CPU to access information stored anywhere in the hierarchy as if it were in main memory. We conclude by discussing the factors that influence memory system design.

7.1 Introduction: The Components of the Memory System

We begin with a short discussion of the parameters that are used to characterize main memory capacity and organization. Then we briefly discuss the characteristics of the various components that comprise the memory hierarchy.

7.1.1 Main Memory Size and Organization

Figure 7.1 and Table 7.1 show the parameters that characterize the interface between the CPU and main memory. Main memory can be pictured as being composed of a large collection of registers, all the same word size, s bits, which we will refer to as *memory words*. Note that the CPU word size, w, may be different than the main memory word size, s. In general, s-bit words are the smallest units that can be accessed in memory. In many processors, words that are fragments of w can be requested and processed. For example, the PowerPC G4 chip has 32-bit registers, uses byte addresses, but can read or write 8-, 16-, 32-, or 64-bit values from or to main memory. A machine with m-bit addresses has a memory capacity of 2^m s-bit memory words, yielding a memory bit-capacity of $2^m \times s$.

For cost reasons, a given memory system may not transmit the entire w bits of a word at once. It may transmit portions of the CPU word serially, where they are reassembled by the CPU. We define the size of the largest word actually transmitted as a unit as b. On the other hand, when an s-bit word, $s < w$, is accessed by the CPU, the memory system may

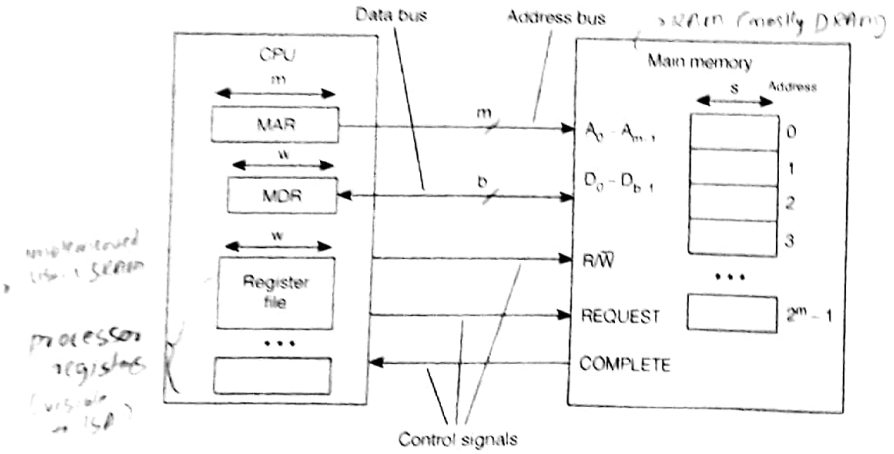


Fig. 7.1 The CPU-Memory Interface

Table 7.1 Some Memory Properties

Symbol	Definition	Intel 8088	Intel 8086	Intel Pentium	PowerPC G4
w	CPU word size	16 bits	16 bits	16 or 32	64 bits
	Bits in a logical memory address	20 bits	20 bits	32	32 bits
s	Bits in smallest addressable unit	8 bits	8 bits	8 bits	8 bits
b	Data bus size	8 bits	16 bits	64 bits	64 bits
2^m	Memory capacity, s-sized words	2^{20} words	2^{20} words	2^{32} words	2^{32} words
$2^m \times s$	Memory bit capacity	$2^{20} \times 8$ bits	$2^{20} \times 8$ bits	$2^{32} \times 8$ bits	$2^{32} \times 8$ bits

transmit an entire w-bit word, leaving the CPU hardware to extract the desired s-bit fragment. This process is transparent to the machine language program.

The CPU makes a memory access request by asserting REQUEST while indicating a read or write on the R/$\overline{W}$ line. The memory system indicates that its process is complete by asserting COMPLETE. Table 7.1 shows three examples from the Intel 8086 family: the 8088, the 8086, and the Pentium processors. The first two processors have a 16-bit CPU word size, 8-bit memory word size, and 20-bit addresses. The 8086 can transmit an entire 16-bit CPU word in 1 bus cycle over its 16-bit data bus. The 8088 was designed, for cost-saving reasons, with only an 8-bit data bus, and therefore must transmit the word as two separate 8-bit bytes, requiring 2 bus cycles. This is yet another of the time (2 bus cycles versus 1) and space (8-bit bus versus 16-bit bus) trade-offs that are possible in system design. The Pentium processor shown in the table remains binary-compatible with the two former members, but operations can be performed on double (32-bit) words, and the bus width has been extended to 64 bits.

Big-Endian and Little-Endian Word-Ordering Schemes. Jonathan Swift, in *Gulliver's Travels*, describes a war between those who believed that an egg should be broken from the little end, the Little Endians, and those who believed that the only proper way to break an egg was from the big end, the Big Endians. These terms have become associated with two possible conventions for storing the w/s bytes of a word when $w/s = n > 1$. *Little-endian* organization stores the least significant byte, the little end, at the lower byte address and the most significant byte at the higher byte address. *Big-endian* organization puts the most significant byte, the big end, at the first byte address and the least significant byte after it. Figure 7.2 shows the two conventions illustrated for a 32-bit PowerPC G4 word.

Little-endian

big-endian

Both storage strategies have their adherents, but in fact there is little difference between them, and various machines use various byte-ordering conventions. The entire Intel x86 family of processors are little-endian machines, and the 16-bit PDP11 is a big-endian machine. The byte-ordering conventions are important under certain circumstances; for example, when using a debugger, it is important that the user know the byte ordering when examining memory contents by byte. Knowledge of byte ordering is also of obvious importance when multibyte words are communicated between hosts from different vendors. The PowerPC has machine instructions to alter the endian-ness of words being transmitted to its CPU.

CPU to access information stored anywhere in the hierarchy as if it were in main memory. We conclude by discussing the factors that influence memory system design.

7.1 Introduction: The Components of the Memory System

We begin with a short discussion of the parameters that are used to characterize main memory capacity and organization. Then we briefly discuss the characteristics of the various components that comprise the memory hierarchy.

7.1.1 Main Memory Size and Organization

Figure 7.1 and Table 7.1 show the parameters that characterize the interface between the CPU and main memory. Main memory can be pictured as being composed of a large collection of registers, all the same word size, s bits, which we will refer to as *memory words*. Note that the CPU word size, w, may be different than the main memory word size, s. In general, s-bit words are the smallest units that can be accessed in memory. In many processors, words that are fragments of w can be requested and processed. For example, the PowerPC G4 chip has 32-bit registers, uses byte addresses, but can read or write 8-, 16-, 32-, or 64-bit values from or to main memory. A machine with m-bit addresses has a memory capacity of 2^m s-bit memory words, yielding a memory bit-capacity of $2^m \times s$.

For cost reasons, a given memory system may not transmit the entire w bits of a word at once. It may transmit portions of the CPU word serially, where they are reassembled by the CPU. We define the size of the largest word actually transmitted as a unit as b. On the other hand, when an s-bit word, $s < w$, is accessed by the CPU, the memory system may

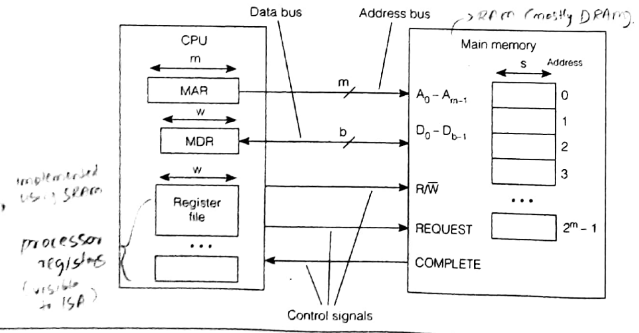


Fig. 7.1 The CPU-Memory Interface

Table 7.1 Some Memory Properties

Symbol	Definition	Intel 8088	Intel 8086	Intel Pentium	PowerPC G4
w	CPU word size	16 bits	16 bits	16/32	64 bits
m	Bits in a logical memory address	20 bits	20 bits	32	32 bits
s	Bits in smallest addressable unit	8 bits	8 bits	8 bits	8 bits
b	Data bus size	8 bits	16 bits	64 bits	64 bits
2^m	Memory capacity, s-sized words	2^{20} words	2^{20} words	2^{32} words	2^{32} words
$2^m \times s$	Memory bit capacity	$2^{20} \times 8$ bits	$2^{20} \times 8$ bits	$2^{32} \times 8$ bits	$2^{32} \times 8$ bits

transmit an entire w-bit word, leaving the CPU hardware to extract the desired s-bit fragment. This process is transparent to the machine language program.

The CPU makes a memory access request by asserting REQUEST while indicating a read or write on the R/$\overline{\text{W}}$ line. The memory system indicates that its process is complete by asserting COMPLETE. Table 7.1 shows three examples from the Intel 8086 family: the 8088, the 8086, and the Pentium processors. The first two processors have a 16-bit CPU word size, 8-bit memory word size, and 20-bit addresses. The 8086 can transmit an entire 16-bit CPU word in 1 bus cycle over its 16-bit data bus. The 8088 was designed, for cost-saving reasons, with only an 8-bit data bus, and therefore must transmit the word as two separate 8-bit bytes, requiring 2 bus cycles. This is yet another of the time (2 bus cycles versus 1) and space (8-bit bus versus 16-bit bus) trade-offs that are possible in system design. The Pentium processor shown in the table remains binary-compatible with the two former members, but operations can be performed on double (32-bit) words, and the bus size has been extended to 64 bits.

Big-Endian and Little-Endian Word-Ordering Schemes. Jonathan Swift, in *Gulliver's Travels*, describes a war between those who believed that an egg should be broken from the little end, the Little Endians, and those who believed that the only proper way to break an egg was from the big end, the Big Endians. These terms have become associated with two possible conventions for storing the w/s bytes of a word when $w/s = n > 1$. *Little-endian* organization stores the least significant byte, the little end, at the lower byte address and the most significant byte at the higher byte address. *Big-endian* organization puts the most significant byte, the big end, at the first byte address and the least significant byte after it. Figure 7.2 shows the two conventions illustrated for a 32-bit PowerPC G4 word.

Little-endian

big-endian

Both storage strategies have their adherents, but in fact there is little difference between them, and various machines use various byte-ordering conventions. The entire Intel x86 family of processors are little-endian machines, and the 16-bit PDP11 is a big-endian machine. The byte-ordering conventions are important under certain circumstances; for example, when using a debugger, it is important that the user know the byte ordering when examining memory contents by byte. Knowledge of byte ordering is also of obvious importance when multibyte words are communicated between hosts from different vendors. The PowerPC has machine instructions to alter the endian-ness of words being transmitted to its CPU.

	CPU Word			
CPU word bit locations	b31 ... b24	b23 ... b16	b15 ... b8	b7 ... b0
Big-endian byte addresses	0	1	2	3
Little-endian byte addresses	3	2	1	0

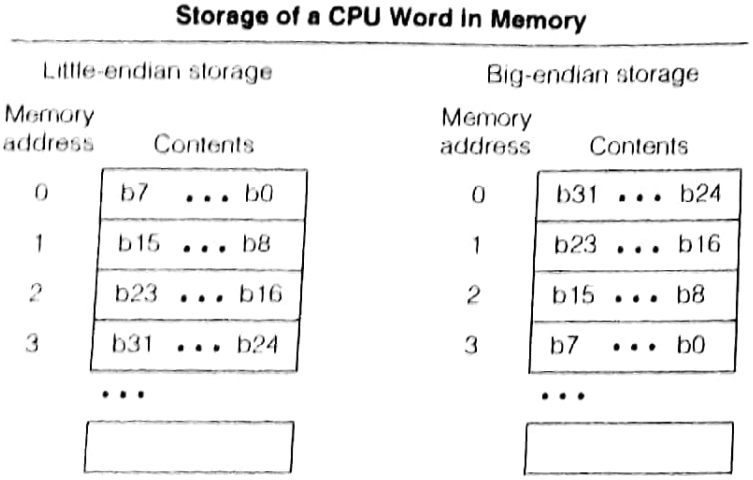


Fig. 7.2 Big-Endian and Little-Endian Storage of a PowerPC 32-Bit Word

random access memory, RAM

read-only memory, ROM

RAM and ROM. The abbreviation *RAM* stands for *random access memory*—memory in which all cells can be accessed in equal time. By convention, this term is reserved for semiconductor memory that can be written to as well as read from. Memory that has been preprogrammed and can only be read from is referred to as *ROM*, *read-only memory*. These terms are somewhat deceiving on the surface, since both RAM and ROM are random access; that is, values can be accessed in the same amount of time regardless of their actual physical location in the memory. This is in contrast to disk memory, for example, where time to access a given value depends on where the read-write head of the disk is located at the precise time an access is requested, and where the particular word is located on the disk. With this *caveat* in mind, we will use the terms RAM and ROM in their generally accepted senses—RAM is semiconductor memory that can be read from and written to; ROM can only be read.

