CHAPTER

5

THE MEMORY SYSTEM

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Basic memory circuits
- Organization of the main memory
- Cache memory concept, which shortens the effective memory access time
- Virtual memory mechanism, which increases the apparent size of the main memory
- Magnetic disks, optical disks, and magnetic tapes 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 a large memory to facilitate execution of programs that are large and deal with huge 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. To solve this problem, much work has gone into developing clever structures that improve the apparent speed and size of the memory, yet keep the cost reasonable.

First, we describe the most common components and organizations used to implement the memory. Then we examine memory speed and discuss how the apparent speed of the memory can be increased by means of caches. Next, we present the virtual memory concept, which increases the apparent size of the memory. Finally, we discuss the secondary storage devices, which provide much larger storage capability.

5.1 SOME BASIC CONCEPTS

The maximum size of the memory that can be used in any computer is determined by the addressing scheme. For example, a 16-bit computer that generates 16-bit addresses is capable of addressing up to $2^{16} = 64K$ memory locations. Similarly, machines whose instructions generate 32-bit addresses can utilize a memory that contains up to $2^{32} = 4G$ (giga) memory locations, whereas machines with 40-bit addresses can access up to $2^{40} = 1T$ (tera) locations. The number of locations represents the size of the address space of the computer.

Most modern computers are byte addressable. Figure 2.7 shows the possible address assignments for a byte-addressable 32-bit computer. The big-endian arrangement is used in the 68000 processor. The little-endian arrangement is used in Intel processors. The ARM architecture can be configured to use either arrangement. As far as the memory structure is concerned, there is no substantial difference between the two schemes.

The memory is usually designed to store and retrieve data in word-length quantities. In fact, the number of bits actually stored or retrieved in one memory access is the most common definition of the word length of a computer. 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. In a Read operation, other bytes may be fetched from the memory, but they are ignored by the processor. If the byte operation is a Write, however, the control circuitry of the memory must ensure that the contents of other bytes of the same word are not changed.

Modern implementations of computer memory are rather complex and difficult to understand on first encounter. To simplify our introduction to memory structures, we

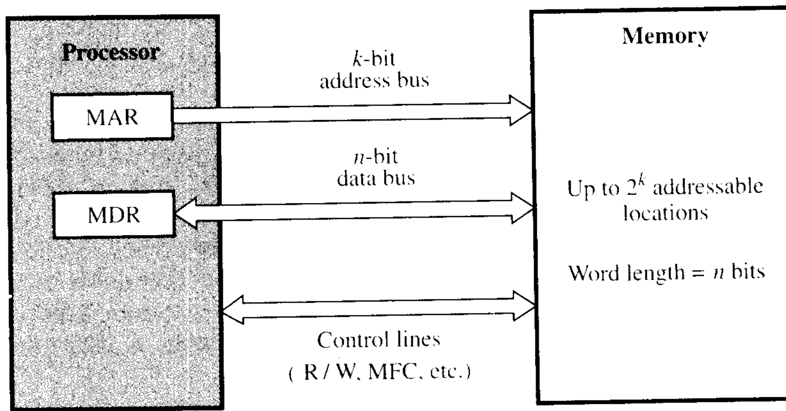


Figure 5.1 Connection of the memory to the processor.

will first present a traditional architecture. Then, in later sections, we will discuss the latest approaches.

From the system standpoint, we can view the memory unit as a black box. Data transfer between the memory and the processor takes place through the use of two processor registers, usually called MAR (memory address register) and MDR (memory data register), as introduced in Section 1.2. If MAR is k bits long and MDR is n bits long, then the memory unit may contain up to 2^k addressable locations. During a memory cycle, n bits of data are transferred between the memory and the processor. This transfer takes place over the processor bus, which has k address lines and n data lines. The bus also includes the control lines Read/$\overline{\text{Write}}$ (R/$\overline{\text{W}}$) and Memory Function Completed (MFC) for coordinating data transfers. Other control lines may be added to indicate the number of bytes to be transferred. The connection between the processor and the memory is shown schematically in Figure 5.1.

The processor reads data from the memory by loading the address of the required memory location into the MAR register and setting the R/$\overline{\text{W}}$ line to 1. The memory responds by placing the data from the addressed location onto the data lines, and confirms this action by asserting the MFC signal. Upon receipt of the MFC signal, the processor loads the data on the data lines into the MDR register.

The processor writes data into a memory location by loading the address of this location into MAR and loading the data into MDR. It indicates that a write operation is involved by setting the R/$\overline{\text{W}}$ line to 0.

If read or write operations involve consecutive address locations in the main memory, then a "block transfer" operation can be performed in which the only address sent to the memory is the one that identifies the first location. We will encounter a need for such block transfers in Section 5.5.

Memory accesses may be synchronized using a clock, or they may be controlled using special signals that control transfers on the bus, using the bus signaling schemes described in Section 4.5.1. Memory read and write operations are controlled as input and output bus transfers, respectively.

A useful measure of the speed of memory units is the time that elapses between the initiation of an operation and the completion of that operation, for example, the time between the Read and the MFC signals. 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 *random-access memory* (RAM) if any location can be accessed for a Read or Write operation in some fixed amount of time that is independent of the location's address. This distinguishes such memory units from serial, or partly serial, access storage devices such as magnetic disks and tapes. Access time on the latter devices depends on the address or position of the data.

The basic technology for implementing the memory 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.

The processor of a computer can usually process instructions and data faster than they can be fetched from a reasonably priced memory unit. The memory cycle time, then, 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 that is inserted between the larger, slower main memory and the processor. It holds the currently active segments of a program and their data.

Virtual memory is another important concept related to memory organization. So far, we have assumed that the addresses generated by the processor directly specify physical locations in the memory. This may not always be the case. For reasons that will become apparent later in this chapter, data may be stored in physical memory locations that have addresses different from those specified by the program. The memory control circuitry translates the address specified by the program into an address that can be used to access the physical memory. In such a case, an address generated by the processor is referred to as a *virtual* or *logical address.* The virtual address space is mapped onto the physical memory where data are actually stored. The mapping function is implemented by a special memory control circuit, often called the *memory management unit.* This mapping function can be changed during program execution according to system requirements.

Virtual memory is used to increase the apparent size of the physical memory. Data are addressed in a virtual address space that can be as large as the addressing capability of the processor. But at any given time, only the active portion of this space is mapped onto locations in the physical memory. The remaining virtual addresses are mapped onto the bulk storage devices used, which are usually magnetic disks. As the active portion of the virtual address space changes during program execution, the memory management unit changes the mapping function and transfers data between the disk and the memory. Thus, during every memory cycle, an address-processing mechanism determines whether the addressed information is in the physical memory unit. If it is, then the proper word is accessed and execution proceeds. If it is not, a *page* of words containing the desired word is transferred from the disk to the memory, as explained in Section 5.7.1. This page displaces some page in the memory that is currently inactive.

Because of the time required to move pages between the disk and the memory, there is a speed degradation if pages are moved frequently. By judiciously choosing which page to replace in the memory, however, there may be reasonably long periods when the probability is high that the words accessed by the processor are in the physical memory unit.

This section has briefly introduced several organizational features of memory systems. These features have been developed to help provide a computer system with as large and as fast a memory as can be afforded in relation to the overall cost of the system. We do not expect the reader to grasp all the ideas or their implications now; more detail is given later. We introduce these terms together to establish that they are related; a study of their interrelationships is as important as a detailed study of their individual features.

5.2 SEMICONDUCTOR RAM MEMORIES

Semiconductor memories are available in a wide range of speeds. Their cycle times range from 100 ns to less than 10 ns. When first introduced in the late 1960s, they were much more expensive than the magnetic-core memories they replaced. Because of rapid advances in VLSI (Very Large Scale Integration) technology, the cost of semiconductor memories has dropped dramatically. As a result, they are now used almost exclusively in implementing memories. In this section, we discuss the main characteristics of semiconductor memories. We start by introducing the way that a number of memory cells are organized inside a chip.

5.2.1 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 5.2. 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*. 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 transmit this information to the output data lines. During a Write operation, the Sense/Write circuits receive input information and store it in the cells of the selected word.

Figure 5.2 is an example of a very small memory chip consisting of 16 words of 8 bits each. This is referred to as a 16×8 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 bus of a computer. Two control lines, $R/\overline{W}$ and CS, are provided in addition to address and data lines. 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. This will be discussed in Section 5.2.4.

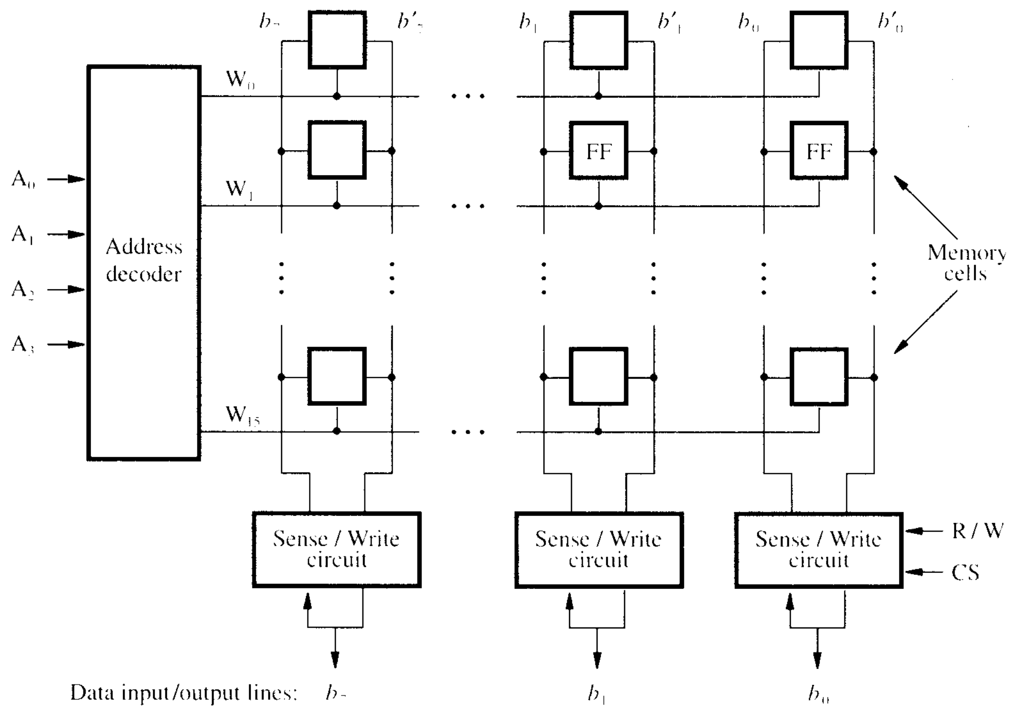


Figure 5.2 Organization of bit cells in a memory chip.

The memory circuit in Figure 5.2 stores 128 bits and requires 14 external connections for address, data, and control lines. Of course, 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 × 8 memory, requiring a total of 19 external connections. Alternatively, the same number of cells can be organized into a 1K × 1 format. In this case, a 10-bit address is needed, but there is only one data line, resulting in 15 external connections. Figure 5.3 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. However, according to the column address, only one of these cells is connected to the external data line by the output multiplexer and input demultiplexer.

Commercially available memory chips contain a much larger number of memory cells than the examples shown in Figures 5.2 and 5.3. We use small examples to make the figures easy to understand. Large chips have essentially the same organization as Figure 5.3 but use a larger memory cell array and have more external connections. For example, a 4M-bit chip may have a 512K × 8 organization, in which case 19 address and 8 data input/output pins are needed. Chips with a capacity of hundreds of megabits are now available.

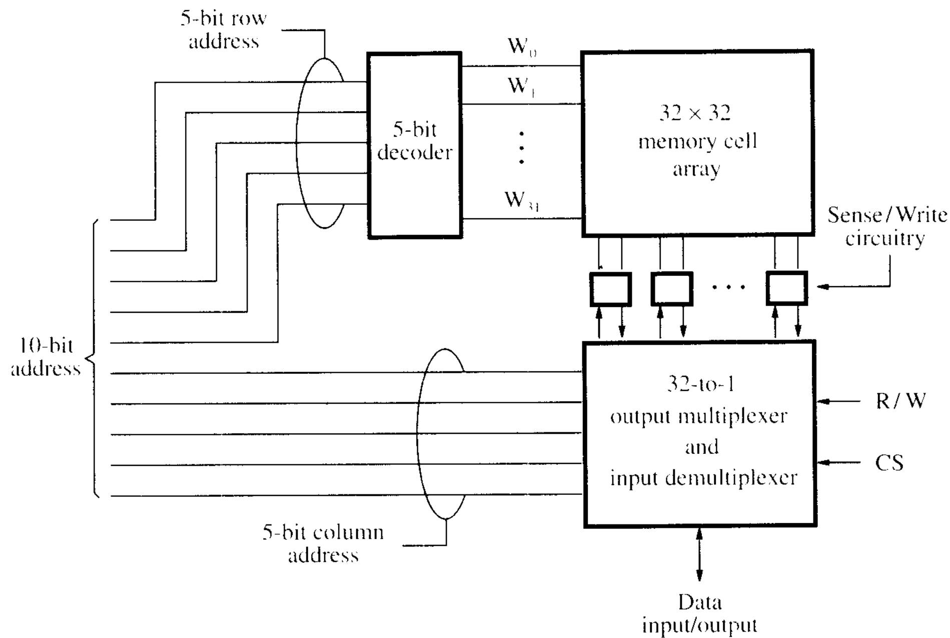


Figure 5.3 Organization of a 1K × 1 memory chip.

5.2.2 STATIC MEMORIES

Memories that consist of circuits capable of retaining their state as long as power is applied are known as *static memories*. Figure 5.4 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 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, let us assume that the cell is in state 1 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.

