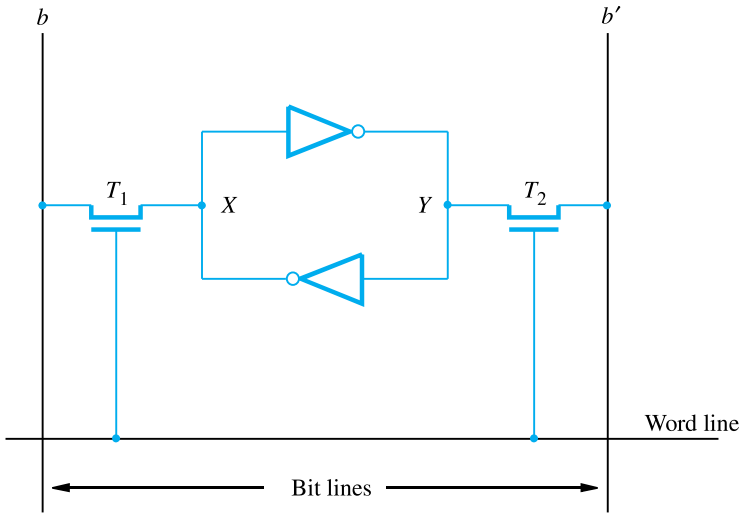

Figure 8.4 A static RAM cell.

closed under control of the word line. When the word line is at ground level, the transistors are turned off and the latch retains its state. For example, if the logic value at point X is 1 and at point Y is 0, this state is maintained as long as the signal on the word line is at ground level. Assume that this state represents the value 1.

Read Operation

In order to read the state of the SRAM cell, the word line is activated to close switches T_1 and T_2. If the cell is in state 1, the signal on bit line b is high and the signal on bit line b' is low. The opposite is true if the cell is in state 0. Thus, b and b' are always complements of each other. The Sense/Write circuit at the end of the two bit lines monitors their state and sets the corresponding output accordingly.

Write Operation

During a Write operation, the Sense/Write circuit drives bit lines b and b', instead of sensing their state. It places the appropriate value on bit line b and its complement on b' and activates the word line. This forces the cell into the corresponding state, which the cell retains when the word line is deactivated.

CMOS Cell

A CMOS realization of the cell in Figure 8.4 is given in Figure 8.5. Transistor pairs (T_3, T_5) and (T_4, T_6) form the inverters in the latch (see Appendix A). The state of the cell is read or written as just explained. For example, in state 1, the voltage at point X is maintained high by having transistors T_3 and T_6 on, while T_4 and T_5 are off. If T_1 and T_2 are turned on, bit lines b and b' will have high and low signals, respectively.

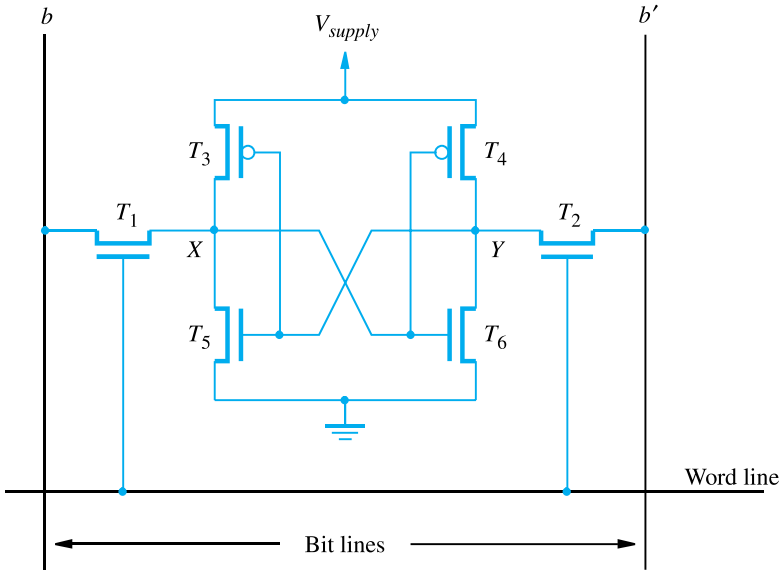

Figure 8.5 An example of a CMOS memory cell.

Continuous power is needed for the cell to retain its state. If power is interrupted, the cell's contents are lost. When power is restored, the latch settles into a stable state, but not necessarily the same state the cell was in before the interruption. Hence, SRAMs are said to be *volatile* memories because their contents are lost when power is interrupted.

A major advantage of CMOS SRAMs is their very low power consumption, because current flows in the cell only when the cell is being accessed. Otherwise, T_1, T_2, and one transistor in each inverter are turned off, ensuring that there is no continuous electrical path between V_{supply} and ground.

Static RAMs can be accessed very quickly. Access times on the order of a few nanoseconds are found in commercially available chips. SRAMs are used in applications where speed is of critical concern.

8.2.3 DYNAMIC RAMS

Static RAMs are fast, but their cells require several transistors. Less expensive and higher density RAMs can be implemented with simpler cells. But, these simpler cells do not retain their state for a long period, unless they are accessed frequently for Read or Write operations. Memories that use such cells are called *dynamic RAMs* (DRAMs).

Information is stored in a dynamic memory cell in the form of a charge on a capacitor, but this charge can be maintained for only tens of milliseconds. Since the cell is required to store information for a much longer time, its contents must be periodically *refreshed* by restoring the capacitor charge to its full value. This occurs when the contents of the cell are read or when new information is written into it.

An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown in Figure 8.6. To store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line. This causes a known amount of charge to be stored in the capacitor.

After the transistor is turned off, the charge remains stored in the capacitor, but not for long. The capacitor begins to discharge. This is because the transistor continues to

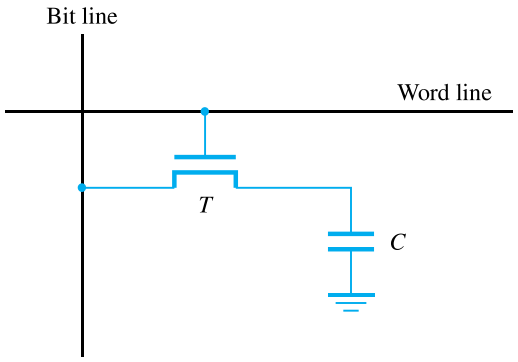

Figure 8.6 A single-transistor dynamic memory cell.

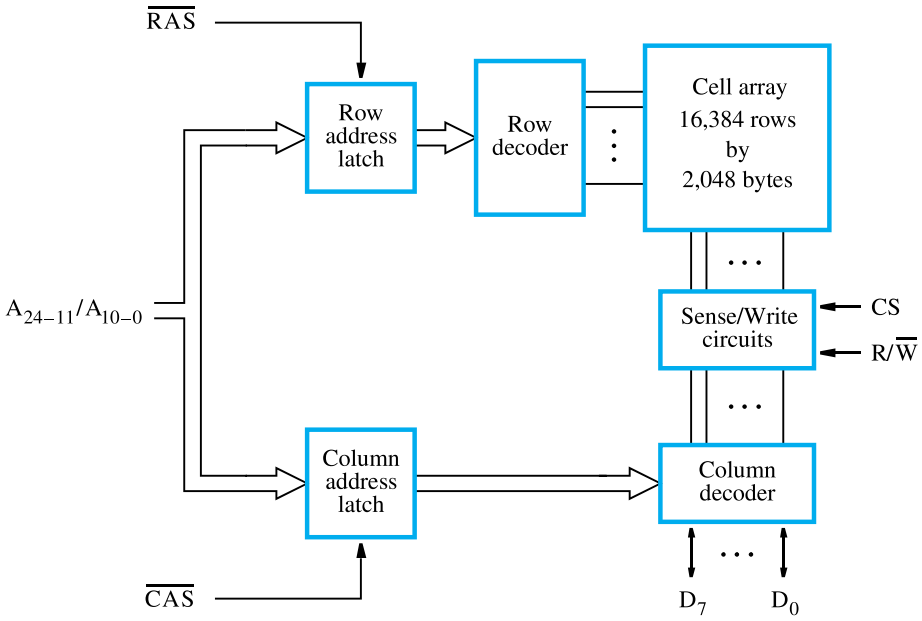

Figure 8.7 Internal organization of a 32M × 8 dynamic memory chip.

conduct a tiny amount of current, measured in picoamperes, after it is turned off. Hence, the information stored in the cell can be retrieved correctly only if it is read before the charge in the capacitor drops below some threshold value. During a Read operation, the transistor in a selected cell is turned on. A sense amplifier connected to the bit line detects whether the charge stored in the capacitor is above or below the threshold value. If the charge is above the threshold, the sense amplifier drives the bit line to the full voltage representing the logic value 1. As a result, the capacitor is recharged to the full charge corresponding to the logic value 1. If the sense amplifier detects that the charge in the capacitor is below the threshold value, it pulls the bit line to ground level to discharge the capacitor fully. Thus, reading the contents of a cell automatically refreshes its contents. Since the word line is common to all cells in a row, all cells in a selected row are read and refreshed at the same time.

A 256-Megabit DRAM chip, configured as 32M × 8, is shown in Figure 8.7. The cells are organized in the form of a 16K × 16K array. The 16,384 cells in each row are divided into 2,048 groups of 8, forming 2,048 bytes of data. Therefore, 14 address bits are needed to select a row, and another 11 bits are needed to specify a group of 8 bits in the selected row. In total, a 25-bit address is needed to access a byte in this memory. The high-order 14 bits and the low-order 11 bits of the address constitute the row and column addresses of a byte, respectively. To reduce the number of pins needed for external connections, the row and column addresses are multiplexed on 14 pins. During a Read or a Write operation, the row address is applied first. It is loaded into the row address latch in response to a signal pulse on an input control line called the Row Address Strobe (RAS). This causes a Read operation to be initiated, in which all cells in the selected row are read and refreshed.

Shortly after the row address is loaded, the column address is applied to the address pins and loaded into the column address latch under control of a second control line called the Column Address Strobe (CAS). The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits is selected. If the R/$\overline{W}$ control signal indicates a Read operation, the output values of the selected circuits are transferred to the data lines, D_{7-0}. For a Write operation, the information on the D_{7-0} lines is transferred to the selected circuits, then used to overwrite the contents of the selected cells in the corresponding 8 columns. We should note that in commercial DRAM chips, the RAS and CAS control signals are active when low. Hence, addresses are latched when these signals change from high to low. The signals are shown in diagrams as $\overline{RAS}$ and $\overline{CAS}$ to indicate this fact.

The timing of the operation of the DRAM described above is controlled by the RAS and CAS signals. These signals are generated by a memory controller circuit external to the chip when the processor issues a Read or a Write command. During a Read operation, the output data are transferred to the processor after a delay equivalent to the memory's access time. Such memories are referred to as *asynchronous DRAMs*. The memory controller is also responsible for refreshing the data stored in the memory chips, as we describe later.

Fast Page Mode

When the DRAM in Figure 8.7 is accessed, the contents of all 16,384 cells in the selected row are sensed, but only 8 bits are placed on the data lines, D_{7-0}. This byte is selected by the column address, bits A_{10-0}. A simple addition to the circuit makes it possible to access the other bytes in the same row without having to reselect the row. Each sense amplifier also acts as a latch. When a row address is applied, the contents of all cells in the selected row are loaded into the corresponding latches. Then, it is only necessary to apply different column addresses to place the different bytes on the data lines.

This arrangement leads to a very useful feature. All bytes in the selected row can be transferred in sequential order by applying a consecutive sequence of column addresses under the control of successive CAS signals. Thus, a block of data can be transferred at a much faster rate than can be achieved for transfers involving random addresses. The block transfer capability is referred to as the *fast page mode* feature. (A large block of data is often called a page.)

It was pointed out earlier that the vast majority of main memory transactions involve block transfers. The faster rate attainable in the fast page mode makes dynamic RAMs particularly well suited to this environment.