Memory Operations: READ and WRITE. The memory operations READ and WRITE can be viewed as functions similar to the functions of programming languages, invoked by the CPU, with in and out parameters. The memory functions *as viewed from the standpoint of the memory system* are mirror images, with **in** replacing **out**, and **out** replacing **in**.

READ (**in**: Addr; **out:** Word, Completion_signal)
On Entry: MAR contains address to be read from.
On Exit: Completion_signal is true and MDR contains Word stored at Addr, of size *w* bits.

	CPU Word			
CPU word bit locations	b31 ... b24	b23 ... b16	b15 ... b8	b7 ... b0
Big-endian byte addresses	0	1	2	3
Little-endian byte addresses	3	2	1	0

Storage of a CPU Word in Memory

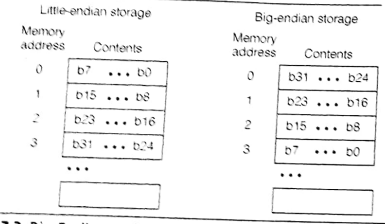


Fig. 7.2 Big-Endian and Little-Endian Storage of a PowerPC 32-Bit Word

random access memory, RAM

read-only memory, ROM

RAM and ROM. The abbreviation *RAM* stands for *random access memory*—memory in which all cells can be accessed in equal time. By convention, this term is reserved for semiconductor memory that can be written to as well as read from. Memory that has been preprogrammed and can only be read from is referred to as *ROM*, *read-only memory*. These terms are somewhat deceiving on the surface, since both RAM and ROM are random access; that is, values can be accessed in the same amount of time regardless of their actual physical location in the memory. This is in contrast to disk memory, for example, where time to access a given value depends on where the read-write head of the disk is located at the precise time an access is requested, and where the particular word is located on the disk. With this *caveat* in mind, we will use the terms RAM and ROM in their generally accepted senses—RAM is semiconductor memory that can be read from and written to; ROM can only be read.

Memory Operations: READ and WRITE. The memory operations READ and WRITE can be viewed as functions similar to the functions of programming languages, invoked by the CPU, with in and out parameters. The memory functions *as viewed from the standpoint of the memory system* are mirror images, with **in** replacing **out**, and **out** replacing **in**.

READ (**in**: Addr; **out:** Word, Completion_signal)
On Entry: MAR contains address to be read from.
On Exit: Completion_signal is true and MDR contains Word stored at Addr, of size w bits.

Table 7.2 Memory Performance Parameters

Symbol	Definition	Units	Meaning
t_a	Access time	time	Time to access a memory word
t_c	Cycle time	time	Time from start of read to start of next
k	Block size	words	Number of words per block
b	Bandwidth	words/sec.	Word transmission rate
ω	Latency	time	Time to access first of a sequence of words
t_l	Block access time	time	Time to access entire block from start of read
$t_b = t_l + k/\omega$			

WRITE (**in**: Addr, Word; **out:** Completion_signal)
On Entry: MAR contains address to be written to; MDR contains Word of size w bits:
On Exit: Completion_signal is true; Word as been written to address Addr.

The CPU reads from the memory by asserting READ and the address from which to read. The memory system responds by loading the data lines with the data stored at Address, and asserts the completion signal. The write cycle is similar, except that the processor loads the data to be written in the MDR and asserts WRITE. The memory asserts the completion function after the value has been stored. The completion signal may be indicated in texts and timing diagrams with signal names such as Done, WAIT, ACKNOWLEDGE, READY, or ACCEPT.

7.1.2 Memory Performance Parameters

The most important memory performance parameters are access time, t_a, the time interval from the start of a read until the assertion of the memory completion signal, and cycle time, t_c, the minimum time interval from the start of a read or write to the start of the next memory operation. The cycle time may be slightly longer than the access time, due to various hardware housekeeping tasks that must be performed by the memory.

Some memories read or write values not as individual words, but in blocks of k words. In these memories, there will be a *latency*, t_l, in accessing the first word of the block, that differs from the *bandwidth*, ω, the rate at which the words can be transmitted, in words per second, once the first word in the block is available. Thus the access time for the entire block, $t_b = t_l + k/\omega$. These parameters are summarized in Table 7.2.

memory latency
memory bandwidth

7.2 RAM Structure: The Logic Designer's Perspective

This section provides an overview of the structure of RAM cells and cell arrays, or chips. We begin with a somewhat idealized view of an individual RAM cell, using logic gates to show functionality. We then show some of the many ways that the cells can be assembled

into arrays as integrated circuits, or chips. We discuss the various ways of implementing the RAM cell array. We then discuss the timing of the read and write operations, and descend to the transistor level to describe the actual physical structure of static and dynamic RAM. *Static RAM*, or *SRAM*, retains values stored in it as long as power is applied to the RAM. *Dynamic RAM*, sometimes referred to as *DRAM*, retains its contents for only a few milliseconds, and thus must be periodically *refreshed*, that is, have its value restored to it. We conclude with a discussion of ROM.

Static RAM, SRAM
Dynamic RAM, DRAM
refresh

7.2.1 Memory Cells and Cell Arrays

Figure 7.3 shows the functional behavior of a static RAM cell. The cell is shown as a clocked D latch with added controls for selecting, reading, and writing to the cell.

The cell's contents are read by asserting R/$\overline{W}$ and Select, whereupon the value stored in the cell appears at DataOut. In this implementation the output is shown as tri-state. This permits many cells in an array to be connected to a single data-bit line. Data is written to the cell by applying the value to be written to DataIn while asserting W and Select. Notice that in this implementation the actual storage cell requires only two NOR gates, while access control hardware requires five gates—the cost of control exceeds the cost of storage. Sections 7.2.3 and 7.2.5 discuss ways to reduce the cost of the selection hardware and the cost of the individual storage cell.

Figure 7.4 shows an 8-bit register implemented as a one-dimensional array of these cells. The figure shows the data bus being driven by tri-state buffers. Strictly speaking, these buffers would not be required, since the output of each cell is already buffered; they are shown in the figure to indicate buffering of the register from the data bus.This cell structure can be viewed as a one-dimensional, or 1-D, memory, since in terms of cells it has only one dimension, length.

1-D memory

Figure 7.5 shows an implementation of a 2-D array of memory cells. An n-bit to 2^n-bit *address decoder*, a 2–4 decoder in the figure, selects all of the cells in one of the 2^n rows. The R/$\overline{W}$ line selects the given row for reading or writing. The figure also shows the cell

2-D memory
memory address decoder

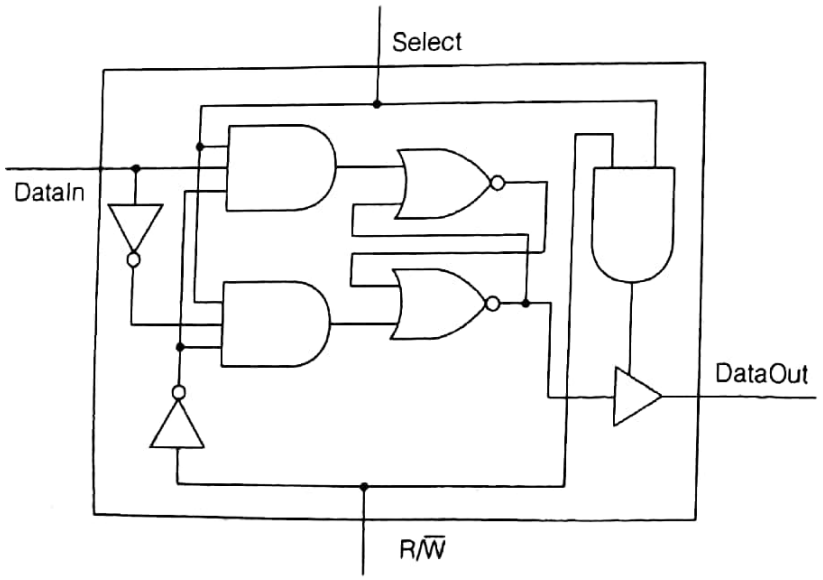


Fig. 7.3 Conceptual Structure of a Memory Cell

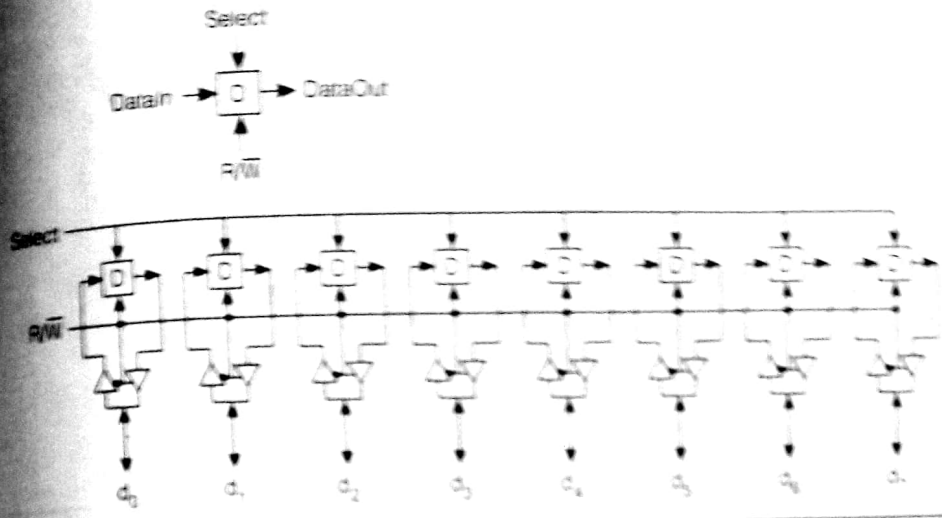


Fig. 7.4 An 8-Bit Register as a 1-D RAM Array

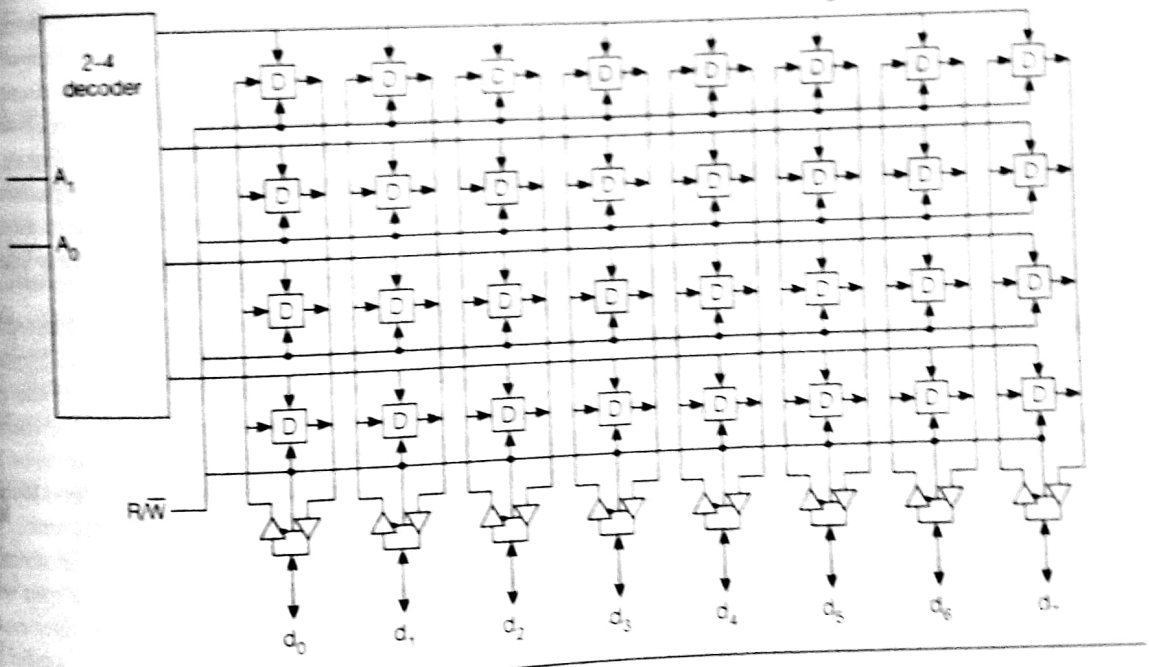


Fig. 7.5 A 4 × 8 2-D Memory Cell Array

array interfaced to a bidirectional data bus, d_0–d_7, through pairs of tri-state buffers that provide access to the data bus for reading or writing as determined by the R/$\overline{W}$ line.

Other implementations may use two unidirectional data buses, with a corresponding change to the R/$\overline{W}$ drivers at the bottom of the figure. They may also employ other combinations of control signals. To recapitulate, the row decoder selects one entire row of cells, through the word line. The R/$\overline{W}$ line specifies whether the row will be read or written, and also

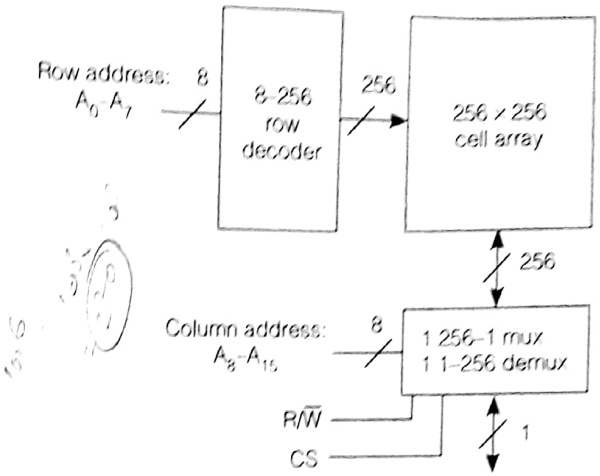


Fig. 7.6 A 64 K × 1 Static RAM Chip

controls access to the data bus through the buffers at the bottom of the figure. This kind of cell array is referred to as a 2-D array because of the arrangement of cells in rows and columns.