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 complements of each other. Sense/Write circuits at the end of the bit lines monitor the state of b and b' and set the output accordingly.

Write Operation

The state of the cell is set by placing the appropriate value on bit line b and its complement on b', and then activating the word line. This forces the cell into the corresponding state. The required signals on the bit lines are generated by the Sense/Write circuit.

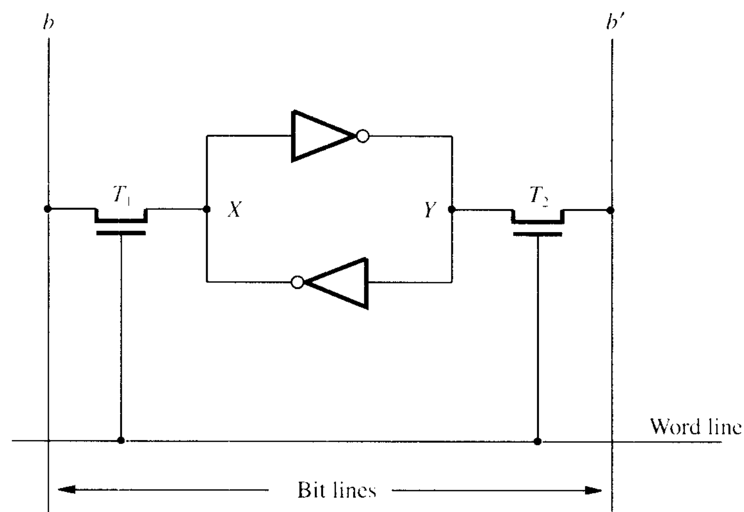


Figure 5.4 A static RAM cell.

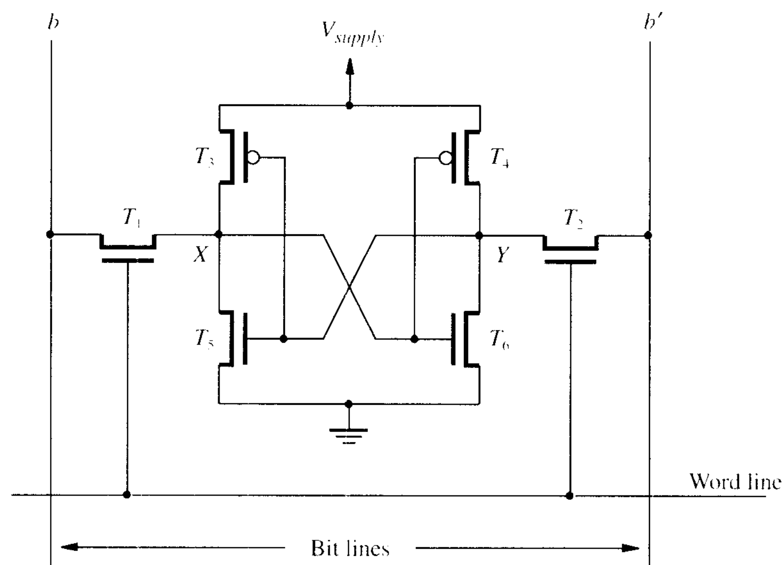


Figure 5.5 An example of a CMOS memory cell.

CMOS Cell

A CMOS realization of the cell in Figure 5.4 is given in Figure 5.5. 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. Thus,

if T_1 and T_2 are turned on (closed), bit lines b and b' will have high and low signals, respectively.

The power supply voltage, V_{supply}, is 5 V in older CMOS SRAMs or 3.3 V in new low-voltage versions. Note that continuous power is needed for the cell to retain its state. If power is interrupted, the cell's contents will be lost. When power is restored, the latch will settle into a stable state, but it will not necessarily be 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 active path between V_{supply} and ground.

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

5.2.3 ASYNCHRONOUS DRAMS

Static RAMs are fast, but they come at a high cost because their cells require several transistors. Less expensive RAMs can be implemented if simpler cells are used. However, such cells do not retain their state indefinitely; hence, they are called *dynamic RAMs* (DRAMs).

Information is stored in a dynamic memory cell in the form of a charge on a capacitor, and 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.

An example of a dynamic memory cell that consists of a capacitor, C, and a transistor, T, is shown in Figure 5.6. In order to store information in this cell, transistor

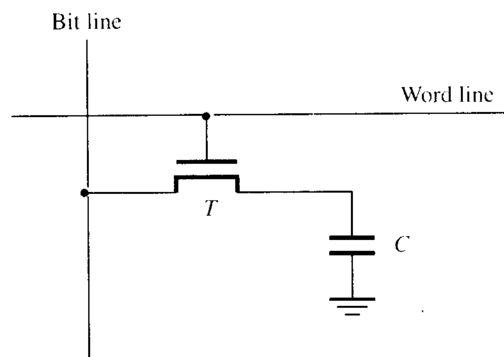


Figure 5.6 A single-transistor dynamic memory cell.

:d on and an appropriate voltage is applied to the bit line. This causes a known ıf charge to be stored in the capacitor.

· the transistor is turned off, the capacitor begins to discharge. This is caused ıacitor's own leakage resistance and by the fact that the transistor continues to tiny amount of current, measured in picoamperes, after it is turned off. Hence, ıation stored in the cell can be retrieved correctly only if it is read before the 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 on the capacitor is above the threshold value. If so, it drives the bit line to a full voltage that represents logic value 1. This voltage recharges the capacitor to the full charge that corresponds to logic value 1. If the sense amplifier detects that the charge on the capacitor is below the threshold value, it pulls the bit line to ground level, which ensures that the capacitor will have no charge, representing logic value 0. Thus, reading the contents of the cell automatically refreshes its contents. All cells in a selected row are read at the same time, which refreshes the contents of the entire row. The detailed implementation of the sense amplifier circuit is beyond the scope of this book.

A 16-megabit DRAM chip, configured as 2M × 8, is shown in Figure 5.7. The cells are organized in the form of a 4K × 4K array. The 4096 cells in each row are divided into 512 groups of 8, so that a row can store 512 bytes of data. Therefore, 12 address bits are needed to select a row. Another 9 bits are needed to specify a group of 8 bits in the selected row. Thus, a 21-bit address is needed to access a byte in this memory. The high-order 12 bits and the low-order 9 bits of the address constitute

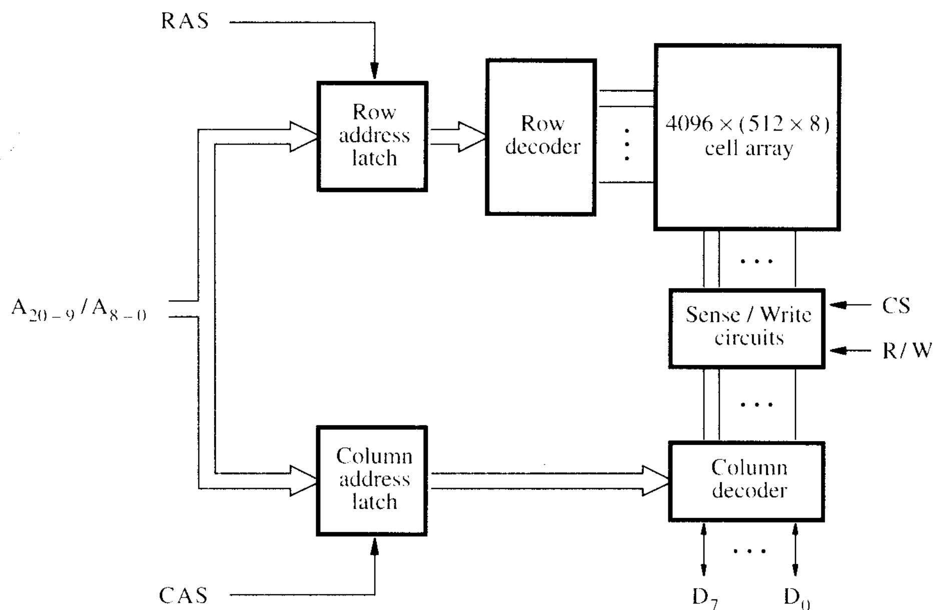


Figure 5.7 Internal organization of a 2M × 8 dynamic memory chip.

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 12 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 the Row Address Strobe (RAS) input of the chip. Then a Read operation is initiated, in which all cells on 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 the Column Address Strobe (CAS) signal. The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits are 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. This information is 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 low so that they cause the latching of addresses when they change from high to low. To indicate this fact, these signals are shown on diagrams as $\overline{RAS}$ and $\overline{CAS}$.

Applying a row address causes all cells on the corresponding row to be read and refreshed during both Read and Write operations. To ensure that the contents of a DRAM are maintained, each row of cells must be accessed periodically. A *refresh circuit* usually performs this function automatically. Many dynamic memory chips incorporate a refresh facility within the chips themselves. In this case, the dynamic nature of these memory chips is almost invisible to the user.

In the DRAM described in this section, the timing of the memory device is controlled asynchronously. A specialized memory controller circuit provides the necessary control signals, RAS and CAS, that govern the timing. The processor must take into account the delay in the response of the memory. Such memories are referred to as *asynchronous DRAMs*.

Because of their high density and low cost, DRAMs are widely used in the memory units of computers. Available chips range in size from 1M to 256M bits, and even larger chips are being developed. To reduce the number of memory chips needed in a given computer, a DRAM chip is organized to read or write a number of bits in parallel, as indicated in Figure 5.7. To provide flexibility in designing memory systems, these chips are manufactured in different organizations. For example, a 64-Mbit chip may be organized as 16M × 4, 8M × 8, or 4M × 16.

Fast Page Mode

When the DRAM in Figure 5.7 is accessed, the contents of all 4096 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_{8-0}. A simple modification can make it possible to access the other bytes in the same row without having to reselect the row. A latch can be added at the output of the sense amplifier in each column. The application of a row address will load the latches corresponding to all bits in the selected row. Then, it is only necessary to apply different column addresses to place the different bytes on the data lines.

The most useful arrangement is to transfer the bytes in sequential order, which is achieved by applying a consecutive sequence of column addresses under the control

ıccessive CAS signals. This scheme allows transferring a block of data at a much er rate than can be achieved for transfers involving random addresses. The block ısfer capability is referred to as the *fast page mode* feature. (Popular jargon refers small groups of bytes as blocks, and larger groups as pages.)

The faster rate attainable in block transfers can be exploited in applications in which ıemory accesses follow regular patterns, such as in graphics terminals. This feature is also beneficial in general-purpose computers for transferring data blocks between the main memory and a cache, as we will explain in Section 5.5.

5.2.4 SYNCHRONOUS DRAMs

More recent developments in memory technology have resulted in DRAMs whose operation is directly synchronized with a clock signal. Such memories are known as *synchronous DRAMs* (SDRAMs). Figure 5.8 indicates the structure of an SDRAM. The cell array is the same as in asynchronous DRAMs. The address and data connections are buffered by means of registers. We should particularly note that the output of each

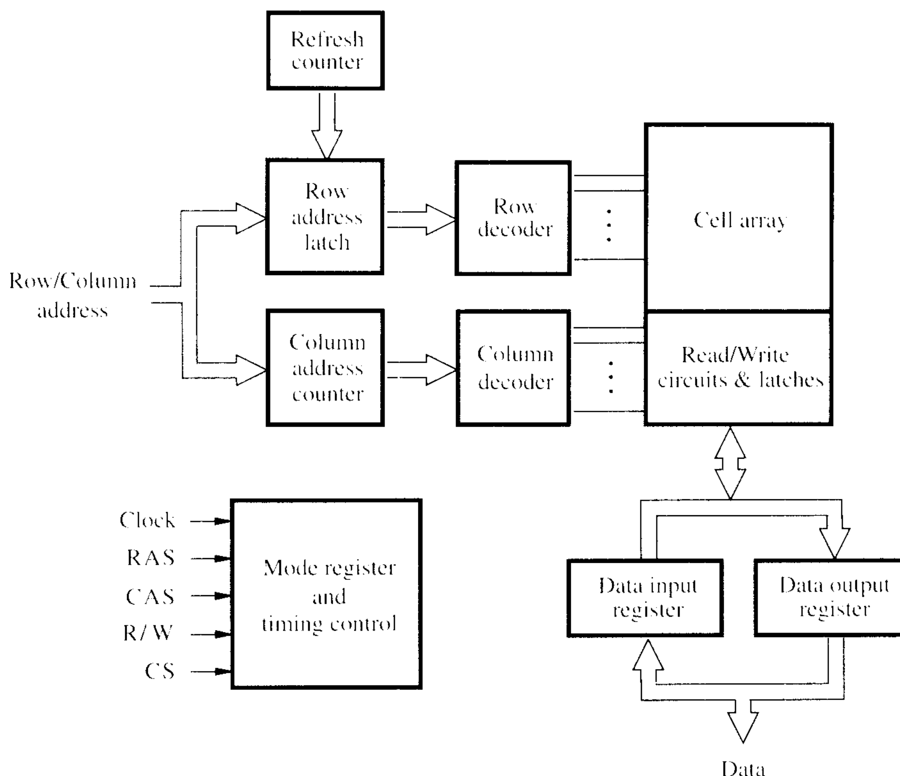


Figure 5.8 Synchronous DRAM.

sense amplifier is connected to a latch. A Read operation causes the contents of all cells in the selected row to be loaded into these latches. But, if an access is made for refreshing purposes only, it will not change the contents of these latches; it will merely refresh the contents of the cells. Data held in the latches that correspond to the selected column(s) are transferred into the data output register, thus becoming available on the data output pins.

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. The burst operations use the block transfer capability described above as the fast page mode feature. In SDRAMs, it is not necessary to provide externally generated pulses on the CAS line to select successive columns. The necessary control signals are provided internally using a column counter and the clock signal. New data can be placed on the data lines in each clock cycle. All actions are triggered by the rising edge of the clock.