8.2.4 SYNCHRONOUS DRAMS

In the early 1990s, developments in memory technology resulted in DRAMs whose operation is synchronized with a clock signal. Such memories are known as *synchronous DRAMs* (SDRAMs). Their structure is shown in Figure 8.8. The cell array is the same as in asynchronous DRAMs. The distinguishing feature of an SDRAM is the use of a clock signal, the availability of which makes it possible to incorporate control circuitry on the chip that provides many useful features. For example, SDRAMs have built-in refresh circuitry, with a refresh counter to provide the addresses of the rows to be selected for refreshing. As a result, the dynamic nature of these memory chips is almost invisible to the user.

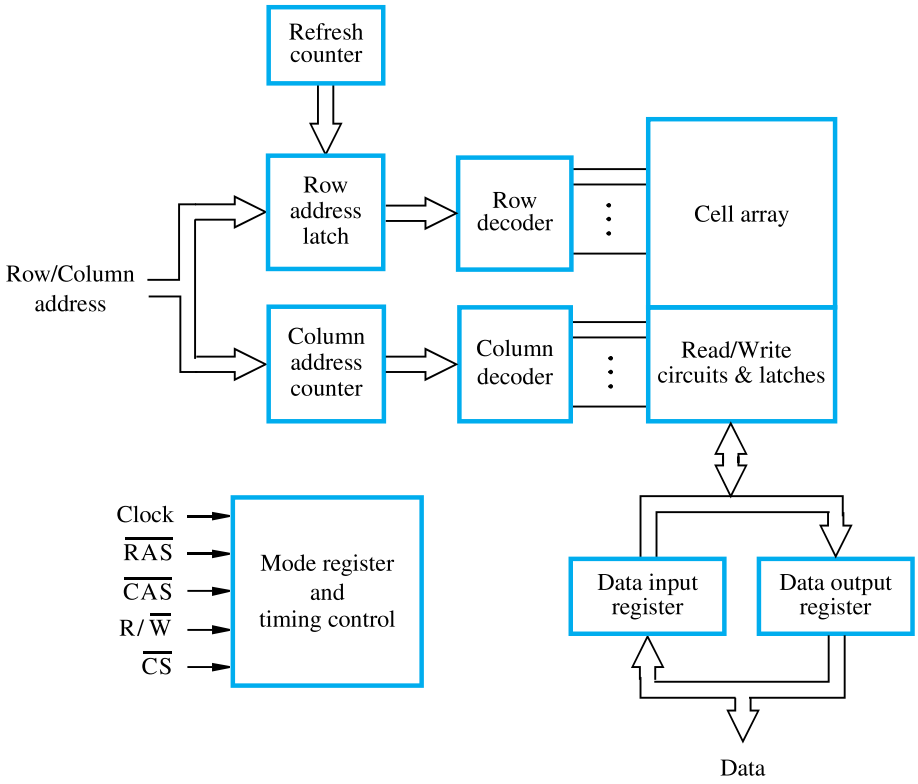

Figure 8.8 Synchronous DRAM.

The address and data connections of an SDRAM may be buffered by means of registers, as shown in the figure. Internally, the Sense/Write amplifiers function as latches, as in asynchronous DRAMs. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. The data in the latches of the selected column are transferred into the data register, thus becoming available on the data output pins. The buffer registers are useful when transferring large blocks of data at very high speed. By isolating external connections from the chip's internal circuitry, it becomes possible to start a new access operation while data are being transferred to or from the registers.

SDRAMs have several different modes of operation, which can be selected by writing control information into a *mode* register. For example, burst operations of different lengths can be specified. It is not necessary to provide externally-generated pulses on the CAS line to select successive columns. The necessary control signals are generated internally using a column counter and the clock signal. New data are placed on the data lines at the rising edge of each clock pulse.

Figure 8.9 shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the RAS signal. The memory typically takes 5 or 6 clock

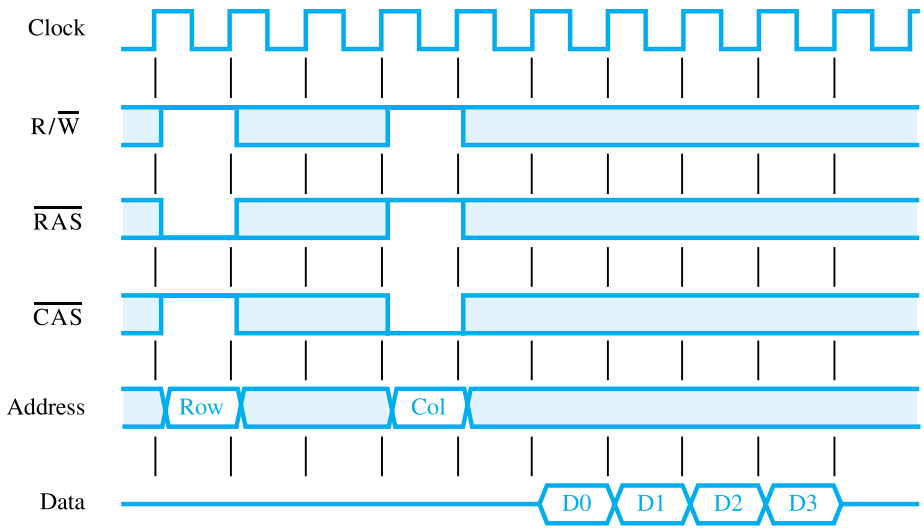

Figure 8.9 A burst read of length 4 in an SDRAM.

cycles (we use 2 in the figure for simplicity) to activate the selected row. Then, the column address is latched under control of the CAS signal. After a delay of one clock cycle, the first set of data bits is placed on the data lines. The SDRAM automatically increments the column address to access the next three sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

Synchronous DRAMs can deliver data at a very high rate, because all the control signals needed are generated inside the chip. The initial commercial SDRAMs in the 1990s were designed for clock speeds of up to 133 MHz. As technology evolved, much faster SDRAM chips were developed. Today's SDRAMs operate with clock speeds that can exceed 1 GHz.

Latency and Bandwidth

Data transfers to and from the main memory often involve blocks of data. The speed of these transfers has a large impact on the performance of a computer system. The memory access time defined earlier is not sufficient for describing the memory's performance when transferring blocks of data. During block transfers, *memory latency* is the amount of time it takes to transfer the first word of a block. The time required to transfer a complete block depends also on the rate at which successive words can be transferred and on the size of the block. The time between successive words of a block is much shorter than the time needed to transfer the first word. For instance, in the timing diagram in Figure 8.9, the access cycle begins with the assertion of the RAS signal. The first word of data is transferred five clock cycles later. Thus, the latency is five clock cycles. If the clock rate is 500 MHz, then the latency is 10 ns. The remaining three words are transferred in consecutive clock cycles, at the rate of one word every 2 ns.

The example above illustrates that we need a parameter other than memory latency to describe the memory's performance during block transfers. A useful performance measure is the number of bits or bytes that can be transferred in one second. This measure is often

referred to as the memory *bandwidth*. It depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel. The rate at which data can be transferred to or from the memory depends on the bandwidth of the system interconnections. For this reason, the interconnections used always ensure that the bandwidth available for data transfers between the processor and the memory is very high.

Double-Data-Rate SDRAM

In the continuous quest for improved performance, faster versions of SDRAMs have been developed. In addition to faster circuits, new organizational and operational features make it possible to achieve high data rates during block transfers. The key idea is to take advantage of the fact that a large number of bits are accessed at the same time inside the chip when a row address is applied. Various techniques are used to transfer these bits quickly to the pins of the chip. To make the best use of the available clock speed, data are transferred externally on both the rising and falling edges of the clock. For this reason, memories that use this technique are called *double-data-rate SDRAMs* (DDR SDRAMs).

Several versions of DDR chips have been developed. The earliest version is known as DDR. Later versions, called DDR2, DDR3, and DDR4, have enhanced capabilities. They offer increased storage capacity, lower power, and faster clock speeds. For example, DDR2 and DDR3 can operate at clock frequencies of 400 and 800 MHz, respectively. Therefore, they transfer data using the effective clock speeds of 800 and 1600 MHz, respectively.

Rambus Memory

The rate of transferring data between the memory and the processor is a function of both the bandwidth of the memory and the bandwidth of its connection to the processor. Rambus is a memory technology that achieves a high data transfer rate by providing a high-speed interface between the memory and the processor. One way for increasing the bandwidth of this connection is to use a wider data path. However, this requires more space and more pins, increasing system cost. The alternative is to use fewer wires with a higher clock speed. This is the approach taken by Rambus.

The key feature of Rambus technology is the use of a differential-signaling technique to transfer data to and from the memory chips. The basic idea of differential signaling is described in Section 7.5.1. In Rambus technology, signals are transmitted using small voltage swings of 0.1 V above and below a reference value. Several versions of this standard have been developed, with clock speeds of up to 800 MHz and data transfer rates of several gigabytes per second.

Rambus technology competes directly with the DDR SDRAM technology. Each has certain advantages and disadvantages. A nontechnical consideration is that the specification of DDR SDRAM is an open standard that can be used free of charge. Rambus, on the other hand, is a proprietary scheme that must be licensed by chip manufacturers.

8.2.5 STRUCTURE OF LARGER MEMORIES

We have discussed the basic organization of memory circuits as they may be implemented on a single chip. Next, we examine how memory chips may be connected to form a much larger memory.

Static Memory Systems

Consider a memory consisting of 2M words of 32 bits each. Figure 8.10 shows how this memory can be implemented using 512K × 8 static memory chips. Each column in the figure implements one byte position in a word, with four chips providing 2M bytes. Four columns implement the required 2M × 32 memory. Each chip has a control input called

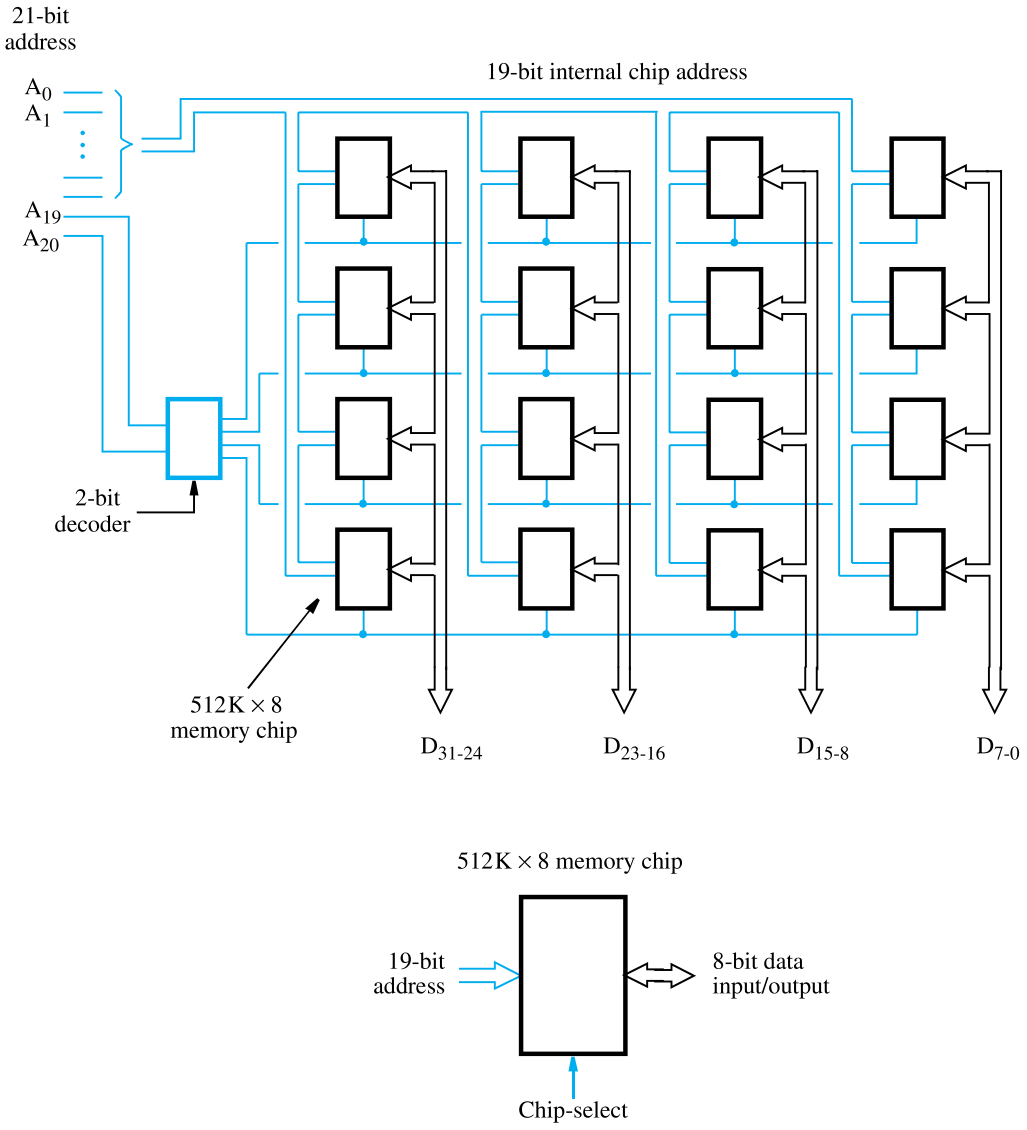

Figure 8.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

Chip-select. When this input is set to 1, it enables the chip to accept data from or to place data on its data lines. The data output for each chip is of the tri-state type described in Section 7.2.3. Only the selected chip places data on the data output line, while all other outputs are electrically disconnected from the data lines. Twenty-one address bits are needed to select a 32-bit word in this memory. The high-order two bits of the address are decoded to determine which of the four rows should be selected. The remaining 19 address bits are used to access specific byte locations inside each chip in the selected row. The R/$\overline{W}$ inputs of all chips are tied together to provide a common Read/$\overline{\text{Write}}$ control line (not shown in the figure).

