CHAPTER 6

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.

Scanned by CamScanner

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

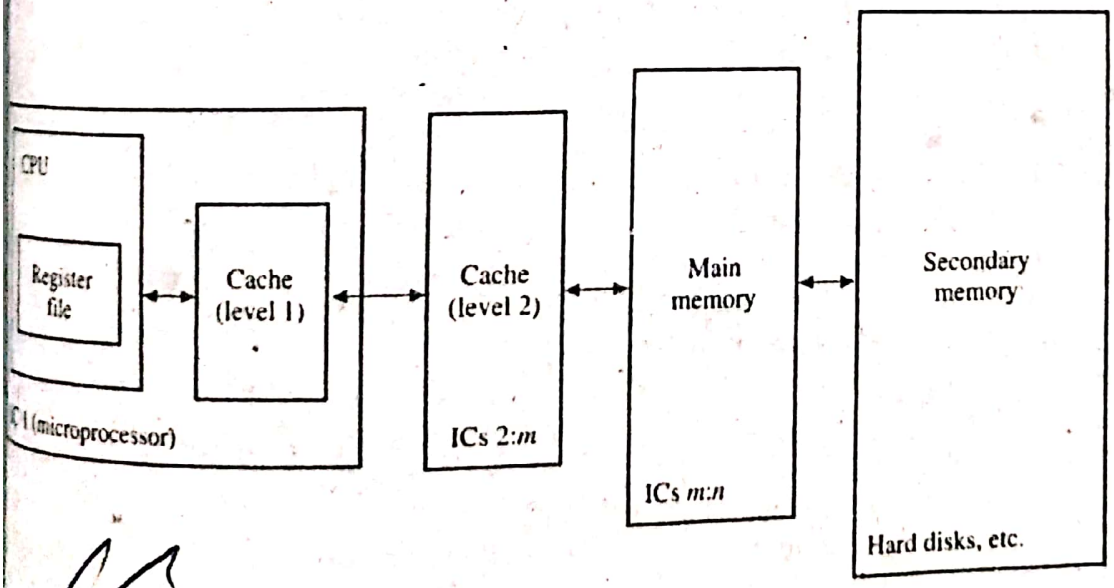


Figure 6.1
Conceptual organization of a multilevel memory system in a computer.

Scanned by CamScanner

that aim to make $CPI \leq 1$. Unlike the three o ther memory types, caches are
mally transparent to the programmer. Toget her, a computer's caches and
memory implement the *external memory* M addressed directly by CPU
tions.

The goal of every memory system is to p ovide adequate storage capacity
an acceptable level of performance and c ost. We can achieve these
employing several memory types—with di fferent cost/performance ratio
are organized to provide a high average perf ormance at a low average cost
The individual memory units form a *multi level hierarchy* of storage
suggested by Figure 6.1. Successful operat ion of the hierarchy requires
storage-control methods that make efficien t use of the available memory
These methods should free the user from explicit management of memory
They should also free programs from the p articular memory environment in
they are executed.

Performance and cost. The comput er architect can choose from a
ing variety of memory devices that emp loy various electronic, magnetic,
cal technologies and offer many cost/ performance trade-offs [Cook and
1994; Prince 1996]. However, all mem ories are based on just a few physical
nomena and organizational principles. We now examine the features
the devices used to build cache, main, and secondary memories.

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

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

The performance of an indi vidual memory device is primarily
the rate at which information ca n be read from or written into the memory.
performance measure is the ave rage time to read a fixed amount of
instance, one word, from the memory. This parameter is called the *read*
time, or simply the *access ti me*, of the memory and is denoted by t_A.
access time is defined similar ly; it is often, but not always, equal to the
time. The access time depend s on the physical nature of the storage
the access mechanisms used. It is calculated from the time the memory
read request to the time at which the requested information becomes
the memory's output terminals.

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

Scanned by CamScanner

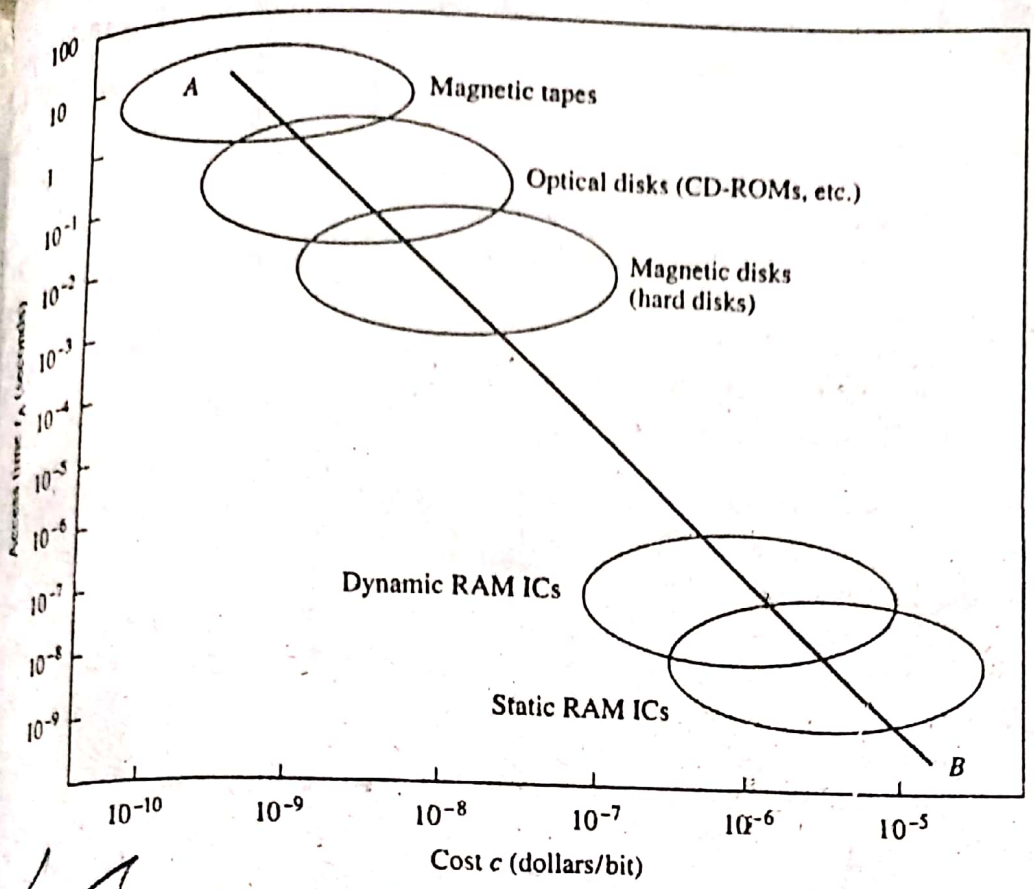


Figure 6.2
Access time versus cost for representative memory technologies.

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.

Scanned by CamScanner

Access modes. A fundamental characteristic of a memory is the order or sequence in which information can be accessed. If storage locations can be accessed in any order and access time is independent of the location being accessed, the memory is termed a *random-access memory* (RAM). IC (semiconductor) memories are generally of this type. Memories whose storage locations can be accessed only in a certain predetermined sequence are called *serial-access memories*. Magnetic disks and tapes, as well as optical memories like CD-ROMs, employ serial-access methods.

Each storage location in a RAM can be accessed independently of the other locations. There is, in effect, a separate access mechanism, or read-write "head", for every location, as suggested in Figure 6.3. In serial memories, on the other hand, the access mechanism is shared by the storage locations and must be assigned to different locations at different times by moving the stored information, the read-write head, or both. Many serial-access memories operate by continually moving the storage locations around a closed path or track, as suggested by Figure 6.4. A particular location can be accessed only when it passes the fixed read-write head. Hence the time to access a particular location depends on its position relative to the read-write head when the memory receives an access request.

Since every location has its own access mechanism, random-access memories tend to be more costly than the serial type. In serial-access memories, however, the time required to bring the desired location into correspondence with a read-write head increases the effective access time, so serial access tends to be slower than random access. Thus the type of access mode contributes significantly to the inverse relationship between cost and access time. In Figure 6.2, for example, the random-access technologies (dynamic and static RAMs based on ICs) and serial-access technologies (magnetic disks, magnetic tapes, and optical disks) are clearly separated into two groups.

Memory devices such as magnetic hard disks and CD-ROMs contain many rotating storage tracks. If each track has its own read-write head, the tracks can be accessed randomly, but access within each track is serial. In such cases the access mode is *semirandom*. Note that the access mode is a function of both memory organization and the inherent characteristics of the storage technology. The IC technologies used for RAMs can also be used to construct serial-access memories; the converse is not true, however.

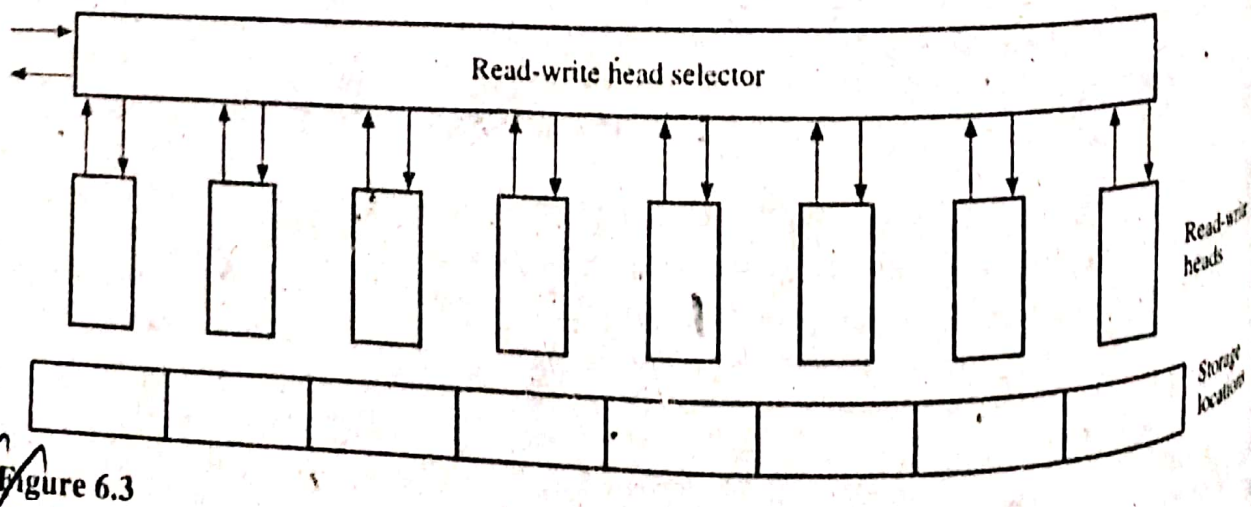


Figure 6.3
Conceptual model of a random-access memory.

Scanned by CamScanner

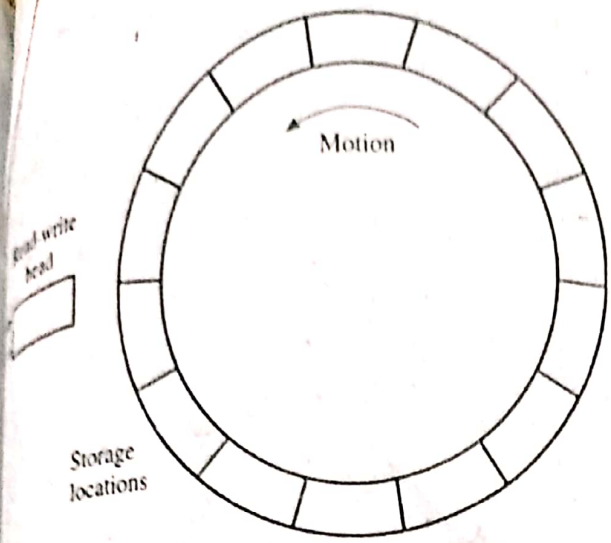


Figure 6.4
Conceptual model of a serial-access memory.

Memory retention. The method of writing information into a memory can be ...ent or irreversible in that once information has been written, it cannot be ...d while the memory is in use or on-line. Printing on paper is an example of a ...ment storage technique. Memories whose contents cannot be altered on-...—if they can be altered at all—are *read-only memories* (ROMs). A ROM is ...efore a nonerasable storage device. ROMs are widely used to store control pro-...s such as microprograms. Compact disk (CD) ROMs are a class of noneras-...e secondary memory devices developed in the 1980s that employ an optical ...r) read-write mechanism. A standard (12 cm diameter) CD-ROM has a capac-...of about 600 MB and is used to store large program and data files. Semiconduc-...ROMs whose contents can be changed off-line—and with some difficulty—are ...ed *programmable read-only memories* (PROMs). Programmable CDs are ...rred to as *CD-recordable* (CD-R) disks.

Memories in which reading or writing can be done with impunity on-line are ...ed *read-write memories* to differentiate them from ROMs. All memories used ...emporary storage purposes are read-write memories. Unless otherwise speci-...d, we will use the terms *memory* and *RAM* to mean read-write memories.

In some technologies the stored information is lost over a period of time unless ...ective action is taken. Three characteristics of memories that destroy informa-...n in this way are destructive readout, dynamic storage, and volatility. In some ...ories the method of reading the memory destroys the stored information; this ...omenon is called *destructive readout* (DRO). Memories in which reading does ...affect the stored data have *nondestructive readout* (NDRO). In DRO memories ...e read operation must be followed by a write operation that restores the mem-...s original state. This restoration is carried out automatically using a buffer reg-...r, as shown in Figure 6.5. The read transfers the word at the addressed (shaded) ...ation to the buffer register where it is available to external devices. The contents ...the buffer are automatically written back into the original location.

Certain memory devices have the property that a stored 1 tends to become a 0, ...vice versa, due to some physical decay process. For example, in some IC memo-...s, an electric charge in a capacitor represents a stored 1; the absence of a stored ...rge represents a 0. Over time, a stored charge tends to leak away, causing a loss

Scanned by CamScanner

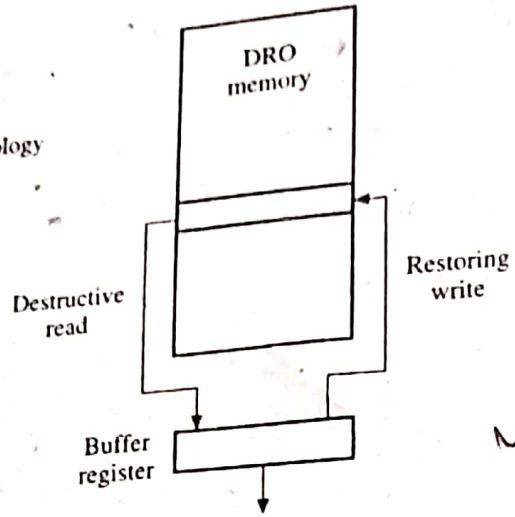


Figure 6.5
Memory restoration in a destructive readout (DRO) memory.

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

Scanned by CamScanner

Technology	Primary storage medium	Access mode	Alterability	Permanence	Typical access time t_A
Bipolar semiconductor	Electronic	Random	Read/write	NDRO, volatile	10 ns
Metal oxide semiconductor (MOS)	Electronic	Random	Read/write	DRO or NDRO, volatile	50 ns
Magnetic (hard) disk	Magnetic	Semirandom	Read/write	NDRO, nonvolatile	10 ms
Magneto-optical disk	Optical	Semirandom	Read/write	NDRO, nonvolatile	50 ms
Compact disk ROM	Optical	Semirandom	Read only	NDRO, nonvolatile	100 ms
Magnetic tape cartridge	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

Scanned by CamScanner

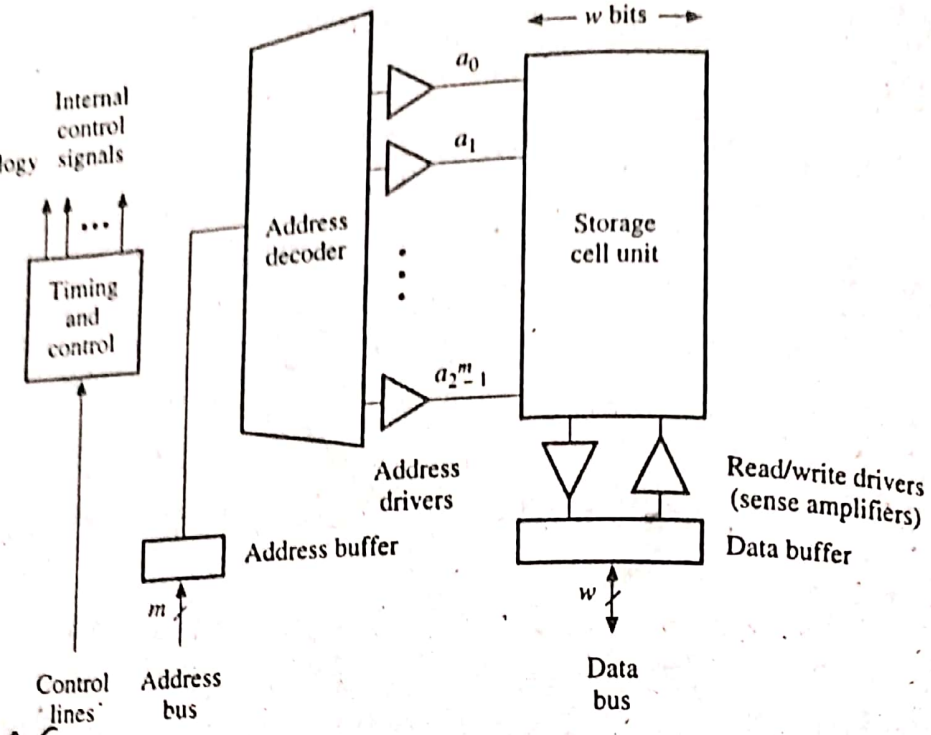


Figure 6.7
One-dimensional (1-D) random-access memory unit.

buffer. The address is then processed by the address decoder, which selects the required location in the storage cell unit. A control line indicates the type of access to be performed. If a read operation (load) is requested, the contents of the addressed location are transferred from the storage cell unit to the data buffer and from there to the data bus. If a write (store) is requested, the word to be stored is transferred from the data bus to the selected location in the storage unit. Since it is not usually necessary or desirable to permit simultaneous reading and writing, the input and output data buses are often combined into a single, bidirectional data bus.