Figure 5.9 shows a timing diagram for a typical burst read of length 4. First, the row address is latched under control of the $\overline{\text{RAS}}$ signal. The memory typically takes 2 or 3 clock cycles (we use 2 in the figure) to activate the selected row. Then, the column address is latched under control of the $\overline{\text{CAS}}$ 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.

SDRAMs have built-in refresh circuitry. A part of this circuitry is a refresh counter, which provides the addresses of the rows that are selected for refreshing. In a typical SDRAM, each row must be refreshed at least every 64 ms.

Commercial SDRAMs can be used with clock speeds above 100 MHz. These chips are designed to meet the requirements of commercially available processors that are used

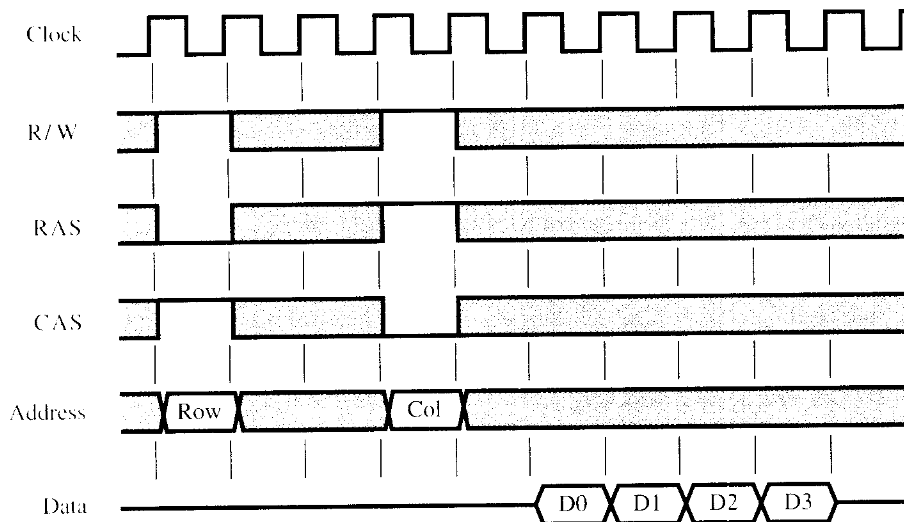


Figure 5.9 Burst read of length 4 in an SDRAM.

in large volume. For example, Intel has defined PC100 and PC133 bus specifications in which the system bus (to which the main memory is connected) is controlled by a 100 or 133 MHz clock, respectively. Therefore, major manufacturers of memory chips produce 100 and 133 MHz SDRAM chips.

Latency and Bandwidth

Transfers between the memory and the processor involve single words of data or small blocks of words (to or from the processor caches which are discussed in Section 5.5). Large blocks, constituting a page of data, are transferred between the memory and the disks, as described in Section 5.7. The speed and efficiency of these transfers have a large impact on the performance of a computer system. A good indication of the performance is given by two parameters: latency and bandwidth.

The term *memory latency* is used to refer to the amount of time it takes to transfer a word of data to or from the memory. In the case of reading or writing a single word of data, the latency provides a complete indication of memory performance. But, in the case of burst operations that transfer a block of data, the time needed to complete the operation depends also on the rate at which successive words can be transferred and on the size of the block. In block transfers, the term latency is used to denote the time it takes to transfer the first word of data. This time is usually substantially longer than the time needed to transfer each subsequent word of a block. For instance, in the timing diagram in Figure 5.9, the access cycle begins with the assertion of the $\overline{RAS}$ signal. The first word of data is transferred five clock cycles later. Thus, the latency is five clock cycles. If the clock rate is 100 MHz, then the latency is 50 ns. The remaining three words are transferred in consecutive clock cycles.

When transferring blocks of data, it is of interest to know how much time is needed to transfer an entire block. Since blocks can be variable in size, it is useful to define a performance measure in terms of the number of bits or bytes that can be transferred in one second. This measure is often referred to as the memory *bandwidth*. The bandwidth of a memory unit (consisting of one or more memory chips) depends on the speed of access to the stored data and on the number of bits that can be accessed in parallel. However, the effective bandwidth in a computer system (involving data transfers between the memory and the processor) is not determined solely by the speed of the memory; it also depends on the transfer capability of the links that connect the memory and the processor, typically the speed of the bus. Memory chips are usually designed to meet the speed requirements of popular buses. The bandwidth clearly depends on the speed of access and transmission along a single wire, as well as on the number of bits that can be transferred in parallel, namely the number of wires. Thus, the bandwidth is the product of the rate at which data are transferred (and accessed) and the width of the data bus.

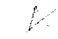
Double-Data-Rate SDRAM

In the continuous quest for improved performance, a faster version of SDRAM has been developed. The standard SDRAM performs all actions on the rising edge of the clock signal. A similar memory device is available, which accesses the cell array in the same way, but transfers data on both edges of the clock. The latency of these devices is the same as for standard SDRAMs. But, since they transfer data on both edges of the

clock, their bandwidth is essentially doubled for long burst transfers. Such devices are known as *double-data-rate SDRAMs* (DDR SDRAMs).

To make it possible to access the data at a high enough rate, the cell array is organized in two banks. Each bank can be accessed separately. Consecutive words of a given block are stored in different banks. Such *interleaving* of words allows simultaneous access to two words that are transferred on successive edges of the clock. We will consider the concept of interleaving in more detail in Section 5.6.1.

DDR SDRAMs and standard SDRAMs are most efficiently used in applications where block transfers are prevalent. This is the case in general-purpose computers in which main memory transfers are primarily to and from processor caches, as we will see in Section 5.5. Block transfers are also done in high-quality video displays.



5.2.5 STRUCTURE OF LARGER MEMORIES

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

Static Memory Systems

Consider a memory consisting of 2M (2,097,152) words of 32 bits each. Figure 5.10 shows how we can implement this memory using 512K × 8 static memory chips. Each column in the figure consists of four chips, which implement one byte position. Four of these sets provide the required 2M × 32 memory. Each chip has a control input called 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 three-state type (see Section A.5.4). Only the selected chip places data on the data output line, while all other outputs are in the high-impedance state. Twenty one address bits are needed to select a 32-bit word in this memory. The high-order 2 bits of the address are decoded to determine which of the four Chip Select control signals should be activated, and the remaining 19 address bits are used to access specific byte locations inside each chip of the selected row. The $R/\overline{W}$ inputs of all chips are tied together to provide a common $Read/\overline{Write}$ control (not shown in the figure).

Dynamic Memory Systems

The organization of large dynamic memory systems is essentially the same as the memory shown in Figure 5.10. However, physical implementation is often done more conveniently in the form of *memory modules*.

Modern computers use very large memories; even a small personal computer is likely to have at least 32M bytes of memory. Typical workstations have at least 128M bytes 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 accessing the information in secondary storage. However, if a large memory is built by placing DRAM chips directly on the main system printed-circuit board that contains the processor, often referred to as a *motherboard*, it will occupy an unacceptably large amount of space on the board. Also, it is awkward to provide for future

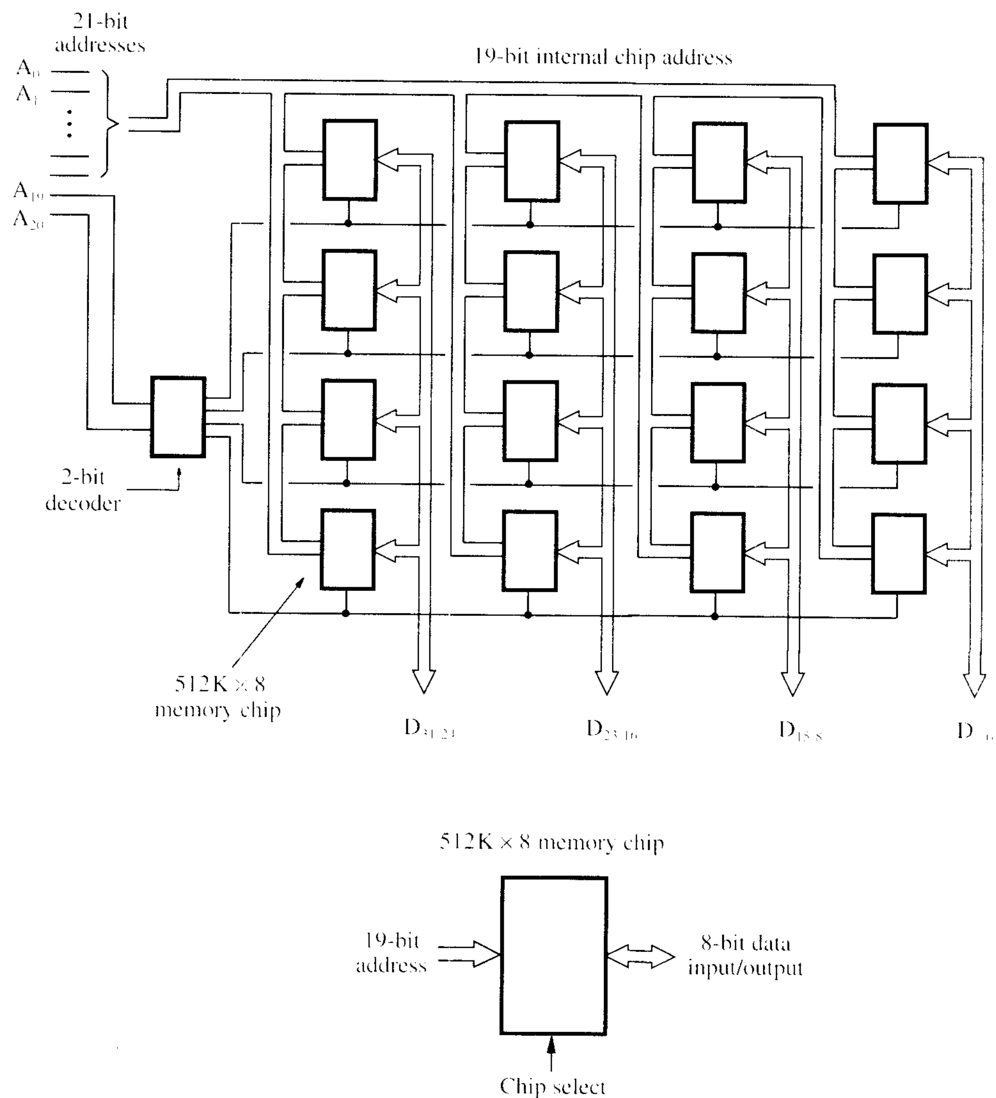


Figure 5.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

expansion of the memory, because space must be allocated and wiring provided for the maximum expected size. These packaging considerations have led to the development of larger memory units known as *SIMMs* (Single In-line Memory Modules) and *DIMMs* (Dual In-line Memory Modules). Such a module is an assembly of several memory chips on a separate small board that plugs vertically into a single socket on the motherboard. SIMMs and DIMMs of different sizes are designed to use the same size socket. For example, 4M × 32, 16M × 32, and 32M × 32 bit DIMMs all use the same

100-pin socket. Similarly, 8M × 64, 16M × 64, 32M × 64, and 64M × 72 DIMMs use a 168-pin socket. Such modules occupy a smaller amount of space on a motherboard, and they allow easy expansion by replacement if a larger module uses the same socket as the smaller one.

5.2.6 MEMORY SYSTEM CONSIDERATIONS

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

Static RAMs are generally used only when very fast operation is the primary requirement. Their cost and size are adversely affected by the complexity of the circuit that realizes the basic cell. They are used mostly in cache memories. Dynamic RAMs are the predominant choice for implementing computer main memories. The high densities achievable in these chips make large memories economically feasible.

Memory Controller

To reduce the number of pins, the dynamic memory chips use multiplexed address inputs. The address is divided into two parts. 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 using the CAS signal.

A typical processor issues all bits of an address at the same time. The required multiplexing of address bits is usually performed by a *memory controller* circuit, which is interposed between the processor and the dynamic memory as shown in Figure 5.11. 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. The controller then forwards the row and column portions of the address to the memory and generates the $\overline{RAS}$ and $\overline{CAS}$ signals. Thus, the controller provides the RAS-CAS timing, in addition to its address multiplexing function. It also sends the R/$\overline{W}$ and CS

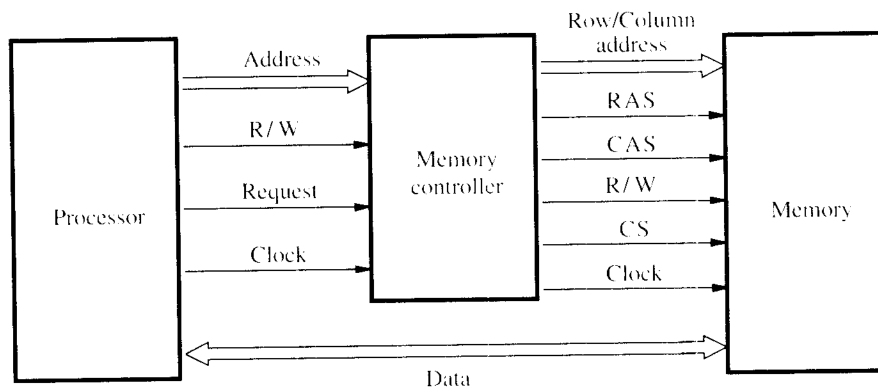


Figure 5.11 Use of a memory controller.

signals to the memory. The CS signal is usually active low, hence it is shown as $\overline{CS}$ in Figure 5.11. Data lines are connected directly between the processor and the memory. Note that the clock signal is needed in SDRAM chips.

When used with DRAM chips, which do not have self-refreshing capability, the memory controller has to provide all the information needed to control the refreshing process. It contains a refresh counter that provides successive row addresses. Its function is to cause the refreshing of all rows to be done within the period specified for a particular device.