Dynamic Memory Systems

Modern computers use very large memories. Even a small personal computer is likely to have at least 1G bytes of memory. Typical desktop computers may have 4G bytes or more of memory. A large memory leads to better performance, because more of the programs and data used in processing can be held in the memory, thus reducing the frequency of access to secondary storage.

Because of their high bit density and low cost, dynamic RAMs, mostly of the synchronous type, are widely used in the memory units of computers. They are slower than static RAMs, but they use less power and have considerably lower cost per bit. Available chips have capacities as high as 2G bits, and even larger chips are being developed. To reduce the number of memory chips needed in a given computer, a memory chip may be organized to read or write a number of bits in parallel, as in the case of Figure 8.7. Chips are manufactured in different organizations, to provide flexibility in designing memory systems. For example, a 1-Gbit chip may be organized as 256M $\times$ 4, or 128M $\times$ 8.

Packaging considerations have led to the development of assemblies known as memory modules. Each such module houses many memory chips, typically in the range 16 to 32, on a small board that plugs into a socket on the computer's motherboard. Memory modules are commonly called *SIMM*s (Single In-line Memory Modules) or *DIMM*s (Dual In-line Memory Modules), depending on the configuration of the pins. Modules of different sizes are designed to use the same socket. For example, 128M $\times$ 64, 256M $\times$ 64, and 512M $\times$ 64 bit DIMMs all use the same 240-pin socket. Thus, total memory capacity is easily expanded by replacing a smaller module with a larger one, using the same socket.

Memory Controller

The address applied to dynamic RAM chips is divided into two parts, as explained earlier. The high-order address bits, which select a row in the cell array, are provided first and latched into the memory chip under control of the RAS signal. Then, the low-order address bits, which select a column, are provided on the same address pins and latched under control of the CAS signal. Since a typical processor issues all bits of an address at the same time, a multiplexer is required. This function is usually performed by a *memory controller* circuit. The controller accepts a complete address and the R/$\overline{W}$ signal from the processor, under control of a *Request* signal which indicates that a memory access operation is needed. It forwards the R/$\overline{W}$ signals and the row and column portions of the address to the memory and generates the RAS and CAS signals, with the appropriate timing. When a memory includes multiple modules, one of these modules is selected based on the high-order bits

of the address. The memory controller decodes these high-order bits and generates the chip-select signal for the appropriate module. Data lines are connected directly between the processor and the memory.

Dynamic RAMs must be refreshed periodically. The circuitry required to initiate refresh cycles is included as part of the internal control circuitry of synchronous DRAMs. However, a control circuit external to the chip is needed to initiate periodic Read cycles to refresh the cells of an asynchronous DRAM. The memory controller provides this capability.

Refresh Overhead

A dynamic RAM cannot respond to read or write requests while an internal refresh operation is taking place. Such requests are delayed until the refresh cycle is completed. However, the time lost to accommodate refresh operations is very small. For example, consider an SDRAM in which each row needs to be refreshed once every 64 ms. Suppose that the minimum time between two row accesses is 50 ns and that refresh operations are arranged such that all rows of the chip are refreshed in 8K (8192) refresh cycles. Thus, it takes $8192 \times 0.050 = 0.41$ ms to refresh all rows. The refresh overhead is $0.41/64 = 0.0064$, which is less than 1 percent of the total time available for accessing the memory.

Choice of Technology

The choice of a RAM chip for a given application depends on several factors. Foremost among these are the cost, speed, power dissipation, and size of the chip.

Static RAMs are characterized by their very fast operation. However, their cost and bit density are adversely affected by the complexity of the circuit that realizes the basic cell. They are used mostly where a small but very fast memory is needed. Dynamic RAMs, on the other hand, have high bit densities and a low cost per bit. Synchronous DRAMs are the predominant choice for implementing the main memory.

8.3 READ-ONLY MEMORIES

Both static and dynamic RAM chips are volatile, which means that they retain information only while power is turned on. There are many applications requiring memory devices that retain the stored information when power is turned off. For example, Chapter 4 describes the need to store a small program in such a memory, to be used to start the bootstrap process of loading the operating system from a hard disk into the main memory. The embedded applications described in Chapters 10 and 11 are another important example. Many embedded applications do not use a hard disk and require nonvolatile memories to store their software.

Different types of nonvolatile memories have been developed. Generally, their contents can be read in the same way as for their volatile counterparts discussed above. But, a special writing process is needed to place the information into a nonvolatile memory. Since its normal operation involves only reading the stored data, a memory of this type is called a *read-only memory* (ROM).

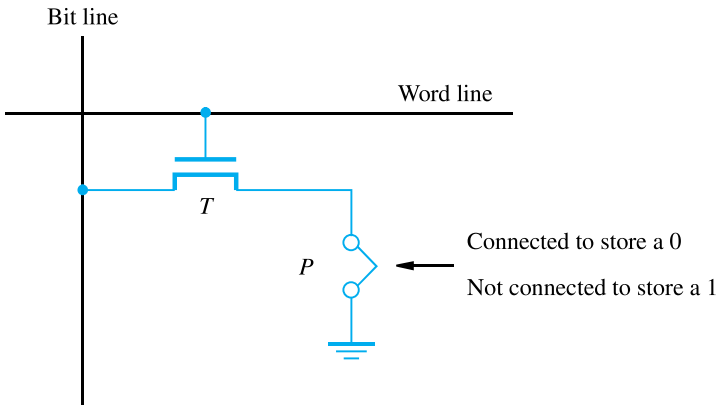

Figure 8.11 A ROM cell.

8.3.1 ROM

A memory is called a read-only memory, or ROM, when information can be written into it only once at the time of manufacture. Figure 8.11 shows a possible configuration for a ROM cell. A logic value 0 is stored in the cell if the transistor is connected to ground at point *P*; otherwise, a 1 is stored. The bit line is connected through a resistor to the power supply. To read the state of the cell, the word line is activated to close the transistor switch. As a result, the voltage on the bit line drops to near zero if there is a connection between the transistor and ground. If there is no connection to ground, the bit line remains at the high voltage level, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. The state of the connection to ground in each cell is determined when the chip is manufactured, using a mask with a pattern that represents the information to be stored.

8.3.2 PROM

Some ROM designs allow the data to be loaded by the user, thus providing a *programmable ROM* (PROM). Programmability is achieved by inserting a fuse at point *P* in Figure 8.11. Before it is programmed, the memory contains all 0s. The user can insert 1s at the required locations by burning out the fuses at these locations using high-current pulses. Of course, this process is irreversible.

PROMs provide flexibility and convenience not available with ROMs. The cost of preparing the masks needed for storing a particular information pattern makes ROMs cost-effective only in large volumes. The alternative technology of PROMs provides a more convenient and considerably less expensive approach, because memory chips can be programmed directly by the user.

8.3.3 EPROM

Another type of ROM chip provides an even higher level of convenience. It allows the stored data to be erased and new data to be written into it. Such an *erasable*, reprogrammable ROM is usually called an *EPROM*. It provides considerable flexibility during the development phase of digital systems. Since EPROMs are capable of retaining stored information for a long time, they can be used in place of ROMs or PROMs while software is being developed. In this way, memory changes and updates can be easily made.

An EPROM cell has a structure similar to the ROM cell in Figure 8.11. However, the connection to ground at point P is made through a special transistor. The transistor is normally turned off, creating an open switch. It can be turned on by injecting charge into it that becomes trapped inside. Thus, an EPROM cell can be used to construct a memory in the same way as the previously discussed ROM cell. Erasure requires dissipating the charge trapped in the transistors that form the memory cells. This can be done by exposing the chip to ultraviolet light, which erases the entire contents of the chip. To make this possible, EPROM chips are mounted in packages that have transparent windows.

8.3.4 EEPROM

An EPROM must be physically removed from the circuit for reprogramming. Also, the stored information cannot be erased selectively. The entire contents of the chip are erased when exposed to ultraviolet light. Another type of erasable PROM can be programmed, erased, and reprogrammed electrically. Such a chip is called an *electrically erasable* PROM, or EEPROM. It does not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. One disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data, which increases circuit complexity. However, this disadvantage is outweighed by the many advantages of EEPROMs. They have replaced EPROMs in practice.

8.3.5 FLASH MEMORY

An approach similar to EEPROM technology has given rise to *flash memory* devices. A flash cell is based on a single transistor controlled by trapped charge, much like an EEPROM cell. Also like an EEPROM, it is possible to read the contents of a single cell. The key difference is that, in a flash device, it is only possible to write an entire block of cells. Prior to writing, the previous contents of the block are erased. Flash devices have greater density, which leads to higher capacity and a lower cost per bit. They require a single power supply voltage, and consume less power in their operation.

The low power consumption of flash memories makes them attractive for use in portable, battery-powered equipment. Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players. In hand-held computers and cell phones, a flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive. A flash memory is used in digital cameras to store picture data. In MP3 players, flash memories store the data that represent sound. Cell phones, digital

cameras, and MP3 players are good examples of embedded systems, which are discussed in Chapters 10 and 11.

Single flash chips may not provide sufficient storage capacity for the applications mentioned above. Larger memory modules consisting of a number of chips are used where needed. There are two popular choices for the implementation of such modules: flash cards and flash drives.

Flash Cards

One way of constructing a larger module is to mount flash chips on a small card. Such flash cards have a standard interface that makes them usable in a variety of products. A card is simply plugged into a conveniently accessible slot. Flash cards with a USB interface are widely used and are commonly known as memory keys. They come in a variety of memory sizes. Larger cards may hold as much as 32 Gbytes. A minute of music can be stored in about 1 Mbyte of memory, using the MP3 encoding format. Hence, a 32-Gbyte flash card can store approximately 500 hours of music.

Flash Drives

Larger flash memory modules have been developed to replace hard disk drives, and hence are called flash drives. They are designed to fully emulate hard disks, to the point that they can be fitted into standard disk drive bays. However, the storage capacity of flash drives is significantly lower. Currently, the capacity of flash drives is on the order of 64 to 128 Gbytes. In contrast, hard disks have capacities exceeding a terabyte. Also, disk drives have a very low cost per bit.

The fact that flash drives are solid state electronic devices with no moving parts provides important advantages over disk drives. They have shorter access times, which result in a faster response. They are insensitive to vibration and they have lower power consumption, which makes them attractive for portable, battery-driven applications.

8.4 DIRECT MEMORY ACCESS

Blocks of data are often transferred between the main memory and I/O devices such as disks. This section discusses a technique for controlling such transfers without frequent, program-controlled intervention by the processor.