The storage unit is made up of many identical 1-bit memory cells and their interconnections. The actual number of lines connected to the cell and their functions depend on the memory technology and the addressing scheme in use. Each cell is connected to a set of data, address, and control signals. One physical line often has several logical functions; for example, it can serve as both an address and data line. In each line connected to the storage cell unit, we can expect to find a driver that acts as either an amplifier or a transducer of physical signals. Thus we see in Figure 6.7 several sets of drivers for the address and data lines. The drivers, decoders, and control circuits form the *access circuitry* of the RAM and can have a significant impact on the total size and cost of the memory.

A RAM's storage cells are physically arranged into regular arrays to reduce the cost of the connections between the cells and the access circuitry. The memory address is partitioned into d components so that the address A_i of cell C_i becomes a d-dimensional vector $(A_{i,1}, A_{i,2}, \ldots, A_{i,d}) = A_i$. Each of the d parts of the address word goes to a separate address decoder and a separate set of address drivers. A cell is selected by simultaneously activating all d of its address lines. A memory unit with this kind of addressing is said to be *d-dimensional*. Thus the basic RAM of Figure 6.7 is *one-dimensional* (1-D).

Scanned by CamScanner

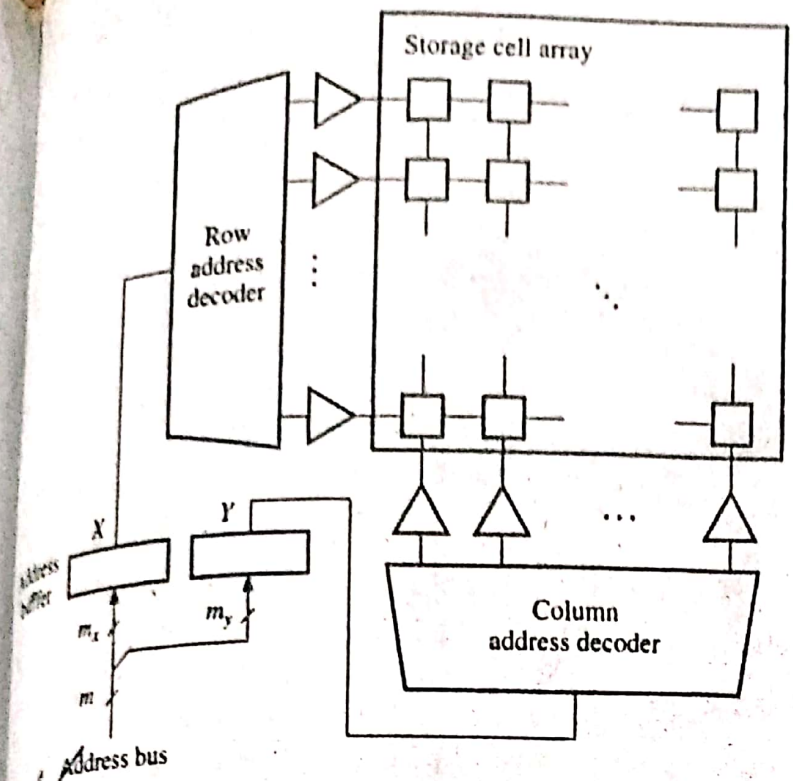


Figure 6.8
Two-dimensional (2-D) RAM addressing scheme.

The most common RAM organization is the *two-dimensional* (2-D) or *row-column* scheme shown in Figure 6.8, where, for simplicity, the data and control circuits are omitted. Here the m-bit address word is divided into two parts, X and Y, consisting of m_x and m_y bits, respectively. The cells are arranged in a rectangular array of $N_x \leq 2^{m_x}$ rows and $N_y \leq 2^{m_y}$ columns, so the total number of cells is $N = N_x N_y$. A cell is selected by the coincidence of signals applied to its X and Y address lines. The 2-D organization requires much less access circuitry than a 1-D organization for the same storage capacity. For example, if $N_x = N_y = \sqrt{N}$, the number of address drivers needed is $2\sqrt{N}$, whereas the 1-D RAM of Figure 6.7 has $N = N_x N_y$ address drivers. Instead of a single one-out-of-N address decoder, two one-out-of-$\sqrt{N}$ address decoders suffice. In addition, the 2-D organization is a good match for the inherently two-dimensional layout structures allowed by VLSI technology.

Semiconductor RAMs. Semiconductor memories in which the storage cells are small transistor circuits have been used for high-speed CPU registers since the 1950s. It was not until the development of VLSI in the 1970s that producing large RAM ICs suitable for main-memory and cache applications became economical. Single-chip RAMs can be manufactured in sizes ranging from a few hundred bits to 1 Gb or more. Both bipolar and MOS transistor circuits are used in RAMs, but MOS is the dominant circuit technology for large RAMs. Current IC manufacturing limitations make it impossible to manufacture, say, a terabit (2^{40}-bit) RAM on a single IC chip. Consequently, very large semiconductor RAMs must be constructed from a set of smaller RAM ICs.

As observed earlier, semiconductor memories fall into two categories—SRAMs and DRAMs—whose data-retention methods are static and dynamic,

Scanned by CamScanner

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

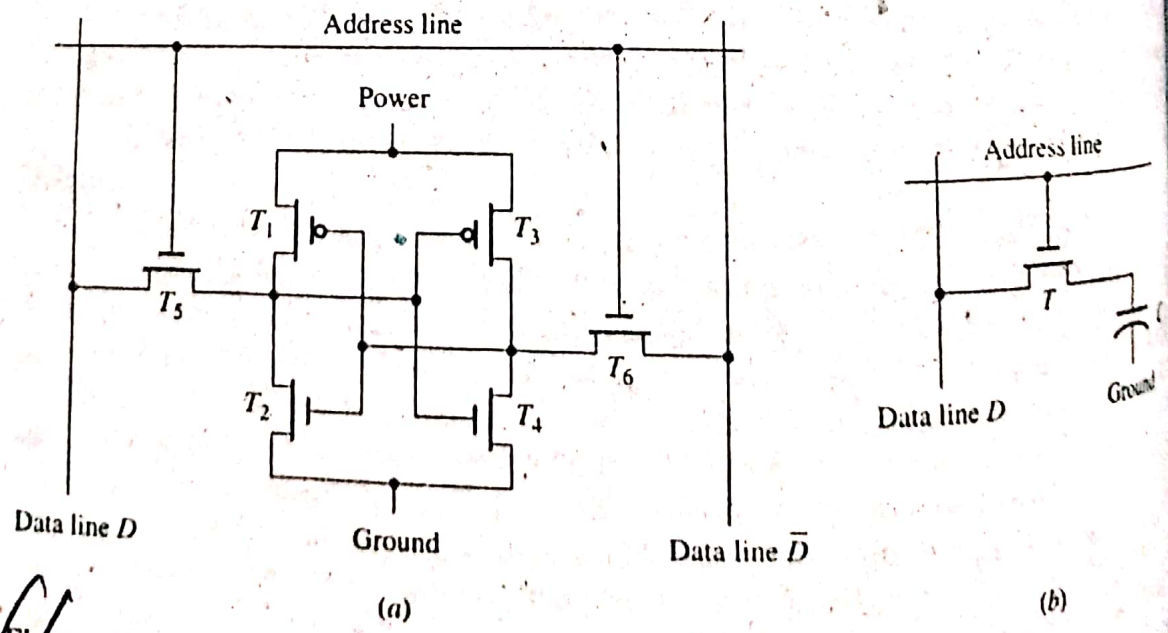


Figure 6.9
(*a*) Static and (*b*) dynamic RAM cells in MOS technology.

Scanned by CamScanner

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

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

A memory-design problem that the computer architect may encounter is the lowing: 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 allest integer greater than or equal to x. In this IC array each row stores N words cept possibly the last row), while each column stores a fixed set of w bits from ery word (except possibly the last column). For example, to construct a 1GB AM using 64M × 1-bit RAM ICs requires $p = 16$, $q = 8$, and a total of $pq = 128$ pies of the 64Mb RAM. When $N' > N$, additional external-address-decoding cir- ry is usually required.

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

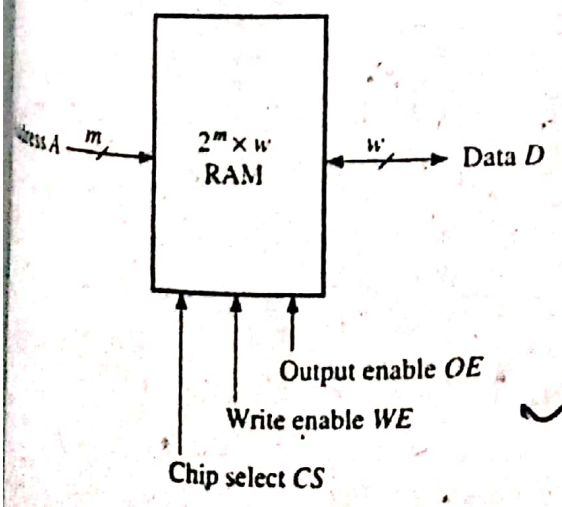


Figure 6.10
A RAM IC showing its major external connections.

Scanned by CamScanner

way to each IC. Their w-bit data buses are concatenated to form a single $4w$-bit bus, as indicated.

Now suppose we want to increase the number of stored words by a factor of four. This time $p = 4$ and $q = 1$, and again we need four RAM ICs. The number of addresses has quadrupled, hence two lines are added to the address bus. Furthermore, a one-out-of-four address decoder must be introduced, as shown in Figure 6.12, to decode the extra address bits. The original m address lines are connected to the m-bit address bus of every RAM IC; the new lines are the main inputs to the decoder. Each decoder output line is connected to the CS inputs of the RAMs in the same row, ensuring that the row has a unique address. The output buses of all RAM ICs in the same column are designed so they can be wired together without additional logic. (This *tristate* busing technique is explained in section 7.1.1.) The remaining control lines WE and OE are attached to every RAM IC as before. The external CS line is connected to an enable input of the decoder. Making this line 0 forces all CS lines to the individual RAM ICs to 0 so that they are all deactivated and no memory operation takes place.

EXAMPLE 6.1 A COMMERCIAL 64Mb DRAM CHIP [MICRON TECHNOLOGY, 1997]. The Micron Technology MT4LC8M8E1, which we will call the 8E1 for short, is a commercial DRAM chip introduced in 1996. It stores 64 Mb, that is, 2^{26} bits of data, in single-transistor storage cells of the kind shown in Figure 6.9*b*. The stored information is organized as 2^{23} 8-bit bytes, so the 8E1 is also referred to as an 8M × 8-bit DRAM. The memory address size $m = 23$, and the data word size $w = 8$.

The internal structure of the 8E1 appears in Figure 6.13. Two-dimensional addressing is employed, with the 23-bit address broken into two parts: a 13-bit row address and a 10-bit column address. Only 13 external address lines are used, allowing the 8E1 to be housed in a small, 32-pin package, which implies that row and column addresses must be multiplexed over the address bus, a common tactic in large RAM chips. This multiplexing is controlled by two lines: $\overline{RAS}$ (row address select) and $\overline{CAS}$ (column address select), which replace the generic CS control line of Figures 6.

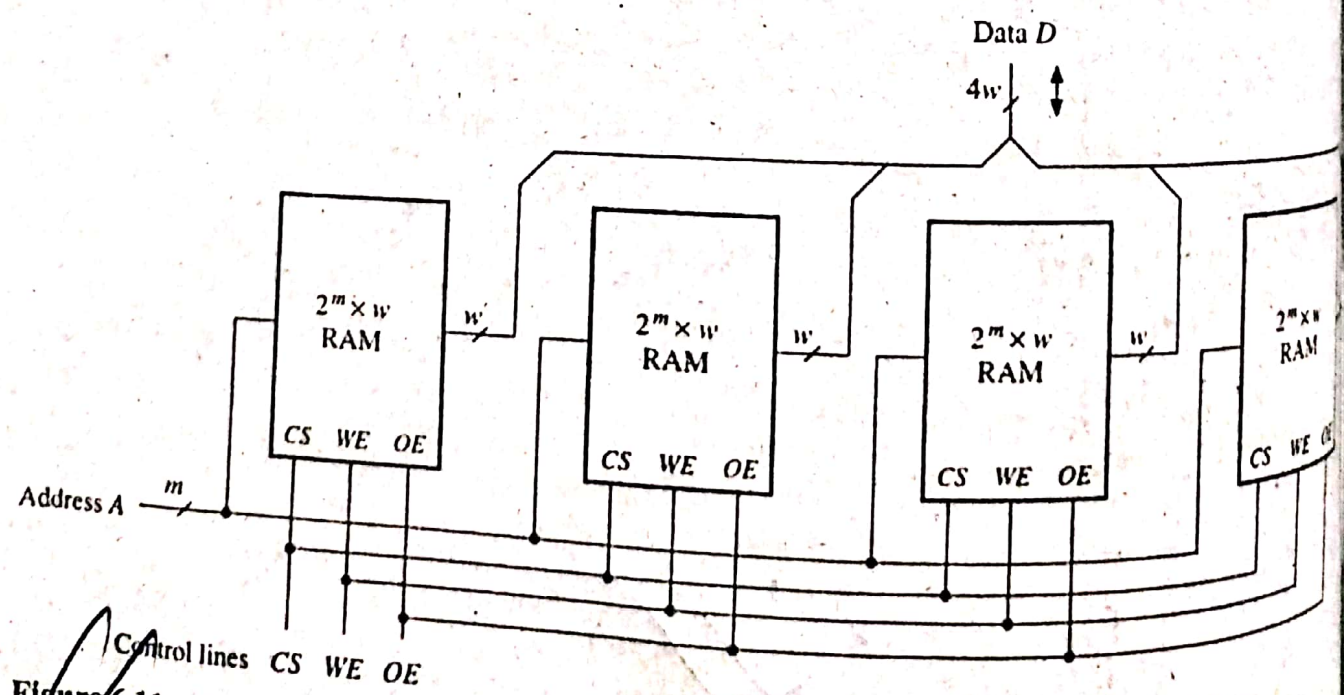


Figure 6.11
Increasing the word size of a RAM by a factor of four.

Scanned by CamScanner

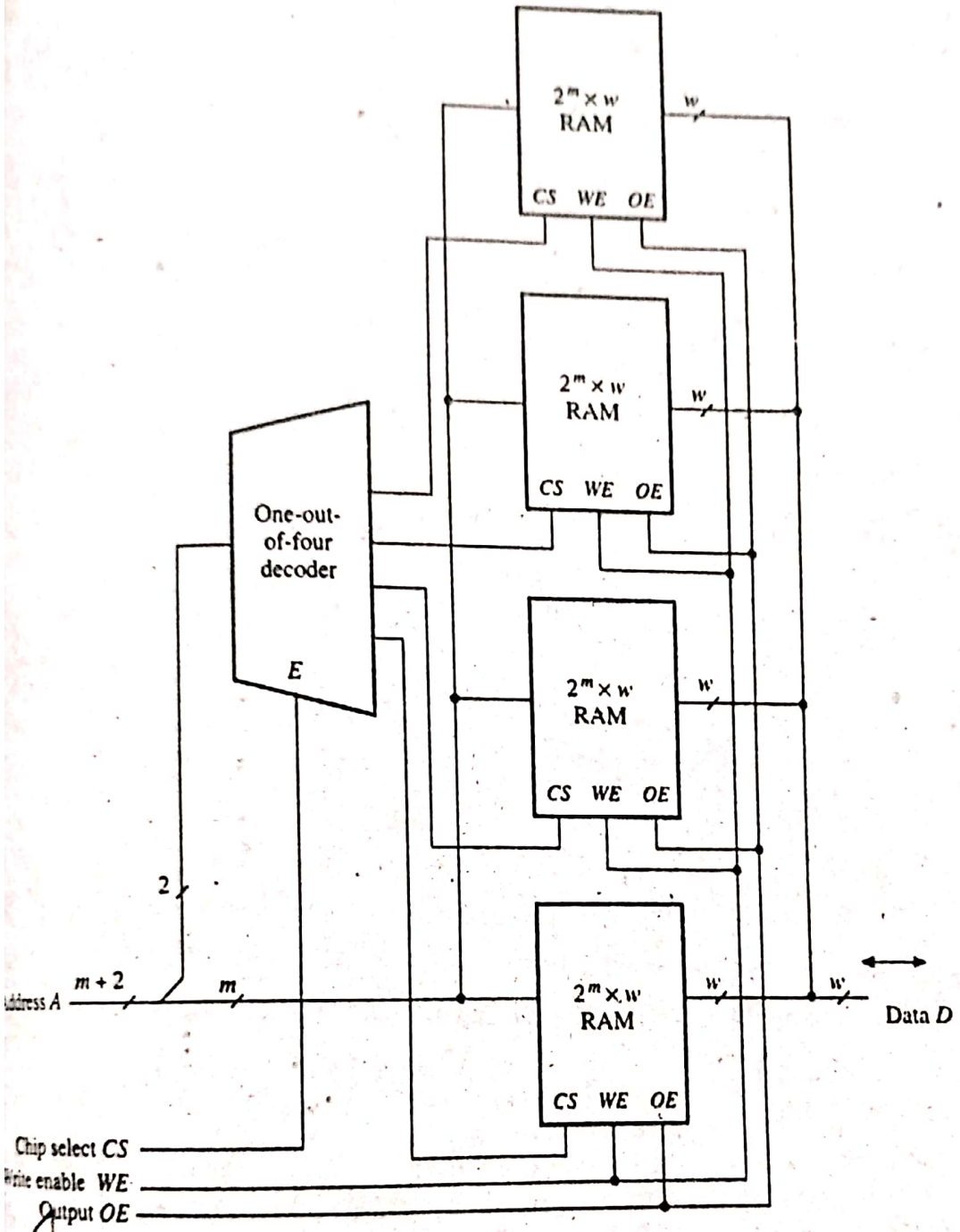


Figure 6.12
Increasing the number of words stored in a RAM by a factor of four.

