line where it is detected. Since the readout process is destructive, the data being read out is amplified and subsequently written back to the cell; this process may be combined with the periodic refreshing operation required by dynamic memories. The advantages of this DRAM cell are its small size, which means that ICs with very high cell density can be manufactured, and its low power consumption.

RAM design. A RAM IC typically contains all required access circuitry, including address decoders, drivers, and control circuits. Figure 6.10 shows a generic $2^m \times w$-bit RAM IC and identifies its control lines. *WE* is the write-enable line: a memory write (read) operation takes place if $WE = 1$ (0). A second control line, the chip-select line *CS*, triggers a memory operation. A word is accessed for either reading or writing only when *CS* is activated. This line signals that the data bus has a word ready to be written into the RAM or, in the case of a read operation, that the data bus is ready to receive a data word. The RAM of Figure 6.10 has a bidirectional data bus *D*, which is directly wired to all addressable storage locations, and so it requires a third control line, output enable *OE*. In write (input) operations this line is deactivated ($OE = 0$), allowing *D* to act as an input bus to all storage locations. Of course, only the addressed location actually stores the word received on *D*. In read (output) operations, *OE* must be activated ($OE = 1$) so that only the addressed memory location transfers its data to *D*.

A memory-design problem that the computer architect may encounter is the following: given that $N \times w$-bit RAM ICs denoted $M_{N,w}$ are available, design an $N' \times w'$-bit RAM, where $N' > N$ and/or $w' > w$. A general approach is to construct a $p \times q$ array of the $M_{N,w}$ ICs, where $p = \lceil N'/N \rceil$, $q = \lceil w'/w \rceil$, and $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. In this IC array each row stores N words (except possibly the last row), while each column stores a fixed set of w bits from every word (except possibly the last column). For example, to construct a 1GB RAM using 64M × 1-bit RAM ICs requires $p = 16$, $q = 8$, and a total of $pq = 128$ copies of the 64Mb RAM. When $N' > N$, additional external-address-decoding circuitry is usually required.

Consider the task of designing an $N \times 4w$-bit RAM using $N \times w$-bit ICs of the type appearing in Figure 6.10. Clearly, four ICs are needed to quadruple the word size in this way, since $p = 1$ and $q = 4$. The four are arranged in the 1×4 array configuration of Figure 6.11. Each RAM IC contains a w-bit slice of every stored word. Note how all the address and control lines are connected in exactly the same

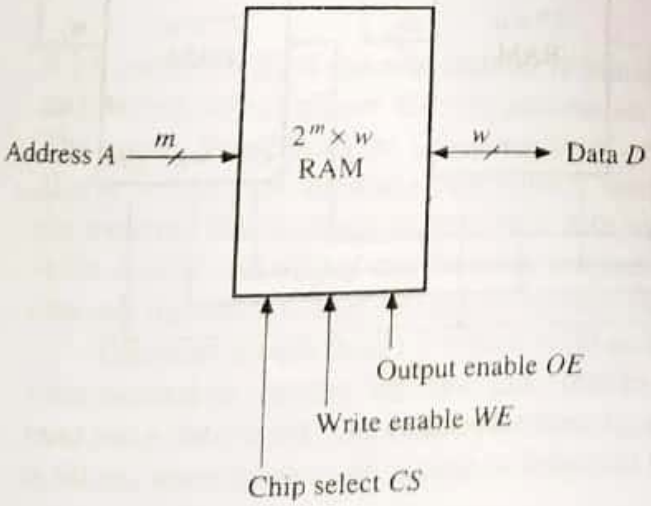


Figure 6.10
A RAM IC showing its major external connections.

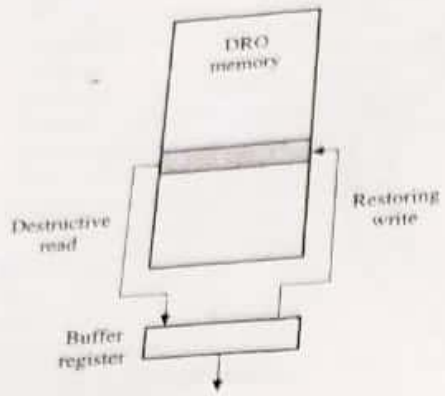


Figure 6.5
Memory restoration in a destructive readout (DRO) memory.

of information unless the charge is restored by a process called *refreshing*. Memories that require periodic refreshing are called *dynamic* memories, as opposed to *static* memories, which require no refreshing. (Note that in this context *dynamic* and *static* do not refer to the presence or absence of mechanical motion in the storage device.) Most memories that employ magnetic or optical storage techniques are static. Main memories are usually built from dynamic ICs referred to as *dynamic RAMS* (DRAMs). ICs can also implement static memories referred to as *static RAMS* (SRAMs). As Figure 6.2 indicates, SRAMs tend to be faster, that is, have lower access time, than DRAMs, but the cost per bit of SRAMs is higher. SRAMs are often used to build caches. A dynamic memory is refreshed in much the same way that data is restored in a DRO memory. The contents of every location are sent periodically to buffer registers and then returned in amplified form to their original locations.

Another physical process that can destroy the contents of a memory is the removal or failure of its power supply. A memory is *volatile* if the loss of power destroys the stored information. Information can be stored indefinitely in a volatile memory by providing battery backup or other means to maintain a continuous supply of power. Most IC memories are volatile, while most magnetic and optical memories are nonvolatile.

Figure 6.6 summarizes these characteristics for some important contemporary memory technologies.

Other characteristics. We defined the access time t_A as the time between the receipt of a read request signal by a memory and the delivery of the requested information to its output terminals. Some DRO and dynamic memories cannot initiate a new access until a restore or refresh operation has been carried out. Therefore, the minimum time that must elapse between the start of two consecutive access operations can be greater than t_A. This elapsed time is called the *cycle time* t_M of the memory and represents the time needed to complete a read or write operation.

The maximum amount of information that can be transferred to or from the memory per unit time is the *data-transfer rate* or *bandwidth* b_M and is measured in

respectively. SRAMs consist of memory cells that resemble the flip-flops used in processor design. SRAM cells differ from flip-flops primarily in the methods used to address the cells and transfer data to and from them. Multifunction lines minimize storage-cell complexity and the number of cell connections, thereby facilitating the manufacture of very large 2-D arrays of storage cells.

In a DRAM cell the 1 and 0 states correspond to the presence or absence of a stored charge in a capacitor controlled by a transistor switching circuit. Since a DRAM cell can be constructed around a single transistor, whereas a static cell requires up to six transistors, higher storage density is achieved with DRAMs. Indeed, DRAMs are among the densest VLSI circuits in terms of transistors per chip. The charge stored in a DRAM cell tends to decay with time, and the cell must be periodically refreshed. Hence a DRAM must contain refreshing circuitry and interleave refreshing operations with normal memory accesses. Both SRAMs and DRAMs are volatile, that is, the stored information is lost when the power source is removed.

Figure 6.9 shows examples of MOS RAM cells of both the static and dynamic varieties. The six-transistor SRAM cell (Figure 6.9*a*) superficially resembles a flip-flop. A signal applied to the address line (also called the *word line*) by the address decoder selects the cell for either the read or write operation. The two data lines (also called *bit lines*) are used in a complex way [Weste and Eshraghian 1992] to transfer the stored data and its complement between the cell and the data drivers.

Figure 6.9*b* shows a particularly simple and useful memory cell based on dynamic charge storage. This one-transistor DRAM cell comprises an MOS transistor T, which acts as a switch, and a capacitor C, which stores a data bit. Apart from power and ground, the cell has only two external connections: a data (bit) line and an address (word) line. To write information into the cell, a voltage signal (either high or low, representing 1 and 0, respectively) is placed on the data line. A signal is then applied to the address line to switch on T. This action transfers a charge to C if the data line is 1; no charge is transferred otherwise. To read the cell, the address line is again activated, transferring any charge stored in C to the data

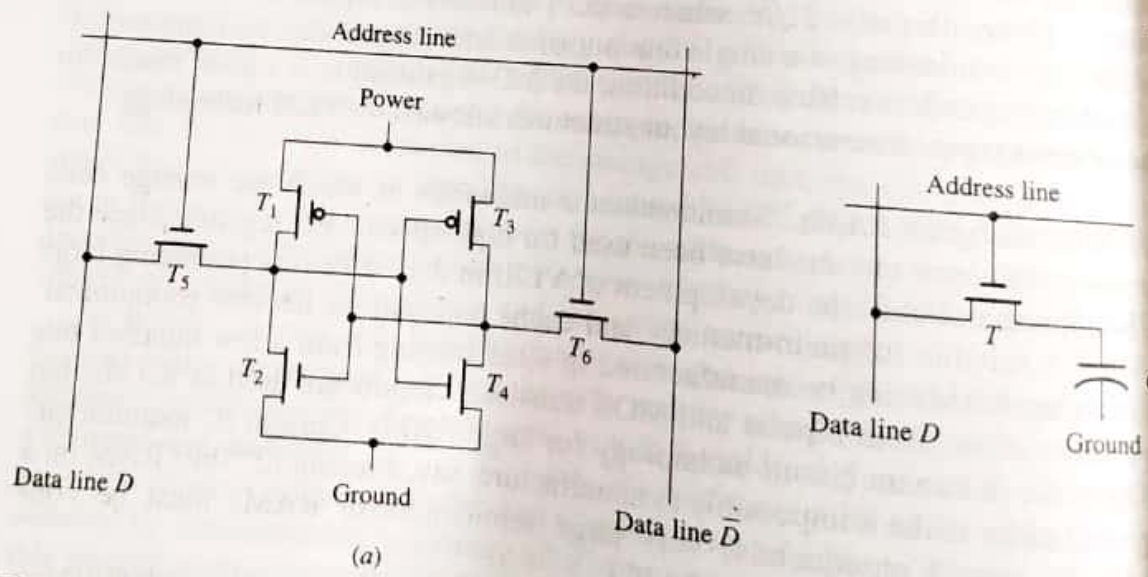


Figure 6.9
(*a*) Static and (*b*) dynamic RAM cells in MOS technology.

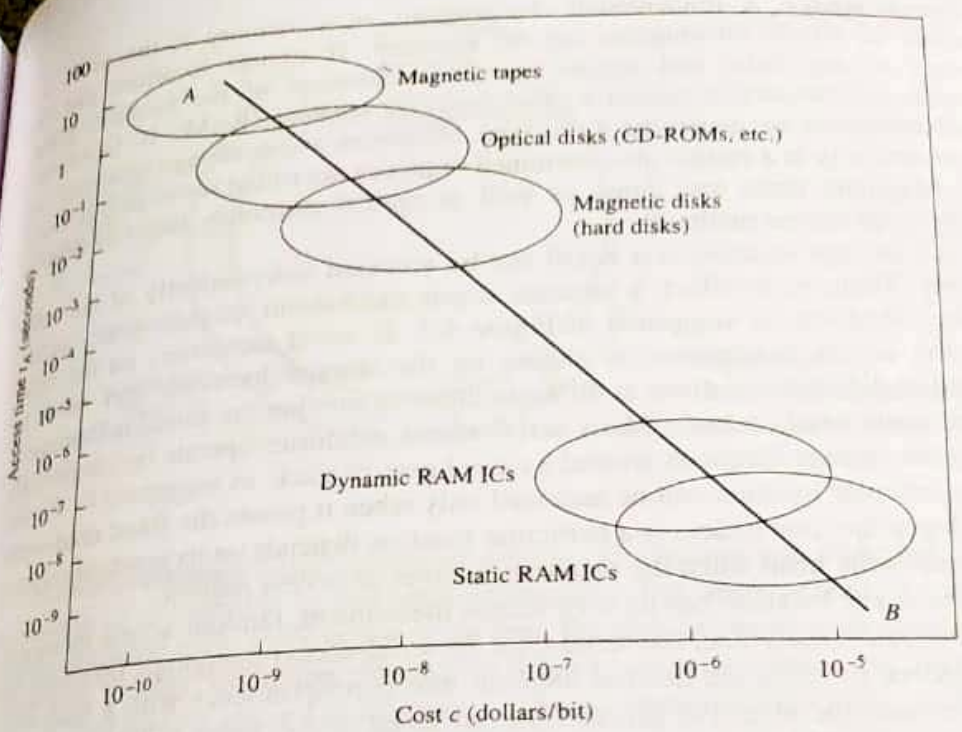


Figure 6.2
Access time versus cost for representative memory technologies.

Manufacturing improvements have steadily reduced the storage cost per bit c for the principal memory technologies. This trend is especially striking in the case of the IC RAMs used to construct main and cache memories, where the storage density per IC has increased steadily while the cost per IC has remained fairly constant. The state of the art in RAM manufacture circa 1975, 1985, and 1995 is represented by single-chip RAMs of capacity 4 Kb, 256 Kb, and 16 Mb, respectively. Here 1 Kb denotes a *kilobit* and equals 2^{10}, or 4096 bits, while 1 Mb denotes a *megabit* and equals 2^{20}, or 1,048,576 bits. At a typical introductory price of $40 for each chip type, the cost per bit c fell from around 0.01 dollars per bit in 1975 to 0.00015 dollars per bit in 1985 and to 0.0000024 dollars per bit a decade later. Similar developments have taken place in other technologies, notably magnetic (hard) disk memories, as storage density has increased steadily with little change in the cost per memory unit.

Although storage density has grown rapidly for the principal memory technologies, access times have decreased at a much slower rate. This disparity has tended to aggravate the speed mismatch—the von Neumann bottleneck—between the CPU and M. Memory speed has increased slowly, but the computing speed of microprocessors has spurted, along with their ability to produce and consume ever-increasing amounts of information. As we will see in this chapter, various design techniques can increase the effective rate at which the CPU can access the information stored in its memory system.