Refresh Overhead

All dynamic memories have to be refreshed. In older DRAMs, a typical period for refreshing all rows was 16 ms. In typical SDRAMs, a typical period is 64 ms.

Consider an SDRAM whose cells are arranged in 8K (=8192) rows. Suppose that it takes four clock cycles to access (read) each row. Then, it takes $8192 \times 4 = 32,768$ cycles to refresh all rows. At a clock rate of 133 MHz, the time needed to refresh all rows is $32,768/(133 \times 10^6) = 246 \times 10^{-6}$ seconds. Thus, the refreshing process occupies 0.246 ms in each 64-ms time interval. Therefore, the refresh overhead is $0.246/64 = 0.0038$, which is less than 0.4 percent of the total time available for accessing the memory.

5.2.7 RAMBUS MEMORY

The performance of a dynamic memory is characterized by its latency and bandwidth. Since all dynamic memory chips use similar organizations for their cell arrays, their latencies tend to be similar if the chips are produced using the same manufacturing process. On the other hand, the effective bandwidth of a memory system depends not only on the structure of the memory chips, but also on the nature of the connecting path to the processor. DDR SDRAMs and standard SDRAMs are connected to the processor bus. Thus, the speed of transfers is not just a function of the speed of the memory device — it also depends on the speed of the bus. A bus clocked at 133 MHz allows at most one transfer every 7.5 ns, or two transfers if both edges of the clock are used. The only way to increase the amount of data that can be transferred on a speed-limited bus is to increase the width of the bus by providing more data lines, thus widening the bus.

A very wide bus is expensive and requires a lot of space on a motherboard. An alternative approach is to implement a narrow bus that is much faster. This approach was used by Rambus Inc. to develop a proprietary design known as *Rambus*. The key feature of Rambus technology is a fast signaling method used to transfer information between chips. Instead of using signals that have voltage levels of either 0 or V_{supply} to represent the logic values, the signals consist of much smaller voltage swings around a reference voltage, V_{ref}. The reference voltage is about 2 V, and the two logic values are represented by 0.3 V swings above and below V_{ref}. This type of signaling is generally known as *differential signaling*. Small voltage swings make it possible to have short transition times, which allows for a high speed of transmission.

Differential signaling and high transmission rates require special techniques for the design of wire connections that serve as communication links. These requirements make

it difficult to make the bus wide. It is also necessary to design special circuit interfaces to deal with the differential signals. Rambus provides a complete specification for the design of such communication links, called the *Rambus channel*. Present designs of Rambus allow for a clock frequency of 400 MHz. Moreover, data are transmitted on both edges of the clock, so that the effective data transfer rate is 800 MHz.

Rambus requires specially designed memory chips. These chips use cell arrays based on the standard DRAM technology. Multiple banks of cell arrays are used to access more than one word at a time. Circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as *Rambus DRAMs* (RDRAMs).

The original specification of Rambus provided for a channel consisting of 9 data lines and a number of control and power supply lines. Eight of the data lines are intended for transferring a byte of data. The ninth data line can be used for purposes such as parity checking. Subsequent specifications allow for additional channels. A two-channel Rambus, also known as *Direct RDRAM*, has 18 data lines intended to transfer two bytes of data at a time. There are no separate address lines.

Communication between the processor, or some other device that can serve as a *master*, and RDRAM modules, which serve as *slaves*, is carried out by means of *packets* transmitted on the data lines. There are three types of packets: request, acknowledge, and data. A request packet issued by the master indicates the type of operation that is to be performed. It contains the address of the desired memory location and includes an 8-bit count that specifies the number of bytes involved in the transfer. The operation types include memory reads and writes, as well as reading and writing of various control registers in the RDRAM chips. When the master issues a request packet, the addressed slave responds by returning a positive acknowledgement packet if it can immediately satisfy the request. Otherwise, the slave indicates that it is "busy" by returning a negative acknowledgement packet, in which case the master will try again.

The number of bits in a request packet exceeds the number of data lines, which means that several clock cycles are needed to transmit the entire packet. Using a narrow communication link is compensated by the very high rate of transmission.

RDRAM chips can be assembled into larger modules, similar to SIMMs and DIMMs. One such module, called RIMM, can hold up to 16 RDRAMs.

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, while RDRAM is a proprietary design of Rambus Inc. for which the manufacturers of chips have to pay a royalty. Finally, we should note that in the memory market, assuming that the performance is adequate, the decisive factor is often the price of components.

5.3 Read-Only Memories

Both SRAM and DRAM chips are volatile, which means that they lose the stored information if power is turned off. There are many applications that need memory devices which retain the stored information if power is turned off. For example, in a typical computer a hard disk drive is used to store a large amount of information,

including the operating system software. When a computer is turned on, the operating system software has to be loaded from the disk into the memory. This requires execution of a program that "boots" the operating system. Since the boot program is quite large, most of it is stored on the disk. The processor must execute some instructions that load the boot program into the memory. If the entire memory consisted of only volatile memory chips, the processor would have no means of accessing these instructions. A practical solution is to provide a small amount of nonvolatile memory that holds the instructions whose execution results in loading the boot program from the disk.

Nonvolatile memory is used extensively in embedded systems, which are presented in Chapter 9. Such systems typically do not use disk storage devices. Their programs are stored in nonvolatile semiconductor memory devices.

Different types of nonvolatile memory have been developed. Generally, the contents of such memory can be read as if they were SRAM or DRAM memories. But, a special writing process is needed to place the information into this memory. Since its normal operation involves only reading of stored data, a memory of this type is called *read-only memory* (ROM).

5.3.1 ROM

Figure 5.12 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. Thus, the transistor switch is closed and 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, indicating a 1. A sense circuit at the end of the bit line generates the proper output value. Data are written into a ROM when it is manufactured.

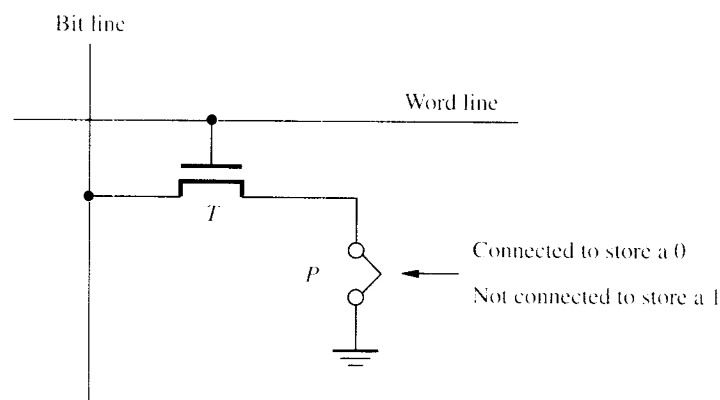


Figure 5.12 A ROM cell.

5.3.2 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 5.12. 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 latter are economically attractive for storing fixed programs and data when high volumes of ROMs are produced. However, the cost of preparing the masks needed for storing a particular information pattern in ROMs makes them very expensive when only a small number are required. In this case, PROMs provide a faster and considerably less expensive approach because they can be programmed directly by the user.

5.3.3 EPROM

Another type of ROM chip allows the stored data to be erased and new data to be loaded. 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 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 5.12. In an EPROM cell, however, the connection to ground is always made at point P and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off. This transistor can be programmed to behave as a permanently open switch, 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.

The important advantage of EPROM chips is that their contents can be erased and reprogrammed. Erasure requires dissipating the charges trapped in the transistors of memory cells; this can be done by exposing the chip to ultraviolet light. For this reason, EPROM chips are mounted in packages that have transparent windows.

5.3.4 EEPROM

A significant disadvantage of EPROMs is that a chip must be physically removed from the circuit for reprogramming and that its entire contents are erased by the ultraviolet light. It is possible to implement another version of erasable PROMs that can be both programmed and erased electrically. Such chips, called EEPROMs, do not have to be removed for erasure. Moreover, it is possible to erase the cell contents selectively. The only disadvantage of EEPROMs is that different voltages are needed for erasing, writing, and reading the stored data.

5.3.5 FLASH MEMORY

An approach similar to EEPROM technology has more recently given rise to *flash memory* devices. A flash cell is based on a single transistor controlled by trapped charge, just like an EEPROM cell. While similar in some respects, there are also substantial differences between flash and EEPROM devices. In EEPROM it is possible to read and write the contents of a single cell. In a flash device it is possible to read the contents of a single cell, but 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 memory makes it attractive for use in portable equipment that is battery driven. Typical applications include hand-held computers, cell phones, digital cameras, and MP3 music players. In hand-held computers and cell phones, flash memory holds the software needed to operate the equipment, thus obviating the need for a disk drive. In digital cameras, flash memory is used to store picture image data. In MP3 players, flash memory stores the data that represent sound. Cell phones, digital cameras, and MP3 players are good examples of embedded systems, which will be discussed in detail in Chapter 9.

Single flash chips do not provide sufficient storage capacity for the applications mentioned above. Larger memory modules consisting of a number of chips are 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 come in a variety of memory sizes. Typical sizes are 8, 32, and 64 Mbytes. A minute of music can be stored in about 1 Mbyte of memory, using the MP3 encoding format. Hence, a 64-MB flash card can store an hour of music.

Flash Drives

Larger flash memory modules have been developed to replace hard disk drives. These flash drives are designed to fully emulate the 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 less than one gigabyte. In contrast, hard disks can store many gigabytes.

The fact that flash drives are solid state electronic devices that have no movable parts provides some important advantages. They have shorter seek and access times, which results in faster response. (Seek and access times are discussed in the context of disks in Section 5.9.) They have lower power consumption, which makes them attractive for battery driven applications, and they are also insensitive to vibration.

The disadvantages of flash drives vis-a-vis hard disk drives are their smaller capacity and higher cost per bit. Disks provide an extremely low cost per bit. Another

disadvantage is that the flash memory will deteriorate after it has been written a number of times. Fortunately, this number is high, typically at least one million times.

5.4 Speed, Size, and Cost

We have already stated that an ideal memory would be fast, large, and inexpensive. From the discussion in Section 5.2, it is clear that a very fast memory can be implemented if SRAM chips are used. But these chips are expensive because their basic cells have six transistors, which precludes packing a very large number of cells onto a single chip. Thus, for cost reasons, it is impractical to build a large memory using SRAM chips. The alternative is to use Dynamic RAM chips, which have much simpler basic cells and thus are much less expensive. But such memories are significantly slower.

Although dynamic memory units in the range of hundreds of megabytes 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 implement large memory spaces. Very large disks are available at a reasonable price, and they are used extensively in computer systems. However, they are much slower than the semiconductor memory units. So we conclude the following: A huge amount of cost-effective storage can be provided by magnetic disks. A large, yet affordable, main memory can be built with dynamic RAM technology. This leaves SRAMs 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. The entire computer memory can be viewed as the hierarchy depicted in Figure 5.13. The fastest access is to data held in processor registers. Therefore, if we consider the registers to be part of the memory hierarchy, then the processor registers are at the top in terms of the 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 instructions and data stored in a much larger memory that is provided externally. The cache memory concept was introduced in Figure 1.6 and is examined in detail in Section 5.5. There are often two levels of caches. A primary cache is always located on the processor chip. This cache is small because it competes for space on the processor chip, which must implement many other functions. The primary cache is referred to as *level 1* (L1) cache. A larger, secondary cache is placed between the primary cache and the rest of the memory. It is referred to as *level 2* (L2) cache. It is usually implemented using SRAM chips.

Including a primary cache on the processor chip and using a larger, off-chip, secondary cache is currently the most common way of designing computers. However, other arrangements can be found in practice. It is possible not to have a cache on the processor chip at all. Also, it is possible to have both L1 and L2 caches on the processor chip.

The next level in the hierarchy is called the *main memory*. This rather large memory is implemented using dynamic memory components, typically in the form of SIMMs,

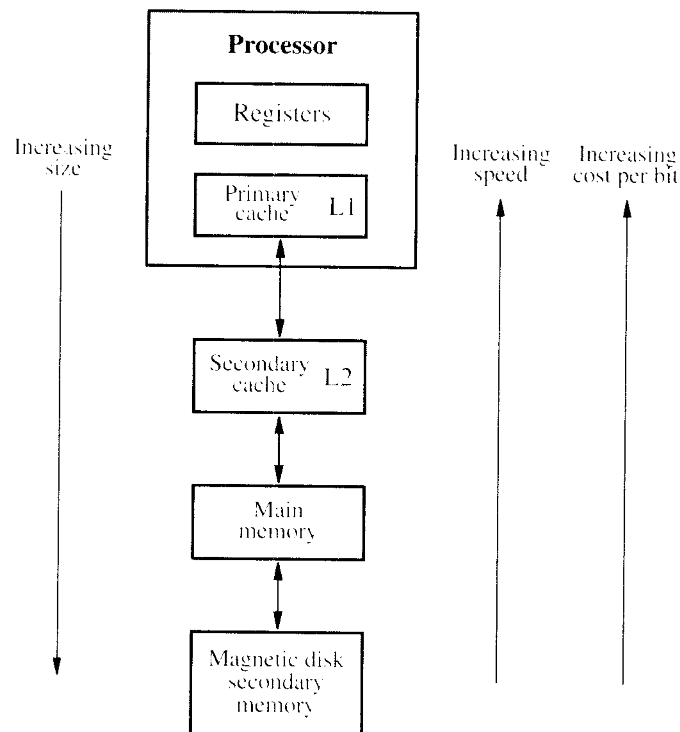


Figure 5.13 Memory hierarchy.

DIMMs, or RIMMs. The main memory is much larger but significantly slower than the cache memory. In a typical computer, the access time for the main memory is about ten times longer than the access time for the L1 cache.

Disk devices provide a huge amount of inexpensive storage. They are very slow compared to the semiconductor devices used to implement the main memory. We will discuss disk technology in Section 5.9.