Designs such as those shown in Figure 7.5 are not actually used for RAM chips of any appreciable size, however, because of the impracticability of building large row decoders. For example, a 1 MB × 8 static RAM employing the design above would require a 20-to-2^{20} line decoder. Such a decoder would require 2^{20} gates, each with a fan-in of 20. Figure 7.6 shows a practical solution to this problem. The figure shows a 64 K × 1 chip organized as an array of 256 × 256 1-bit storage cells. RAM chips are most often designed to be only 1 bit wide, because of the savings in pins. A 64 K × 1 cell array requires 16 address lines, a read/write line, R/$\overline{W}$, a chip select line, CS, and only a single data line.

row decoder

Notice how the address lines have been split—the low-order eight address lines select 1 of 256 rows using an 8-to-256 line *row decoder;* thus the selected row contains 256 bits. The high-order eight address lines select one of those 256 bits. The 256 bits in the row selected flow through a 256-to-1 line multiplexer on a read. On a memory write, the incoming bit flows through a 1-to-256 line demultiplexer that selects the correct column of the 256 possible columns. The square 256:256 aspect ratio fits well with IC design and layout, and practical designs do not usually deviate by more than a factor of 2 from this ratio.

colum select

The particular row and column configuration shown in Figure 7.6 is only one of many possibilities. For example, the row and column address lines could have been split 9–7 instead of 8–8; the cell array could be composed of 4 64 × 256 arrays; in the latter case, there would then be four column multiplexers and demultiplexers, each of which would be 1-to-64 line. Figure 7.7 depicts the layout of such a chip. Since there are four independent 2-D arrays in the chip, it is referred to as a 2½-D array.

2½-D memory

3-D memory

It is also possible to build three-dimensional (3-D) arrays of memory cells. Such an implementation requires an implementation domain that supports an arrangement of memory cells in 3-D space. In the past, small magnetic cores were used as memory elements. These ring-shaped cores naturally supported a 3-D architecture, as select wires could be routed through the cores from all three spatial dimensions. Selection of the third dimension involves splitting the address bits, *m,* into *three* fields, the first selecting the row, the second selecting a column, and the third selecting a particular *plane.* 3-D memory systems

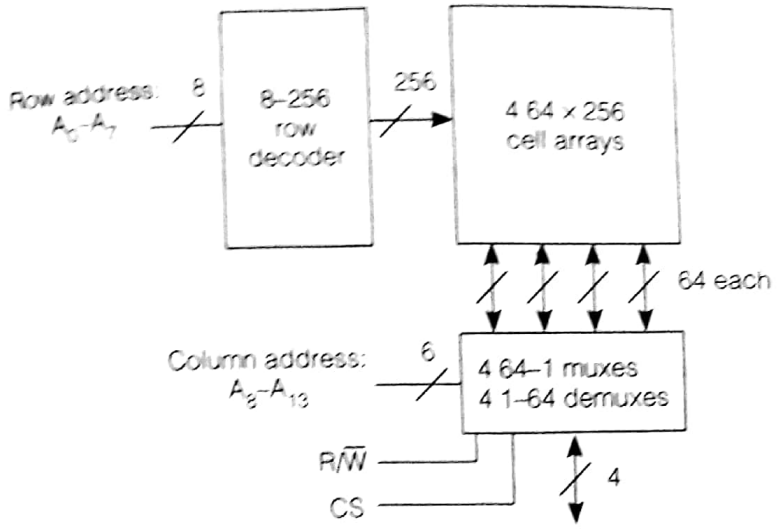


Fig. 7.7 A 16 K × 4 Static RAM Chip

may reemerge upon the computing scene. Considerable research effort is being expended on spatially accessed solid state memories implemented in crystals of lithium niobate and other photorefractive materials.

7.2.2 Advanced Topic: RAM Chip Costs and Matrix Decoders

The following sections briefly treat the cost of RAM chips and the design of matrix decoders. They may be omitted without loss of continuity.

Hardware Cost Parameters. RAM cost has two components: the cost of the memory cells themselves, and the cost of the hardware required to actually select a cell or group of cells within the cell array. In this discussion we express cost in terms of gate count. We begin by calculating the cost of a memory similar to that of Figure 7.5, with an m-bit address and a word size of s bits.

The cost of the memory in Figure 7.5 is composed of the cost of the memory cells themselves plus the cost of the decoders and drivers for columns and rows of the array. We ignore the cost of the word and bit lines inside the cell array itself, as well as costs of pins, package, socket, and bus exterior to the cell array. If these costs are considerable, as they well may be in IC designs, then they can be added in a manner analogous to the terms shown below.

The following equation relates the cost of the memory, C, to the sum of the costs of the cells, plus the cost of the row decoder plus the cost of the column drivers.

$$\begin{aligned} C &= C_{\text{Cells}} + C_{\text{RowDecoder}} + C_{\text{ColDrivers}} \\ &= C_C \times 2^m \times s + C_{\text{RDG}} \times 2^m + 2 \times s \times C_{\text{CDG}} \end{aligned} \qquad \text{[Eq. 7.1]}$$

The cost of the cell array, C_{Cells}, is equal to the cost per cell, C_C, times the number of cells, $2^m \times s$. The cost of the row decoder is equal to the cost of a row decoder gate, C_{RDG}, times the number of row decoder gates. There will be at least 2^m of these gates, one for each word line. There may be more than that number, because of gate fan-in limitations, as we will see in the following section on tree and matrix decoders. The cost of the column drivers will be equal

to the cost of a column driver gate, C_{CDG} times twice the number of bits in an addressable unit, s—twice because two drivers will be required for each bit line, one for read and one for write. This equation is extended to the 2½-D case, shown in Figure 7.7, by observing that there will be s cell arrays, each of size $2^m \times 1$, with decoder and column multiplexer size reduced to $2^{m/2}$. One bit of each addressable unit is then stored in each cell.

Tree and Matrix Decoders. The address decoders play an important role in memory array design. Recall the previous example, in which a 1 MB memory is implemented as a 2-D array, requiring 2^{20} decoder gates, each with a fan-in of 20. The typical decoder studied in beginning logic circuits classes has m inputs and 2^m AND gates, using one level of gates, each with a fan-in of m. This kind of decoder is called a *one-level* decoder, because there is only one level of gates: Each of the 2^m outputs is computed by an AND gate with m inputs. Gates with fan-ins greater than 8 are generally not available, however, at least in most electronic implementation domains. Most gates are limited to fan-ins of 6 to 8, and thus decoders must often be built using gates that have insufficient fan-in to allow a one-gate-level implementation. In this case, *tree* or *matrix decoders* are used. Figure 7.8a and 7.8b shows tree and matrix decoders implemented from 2-input AND gates, but the concept can be generalized to gates with 3 or more inputs.

tree decoder

The 3-to-8 line tree decoder shown in Figure 7.8a shows how a 3-to-8 line decoder that would require three-input AND gates in a one-level implementation can be constructed from 2-input AND gates at a space cost of four additional 2-input AND gates for the 2-to-4 line decoder, and a time cost of one additional gate delay. Notice that the 2-to-4 line decoder produces the four terms $\overline{\overline{x}_0\overline{x}_1}$, $\overline{\overline{x}_0 x_1}$, $\overline{x_0\overline{x}_1}$ and x_0x_1 at the root of the tree, with x_2 being a "don't care." This first level decoder is sometimes referred to as a "predecoder." At the next level in the tree, each of the four terms is ANDed with x_2 or $\overline{x}_2$, producing the required eight minterms.

This approach can be extended by ANDing the incomplete minterms with an additional set of incomplete minterms produced by a second 2-to-4 line decoder. Figure 7.8b shows

matrix decoder

a 4-to-16 line matrix decoder produced from 16 2-input AND gates plus eight additional gates used to form the "incomplete" minterms. Here the space cost is eight additional AND gates, and the time cost is the propagation through the additional layer of AND gates.

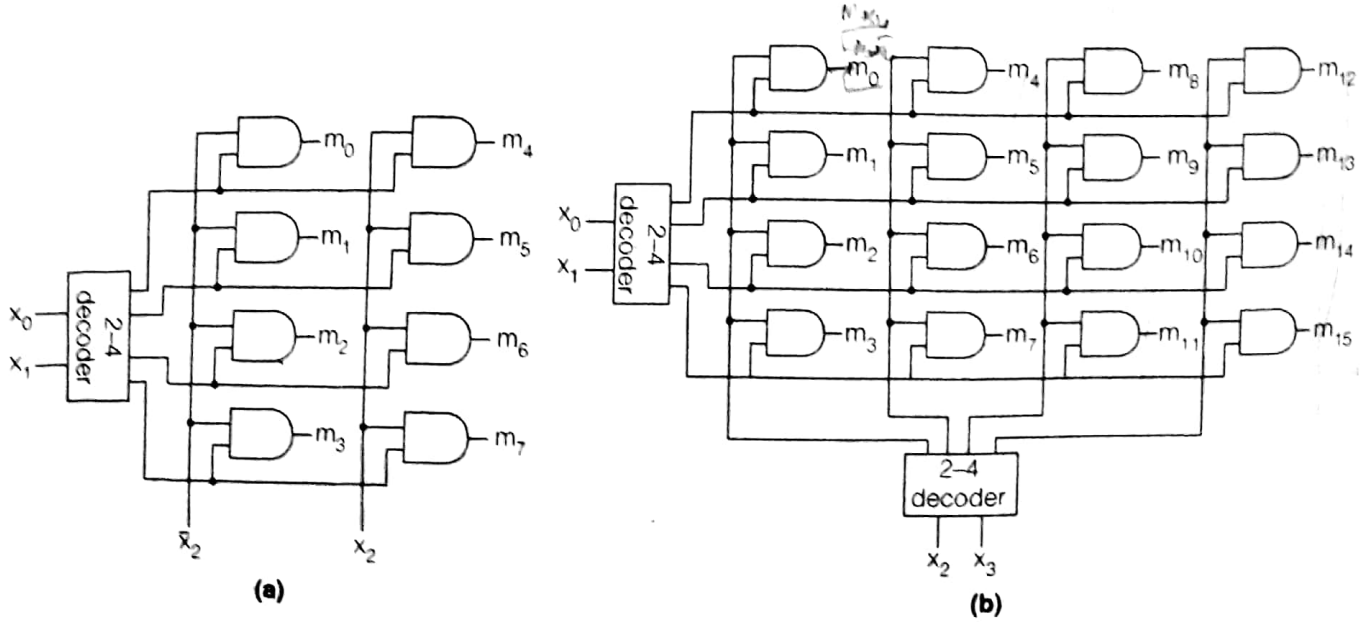


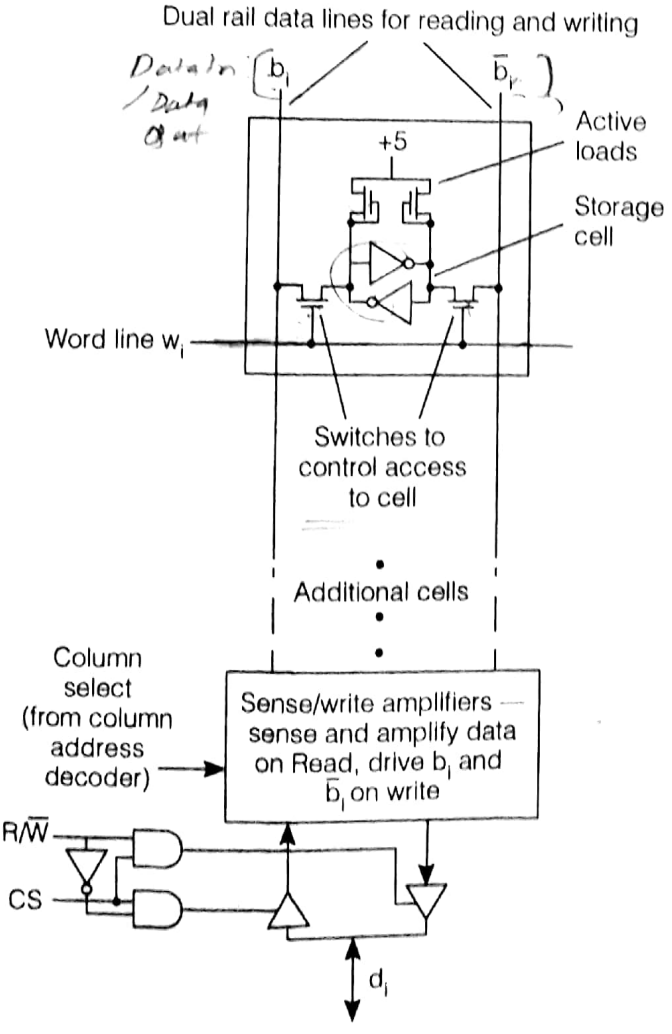
Fig. 7.8 (a) 3-to-8 Line Tree Decoder; (b) 4-to-16 Line Matrix Decoder

7.2.3 Static RAM Cell Design

Figure 7.9 shows a practical version of the static RAM cell in Figure 7.3. It uses cross-coupled inverters instead of cross-coupled NOR gates, resulting in considerable savings in chip area. The upper portion of the figure shows one storage cell in a column of storage cells, and the lower portion shows the mechanism for accessing the cell column. This implementation is known as a *six-transistor cell:* one transistor is used to implement each of the two inverters, two transistors are used to control access to the inverters for reading and writing, and two are used as active loads. This is one of the few instances in this text where we must descend below the logic gate level to explain the structure of a circuit.