- *CPU registers.* These high-speed registers in the CPU serve as the working memory for temporary storage of instructions and data. They usually form a general-purpose *register file* for storing data as it is processed. A capacity of 32 data words is typical of a register file, and each register can be accessed, that is, read from or written into, within a single clock cycle (a few nanoseconds).
- *Main (primary) memory.* This large, fairly fast external memory stores programs and data that are in active use. Storage locations in main memory are addressed directly by the CPU's load and store instructions. While an IC technology similar to that of a CPU register file is used, access is slower because of main memory's large capacity and the fact that it is physically separated from the CPU. Main memory capacity is typically between 1 and 2^{10} megabytes, where a *megabyte*, also denoted 1 MB, is 2^{20} bytes, and 2^{10} MB = 2^{30} bytes is referred to as a *gigabyte* (1 GB). Access times of five or more clock cycles are usual.
- *Secondary memory.* This memory type is much larger in capacity but also much slower than main memory. Secondary memory stores system programs, large data files, and the like that are not continually required by the CPU. It also acts as an overflow memory when the capacity of the main memory is exceeded. Information in secondary storage is considered to be on-line but is accessed indirectly via input/output programs that transfer information between main and secondary memory. Representative technologies for secondary memory are magnetic hard disks and CD-ROMs (compact disk read-only memories), both of which have relatively slow electromechanical access mechanisms. Storage capacities of many gigabytes are common, while access times are measured in milliseconds.
- *Cache.* Most computers now have another level of IC memory—sometimes several such levels—called cache memory, which is positioned logically between the CPU registers and main memory. A cache's storage capacity is less than that of main memory, but with an access time of one to three cycles, the cache is much faster than main memory because some or all of it can reside on the same IC as the CPU. Caches are essential components of high-performance computers

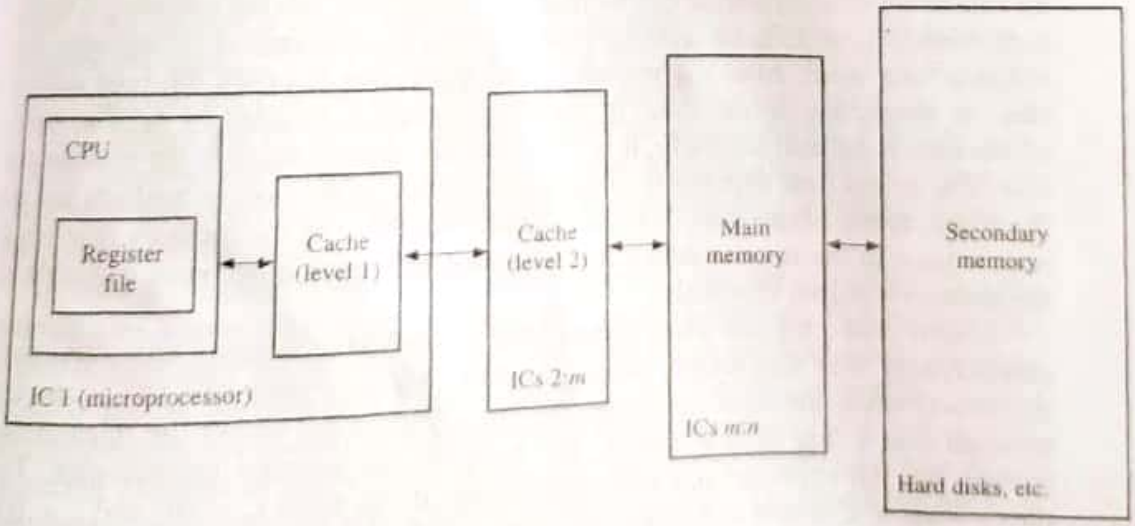


Figure 6.1
Conceptual organization of a multilevel memory system in a computer.

Memory Organization

This chapter is concerned with the design of a computer's memory system and its impact on performance. The characteristics of the most important storage-device technologies are surveyed. The behavior and management of multilevel hierarchical memory systems are discussed, and cache memories are examined in detail.

6.1 MEMORY TECHNOLOGY

Every computer contains several types of devices to store the instructions and data required for its operation. These storage devices plus the algorithms—implemented by hardware and/or software—needed to manage the stored information form the memory system of the computer.

6.1.1 Memory Device Characteristics

A CPU should have rapid, uninterrupted access to the external memories where its programs and the data they process are stored so that the CPU can operate at or near its maximum speed. Unfortunately, memories that operate at speeds comparable to processor speeds are expensive, and generally only very small systems can afford to employ a single memory using just one type of technology. Instead, the stored information is distributed, often in complex fashion, over various memory units that have very different performance and cost.

Memory types. The information-storage components of a computer can be placed in four groups, as illustrated in Figure 6.1.

Technology	storage medium	mode	Alterability	Permanence	access time t_A
[illegible]	Electronic	Random	Read/write	NDRO, volatile	10 ns
[illegible]	Electronic	Random	Read/write	DRO or NDRO, volatile	50 ns
[illegible]	Magnetic	Semirandom	Read/write	NDRO, nonvolatile	10 ms
[illegible]	Optical	Semirandom	Read/write	NDRO, nonvolatile	50 ms
[illegible]	Optical	Semirandom	Read only	NDRO, nonvolatile	100 ms
[illegible]	Magnetic	Serial	Read/write	NDRO, nonvolatile	1 s

Figure 6.6
Characteristics of some common memory technologies.

bits or words per second. If w is the number of bits that can be transferred simultaneously to or from the memory, then $b_M = w/t_M$ bits/s. If $t_M = t_A$, then $b_M = w/t_A$. Some memory types, particularly serial memories, require a long access time t_A to initiate a new access operation; once the operation is initiated, however, data transfer can proceed at a rate b_M much greater than w/t_A. In such cases the manufacturer provides independent specifications for t_A, t_M, b_M, and related performance parameters.

Finally, we mention *reliability*, which is measured by the mean time before failure (MTBF). In general, memories with no moving parts have much higher reliability than memories such as magnetic disks, which involve considerable mechanical motion. Even in memories without moving parts, reliability problems arise, particularly when very high storage densities or data-transfer rates are used. Error-detecting and error-correcting codes can increase the reliability of any memory.

6.1.2 Random-Access Memories

RAMs are distinguished by the fact that each storage location can be accessed independently with fixed access and cycle times that are independent of the position of the accessed location.

Organization. Figure 6.7 shows the main components of a RAM device such as a DRAM IC. At its heart is a storage unit composed of a large number (2^m) of addressable locations, each of which stores a w-bit word. Individual bits are not directly addressable unless $w = 1$. A RAM of this sort is referred to as a $2^m \times w$-bit or 2^m-word memory. The RAM operates as follows: First the address of the target location to be accessed is transferred via the address bus to the RAM's address

that aim to make $CPI \leq 1$. Unlike the three other memory types, caches are normally transparent to the programmer. Together, a computer's caches and main memory implement the *external memory* M addressed directly by CPU instructions.

The goal of every memory system is to provide adequate storage capacity with an acceptable level of performance and cost. We can achieve these goals by employing several memory types—with different cost/performance ratios—that are organized to provide a high average performance at a low average cost per bit. The individual memory units form a *multilevel hierarchy* of storage devices, as suggested by Figure 6.1. Successful operation of the hierarchy requires automatic storage-control methods that make efficient use of the available memory capacity. These methods should free the user from explicit management of memory space. They should also free programs from the particular memory environment in which they are executed.

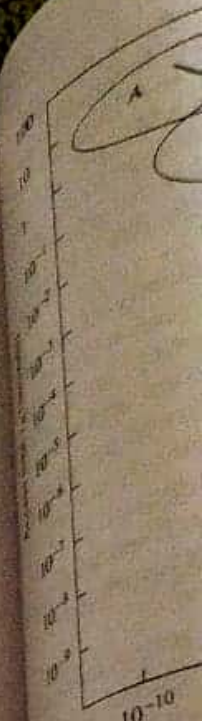
Performance and cost. The computer architect can choose from a bewildering variety of memory devices that employ various electronic, magnetic, and optical technologies and offer many cost/performance trade-offs [Cook and White 1994; Prince 1996]. However, all memories are based on just a few physical phenomena and organizational principles. We now examine the features common to the devices used to build cache, main, and secondary memories.

The most meaningful measure of the cost of a memory device is the purchase price to the user of a complete unit. The price should include not only the cost of the information storage medium itself but also the cost of the peripheral equipment (access circuitry) needed to operate the memory. Let C be the price in dollars of a complete memory system with S bits of storage capacity. We define the *cost* c of the memory as follows:

$$c = \frac{C}{S} \text{ dollars/bit}$$

The performance of an individual memory device is primarily determined by the rate at which information can be read from or written into the memory. A basic performance measure is the average time to read a fixed amount of information, for instance, one word, from the memory. This parameter is called the *read access time*, or simply the *access time*, of the memory and is denoted by t_A. The write access time is defined similarly; it is often, but not always, equal to the read access time. The access time depends on the physical nature of the storage medium and on the access mechanisms used. It is calculated from the time the memory receives a read request to the time at which the requested information becomes available at the memory's output terminals.

Clearly, low cost and short access time are desirable memory characteristics; unfortunately, they also tend to be incompatible. Memory units with fast access are expensive, while low-cost memories are slow. Figure 6.2 shows the relationship between cost c and access time t_A for some recent memory technologies. The straight line AB approximates this relationship. If we write $t_A = 10^y$ and $c = 10^x$, then $y = mx + k$, where m denotes the slope of AB and k is a constant. Hence $t_A = 10^{mx+k} = kc^m + k''$. From the data in Figure 6.2, we can conclude that $m \approx -0.5$. Hence to decrease t_A by a factor of 10, the cost c must increase by about 100.



igure 6.2
ccess time versus

Manufactur
or the principal
f the IC RAM
ensity per IC h
ast. The state
ased by single
here 1 Kb den
egabit and eq
ach chip type,
0015 dollar
imilar develo
ard) disk me
e cost per me
Although
nologies,
ded to aggr
e CPU and
processo
ssing ar
niques ca
e used in

and the desired access (read or write) operation. Each Rambus DRAM chip examines the address, and the DRAM unit R_i containing that address returns either a "ready" or a "busy" control signal to the master. If R_i is ready, the master then proceeds to transfer to R_i a data packet of up to 256 bytes (write case) or R_i sends the master a data packet (read case). This data transmission takes place in burst mode at speeds up to 500 MB/s, which implies accessing and transferring up to 1 byte every 2 ns. If R_i is busy with an earlier operation when an access request arrives, the master must try again later and a significant delay in response time occurs.

6.1.3 Serial-Access Memories

The data in a serial-access memory must be accessed in a predetermined order via read-write circuitry that is shared by different storage locations. Large serial memories typically store information in a fixed set of *tracks*, each consisting of a sequence of 1-bit storage cells. A track has one or more access points at which a read-write "head" can transfer information to or from the track. A stored item is accessed by moving either the stored information or the read-write heads or both. Functionally, a storage track in a serial memory resembles a shift register, so data transfer to and from a track is essentially serial.

Serial-access memories find their main application as secondary computer memories because of their low cost per bit and relatively long access times. Low cost is achieved by using very simple and small storage cells. Long access time is due to several factors:

- The read-write head positioning time.
- The relatively slow speed at which the tracks move.
- The fact that data transfer to and from the memory is serial rather than parallel.

Because access speed is so important, we now consider this factor in detail.

Access methods. Serial memories such as magnetic hard disks can be divided into those where each track has one or more fixed read-write heads and those whose read-write heads are shared among different tracks. In memories that share read-write heads, the need to move the heads between tracks introduces a delay. The average time to move a head from one track to another is the *seek time* t_S of the memory. Once the head is in position, the desired cell may be in the wrong part of the moving storage track. Some time is required for this cell to reach the read-write head so that data transfer can begin. The average time for this movement to take place is the *latency* t_L of the memory. In memories where information rotates around a closed track, t_L is called the *rotational latency*.

Each storage cell in a track stores a single bit. A w-bit word may be stored in two different ways. It can consist of w consecutive bits along a single track. Alternatively, w tracks may be used to store the word, with each track storing a different bit. By synchronizing the w tracks and providing a separate read-write head for each track, all w bits can be accessed simultaneously. In either case it is inefficient to read or write just one word per serial access, since the seek time and the rotational latency consume so much time. Words are therefore grouped into larger units called *blocks*. All the words in a block are stored in consecutive locations so that the time to access an entire block includes only one seek and one latency time.

Once the read-write head is positioned at the start of the requested word or block, data is transferred at a rate that depends on two factors: the speed of the stored information relative to the read-write head and the storage density along the track. The speed at which data can be transferred continuously to or from the track under these circumstances is the *data-transfer rate*. If a track has a storage density of T bits/cm and moves at a velocity of V cm/s past the read-write head, then the data-transfer rate is TV bits/s.

The time t_B needed to access a block of data in a serial-access memory can be estimated as follows. Assume that the memory has closed, rotating storage tracks of the type shown in Figure 6.4. Let each track have a fixed (average) capacity of N words and rotate at r revolutions per second. Let n be the number of words per block. The data-transfer rate of the memory is then rN words/s. Once the read-write head is positioned at the start of the desired block, its data can be transferred in approximately $n/(rN)$ seconds. The average latency is $1/(2r)$ seconds, which is the time needed for half a revolution. If t_S is the average seek time, then an appropriate formula for t_B is

$$t_B = t_S + \frac{1}{2r} + \frac{n}{rN} \qquad (6.1)$$

Memory organization. Figure 6.15 shows the overall organization of a serial-memory unit. Assume that each word is stored along a single track and that each access results in the transfer of a block of words. The address of the data to be accessed is applied to the address decoder, whose output determines the track to be

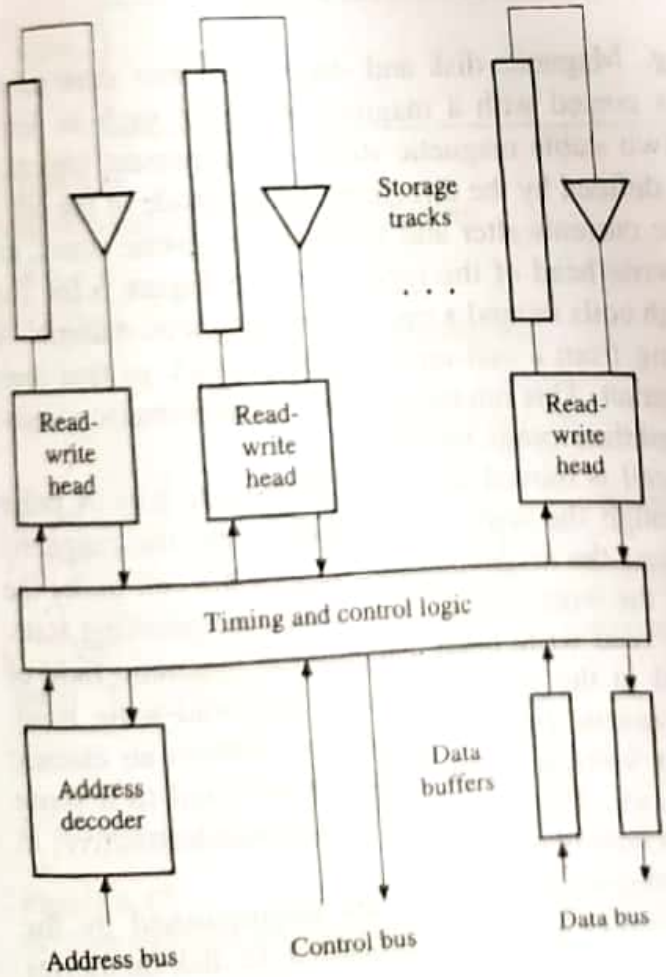


Figure 6.15
Organization of a serial-access memory unit.

tape speed is 50 in/s, the maximum data-transfer rate d is $110{,}000 \times 80/8 \times 50 = 55$ MB/s. A 200 m tape of this type can store about $55/50 \times 200/0.0254 = 8.661$ GB, a number that is reduced by formatting requirements. The time to scan or rewind an entire tape is about a minute.