6.11, and 6.12. First the row address is transferred to the DRAM by the external (master) device, which places the row address on the 8E1's address bus and activates $\overline{RAS}$. The master then places the column address on the address bus, and activates $\overline{CAS}$. $\overline{CAS}$ also serves to indicate that a data word is ready on the data bus (write operation) or that the external bus is ready to receive a data word (read operation). $\overline{WE}$ and $\overline{OE}$ are the write-enable and output-enable lines, respectively. As the overbars in their names indicate, all the control lines are active in the 0 state.

The read access time t_A, which is 50 ns in faster versions of the 8E1, includes the time needed to transfer the row and column addresses to the DRAM and the time to read out a data word. The read cycle time t_M with respect to a "random" address stream is 90 ns, since every such access is followed by an internal restoring write, as depicted

Scanned by CamScanner

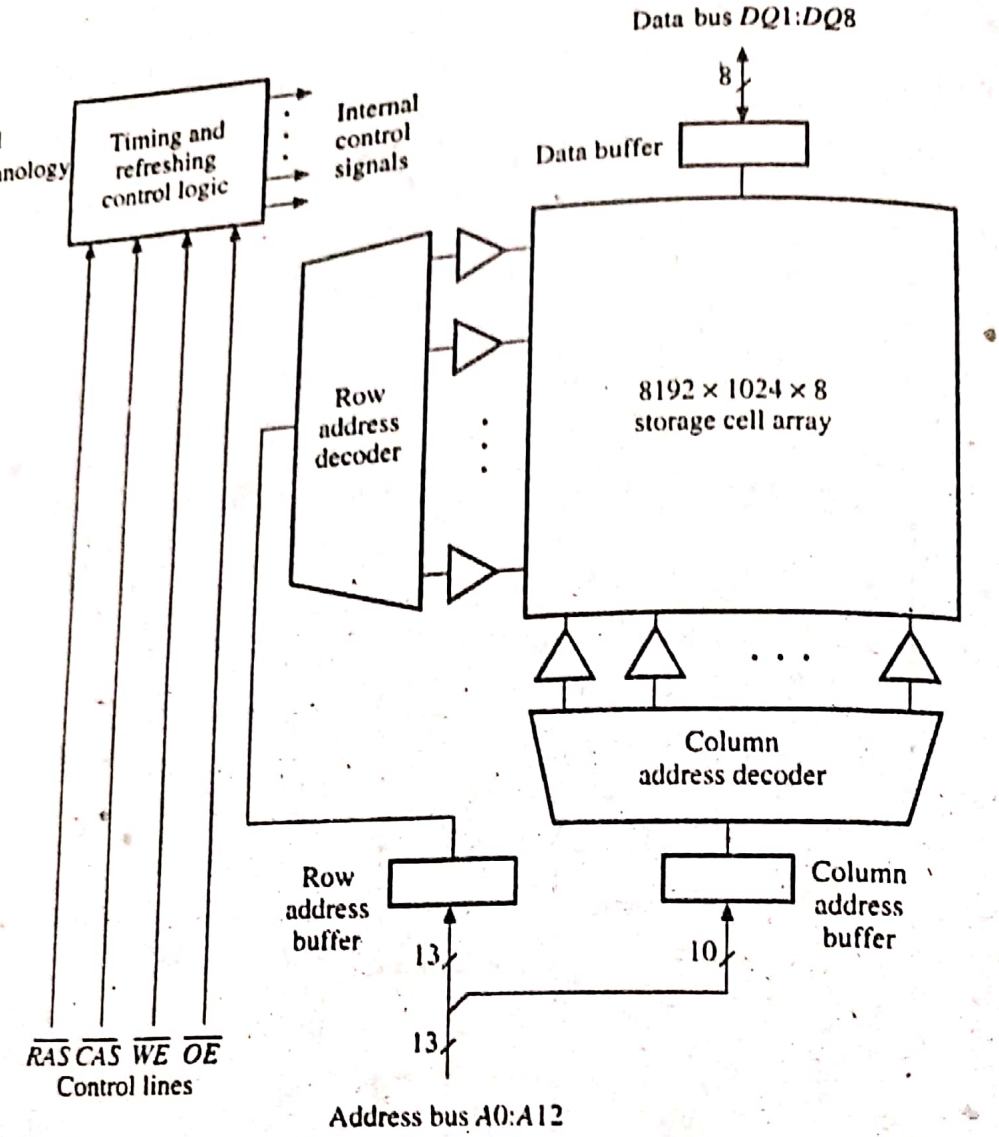


Figure 6.13
Structure of a commercial 8M × 8-bit DRAM chip.

in Figure 6.5. If a sequence of memory accesses share the same row address, then it is sufficient to transfer the row address to the DRAM once at the start of the sequence. This transfer causes an entire row of data, referred to as a *page*, to be read out and held in an internal buffer. A subsequent memory access to the same page needs to transfer only a column address, thus reducing the effective memory cycle time t_M. This time is further reduced by the fact that there is no need to write back and restore the page data every time a word from it is accessed. A fast access method of this type is called *page mode*, and in the 8E1 case it reduces t_M to 30 ns. The memory address space of the 8E1 consists of 8192 rows or pages, each containing 1024 locations. Both $\overline{RAS}$ and $\overline{CAS}$ are normally deactivated before the start of a new read or write cycle. Page mode is established by activating $\overline{RAS}$ to load the row address and then maintaining it active for the duration of a sequence of column-address transfers in which $\overline{CAS}$ is toggled in the normal way.

To ensure that the stored data does not decay, every cell in the memory must be read to refresh it at least once every 64 ms, which is the specified refresh period t_{REF}. An internal restoring write that performs the refreshing accompanies each such read, as in Figure 6.5. The 2-D addressing structure makes it possible to read and restore the contents of an entire row of storage locations in a single read cycle. Hence the refresh

Scanned by CamScanner

controller need only sweep through all the row addresses in a sequence of internal read cycles to implement the refreshing. If a one-row read operation takes 90 ns, then the total time needed to refresh the DRAM once is 90×8192 ns $= 0.737$ ms. Thus the fraction of time devoted to refreshing is $0.737/64 = 1.15$ percent—a negligible amount.

Other semiconductor memories. Techniques similar to those employed in DRAM technology are also used to build several other types of high-density semiconductor memories for computer applications. Read-only memories (ROMs), as their name implies, cannot have their contents rewritten once they are installed in a system, that is, on-line. They are read using random-addressing methods like those of RAM chips. A ROM has essentially the same internal organization and external interface as a RAM, but without the latter's writing ability. However, ROMs have the advantage of being nonvolatile, so they are widely used to store permanent code at the instruction and microinstruction levels. Various ROM types are distinguished by the methods used to program them. Some types can be programmed only once. Others, known as programmable ROMS (PROMs), can be programmed repeatedly, which requires their contents to be erased in bulk off-line and then replaced via a special writing process referred to as "programming." This programming step resembles that of programmable logic devices such as FPGAs (section 2.2.2).

A recent semiconductor technology called *flash memory* offers the same nonvolatility as a PROM, but it can be programmed and erased on-line. The programming can be done a bit at a time, but erasure is done in large blocks—a "flash erase" process from which this memory gets its name. Thus individual bits can be read randomly, but writing must be done in blocks. The storage densities and read-access times of flash memories are comparable to those of DRAMs, but a simpler single-transistor storage cell makes a flash memory potentially cheaper to produce than a DRAM. Flash memories are suitable for writable control stores and as replacements for secondary memories in some applications.

Fast RAM interfaces. The gap between microprocessor and RAM data-transfer rates (bandwidth), especially those of cheap but slow DRAMs, has given rise to novel methods for enabling RAM units to communicate at higher-than-normal speeds. The use of multiple memory types, serial as well as random access, in a memory hierarchy is a separate speedup issue that we examine later. Here we are just concerned with one level in the hierarchy—main memory, for example.

Suppose a particular RAM technology must supply a faster external processor with individually addressable n-bit words. There are two basic ways we can increase the data-transfer rate across its external interface by a factor of S:

- *Use a bigger memory word.* We can design the RAM with an internal memory word size of $w = Sn$ bits. This size permits Sn bits to be accessed as a unit in one memory cycle time T_M. We then need fast circuits inside the RAM that, in the case of a read operation, can access an Sn-bit word, break it into S parts, and output them to the processor, all within the period T_M. During write operations, these circuits must accept up to S n-bit words from the processor, assemble them into an nS-bit word, and store the result, again within the period T_M.
- *Access more than one word at a time.* We can partition the RAM into S separate banks $M_0, M_1, \ldots, M_{S-1}$, each covering part of the memory address space and each

Scanned by CamScanner

provided with its own addressing circuitry. Then it is possible to carry out S independent accesses simultaneously in one memory clock period T_M. Once more, we need fast circuits inside the RAM unit to assemble and disassemble the words being accessed.

Both approaches increase the memory bandwidth by increasing the amount of parallelism in memory accesses, and both require fast parallel-to-serial and serial-to-parallel circuits at the processor-memory interface. Hence an interface technology different from that of the RAM itself may be necessary; therefore, these approaches may not be suitable for single-chip RAM designs. The special interface circuits also add substantially to the overall cost of the memory system.

The S words produced or consumed by the processor in each memory cycle normally have consecutive memory addresses, so we must consider how these addresses are implemented inside the RAM, particularly when S independent memory banks are used. Let $X_h, X_{h+1}, X_{h+2}, \ldots$ be words that are expected to be accessed in sequence by the processor, for example, consecutive instruction words in a program. They will normally be mapped to consecutive physical addresses $A_i, A_{i+1}, A_{i+2}, \ldots$ in the RAM. The following rule is employed to distribute these addresses among S memory banks:

- *Interleaving rule:* Assign address A_i to bank M_j if $j = i$ (modulo S).

Thus $A_0, A_S, A_{2S}, \ldots$ are assigned to M_0; $A_1, A_{S+1}, A_{2S+1}, \ldots$ are assigned to M_1; and so on. This way of distributing addresses among memory banks is *address interleaving*. The interleaving of addresses among S banks according to the above rule is *S-way interleaving*. It is convenient to make S, the number of banks, a power of two, say, $S = 2^p$. Then the least significant p bits of a memory address immediately identify the bank to which the address belongs.

The appropriate number of memory banks S is determined by comparing the cycle time of the RAM technology to the data requirements of its host processor. Consider the case of the Cray-1, an influential supercomputer of the mid-1970s, which uses address interleaving in its main memory M. The CPU cycle time is 12.5 ns, and the semiconductor main memory has a cycle time of $t_M = 50$ ns and a word size of $w = 64$ bits. (The Cray-1 has no cache, however.) Although the number of memory accesses associated with each CPU cycle varies from cycle to cycle, a reasonable estimate is that when operating at maximum speed, one instruction word and two input operand words are read from M and one result word is written into M. Hence a memory bandwidth of four 64-bit words per CPU cycle, or 16 words per memory cycle, is required. Consequently, the Cray-1 has 16 memory banks and uses 16-way address interleaving.

The efficiency of an interleaved memory system is highly dependent on the order in which memory addresses are generated; this order is determined by the programs being executed. If two or more addresses require simultaneous access to the same module, then memory *interference* or *contention* occurs. The memory accesses in question cannot be executed simultaneously. In the worst case, if all addresses refer to the same module, the advantages of interleaving are entirely lost.

Various high-performance interfacing techniques have been devised for RAMs [Kumanoya, Ogaywa, and Inoue 1995]. As discussed in Example 6.1, a 2-D RAM organization with multiplexed row and column addresses facilitates *page addressing*, in which the row or page address remains fixed while the processor supplies a

Scanned by CamScanner

of column addresses. This technique, which exists in several variations, has effect of approximately doubling the data-transfer rate compared with pure dom" addressing. Another DRAM design style called *synchronous DRAM* RAM) achieves a speed doubling by pipelining its internal operations and by mentingtwo-way address interleaving. To facilitate this internal architecture, ming relationships among the SDRAM's control signals (*WE*, *CS*, and so on) streamlined so that the SDRAM presents a synchronous (clocked) interface to outside world. The so-called *cached DRAMs* (CDRAMs) feature an on-chip he realized by a small, fast SRAM that acts as a high-speed buffer or front-end mory for the main DRAM. A common characteristic of the preceding RAM es is that they can have a fast *burst mode* of operation, where an initial slow ess is followed by a sequence or burst of much faster accesses.

EXAMPLE 6.2 THE RAMBUS DRAM AND INTERFACE [PRINCE 1996]. First announced in 1992, this is a proprietary DRAM design with a supporting processor-memory interface that aims to transfer memory data at very high rates over a narrow processor-memory link. It employs several speedup techniques, including a synchronous interface, address interleaving and caching inside the DRAM units, very fast signal timing, and stringent electrical design rules. The Rambus data bus is 8 or 9 bits wide, with the 9th bit typically serving as a parity check. The peak data transfer rate is 500 MB/s which, however, is achievable only in burst mode.

Figure 6.14 depicts the overall Rambus organization. As we will see in Chapter 7, it is closer in style to that of a typical IO interface than a traditional memory interface. Rambus DRAM units are attached to this shared interface—the Rambus channel—which consists of a nine-line data bus D and a small set of control lines. Access is controlled either by the host CPU or a special Rambus controller chip acting as the master control unit. Each Rambus DRAM unit covers part of the memory-address space and acts as an independent slave device that communicates with the master via the Rambus channel. The S_{in}/S_{out} lines that link the DRAM units in daisy-chain fashion are used for initialization. They enable the master to visit each DRAM unit in turn to load a configuration register that determines the range of addresses to which that unit responds.

Normal Rambus operation is as follows. The master transmits an initial "packet" of information on the Rambus channel; this packet contains a target memory address

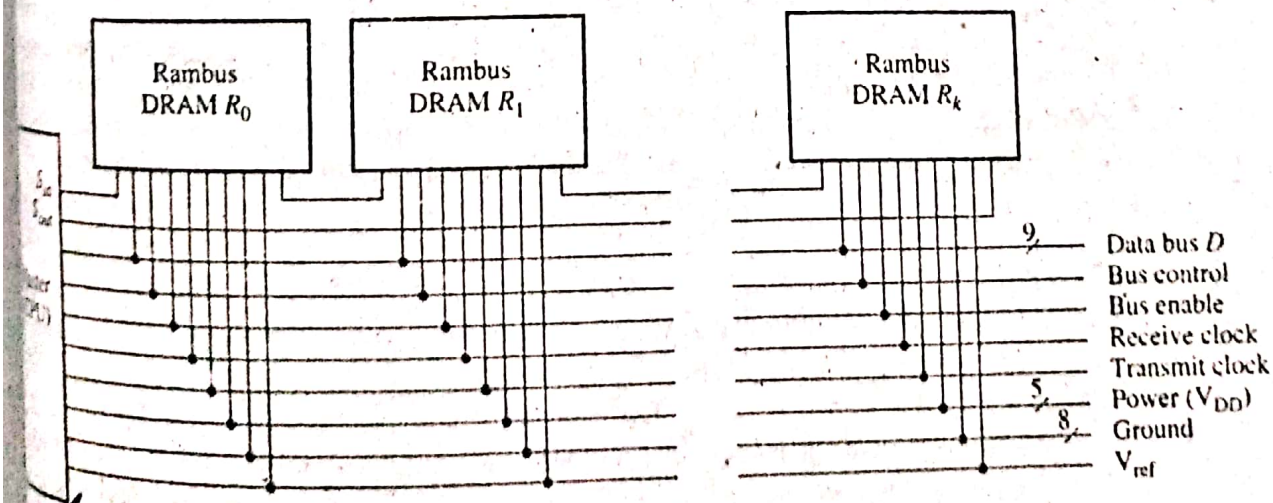


Figure 6.14
Rambus DRAM interface.

Scanned by CamScanner

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 transmit 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 MHz, 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 through 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 times are 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.