The following list summarizes the operation of the six-transistor cell:

- The data storage latch is implemented by cross-coupled inverters instead of cross-coupled NOR gates.
- The two access-control transistors enable the cell for both read and write. The cell is selected by the word line as before, but in the six-transistor cell the word line also enables the cell for both reading and writing through the action of the two access-control transistors.



Address placed on address lines

Read

(1) Precharging
(2) Asserting the word line
(3) Data is on lines
(4) Sense & Amplify
(5) Multiplexing steps
(6) Finally, data on data lines

dual rail representation

- The DataIn and DataOut lines are replaced by dual rail inputs to the cell. *Dual rail* means that instead of using a single line d to represent a Boolean value, both d and $\overline{d}$ are used. These dual rail inputs, labeled b_i and $\overline{b}_i$ in the figure, are used for both reading and writing values.
- A value is written into the cell by applying the value to b_i and $\overline{b}_i$ through the sense-write amplifiers while asserting the word line. This causes the new value to be written into the latch.

precharg

- A value is read from the cell by *precharging* b_i and $\overline{b}_i$ to a voltage halfway between a 0 and a 1 and then asserting the word line. The value in the latch drives b_i and $\overline{b}_i$ to high and low or low and high. Those values are sensed and amplified by the sense/write amplifier at the base of the column.
- The control logic at the very bottom of the figure is used to select the column for reading or writing.

7.2.4 Static RAM Timing

We now take up the timing of static RAM operations. Consider the time required to read the contents of a cell in the cell array of Figure 7.6. Assume that the processor issues an address and a Read command simultaneously. The access time for a read will include the time for the address and Read command to propagate through the array to the cell, plus the time for the cell's contents to appear at the internal bus lines, plus the propagation time through the internal bus and through the multiplexer. The read operation timing is summarized in Figure 7.10. The Memory Address, Read, and CS, chip select, signals are presumed to have been applied to the chip simultaneously. The designation t_{AA} stands for the access time from address, the time required for the address and control signals to propagate to the cell and the cell data to propagate to the data line control buffers. After a time period equal to t_{AA}, the data appears at the outputs of the cell, d_0–d_7. The chip retains this data on the data lines until the control signals are deasserted.

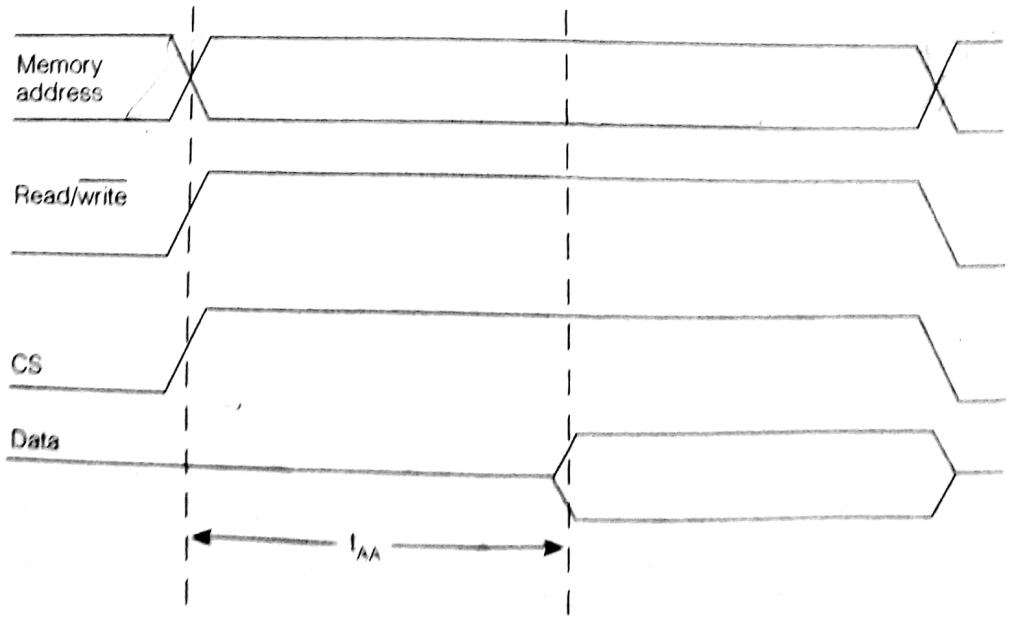


Fig. 7.10 Static RAM Read Operation

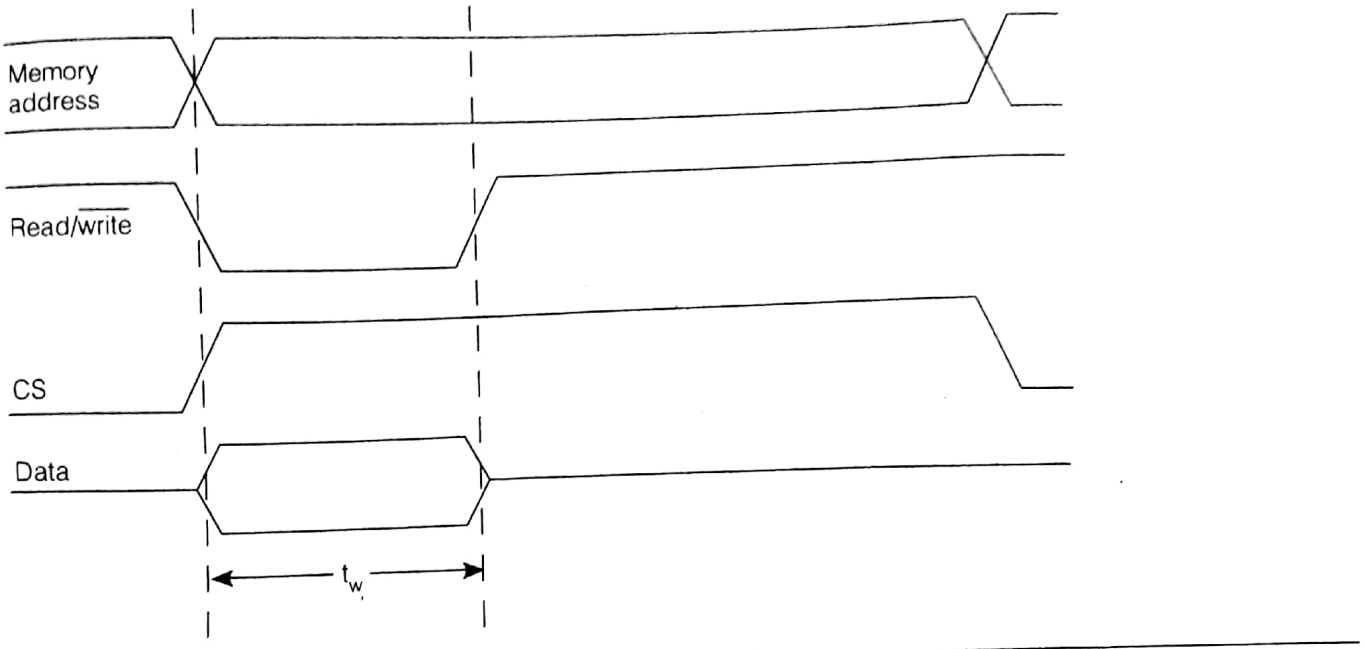


Fig. 7.11 Static RAM Write Operation

Timing for a write operation, shown in Figure 7.11, is similar, except that data to be written to the cell is applied to the data lines simultaneously with the address and control information. The R/$\overline{W}$ line must be held valid for a minimum time interval t_w, the write time, until data, address, and control information have been propagated to the cell and strobed into it. During this period the data lines must be driven with the data to be written.

There are a number of other practical timing constraints that must be adhered to when implementing a static memory system using RAM chips from a given supplier. The system designer should consult the appropriate data sheets and manuals for details.

7.2.5 Dynamic RAM

Information can be stored in a single capacitor instead of a pair of cross-coupled gates, with considerable savings in cell area. Figure 7.12 shows a dynamic RAM cell with its associated control circuitry. Notice that one transistor and a capacitor replace six transistors! Furthermore, there is only one bit line instead of two. The following list summarizes the operation:

- To write a value into the cell, the value, 0 or 1, is placed on the bit line, and the word line asserted. This charges the capacitor if a 1 is being stored, or discharges it if a 0 is being stored.
- To read the value, the bit line is precharged as in the static RAM case and the word line is asserted. The value in the capacitor appears on the bit line, where it is sensed and amplified, discharging the capacitor, and thus destroying its contents. Therefore, the following step occurs.
- The sensed and amplified value is placed back on the bit line as a part of the read process, thus *refreshing* the capacitor.

DRAM Refresh Overhead. If the capacitor is not read, and thus refreshed every 2 to 100 ms or so, its value decays to zero. Therefore it must be refreshed every few ms if it is to retain

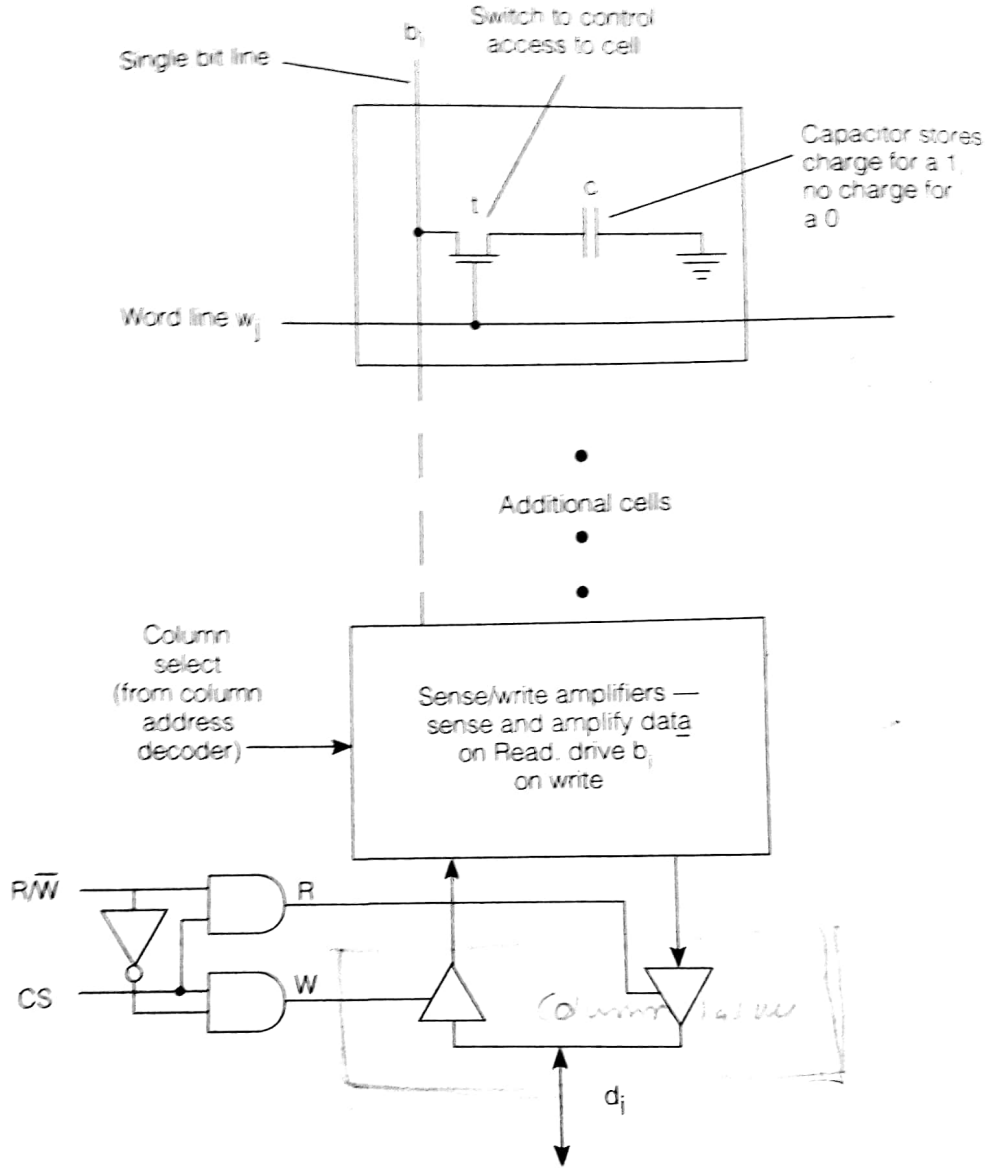


Fig. 7.12 Dynamic RAM Cell Organization

its value. This requirement that every cell in the chip be refreshed periodically complicates the design of dynamic RAM memories. The square aspect ratio is an aid in this regard, since an entire row can be refreshed at once by doing an internal read operation—that is, reading and refreshing the values of an entire row, without placing the read values on the external data bus. This *refresh overhead* is not severe. Consider a 4 MB × 1 DRAM that must be refreshed every 4 ms. If the cells are organized as a 2048 × 2048 array, then there must be 2048 refresh operations every 4 ms. If a refresh takes 80 ns, then the fraction of time devoted to refresh is $2048 \times 80 \times 10^{-9} \times 100/(4 \times 10^{-3}) \approx 4.1\%$, a not unreasonable overhead in most situations. Furthermore, as we shall discuss in DRAM timing, much or all of this refresh overhead can be hidden in the total cycle time of the memory operation.