Information stored on magnetic tapes is organized into blocks, usually of fixed length. A relatively large gap is inserted at the end of each block to permit the tape to start and stop between blocks. If the block length is bl and the interblock gap length is gl, then the tape's (space) utilization u is measured by

$$u = \frac{bl}{bl + gl} \tag{6.2}$$

For example, if $gl = 0.6$ in, the storage density $s = 3200$ b/in, and a block stores $bs = 4$KB of data, then $bl = 4096/3200 = 1.28$ in. Equation (6.2) implies that $u = 1.28/1.88 = 0.68$.

Because of the interblock gaps and the time needed to start and stop the tape between accesses, the effective data-transfer rate d_{eff} seen by the user is less than the quoted, maximum rate d. Let t_D denote the time to scan a data block, let t_G be the time to scan an interblock gap, and let t_{SS} be the time to start and stop the tape. Then

$$d_{\text{eff}} = \frac{t_D d}{t_D + t_G + t_{SS}}$$

If the block and gap sizes in bytes are bs and gs, respectively, then $t_D = bs/d$, and $t_G = gs/d$, so this equation becomes

$$d_{\text{eff}} = \frac{bs \cdot d}{bs + gs + t_{SS} \cdot d} \tag{6.3}$$

and the effective block access time t_B is $1/d_{\text{eff}}$. For example, with $bs = 4096$ bytes; $gl = 0.6$ in, corresponding to $gs = 1.92$ bytes; $d = 100{,}000$ bytes/s; and $t_{SS} = 2$ ms, Equation (6.3) yields $d_{\text{eff}} = 65{,}894$ bytes/s, a reduction of 34 percent from the maximum data-transfer rate.

Optical memories. Optical or light-based techniques for data storage have been the subject of intensive research for many years. Such memories usually employ optical disks, which resemble magnetic disks in that they store binary information in concentric tracks (or a spiral track in the CD-ROM case) on an electromechanically rotated disk. The information is read or written optically, however, with a laser replacing the read-write arm of a magnetic-disk drive. Optical memories offer extremely high storage capacities, but their access rates are generally less than those of magnetic disks. Read-only optical memories are well developed, but low-cost read-write memories have proven difficult to build.

The CD-ROM is a well-established read-only optical memory. CD-ROMs are an offshoot of the audio compact disks (CDs) introduced in the 1980s. They are manufactured in the same 12 cm format and can be mass-produced at very low cost per disk by injection molding. Binary data is stored in the form of 0.1 μm wide *pits* and *lands* (nonpitted areas) in circular tracks on a plastic substrate; see Figure 6.20. A laser beam scans the tracks and is reflected differently by the pits and lands. A mirror-and-lens system forms a read arm that can move back and forth across the tracks. The mirror can also be tilted slightly to provide fine tracking adjustments.

select a particular set of tracks for reading or writing. The recording surface is divided into *sectors* so that the part of a track within a sector stores a fixed amount of information corresponding to the memory unit's block size. Memory control is simplified if all tracks store the same amount of data, in which case the track density (bits stored per cm) on the outer tracks is less than the maximum possible.

Since their introduction in the 1950s by IBM, magnetic-disk memories have undergone steady evolution characterized by decreasing physical size and increasing storage density. Small flexible magnetic disks referred to as *floppy* disks form a compact, inexpensive, and portable medium for off-line storage of small amounts of data, for instance, 1.4 MB. They are contrasted with *hard* disks, which are often sealed into their drive units and have much higher storage capacity and reliability.

EXAMPLE 6.3 A COMMERCIAL MAGNETIC HARD-DISK MEMORY UNIT [QUANTUM CORP. 1996]. The XP39100 is a 9.3 GB hard-disk memory in the Atlas II series manufactured by Quantum Corp. and introduced in the mid-1990s. It is housed in a rectangular box whose dimensions are approximately 14.6 × 10.2 × 4.14 cm. It contains ten 3.5 in (8.89 cm) diameter disks, supplying a total of 20 recording surfaces, each with its own read-write head. Figure 6.18 summarizes the main features of this device. The cited capacity of 9.1 GB is for a *formatted* disk, which stores a directory and other control information needed to make the disk drive ready for use. The number of sectors along a track varies from 108 to 180, and each sector within a track accommodates a 512-byte block. While the sector size is fixed, the number of sectors per track varies due to the fact that the inner tracks are smaller and can therefore store less information at the maximum recording density of the magnetic medium. The average block access time given by Equation (6.1) with the data from Figure 6.18 is

$$t_B = 7.9 + 4.2 + 0.6 = 12.7 \text{ ms}$$

where $t_S = 7.9$ ms, $r = 0.120$ revs/ms, $n = 8$, and we take $(108 + 180)/2 = 144$ to be the average number of sectors per track, implying that $N = 144 \times 512 = 73{,}728$ bytes/track. Observe that the seek time is the major factor in t_B. The data-transfer rate $rN = 120 \times$

Parameter	Size
Disk diameter (form factor)	3.5 in (8.89 cm)
Number of disks	10
Number of recording surfaces	20
Number of read-write heads per recording surface	1
Number of tracks per recording surface	5964
Number of sectors per track	108 to 180
Storage capacity per track sector (block size)	512 bytes
Track-recording density	110,000 bits/in
Storage capacity per recording surface (formatted)	445 MB
Storage capacity of disk drive (formatted)	9.1 GB
Disk-rotation speed	7200 rev/min
Average seek time	7.9 ms
Average rotational latency	4.2 ms
Internal data-transfer rate	8.7 to 13.8 MB/s
External (buffered) data-transfer rate	20 to 40 MB/s

Figure 6.18

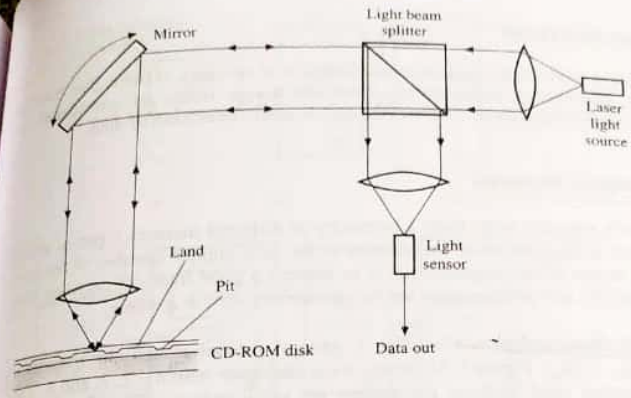


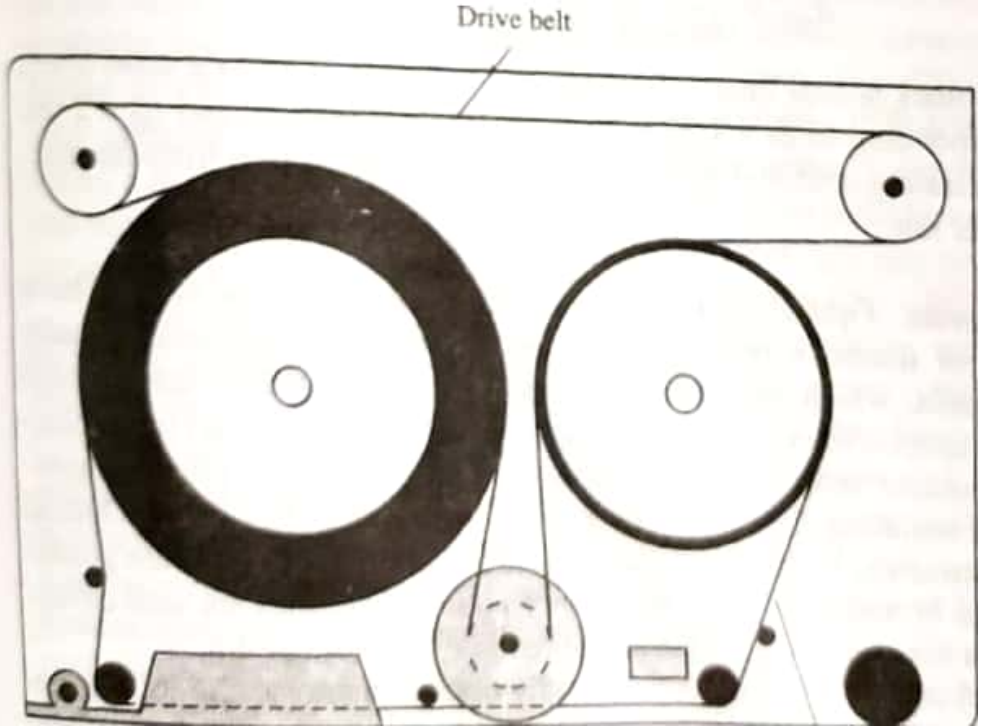
Figure 6.20
Optical readout mechanism for a CD-ROM.

The reflected light from the laser is picked up by a sensor and decoded to extract the stored information, which is then converted to electronic form for further processing. A standard 12 cm CD-ROM has a capacity of around 600 MB, which is enough to store some 240,000 pages of printed text—a large encyclopedia, for instance. Access time is about 100 ms, and data is transferred from the disk at a rate of 3.6 MB/s (in so-called 24-speed CD-ROM drives). Low-cost CD drives are available that allow computer users to create their own CD ROMs under such names as CD-recordable (CD-R) and CD-rewritable (CD-RW). They employ a laser to create (burn) pits on the surface of blank disks. A much denser type of CD called a *digital video disk* (DVD) has recently been introduced in both read-only and read-write forms. With two recording surfaces and one or two storage layers per surface, a DVD can have a capacity as high as 16 GB.

A few types of secondary memory devices combine magnetic and optical recording methods. A *magneto-optical disk* memory uses rotating disks that store information in magnetic form but are accessed by a laser beam similar to that in a CD-ROM drive. Like a magnetic disk, a magneto-optical disk has a magnetizable surface coating whose direction of magnetization can be polarized (up or down corresponding to 0 or 1) as depicted in Figure 6.16. A cell is read by bouncing a laser beam off it. The beam's angle of polarization is affected by the cell's magnetization direction, a phenomenon known as the *Kerr effect*. The slight change in the polarization angle of the reflected laser beam is sensed and decoded by the read mechanism. Writing is accomplished by using the laser beam to briefly heat a chosen cell above a specific temperature (the *Curie temperature* of the magnetic medium), at which point the cell's magnetic coercivity becomes zero, making the cell sensitive to external magnetic fields. An electromagnetic coil placed below the rotating disk then supplies a magnetic field of the required direction. The heated cell captures the magnetic field's direction which is retained after the cell cools below its Curie temperature.

Magnetic-tape memories. The magnetic-tape unit is one of cheapest forms of mass memory. Its main use today is to provide for a computer system in the event of failure of its hard disk subsy tape memories resemble domestic tape recorders, but instead of sound, they store binary digital information. The storage mediun strate a flexible plastic tape, usually packaged in a small cassette c ure 6.19 shows a standard memory of the data-cartridge type contai tape, which is 0.25 in (6.35 mm) wide and about 200 m long.

Data is stored on a tape in parallel, longitudinal tracks. Older nine such tracks designed to store one data byte and a parity bit newer tapes have as many as several hundred tracks. A read-write taneously access all tracks. Data transfer takes place when the ta constant velocity relative to a read-write head; hence the maximu rate depends largely on the storage density along the tape and the t example, if an 80-track tape has a per-track storage density of 11



level to n-level hierarchies. The average cost per bit of memory is given by

$$c = \frac{c_1 S_1 + c_2 S_2}{S_1 + S_2} \tag{6.4}$$

where c_i denotes the cost per bit of M_i and S_i denotes the storage capacity in bits of M_i. To reach the goal of making c approach c_2, S_1 must be much smaller than S_2.

The performance of a two-level memory is often measured in terms of the *hit ratio H*, which is defined as the probability that a virtual address generated by the CPU refers to information currently stored in the faster memory M_1. Since references to M_1 (*hits*) can be satisfied much more quickly than references to M_2 (*misses*), it is desirable to make H as close to one as possible. Hit ratios are generally determined experimentally as follows. A set of representative programs is executed or simulated. The number of address references satisfied by M_1 and M_2, denoted by N_1 and N_2, respectively, are recorded. H is calculated from the equation

$$H = \frac{N_1}{N_1 + N_2} \tag{6.5}$$

and is highly program dependent. The quantity $1 - H$ is called the *miss ratio*.

Let t_{A_1} and t_{A_2} be the access times of M_1 and M_2, respectively, relative to the CPU. The average time t_A for the CPU to access a word in the two-level memory is given by

$$t_A = H t_{A_1} + (1 - H) t_{A_2} \tag{6.6}$$

In most two-level hierarchies, a request for a word not in the fast level M_1 causes a block of information containing the requested word to be transferred to M_1 from M_2. When the block transfer has been completed, the requested word is available in M_1. The time t_B required for the block transfer is called the *block-access* or *block-transfer* time. Hence we can write $t_{A_2} = t_B + t_{A_1}$. Substituting into Equation (6.6) yields

$$t_A = t_{A_1} + (1 - H) t_B \tag{6.7}$$

In many cases $t_{A_2} >> t_{A_1}$; therefore, $t_{A_2} \approx t_B$. For example, a block transfer from secondary to main memory requires a relatively slow IO operation, making t_{A_2} and t_B much greater than t_{A_1}.

Let $r = t_{A_2}/t_{A_1}$ denote the access-time ratio of the two levels of memory. Let $e = t_{A_1}/t_A$, which is the factor by which t_A differs from its minimum possible value; e is called the *access efficiency* of the two-level memory. From Equation (6.6) we obtain

$$e = \frac{1}{r + (1 - r)H}$$

Figure 6.23 plots e as a function of H for various values of r. This graph shows the importance of achieving high values of H in order to make $e \approx 1$; that is, $t_A \approx t_{A_1}$. For example, suppose that $r = 100$. In order to make $e > 0.9$, we must have $H > 0.998$.

Memory capacity is limited by cost considerations; therefore, we do not want to waste memory space. The efficiency with which space is being used at any time can be defined as the ratio of the memory space S_u occupied by "active" or "useful"

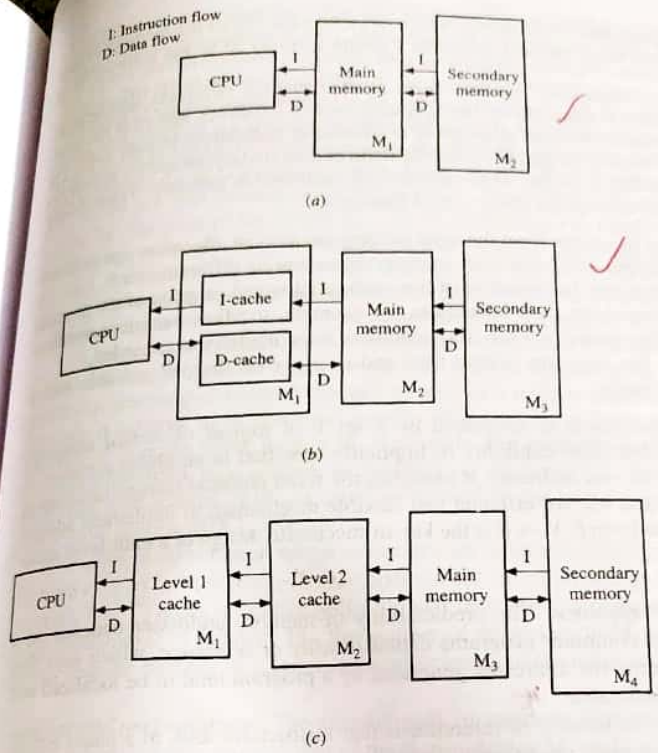


Figure 6.21
Common memory hierarchies with (*a*) two, (*b*) three, and (*c*) four levels.