Scanned by CamScanner

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

$$t_B = t_S + \frac{1}{2r} + \frac{n}{rN} \tag{6.1}$$

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

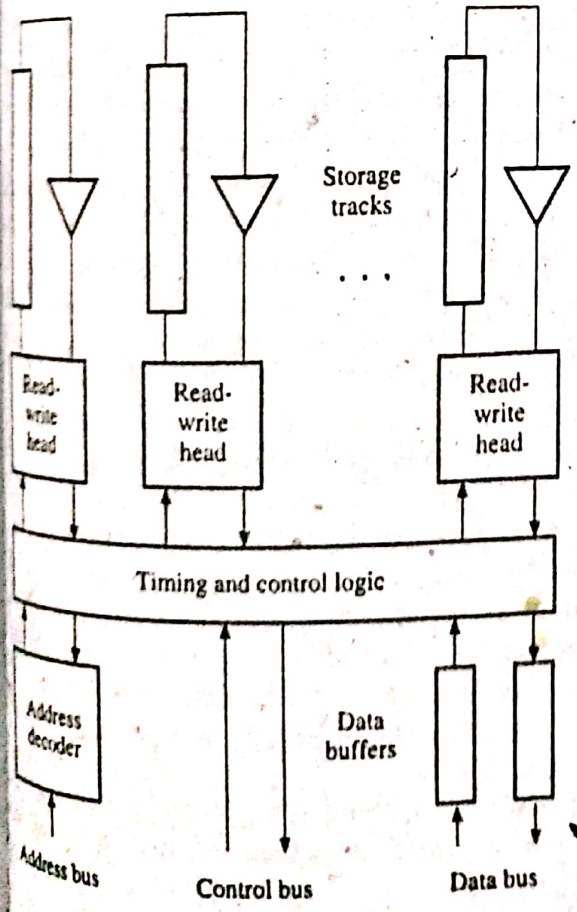


Figure 6.15
Organization of a serial-access memory unit.

Scanned by CamScanner

tion to tra... accessed until it coincides w... generates the address of the block that is c... tion occurs, a track-position indicator generates the address of the block that is c... rently passing the read-write head. The generated address is compared with t... block address produced by the address decoder. When they match, the selec... head is enabled and data transfer between the storage track and the memory da... buffer registers begins. The read-write head is disabled when a complete block information has been transferred.

The number of different types of storage media and access mechanisms used construct serial memories is quite large. In many such memories the read-wr heads or the storage locations are moved through space by electromechanic devices, such as electric motors, in order to perform an access. The most wide used group of secondary memory devices, magnetic-disk and -tape units, fall in this category, as do many optical memories. Some optical memories—CD-RO are an example—employ laser beams with electromechanical focusing as th read-write heads. Only a few serial memories have no moving parts, for examp the so-called solid-state disks, which use semiconductor RAM technology to sim late the behavior of disk memories in applications that need unusually fast (a therefore expensive) secondary memory. Magnetic memories with electromecha cal access have had many years of development. The storage media (magne disks and tape cartridges) are inexpensive and portable. Electromechanical equ ment is less reliable than electronic equipment, however, and is a common sou of computer system failure.

Magnetic-surface recording. Magnetic-disk and -tape memories store inf mation on the surface of tracks coated with a magnetic medium such as fer oxide. Each cell of a track has two stable magnetic states that represent logica and 1. These magnetic states are defined by the direction or magnitude of the cel magnetic flux in the cell. Electric currents alter and sense the magnetic states, example, via an inductive read-write head of the type shown in Figure 6.16. T read and write signals pass through coils around a ring of soft magnetic material. very narrow gap separates the ring from a cell on the storage track so that th respective magnetic fields can interact. This interaction permits information tran fer between the read-write head and the storage medium.

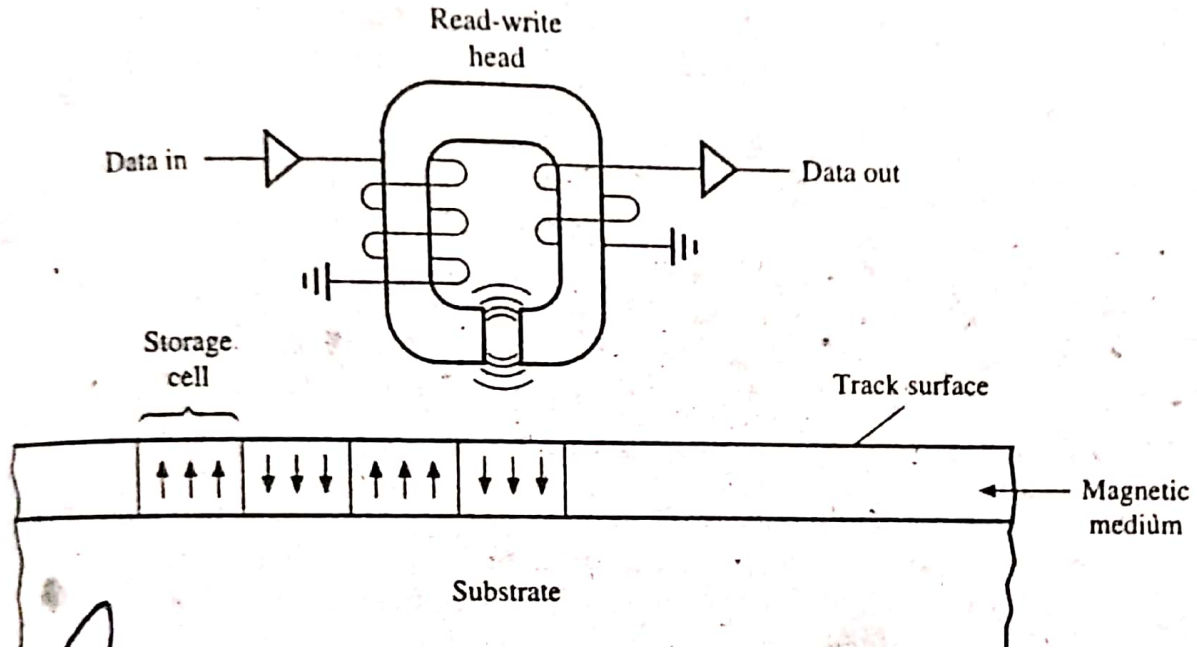
To write data, the addressed cell is moved under the read-write gap. A pul of current is then transmitted through the write coil, which alters the magnet field at the ring gap; this in turn alters the magnetization state of the cell under th gap. The direction or magnitude of the write current determines the resulting stat To read a cell, it is moved past the read-write head, causing the magnetic field c the cell to induce a magnetic field in the core material of the read-write head Since the cell is in motion, this magnetic field varies and so induces an electro voltage pulse in the read coil. This voltage pulse, which is then fed to a sens amplifier, identifies the state of the cell. The readout process is nondestructive; it ... by the

Scanned by CamScanner

cks form concentric circles on the surface of a plastic or metal disk. In tape ies the tracks form parallel lines on the surface of a long, narrow plastic tape.

agnetic-disk memories. A magnetic-disk unit employs storage media con- s of thin disks with a coating of magnetic material on which data can be ded. One or both surfaces of a disk contain thousands of recording tracks ed in concentric circles as shown in Figure 6.17*a*. Several disks can be ed to a common spindle; the four disks in Figure 6.17*b* provide up to eight ding surfaces.

During operation of the memory, the disks are rotated at a constant speed by a *drive unit*. Each recording surface is supplied with at least one read-write The read-write heads can be connected to form a read-write arm, as shown in e 6.17*b*, so that all heads move in unison. This arm moves back and forth to



Scanned by CamScanner

EXAMPLE 6.3 A COMMERCIAL MAGNETIC HARD-DISK MEMORY UNIT [QUA-TUM CORP. 1996]. The XP39100 is a 9.3 GB hard-disk memory in the Atlas II se manufactured by Quantum Corp. and introduced in the mid-1990s. It is housed i rectangular box whose dimensions are approximately 14.6 × 10.2 × 4.14 cm. It c tains ten 3.5 in (8.89 cm) diameter disks, supplying a total of 20 recording surfa each with its own read-write head. Figure 6.18 summarizes the main features of device. The cited capacity of 9.1 GB is for a *formatted* disk, which stores a direc and other control information needed to make the disk drive ready for use. The num of sectors along a track varies from 108 to 180, and each sector within a track acc modates a 512-byte block. While the sector size is fixed, the number of sectors track varies due to the fact that the inner tracks are smaller and can therefore store information at the maximum recording density of the magnetic medium. The ave 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 average number of sectors per track, implying that $N = 144 \times 512 = 73{,}728$ bytes/t Observe that the seek time is the major factor in t_B. The data-transfer rate $rN =$ 1

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

Scanned by CamScanner

6.2 MEMORY SYSTEMS

This section examines the general characteristics of memory systems that have 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

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

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

Scanned by CamScanner

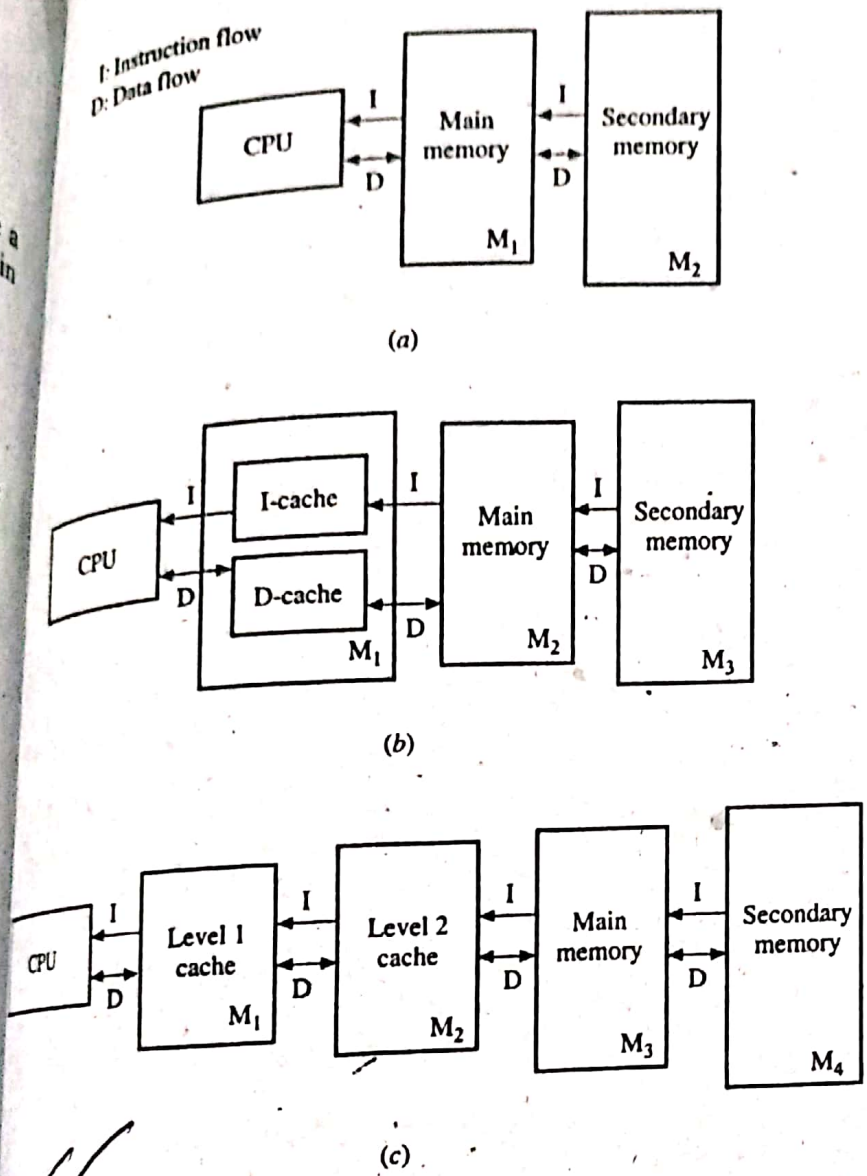


Figure 6.21
Common memory hierarchies with (*a*) two, (*b*) three, and (*c*) four levels.

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

Scanned by CamScanner

- 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-

Scanned by CamScanner

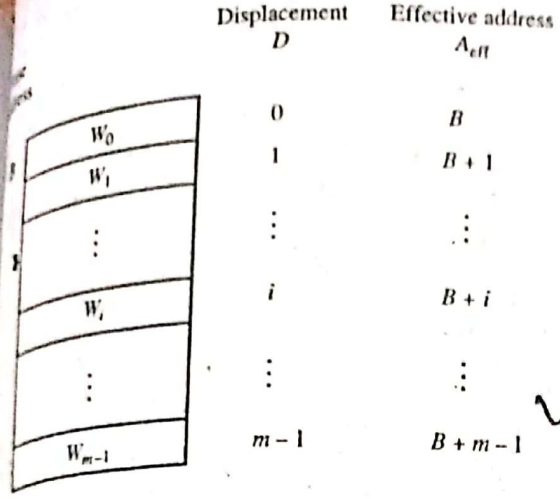


Figure 6.24
Block of m words with (base) address B.

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,

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

When 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 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

Scanned by CamScanner

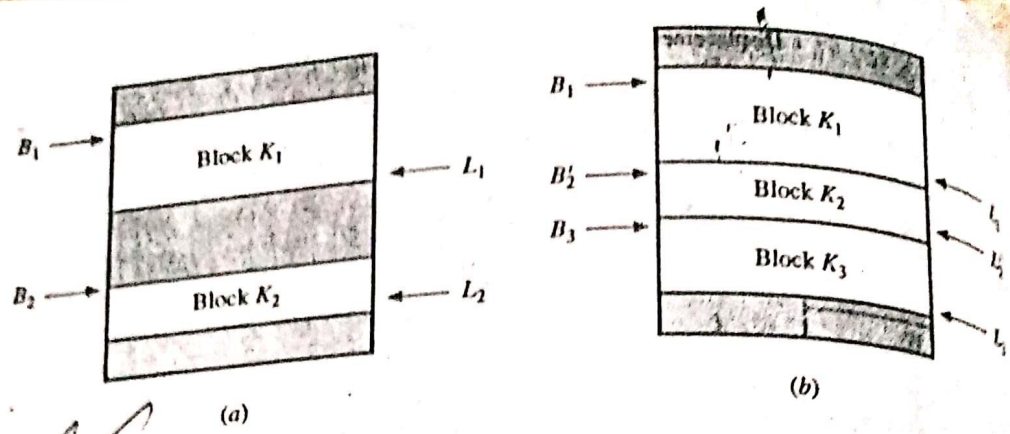


Figure 6.25
Relocation of blocks in memory using base and limit addresses.

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

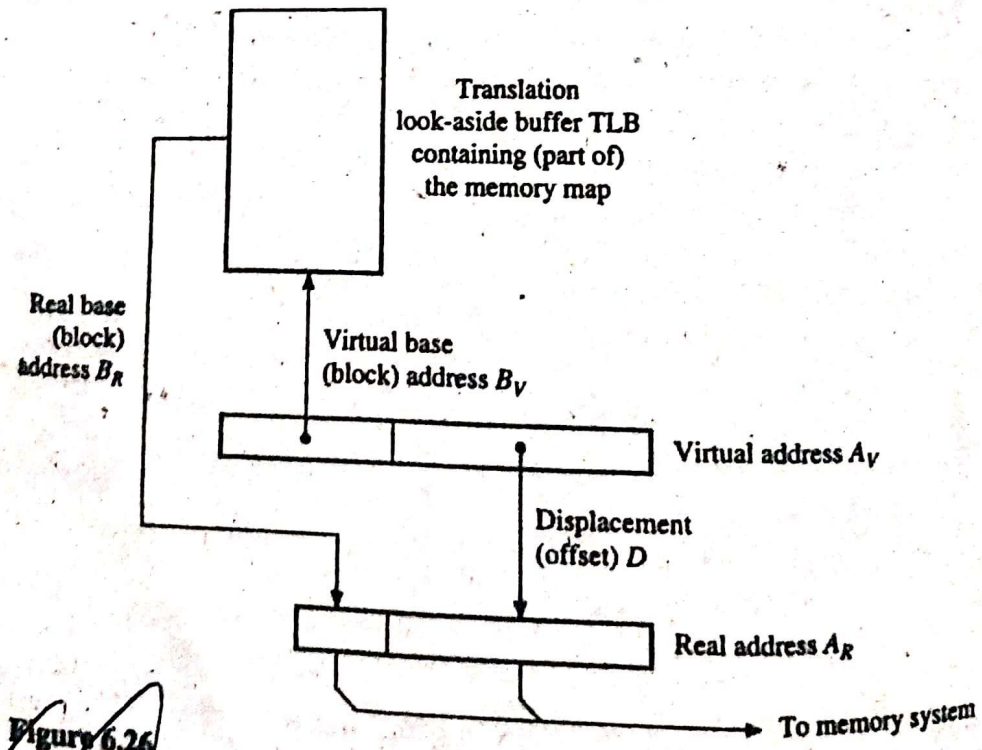


Figure 6.26
Structure of a dynamic address-translation system.

Scanned by CamScanner

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-

Scanned by CamScanner

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 seems 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

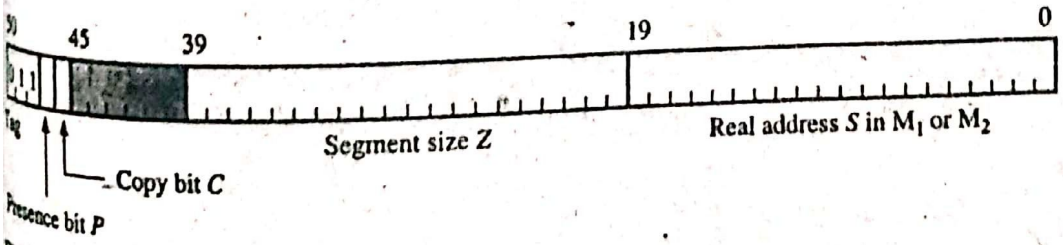


Figure 6.28
Segment descriptor of the Burroughs B6500/7500.