refresh overhead

Dynamic RAM Chip Organization. Several factors influence the design of DRAM cell arrays. The most important are the number of I/O pins and the need to refresh the cells. Consider that when an IC has achieved maturity in the marketplace—that is, when its R&D

costs have been largely paid off, and when the parts yield is high—that is, few parts have to be discarded because of manufacturing defects, the packaging of the circuit may cost more than the silicon inside. Furthermore, packaging cost is directly related to the number of pins required by the IC, and DRAM parts are among the most cost-competitive of ICs. A 16 MB × 1 DRAM has 24 address pins, 1 or 2 data pins, at least 2 control pins, and 2 pins for power and ground—30 pins in all.

One important scheme for saving pins is to transmit the row address and column address over the same pins, one after the other. This technique, referred to as *time-multiplexing*, cuts the number of pins devoted to the address in half, from 20 to 10 in the case of a 1 M × 1 chip. Two additional control signals must be added, one to inform the chip when the row address is valid on the address lines, and the other to inform the chip when the column address is valid on the address lines. These two are referred to as RAS, row address strobe, and CAS, column address strobe, respectively. An additional pin is saved if the CAS signal also serves as the CS signal, which is usually the case. This is fortuitous, since an entire row can be refreshed by asserting RAS without asserting CAS. These signals are usually active low and appear as $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and $\overline{\text{CS}}$.

row and column address strobes

Figure 7.13 shows the block diagram of a generic 1 M × 1 dynamic RAM chip. Notice the ten incoming address lines, over which the 20-bit address is time multiplexed, and the

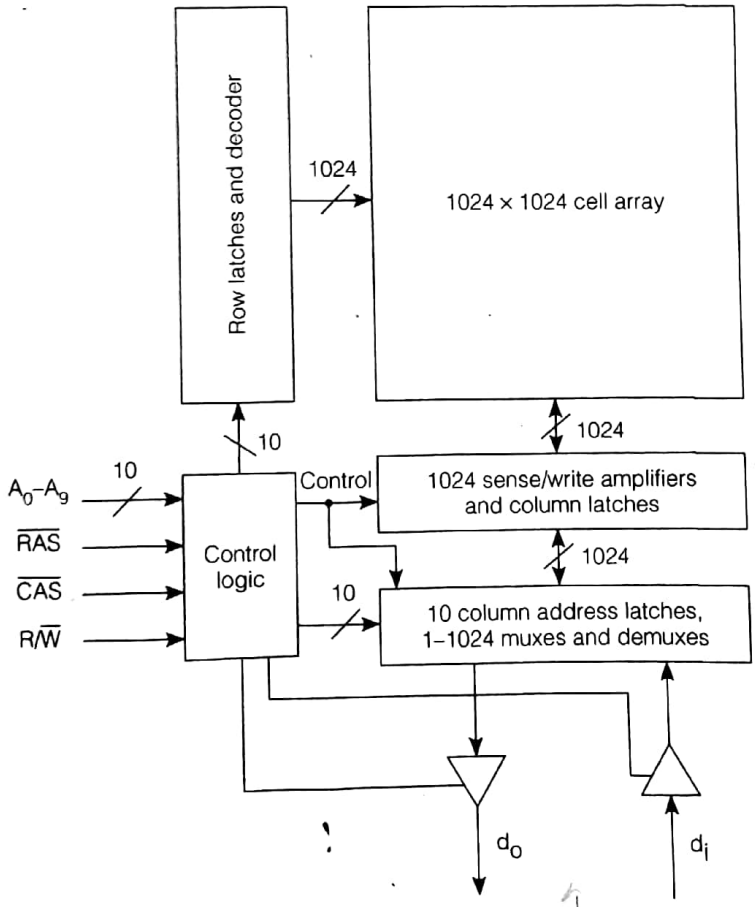


Fig. 7.13 Dynamic RAM Chip Organization

control lines, $\overline{RAS}$ and $\overline{CAS}$, and R/$\overline{W}$. The row and column hardware includes address latches for storing the row and column addresses for the duration of a memory cycle and an array of column latches for temporary storage of the column that was read out from the cell array. A memory access consists of a $\overline{RAS}$ and the row address, RA, followed by a $\overline{CAS}$ and the column address, CA.

Figure 7.14 shows a typical read cycle for a DRAM chip. The cycle time, t_C, is composed of t_{RAS}, the minimum time RAS must be asserted, and t_{Prechg}, the minimum time RAS must be deasserted to allow the chip to precharge its bit lines. Data are available after the access time, t_A. The assertion of the CAS signal causes the chip to output data, obviating the need for a separate CS signal. The read cycle is terminated when both RAS and CAS are deasserted. Figure 7.15 shows the analogous write cycle for our DRAM. Notice the important timing parameter, t_{DHR}, the time that data must be held after the assertion of RAS initiates a cycle.

Many additional timing constraints must be observed for proper DRAM operation. Consult the original equipment manufacturer's data sheet for timing details.

SDRAM and DDR SDRAM. There is an ever-increasing need for larger amounts of faster RAM to counteract processor memory starvation. This has led to the development of clocked, or synchronous DRAM, SDRAM, and its even faster relative, DDR, or Double Data Rate SDRAM. Unlike proprietary RAM designs, these RAM designs are based on an open JEDEC (Joint Electron Device Engineering Council) standard. The activities of the RAM chip are synchronized with the CPU bus clock, allowing a shorter bus cycle time. SDRAM chips are available in word sizes of 4 to 32 bits per chip.

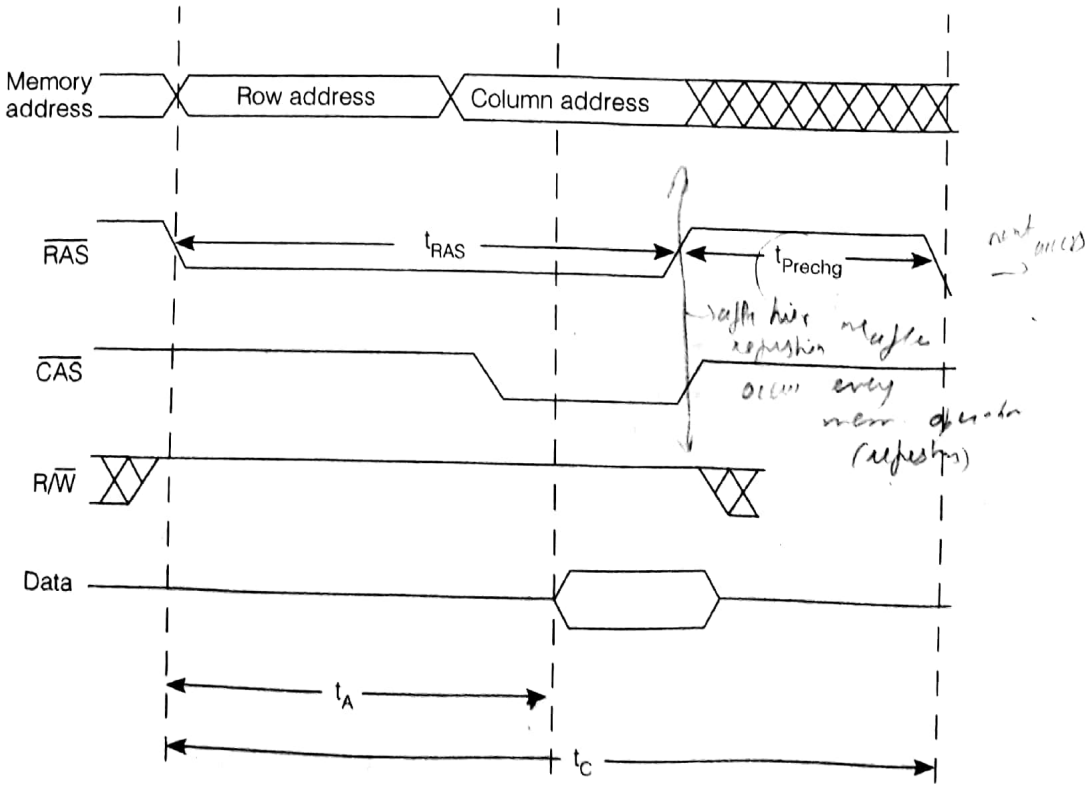


Fig. 7.14 DRAM Read Cycle

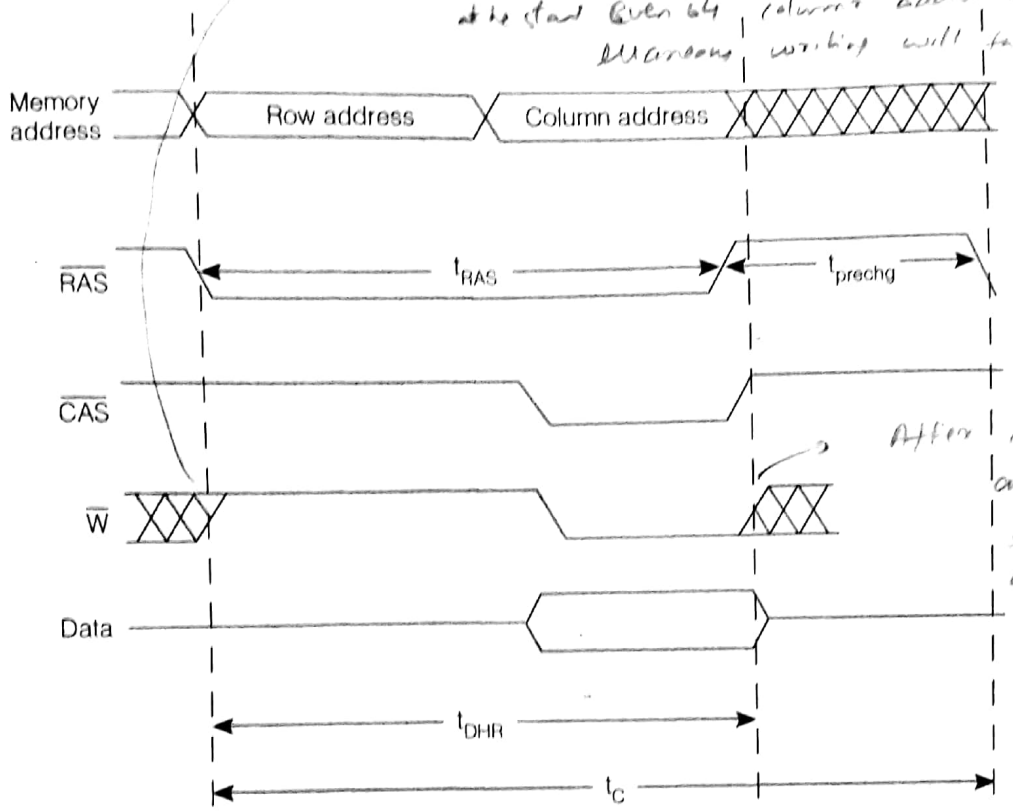


Fig. 7.15 DRAM Write Cycle

They have multiple internal memory "banks," allowing some banks to be read while others are being precharged. They are operated in a programmable "burst" mode. In the burst mode, accesses are started at a given location and continue for a programmable number of locations, typically 2, 4, or 8, in a programmable sequence. SDRAM chips are characterized by their clock speed and initial access latency. Typical clock speeds are in the 100–167 MHz range, with 200 MHz on the horizon. Latency between address and the first data item is 2 or 3 clock periods, and additional words in the burst arrive on succeeding clock pulses. DDR SDRAM doubles the data rate by providing data values at both rising and falling clock edges. Thus 167 MHz DDR SDRAM on a 64-bit bus can provide a burst bandwidth of 167 MHz × 2 × 8 or 2.4 GB/sec.