M_i where $i \neq 1$, the address must be reassigned to M_1, the level of the memory hierarchy that the CPU can access directly. This relocation of addresses involves the transfer of data between levels M_i and M_1—a relatively slow process. For a memory hierarchy to work efficiently, the addresses generated by the CPU should be found in M_1 as often as possible. This approach requires that future addresses be to some extent predictable so that information can be transferred to M_1 before it is actually referenced by the CPU. If the desired data cannot be found in M_1, then the program originating the memory request must be suspended until an appropriate reallocation of storage is made.

Cache and virtual memory. The various parts of a memory hierarchy are controlled in very different fashions. Cache and main memory form a distinct subhierarchy whose design objective is to support CPU accesses with a minimum of delay. Hence hardware controllers that are transparent to both user and system programs

6.2 MEMORY SYSTEMS

This section examines the general characteristics of memory systems that have a multilevel, hierarchical organization. Two key design issues are considered in detail: automatic translation of addresses and dynamic relocation of data.

6.2.1 Multilevel Memories

A computer's memory units form a hierarchy of different memory types in which each member is in some sense subordinate to the next-highest member of the hierarchy. The object of this organization is to achieve a good trade-off between cost, storage capacity, and performance for the memory system as a whole.

General characteristics. Consider a general n-level system of n memory types $(M_1, M_2, \ldots, M_n)$. Figure 6.21 shows some examples with $n = 2$, 3, and 4. Typical technologies used in these hierarchies are semiconductor SRAMs for cache memory, semiconductor DRAMs for main memory, and magnetic-disk units for secondary memory. The two-level hierarchy of Figure 6.21*a* is typical of early computers. Figure 6.21*b* adds a cache of a type called a *split* cache, since it has separate areas for storing instructions (the I-cache) and data (the D-cache). The third example (Figure 6.21*c*) has two cache levels, both of the nonsplit or *unified* type. Embedded microcontrollers also use the various hierarchical organizations depicted in the figure, but often lack the secondary or the cache levels.

The following relations normally hold between adjacent memory levels M_i and M_{i+1} in a memory hierarchy:

Cost per bit	$c_i > c_{i+1}$
Access time	$t_{A_i} < t_{A_{i+1}}$
Storage capacity	$S_i < S_{i+1}$

The differences in cost, access time, and capacity between M_i and M_{i+1} can be several orders of magnitude. Considerable system resources are devoted to shielding the CPU from these differences, so it almost always sees a very large and inexpensive memory space and rarely sees an access time greater than that of M_1, the first (highest) level of the memory hierarchy.

The CPU and other processors can communicate directly with M_1 only, M_1 can communicate with M_2, and so on. Consequently, for the CPU to read information held in some memory level M_i requires a sequence of i data transfers of the form

$$M_{i-1} := M_i; \quad M_{i-2} := M_{i-1}; \quad M_{i-3} := M_{i-2}; \quad \ldots \quad M_1 := M_2; \quad CPU := M_1.$$

An exception is allowed in the case of caches; the CPU is designed to bypass the cache level(s) and go directly to main memory, as we will see later. In general, all the information stored in M_i at any time is also stored in M_{i+1}, but not vice versa.

During program execution the CPU produces a steady stream of memory addresses. At any time these addresses are distributed in some fashion throughout the memory hierarchy. If an address is generated that is currently assigned only to

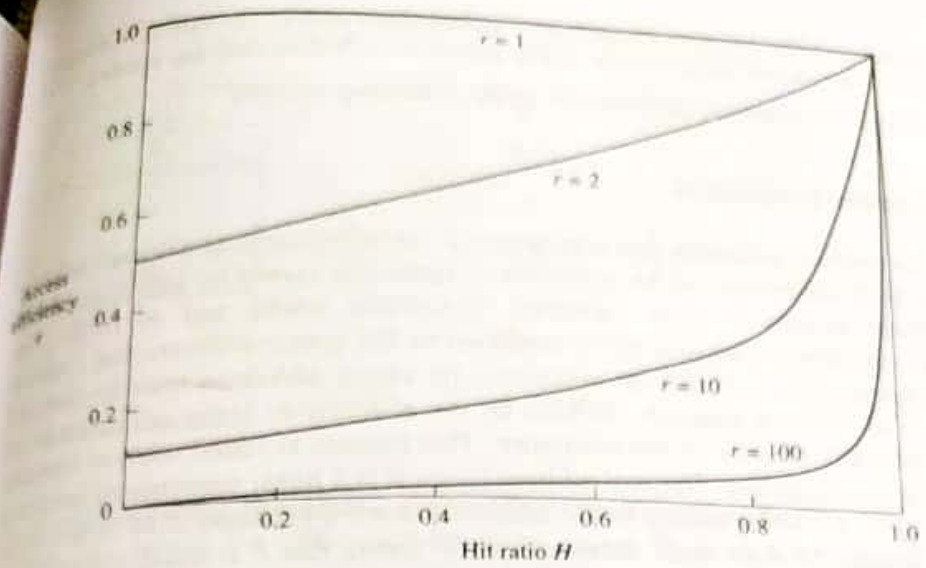


Figure 6.23
Access efficiency $e = t_{A_1}/t_A$ of a two-level memory as a function of hit ratio H for various values of $r = t_{A_2}/t_{A_1}$.

user programs and data to the total amount of memory space available S. We call this the *space utilization* u and write

$$u = \frac{S_u}{S}$$

Since memory space is more valuable in M_1 than in M_2, it is useful to restrict u to measuring M_1's space utilization. In that case the $S - S_u$ words of M_1 that represent "wasted" space can be attributed to several sources.

- *Empty regions.* The blocks of instructions and data occupying M_1 at any time are generally of different lengths. As the contents of M_1 are changed, unoccupied regions or holes of various sizes tend to appear between successive blocks. This phenomenon is called *fragmentation.*
- *Inactive regions.* Data may be transferred to M_1, for example, as part of a page, and may be subsequently transferred back to M_2 without ever being referenced by a processor. Some superfluous transfers of this kind are unavoidable, since address references are not fully predictable.
- *System regions.* These regions are occupied by the memory-management software.

A central issue in managing (M_1,M_2), or any multilevel memory, is to make it appear to its users like a single, fast memory of high capacity. This goal can be achieved in a way that is largely transparent to the users by providing a memory management system that automatically performs the following tasks:

- Translation of memory addresses from the *virtual* addresses encountered in program execution to the *real* addresses that identify physical storage locations.

- *Dynamic (re)allocation or swapping of information among the different memory levels so that stored items reside in the fastest level before they are needed.*

These issues are explored individually in the following sections.

6.2.2 Address Translation

The set of abstract locations that a program Q can reference is Q's virtual address space V. Such addresses can be explicitly or implicitly named by identifiers that a programmer assigns to data variables, instruction labels, and so forth. The addresses can also be constructed or modified by the system software that controls Q. To execute Q on a particular computer, its virtual addresses must be mapped onto the real address space R, defined by the addressable (external) memory M that is physically present in the computer. This process is called address translation or address mapping. The real address space R is a linear sequence of numbers $0, 1, 2, \ldots, n-1$ corresponding to the addressable word locations in M. It is convenient to identify M with main memory, while noting that R is usually distributed over several levels of the memory hierarchy, including the cache and the level labeled "main" memory. V is a loose collection of lists, multidimensional arrays, and other nonlinear structures, so it is much more complex than R.

Address translation can be viewed abstractly as a function $f: V \rightarrow R$. This function is not easily characterized, since address assignment and translation is carried out at various stages in the life of a program, specifically:

1. By the programmer while writing the program.
2. By the compiler during program compilation.
3. By the loader at initial program-load time.
4. By run-time memory management hardware and/or software.

Explicit specification of real addresses by the programmer was necessary in early computers, which had neither hardware nor software support for memory management. With modern computers, however, programmers normally deal only with virtual addresses. Specialized hardware and software within the computer automatically determine the real addresses required for program execution.

A compiler transforms the symbolic identifiers of a program into binary addresses. If the program is sufficiently simple, the compiler can completely map virtual addresses to real addresses. Address translation can also be completed when the program is first loaded for execution. This process is called *static* translation, since the real address space of the program is fixed for the duration of its execution. It is often desirable to vary the virtual space of a program dynamically during execution; this process is *dynamic* translation. For example, a recursive procedure—one that calls itself—is typically controlled by a stack containing the linkage between successive calls. The size of this stack cannot be predicted in advance because it depends on the number of times the procedure is called; therefore, it is desirable to allocate stack addresses on the fly. Hardware-implemented *memory management units* (MMUs) have come into widespread use for run-time address translation.

Base addressing. An executable program comprises a set of instruction and data blocks each of which is a sequence of words to be stored in consecutive mem-

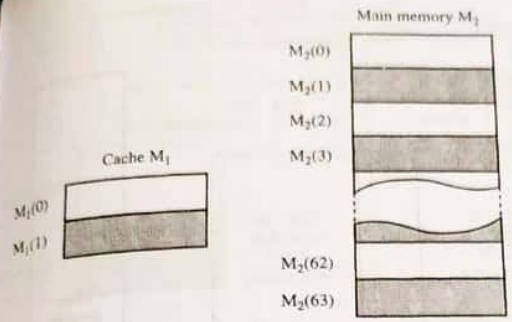


Figure 6.47
Direct-mapped cache with block capacity of two.

one specific set $M_1(j)$ in M_1. The set address j is determined from i by the rule

$$j = i \qquad \text{(modulo } s_1\text{)}$$

For example, if $s_1 = 2$ as in Figure 6.47, every even-address (unshaded) block in M_2 is mapped into $M_1(0)$ and every odd-address (shaded) block in M_2 is mapped into $M_1(1)$.

The hardware needed to implement direct mapping is fairly simple. The low-order s bits of each block address A form a set address that identifies the unique cache set that can store the block in question. The remaining t high-order bits of A now constitute the tag, and only these bits need be stored in the cache's tag memory. Consequently, the cache tag memory can be an ordinary RAM that is addressed by the s-bit set-address part of an incoming memory address A. If there are 2^d words per set, then the low-order d bits of A form the displacement address of the word in question within its block. Thus an incoming address has three parts: a t-bit tag, an s-bit set address, and a d-bit displacement.

The main drawback of direct mapping is that the cache's hit ratio drops sharply if two or more frequently used blocks happen to map onto the same region in the cache. This possibility is minimized by the fact that such blocks are relatively far apart in the memory-address space. For example, if $s_1 = 2^6 = 64$, then only the blocks with addresses i, $i + 64$, $i + 128$, $i + 192$,... can be mapped into the same cache set $M_1(i)$.

EXAMPLE 6.8 DESIGN OF A DIRECT-MAPPED CACHE [INTEGRATED DEVICE TECHNOLOGY 1994]. In this example we will use off-the-shelf ICs to design an add-on direct-mapped cache memory for a high-end microprocessor, such as the PowerPC. If the CPU has a built-in (level 1) cache, as is frequently the case, this design applies to a level 2 (secondary) cache. The CPU is linked to a byte-addressable external memory via a 32-bit address bus and a 64-bit bidirectional data bus using the look-aside design style of Figure 6.41*a*. The desired cache capacity is 256 KB, and the cache block (set) size is assumed to be 32 bytes (32 B). Hence the cache must store $8K = 2^{13}$ blocks, implying that we need a cache tag memory of capacity $8K \times t$ bits to store tags of length t. We also need a cache data memory of capacity $32K \times 64$ bits, where the 64-bit word size is determined by the system data bus. As shown in Figure 6.48, a 32-bit address generated by the CPU contains a 5-bit displacement to address a

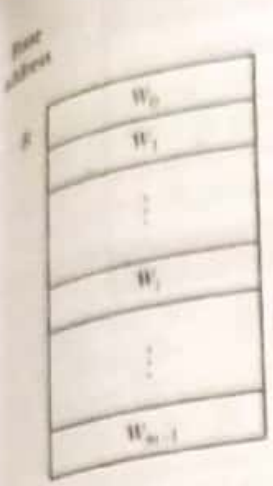


Figure 6.24
Block of m words with (base) address B.

ory locations during execution. A word W within a block has its own *effective address* A_{eff}, which the CPU must know to access W. (For the moment, we will ignore the distinction between the real and virtual address spaces.) W is also specified by the address B, called the *base address*, of the block that contains it, along with W's relative address or *displacement* D (also called an offset or index) within the block, as shown in Figure 6.24. Clearly,

$$A_{eff} = B + D \tag{6.8}$$

Often the address is designed so that B supplies the high-order bits of A_{eff} while D supplies the low-order bits thus:

$$A_{eff} = B.D \tag{6.9}$$

Now A_{eff} is formed simply by concatenating B and D, a process that does not significantly increase the time for address generation.

A simple way to implement static and dynamic address mapping is to put base addresses in a *memory map* or *memory address table* controlled by the memory management system. The table can be stored in memory, in CPU registers, or in both. The address-generation logic of the CPU computes an effective address A_{eff} by combining the displacement D with the corresponding base address B_i according to (6.8) or (6.9).

Blocks are easily relocated in memory by manipulating their base addresses. Figure 6.25 illustrates block relocation using base-address modification. Suppose that two blocks are allocated to main memory M as shown in Figure 6.25*a*. It is desired to load a third block K_3 into M; however, a contiguous empty space, or "hole," of sufficient size is unavailable. A solution to this problem is to move block K_2, as shown in Figure 6.25*b*, by assigning it a new base address B_2' and reloading it into memory. This creates a gap into which block K_3 can be loaded by assigning to it an appropriate base address.

With dynamic memory allocation, we must control the references made by a block to locations outside the memory area currently assigned to it. The block can be permitted to read from certain locations, but writing outside its assigned area must be prevented. A common way of doing this is by specifying the highest address L_i, called the *limit address*, that the block can access. Equivalently, the size of the block may be specified. The base address B_i and the limit address L_i are

no-match signal identifies a cache miss, and the memory access requested is forwarded to main memory for service. A cache block containing the target address is then sent from main memory to the cache, and at the same time, a data word is sent to the CPU or transferred from the CPU to the cache, in response to the original access request.