The discussion in Chapter 3 concentrates on single-word or single-byte data transfers between the processor and I/O devices. Data are transferred from an I/O device to the memory by first reading them from the I/O device using an instruction such as

```
Load    R2, DATAIN
```

which loads the data into a processor register. Then, the data read are stored into a memory location. The reverse process takes place for transferring data from the memory to an I/O device. An instruction to transfer input or output data is executed only after the processor determines that the I/O device is ready, either by polling its status register or by waiting for an interrupt request. In either case, considerable overhead is incurred, because several program instructions must be executed involving many memory accesses for each data word

transferred. When transferring a block of data, instructions are needed to increment the memory address and keep track of the word count. The use of interrupts involves operating system routines which incur additional overhead to save and restore processor registers, the program counter, and other state information.

An alternative approach is used to transfer blocks of data directly between the main memory and I/O devices, such as disks. A special control unit is provided to manage the transfer, without continuous intervention by the processor. This approach is called *direct memory access*, or DMA. The unit that controls DMA transfers is referred to as a *DMA controller*. It may be part of the I/O device interface, or it may be a separate unit shared by a number of I/O devices. The DMA controller performs the functions that would normally be carried out by the processor when accessing the main memory. For each word transferred, it provides the memory address and generates all the control signals needed. It increments the memory address for successive words and keeps track of the number of transfers.

Although a DMA controller transfers data without intervention by the processor, its operation must be under the control of a program executed by the processor, usually an operating system routine. To initiate the transfer of a block of words, the processor sends to the DMA controller the starting address, the number of words in the block, and the direction of the transfer. The DMA controller then proceeds to perform the requested operation. When the entire block has been transferred, it informs the processor by raising an interrupt.

Figure 8.12 shows an example of the DMA controller registers that are accessed by the processor to initiate data transfer operations. Two registers are used for storing the starting address and the word count. The third register contains status and control flags. The R/$\overline{W}$ bit determines the direction of the transfer. When this bit is set to 1 by a program instruction, the controller performs a Read operation, that is, it transfers data from the memory to the I/O device. Otherwise, it performs a Write operation. Additional information is also transferred as may be required by the I/O device. For example, in the case of a disk, the processor provides the disk controller with information to identify where the data is located on the disk (see Section 8.10.1 for disk details).

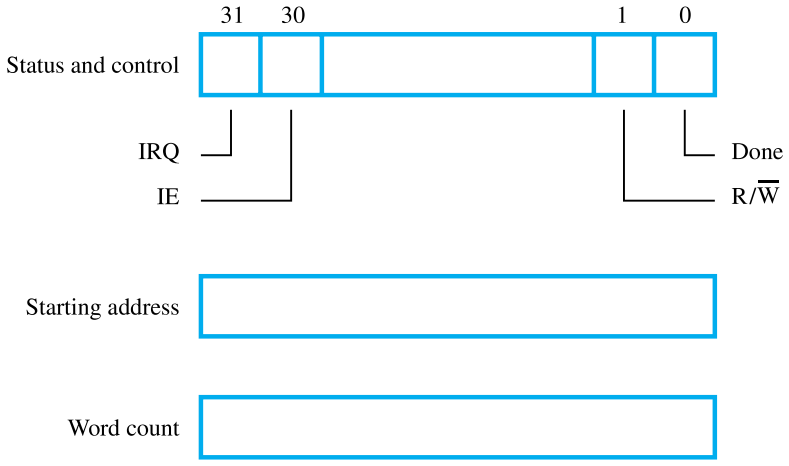

Figure 8.12 Typical registers in a DMA controller.

When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1. Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data. Finally, the controller sets the IRQ bit to 1 when it has requested an interrupt.

Figure 8.13 shows how DMA controllers may be used in a computer system such as that in Figure 7.18. One DMA controller connects a high-speed Ethernet to the computer's I/O bus (a PCI bus in the case of Figure 7.18). The disk controller, which controls two disks, also has DMA capability and provides two DMA channels. It can perform two independent DMA operations, as if each disk had its own DMA controller. The registers needed to store the memory address, the word count, and so on, are duplicated, so that one set can be used with each disk.

To start a DMA transfer of a block of data from the main memory to one of the disks, an OS routine writes the address and word count information into the registers of the disk controller. The DMA controller proceeds independently to implement the specified operation. When the transfer is completed, this fact is recorded in the status and control register of the DMA channel by setting the Done bit. At the same time, if the IE bit is set, the controller sends an interrupt request to the processor and sets the IRQ bit. The status register may also be used to record other information, such as whether the transfer took place correctly or errors occurred.

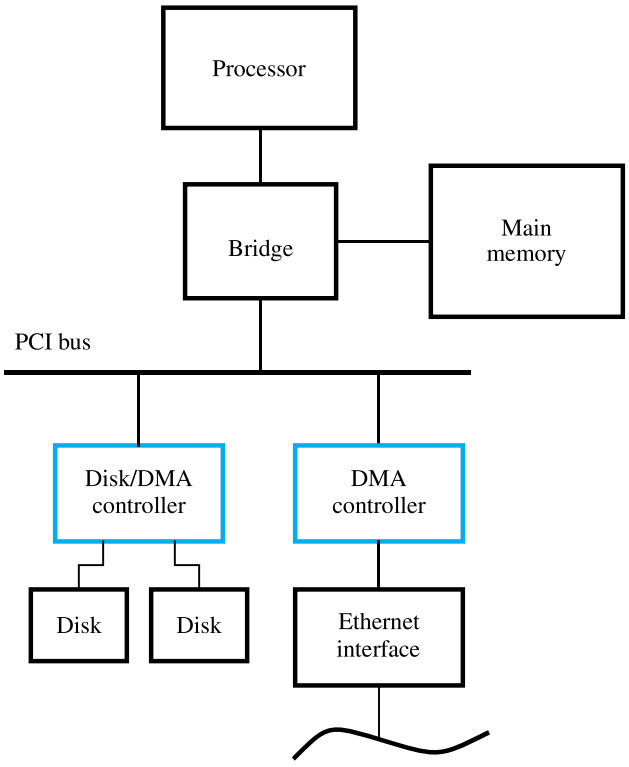

Figure 8.13 Use of DMA controllers in a computer system.

8.5 MEMORY HIERARCHY

We have already stated that an ideal memory would be fast, large, and inexpensive. From the discussion in Section 8.2, it is clear that a very fast memory can be implemented using static RAM chips. But, these chips are not suitable for implementing large memories, because their basic cells are larger and consume more power than dynamic RAM cells.

Although dynamic memory units with gigabyte capacities can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data. A solution is provided by using secondary storage, mainly magnetic disks, to provide the required memory space. Disks are available at a reasonable cost, and they are used extensively in computer systems. However, they are much slower than semiconductor memory units. In summary, a very large amount of cost-effective storage can be provided by magnetic disks, and a large and considerably faster, yet affordable, main memory can be built with dynamic RAM technology. This leaves the more expensive and much faster static RAM technology to be used in smaller units where speed is of the essence, such as in cache memories.

All of these different types of memory units are employed effectively in a computer system. The entire computer memory can be viewed as the hierarchy depicted in Figure 8.14. The fastest access is to data held in processor registers. Therefore, if we consider the

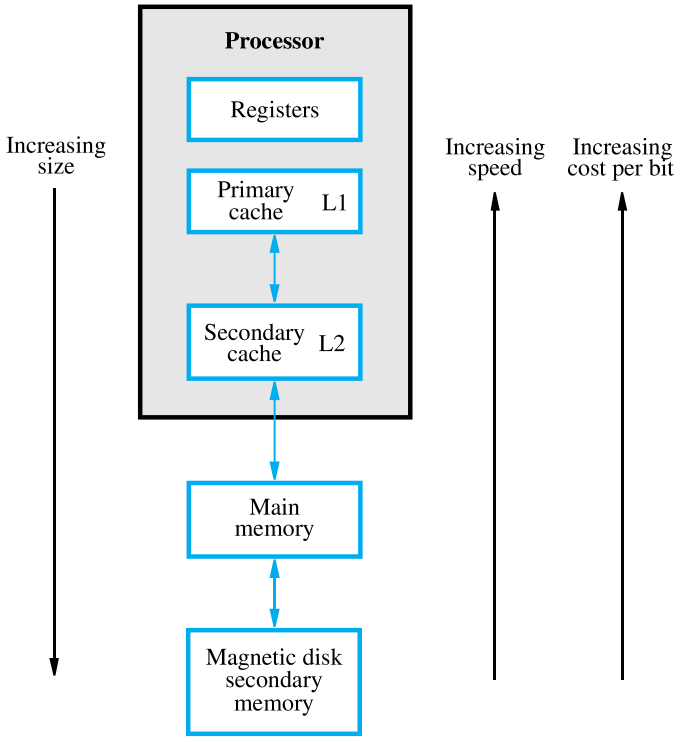

Figure 8.14 Memory hierarchy.

registers to be part of the memory hierarchy, then the processor registers are at the top in terms of speed of access. Of course, the registers provide only a minuscule portion of the required memory.

At the next level of the hierarchy is a relatively small amount of memory that can be implemented directly on the processor chip. This memory, called a *processor cache*, holds copies of the instructions and data stored in a much larger memory that is provided externally. The cache memory concept was introduced in Section 1.2.2 and is examined in detail in Section 8.6. There are often two or more levels of cache. A primary cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers. The primary cache is referred to as the *level 1* (L1) cache. A larger, and hence somewhat slower, secondary cache is placed between the primary cache and the rest of the memory. It is referred to as the *level 2* (L2) cache. Often, the L2 cache is also housed on the processor chip.

Some computers have a *level 3* (L3) cache of even larger size, in addition to the L1 and L2 caches. An L3 cache, also implemented in SRAM technology, may or may not be on the same chip with the processor and the L1 and L2 caches.

The next level in the hierarchy is the *main memory*. This is a large memory implemented using dynamic memory components, typically assembled in memory modules such as DIMMs, as described in Section 8.2.5. The main memory is much larger but significantly slower than cache memories. In a computer with a processor clock of 2 GHz or higher, the access time for the main memory can be as much as 100 times longer than the access time for the L1 cache.

Disk devices provide a very large amount of inexpensive memory, and they are widely used as secondary storage in computer systems. They are very slow compared to the main memory. They represent the bottom level in the memory hierarchy.

During program execution, the speed of memory access is of utmost importance. The key to managing the operation of the hierarchical memory system in Figure 8.14 is to bring the instructions and data that are about to be used as close to the processor as possible. This is the main purpose of using cache memories, which we discuss next.

8.6 CACHE MEMORIES

The cache is a small and very fast memory, interposed between the processor and the main memory. Its purpose is to make the main memory appear to the processor to be much faster than it actually is. The effectiveness of this approach is based on a property of computer programs called *locality of reference*. Analysis of programs shows that most of their execution time is spent in routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. The actual detailed pattern of instruction sequencing is not important—the point is that many instructions in localized areas of the program are executed repeatedly during some time period. This behavior manifests itself in two ways: temporal and spatial. The first means that a recently executed instruction is likely to be executed again very soon. The spatial aspect means that instructions close to a recently executed instruction are also likely to be executed soon.

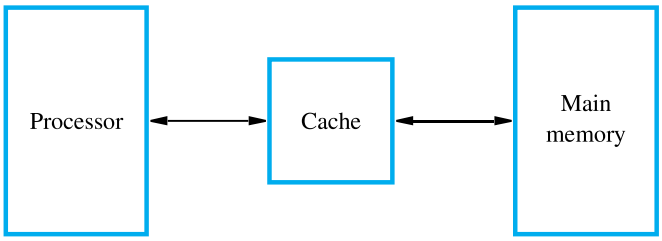

Figure 8.15 Use of a cache memory.

Conceptually, operation of a cache memory is very simple. The memory control circuitry is designed to take advantage of the property of locality of reference. Temporal locality suggests that whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon. Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as well. The term *cache block* refers to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is a *cache line*.

Consider the arrangement in Figure 8.15. When the processor issues a Read request, the contents of a block of memory words containing the location specified are transferred into the cache. Subsequently, when the program references any of the locations in this block, the desired contents are read directly from the cache. Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a *mapping function*. When the cache is full and a memory word (instruction or data) that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word. The collection of rules for making this decision constitutes the cache's *replacement algorithm*.