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 5.13 is to bring the instructions and data that will be used in the near future as close to the processor as possible. This can be done by using the mechanisms presented in the sections that follow. We begin with a detailed discussion of cache memories.

5.5 CACHE MEMORIES

The speed of the main memory is very low in comparison with the speed of modern processors. For good performance, the processor cannot spend much of its time waiting to access instructions and data in main memory. Hence, it is important to devise a scheme

that reduces the time needed to access the necessary information. Since the speed of the main memory unit is limited by electronic and packaging constraints, the solution must be sought in a different architectural arrangement. An efficient solution is to use a fast *cache memory* which essentially makes the main memory appear to the processor to be faster than it really is.

The effectiveness of the cache mechanism is based on a property of computer programs called locality of reference. Analysis of programs shows that most of their execution time is spent on 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, and the remainder of the program is accessed relatively infrequently. This is referred to as *locality of reference*. It 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 in close proximity to a recently executed instruction (with respect to the instructions' addresses) are also likely to be executed soon.

If the active segments of a program can be placed in a fast cache memory, then the total execution time can be reduced significantly. 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. The temporal aspect of the locality of reference suggests that whenever an information item (instruction or data) is first needed, this item should be brought into the cache where it will hopefully remain until it is needed again. The spatial aspect suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that reside at adjacent addresses as well. We will use the term *block* to refer to a set of contiguous address locations of some size. Another term that is often used to refer to a cache block is *cache line*.

Consider the simple arrangement in Figure 5.14. When a Read request is received from the processor, the contents of a block of memory words containing the location specified are transferred into the cache one word at a time. 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

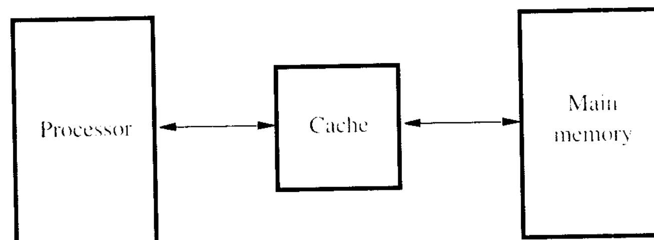


Figure 5.14 Use of a cache memory

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 *replacement algorithm*.

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. In a Read operation, the main memory is not involved. For a Write operation, the system can proceed in two ways. In the first technique, called the *write-through* protocol, the cache location and the main memory location are updated simultaneously. The second technique is to update only the cache location and to mark it as updated 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 to be 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, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency. Note that the write-back protocol may also result in unnecessary Write operations because when a cache block is written back to the memory all words of the block are written back, even if only a single word has been changed while the block was in the cache.

When the addressed word in a Read operation is not in the cache, a *read miss* occurs. The block of words that contains the requested word is 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 period somewhat, but at the expense of more complex circuitry.

During a Write operation, if the addressed word is not in the cache, a *write miss* occurs. Then, if the write-through protocol is used, the information is written directly into the main memory. In the case of 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.

5.5.1 Mapping Functions

To discuss possible methods for specifying where memory blocks are placed in the cache, we use 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 will assume 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 5.15. Thus, whenever one of the main memory blocks 0, 128, 256, ... is loaded in the cache, it is stored in cache block 0. Blocks 1, 129, 257, ... 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

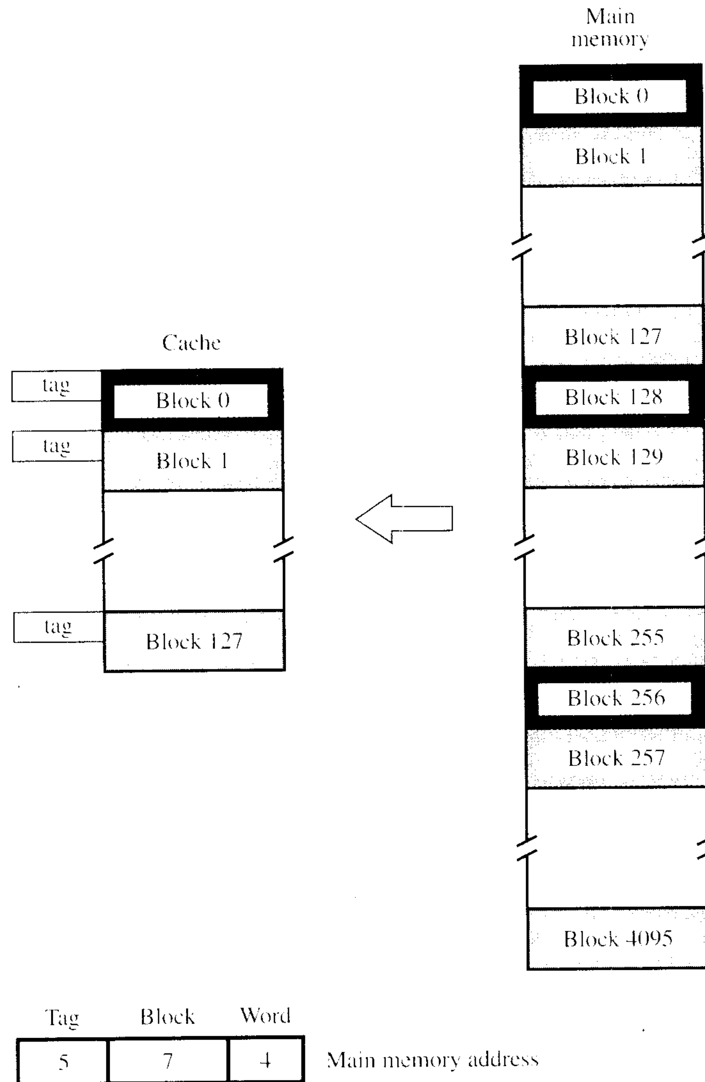


Figure 5.15 Direct-mapped cache.

program may start in block 1 and continue in block 129, possibly after a branch. As this program is executed, 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. In this case, the replacement algorithm is trivial.

Placement of a block in the cache is determined from the memory address. The memory address can be divided into three fields, as shown in Figure 5.15. 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. They identify which of the 32 blocks that are mapped into this cache position are 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 5.16 shows a much more 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 choosing the cache location in which to place the memory block. Thus, the space in the cache can be used more efficiently. A new block that has to be brought into the cache has to replace (eject) 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 5.5.2. The cost of an associative cache is higher than the cost 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. A search of this kind is called an *associative search*. For performance reasons, the tags must be searched in parallel.

Set-Associative Mapping

A combination of the direct- and associative-mapping techniques can be used. 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 5.17 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.

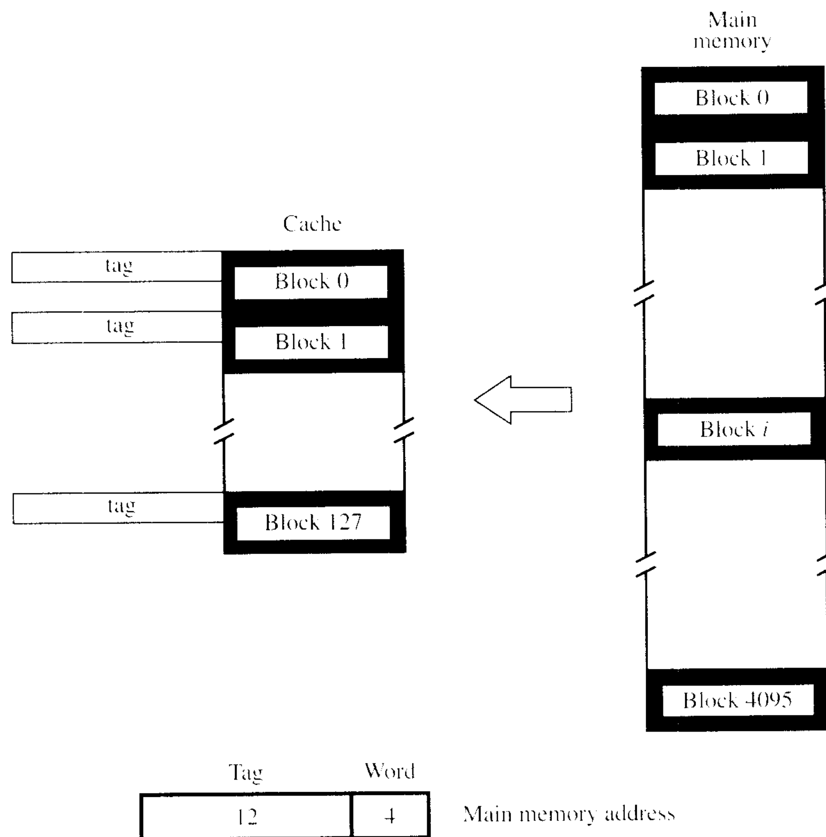


Figure 5.16 Associative-mapped cache.

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 5.17, 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.

One more control bit, called the *valid bit*, must be provided for each block. This bit indicates whether the block contains valid data. It should not be confused with the modified, or dirty, bit mentioned earlier. The dirty bit, which indicates whether the block has been modified during its cache residency, is needed only in systems that do not use the write-through method. The valid bits are all set to 0 when power is initially applied to the system or when the main memory is loaded with new programs and data from the disk. Transfers from the disk to the main memory are carried out by a DMA mechanism. Normally, they bypass the cache for both cost and performance reasons. The valid bit of a particular cache block is set to 1 the first time this block is loaded

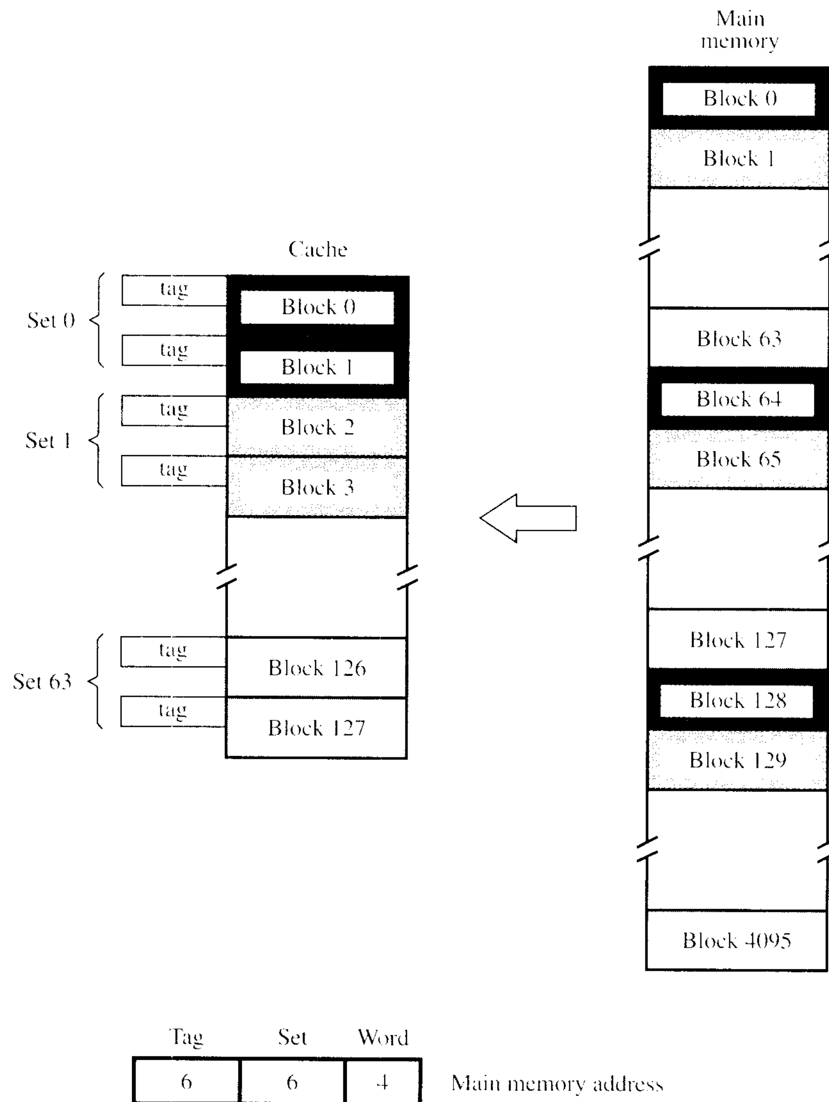


Figure 5.17 Set-associative-mapped cache with two blocks per set.

from the main memory. Whenever a main memory block is updated by a source that bypasses the cache, a check is made to determine whether the block being loaded is currently in the cache. If it is, its valid bit is cleared to 0. This ensures that *stale* data will not exist in the cache.

A similar difficulty arises when a DMA transfer is made from the main memory to the disk, and the cache uses the write-back protocol. In this case, the data in the memory might not reflect the changes that may have been made in the cached copy.

One solution to this problem is to *flush* the cache by forcing the dirty data to be written back to the memory before the DMA transfer takes place. The operating system can do this easily, and it does not affect performance greatly, because such disk transfers do not occur often. This need to ensure that two different entities (the processor and DMA subsystems in this case) use the same copies of data is referred to as a *cache-coherence* problem.

5.5.2 REPLACEMENT ALGORITHMS

In a direct-mapped cache, the position of each block is predetermined; hence, no replacement strategy exists. 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. However, 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 programs usually stay 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 5.5.3 and Problem 5.12). 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.