Getting Specific: The Micron Technology MT46V32M16TG-xx DDR SDRAM chip The MT46V32M16TG-xx is a 512 Mb chip organized as 32M × 16 bits. Internally it is organized into 4 banks of 8,192 × 512 × 32bits. It has a programmable read and write latency of 2 or 2.5 clock cycles, after which it will yield bursts of two 4, 8, or 16 16-bit words per clock cycle. Each of the 4K rows are refreshed once each 64 ms. It also has input mask modes that enable writing of only the low- or high-order byte. ■

Dynamic RAM Refresh Mechanisms. The intricacies of DRAM refresh go far beyond what is covered in this section, but the basic approach is to guarantee that every row of the chip gets refreshed according to the manufacturer's specifications. At a minimum, DRAM chip manufacturers provide the capability to refresh a row by a "RAS-only" refresh cycle.

In a RAS-only refresh cycle, the row address is placed on the address lines, and RAS is asserted for a period allowing the row to be read out, refreshed, and written back, without ever asserting CAS. The absence of a CAS phase is a signal to the DRAM chip that a refresh cycle is being requested; consequently, the chip refreshes the specified row without ever placing data on the external data bus.

Many DRAM chips also have a "CAS-before-RAS" cycle. The DRAM chip maintains an r-bit counter, where r is the number of row address bits. Every time the chip encounters a cycle where CAS is asserted before RAS, it initiates a refresh cycle, refreshing the row pointed to by the internal counter, and then increments the counter to point to the next row to be refreshed. It is still the responsibility of an external refresh controller to initiate sufficient refresh cycles to meet refresh time constraints. Most DRAM chip manufacturers also provide DRAM controller IC's that encapsulate the refresh function within the IC. We will revisit the refresh operation during the discussion of memory boards in the next section.

Special Purpose DRAMs. Several variations on the basic DRAM structure described thus far permit faster than normal access to DRAM contents under certain conditions. The variations all take advantage of the fact that DRAMs read an entire row of data at once, and thus can provide faster access to the data in the row that was read and is present in the column latches.

Page mode DRAMs permit access to the entire row by asserting RAS and the row address, RA, only once, while repeatedly asserting CAS and the appropriate column address, CA. The page mode begins with the usual RAS CAS cycle, but with RAS continually asserted. In subsequent cycles only CAS and the appropriate column address are asserted. (Memory cycles are terminated with semicolons in the following discussion: RAS RA CAS CA_1; CAS CA_2; CAS CA_1; CAS CA_4; and so on.) This results in much faster access time to the values stored in the subsequent columns present in the column latches. Data can be both read and written in page mode.

DRAMS fast access modes

Nibble mode DRAMS provide a variation on page mode access by permitting access to a *nibble*—4 bits—without altering the column address. The signaling convention begins with an ordinary RAS CAS cycle, but with RAS continually asserted, while CAS is asserted three more times: RAS CAS; CAS; CAS; CAS; RAS CAS; CAS; CAS; CAS; and so on. Each additional assertion of CAS causes the next bit in the row to be accessed without incrementing the column address.

Static column mode DRAMs provide yet another variation on the basic approach. The static column mode begins with a normal RAS CAS cycle, after which RAS is continually asserted, with only the column address being changed in each subsequent cycle: RAS CAS CA_1; CA_2; CA_1; CA_4; and so on. Generally, values may only be read using static column mode, unless special write signaling is employed.

Video RAM

Video RAM (VRAM) is specifically designed to be used as a video display buffer, where the contents of the VRAM are always read out sequentially to display the contents of a line of video. The VRAM chip clocks the entire row into a shift register for shifting out bit by bit to the display. While this shifting occurs, new values can be stored in the cell array, leading to what is in effect a dual-ported RAM; that is, RAM that can be read from sequentially as it is being updated.

You will have noticed that much of the timing and operation of DRAMs is not obvious from inspection of the timing diagrams. The complexity caused by the need to refresh the chip has forced chip vendors to introduce various special purpose cycles. Rather than add

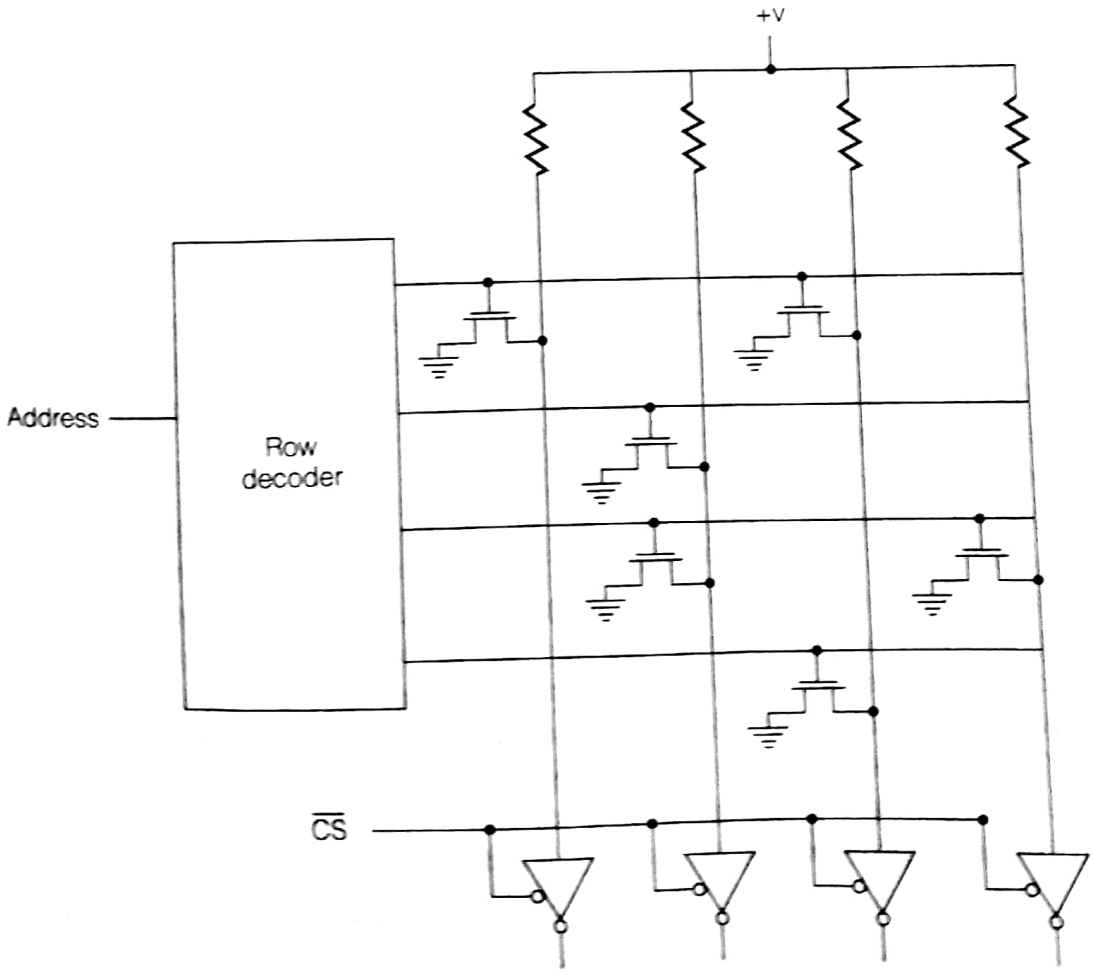
additional pins, special signaling protocols have been introduced that involve the pulsing of the RAS, CAS, and R/$\overline{W}$ lines. Again, see the vendor's data sheets for details.

7.2.6 Read-Only and Read-Mostly Memory

There is also a need for memory that has been preprogrammed with information permanently encoded in the chip, read-only memory, abbreviated ROM. ROM is by definition *nonvolatile*. That is, it retains the information in it when power is removed from it. Examples range from low-level operating system routines to the microcode control store discussed earlier, and to video game cartridges. There is also a need for memory in which to store information that is updated infrequently, such as automobile engine control parameters and engine performance history. ROMs are also used to implement combinational logic functions. We briefly discuss the kinds of ROM technologies available and then the various ways in which ROM is used in computer systems.

There is considerable price/performance/capability trade-off possible in read-only memories, depending on user requirements.

Mask-programmed ROMs are the least expensive. The customer specifies the size and encoding pattern of the ROM to the ROM supplier, and receives the programmed ROMs several weeks later, after paying a one-time mask-making charge. The typical ROM structure is similar to 2-D RAM structure, with row decoders and possibly column multiplexers, but without the complications of the write circuitry. Figure 7.16 shows a 4 × 4 CMOS



7.3 Memory Boards and Modules

Large main memories are often more than just memory chip arrays with decoders, multiplexers, and I/O. They can have several levels of structure, and the level described in the previous section may be integrated onto memory chips. Multiple chips may be organized into a memory board providing more bits per word and/or more words than are contained in a single chip. One or more boards can be combined to form a *memory module*, an independent unit satisfying the requirements of the processor interface and often including registers for memory address and data. The complete main memory of a computer may be made up of several memory modules, connected either to expand the memory capacity or to reduce the effective access time by allowing overlapped operation. This section treats the memory structures between the chip and system level.

memory module

7.3.1 Arrays of Memory Chips

We start the discussion of higher-level memory organization by abstracting the structure of an individual memory chip, as shown in Figure 7.17. Data input and output are shown as multiplexed on the same set of pins using tri-state technology, but all the organizations to be described could also be done with separate input and output connections. The figure shows one signal, R/$\overline{W}$, which is high for a read operation and set low for a write. When combined with chip selection, it both enables the tri-state data outputs and generates the strobe for writing data into the selected cell. This technique is commonly applied with the transition of the R/$\overline{W}$ signal timing the strobe. The presence of all select signals and a high R/$\overline{W}$ enables the tri-state data outputs.

Static and Dynamic RAM Chips. Static RAM chips can be assembled into systems without changing the timing characteristics of a memory access by very much. Dynamic

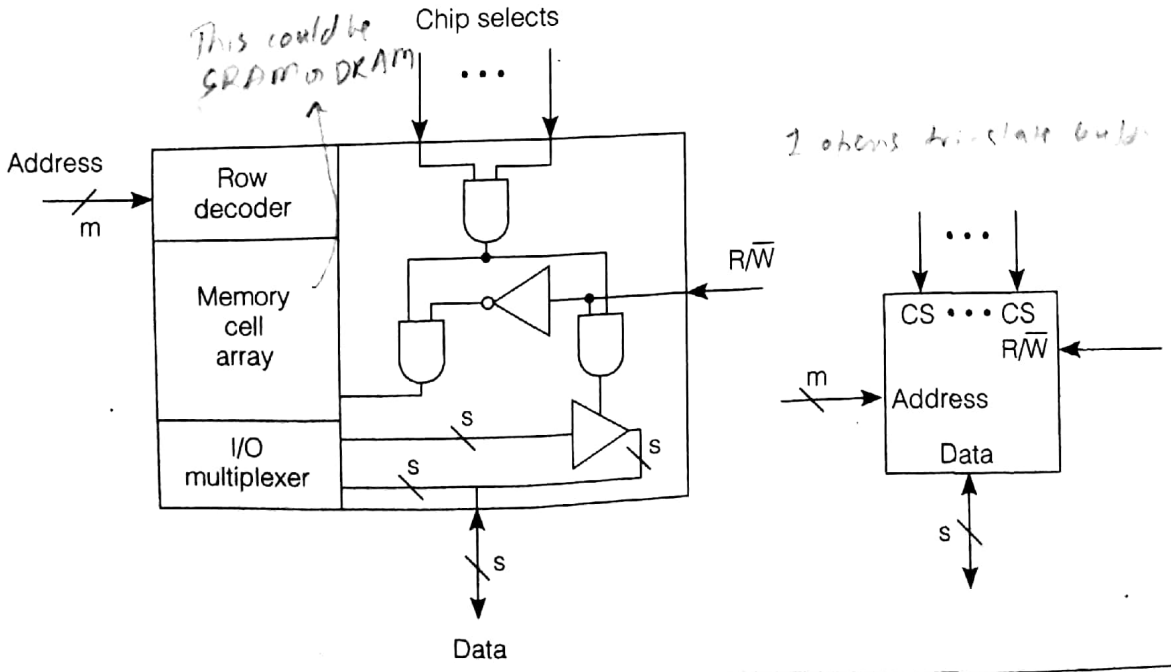


Fig. 7.17 General Structure of a Memory Chip

RAM chips, however, have enough timing complexity that a memory module built from dynamic RAM chips will have complex control. The causes of timing complexity are the time-multiplexed row and column addresses, and the refresh.

DRAM access time

The multiplexed address reduces the number of address pins in the memory chip diagram to $m/2$ and replaces the chip selects by RAS and CAS strobes. It increases access time because two half addresses must be transmitted in sequence. The refresh operation, also discussed in the previous section, requires control circuitry, which could be integrated onto a memory chip. If this is done, the simple chip diagram of Figure 7.17 must have an extra output associated with it to indicate whether the chip has to wait for a refresh cycle before responding or not. This usually takes the form of a Ready signal that can be connected to other chips' ready signals by wired OR, so that the combined signal is low (not ready) if any chip is not ready. Only selected chips will cause Ready to go low, and only when a refresh cycle is in progress.