Cache Hits

The processor does not need to know explicitly about the existence of the cache. It simply issues Read and Write requests using addresses that refer to locations in the memory. The cache control circuitry determines whether the requested word currently exists in the cache. If it does, the Read or Write operation is performed on the appropriate cache location. In this case, a *read* or *write hit* is said to have occurred. The main memory is not involved when there is a cache hit in a Read operation. For a Write operation, the system can proceed in one of two ways. In the first technique, called the *write-through* protocol, both the cache location and the main memory location are updated. The second technique is to update only the cache location and to mark the block containing it with an associated flag bit, often called the *dirty* or *modified bit*. The main memory location of the word is updated later, when the block containing this marked word is removed from the cache to make room for a new block. This technique is known as the *write-back*, or *copy-back*, protocol.

The write-through protocol is simpler than the write-back protocol, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache. The write-back protocol is used most often, to take advantage of the high speed with which data blocks can be transferred to memory chips.

Cache Misses

A Read operation for a word that is not in the cache constitutes a *Read miss*. It causes the block of words containing the requested word to be copied from the main memory into the cache. After the entire block is loaded into the cache, the particular word requested is forwarded to the processor. Alternatively, this word may be sent to the processor as soon as it is read from the main memory. The latter approach, which is called *load-through*, or *early restart*, reduces the processor's waiting time somewhat, at the expense of more complex circuitry.

When a *Write miss* occurs in a computer that uses the write-through protocol, the information is written directly into the main memory. For the write-back protocol, the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.

Recall from Section 6.7 that resource limitations in a pipelined processor can cause instruction execution to stall for one or more cycles. This can occur if a Load or Store instruction requests access to data in the memory at the same time that a subsequent instruction is being fetched. When this happens, instruction fetch is delayed until the data access operation is completed. To avoid stalling the pipeline, many processors use separate caches for instructions and data, making it possible for the two operations to proceed in parallel.

8.6.1 MAPPING FUNCTIONS

There are several possible methods for determining where memory blocks are placed in the cache. It is instructive to describe these methods using a specific small example. Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each. For simplicity, we have assumed that consecutive addresses refer to consecutive words.

Direct Mapping

The simplest way to determine cache locations in which to store memory blocks is the *direct-mapping* technique. In this technique, block j of the main memory maps onto block j modulo 128 of the cache, as depicted in Figure 8.16. Thus, whenever one of the main memory blocks $0, 128, 256, \ldots$ is loaded into the cache, it is stored in cache block 0. Blocks $1, 129, 257, \ldots$ are stored in cache block 1, and so on. Since more than one memory block is mapped onto a given cache block position, contention may arise for that position even when the cache is not full. For example, instructions of a program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed,

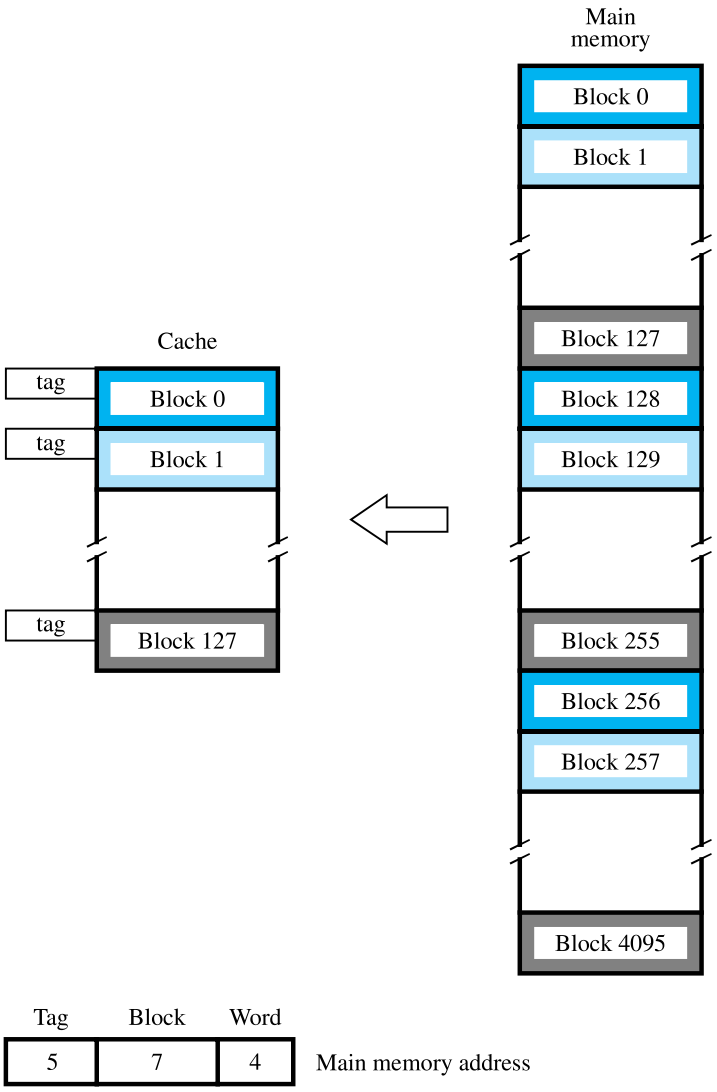

Figure 8.16 Direct-mapped cache.

both of these blocks must be transferred to the block-1 position in the cache. Contention is resolved by allowing the new block to overwrite the currently resident block.

With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The memory address can be divided into three fields, as shown in Figure 8.16. The low-order 4 bits select one of 16 words in a block. When a new block enters the cache, the 7-bit cache block field determines the cache position in which this block must be stored. The high-order 5 bits of the memory address of the

block are stored in 5 *tag* bits associated with its location in the cache. The tag bits identify which of the 32 main memory blocks mapped into this cache position is currently resident in the cache. As execution proceeds, the 7-bit cache block field of each address generated by the processor points to a particular block location in the cache. The high-order 5 bits of the address are compared with the tag bits associated with that cache location. If they match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache. The direct-mapping technique is easy to implement, but it is not very flexible.

Associative Mapping

Figure 8.17 shows the most flexible mapping method, in which a main memory block can be placed into any cache block position. In this case, 12 tag bits are required to identify a memory block when it is resident in the cache. The tag bits of an address received from the processor are compared to the tag bits of each block of the cache to see if the desired block is present. This is called the *associative-mapping* technique. It gives complete freedom in

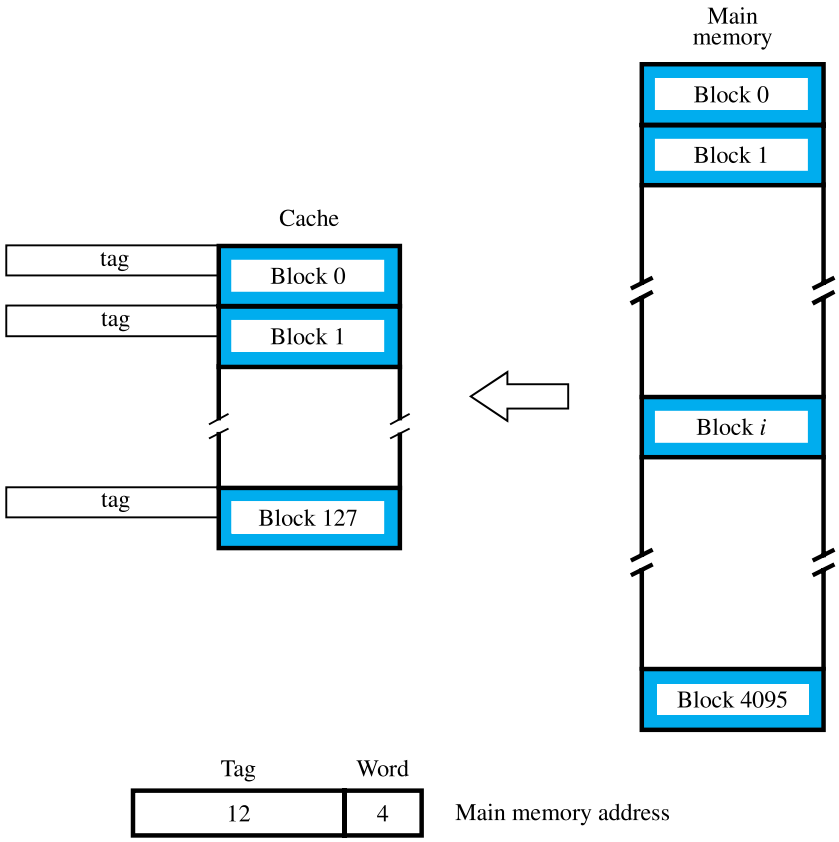

Figure 8.17 Associative-mapped cache.

choosing the cache location in which to place the memory block, resulting in a more efficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. In this case, we need an algorithm to select the block to be replaced. Many replacement algorithms are possible, as we discuss in Section 8.6.2. The complexity of an associative cache is higher than that of a direct-mapped cache, because of the need to search all 128 tag patterns to determine whether a given block is in the cache. To avoid a long delay, the tags must be searched in parallel. A search of this kind is called an *associative search*.

Set-Associative Mapping

Another approach is to use a combination of the direct- and associative-mapping techniques. The blocks of the cache are grouped into sets, and the mapping allows a block of the main memory to reside in any block of a specific set. Hence, the contention problem of the direct method is eased by having a few choices for block placement. At the same time, the hardware cost is reduced by decreasing the size of the associative search. An example of this *set-associative-mapping* technique is shown in Figure 8.18 for a cache with two blocks per set. In this case, memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within this set. Having 64 sets means that the 6-bit set field of the address determines which set of the cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present. This two-way associative search is simple to implement.

The number of blocks per set is a parameter that can be selected to suit the requirements of a particular computer. For the main memory and cache sizes in Figure 8.18, four blocks per set can be accommodated by a 5-bit set field, eight blocks per set by a 4-bit set field, and so on. The extreme condition of 128 blocks per set requires no set bits and corresponds to the fully-associative technique, with 12 tag bits. The other extreme of one block per set is the direct-mapping method. A cache that has k blocks per set is referred to as a k-way set-associative cache.

Stale Data

When power is first turned on, the cache contains no valid data. A control bit, usually called the *valid bit*, must be provided for each cache block to indicate whether the data in that block are valid. This bit should not be confused with the modified, or dirty, bit mentioned earlier. The valid bits of all cache blocks are set to 0 when power is initially applied to the system. Some valid bits may also be set to 0 when new programs or data are loaded from the disk into the main memory. Data transferred from the disk to the main memory using the DMA mechanism are usually loaded directly into the main memory, bypassing the cache. If the memory blocks being updated are currently in the cache, the valid bits of the corresponding cache blocks are set to 0. As program execution proceeds, the valid bit of a given cache block is set to 1 when a memory block is loaded into that location. The processor fetches data from a cache block only if its valid bit is equal to 1. The use of the valid bit in this manner ensures that the processor will not fetch *stale* data from the cache.

A similar precaution is needed in a system that uses the write-back protocol. Under this protocol, new data written into the cache are not written to the memory at the same time.