Associative memory. Figure 6.44 shows the general structure of an associative memory. Each unit of stored information is a fixed-length word. Any subfield of the word can be chosen as the key. Here the desired key is specified by a *mask register*, whose contents identify the bit positions (which need not be adjacent) that define the key. The current key is compared simultaneously with all stored words; those that match the key output a match signal, which enters a *select circuit*, which enables the data field to be accessed. If several entries have the same key, then the select circuit determines which data field is to be read out. It can, for example, read out all matching entries in some predetermined order. Since all words in the memory are required to compare their keys with the input key simultaneously, each needs its own *match circuit*. The match and select circuits make associative memories much more complex and expensive than conventional memories. Although VLSI techniques have made associative memories economically feasible, cost considerations still limit them to applications in which a relatively small amount of information must be accessed very rapidly, such as address mapping for caches.

The logic circuit for a 1-bit associative memory cell appears in Figure 6.45 [Triebel and Chu 1982]. The cell comprises a D flip-flop for data storage, a match circuit (the EXCLUSIVE-NOR gate) for comparing the flip-flop's contents to an external data bit D, and circuits for reading from and writing into the cell. The results of a comparison appear on the match output M, where $M = 1$ denotes a match and $M = 0$ denotes no match. The cell is selected or addressed for both read and write operations by setting the select line S to 1. New data is written into the cell by setting the write enable line WE to 1, which in turn enables the D flip-flop's

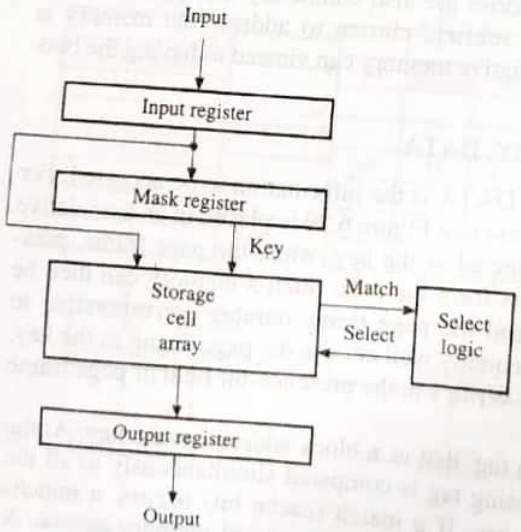
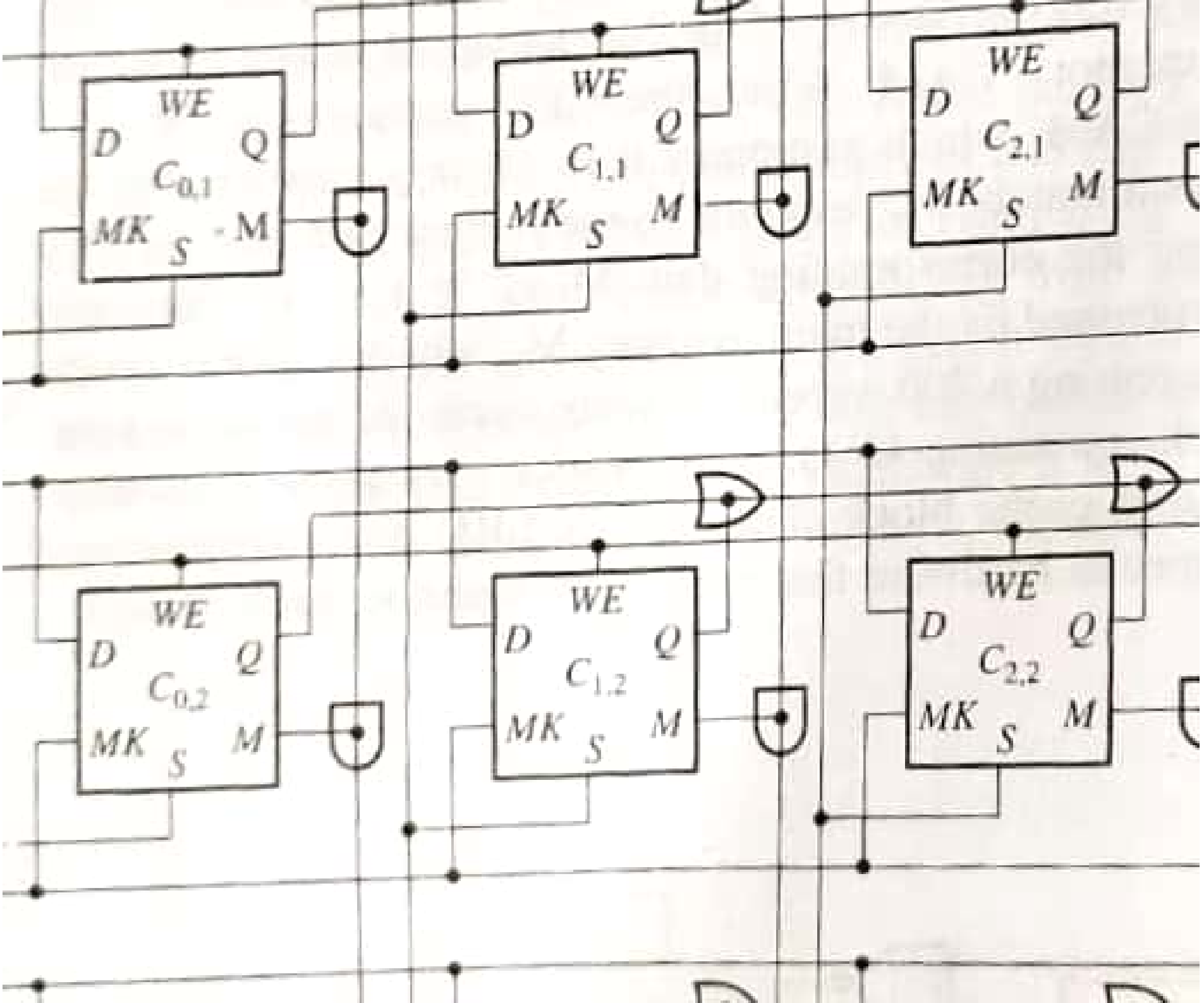


Figure 6.44
Structure of an associative (content addressable) memory.



clock input CK. The stored data is read out via the Q line. The mask control line MK is activated ($MK = 1$) to force the match line M to 0 independently of the data stored in the D flip-flop; MK also disables the input circuits of the flip-flop by forcing CK to 0. A cell like that of Figure 6.45 can be realized with about 10 transistors—far more than the single transistor required for a dynamic RAM cell (refer to Figure 6.9*b*). This high hardware cost is the main reason that large associative memories are rarely used outside caches.

Associative cells of the preceding type can be combined into word-organized associative memory arrays. Figure 6.46 shows a 16-bit associative memory that stores four words (columns) of 4 bits each. The words are individually addressable via their S lines. All words share a common set of data and mask lines for each bit position. Consequently, an external data bit D_i can be compared simultaneously to the ith stored bit of every word in the memory. The output lines of the cells are designed so that they can be connected to form wired OR or AND gates, as indicated in the figure.

A small associative cache is found in Data General Corp.'s ECLIPSE, a 16-bit computer from the 1970s. This computer has a modular memory design in which each 8K-word main-memory module M_2 is paired with a cache M_1 that stores sixteen 16-bit words forming four 4-word blocks. M_2 is constructed from MOS RAM chips with a 700 ns cycle time, while the cache M_1 uses bipolar RAMs with a cycle time of 200 ns. The memory (tag) addresses of the blocks stored in the cache are placed in an associative memory CAM. When the CPU generates a memory address A, it is sent to the CAM, which compares it to all tags currently in the cache. If the CAM indicates a match, M_1 responds to the memory request directly by either reading or writing the corresponding data $M(A)$. If A is not currently assigned to M_1, then A is processed by the main memory M_2, which responds to the original CPU request by executing a 700 ns read or write cycle. At the same time, M_2 sends a four-word block containing $M(A)$ to M_1, which uses the new block to replace the least recently used cache block. The cache's LRU block replacement policy is implemented by special hardware that constantly monitors cache usage.

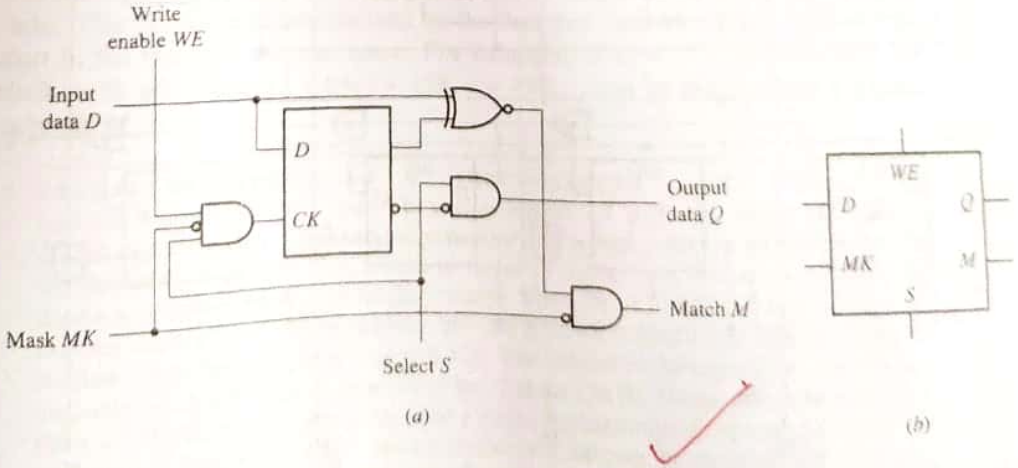


Figure 6.45
Associative memory cell: (*a*) logic circuit and (*b*) symbol.

A faster, but more costly organization called a *look-through* cache appears in Figure 6.41*b*. The CPU communicates with the cache via a separate (local) bus that is isolated from the main system bus. The system bus is available for use by other units, such as IO controllers, to communicate with main memory. Hence cache accesses and main-memory accesses not involving the CPU can proceed concurrently. Unlike the look-aside case, with a look-through cache the CPU does not automatically send all memory requests to main memory; it does so only after a cache miss. A look-through cache allows the local bus linking M_1 and M_2 to be wider than the system bus, thus speeding up cache–main-memory transfers. For example, if the system data bus is 32 bits wide and the cache block size is 128 bits = 16 bytes (a typical value), a 128-bit data bus might be provided to link M_1 and M_2, which would allow a cache block to be replaced in as little as a single clock cycle. The main disadvantage of the look-through design, besides its higher complexity and cost, is that it takes longer for M_2 to respond to the CPU when a miss occurs.

Cache operation. Figure 6.42 shows a small cache system that illustrates the relationship between the data stored in the cache M_1 and the data stored in main memory M_2. Here a cache block (line) size of 4 bytes is assumed. Each memory address is 12 bits long, so the 10 high-order bits form the tag or block address, and the 2 low-order bits define a displacement address within the block. When a block is assigned to M_1's data memory, its tag is also placed in M_1's tag memory. Figure 6.42 shows the contents of two blocks assigned to the cache data memory; note the locations of the same blocks in main memory. To read the shaded word, its address A_i = 101111000110 is sent to M_1, which compares A_i's tag part to its stored tags and finds a match (hit). The stored tag pinpoints the corresponding block in M_1's

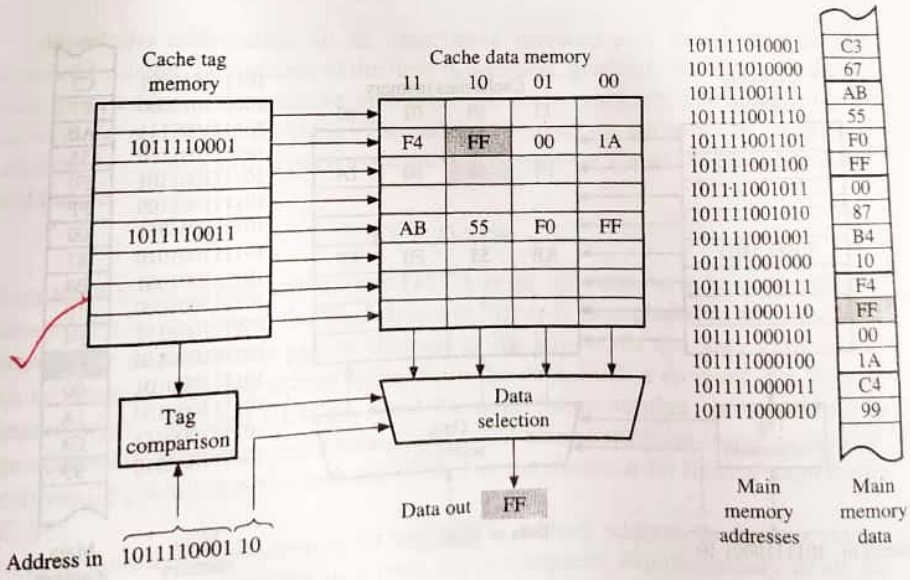


Figure 6.42
Cache execution of a read operation.

at which TLB contents. There is no delay and replacement policy. Although less efficient than LRU, this policy appears to work quite well in practice.

6.3 CACHES

The term *cache* refers to a fast intermediate memory within a larger memory system [Smith 1982; Handy 1993]. Although caches appeared as early as 1968 in the IBM System/360 Model 91, they did not come into wide use until the appearance of low-cost, high-density RAM and microprocessor ICs in the 1980s. Caches directly address the von Neumann bottleneck by providing the CPU with fast, single-cycle access to its external memory. They also provide an efficient way to place a small portion of memory on the same chip as a microprocessor. If an additional off-chip cache is used that employs, say, fast SRAM technology—and the continuing disparity between processor and DRAM speeds makes that desirable—a two-level cache organization results (refer to Figure 6.21*c*).

A cache serves as a buffer between a CPU and its main memory; in this section we focus on caches used in this way. However, caches appear as buffer memories in several other contexts. We saw in section 6.2 that the translation look-aside buffers (TLBs) used within a memory management system are specialized caches that permit very fast translation of memory addresses. Data buffers built into high-speed secondary memory devices such as hard disk drives are also called caches.

6.3.1 Main Features

The cache and main memory form a two-level subhierarchy (M_1,M_2) that differs in important ways from the main–secondary system (M_2,M_3); Figure 6.39 summarizes these differences. Because it is higher in the memory hierarchy, the pair (M_1,M_2) functions at much higher speed than (M_2, M_3). The access time ratio t_{A_2}/t_{A_1} is around 5/1, while t_{A_3}/t_{A_2} is about 1000/1. These speed differences require (M_1,M_2) to be managed by high-speed hardware circuits rather than by software routines; (M_2,M_3), on the other hand, is controlled mainly by the operating system. Thus while the (M_2,M_3) hierarchy is transparent to the application programmer but visible to the system programmer, (M_1,M_2) is largely transparent to both. Another difference lies in the block size used. Communication within (M_1,M_2) is by pages,

Two-level hierarchy (M_{i-1}, M_i)	Cache–main memory (M_1, M_2)	Main–secondary memory (M_2, M_3)
Typical access time ratios $t_{A_i}/t_{A_{i-1}}$	5/1	1000/1
Memory management system	Mainly implemented by hardware	Mainly implemented by software
Typical page size	8 B	4 KB
Access of processor to second level M_i	Processor has direct access to M_2	All access to M_3 is via M_2

Figure 6.39
Major differences between cache–main and main–secondary-memory hierarchies.

but the page size is much smaller than that used in (M_2,M_3). Finally, we note that the CPU generally has direct access to both M_1 and M_2, whereas it does not have direct access to M_3.