Scanned by CamScanner

is suspended while the operating system transfers the required segment from M_2 to M_1. When $P = 1$, the CPU compares D to the segment size field Z in the descriptor. If $D < Z$, the address field S from the descriptor is added to the displacement D. The result $S + D$ is the address of the required word in M_1, which can then be accessed. If $D \geq Z$, then D is invalid and an interrupt occurs.

The main advantage of segmentation is that segment boundaries correspond to natural program and data boundaries. Consequently, information that is shared among different users is often organized into segments. Because of their logical independence, a program segment can be changed or recompiled at any time without affecting other segments. Certain properties of programs such as the scope (range of definition) of a variable and access rights are naturally specified by segment. These properties require that accesses to segments be checked to provide protection against unauthorized use; this protection is most easily implemented when the units of allocation are segments. Certain segment types—stacks and queues, for instance—vary in length during program execution. Segmentation varies the region assigned to such a segment as it expands and contracts, thus efficiently using the available memory space. On the other hand, the fact that segments can be of different lengths requires a relatively complex allocation method to avoid excessive fragmentation of main-memory space. This problem is alleviated by combining segmentation with paging, as discussed later.

Some computers implement a more specialized form of segmentation. The MIPS R2/3000 divides the virtual address space into four large regions that are treated as segments; see Example 6.4. Just 3 bits of the virtual address define the current segment. Microprocessors in the Intel 80X86 series, including the Pentium, have four 16-bit segment registers forming a segment table that supports a very large number of segments.

Pages. A page is a fixed-length block that can be assigned to fixed regions of physical memory called *page frames*. The chief advantage of paging is that the transfer between memory levels is simplified: an incoming page can be assigned to any available page frame. In a pure paging system, each virtual address consists of two parts: a page address and a displacement. The memory map, now referred to as a *page table,* typically contains the information shown in Figure 6.29. Each (virtual) page address has a corresponding (real) address of a page frame in main or secondary memory. When the presence bit $P = 1$, the page in question is present in main memory, and the page table contains the base address of the page frame to which the page has been assigned. If $P = 0$, a page fault occurs and a page swap ensues. The change bit C indicates whether or not the page has been changed since it was last loaded into main memory. If a change has occurred ($C = 1$), the page

Page address	Page frame	Presence bit P	Change bit C	Access rights
A	0000000	1	0	R, X
C	D6C7F9	0	*d*	R, W, X
E	0000024	1	1	R, W, X
F	0000016	1	0	R

Figure 6.29
Representative organization of a page table.

Scanned by CamScanner

be copied onto secondary memory when it is preempted. The page table can contain memory protection data that specifies the access rights of the current user to read from, write into, or execute the page in question. Page tables differ from segment tables primarily in the fact that they contain no block size informa-

As noted earlier, pages require a simpler memory allocation system than seg-ments since block size is not a factor in paging. On the other hand, pages have no logical significance, as they do not represent program elements. Paging and seg-mentation can also be compared in terms of memory fragmentation. In systems without segmentation, holes of different sizes tend to proliferate throughout main memory; they can be eliminated by the time-consuming process of memory com-paction. Unusable space between occupied regions is called *external fragmenta-tion*. Since page frames are contiguous, no external fragmentation occurs in paged systems. However, if a k-word block is divided into p n-word pages, and k is not a multiple of n, the last page frame to which the block is assigned will not be filled. Unusable space within a partially filled page frame is called *internal fragmentation*.

Paging and segmentation can be combined in an attempt to gain the advantages of both. The great advantage of breaking a segment into pages is that it eliminates the need to store the segment in a contiguous region of main memory. Instead, all that is required is a number of page frames equal to the number of pages into which the segment has been broken. Since these page frames need not be contiguous, the problem of placing a large segment in main memory is eased.

When segmentation is used with paging, a virtual address has three compo-nents: a segment index SI, a page index PI, and a displacement (offset) D. The memory map then consists of one or more segment tables and page tables. For fast address translation, two TLBs can be used as shown in Figure 6.30, one for seg-ment tables and one for page tables. As discussed earlier, the TLBs serve as fast caches for the memory maps. Every virtual address A_V generated by a program goes through a two-stage translation process. First, the segment index SI is used to index the current segment table to obtain the base address PB of the required page table. This base address is combined with the base index PI (which is just a dis-placement within the page table) to produce a page address, which is then used to access a page table. The result is a real page address, that is, a page frame number, which can be combined with the displacement part D of A_V to give the final (real) address A_R. This system, as depicted in Figure 6.30, is very flexible. All the various memory maps can be treated as paged segments and can be relocated anywhere in physical memory space.

EXAMPLE 6.5 MEMORY ADDRESS TRANSLATION IN THE INTEL PENTIUM [INTEL 1994]. The Pentium is a 32-bit microprocessor introduced in 1993 that pro-vides direct hardware support for both segmentation and paging. It is a member of Intel's 80X86 microprocessor family and maintains some degree of compatibility at the object-code level with its predecessors back to the original, 1976-vintage 8086 CPU. Most of the memory-addressing features discussed here originated with the 80386, introduced in 1985.

Like the MIPS R2/3000 (Example 6.4), the Pentium's real address space can be as large as 4 GB (2^{32} bytes); however, the virtual address space can be an extremely large 64 TB (64 terabytes = 2^{46} bytes). An on-chip MMU has a segmentation unit that per-forms address translation for segments ranging in size from 1 to 2^{32} bytes. A separate paging unit handles address translation for pages of size 4 KB or 4 MB. Any one of the

Scanned by CamScanner

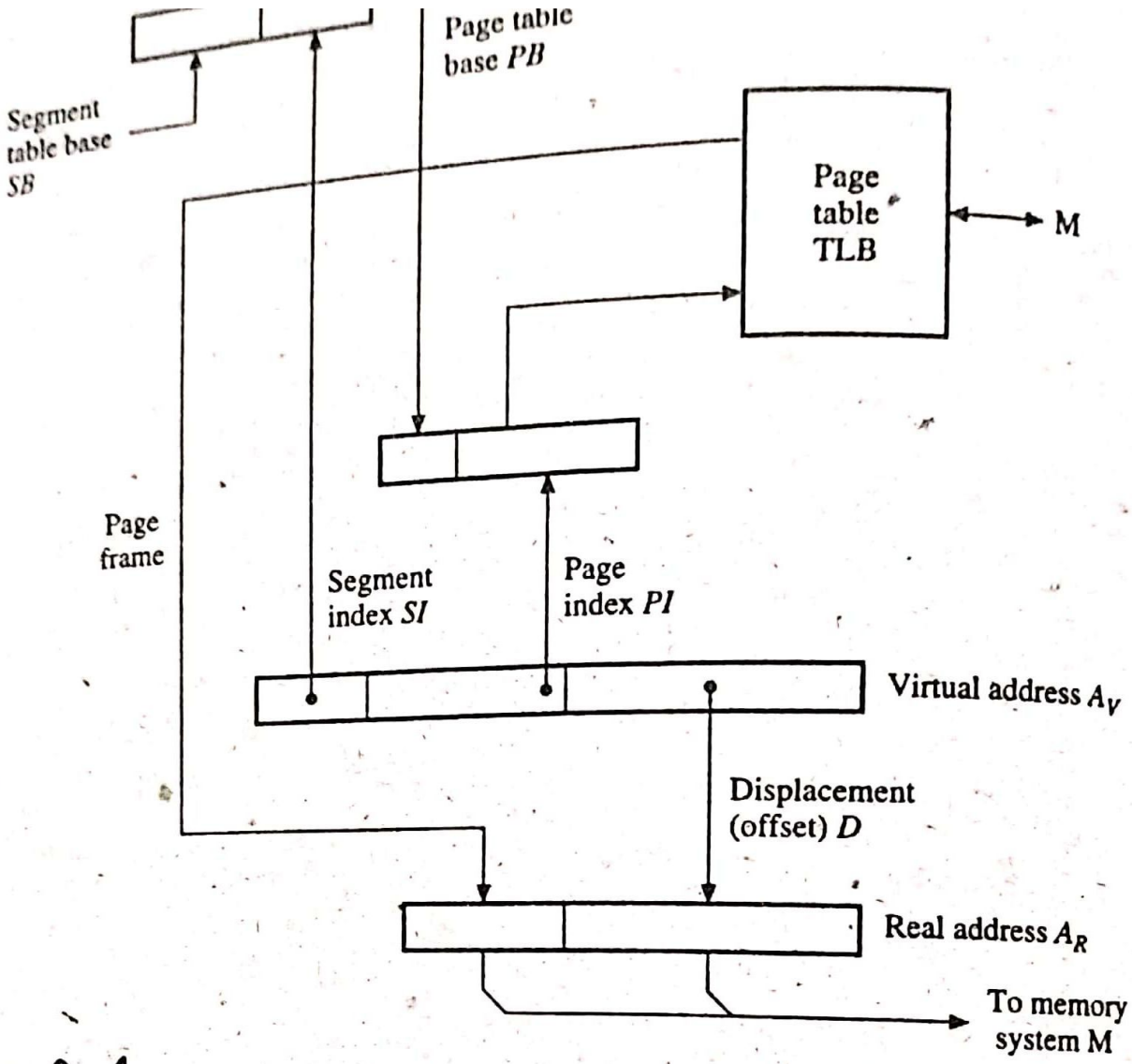


Figure 6.30
Two-stage address translation with segments and pages.

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 contains

Scanned by CamScanner

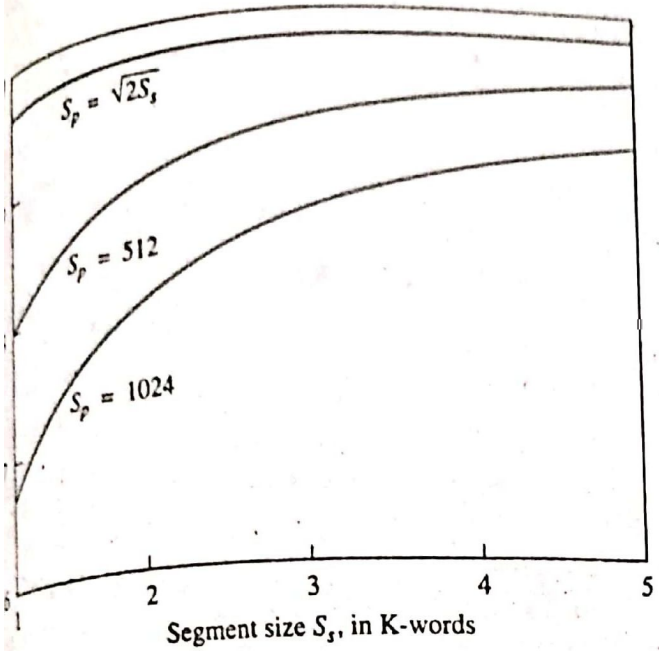


Figure 6.32
Influence of page size S_p and segment size S_s on space utilization u.

M_1. This likelihood tends to increase with the number of pages stored in M_1; it erefore 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, ncreases with S_p. However, when S_p exceeds a certain value, H begins to ease. Figure 6.33 shows some typical curves relating H and S_p for various n-memory capacities. Simulation studies indicate that in large systems, the val- of S_p yielding the maximum hit ratios can be greater than the "optimum" page given by Equation (6.11). Since high H is important in achieving small t_A (due he relatively slow rates at which page swapping takes place), values of S_p that ximize H are preferred. The first computer with a paging system (the University Manchester's Atlas computer) had a 512-word page, while the Pentium dis- sed in Example 6.5 supports page sizes of 4 KB and 4 MB.

Memory Allocation

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

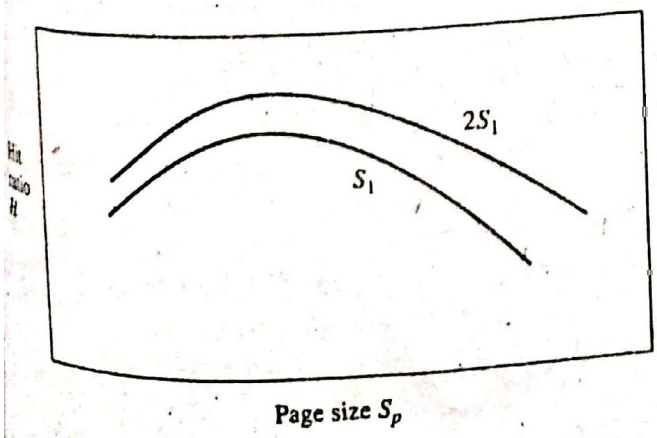


Figure 6.33
Influence of page size S_p on hit ratio H.

Scanned by CamScanner

blocks of data. Blocks are swapped automatically among the levels in order to minimize the access time seen by the processor. Swapping generally occurs in response to processor requests (*demand swapping*). However, to avoid making the processor wait while a requested item is being moved to the fastest level of memory M_1, some kind of *anticipatory swapping* must be implemented, which involves transferring blocks to M_1 in anticipation that they will be required soon. Good short-range prediction of access-request patterns is possible because of locality of reference.

The placement of blocks of information in a memory system is called *memory allocation* and is the topic of this section. The method of selecting the part of M_1 in which an incoming block K is to be placed is the *replacement policy*. Simple replacement policies assign K to M_1 only when an unoccupied or inactive region of sufficient size is available. More aggressive policies preempt occupied blocks to make room for K. In general, successful memory allocation methods result in a high hit ratio and a low average access time. If the hit ratio is low, an excessive amount of swapping between memory levels occurs, a phenomenon known as *thrashing*. Good memory allocation also minimizes the amount of unused or underused space in M_1.

The information needed for allocation within a two-level hierarchy (M_1,M_2)—unless otherwise stated, we will assume the main–secondary-memory hierarchy—can be held in a memory map that contains the following information:

- *Occupied space list for M_1*. Each entry of this list specifies a block name, the (base) address of the region it occupies, and, if variable, the block size. In systems using preemptive allocation, additional information is associated with each block to determine when and how it can be preempted.
- *Available space list for M_1*. Each entry of this list specifies the address of an unoccupied region and, if necessary, its size.
- *Directory for M_2*. This list specifies the unit(s) that contain the directories for all the blocks associated with the current programs. These directories, in turn, define the regions of the M_2 space to which each block is assigned.

When a block is transferred from M_2 to M_1, the memory management system makes an appropriate entry in the occupied space list. When the block is no longer required in M_1, it is *deallocated* and the region it occupies is transferred from the occupied space list to the available space list. A block is deallocated when a program using it terminates execution or when the block is replaced to make room for one with higher priority.

Many preemptive and nonpreemptive algorithms have been developed for dynamic memory allocation. Accurate analysis of their performance is difficult; as a result, simulation is the most widely used evaluation tool. The performance of an allocation algorithm can be estimated by the various parameters introduced in section 6.2.1, such as the hit ratio H, the access time t_A, and the space utilization u.

Nonpreemptive allocation. Suppose a block K_i of n_i words is to be transferred from M_2 to M_1. If none of the blocks already occupying M_1 can be preempted (overwritten or moved) by K_i, then it is necessary to find or create an "available" region of n_i or more words to accommodate K_i. This process is termed *nonpreemptive* allocation. The problem is more easily solved in a paging system where all

Scanned by CamScanner

(pages) have size S_p words and M_1 is divided into fixed S_p-word regions (frames). The memory map (page table) is searched for an available page if one is found, it is assigned to the incoming block K_i. This easy allocation is the principal reason for the widespread use of paging. If memory space into regions of variable length, however, then it becomes more difficult incoming blocks efficiently.

Two widely used algorithms for nonpreemptive allocation of variable-sized —unpaged segments, for example—are first fit and best fit. The *first-fit* scans the memory map sequentially until an available region R_j of n_i or words is found, where n_i is the size of the incoming block K_i. It then allocates The *best-fit* approach requires searching the memory map completely and K_i to a region of $n_j \geq n_i$ words such that $n_j - n_i$ is minimized.

Suppose, for example, that at some point in time M_1 stores three blocks, as in 6.34*a*. There are three available (shaded) regions, and the available space has the form:

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

Further, suppose that two new blocks K_4 and K_5 whose sizes are 100 and 250 words, respectively, are to be assigned to M_1. Figures 6.34*b* and *c* show the results obtained using the first-fit and best-fit methods, respectively, when the memory scan begins at address 0.