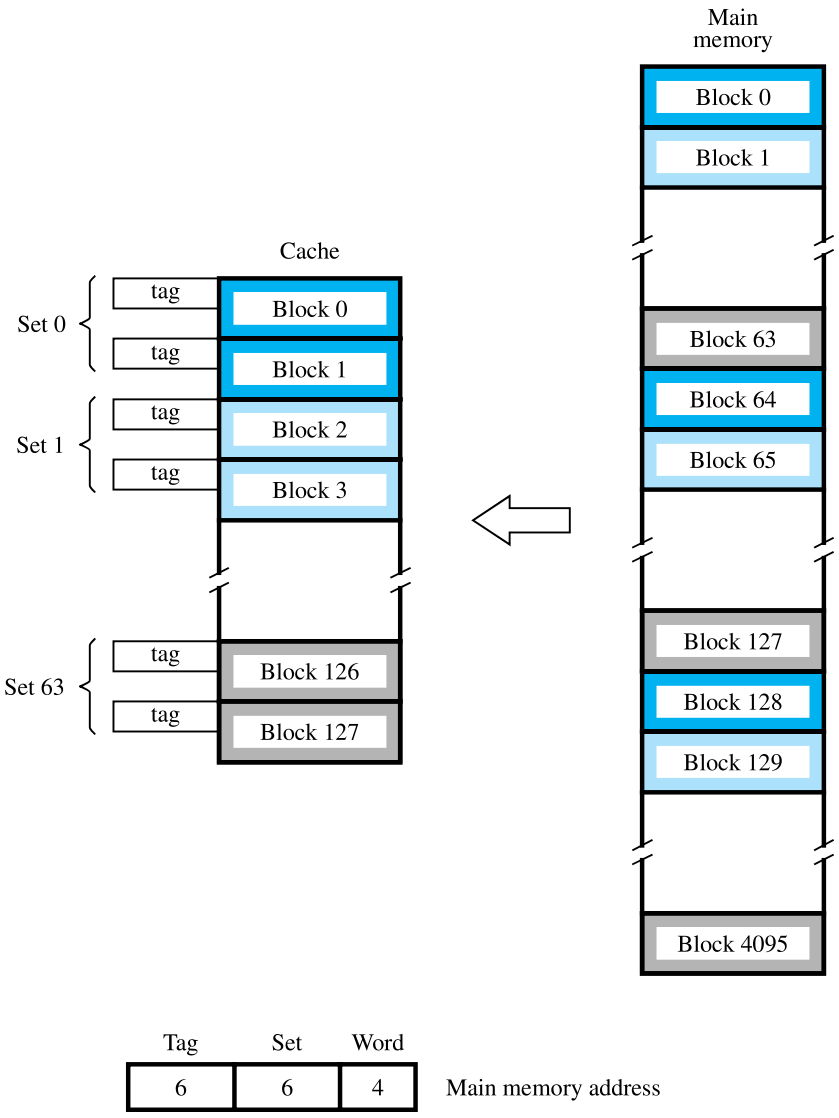

Figure 8.18 Set-associative-mapped cache with two blocks per set.

Hence, data in the memory do not always reflect the changes that may have been made in the cached copy. It is important to ensure that such stale data in the memory are not transferred to the disk. One solution is to *flush* the cache, by forcing all dirty blocks to be written back to the memory before performing the transfer. The operating system can do this by issuing a command to the cache before initiating the DMA operation that transfers the data to the disk. Flushing the cache does not affect performance greatly, because such disk transfers do

not occur often. The need to ensure that two different entities (the processor and the DMA subsystems in this case) use identical copies of the data is referred to as a *cache-coherence* problem.

8.6.2 REPLACEMENT ALGORITHMS

In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. In associative and set-associative caches there exists some flexibility. When a new block is to be brought into the cache and all the positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is an important issue, because the decision can be a strong determining factor in system performance. In general, the objective is to keep blocks in the cache that are likely to be referenced in the near future. But, it is not easy to determine which blocks are about to be referenced. The property of locality of reference in programs gives a clue to a reasonable strategy. Because program execution usually stays in localized areas for reasonable periods of time, there is a high probability that the blocks that have been referenced recently will be referenced again soon. Therefore, when a block is to be overwritten, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the *least recently used* (LRU) block, and the technique is called the *LRU replacement algorithm.*

To use the LRU algorithm, the cache controller must track references to all blocks as computation proceeds. Suppose it is required to track the LRU block of a four-block set in a set-associative cache. A 2-bit counter can be used for each block. When a hit occurs, the counter of the block that is referenced is set to 0. Counters with values originally lower than the referenced one are incremented by one, and all others remain unchanged. When a *miss* occurs and the set is not full, the counter associated with the new block loaded from the main memory is set to 0, and the values of all other counters are increased by one. When a miss occurs and the set is full, the block with the counter value 3 is removed, the new block is put in its place, and its counter is set to 0. The other three block counters are incremented by one. It can be easily verified that the counter values of occupied blocks are always distinct.

The LRU algorithm has been used extensively. Although it performs well for many access patterns, it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache (see Section 8.6.3 and Problem 8.11). Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Several other replacement algorithms are also used in practice. An intuitively reasonable rule would be to remove the "oldest" block from a full set when a new block must be brought in. However, because this algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm in choosing the best blocks to remove. The simplest algorithm is to randomly choose the block to be overwritten. Interestingly enough, this simple algorithm has been found to be quite effective in practice.

8.6.3 EXAMPLES OF MAPPING TECHNIQUES

We now consider a detailed example to illustrate the effects of different cache mapping techniques. Assume that a processor has separate instruction and data caches. To keep the example simple, assume the data cache has space for only eight blocks of data. Also assume that each block consists of only one 16-bit word of data and the memory is word-addressable with 16-bit addresses. (These parameters are not realistic for actual computers, but they allow us to illustrate mapping techniques clearly.) Finally, assume the LRU replacement algorithm is used for block replacement in the cache.

Let us examine changes in the data cache entries caused by running the following application. A 4×10 array of numbers, each occupying one word, is stored in main memory locations 7A00 through 7A27 (hex). The elements of this array, A, are stored in column order, as shown in Figure 8.19. The figure also indicates how tags for different cache mapping techniques are derived from the memory address. Note that no bits are needed to identify a word within a block, as was done in Figures 8.16 through 8.18, because we have assumed that each block contains only one word. The application normalizes the elements of the first row of A with respect to the average value of the elements in the row. Hence, we need to compute the average of the elements in the row and divide each element by that average. The required task can be expressed as

$$A(0,i) \leftarrow \frac{A(0,i)}{\left(\sum_{j=0}^{9} A(0,j)\right)/10} \qquad \text{for } i = 0, 1, \ldots, 9$$

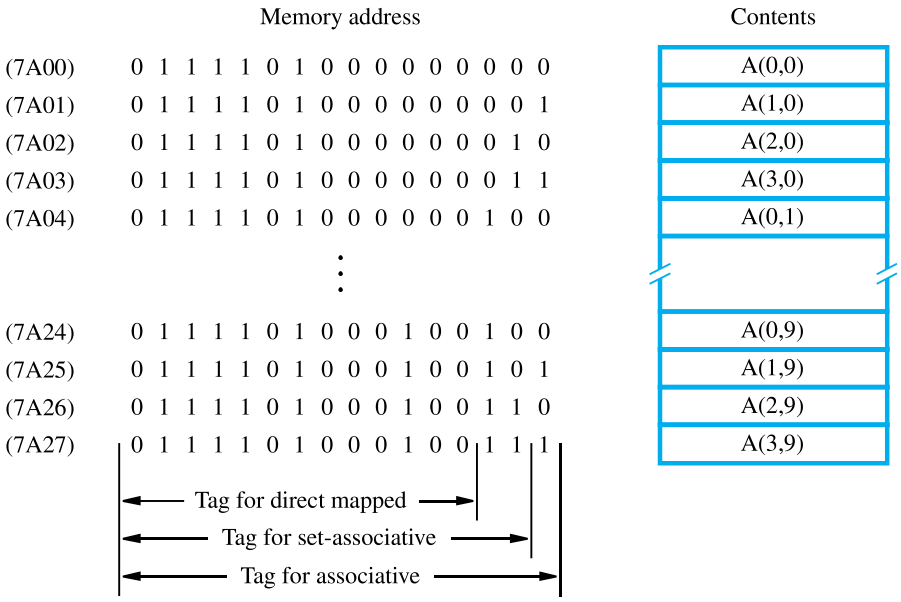

Figure 8.19 An array stored in the main memory.

```
SUM := 0
for j := 0 to 9 do
        SUM := SUM + A(0,j)
end
AVG := SUM/10
for i := 9 downto 0 do
        A(0,i) := A(0,i)/AVG
end
```

Figure 8.20 Task for example in Section 8.6.3.

Figure 8.20 gives the structure of a program that corresponds to this task. We use the variables SUM and AVE to hold the sum and average values, respectively. These variables, as well as index variables i and j, are held in processor registers during the computation.

Direct-Mapped Cache

In a direct-mapped data cache, the contents of the cache change as shown in Figure 8.21. The columns in the table indicate the cache contents after various passes through the two program loops in Figure 8.20 are completed. For example, after the second pass through the first loop ($j = 1$), the cache holds the elements $A(0, 0)$ and $A(0, 1)$. These elements are in block positions 0 and 4, as determined by the three least-significant bits of the address. During the next pass, the $A(0, 0)$ element is replaced by $A(0, 2)$, which maps into the same block position. Note that the desired elements map into only two positions in the cache, thus leaving the contents of the other six positions unchanged from whatever they were before the normalization task started.

Elements $A(0, 8)$ and $A(0, 9)$ are loaded into the cache during the ninth and tenth passes through the first loop ($j = 8, 9$). The second loop reverses the order in which the elements are handled. The first two passes through this loop ($i = 9, 8$) find the required data in the cache. When $i = 7$, element $A(0, 9)$ is replaced with $A(0, 7)$. When $i = 6$, element $A(0, 8)$

	Contents of data cache after pass:								
Block position	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Figure 8.21 Contents of a direct-mapped data cache.

is replaced with $A(0, 6)$, and so on. Thus, eight elements are replaced while the second loop is executed. In total, there are only two hits during execution of this task.

The reader should keep in mind that the tags must be kept in the cache for each block. They are not shown to keep the figure simple.

Associative-Mapped Cache

Figure 8.22 presents the changes in cache contents for the case of an associative-mapped cache. During the first eight passes through the first loop, the elements are brought into consecutive block positions, assuming that the cache was initially empty. During the ninth pass ($j = 8$), the LRU algorithm chooses $A(0, 0)$ to be overwritten by $A(0, 8)$. In the next and last pass through the j loop, element $A(0, 1)$ is replaced with $A(0, 9)$. Now, for the first eight passes through the second loop ($i = 9, 8, \ldots, 2$) all the required elements are found in the cache. When $i = 1$, the element needed is $A(0, 1)$, so it replaces the least recently used element, $A(0, 9)$. During the last pass, $A(0, 0)$ replaces $A(0, 8)$.

In this case, when the second loop is executed, only two elements are not found in the cache. In the direct-mapped case, eight of the elements had to be reloaded during the second loop. Obviously, the associative-mapped cache benefits from the complete freedom in mapping a memory block into any position in the cache. In both cases, better utilization of the cache is achieved by reversing the order in which the elements are handled in the second loop of the program. It is interesting to consider what would happen if the second loop dealt with the elements in the same order as in the first loop. Using either direct mapping or the LRU algorithm, all elements would be overwritten before they are used in the second loop (see Problem 8.10).

Set-Associative-Mapped Cache

For this example, we assume that a set-associative data cache is organized into two sets, each capable of holding four blocks. Thus, the least-significant bit of an address determines which set a memory block maps into, but the memory data can be placed in any of the four blocks of the set. The high-order 15 bits of the address constitute the tag.

	Contents of data cache after pass:				
Block position	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

Figure 8.22 Contents of an associative-mapped data cache.

	Contents of data cache after pass:					
	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Set 1						

Figure 8.23 Contents of a set-associative-mapped data cache.

Changes in the cache contents are depicted in Figure 8.23. Since all the desired blocks have even addresses, they map into set 0. In this case, six elements are reloaded during execution of the second loop.

Even though this is a simplified example, it illustrates that in general, associative mapping performs best, set-associative mapping is next best, and direct mapping is the worst. However, fully-associative mapping is expensive to implement, so set-associative mapping is a good practical compromise.

8.7 PERFORMANCE CONSIDERATIONS

Two key factors in the commercial success of a computer are performance and cost; the best possible performance for a given cost is the objective. A common measure of success is the *price/performance ratio*. Performance depends on how fast machine instructions can be brought into the processor and how fast they can be executed. Chapter 6 shows how pipelining increases the speed of program execution. In this chapter, we focus on the memory subsystem.

The memory hierarchy described in Section 8.5 results from the quest for the best price/performance ratio. The main purpose of this hierarchy is to create a memory that the processor sees as having a short access time and a large capacity. When a cache is used, the processor is able to access instructions and data more quickly when the data from the referenced memory locations are in the cache. Therefore, the extent to which caches improve performance is dependent on how frequently the requested instructions and data are found in the cache. In this section, we examine this issue quantitatively.