Cache organization. Figure 6.40 shows the principal components of a cache. Memory words are stored in a *cache data memory* and are grouped into small pages called *cache blocks* or *lines*. The contents of the cache's data memory are thus copies of a set of main-memory blocks. Each cache block is marked with its block address, referred to as a *tag*, so the cache knows to what part of the memory space the block belongs. The collection of tag addresses currently assigned to the cache, which can be noncontinguous, is stored in a special memory, the *cache tag memory* or *directory*. For example, if block B_j containing data entries D_j is assigned to M_1, then B_j is in the cache's tag memory and D_j is in the cache's data memory.

Obviously for a cache to improve the performance of a computer, the time required to check tag addresses and access the cache's data memory must be less than the time required to access main memory. Thus if main memory is implemented with a DRAM technology having an access time $t_{A_2} = 50$ ns, the cache's data memory might be implemented with an SRAM technology having an access time of $t_{A_1} = 10$ ns. A basic issue in cache design, which we examine in section 6.3.2, is how to make the matching of tag addresses extremely fast.

Two general ways of introducing a cache into a computer appear in Figure 6.41. In the *look-aside* design of Figure 6.41*a*, the cache and the main memory are directly connected to the system bus. In this design the CPU initiates a memory access by placing a (real) address A_i on the memory address bus at the start of a

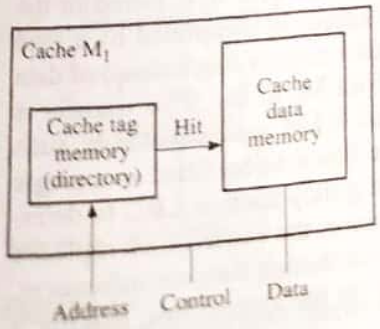


Figure 6.40
Basic structure of a cache.

computations. Let N_1^* and N_2^* denote the number of references to M_1 and M_2, respectively, in the block address stream. The *block hit ratio* H^* is defined by

$$H^* = \frac{N_1^*}{N_1^* + N_2^*}$$

which is analogous to the (word) hit ratio H defined by Equation (6.5). Let n^* denote the average number of consecutive word address references within each block. H can be estimated from H^* using the following relation:

$$H = 1 - \frac{1 - H^*}{n^*}$$

In a paging system, H^* is the page-hit ratio. $1 - H^*$, the page-miss ratio, is also called the *page fault probability*.

EXAMPLE 6.6 COMPARISON OF SEVERAL REPLACEMENT POLICIES. Consider a paging system in which M_1 has a capacity of three pages. The execution of a program Q requires reference to five distinct pages P_i, where $i = 1, 2, 3, 4, 5$, and i is the page address. The page address stream formed by executing Q is

2 3 2 1 5 2 4 5 3 2 5 2

which means that the first page referenced is P_2, the second is P_3, and so on. Figure 6.36 shows the manner in which the pages are assigned to M_1 using FIFO, LRU, and the ideal OPT replacement policies. The next block to be selected for replacement is marked by an asterisk in the FIFO and LRU cases. It will be observed that LRU recognizes that P_2 and P_5 are referenced more frequently than other pages, whereas FIFO

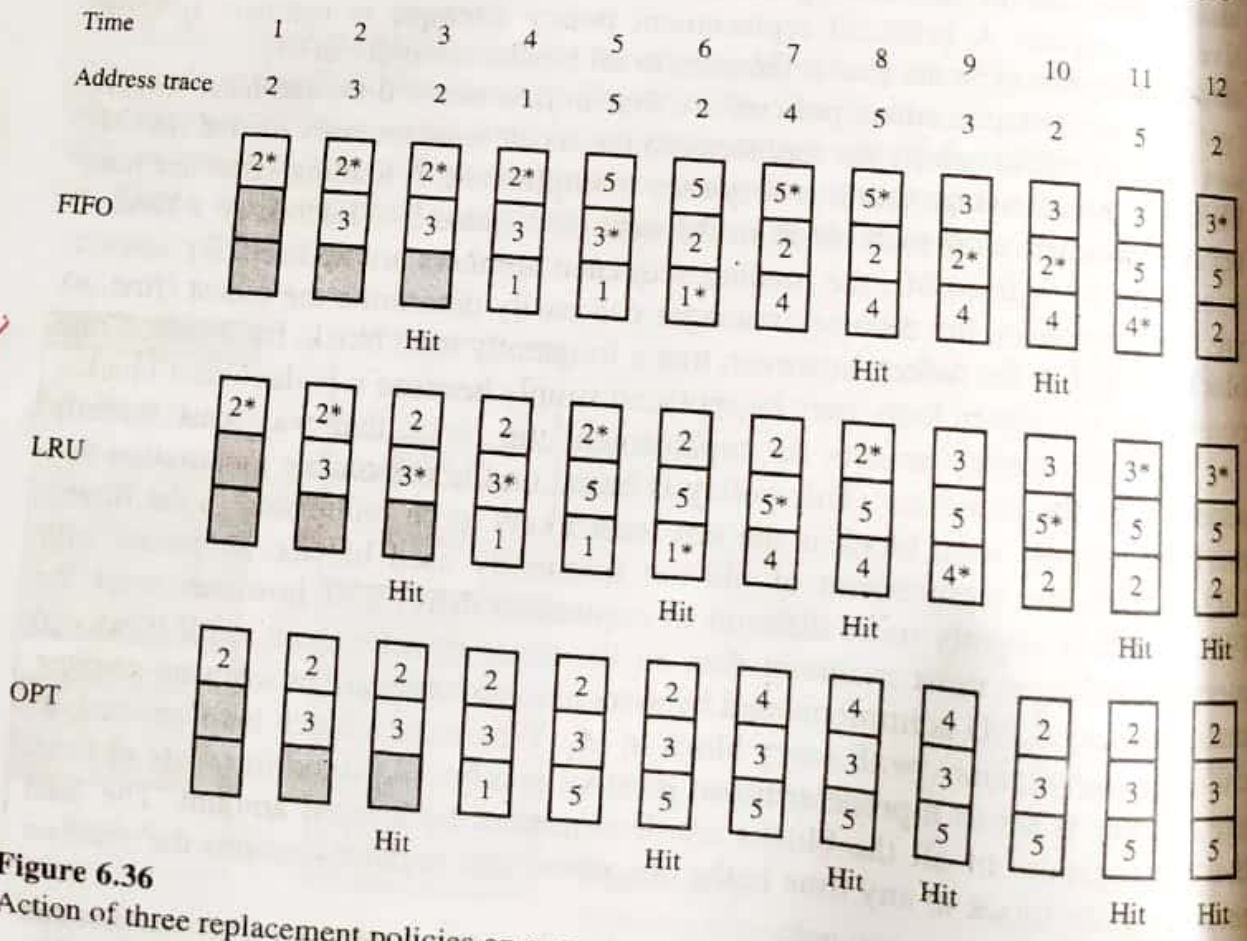


Figure 6.36
Action of three replacement policies on a common address trace.

is viewed as having a single available region; new available regions due to freed blocks are ignored. When the gap at the end of the memory is eventually filled, compaction is carried out again. The advantages of this scheme are its simplicity and the fact that it eliminates the task of selecting an available region; its drawback is the long compaction time.

Replacement policies. The second major approach to preemptive allocation involves preempting a region R occupied by block K and allocating it to an incoming block K'. The criteria for selecting K as the block to be replaced constitute the replacement policy. The main goal in choosing a replacement policy is to maximize the hit ratio of the faster memory M_1 or, equivalently, minimize the number of times a referenced block is not in M_1, a condition called a *memory fault* or *miss*.

It is generally accepted that the hit ratio tends to a maximum if the time intervals between successive memory faults are maximized. An *optimal replacement strategy* would therefore at time t_i determine the time $t_j > t_i$ at which the next reference to block K is to occur; the K to be replaced is the one for which $t_j - t_i$ has the maximum value t_K. This ideal strategy has been called OPT [Mattson et al. 1970; Stone 1993]. In principle, OPT can be implemented by making two passes through the executing program. The first is a simulation run to determine the sequence S_B of distinct virtual block addresses generated by the program; the sequence is called the *block address trace*. The values of t_K at each point in time can be computed from S_B and used to construct the optimal sequence S_B^{OPT} of blocks to be replaced. The second run is the execution run, which uses S_B^{OPT} to specify the blocks to be replaced. OPT is not a practical replacement policy because of the cost of the simulation runs and the fact that S_B can be extremely long, making S_B^{OPT} too expensive to compute. A practical replacement policy attempts to estimate t_K using statistics it gathers on the past references to all blocks currently in M_1.

Two useful replacement policies are *first-in first-out* (FIFO) and *least recently used* (LRU). FIFO selects for replacement the block least recently loaded into M_1. FIFO has the advantage that it is very easy to implement. A loading-sequence number is associated with each block in the occupied space list. Each time a block is transferred to or from M_1, the loading-sequence numbers are updated. By inspecting these numbers, the memory manager can easily determine the oldest (first-in) block. FIFO has the defect, however, that a frequently used block, for instance, one containing a program loop, may be replaced simply because it is the oldest block.

The LRU policy selects for replacement the block that was least recently accessed by the processor. This policy is based on the reasonable assumption that the least recently used block is the one least likely to be referenced in the future. LRU avoids the replacement of old but frequently used blocks, as occurs with FIFO. LRU is slightly more difficult to implement than FIFO, however, since the memory manager must maintain data on the times of references to all blocks in main memory. LRU is implemented by associating a hardware or software counter, called an *age register*, with every block in M_1. Whenever a block is referenced, its age register is set to a predetermined positive number. At fixed intervals of time, the age registers of all the blocks are decremented by a fixed amount. The least recently used block at any time is the one whose age register contains the smallest number.

The performance of a replacement policy in a given memory organization can be [illegible] the block address stream generated by a set of representative

Page size. The page size S_p has a big impact on both storage utilization and the effective memory data-transfer rate. Consider first the influence of S_p on the space-utilization factor u defined earlier. If S_p is too large, excessive internal fragmentation results; if it is too small, the page tables become very large and tend to reduce space utilization. A good value of S_p should achieve a balance between these two extremes. Let S_s denote the average segment size in words. If $S_s >> S_p$, the last page assigned to a segment should contains about $S_p/2$ words. The size of the page table associated with each segment is approximately S_s/S_p words, assuming each entry in the table is a word. Hence the memory space overhead associated with each segment is

$$S = \frac{S_p}{2} + \frac{S_s}{S_p}$$

The space utilization u is

$$u = \frac{S_s}{S_s + S} = \frac{2S_sS_p}{S_p^2 + 2S_s(1 + S_p)} \tag{6.10}$$

The optimum page size S_p^{OPT} can be defined as the value of S_p that maximizes u or, equivalently, that minimizes S. Differentiating S with respect to S_p, we obtain

$$\frac{dS}{dS_p} = \frac{1}{2} - \frac{S_s}{S_p^2}$$

S is a minimum when $dS/dS_p = 0$, from which it follows that

$$S_p^{OPT} = \sqrt{2S_s} \tag{6.11}$$

The optimum space utilization is

$$u^{OPT} = \frac{1}{1 + \sqrt{2/S_s}}$$

Figure 6.32 shows the space utilization u defined by Equation (6.10) plotted against S_s for some representative values of S_p.

The influence of page size on hit ratio is complex, depending on the program reference stream and the amount of space available in M_1. Let the virtual address space of a program be a sequence of numbers $A_0, A_1, \ldots, A_{L-1}$. Let A_i be the virtual address referenced at some point in time, and let A_{i+d} be the next address generated, where d is the "distance" between A_i and A_{i+d}. For example, if both addresses point to instructions, A_{i+d} points to the $(d + 1)$st instruction either preceding or following the instruction whose virtual address is A_i. Let S_p be the page size and suppose that an efficient replacement policy such as LRU is being used. The probability of A_{i+d} being in M_1 is high if one of the following conditions is satisfied:

- d is small compared with S_p, so A_i and A_{i+d} are in the same page P. The probability of these addresses both being in P increases with the page size.
- d is large relative to S_p but A_{i+d} is associated with a set of words that are frequently referenced. A_{i+d} is therefore likely to be in a page $P' \neq P$, which is also

fewer entries. The relative efficiency of the two techniques has long been a subject of debate, since both have been implemented with satisfactory results [Knuth 1973; Shore 1975]. The performance obtained in a particular environment depends on the distribution of the block sizes to be allocated. Simulation studies suggest that, in practice, first fit tends to outperform best fit.

Preemptive allocation. Nonpreemptive allocation cannot make efficient use of memory in all situations. *Memory overflow*, that is, rejection of a memory allocation request due to insufficient space, can be expected to occur with M_1 only partially full. Much more efficient use of the available memory space is possible if the occupied space can be reallocated to make room for incoming blocks. Reallocation may be done in two ways:

- The blocks already in M_1 can be relocated within M_1 to create a gap large enough for the incoming block.
- One or more occupied regions can be made available by deallocating the blocks they contain. This method requires a rule—a *replacement policy*—for selecting blocks to be deallocated and replaced.

Deallocation requires that a distinction be made between "dirty" blocks, which have been modified since being loaded into M_1, and "clean" blocks, which have not been modified. Blocks of instructions remain clean, whereas blocks of data can become dirty. To replace a clean block, the memory management system can simply overwrite it with the new block and update its entry in the memory map. Before a dirty block is overwritten, it should be copied to M_2, which involves a slow block transfer.