5.5.3 EXAMPLE 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 5.18. 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 5.15 through 5.17, 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

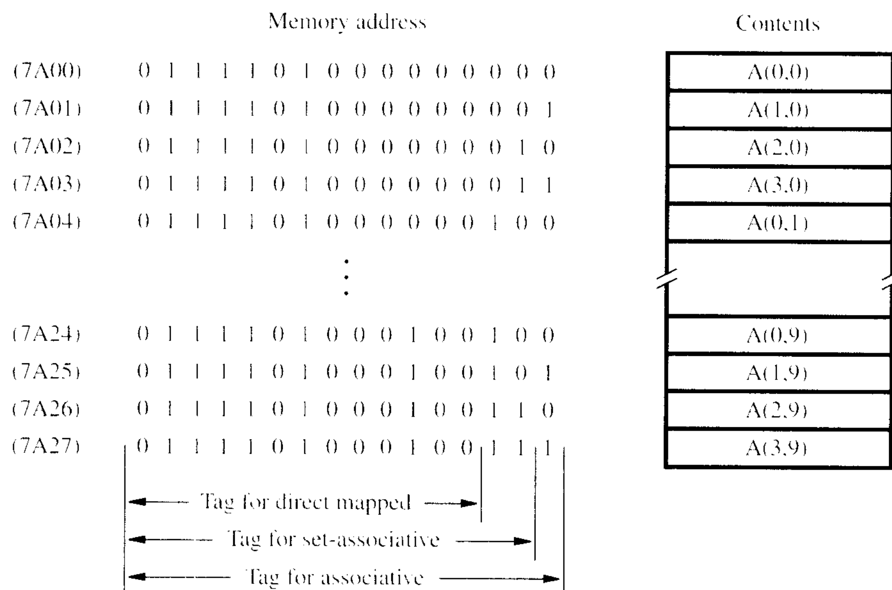


Figure 5.18 An array stored in the main memory.

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

Figure 5.19 Task for example in Section 5.5.3.

Block position	Contents of data cache after pass:								
	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
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 5.20 Contents of a direct-mapped data cache.

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 5.19 gives the structure of a program that corresponds to this task. In a machine language implementation of this program, the array elements will be addressed as memory locations. 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, will be 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 5.20. The columns in the table indicate the cache contents after various passes through the two program loops in Figure 5.19 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 was executed.

After the tenth pass through the first loop ($j = 9$), the elements $A(0, 8)$ and $A(0, 9)$ are found in the cache. Since the second loop reverses the order in which the elements are handled, the first two passes through this loop ($i = 9, 8$) will find the required data in the cache. When $i = 7$, the element $A(0, 9)$ is replaced with $A(0, 7)$. When $i = 6$, element $A(0, 8)$ is replaced with $A(0, 6)$, and so on. Thus, eight elements are replaced while the second loop is executed.

The reader should keep in mind that the tags must be kept in the cache for each block. We have not shown them in the figure for space reasons.

Associative-Mapped Cache

Figure 5.21 presents the changes if the cache is associative-mapped. 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 $A(0, 8)$. The next and last pass through the j loop sees $A(0, 1)$ replaced by $A(0, 9)$. Now, for the first eight passes through the second loop ($i = 9, 8, \ldots, 2$) all 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. Good utilization

Block position	Contents of data cache after pass:				
	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
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 5.21 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 5.22 Contents of a set-associative-mapped data cache.

of this cache also occurred because we chose to reverse 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 (see Problem 5.12). Using the LRU algorithm, all elements would be overwritten before they are used in the second loop. This degradation in performance would not occur if a random replacement algorithm were used.

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 the corresponding memory block maps into. The high-order 15 bits constitute the tag.

Changes in the cache contents are depicted in Figure 5.22. Since all the desired blocks have even addresses, they map into set 0. Note that, in this case, six elements must be 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.

5.5.4 Examples of Caches in Commercial Processors

We now consider the implementation of caches in the 68040, ARM710T, and Pentium III and 4 processors.

68040 Caches

Motorola's 68040 has two caches included on the processor chip — one used for instructions and the other for data. Each cache has a capacity of 4K bytes and uses a

four-way set-associative organization illustrated in Figure 5.23. The cache has 64 sets, each of which can hold 4 blocks. Each block has 4 long words, and each long word has 4 bytes. For mapping purposes, an address is interpreted as shown in the blue box in the figure. The least-significant 4 bits specify a byte position within a block. The next 6 bits identify one of 64 sets. The high-order 22 bits constitute the tag. To keep the notation in the figure compact, the contents of each of these fields are shown in hex coding.

The cache control mechanism includes one *valid* bit per block and one *dirty* bit for each long word in the block. These bits are explained in Section 5.5.1. The valid bit is set to 1 when a corresponding block is first loaded into the cache. An individual dirty bit is associated with each long word, and it is set to 1 when the long-word data are changed during a write operation. The dirty bit remains set until the contents of the block are written back into the main memory.

When the cache is accessed, the tag bits of the address are compared with the four tags in the specified set. If one of the tags matches the desired address and if the valid bit for the corresponding block is equal to 1, then a hit has occurred. Figure 5.23 gives an example in which the addressed data are found in the third long word of the fourth block in set 0.

The data cache can use either the write-back or the write-through protocol, under control of the operating system software. The contents of the instruction cache are changed only when new instructions are loaded as a result of a read miss. When a new block must be brought into a cache set that is already full, the replacement algorithm chooses at random the block to be ejected. Of course, if one or more of the dirty bits in this block are equal to 1, then a write-back must be performed first.

ARM710T Cache

The ARM family comprises processors that have an efficient RISC-type architecture, characterized by low cost and low power consumption. The ARM710T is one of the processors in this family. It has a single cache for both instructions and data.

The organization of the ARM710T cache is similar to the cache depicted in Figure 5.23. It is arranged as a four-way set-associative cache. Each block comprises 16 bytes, composed of four 32-bit words.

The write-through protocol is used when the processor writes to the cache. A random-replacement algorithm is used to decide which cache block is to be overwritten when space for a new block is needed.

The ARM710T cache structure is consistent with the low cost and low power consumption objective. A single unified cache, holding both instructions and data, is simpler than two separate caches. The write-through protocol and the random-replacement algorithm are also conducive to simple implementations.

Pentium III Caches

Pentium III is a high performance processor. Since high performance depends on fast access to instructions and data, Pentium III employs two cache levels. Level 1 consists of a 16-Kbyte instruction cache and a 16-Kbyte data cache. The data cache has a four-way set-associative organization, and it can use either the write-back or the write-through policy. The instruction cache has a two-way set-associative organization.

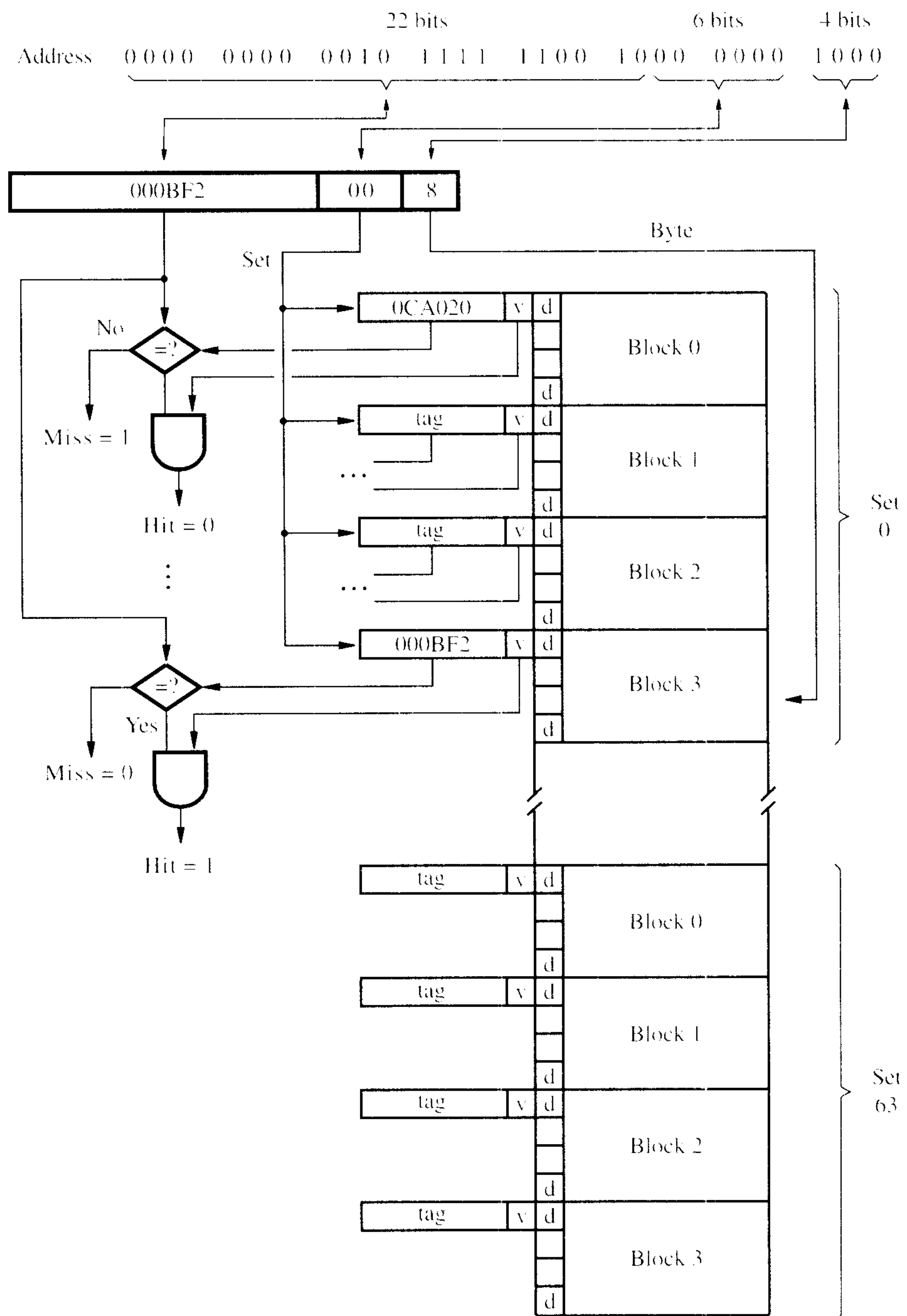


Figure 5.23 Data cache organization in the 68040 microprocessor.

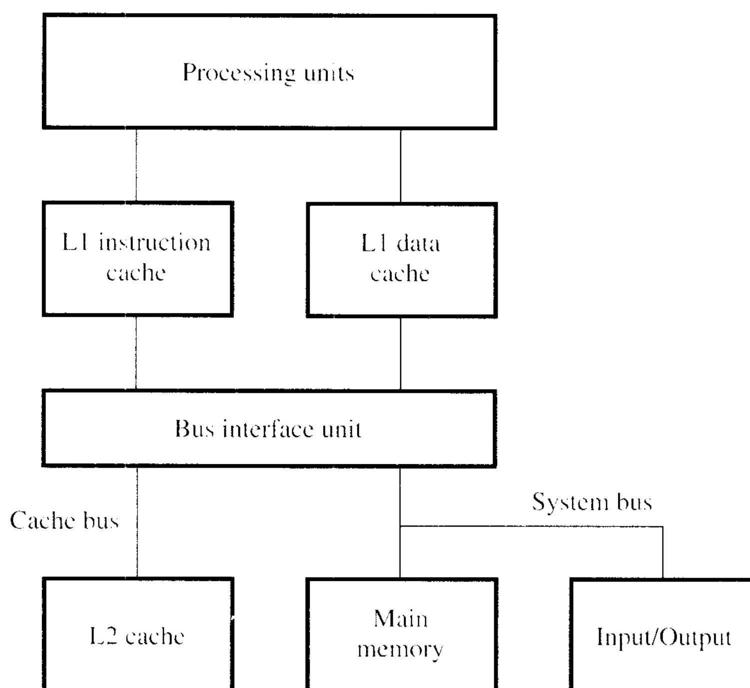


Figure 5.24 Caches and external connections in Pentium III processor.

Since instructions are normally not modified during execution of a program, there is no need for a write policy for the instruction cache.

The L2 cache is much larger. It holds both instructions and data. It is connected to the rest of the system as shown in Figure 5.24. A bus interface unit interconnects the caches, the main memory, and the I/O devices. Two separate buses are used: a fast *cache bus* connects the L2 cache to the processor, while a slower *system bus* connects the main memory and I/O devices.

The L2 cache can be implemented external to the processor chip, as is done in the Pentium III version known as Katmai. In this case, the cache contains 512 Kbytes and is implemented using SRAM memory. Its organization is four-way set-associative. It uses either the write-back or the write-through protocol, programmable on a per-block basis. The cache bus is 64 bits wide.

Improvements in VLSI technology have made it possible to integrate the L2 cache on the processor chip. This was done in the Pentium III version known as Coppermine. In this case, the cache size is 256 Kbytes. An eight-way set-associative organization is used. Since the L2 cache is on the processor chip, it is possible to use a wider 256-bit cache bus.

These examples raise an interesting question — is it better to implement the L2 cache externally or on the processor chip? External implementation allows for a larger cache. However, it is not conducive to a wide data path connection to the processor

because of the pins needed and increased power consumption of the output drivers. Also, external caches have lower clock speeds. The Katmai L2 cache is driven at half the speed of the processor clock, while Coppermine L2 cache is driven at full processor clock speed. Placing the L2 cache on the processor chip reduces the latency of access and increases the bandwidth because data are transferred using a wider path. This results in superior performance. The main drawback of integrating the L2 cache is that the processor chip becomes much larger, which makes it more difficult to fabricate.

Pentium 4 Caches

The Pentium 4 processor can have up to three levels of caches. The L1 cache consists of separate data and instruction caches. The data cache has a capacity of 8K bytes, organized in a 4-way set-associative manner. Each cache block has 64 bytes. The write-through policy is used on writes to the cache. Integer data can be accessed from the data cache in two clock cycles. Pentium 4 chips can use clock signals in excess of 1.3 GHz, which means that the data can be accessed in less than 2 ns. The instruction cache does not hold normal machine instructions. Instead, it holds already decoded versions of instructions, as will be discussed in Chapter 11.