8.7.1 HIT RATE AND MISS PENALTY

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various levels of the hierarchy. Recall that a successful access to data in a cache is called a hit. The number of hits stated as a fraction of all attempted accesses is called the *hit rate*, and the *miss rate* is the number of misses stated as a fraction of attempted accesses.

Ideally, the entire memory hierarchy would appear to the processor as a single memory unit that has the access time of the cache on the processor chip and the size of the magnetic disk. How close we get to this ideal depends largely on the hit rate at different levels of the hierarchy. High hit rates well over 0.9 are essential for high-performance computers.

Performance is adversely affected by the actions that need to be taken when a miss occurs. A performance penalty is incurred because of the extra time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. During that period, the processor is stalled waiting for instructions or data. The waiting time depends on the details of the operation of the cache. For example, it depends on whether or not the load-through approach is used. We refer to the total access time seen by the processor when a miss occurs as the *miss penalty*.

Consider a system with only one level of cache. In this case, the miss penalty consists almost entirely of the time to access a block of data in the main memory. Let h be the hit rate, M the miss penalty, and C the time to access information in the cache. Thus, the average access time experienced by the processor is

$$t_{avg} = hC + (1 - h)M$$

The following example illustrates how the values of these parameters affect the average access time.

Example 8.1

Consider a computer that has the following parameters. Access times to the cache and the main memory are τ and 10τ, respectively. When a cache miss occurs, a block of 8 words is transferred from the main memory to the cache. It takes 10τ to transfer the first word of the block, and the remaining 7 words are transferred at the rate of one word every τ seconds. The miss penalty also includes a delay of τ for the initial access to the cache, which misses, and another delay of τ to transfer the word to the processor after the block is loaded into the cache (assuming no load-through). Thus, the miss penalty in this computer is given by:

$$M = \tau + 10\tau + 7\tau + \tau = 19\tau$$

Assume that 30 percent of the instructions in a typical program perform a Read or a Write operation, which means that there are 130 memory accesses for every 100 instructions executed. Assume that the hit rates in the cache are 0.95 for instructions and 0.9 for data. Assume further that the miss penalty is the same for both read and write accesses. Then,

a rough estimate of the improvement in memory performance that results from using the cache can be obtained as follows:

$$\frac{\text{Time without cache}}{\text{Time with cache}} = \frac{130 \times 10\tau}{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)} = 4.7$$

This result shows that the cache makes the memory appear almost five times faster than it really is. The improvement factor increases as the speed of the cache increases relative to the main memory. For example, if the access time of the main memory is 20τ, the improvement factor becomes 7.3.

High hit rates are essential for the cache to be effective in reducing memory access time. Hit rates depend on the size of the cache, its design, and the instruction and data access patterns of the programs being executed. It is instructive to consider how effective the cache of this example is compared to the ideal case in which the hit rate is 100 percent. With ideal cache behavior, all memory references take one τ. Thus, an estimate of the increase in memory access time caused by misses in the cache is given by:

$$\frac{\text{Time for real cache}}{\text{Time for ideal cache}} = \frac{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)}{130\tau} = 2.1$$

In other words, a 100% hit rate in the cache would make the memory appear twice as fast as when realistic hit rates are used.

How can the hit rate be improved? One possibility is to make the cache larger, but this entails increased cost. Another possibility is to increase the cache block size while keeping the total cache size constant, to take advantage of spatial locality. If all items in a larger block are needed in a computation, then it is better to load these items into the cache in a single miss, rather than loading several smaller blocks as a result of several misses. The high data rate achievable during block transfers is the main reason for this advantage. But larger blocks are effective only up to a certain size, beyond which the improvement in the hit rate is offset by the fact that some items may not be referenced before the block is ejected (replaced). Also, larger blocks take longer to transfer, and hence increase the miss penalty. Since the performance of a computer is affected positively by increased hit rate and negatively by increased miss penalty, block size should be neither too small nor too large. In practice, block sizes in the range of 16 to 128 bytes are the most popular choices.

Finally, we note that the miss penalty can be reduced if the load-through approach is used when loading new blocks into the cache. Then, instead of waiting for an entire block to be transferred, the processor resumes execution as soon as the required word is loaded into the cache.

8.7.2 CACHES ON THE PROCESSOR CHIP

When information is transferred between different chips, considerable delays occur in driver and receiver gates on the chips. Thus, it is best to implement the cache on the processor

chip. Most processor chips include at least one L1 cache. Often there are two separate L1 caches, one for instructions and another for data.

In high-performance processors, two levels of caches are normally used, separate L1 caches for instructions and data and a larger L2 cache. These caches are often implemented on the processor chip. In this case, the L1 caches must be very fast, as they determine the memory access time seen by the processor. The L2 cache can be slower, but it should be much larger than the L1 caches to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 caches. A typical computer may have L1 caches with capacities of tens of kilobytes and an L2 cache of hundreds of kilobytes or possibly several megabytes.

Including an L2 cache further reduces the impact of the main memory speed on the performance of a computer. Its effect can be assessed by observing that the average access time of the L2 cache is the miss penalty of either of the L1 caches. For simplicity, we will assume that the hit rates are the same for instructions and data. Thus, the average access time experienced by the processor in such a system is:

$$t_{avg} = h_1C_1 + (1 - h_1)(h_2C_2 + (1 - h_2)M)$$

where

h_1 is the hit rate in the L1 caches.

h_2 is the hit rate in the L2 cache.

C_1 is the time to access information in the L1 caches.

C_2 is the miss penalty to transfer information from the L2 cache to an L1 cache.

M is the miss penalty to transfer information from the main memory to the L2 cache.

Of all memory references made by the processor, the number of misses in the L2 cache is given by $(1 - h_1)(1 - h_2)$. If both h_1 and h_2 are in the 90 percent range, then the number of misses in the L2 cache will be less than one percent of all memory accesses. This makes the value of M, and in turn the speed of the main memory, less critical. See Problem 8.14 for a quantitative examination of this issue.

8.7.3 OTHER ENHANCEMENTS

In addition to the main design issues just discussed, several other possibilities exist for enhancing performance. We discuss three of them in this section.

Write Buffer

When the write-through protocol is used, each Write operation results in writing a new value into the main memory. If the processor must wait for the memory function to be completed, as we have assumed until now, then the processor is slowed down by all Write requests. Yet the processor typically does not need immediate access to the result of a Write operation; so it is not necessary for it to wait for the Write request to be completed.

To improve performance, a *Write buffer* can be included for temporary storage of Write requests. The processor places each Write request into this buffer and continues execution of the next instruction. The Write requests stored in the Write buffer are sent to the main memory whenever the memory is not responding to Read requests. It is important that the Read requests be serviced quickly, because the processor usually cannot proceed before receiving the data being read from the memory. Hence, these requests are given priority over Write requests.

The Write buffer may hold a number of Write requests. Thus, it is possible that a subsequent Read request may refer to data that are still in the Write buffer. To ensure correct operation, the addresses of data to be read from the memory are always compared with the addresses of the data in the Write buffer. In the case of a match, the data in the Write buffer are used.

A similar situation occurs with the write-back protocol. In this case, Write commands issued by the processor are performed on the word in the cache. When a new block of data is to be brought into the cache as a result of a Read miss, it may replace an existing block that has some dirty data. The dirty block has to be written into the main memory. If the required write-back is performed first, then the processor has to wait for this operation to be completed before the new block is read into the cache. It is more prudent to read the new block first. The dirty block being ejected from the cache is temporarily stored in the Write buffer and held there while the new block is being read. Afterwards, the contents of the buffer are written into the main memory. Thus, the Write buffer also works well for the write-back protocol.

Prefetching

In the previous discussion of the cache mechanism, we assumed that new data are brought into the cache when they are first needed. Following a Read miss, the processor has to pause until the new data arrive, thus incurring a miss penalty.

To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed. The simplest way to do this is through software. A special *prefetch* instruction may be provided in the instruction set of the processor. Executing this instruction causes the addressed data to be loaded into the cache, as in the case of a Read miss. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache shortly before they are needed in the program. Then, the processor will not have to wait for the referenced data as in the case of a Read miss. The hope is that prefetching will take place while the processor is busy executing instructions that do not result in a Read miss, thus allowing accesses to the main memory to be overlapped with computation in the processor.

Prefetch instructions can be inserted into a program either by the programmer or by the compiler. Compilers are able to insert these instructions with good success for many applications. Software prefetching entails a certain overhead because inclusion of prefetch instructions increases the length of programs. Moreover, some prefetches may load into the cache data that will not be used by the instructions that follow. This can happen if the prefetched data are ejected from the cache by a Read miss involving other data. However, the overall effect of software prefetching on performance is positive, and many processors have machine instructions to support this feature. See Reference [1] for a thorough discussion of software prefetching.

Prefetching can also be done in hardware, using circuitry that attempts to discover a pattern in memory references and prefetches data according to this pattern. A number of schemes have been proposed for this purpose, as described in References [2] and [3].

Lockup-Free Cache

Software prefetching does not work well if it interferes significantly with the normal execution of instructions. This is the case if the action of prefetching stops other accesses to the cache until the prefetch is completed. While servicing a miss, the cache is said to be locked. This problem can be solved by modifying the basic cache structure to allow the processor to access the cache while a miss is being serviced. In this case, it is possible to have more than one outstanding miss, and the hardware must accommodate such occurrences.

A cache that can support multiple outstanding misses is called *lockup-free*. Such a cache must include circuitry that keeps track of all outstanding misses. This may be done with special registers that hold the pertinent information about these misses. Lockup-free caches were first used in the early 1980s in the Cyber series of computers manufactured by the Control Data company [4].

We have used software prefetching to motivate the need for a cache that is not locked by a Read miss. A much more important reason is that in a pipelined processor, which overlaps the execution of several instructions, a Read miss caused by one instruction could stall the execution of other instructions. A lockup-free cache reduces the likelihood of such stalls.

8.8 VIRTUAL MEMORY

In most modern computer systems, the physical main memory is not as large as the address space of the processor. For example, a processor that issues 32-bit addresses has an addressable space of 4G bytes. The size of the main memory in a typical computer with a 32-bit processor may range from 1G to 4G bytes. If a program does not completely fit into the main memory, the parts of it not currently being executed are stored on a secondary storage device, typically a magnetic disk. As these parts are needed for execution, they must first be brought into the main memory, possibly replacing other parts that are already in the memory. These actions are performed automatically by the operating system, using a scheme known as *virtual memory*. Application programmers need not be aware of the limitations imposed by the available main memory. They prepare programs using the entire address space of the processor.

Under a virtual memory system, programs, and hence the processor, reference instructions and data in an address space that is independent of the available physical main memory space. The binary addresses that the processor issues for either instructions or data are called *virtual* or *logical addresses*. These addresses are translated into physical addresses by a combination of hardware and software actions. If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.

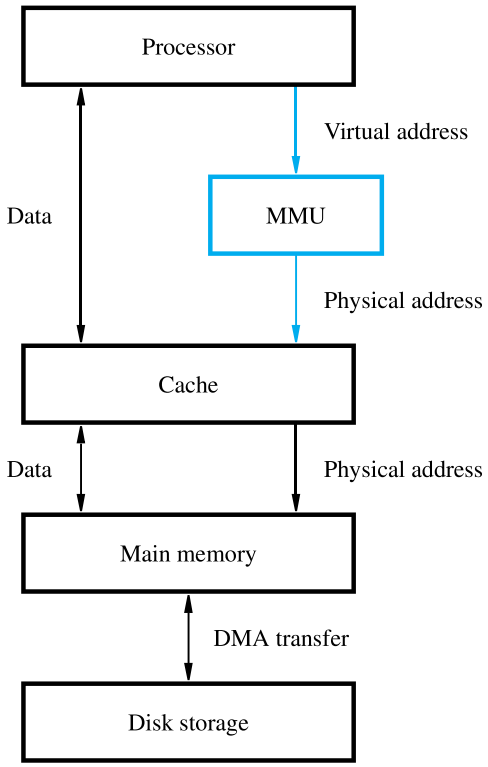

Figure 8.24 Virtual memory organization.