DRAM memory controller

On-chip refresh control is not common in dynamic RAMs that have only a few bits per chip. If a memory with 32-bit words is made of 32 1-bit dynamic RAM chips, each of which refreshes independently, the probability of finding some chip busy is about 32 times higher than it would be with a single refresh cycle for all bits. The more common approach is to build wide word memories from narrow dynamic RAM chips without internal refresh and to use a separate *memory controller* chip. This chip contains refresh and interface circuitry for all memory chips and refreshes them simultaneously. Therefore, we treat refresh in more detail at the memory module level, where it is usually implemented.

Expanding Memory Word Size. We begin the discussion of multiple-chip memory structures from the point of view of static RAM chips and add comments about things special to dynamic RAM as necessary. Figure 7.18 shows how to combine chips to expand the memory word size while keeping the same number of words. Address, chip selects, and write signals are connected in parallel to all chips. Only the data signals are kept separate, with those from each chip supplying different bits of the wider word. The cost of pins on a chip encourages narrow words for high-capacity memory chips—often only 1 bit. This is because adding a data pin to a chip with 2^m words of s bits increases the number of bits it can store by only a factor of $(s + 1)/s$, while adding an address pin always doubles the capacity. Except for the change from $s = 1$ to $s = 2$, more chip capacity can be accommodated with fewer pins

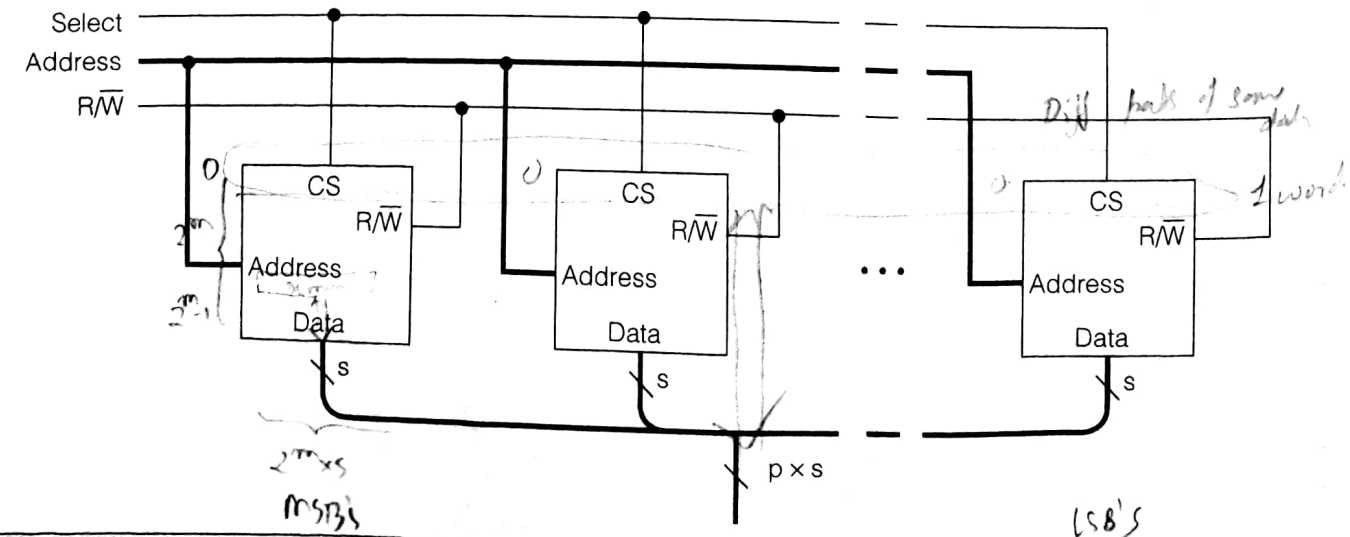


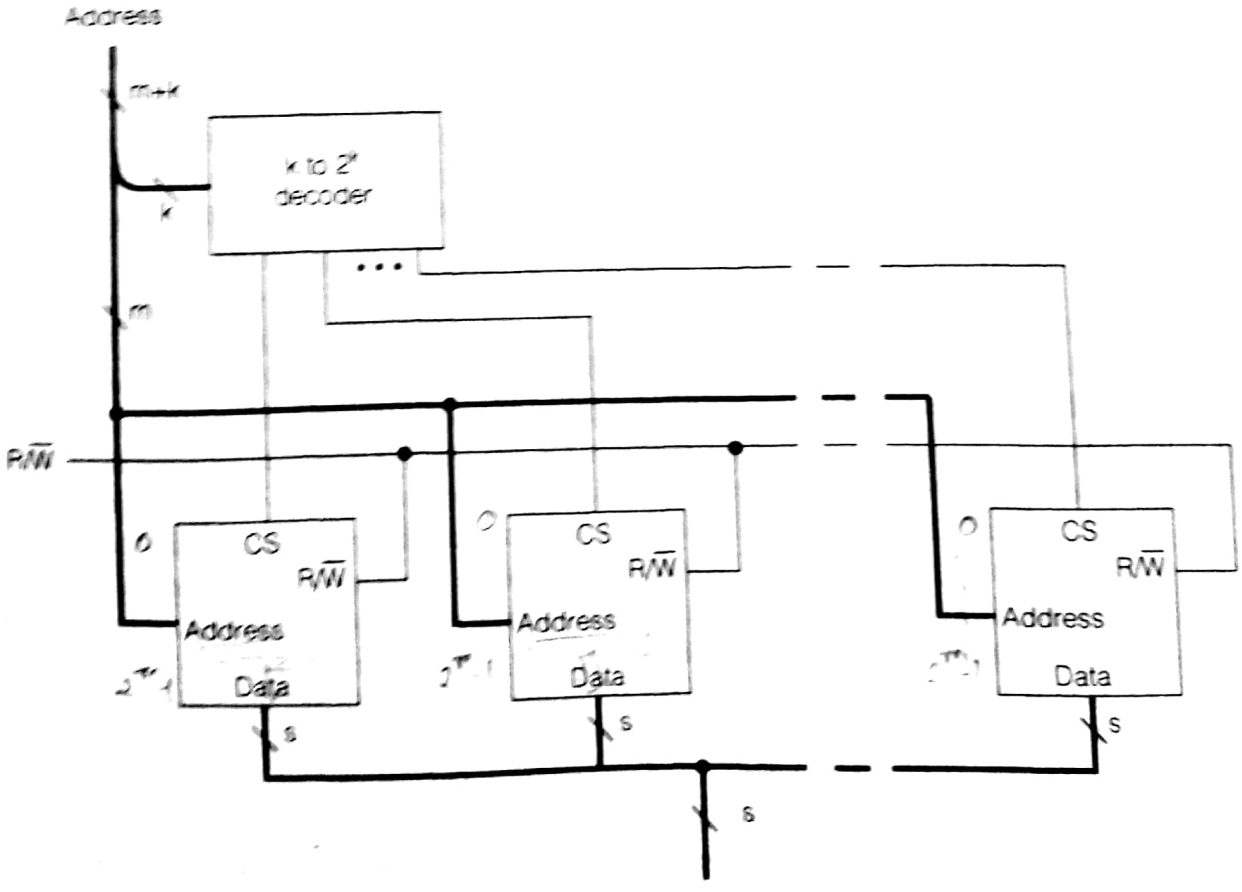
Fig. 7.18 Word Assembly from Narrow Chips

by increasing the address size rather than the word size. Figure 7.18 uses static RAM chips. If dynamic RAM chips are used, the address will be multiplexed on $m/2$ signal lines, and the chip select will be replaced by row address and column address strobes.

The timing of a multichip memory board is directly related to the timing of the individual chips of which it is made. In the case of static RAM, the only change in timing is that it is necessary to account for the increased fan-out of the signals that drive p chips in parallel. Slightly longer setup times may be required by the p-chip structure because of this fan-out delay. For dynamic RAM chips, RAS, CAS, and R/$\overline{W}$ may need to be delayed with respect to the multiplexed address signals, but otherwise, the timing is the same. Narrow dynamic RAM chips are even more common than with static RAM, since one extra address pin increases both row and column address sizes by 1 bit and thus can accommodate a fourfold increase in chip capacity. With the parallel connection described, all bits in a word and all words in a row will be refreshed simultaneously.

7.3.2 Increasing the Number of Words—Address Space Expansion

To increase the number of words in a memory beyond the capacity of a single chip, the chip select lines of several chips are driven by the outputs of a decoder whose inputs are additional address bits. In designing a board, the word capacity is usually increased by a power of 2, so that all outputs of a binary decoder are used, as shown in Figure 7.19.



The board may not have all the chips installed with smaller memory configurations. The figure shows 2^k chips of 2^m words per chip being combined to form a 2^{m+k} word memory with the chip size of s bits being the word size of the total system. As in expanding the word size, the R/$\overline{W}$ and address inputs of all chips are connected in parallel. The m chip address lines are connected to m bits of the $m + k$ bit memory address. The other k bits are inputs to the decoder, which outputs a distinct chip select for each chip. If all decoder outputs go to an installed chip, it really makes no difference which address bits go to the decoder and which to the chips. By convention the high-order address bits select the chip. This becomes necessary to make installed words consecutively numbered when some chips are missing. Each of the s input/output pins on a chip is connected in parallel with the corresponding I/O pin of all other chips, taking advantage of the tri-state nature of the output and the fact that one and only one chip is selected. An enable input to the decoder will provide an enable for the whole memory array.

Both the memory word size and the number of words can be expanded simultaneously. Each chip of Figure 7.19 is replaced by p chips with their address, chip select, and read/write pins all connected in parallel, as in Figure 7.18. Each I/O pin of a chip for a given select signal is wire ORed to an I/O pin of one chip in each group of p chips connected to other select signals. Each of the p sets of 2^k chips corresponding to s wired OR I/O signals supplies s different bits of the $(p \times s)$-bit memory word.

Dynamic RAMs require more connections. The lower m bits of the address are multiplexed onto $m/2$ address lines timed by the RAS and CAS strobes. The RAS pins of all the chips are wired in parallel, but since CAS also acts as a chip select, it must be routed through the k-bit to 2^k decoder driven by the upper k address bits. The decoder enable input then becomes a CAS signal for the whole chip array, and each decoder output is a CAS signal for one chip, or for a set of p chips if words are wider than chips. The timing of the overall CAS signal must match the timing required for the individual chip CAS signals, with allowance made for the propagation time through the decoder. Since column address strobes are often active low, the outputs and enable input of the decoder may need to be inverted. The parallel connection of the RAS pins implies that a refresh cycle, which only requires RAS, will refresh one row of all chips in the array.

The general diagram of a memory chip in Figure 7.17 shows more than one chip select input per chip. Since all must be true to select the chip, multiple chip selects can simplify the construction of the decoder shown in Figure 7.19 by supplying the last level of AND gates in the decoder. This is particularly useful with the matrix decoder structure shown in Figure 7.20. This structures the memory as a two-dimensional (2-D) array of chips. The q-bit vertical and k-bit horizontal decoders drive two different chip select lines to select one of 2^{k+q} chips. The chip selects are the only connections that are different for each chip. The address, data, and read/write signals are all connected in parallel. An enable input can be added to either the horizontal or vertical decoder if an enable for the whole array is desired. Again, it is simple to expand the word size to $p \times s$ bits by replacing each chip in the array by p chips connected as described with regard to Figure 7.19. Alternatively, one can think of a 2-D matrix of chips for each bit. These matrices are stacked in the third dimension to form a true 3-D memory structure. A physical situation that would correspond directly to a 3-D logical structure would be a daughter board for each 2-D chip array, with daughter boards mounted perpendicularly on a mother board. In current PCs such daughter boards are commonly called *SIMMs*, for single, in-line memory modules.