The L2 cache is a unified cache with a capacity of 256K bytes, organized in an 8-way set-associative manner. Each of its blocks comprises 128 bytes. The write-back policy is used on writes to the cache. The access latency of this cache is seven clock cycles.

Both L1 and L2 caches are implemented on the processor chip. The architecture also allows for inclusion of an on-chip L3 cache. However, this cache is not implemented in the Pentium 4 chips targeted for desktop computers. It is intended for processor chips used in server systems.

5.6 PERFORMANCE CONSIDERATIONS

Two key factors in the commercial success of a computer are performance and cost; the best possible performance at the lowest cost is the objective. The challenge in considering design alternatives is to improve the performance without increasing the cost. A common measure of success is the *price/performance ratio*. In this section, we discuss some specific features of memory design that lead to superior performance.

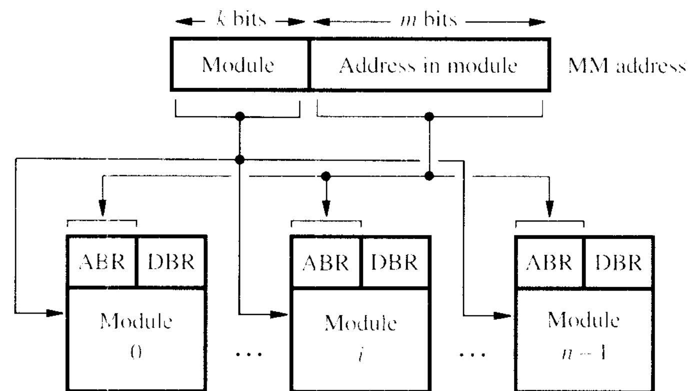
Performance depends on how fast machine instructions can be brought into the processor for execution and how fast they can be executed. We will discuss the speed of execution in Chapters 7 and 8, and show how additional circuitry can be used to speed up the execution phase of instruction processing. In this chapter, we focus on the memory subsystem.

The memory hierarchy described in Section 5.4 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. Each level of the hierarchy plays an important role. The speed and efficiency of data transfer between various levels of the hierarchy are also of great significance. It is beneficial if transfers to and from the faster units can be done at a rate equal to that of the

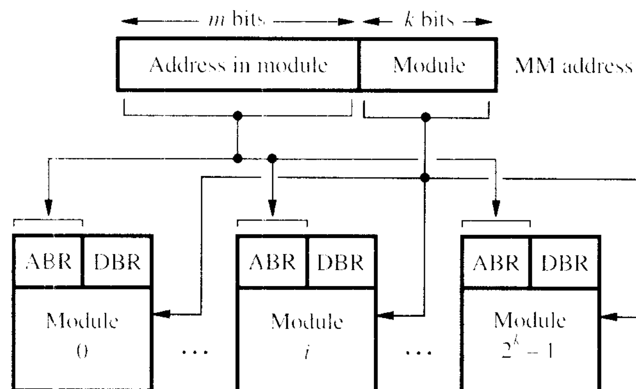
faster unit. This is not possible if both the slow and the fast units are accessed in the same manner, but it can be achieved when parallelism is used in the organization of the slower unit. An effective way to introduce parallelism is to use an interleaved organization.

5.6.1 INTERLEAVING

If the main memory of a computer is structured as a collection of physically separate modules, each with its own address buffer register (ABR) and data buffer register (DBR), memory access operations may proceed in more than one module at the same



(a) Consecutive words in a module



(b) Consecutive words in consecutive modules

Figure 5.25 Addressing multiple-module memory systems.

time. Thus, the aggregate rate of transmission of words to and from the main memory system can be increased.

How individual addresses are distributed over the modules is critical in determining the average number of modules that can be kept busy as computations proceed. Two methods of address layout are indicated in Figure 5.25. In the first case, the memory address generated by the processor is decoded as shown in Figure 5.25*a*. The high-order k bits name one of n modules, and the low-order m bits name a particular word in that module. When consecutive locations are accessed, as happens when a block of data is transferred to a cache, only one module is involved. At the same time, however, devices with direct memory access (DMA) ability may be accessing information in other memory modules.

The second and more effective way to address the modules is shown in Figure 5.25*b*. It is called *memory interleaving*. The low-order k bits of the memory address select a module, and the high-order m bits name a location within that module. In this way, consecutive addresses are located in successive modules. Thus, any component of the system that generates requests for access to consecutive memory locations can keep several modules busy at any one time. This results in both faster access to a block of data and higher average utilization of the memory system as a whole. To implement the interleaved structure, there must be 2^k modules; otherwise, there will be gaps of nonexistent locations in the memory address space.

Example 5.1

The effect of interleaving is substantial. Consider the time needed to transfer a block of data from the main memory to the cache when a read miss occurs. Suppose that a cache with 8-word blocks is used, similar to our examples in Section 5.5. On a read miss, the block that contains the desired word must be copied from the memory into the cache. Assume that the hardware has the following properties. It takes one clock cycle to send an address to the main memory. The memory is built with relatively slow DRAM chips that allow the first word to be accessed in 8 cycles, but subsequent words of the block are accessed in 4 clock cycles per word. (Recall from Section 5.2.3 that, when consecutive locations in a DRAM are read from a given row of cells, the row address is decoded only once. Addresses of consecutive columns of the array are then applied to access the desired words, which takes only half the time per access.) Also, one clock cycle is needed to send one word to the cache.

If a single memory module is used, then the time needed to load the desired block into the cache is

$$1 + 8 + (7 \times 4) + 1 = 38 \text{ cycles}$$

Suppose now that the memory is constructed as four interleaved modules, using the scheme in Figure 5.25*b*. When the starting address of the block arrives at the memory, all four modules begin accessing the required data, using the high-order bits of the address. After 8 clock cycles, each module has one word of data in its DBR. These words are transferred to the cache, one word at a time, during the next 4 clock cycles. During this time, the next word in each module is accessed. Then it takes another 4 cycles to transfer these words to the cache. Therefore, the total time needed to load the

block from the interleaved memory is

$$1 + 8 + 4 + 4 = 17 \text{ cycles}$$

Thus, interleaving reduces the block transfer time by more than a factor of 2.

In Section 5.2.4, we mentioned that interleaving is used within SDRAM chips to improve the speed of accessing successive words of data. The memory array in most SDRAM chips is organized as two or four banks of smaller interleaved arrays. This improves the rate at which a block of data can be transferred to or from the main memory.

5.6.2 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 a cache on the processor chip and the size of a 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 must be taken after a miss. The extra time needed to bring the desired information into the cache is called the *miss penalty*. This penalty is ultimately reflected in the time that the processor is stalled because the required instructions or data are not available for execution. In general, the miss penalty is the time needed to bring a block of data from a slower unit in the memory hierarchy to a faster unit. The miss penalty is reduced if efficient mechanisms for transferring data between the various units of the hierarchy are implemented. The previous section shows how an interleaved memory can reduce the miss penalty substantially.

Example 5.2 Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the processor is

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

We use the same parameters as in Example 5.1. If the computer has no cache, then, using a fast processor and a typical DRAM main memory, it takes 10 clock cycles for each memory read access. Suppose the computer has a cache that holds 8-word blocks and an interleaved main memory. Then, as we showed in Section 5.6.1, 17 cycles are

needed to load a block into the cache. 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. Let us further assume that the miss penalty is the same for both read and write accesses. Then, a rough estimate of the improvement in performance that results from using the cache can be obtained as follows:

$$\frac{\textit{Time without cache}}{\textit{Time with cache}} = \frac{130 \times 10}{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)} = 5.04$$

This result suggests that the computer with the cache performs five times better.

It is also interesting to consider how effective this cache is compared to an ideal cache that has a hit rate of 100 percent (in which case, all memory references take one cycle). Our rough estimate of relative performance for these caches is

$$\frac{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)}{130} = 1.98$$

This means that the actual cache provides an environment in which the processor effectively works with a large DRAM-based main memory that appears to be only two times slower than the circuits in the cache.

In this example, we made a simplifying assumption that the same clock is used to access the on-chip cache and the main memory via the system bus. A high-performance processor is likely to operate under the control of a clock that is much faster than the system bus clock, perhaps up to ten times faster. Let us consider the impact of a cache in a system of this type.

Example 5.3

Suppose that there is a single cache that is implemented on the processor chip and that the main memory is realized using SDRAM chips. Assume that the system bus clock is four times slower than the processor clock. As in Example 5.2, assume that a cache block contains 8 words, and that the hit rates in the cache are 0.95 for instructions and 0.9 for data. The SDRAM timing diagram is similar to Figure 5.9. The only difference is that there is a burst of 8 data words rather than four. Thus, according to Figure 5.9, it will take 14 clock cycles from when the RAS signal is asserted to transfer a block of data between the main memory and the cache. Since the RAS and CAS signals are generated by the memory controller, as indicated in Figure 5.11, one more cycle is needed during which the processor sends the address of the first word in a block to the memory controller. Therefore, a total of 15 cycles is needed to transfer a block. The cycles shown in Figure 5.9 are the system bus clock cycles. If the processor clock is four times faster, then it takes 60 processor cycles to transfer an 8-word block to or from the main memory. Note also that Figure 5.9 indicates that the processor can read or write a single word in the main memory in 9 bus clock cycles, consisting of the 8 cycles indicated in Figure 5.9 plus one cycle needed to send an address to the memory controller. Hence, 36 processor cycles are needed to access a single word in the main memory. Yet, the processor accesses a word in the cache in one processor cycle!

Repeating the calculation in Example 5.2 gives:

$$\frac{\textit{Time without cache}}{\textit{Time with cache}} = \frac{130 \times 36}{100(0.95 \times 1 + 0.05 \times 60) + 30(0.9 \times 1 + 0.1 \times 60)} = 7.77$$

Thus, accounting for the differences between processor and system bus clock speeds shows that the cache has an even greater positive effect on the performance.

In the preceding examples, we distinguish between instructions and data as far as the hit rate is concerned. Although hit rates above 0.9 are achievable for both, the hit rate for instructions is usually higher than that for data. The hit rates depend on the design of the cache and on the instruction and data access patterns of the programs being executed.

How can the hit rate be improved? An obvious possibility is to make the cache larger, but this entails increased cost. Another possibility is to increase the 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 as a consequence of a single miss, rather than loading several smaller blocks as a result of several misses. The efficiency of parallel access to blocks in an interleaved memory is the basic reason for this advantage. Larger blocks are effective up to a certain size, but eventually any further improvement in the hit rate tends to be offset by the fact that, in a larger block, some items may not be referenced before the block is ejected (replaced). The miss penalty increases as the block size increases. Since the performance of a computer is affected positively by increased hit rate and negatively by increased miss penalty, the block sizes that are neither very small nor very large give the best results. In practice, block sizes in the range of 16 to 128 bytes have been 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 the completion of the block transfer, the processor can continue as soon as the required word is loaded in the cache.

5.6.3 CACHES ON THE PROCESSOR CHIP

When information is transferred between different chips, considerable delays are introduced in driver and receiver gates on the chips. Thus, from the speed point of view, the optimal place for a cache is on the processor chip. Unfortunately, space on the processor chip is needed for many other functions; this limits the size of the cache that can be accommodated.

All high-performance processor chips include some form of a cache. Some manufacturers have chosen to implement two separate caches, one for instructions and another for data, as in the 68040, Pentium III, and Pentium 4 processors. Others have implemented a single cache for both instructions and data, as in the ARM710T processor.

A combined cache for instructions and data is likely to have a somewhat better hit rate because it offers greater flexibility in mapping new information into the cache. However, if separate caches are used, it is possible to access both caches at the same time.

which leads to increased parallelism and, hence, better performance. The disadvantage of separate caches is that the increased parallelism comes at the expense of more complex circuitry.

In high-performance processors two levels of caches are normally used. The L1 cache(s) is on the processor chip. The L2 cache, which is much larger, may be implemented externally using SRAM chips. But, a somewhat smaller L2 cache may also be implemented on the processor chip, as illustrated by the Coppermine version of Pentium III processors described in Section 5.5.4.

If both L1 and L2 caches are used, the L1 cache should be designed to allow very fast access by the processor because its access time will have a large effect on the clock rate of the processor. A cache cannot be accessed at the same speed as a register file because the cache is much bigger and, hence, more complex. A practical way to speed up access to the cache is to access more than one word simultaneously and then let the processor use them one at a time. This technique is used in many commercial processors.

The L2 cache can be slower, but it should be much larger to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 cache. A workstation computer may include an L1 cache with the capacity of tens of kilobytes and an L2 cache of several megabytes.

Including an L2 cache further reduces the impact of the main memory speed on the performance of a computer. The average access time experienced by the processor in a system with two levels of caches is

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

where

h_1 is the hit rate in the L1 cache.

h_2 is the hit rate in the L2 cache.

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

C_2 is the time to access information in the L2 cache.

M is the time to access information in the main memory.

The number of misses in the L2 cache, given by the term $(1 - h_1)(1 - h_2)$, should be low. If both h_1 and h_2 are in the 90 percent range, then the number of misses will be less than 1 percent of the processor's memory accesses. Thus, the miss penalty M will be less critical from a performance point of view. See Problem 5.18 for a quantitative examination of this issue.

5.6.4 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 immediately depend on the result of a write operation, so it is not necessary for the processor 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. Note that it is important that the read requests be serviced immediately because the processor usually cannot proceed without the data that are to be 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 compared with the addresses of the data in the write buffer. In case of a match, the data in the write buffer are used.