Figure 8.24 shows a typical organization that implements virtual memory. A special hardware unit, called the *Memory Management Unit* (MMU), keeps track of which parts of the virtual address space are in the physical memory. When the desired data or instructions are in the main memory, the MMU translates the virtual address into the corresponding physical address. Then, the requested memory access proceeds in the usual manner. If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory. Such transfers are performed using the DMA scheme discussed in Section 8.4.

8.8.1 ADDRESS TRANSLATION

A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called *pages*, each of which consists of a block of words that occupy contiguous locations in the main memory. Pages commonly range from 2K to 16K bytes in length. They constitute the basic unit of information that is transferred between the main memory and the disk whenever the MMU determines that a transfer is required. Pages should not be too small, because the access time of a magnetic disk is much longer (several milliseconds) than the access time of the main memory. The

reason for this is that it takes a considerable amount of time to locate the data on the disk, but once located, the data can be transferred at a rate of several megabytes per second. On the other hand, if pages are too large, it is possible that a substantial portion of a page may not be used, yet this unnecessary data will occupy valuable space in the main memory.

This discussion clearly parallels the concepts introduced in Section 8.6 on cache memory. The cache bridges the speed gap between the processor and the main memory and is implemented in hardware. The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques. Conceptually, cache techniques and virtual-memory techniques are very similar. They differ mainly in the details of their implementation.

A virtual-memory address-translation method based on the concept of fixed-length pages is shown schematically in Figure 8.25. Each virtual address generated by the proces-

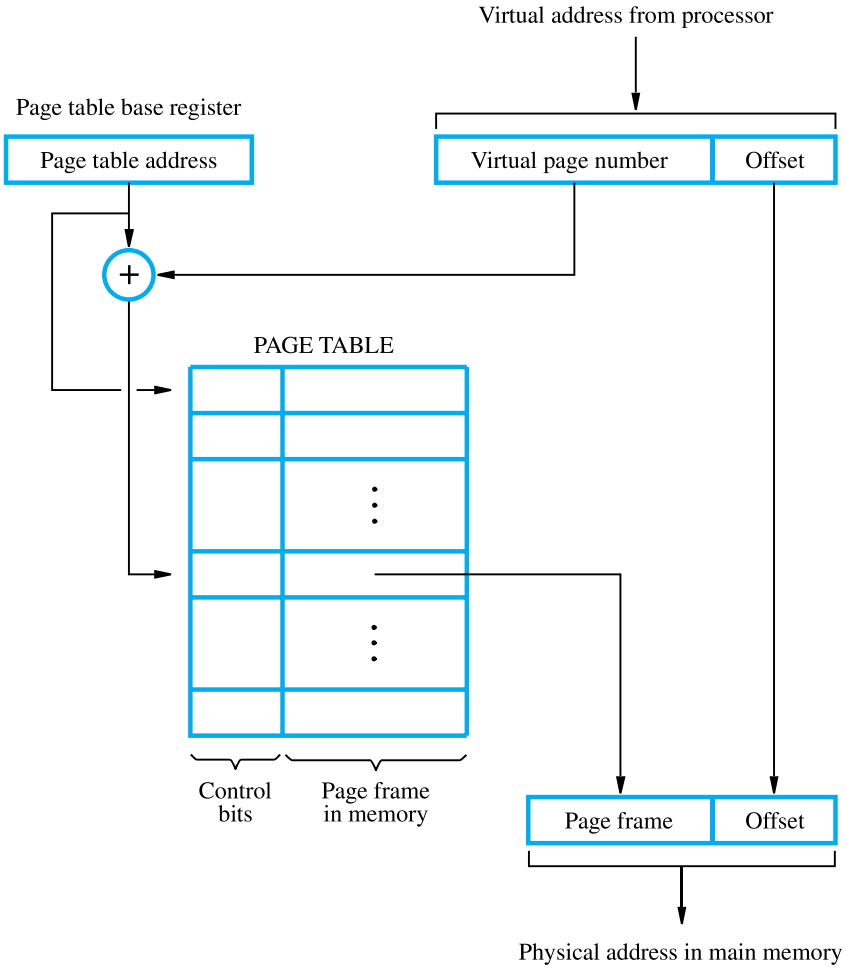

Figure 8.25 Virtual-memory address translation.

sor, whether it is for an instruction fetch or an operand load/store operation, is interpreted as a *virtual page number* (high-order bits) followed by an *offset* (low-order bits) that specifies the location of a particular byte (or word) within a page. Information about the main memory location of each page is kept in a *page table*. This information includes the main memory address where the page is stored and the current status of the page. An area in the main memory that can hold one page is called a *page frame*. The starting address of the page table is kept in a *page table base register*. By adding the virtual page number to the contents of this register, the address of the corresponding entry in the page table is obtained. The contents of this location give the starting address of the page if that page currently resides in the main memory.

Each entry in the page table also includes some control bits that describe the status of the page while it is in the main memory. One bit indicates the validity of the page, that is, whether the page is actually loaded in the main memory. It allows the operating system to invalidate the page without actually removing it. Another bit indicates whether the page has been modified during its residency in the memory. As in cache memories, this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page. Other control bits indicate various restrictions that may be imposed on accessing the page. For example, a program may be given full read and write permission, or it may be restricted to read accesses only.

Translation Lookaside Buffer

The page table information is used by the MMU for every read and write access. Ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large. Since the MMU is normally implemented as part of the processor chip, it is impossible to include the complete table within the MMU. Instead, a copy of only a small portion of the table is accommodated within the MMU, and the complete table is kept in the main memory. The portion maintained within the MMU consists of the entries corresponding to the most recently accessed pages. They are stored in a small table, usually called the *Translation Lookaside Buffer* (TLB). The TLB functions as a cache for the page table in the main memory. Each entry in the TLB includes a copy of the information in the corresponding entry in the page table. In addition, it includes the virtual address of the page, which is needed to search the TLB for a particular page. Figure 8.26 shows a possible organization of a TLB that uses the associative-mapping technique. Set-associative mapped TLBs are also found in commercial products.

Address translation proceeds as follows. Given a virtual address, the MMU looks in the TLB for the referenced page. If the page table entry for this page is found in the TLB, the physical address is obtained immediately. If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated.

It is essential to ensure that the contents of the TLB are always the same as the contents of page tables in the memory. When the operating system changes the contents of a page table, it must simultaneously invalidate the corresponding entries in the TLB. One of the control bits in the TLB is provided for this purpose. When an entry is invalidated, the TLB acquires the new information from the page table in the memory as part of the MMU's normal response to access misses.

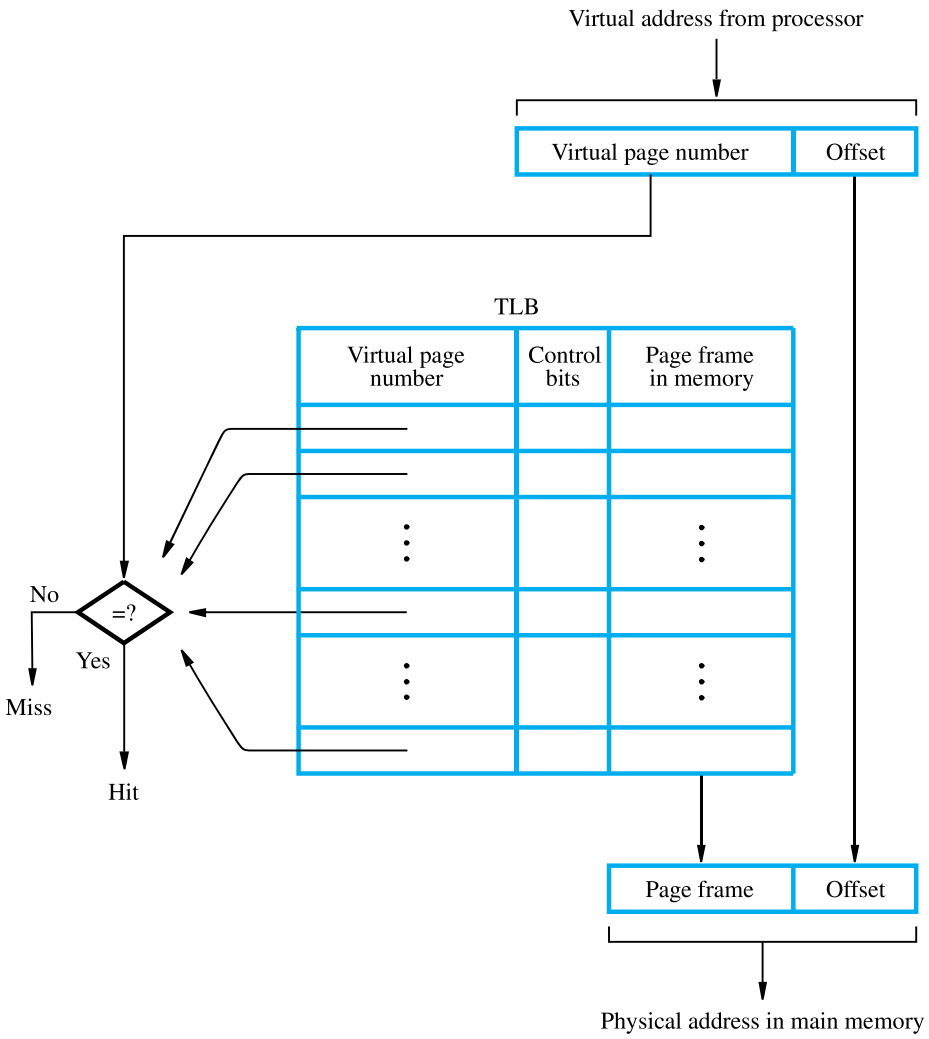

Figure 8.26 Use of an associative-mapped TLB.

Page Faults

When a program generates an access request to a page that is not in the main memory, a *page fault* is said to have occurred. The entire page must be brought from the disk into the memory before access can proceed. When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt). Processing of the program that generated the page fault is interrupted, and control is transferred to the operating system. The operating system copies the requested page from the disk into the main memory. Since this process involves a long delay, the operating system may begin execution of another

program whose pages are in the main memory. When page transfer is completed, the execution of the interrupted program is resumed.

When the MMU raises an interrupt to indicate a page fault, the instruction that requested the memory access may have been partially executed. It is essential to ensure that the interrupted program continues correctly when it resumes execution. There are two options. Either the execution of the interrupted instruction continues from the point of interruption, or the instruction must be restarted. The design of a particular processor dictates which of these two options is used.

If a new page is brought from the disk when the main memory is full, it must replace one of the resident pages. The problem of choosing which page to remove is just as critical here as it is in a cache, and the observation that programs spend most of their time in a few localized areas also applies. Because main memories are considerably larger than cache memories, it should be possible to keep relatively larger portions of a program in the main memory. This reduces the frequency of transfers to and from the disk. Concepts similar to the LRU replacement algorithm can be applied to page replacement, and the control bits in the page table entries can be used to record usage history. One simple scheme is based on a control bit that is set to 1 whenever the corresponding page is referenced (accessed). The operating system periodically clears this bit in all page table entries, thus providing a simple way of determining which pages have not been used recently.

A modified page has to be written back to the disk before it is removed from the main memory. It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data.

Looking up entries in the TLB introduces some delay, slowing down the operation of the MMU. Here again we can take advantage of the property of locality of reference. It is likely that many successive TLB translations involve addresses on the same program page. This is particularly likely when fetching instructions. Thus, address translation time can be reduced by keeping the most recently used TLB entries in a few special registers that can be accessed quickly.

8.9 MEMORY MANAGEMENT REQUIREMENTS

In our discussion of virtual-memory concepts, we have tacitly assumed that only one large program is being executed. If all of the program does not fit into the available physical memory, parts of it (pages) are moved from the disk into the main memory when they are to be executed. Although we have alluded to software routines that are needed to manage this movement of program segments, we have not been specific about the details.

Memory management routines are part of the operating system of the computer. It is convenient to assemble the operating system routines into a virtual address space, called the *system space*, that is separate from the virtual space in which user application programs reside. The latter space is called the *user space*. In fact, there may be a number of user spaces, one for each user. This is arranged by providing a separate page table for each user program. The MMU uses a page table base register to determine the address of the table