Relocation of the blocks already occupying M_1 can be done by a method called *compaction*, which is illustrated in Figure 6.35. The blocks currently in memory are compressed into a single contiguous group at one end of the memory. This creates an available region of maximum size. Once the memory is compacted, incoming blocks are assigned to contiguous regions at the unoccupied end. The memory

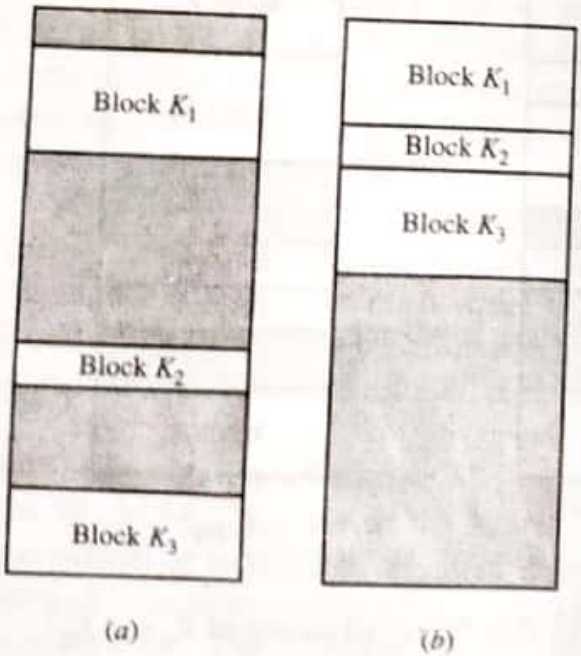
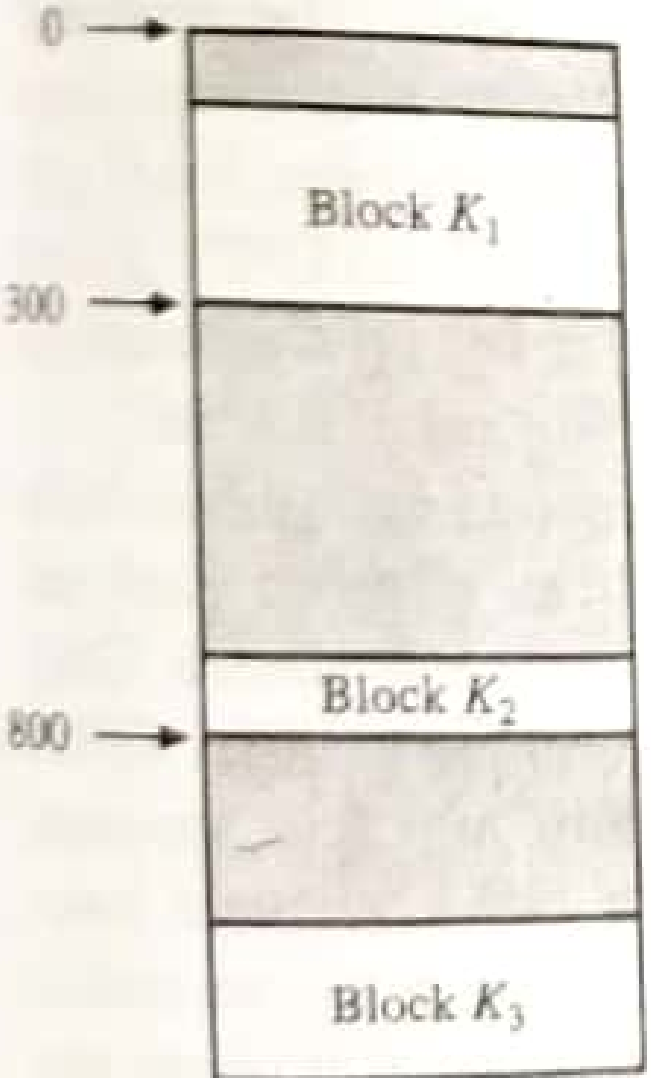


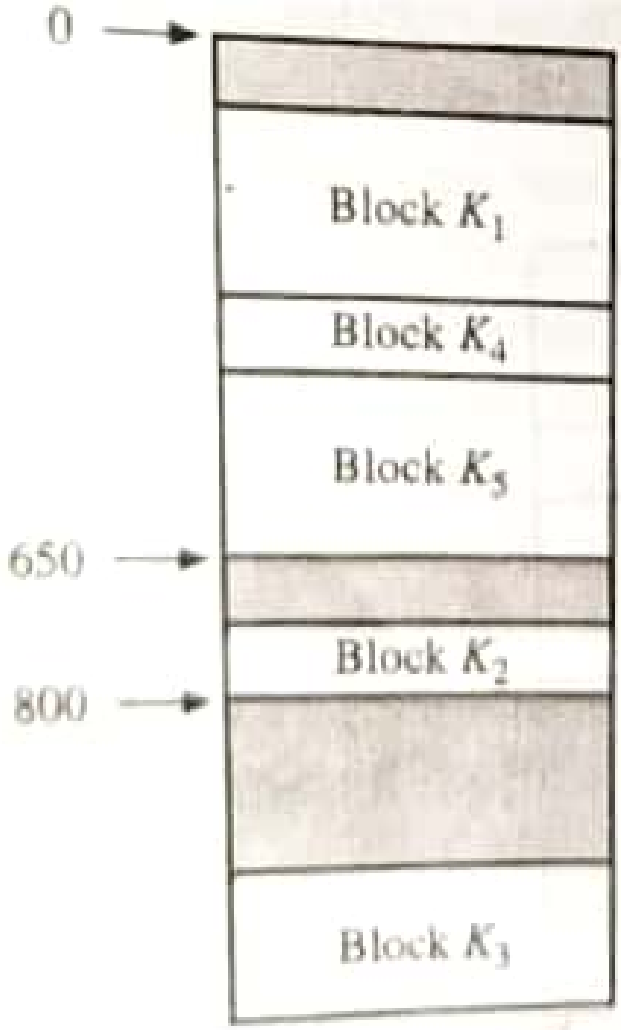
Figure 6.35
Memory allocation (*a*) before and (*b*) after compaction.

Region address	Size (words)
0	50
300	400
800	200

Further, suppose that two new blocks K_4 and K_5 words, respectively, are to be assigned to M_1. Figur obtained using the first-fit and best-fit methods, re scan begins at address 0.

The first-fit algorithm has the advantage of nee the best-fit approach. If the best-fitting available reg





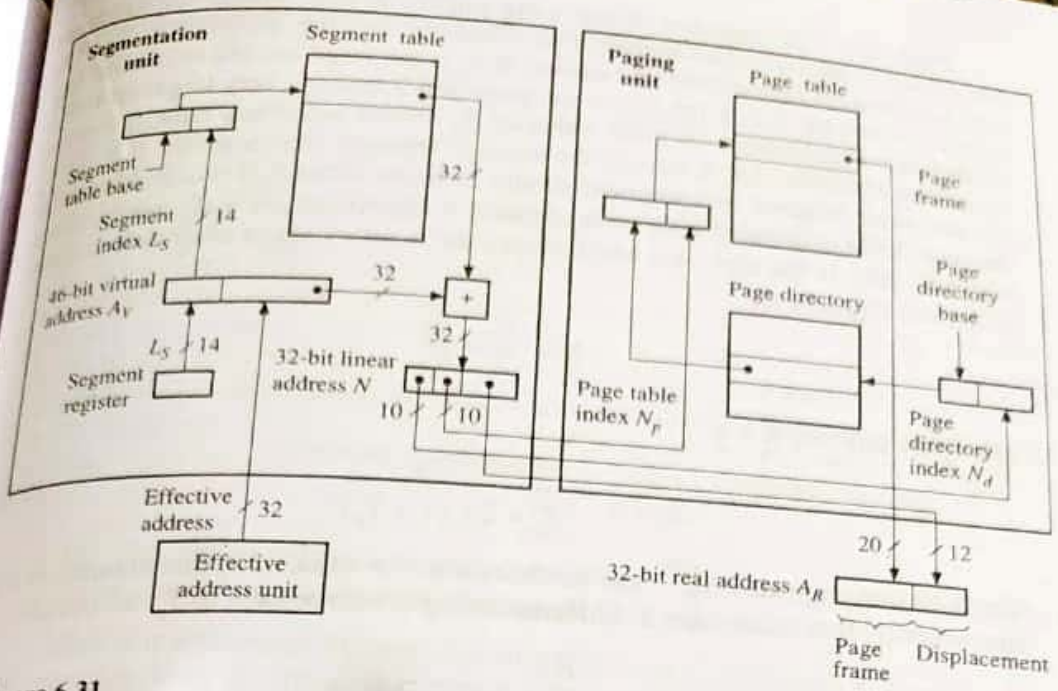


Figure 6.31
Address translation with segmentation and paging in the Intel Pentium.

address organization of main memory. The CPU contains six segment registers that store pointers to the segments in current use. For example, the segment registers CS and SS address a code (program) and stack segment, respectively. These registers are typically used in a manner that is transparent to the application programmer. For instance, when an instruction fetch is initiated, a 32-bit (effective) address obtained from the program counter PC is appended to a 14-bit segment index L_s obtained from the CS register to form a 46-bit virtual address L. As Figure 6.31 indicates, L_s serves as a relative address for an 8-byte segment descriptor stored in one of many possible segment tables. The descriptor specifies the base address and length of the segment S referred to by L_s. It also indicates S's type and access rights, and whether S is present in main memory. The linear address N is constructed by adding the base address obtained from the segment descriptor to the program-derived effective address.

Figure 6.31 also shows how the paging unit processes the linear address N to produce a real address A_R, assuming a page size of 4 KB. A two-step table lookup process is employed to obtain A_R from N. The right-most 12 bits of N form a displacement within the page containing the desired information; they therefore supply the right-most 12 bits of A_R. The remaining 20 bits of N yield a real page address as follows. First a page directory is accessed, which contains entries defining up to 1024 page tables. The left-most 10 bits N_d of N form the relative address of a 32-bit entry E in the page table directory. E contains the 20-bit base address of a page table T, as well as such standard information as a presence bit, a change bit (indicating whether or not the page has been written into), and some protection information. Using the base address derived from E, the page table T is then accessed, and the word E', which is stored at the relative address pointed to by the 10-bit field N_p of the linear address N, is fetched. E', which has the same format as E, provides the 20-bit page address (page frame number) of the desired real address A_R.

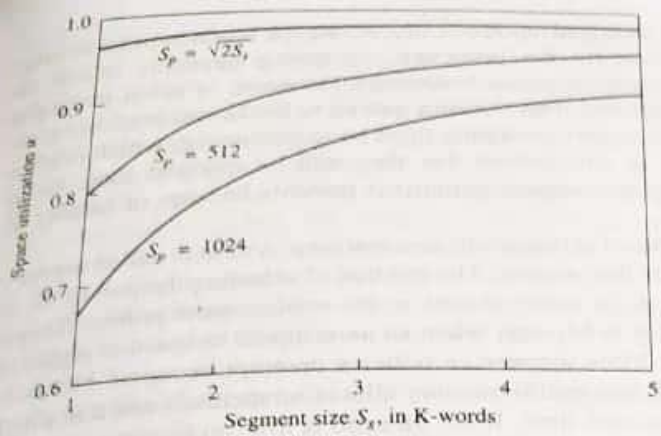


Figure 6.32
Influence of page size S_p and segment size S_s on space utilization u.

in M_1. This likelihood tends to increase with the number of pages stored in M_1; it therefore tends to decrease with the size of S_p.

Thus H is influenced by two opposing forces as S_p is varied. When S_p is small, H increases with S_p. However, when S_p exceeds a certain value, H begins to decrease. Figure 6.33 shows some typical curves relating H and S_p for various main-memory capacities. Simulation studies indicate that in large systems, the values of S_p yielding the maximum hit ratios can be greater than the "optimum" page size given by Equation (6.11). Since high H is important in achieving small t_A (due to the relatively slow rates at which page swapping takes place), values of S_p that maximize H are preferred. The first computer with a paging system (the University of Manchester's Atlas computer) had a 512-word page, while the Pentium discussed in Example 6.5 supports page sizes of 4 KB and 4 MB.

5.6.1 Memory Allocation

As we have seen, the various levels of a memory system are divided into sets of contiguous locations, variously called regions, segments, or pages, which store

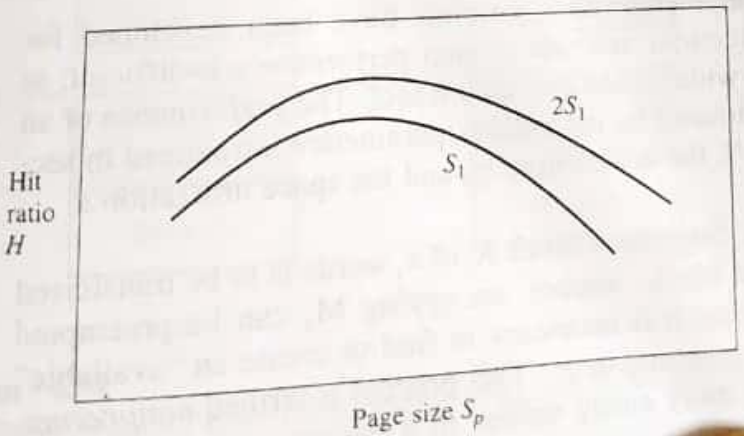


Figure 6.33
Influence of page size S_p on hit ratio H.

Segments. The basic unit of information for swapping purposes in a multilevel memory is a fixed-size block called a page. Pages are allocated to page-sized storage regions (*page frames*), whose fixed size and address formats make paging systems easy to implement. Pages are convenient blocks for the *physical* partitioning and swapping of the information stored in a multilevel memory. It is often desirable to have higher-level information blocks, termed *segments*, that correspond to *logical* entities such as programs or data sets. Segments facilitate the mapping of individual programs, as well as the assignment and checking of different storage properties. For example, write operations may not be permitted into certain regions of the virtual address space in order to protect critical items. It is easier to protect the information in question by making it a read-only segment S, rather than assigning access restrictions to the possibly large number of pages that compose S.

Formally, a *segment* is a set of logically related, contiguous words; it is therefore a special type of block in the sense used in section 6.2.1. A word in a segment is referred to by specifying a base address—the *segment address*—and a displacement within the segment. A program and its data can be viewed as a collection of linked segments. The links arise from the fact that a program segment uses, or calls, other segments. Some computers have a memory management technique that allocates main memory by M_1 segments alone. When a segment not currently resident in M_1 is required, the entire segment is transferred from secondary memory M_2. The physical addresses assigned to the segments are kept in a memory map called a *segment table* (which can itself be a relocatable segment).

Segmentation was implemented in this general form in the Burroughs B6500/7500 series [Hauck and Dent 1968]. Each program has a segment called its program reference table (PRT), which serves as its segment table. All segments associated with the program are defined by special words called *segment descriptors* in the corresponding PRT. As shown in Figure 6.28, a B6500/7500 segment descriptor contains the following information:

- A presence bit P that indicates whether the segment is currently assigned to M_1.
- A copy bit C that specifies whether this is the original (master) copy of the descriptor.
- A 20-bit size field Z that specifies the number of words in the segment.
- A 20-bit address field S that is the segment's real address in M_1 (when $P = 1$) or M_2 (when $P = 0$).

A program refers to a word within a segment by specifying the segment descriptor word W in its PRT and the displacement D. The CPU fetches and examines W. If the presence bit $P = 0$, an interrupt occurs and execution of the requesting program

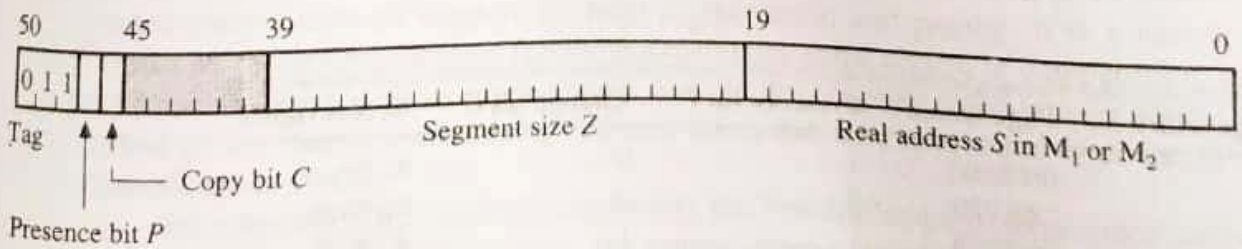


Figure 6.28
Segment descriptor of the Burroughs B6500/7500.