A different situation occurs with the write-back protocol. In this case, the write operations are simply performed on the corresponding word in the cache. But consider what happens when a new block of data is to be brought into the cache as a result of a read miss, which replaces 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 will have to wait longer for the new block to be read into the cache. It is more prudent to read the new block first. This can be arranged by providing a fast write buffer for temporary storage of the dirty block that is ejected from the cache while the new block is being read. Afterward, 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. A read miss occurs, and the desired data are loaded from the main memory. The processor has to pause until the new data arrive, which is the effect of the 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. However, the processor does not wait for the referenced data. A prefetch instruction is inserted in a program to cause the data to be loaded in the cache by the time they are needed in the program. 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. It is obviously preferable to have the compiler insert these instructions, which can be done with good success for many applications. Note that 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 through hardware. This involves adding circuitry that attempts to discover a pattern in memory references and then prefetches data according to this pattern. A number of schemes have been proposed for this purpose, but they are beyond the scope of this book. A description of these schemes is found in References [2] and [3].

Intel's Pentium 4 processor has facilities for prefetching information into its caches using both software and hardware approaches. There are special prefetch instructions that can be included in programs to bring a block of data into a desired level of cache. Harware-controlled prefetching brings cache blocks into the L2 cache based on the patterns of previous usage.

Lockup-Free Cache

The software prefetching scheme just discussed 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. A cache of this type is said to be locked while it services a miss. We can solve this problem by modifying the basic cache structure to allow the processor to access the cache while a miss is being serviced. In fact, it is desirable that more than one outstanding miss can be supported.

A cache that can support multiple outstanding misses is called *lockup-free*. Since it can service only one miss at a time, it 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 Control Data company [4].

We have used software prefetching as an obvious motivation for a cache that is not locked by a read miss. A much more important reason is that, in a processor that uses a pipelined organization, 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 stalling. We return to this topic in Chapter 8, where the pipelined organization is examined in detail.

5.7 VIRTUAL MEMORIES

In most modern computer systems, the physical main memory is not as large as the address space spanned by an address issued by 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 ranges from a few hundred megabytes to 1G bytes. When a program does not completely fit into the main memory, the parts of it not currently being executed are stored on secondary storage devices, such as magnetic disks. Of course, all parts of a program that are eventually executed are first brought into the

main memory. When a new segment of a program is to be moved into a full memory, it must replace another segment already in the memory. In modern computers, the operating system moves programs and data automatically between the main memory and secondary storage. Thus, the application programmer does not need to be aware of limitations imposed by the available main memory.

Techniques that automatically move program and data blocks into the physical main memory when they are required for execution are called *virtual-memory* techniques. Programs, and hence the processor, reference an instruction and data 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 components. 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. On the other hand, if the referenced address is not in the main memory, its contents must be brought into a suitable location in the memory before they can be used.

Figure 5.26 shows a typical organization that implements virtual memory. A special hardware unit, called the *Memory Management Unit* (MMU), translates virtual

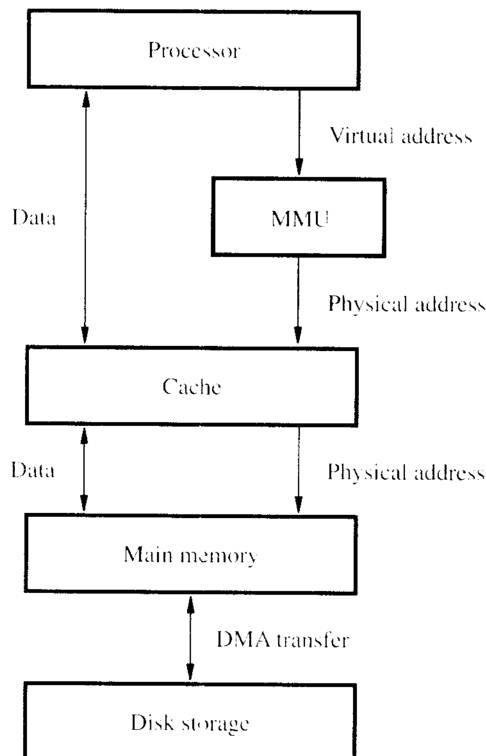


Figure 5.26 Virtual memory organization.

addresses into physical addresses. When the desired data (or instructions) are in the main memory, these data are fetched as described in our presentation of the cache mechanism. If the data are not in the main memory, the MMU causes the operating system to bring the data into the memory from the disk. Transfer of data between the disk and the main memory is performed using the DMA scheme discussed in Chapter 4.

5.7.1 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 moved between the main memory and the disk whenever the translation mechanism determines that a move 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 5.5 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 5.27. Each virtual address generated by the processor, whether it is for an instruction fetch or an operand fetch/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. This bit 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,

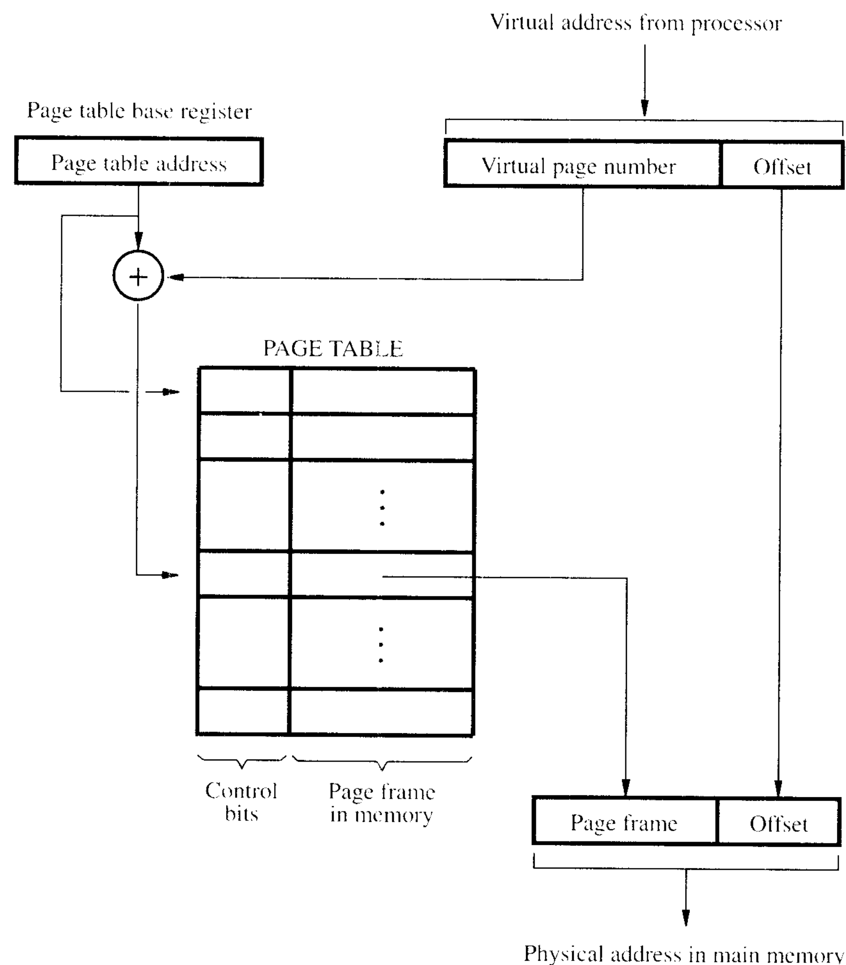


Figure 5.27 Virtual-memory address translation.

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.

The page table information is used by the MMU for every read and write access, so ideally, the page table should be situated within the MMU. Unfortunately, the page table may be rather large, and since the MMU is normally implemented as part of the processor chip (along with the primary cache), it is impossible to include a complete page table on this chip. Therefore, the page table is kept in the main memory. However, a copy of a small portion of the page table can be accommodated within the MMU.

This portion consists of the page table entries that correspond to the most recently accessed pages. A small cache, usually called the *Translation Lookaside Buffer* (TLB) is incorporated into the MMU for this purpose. The operation of the TLB with respect to the page table in the main memory is essentially the same as the operation we have discussed in conjunction with the cache memory. In addition to the information that constitutes a page table entry, the TLB must also include the virtual address of the entry. Figure 5.28 shows a possible organization of a TLB where the associative-mapping technique is used. Set-associative mapped TLBs are also found in commercial products.

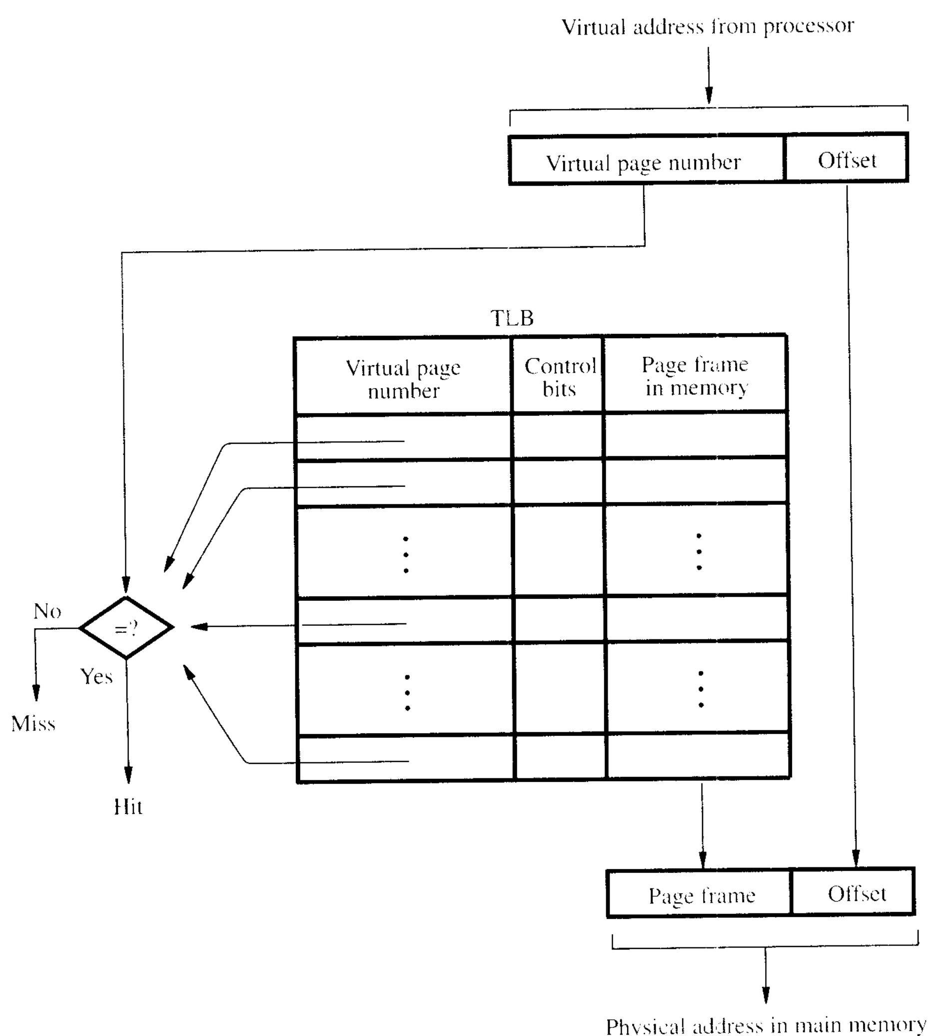


Figure 5.28 Use of an associative-mapped TLB.

physical memory, parts of it (pages) are moved from the
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.
Management routines are part of the operating system of the computer. It is conve-
nient 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 to be used in the translation process. Hence, by changing the contents of this regis-
ter, the operating system can switch from one space to another. The physical main mem-
ory is thus shared by the active pages of the system space and several user spaces. How-
ever, only the pages that belong to one of these spaces are accessible at any given time.
In any computer system in which independent user programs coexist in the main
memory, the notion of *protection* must be addressed. No program should be allowed to
destroy either the data or instructions of other programs in the memory. Such protection

5.9 SECONDARY STORAGE

Semiconductor memories discussed in the previous sections cannot be used to provide all of the storage capability needed in computers. Their main limitation is the cost per bit of stored information. Large storage requirements of most computer systems are economically realized in the form of magnetic disks, optical disks, and magnetic tapes, which are usually referred to as secondary storage devices.

5.9.1 MAGNETIC HARD DISKS

As the name implies, the storage medium in a magnetic-disk system consists of one or more disks mounted on a common spindle. A thin magnetic film is deposited on each disk, usually on both sides. The disks are placed in a rotary drive so that the magnetized surfaces move in close proximity to read/write heads, as shown in Figure 5.29*a*. The disks rotate at a uniform speed. Each head consists of a magnetic yoke and a magnetizing coil, as indicated in Figure 5.29*b*.

Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately underneath the head to switch to a direction parallel to the applied field. The same head can be used for reading the stored information. In this case, changes in the magnetic field in the vicinity of the head caused by the movement of the film relative to the yoke induce a voltage in the coil, which now serves as a sense coil. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film. Only changes in the magnetic field under the head can be sensed during the Read operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-to-1 and at 1-to-0 transitions in the bit stream. A long string of 0s or 1s causes an induced voltage only at the beginning and end of the string. To determine the number of consecutive 0s or 1s stored, a clock must provide information for synchronization. In some early designs, a clock was stored on a separate track, where a change in magnetization is forced for each bit period. Using the clock signal as a reference, the data stored on other tracks can be read correctly.

The modern approach is to combine the clocking information with the data. Several different techniques have been developed for such encoding. One simple scheme, depicted in Figure 5.29*c*, is known as *phase encoding* or *Manchester encoding*. In this scheme, changes in magnetization occur for each data bit, as shown in the figure. Note that a change in magnetization is guaranteed at the midpoint of each bit period, thus providing the clocking information. The drawback of Manchester encoding is its poor bit-storage density. The space required to represent each bit must be large enough to accommodate two changes in magnetization. We use the Manchester encoding example to illustrate how a *self-clocking* scheme may be implemented, because it is easy to understand. Other, more compact codes have been developed. They are much more efficient and provide better storage density. They also require more complex control circuitry. The discussion of such codes is beyond the scope of this book.