The first-fit algorithm has the advantage of needing less time to execute than the best-fit approach. If the best-fitting available region can be found by scanning *k*

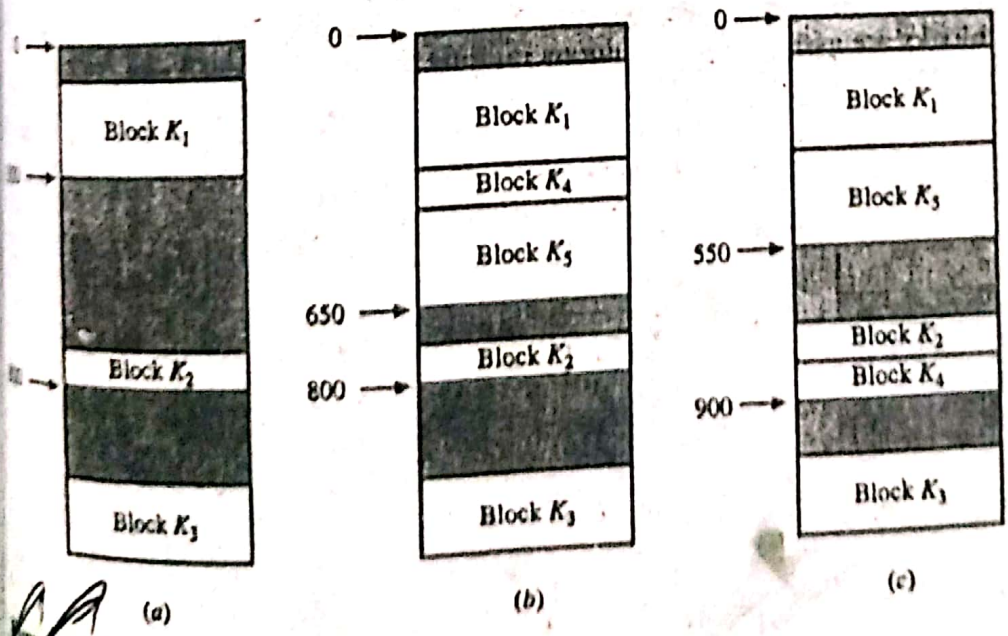


Figure 6.34
(a) Initial memory state; (b) allocation of K_4 and K_5 by first fit; (c) allocation of K_4 and K_5 by best fit.

Scanned by CamScanner

entries of the available space list, the first fit can always be found by scanning 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

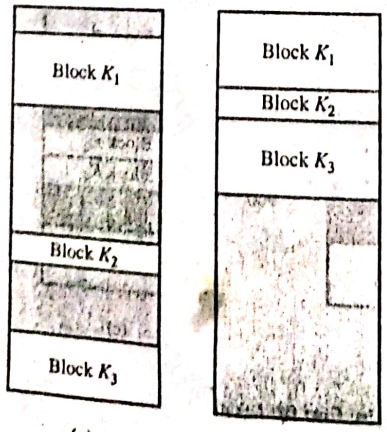


Figure 6.35
Memory allocation (*a*) before and (*b*) after compaction.

Scanned by CamScanner

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.

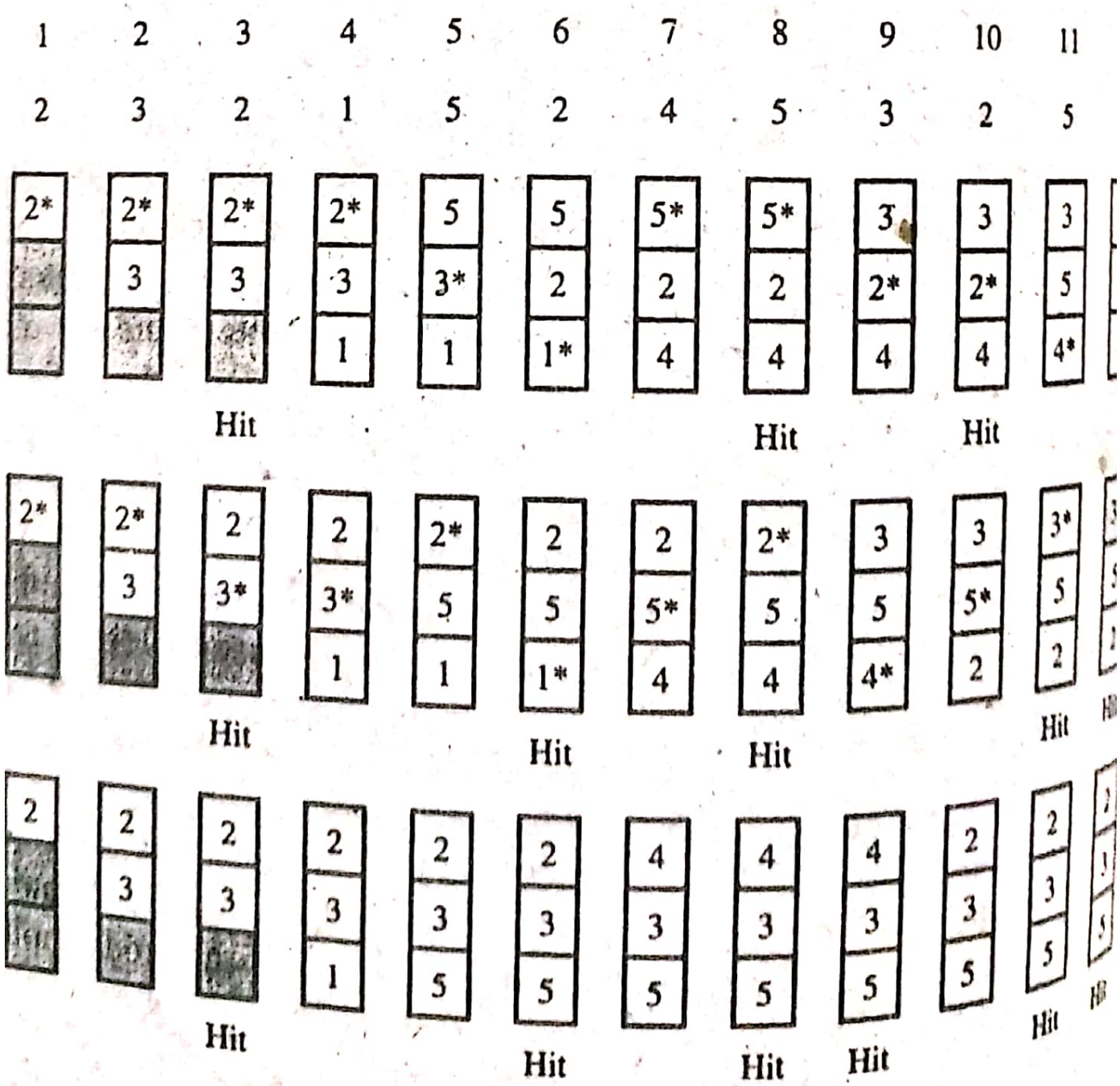
The performance of a replacement policy in a given memory organization can be analyzed using the block address stream generated by a set of representative

Scanned by CamScanner

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

$$2 \quad 3 \quad 2 \quad 1 \quad 5 \quad 2 \quad 4 \quad 5 \quad 3 \quad 2 \quad 5 \quad 2$$

eans that the first page referenced is P_2, the second is P_3, and so on. Fig
ws the manner in which the pages are assigned to M_1 using FIFO, LRU,
OPT replacement policies. The next block to be selected for replacemen
y an asterisk in the FIFO and LRU cases. It will be observed that LRU rec
t P_2 and P_5 are referenced more frequently than other pages, whereas FI



replacement policies

Scanned by CamScanner

does not. Thus FIFO replaces P_2 twice, but LRU does so only once. The highest page-hit ratio is achieved by OPT, the lowest by FIFO. The page-hit ratio of LRU is quite close to that of OPT, a property that seems to hold generally.

Stack replacement policies. As discussed in section 6.2.1, the cost and performance of a memory hierarchy can be measured by average cost per bit c and average access time t_A. Equations (6.4) and (6.7) repeated here are convenient expressions for c and t_A:

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

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

The quantities c, t_A, and t_B are determined primarily by the memory technologies used for M_1 and M_2. Once these technologies have been chosen, the hit ratio H can be computed for various possible system configurations. The major variables on which H depends are

The address streams encountered.
The average block size.
The capacity of M_1.
The replacement policy.

Simulation is perhaps the most practical technique used for evaluating different memory system designs. H is determined for representative address traces, memory technologies, block sizes, memory capacities, and replacement policies. Figure 6.36 shows a sample point in this simulation process. Here the address trace, block size, and memory capacity are fixed, and three different replacement strategies are being tested.

Due to the many alternatives that exist, the amount of simulation required to optimize the design of a multilevel memory system can be huge. A number of analytic models for optimizing memory design have been proposed. Notable among these is a technique called *stack processing*, which is applicable to paging systems that use a class of replacement algorithms called stack algorithms [Stone 1993]. Let AT be any page address trace of length L to be processed using a replacement policy RP. Let t denote the point in time when the first t pages of AT have been processed. Let n be a variable denoting the page capacity of M_1. $B_t(n)$ denotes the set of pages in M_1 at time t, and L_t denotes the number of distinct pages that have been encountered at time t. Policy RP is called a *stack algorithm* if it has the following *inclusion property:*

$$B_t(n) \subset B_t(n+1) \qquad \text{if } n < L_t$$

$$B_t(n) = B_t(n+1) \qquad \text{if } n \geq L_t$$

LRU retains in M_1 the n most recently used pages. Since these are always included in the $n + 1$ most recently used pages, it can be seen right away that LRU is a stack algorithm. Some other replacement policies are also of this type. FIFO is a notable exception, however. Consider the following page address stream:

Scanned by CamScanner

CHAPTER 5

Control Design

In this chapter we study the register-level design of the control part of an instruction-set processor; the data-processing part was covered in Chapter 4. The two basic approaches to control-unit design—hardwired and microprogrammed—are discussed in detail. The complex task of controlling pipelined and superscalar processors is also examined.

5.1 BASIC CONCEPTS

First we discuss the general structure and behavior of control units. Then we examine the design of hardwired controllers, which are characterized by the use of fixed (nonprogrammable) logic circuits.

5.1.1 Introduction

We saw in section 2.1.1 that it is useful to separate a digital system into two parts: a datapath (data processing) unit and a control unit. The datapath is a network of functional and storage units capable of performing certain (micro) operations on data words. The purpose of the control unit is to issue control signals to the datapath. These control signals enter the datapath at "control points," where they select the functions to be performed at specific times and route the data through the appropriate parts of the datapath unit. In other words, the control unit logically reconfigures the datapath to implement some specified instruction or program.

A CPU's datapath contains circuits to perform arithmetic and logical operations on words such as fixed-point or floating-point numbers. The internal struc-

Scanned by CamScanner

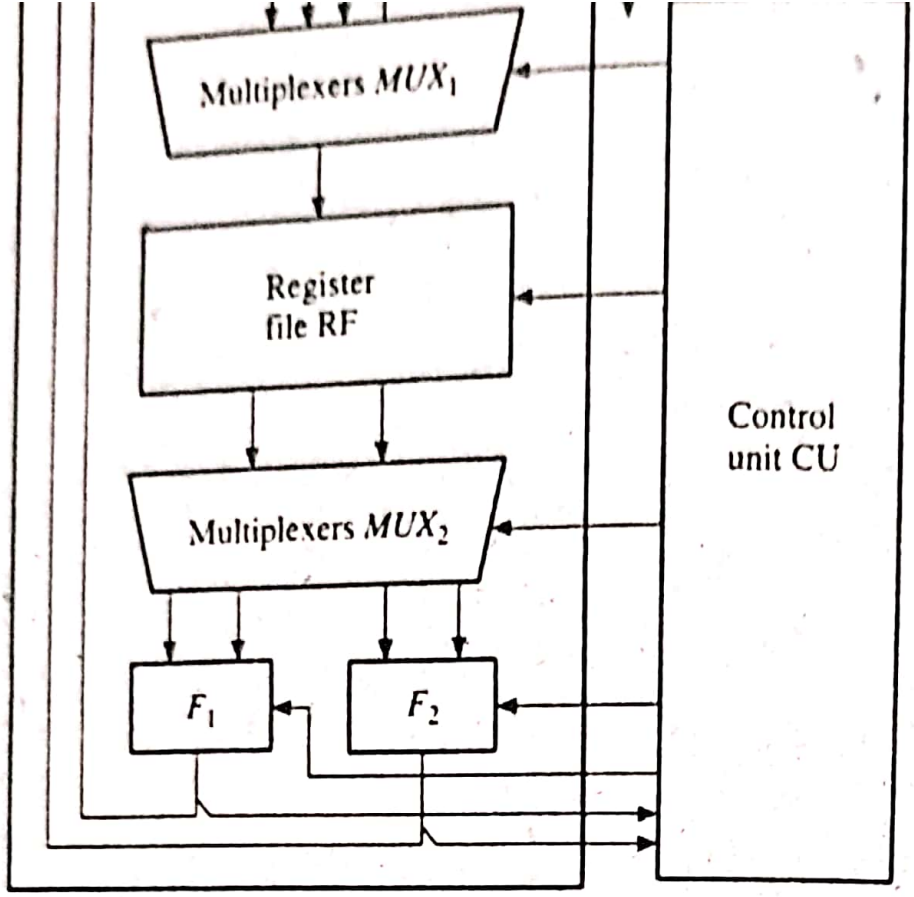


Figure 5.1
Processor composed of a datapath unit DP and a contr unit CU.

ture of the datapath circuit DP of a small microprocessor is depicted in Figure 5 It contains a register file RF for temporary storage of operands, two functio units F_1 and F_2 responsible for data processing, and multiplexers to allow the d to be steered through DP. Typical functional units are an ALU performing ad tion, subtraction, and logical operations; a shifter; or a multiplier. The control u CU receives external instructions or commands, which it converts into a sequen of control signals that the CU applies to DP to implement a sequence of regist transfer operations.

Figure 5.2 shows the control signals that implement an addition instruction the form ADD A,B, which we write as

$$A := A + B; \tag{5.1}$$

in our HDL notation. Assume that this operation can be executed in a single clo cycle, whose timing details are not of concern at this level of abstraction. The inp variables A and B are obtained from registers of the same name in RF, and th result is stored back into register A. Observe that the registers of RF permit the contents to be read from and written into in the same clock cycle, a basic propert of the (edge-triggered) flip-flops from which such registers are constructed. RF is configured with one input and two output ports to support operations like (5.1) with two or three addresses. Besides selecting the data registers to be used, the con trol unit CU must also select the operation to be performed on the data, in this case functional unit F_1 in DP. ...nections for

Scanned by CamScanner

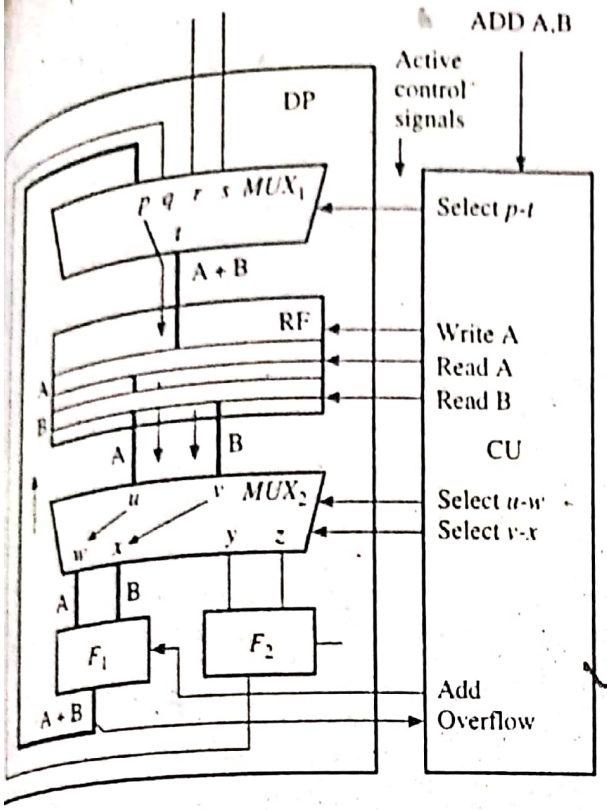


Figure 5.2
The processor of Figure 5.1 configured to implement the add operation A := A + B.

Thus we see that CU must activate the following three types of control signals ring the clock cycle in which the ADD A,B instruction is executed.

Function select: Add.
Storage control: Read A, Read B, Write A.
Data routing: Select *p-t*, Select *u-w*, Select *v-x*.

ere is usually some feedback of control information from DP to CU to indicate ceptional conditions encountered during instruction execution. In the example of gure 5.2, the functional unit F_1 performing the addition sends an overflow signal CU whenever the sum A + B exceeds the normal word size.

Multicycle operations. Many types of instructions are executed in a single ock cycle—indeed, single-cycle execution is a central goal of RISC design. Some ructions require more than one clock cycle for their execution, however. For ample, double-precision addition can be implemented by a two-instruction quence (program) of the form

$$\begin{array}{ll} \text{ADD} & \text{AL, BL} \\ \text{ADDC} & \text{AH, BH} \end{array} \qquad (5.2)$$

hich involves two double-word operands A and B. The first (ADD) instruction in 5.2) adds the low-order half (right word) of B to the low-order half of A, implicitly enerating and storing a carry-out signal C. The second (ADDC) instruction adds he high-order half of B to the high-order half of A along with the carry C, thus nsuring that carries are propagated across the full double-length result. This short rogram is implemented in two consecutive clock cycles by activating the control ignals listed below, not all of which appear in Figure 5.2.

Scanned by CamScanner

Microprogramming makes control unit design more systematic by organizing control signals into formatted words (microinstructions). Since the control signals are embedded in a kind of low-level software—this is referred to as *firmware*—design changes can be easily made just by altering the contents of the control memory. On the negative side, microprogrammed control units are more costly to manufacture than hardwired units due to the presence of the control memory and its access circuitry. Microprogrammed units also tend to be slower because of the extra time required to fetch microinstructions from the control memory. RISC processors, with their emphasis on small, fast instruction sets, favor the use of hardwired control units.