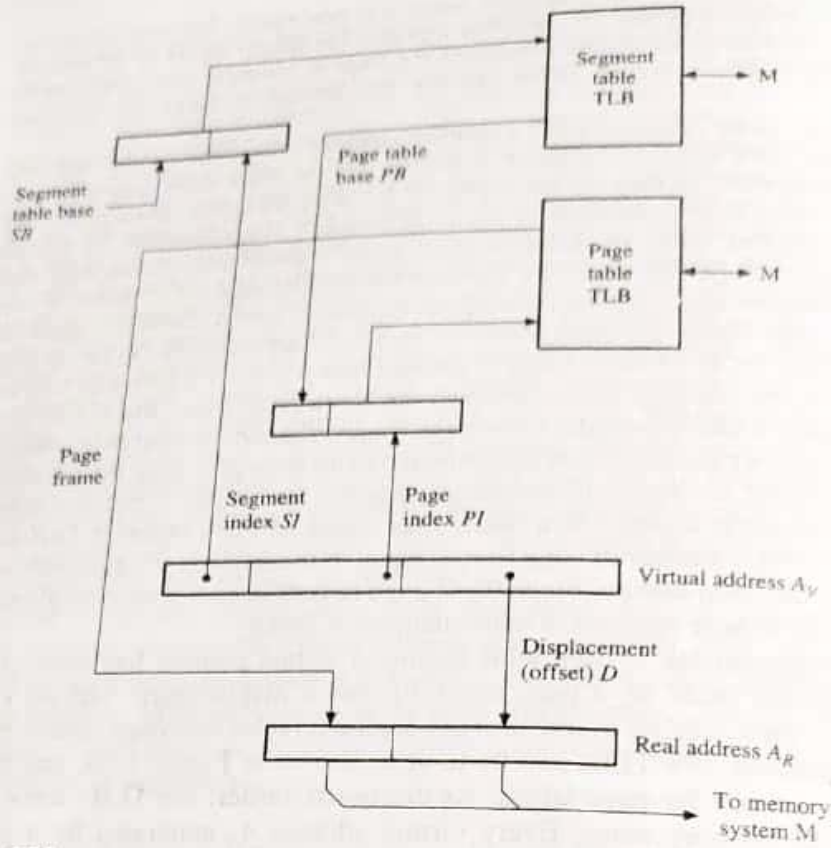


Figure 6.30
Two-stage address translation with segments and pages.

following four memory access methods can be selected under program control: unsegmented and unpaged, segmented and unpaged, unsegmented and paged, and segmented and paged. The output of the paging unit is a 32-bit real address, while that of the segmentation unit is a 32-bit word called a *linear* address. If both segmentation and paging are used, every memory address generated by a program goes through a two-stage translation process

$$\text{Virtual address } A_V \rightarrow \text{linear address } N \rightarrow \text{real address } A_R$$

as depicted in Figure 6.31. Without segmentation $A_V = N$, while without paging $N = A_R$. The segmentation and paging units both contain TLBs to store the active portions of the various memory maps needed for address translation, so the delay of the translation process is small. This delay is further diminished by overlapping (pipelining) the formation of the virtual, linear, and real addresses, as well as by overlapping memory addressing and fetching, so the next real address is ready by the time the current memory cycle is completed.

An active process controlled by the Pentium has several segments associated with it, such as the object program code, a program control stack, and one or more data sets. Each segment can be thought of as a virtual memory of size 4 GB, which has the linear

system-specific control information—a segment address, for example—as we will see later. The real address $B_R = f(B_V)$ assigned to B_V is stored in a memory map somewhere in the memory system; this map can be quite large. To speed up the mapping process, part (or occasionally all) of the memory map is placed in a small high-speed memory in the CPU called a *translation look-aside buffer* (*TLB*). The TLB's input is thus the base-address part B_V of A_V; its output is the corresponding real base address B_R. This address is then concatenated with the D part of A_V to obtain the full physical address A_R.

If the virtual address B_V is not currently assigned to the TLB, then the part of the memory map that contains B_V is first transferred from the external memory into the TLB. Hence the TLB itself forms a cachelike level within a multilevel address-storage system for memory maps. For this reason, the TLB is sometimes referred to as an *address cache*.

EXAMPLE 6.4 MEMORY ADDRESS TRANSLATION IN THE MIPS R2/3000 [KANE 1988]. The MIPS R2/3000 microprocessor, whose main features were introduced earlier (Examples 3.5 and 3.7), employs an on-chip MMU. The MMU's primary function is to map 32-bit virtual addresses to 32-bit real addresses. (Later members of the RX000 family like the R10000 support 64-bit addresses.) A 32-bit address allows the R2/3000 to have a virtual address space of 2^{32} bytes, or 4 GB. Both address spaces are composed of 4KB *pages*, which are convenient block sizes for information transfer within a conventional memory hierarchy comprising a cache (of the split kind), main memory, and secondary memory. The 4GB virtual-address space is further partitioned into four parts called *segments*, three of which form the system region (or "kernel region" in MIPS parlance) devoted to operating system functions, while the other is the user region, where application programs, data, and control stacks are stored.

The format of an R2/3000 virtual address appears in Figure 6.27. It consists of a 20-bit virtual page address, referred to as the virtual page name *VPN*, and a 12-bit displacement *D*, which specifies the address of a byte within the virtual page. The high-order 3 bits 31:29 of *VPN* form a type of tag that identifies the segment being addressed. Bit 31 of *VPN* is 0 for a user segment and 1 for a supervisor segment; it thus distinguishes the user and supervisor (privileged) control states of the CPU. The user segment is *kuseg* and occupies half the virtual address space. The supervisor region is divided into three segments, *kseg*0, *kseg*1, and *kseg*2, each of which has different access characteristics.

- *kuseg*: This 2GB segment is designed to store all user code and data. Addresses in this region make full use of the cache and are mapped to real addresses via the TLB.
- *kseg*0: This 512MB system segment is cached and unmapped; that is, virtual addresses within *kseg*0 are mapped directly into the first 512 MB of the real address space, which includes the cache, but no virtual address translation takes place. This segment typically stores active parts of the operating system.
- *kseg*1: This is also a 512MB segment, but is both uncached and unmapped. It is intended for such purposes as storing boot-up code (which cannot be cached) and for other instructions and data—high-speed IO data, for instance—that might seriously slow down cache operation.
- *kseg*2: This is a 1GB segment which, like *kuseg*, is both cached and mapped.

The MMU contains a TLB to provide fast virtual-to-real address translation. The TLB stores a 64-entry portion of the memory map (page table) assigned to each process by the operating system. The current virtual page address *VPN* is used to access a 64-bit entry in the TLB, which, as shown in Figure 6.27, contains among other items, a 20-

Page size. The page size S_p has a big impact on both storage utilization and the effective memory data-transfer rate. Consider first the influence of S_p on the space-utilization factor u defined earlier. If S_p is too large, excessive internal fragmentation results; if it is too small, the page tables become very large and tend to reduce space utilization. A good value of S_p should achieve a balance between these two extremes. Let S_s denote the average segment size in words. If $S_p \gg S_s$, the last page assigned to a segment should contains about $S_p/2$ words. The size of the page table associated with each segment is approximately S_s/S_p words, assuming each entry in the table is a word. Hence the memory space overhead associated with each segment is

$$S = \frac{S_p}{2} + \frac{S_s}{S_p}$$

The space utilization u is

$$u = \frac{S_s}{S_s + S} = \frac{2S_sS_p}{S_p^2 + 2S_s(1 + S_p)} \tag{6.10}$$

The optimum page size S_p^{OPT} can be defined as the value of S_p that maximizes u or, equivalently, that minimizes S. Differentiating S with respect to S_p, we obtain

$$\frac{dS}{dS_p} = \frac{1}{2} - \frac{S_s}{S_p^2}$$

S is a minimum when $dS/dS_p = 0$, from which it follows that

$$S_p^{OPT} = \sqrt{2S_s} \tag{6.11}$$

The optimum space utilization is

$$u^{OPT} = \frac{1}{1 + \sqrt{2/S_s}}$$

Figure 6.32 shows the space utilization u defined by Equation (6.10) plotted against S_s for some representative values of S_p.

The influence of page size on hit ratio is complex, depending on the program reference stream and the amount of space available in M_1. Let the virtual address space of a program be a sequence of numbers $A_0, A_1, \ldots, A_{L-1}$. Let A_i be the virtual address referenced at some point in time, and let A_{i+d} be the next address generated, where d is the "distance" between A_i and A_{i+d}. For example, if both addresses point to instructions, A_{i+d} points to the $(d + 1)$st instruction either preceding or following the instruction whose virtual address is A_i. Let S_p be the page size and suppose that an efficient replacement policy such as LRU is being used. The probability of A_{i+d} being in M_1 is high if one of the following conditions is satisfied:

- d is small compared with S_p, so A_i and A_{i+d} are in the same page P. The probability of these addresses both being in P increases with the page size.
- d is large relative to S_p but A_{i+d} is associated with a set of words that are frequently referenced. A_{i+d} is therefore likely to be in a page $P' \neq P$, which is also

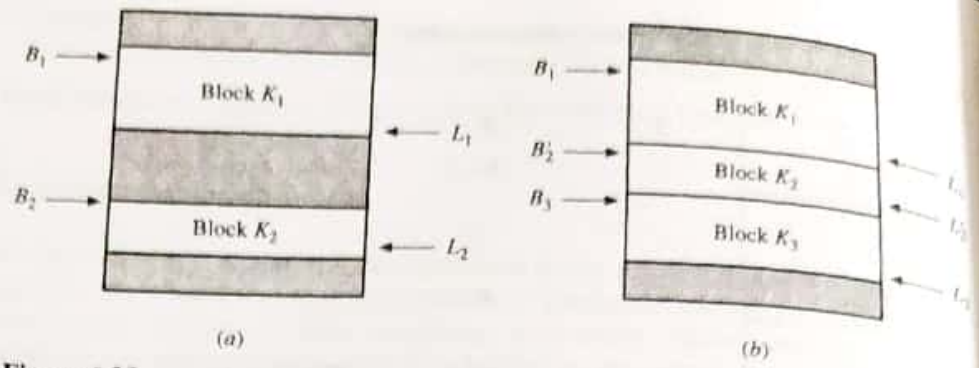


Figure 6.25
Relocation of blocks in memory using base and limit addresses.

stored in the memory map. Every real address A_r generated by the block is compared to B_i and L_i; the memory access is completed if and only if the condition

$$B_i \leq A_r \leq L_i$$

is satisfied.

Translation look-aside buffer. Figure 6.26 shows how various parts of a multilevel memory management typically realize the address-translation ideas just discussed. The input address A_V is a virtual address consisting of a (virtual) base address B_V concatenated with a displacement D. A_V contains an effective address computed in accordance with some program-defined addressing mode (direct, indirect, indexed, and so on) for the memory item being accessed. It also can contain

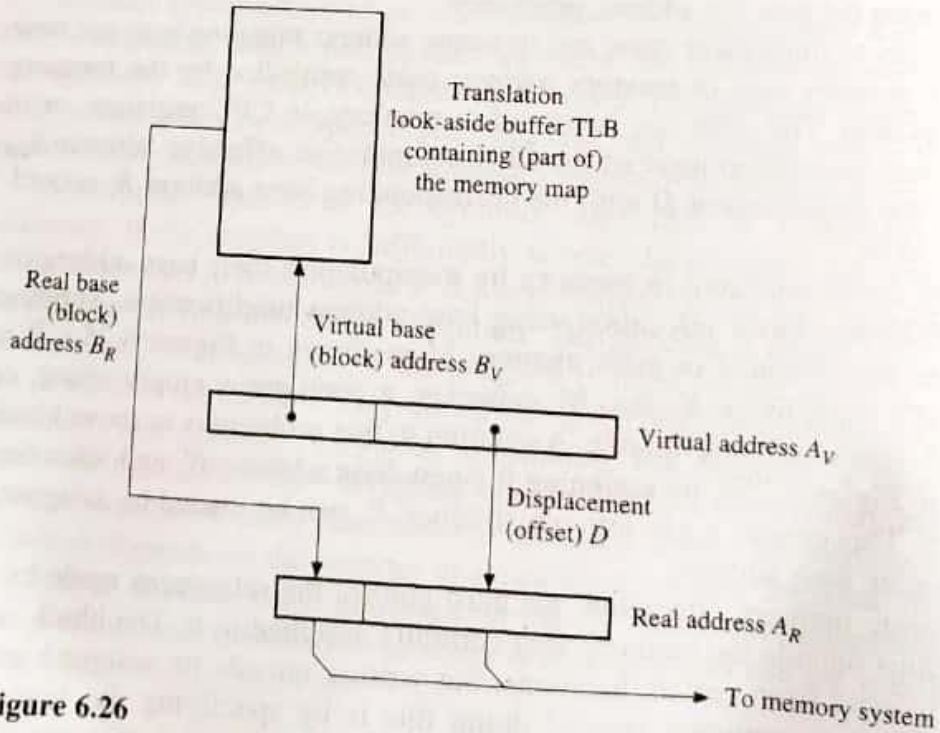


Figure 6.26
Structure of a dynamic address-translation system.

bit page frame number *PFN*. This real page address is fetched from the TLB and appended to the displacement *D* to obtain the desired 32-bit real address. An R2/3000-based system often has less than 4 GB of physical memory, in which case not all the available real address combinations are used.

Observe that the *VPN* itself is also part of the TLB entry because a fast access method called *associative addressing* is used; see section 6.3.2. Another major item stored in each TLB entry is a 6-bit process identification field *PID*. This field distinguishes each active program (process); hence up to 64 processes can share the available virtual page numbers without interference. There are also 4 control bits denoted *NDVG*, which define the types of memory accesses permitted for the corresponding TLB entry. For example, *N* denotes noncachable; when set to 1, it causes the CPU to go directly to main memory, instead of first accessing the cache. *D* is a write-protection (read-only) bit; an attempt to write when $D = 0$ causes a CPU interrupt or trap.

The MMU has some features not shown in Figure 6.27, which are designed to trap error conditions that are collectively referred to as *address translation exceptions*. When a trap occurs, relevant information about the exception is stored in MMU registers, which can be examined and modified by certain privileged instructions. A common address translation exception is a *TLB miss*, which occurs when there is no (valid) entry in the TLB that matches the current *VPN*. The operating system responds to a TLB miss by accessing the current process's page table, which is stored in a known location in *kseg2*, and copying the missing entry to the TLB. Another address-translation exception type is an illegal access—for instance, a write operation addressed to a page with $D = 0$ (read only) in its TLB entry.