CPU control units, both hardwired and microprogrammed, are often organized as (*micro*) *instruction pipelines* in order to improve their performance. As we saw in section 4.3.2, pipelining is a relatively low-cost way of increasing a processor's throughput by decomposing its operation into a sequence of relatively independent steps. Program control naturally involves a sequence of steps (instruction fetching and decoding, input operand fetching, operation execution, and result storage) that can be carried out concurrently with different instructions. Modern CPUs make extensive use of pipelines to increase their effective instruction execution rate [Stone 1993].

5.1.2 Hardwired Control

Next we examine the design of control units that use fixed logic circuits to interpret instructions and generate control signals from them.

Design methods. Control-unit design involves various trade-offs between the amount of hardware used, the speed of operation, and the cost of the design process itself. To illustrate these issues, we consider two systematic approaches to the design of hardwired controllers [Hayes 1993; Baranov 1994]. These methods are representative of those used in practice, but by themselves are suitable only for small control units such as might be encountered in simple RISC processors or application-specific controllers.

- Method 1: The *classical method* of sequential circuit design, which was discussed briefly in section 2.1.3. It attempts to minimize the amount of hardware, in particular, by using only $\lceil \log_2 P \rceil$ flip-flops to realize a P-state circuit.
- Method 2: An approach that uses one flip-flop per state and is known as the *one-hot method*. While expensive in terms of flip-flops, this method simplifies the design and debugging.

In practice, processor control units are often so complex that no one design method by itself can yield a satisfactory circuit at an acceptable cost. The most acceptable design may consist of several linked, but independently designed, sequential circuits.

State tables. The behavior required of a control unit, like that of any finite state machine, can be represented by a state table of the general type shown in Figure 5.4*a*. The rows of the state table correspond to the set of internal states $\{S\}$. These states are determined by the information stored in the machine at discrete points of time (clock cycles). Let X and Z denote the input and output variables

Scanned by CamScanner

	Inputs		
	I_1	I_2	I_m
$S_{1,1}, O_{1,1}$	$S_{1,2}, O_{1,2}$	...	$S_{1,m}, O_{1,m}$
$S_{2,1}, O_{2,1}$	$S_{2,2}, O_{2,2}$	...	$S_{2,m}, O_{2,m}$
		...	
$S_{n,1}, O_{n,1}$	$S_{n,2}, O_{n,2}$	...	$S_{n,m}, O_{n,m}$

(*a*)

	Inputs			
I_1	I_2		I_m	Outputs
$S_{1,1}$	$S_{1,2}$	...	$S_{1,m}$	O_1
$S_{2,1}$	$S_{2,2}$	...	$S_{2,m}$	O_2
		...		
$S_{n,1}$	$S_{n,2}$	...	$S_{n,m}$	O_n

(*b*)

5.4
tables for a finite-state machine: (*a*) Mealy type and (*b*) Moore type.

columns correspond to the combinations of the X signals that can be applied to machine and are denoted here by $\{I_j\}$. The entry in row S_i and column I_j has the $S_{i,j}, O_{i,j}$, where $S_{i,j}$ is the next state of the machine that results from the applica- of input combination I_j, and $O_{i,j}$ denotes the output signals that appear on Z ever the machine is in state S_i with input I_j applied. In general, an entry in the able defines a specific, one-cycle transition between two states.

Control units have a feature that favors a slightly different style of state table: output signal values often depend on the current state S_i only and so are inde- ent of the input combination I_j. If all outputs are of this type, the circuit is a *Moore machine*, in contrast with the more general *Mealy machine* of Fig- 5.4*a*. (These names honor G. H. Mealy and E. F. Moore who were early chers into finite-state machine theory [Mealy 1955; Moore 1956].) The state of Figure 5.4*a* becomes a Moore machine if for every row, i, we have $O_{i,j} =$ O_i for all j, $k = 1,2,\ldots,m$. In that case we can represent the machine's behavior more compact format of Figure 5.4*b*, where the output signals associated each row are placed in a separate column.

GCD processor. To illustrate the classical and one-hot approaches to control- design, we will apply them to a special-purpose processor that computes the common divisor $gcd(X,Y)$ of two positive integers X and Y; $gcd(X,Y)$ is as the largest integer that divides exactly into both X and Y. For example, $(12,18) = 6$, and $gcd(12,17) = 1$. It is customary to assume that $gcd(0,0) = 0$. We use a variant of Euclid's algorithm [Cormen, Leiserson, and Rivest 1990] alculate $gcd(X,Y)$. Figure 5.5 gives an HDL description of this method.

Scanned by CamScanner

```
gcd(in: X,Y; out: Z);
  register XR, YR, TEMPR;
  XR := X;                          (Input the data)
  YR := Y;
  while XR > 0 do begin
    if XR ≤ YR then begin           (Swap XR and YR)
      TEMPR := YR;
      YR := XR;
      XR := TEMPR; end
      XR := XR - YR;                (Subtract YR from XR)
  end
  Z := YR;                          (Output the result)
end gcd;
```

Figure 5.5
Procedure *gcd* to compute the g
common divisor of two numbers

The basic idea is to subtract the smaller of the two numbers from th repeatedly—recall that division corresponds to repeated subtraction—u obtain a number that divides the other. For example, with $X = 20$ and $Y =$ *gcd* algorithm proceeds as follows:

Conditions		Actions	
		$XR := 20;\ YR := 12;$	
$XR > 0$:	$XR > YR$:	$XR := XR - YR = 8;$	
$XR > 0$:	$XR \le YR$:	$YR := 8;\ XR := 12;$	$XR := XR - YR = 4;$
$XR > 0$:	$XR \le YR$:	$YR := 4;\ XR := 8;$	$XR := XR - YR = 4;$
$XR > 0$	$XR \le YR$:	$YR := 4;\ XR := 4;$	$XR := XR - YR = 0;$
$XR \le 0$:		$Z := 4;$	

Hence we conclude that $gcd(20,12) = 4$.

Analysis of the *gcd* procedure suggests that its datapath unit DP should a pair of registers *XR* and *YR* to store the corresponding variables, one c functional units to perform subtraction and magnitude comparison, and mu ers for data routing, as indicated in Figure 5.6. We do not need to include a for the "temporary" variable *TEMPR*, as we would in a typical programmed mentation, because we can read from and write to a register in the sam cycle. The swap operation can therefore be done without conflict in one cycl

$$X := Y, \quad Y := X;$$

The control unit CU generates control signals *Load XR* and *Load YR* each register independently with the input data *X* and *Y*. A control signal *Sel* routes *X* and *Y* to *XR* and *YR*, respectively. Another signal *Swap* controls th operation defined by (5.6), which requires routing the outputs of the *XR* a registers to each other's inputs. A final signal *Subtract* is assumed to contr subtraction $XR := XR - YR$ by routing the output of the subtracter to *XR*. The signals to CU are an asynchronous *Reset* signal, two comparison signals (XR and ($XR > 0$) generated by DP, and the usual, implicit clock signal.

We can identify a set of states for CU by examining the behavior defined i HDL specification (refer to Figure 5.5)—a simple process here, but one th

Scanned by CamScanner

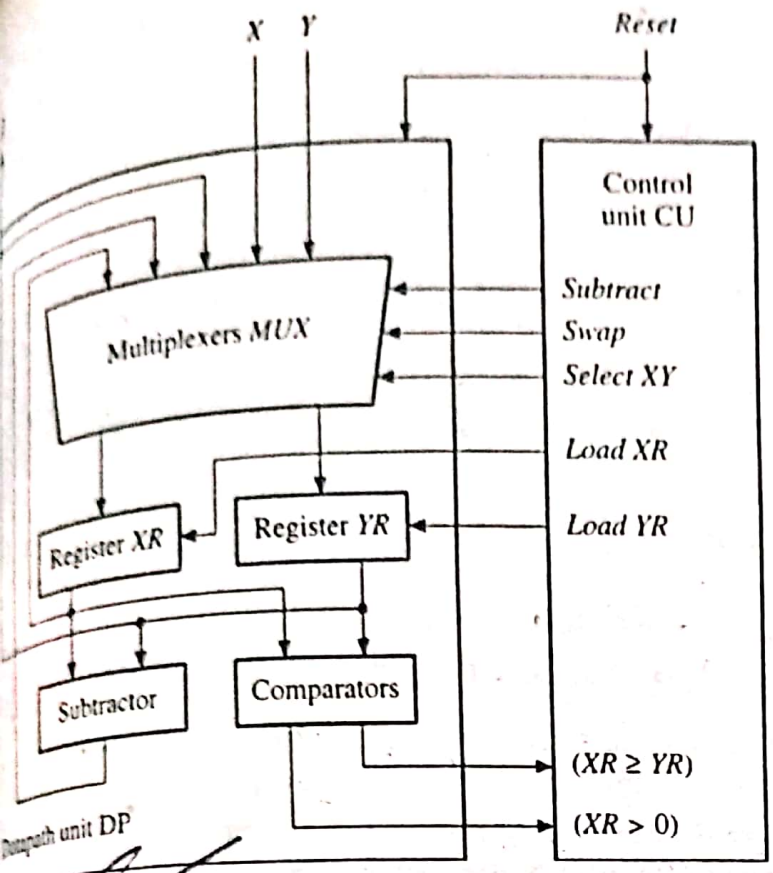


Figure 5.6
Hardware needed to implement the *gcd* procedure.

ious and error-prone in the case of large control units. A start state S_0 is entered when *Reset* becomes 1; this state also loads X and Y into the DP registers. The subsequent actions of the *gcd* processor are either a swap or a subtraction, for which we define the states S_1 and S_2, respectively. A final state S_3 is entered when gcd(X,Y) has been computed. Figure 5.7 gives a Moore-type state table defining the CU's behavior. Each state transition is deduced directly from the HDL description. If the input control signal $(XR > 0) = 0$, indicating that the **while** loop should be skipped, a transition is made from S_0 to S_3; this yields the first next-state entry in the top row of Figure 5.7. If, on the other hand, $(XR > 0) = 1$, the **while** loop is entered, and a transition is made to S_1 to perform a swap if $(XR \geq YR) = 0$; otherwise, the transition is to S_2 to perform a subtraction. The latter case defines to the third entry of the state table, whose input combination is $(XR > 0)(XR \geq YR) = 11$.

Inputs $(XR > 0)$ $(XR \geq YR)$	Outputs

Scanned by CamScanner

Since a subtraction always follows a swap, all next-state entries in the second row are S_2. The corresponding active outputs are the two register-load signals *Load YR* and *Load YR*, along with *Swap*, which route the outputs of XR and YR to YR and XR, respectively. The next states for state S_2 are the same as those for S_0; the active outputs are *Subtract*, which routes the output $XR - YR$ of the subtracter to XR, and *Load XR*. The final state S_3 is assumed to be a "dead" state that is unaffected by all inputs (except *Reset*) and produces no active outputs.

Classical method. The major steps of the classical design method are as follows:

1. Construct a P-row state table that defines the desired input-output behavior.
2. Select the minimum number p of D-type flip-flops and assign a p-bit binary code to each state.
3. Design a combinational circuit C that generates the primary output signals $\{z_j\}$ and the secondary outputs $\{D_i\}$ that must be applied to the flip-flops.

We now apply this method to the design of the control unit CU for the *gcd* processor. We have already constructed the necessary state table (Figure 5.7). Since there are four states, we require two flip-flops, whose outputs $D_1D_0 = y_1y_0$ define CU's internal states. We assign the binary patterns to the four states in the following obvious way:

$$\begin{aligned} S_0 &= 0\,0 \\ S_1 &= 0\,1 \\ S_2 &= 1\,0 \\ S_3 &= 1\,1 \end{aligned} \tag{5.7}$$

We note in passing that the state assignment pattern affects the complexity of the circuit in subtle ways.

At this point we can construct a binary version of the state table, the *excitation table*, as shown in Figure 5.8. The D flip-flop's characteristic equation $D_i^+(t+1) = D_i(t)$ defines the inputs D_1^+ and D_0^+ to the flip-flops. CU's combinational logic C can now be derived from the excitation table using any available manual or automatic method. Suppose, for instance, that we use two-level sum-of-products (SOP) minimization. It is easily checked that C is defined by the following SOP equations, which lead directly to the design of Figure 5.9. Note that all gates in an AND-OR SOP circuit can be changed to NANDs to produce a NAND-NAND realization of the original function.

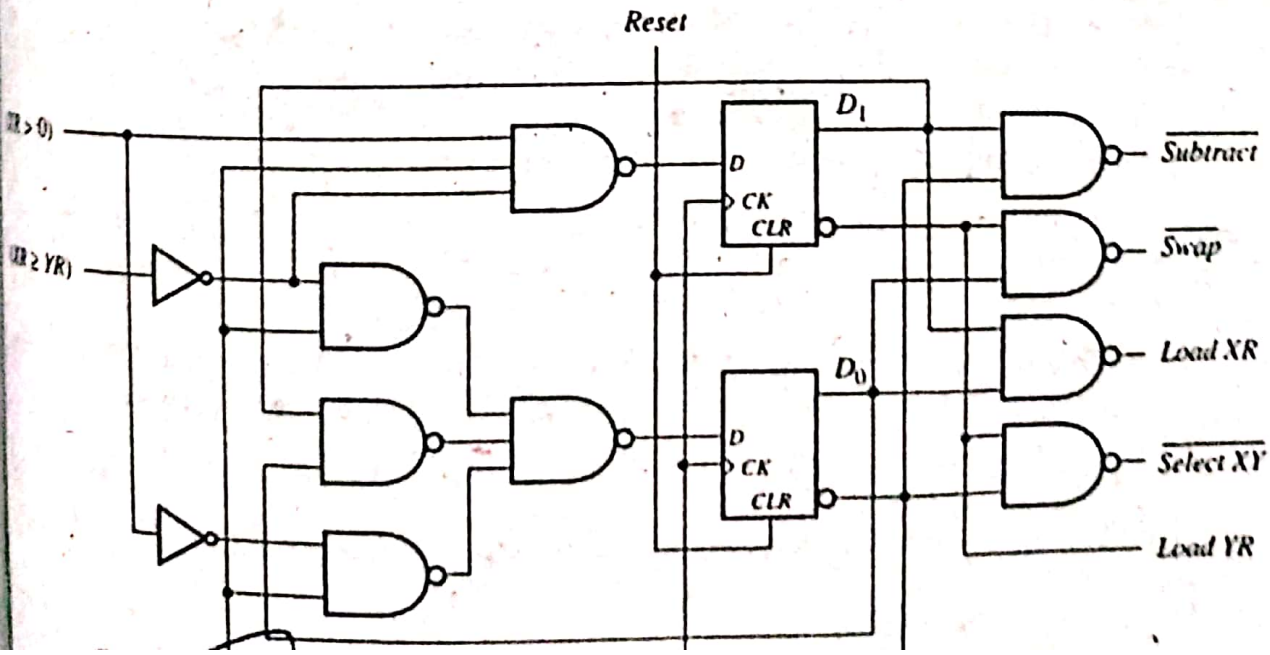
$$\begin{aligned} D_1^+ &= (\overline{XR > 0}) + (XR \geq YR) + D_0 \\ D_0^+ &= D_1 \cdot D_0 + (\overline{XR \geq XR}) \cdot \overline{D}_0 + (\overline{XR > 0}) \cdot \overline{D}_0 \\ Subtract &= D_1 \cdot \overline{D}_0 \\ Swap &= \overline{D}_1 \cdot D_0 \\ Select\ XY &= \overline{D}_1 \cdot \overline{D}_0 \\ Load\ XR &= \overline{D}_0 + \overline{D}_1 \\ Load\ YR &= \overline{D}_1 \end{aligned} \tag{5.8}$$

Scanned by CamScanner

Inputs		Present state		Next state		Outputs				
$(XR > 0)$	$(XR \geq YR)$	D_1	D_0	D_1^*	D_0^*	*Subtract*	*Swap*	*Select XY*	*Load XR*	*Load YR*
0	*d*	0	0	1	1	0	0	1	1	1
0	*d*	0	1	1	0	0	1	0	1	1
0	*d*	1	0	1	1	1	0	0	1	0
0	*d*	1	1	1	1	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1	1
1	0	0	1	1	0	0	1	0	1	1
1	0	1	0	0	1	1	0	0	1	0
1	0	1	1	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	1	1	1
1	1	0	1	1	0	0	1	0	1	1
1	1	1	0	1	0	1	0	0	1	0
1	1	1	1	1	1	0	0	0	0	0

Figure 5.8
Excitation table for the control unit of the *gcd* processor.

One-hot method. While the classical design method minimizes a control unit's memory elements, its effect on the amount of combinational logic *C* is less obvious. Furthermore, control units designed by this technique tend to have a complicated, "random" structure, which makes design debugging and subsequent maintenance of the circuit difficult. An alternative approach that simplifies the



Scanned by CamScanner

design process and gives C a regular and predictable structure, is the one-hot method, so called because its binary state assignment always contains a single 1, the "hot" bit—while all the remaining bits are 0. Thus the state assignment for a four-state machine like the *gcd* processor takes the following form:

$$S_0 = 0\,0\,0\,1$$
$$S_1 = 0\,0\,1\,0$$
$$S_2 = 0\,1\,0\,0$$
$$S_3 = 1\,0\,0\,0$$

In general, P flip-flops are needed to represent P states, so the one-hot method is restricted to fairly small values of P.

A key feature of this technique is that the next-state and output equations have a simple, systematic form and can be written down directly from the control unit's original symbolic state table. Because the binary pattern assigned to each state is in effect, fully decoded, we can find out whether the machine is in state S_i merely by inspecting the corresponding hot state variable D_i. The classical method requires us to check all state variables to get this information.

Suppose that state S_i in a one-hot design has the hot variable D_i. Further, suppose that $I_{j,1}, I_{j,2}, \ldots, I_{j,n_j}$ denote all input combinations that cause a state transition from S_j to S_i. Then each AND combination of the form $D_j \cdot I_{j,k}$ must make $D_i = 1$. Hence, considering all such combinations that cause transitions to S_i, we can write

$$D_i^+ = D_1(I_{1,1} + I_{1,2} + \ldots + I_{1,n_1}) + D_2(I_{2,1} + I_{2,2} + \ldots + I_{2,n_2}) + \ldots \quad (5.10)$$

This immediately yields the SOP form

$$D_i^+ = D_1 I_{1,1} + D_1 I_{1,2} + \ldots + D_1 I_{1,n_1} + D_2 I_{2,1} + D_2 I_{2,2} + \ldots + D_2 I_{2,n_2} + \ldots$$

which is practical to implement by an AND-OR or NAND-NAND circuit, provided that each state transition is determined by relatively few states and input variables, as is common in control-unit behavior. Equation (5.10) can also lead directly to fairly simple factored forms. Consider the state table of Figure 5.7 for the *gcd* processor's CU. State S_1 appears as a next state only for S_0 and S_2, in each case with the input combination $(XR > 0)\cdot(\overline{XR \geq XR})$. Hence (5.10) becomes

$$D_1^+ = D_0 \cdot (XR > 0) \cdot (\overline{XR \geq XR}) + D_2 \cdot (XR > 0) \cdot (\overline{XR \geq XR})$$
$$= (D_0 + D_2) \cdot (XR > 0) \cdot (\overline{XR \geq XR})$$

The primary output equations are even easier to derive for one-hot designs. If output signal z_k is 1 (active) only in rows k,h for $h = 1,2,\ldots,m_k$, then we have

$$z_k = D_{k,1} + D_{k,2} + \ldots + D_{k,m_k} \quad (5.11)$$

De Morgan's law of Boolean algebra allows us to rewrite this OR equation as

$$z_k = \overline{\overline{D}_{k,1}\overline{D}_{k,2}\ldots\overline{D}_{k,m_k}}$$

in which form it can be generated by a single NAND whose inputs are the complemented outputs of the flip-flops. In the *gcd* processor case, output *Load YR* = 1 in states S_0 and S_1 only; therefore

$$Load\ YR = D_0 + D_1 = \overline{\overline{D}_0\overline{D}_1}$$

Scanned by CamScanner

entire set of next-state and output equations obtained by applying (5.10) to the gcd processor's CU follows.

$$D_0^+ = 0$$

$$D_1^+ = D_0 \cdot (XR > 0) \cdot (\overline{XR \geq XR}) + D_2 \cdot (XR > 0) \cdot (\overline{XR \geq XR})$$

$$D_2^+ = D_0 \cdot (XR > 0) \cdot (XR \geq XR) + D_1 + D_2 \cdot (XR > 0) \cdot (XR \geq XR)$$

$$D_3^+ = D_0 \cdot (\overline{XR > 0}) + D_2 \cdot (\overline{XR > 0}) + D_3$$

$$Subtract = D_2$$

$$Swap = D_1$$

$$Select\ XY = D_0$$

$$Load\ XR = D_0 + D_1 + D_2$$

$$Load\ YR = D_0 + D_1$$

ND implementation of these equations appears in Figure 5.10. Note that the nous *Reset* line must set D_0 to 1 and all other state variables to 0.

e steps of the one-hot design method for a Moore machine can be summa- s follows:

stract a P-row state table that defines the desired input-output behavior.

sociate a separate D-type flip-flop D_i with each state S_i, and assign the P-bit e-hot binary code $D_1, D_2, \ldots, D_{i-1}, D_i, D_{i+1}, \ldots, D_P = 0,0,\ldots,0,1,0,\ldots,0$ to S_i.

sign a combinational circuit C that generates the primary and secondary out- t signals $\{D_i\}$ and $\{z_k\}$, respectively. D_i^+ is defined by the logic equation

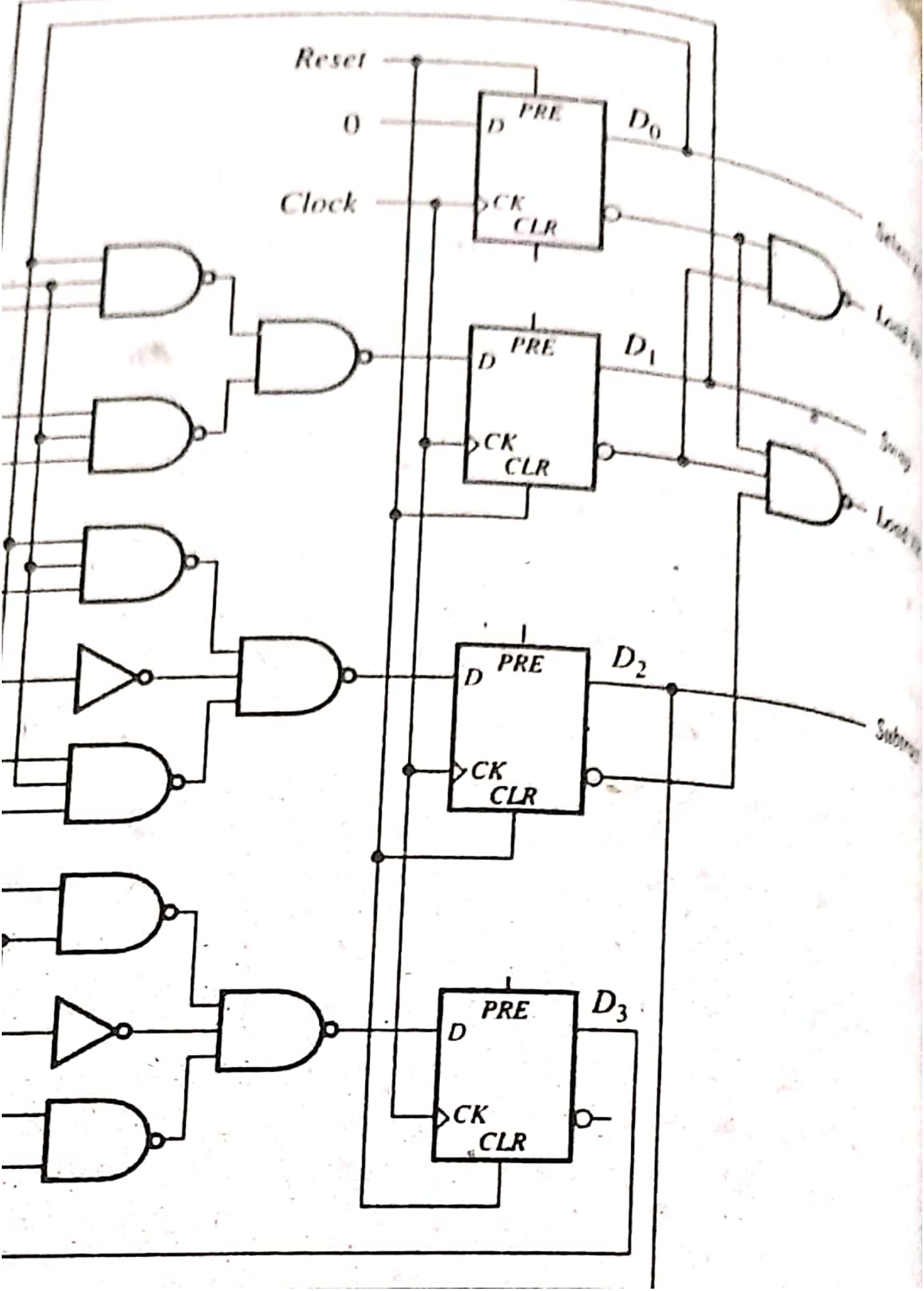
$$D_i^+ = \sum_{i=1}^{P} D_i(I_{j,1} + I_{j,2} + \ldots + I_{j,n_j})$$

here $I_{j,1}, I_{j,2}, \ldots, I_{j,n_j}$ denote all input combinations that cause a transition from S_j S_i. If $z_k = 1$ (active) only in rows k,h for $h = 1,2,\ldots,m_k$, then z_k is defined by

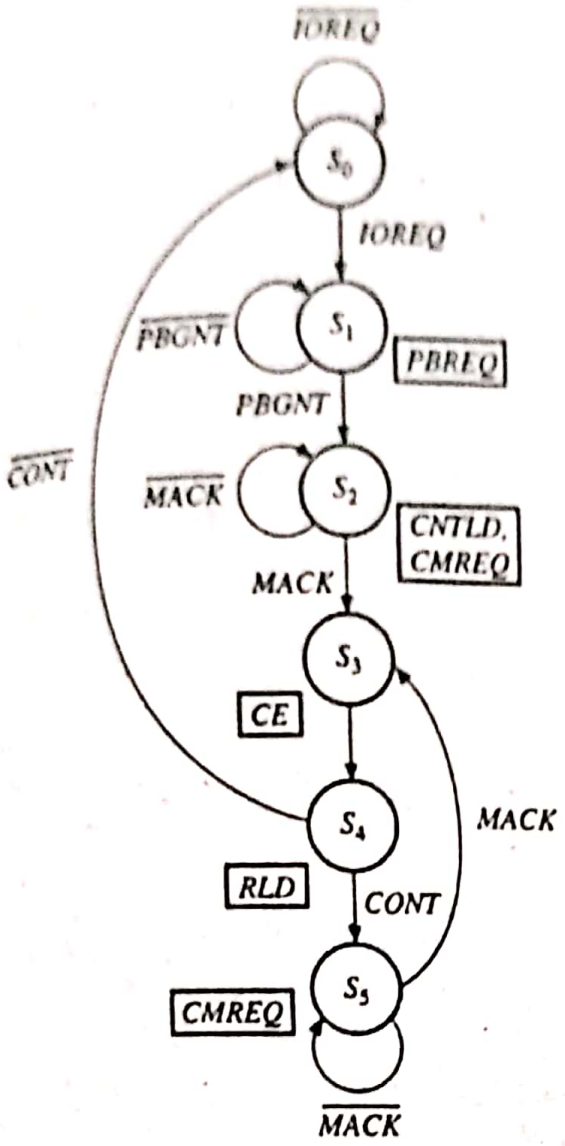
$$z_k = D_{k,1} + D_{k,2} + \ldots + D_{k,m_k} = \overline{\overline{D}_{k,1}\overline{D}_{k,2}\ldots\overline{D}_{k,m_k}}$$

We next present an example that illustrates the application of the one-hot hod to a computer's IO interface, specifically to a direct-memory access MA) controller, which handles data transfers between main memory and high- ed IO devices. (DMA communication is discussed in Chapter 7.)

Scanned by CamScanner



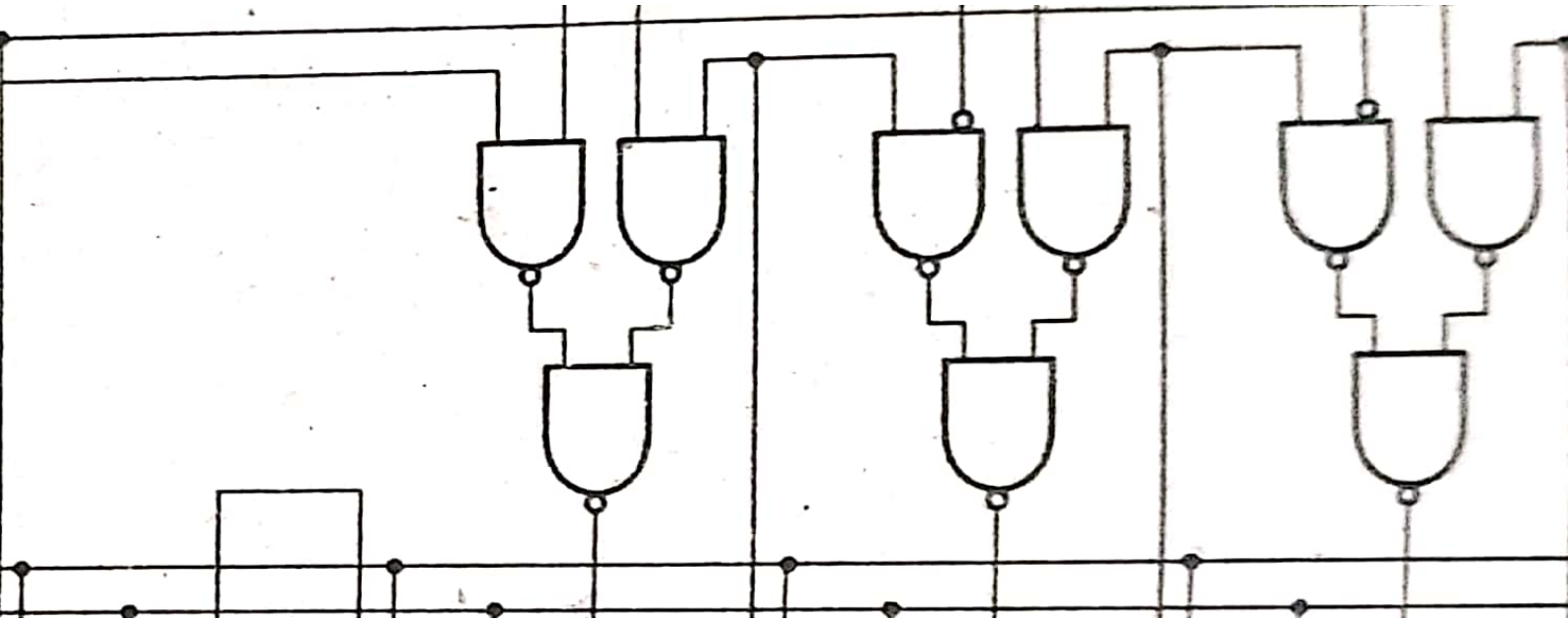
Scanned by CamScanner



(a)

Inputs				Present state	Next state	Outputs				
IOREQ	CONT	MACK	PBGNT			PBREQ	CNTLD	CMREQ	RLD	CE
d	d	d	d	S_0	S_0	0	0	0	0	0
d	d	d	d	S_0	S_1	0	0	0	0	0
d	d	d	0	S_1	S_1	1	0	0	0	0
d	d	d	1	S_1	S_2	1	0	0	0	0
d	d	0	d	S_2	S_2	0	1	1	0	0
d	d	1	d	S_2	S_3	0	1	1	0	0
d	d	d	d	S_3	S_4	0	0	0	0	1
d	0	d	d	S_4	S_0	0	0	0	1	0
d	1	d	d	S_4	S_5	0	0	0	1	0
d	d	0	d	S_5	S_5	0	0	1	0	0
d	d	1	d	S_5	S_3	0	0	1	0	0

Scanned by CamScanner



Scanned by CamScanner

$$D_4^* = D_3$$

$$D_5^* = D_4 \cdot CONT + D_5 \cdot \overline{MACK}$$

output equations are also immediately obtained from Figure 5.11.

$$CE = D_3$$

$$CMREQ = D_2 + D_5$$

$$CNTLD = D_2$$

$$RLD = D_4$$

$$PBREQ = D_1$$

ure 5.12 shows an all-NAND circuit derived from these equations. Note the regular ucture of the combinational logic, which is typical of one-hot designs. An equivalent ssical design has three flip-flops but a much more irregular combinational part.

Design Examples

ction presents some examples to illustrate the foregoing methods for design- rdwired control units. We will use these examples again in our discussion of programmed control.

ltiplier control. First consider the design of a control unit CU for the twos- lement (Robertson) multiplier introduced in Example 4.2 (section 4.1.2). The diagram of the multiplier's datapath unit DP (Figure 4.12) is redrawn in ded form in Figure 5.13 to show a set of control points, which represent actly the control signals and associated logic circuits needed to link CU and These control signals are derived from the multiplication algorithm in Figure and are listed in Figure 5.14. In general, a control point can be associated with distinct action (register-transfer operation) op_i appearing in the algorithm implemented. Its enabling control signal c_i is inserted into the component or connections associated with op_i. Operations that take place simultaneously be able to share control signals. (Procedures to eliminate redundant control als are considered later.) The statement labeled BEGIN in Figure 4.13, for nce, requires the registers A, COUNT, and F to be reset simultaneously to the ero state. A single control signal c_{10} is therefore provided for this purpose. It be connected directly to the CLEAR inputs of the three registers in question, so additional logic is needed to implement the c_{10} control point. Control signals c_8 c_9 transfer a data word from the input bus INBUS to registers Q and M, respec- ly, and are shown in the corresponding data paths of Figure 5.13; these signals y be connected to the registers' (parallel) LOAD inputs. Control signal c_5 serves hange the function performed by the parallel adder from addition to subtraction the correction step; it also resets Q[0] to 0. The remaining control signals of ure 5.14 are defined similarly. Figure 5.13 introduces a control signal called OUNT7, which is set to 1 when COUNT = 111_2 and is set to 0 otherwise. OUNT7, the right-most bit Q[0] of the multiplier register Q and the external EGIN signal serve as the primary inputs to CU.

Scanned by CamScanner