two-dimensional chip arrays

SIMMs

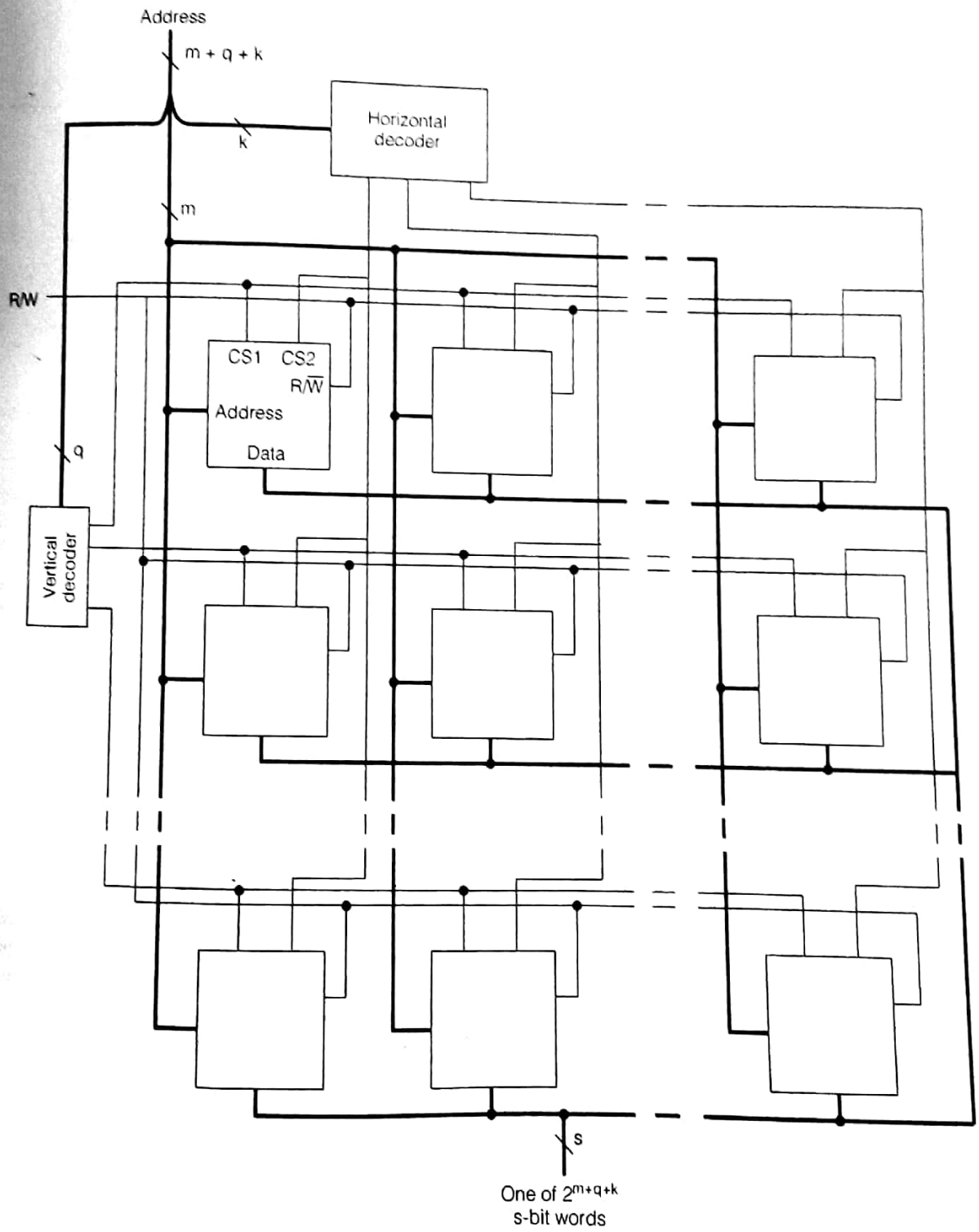


Fig. 7.20 Chip Matrix Using Two Chip Selects

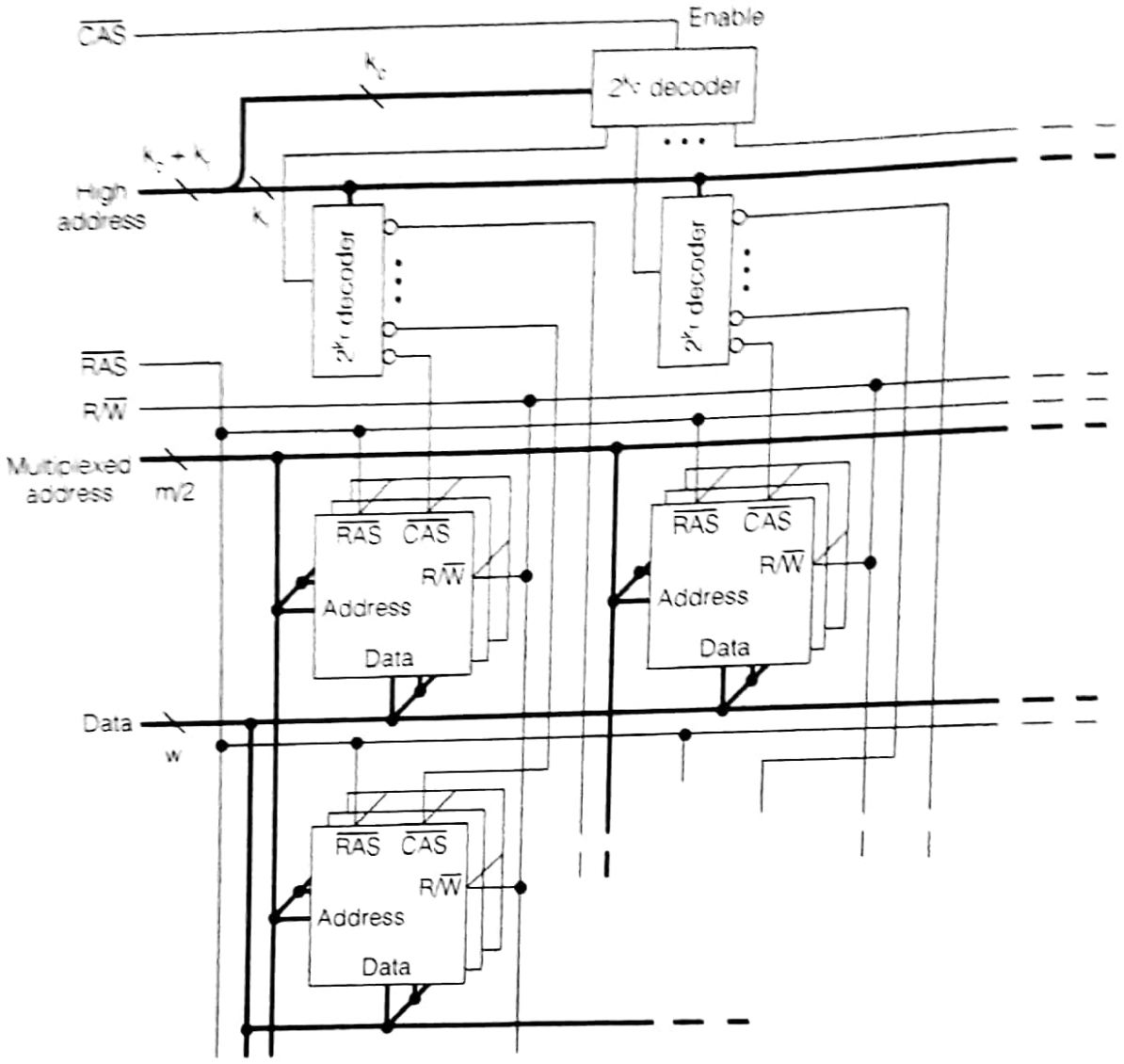


Fig. 7.21 Three-Dimensional Dynamic RAM Array

The combination of chip select with the column address strobe in dynamic RAMs makes it less common for them to have multiple chip selects. A gate external to each chip could combine two or more select lines into a single CAS for the chip. However, the external gates and their packaging and wiring reduce the benefit of the simple horizontal and vertical wiring patterns of the 2-D matrix layout. An alternative is to enable a 2^k-way decoder with the overall CAS signal to produce a CAS for each column. A column CAS signal is then the enable input to a 2^q-way decoder for each column. The column decoders produce individual CAS signals for each chip. A 3-D dynamic RAM array is shown in Figure 7.21.

7.3.3 Advanced Topic: The Memory Module

A memory module presents a specific memory interface to the processor or other unit that references memory. It usually contains buffer registers for the address and data, so that the processor need not hold these values constant for the duration of the memory cycle, or strobe the output data at a precise time in the case of a read. A module operates

independently by accepting address, read or write commands, and perhaps data, storing the data on write or returning new data on read. It may be a single array of chips, or it may contain several memory boards. The interface to a memory module is a bus that has signal wires corresponding to

memory bus

- Read and Write—start signals for memory cycles
- Ready—memory is ready for next write or data is available from last read
- Address—must be sent to memory at time of Read or Write signal
- Data—sent with Write or available from memory when Ready becomes true after Read
- Module select—needed when several modules share a bus

Figure 7.22 shows a memory module and its bus interface. The bus timing does not have to be bound to the memory access timing. On a read, for example, the memory bus may be used for a short time to send the address and start a read of a slow memory module. That memory module only needs the bus again after the read is complete and the data must be accepted by the processor. The Ready signal tells when this may happen. The control signal generator matches the bus timing to the memory chips, so it depends on the specifics

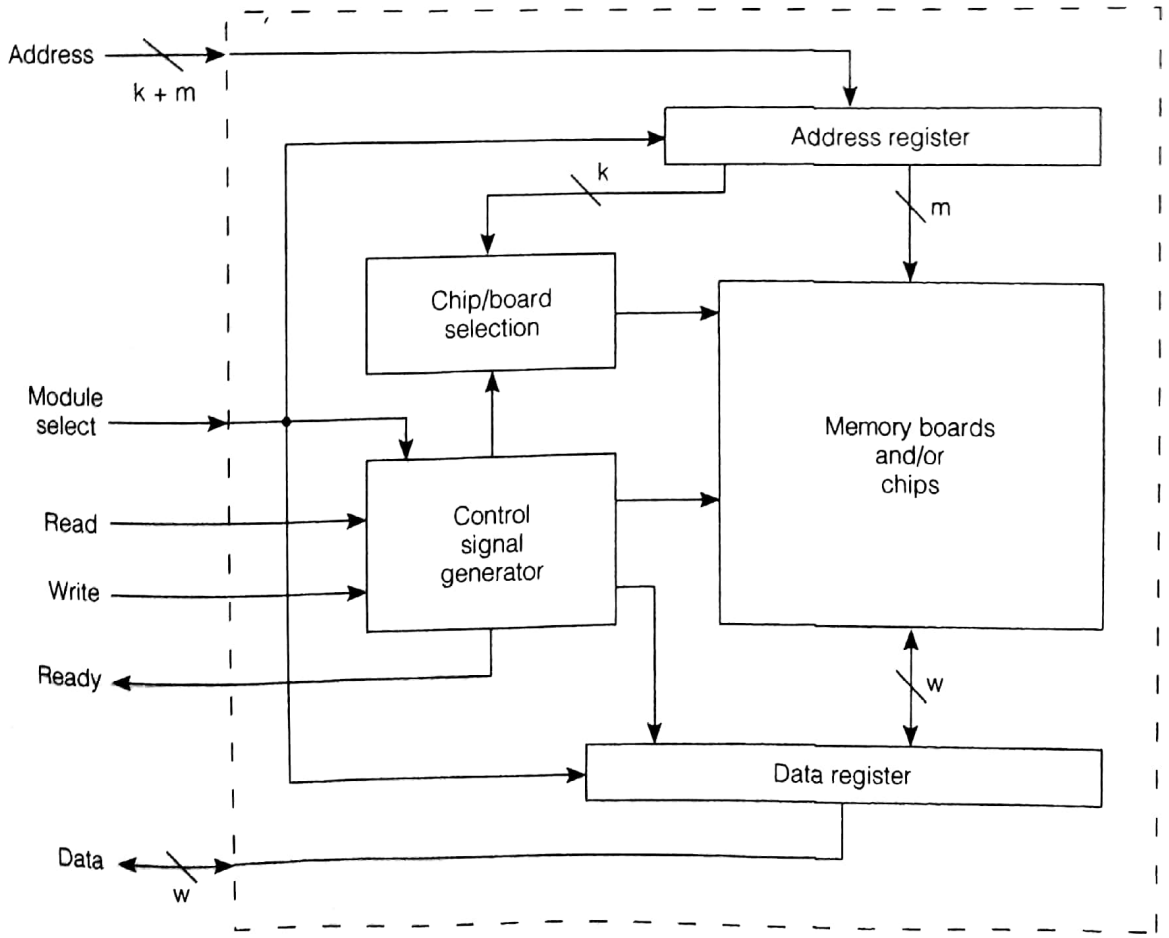


Fig. 7.22 A Memory Module and Its Interface

of the processor interface and the memory chips. The general concepts discussed here must be adapted to the details of any real system.

The control signal generator shown in the figure is fairly simple for static RAM. It is limited to strobing the data register after the correct access time delay on a read and generating the Ready signal on completion of a read or write. For dynamic RAM the controller is much more complicated, since it not only generates correctly timed RAS, CAS, and R/$\overline{W}$ signals and multiplexed address lines, but usually handles refresh for all the chips of the memory too. A dynamic RAM module is shown in Figure 7.23, which also shows the major components of the refresh hardware.

refresh counter

Refresh is controlled by a clock that generates a refresh request for each row in a chip within the few-millisecond period specified by the manufacturer for refresh. This is independent of the number of chips in the memory, since parallel connection of the address and RAS lines allows a given row of all chips to be refreshed simultaneously. An $m/2$ bit *refresh counter* supplies the row address for refresh. The refresh counter, row address, and column address are inputs to a three-way multiplexer that drives the chip address lines. Row and column addresses are taken from the upper and lower halves of the lower m address register bits, leaving the high-order k bits to select board and chip. The CAS signal and the board and chip selects are independent of refresh, but the RAS signal to the memory array can be generated

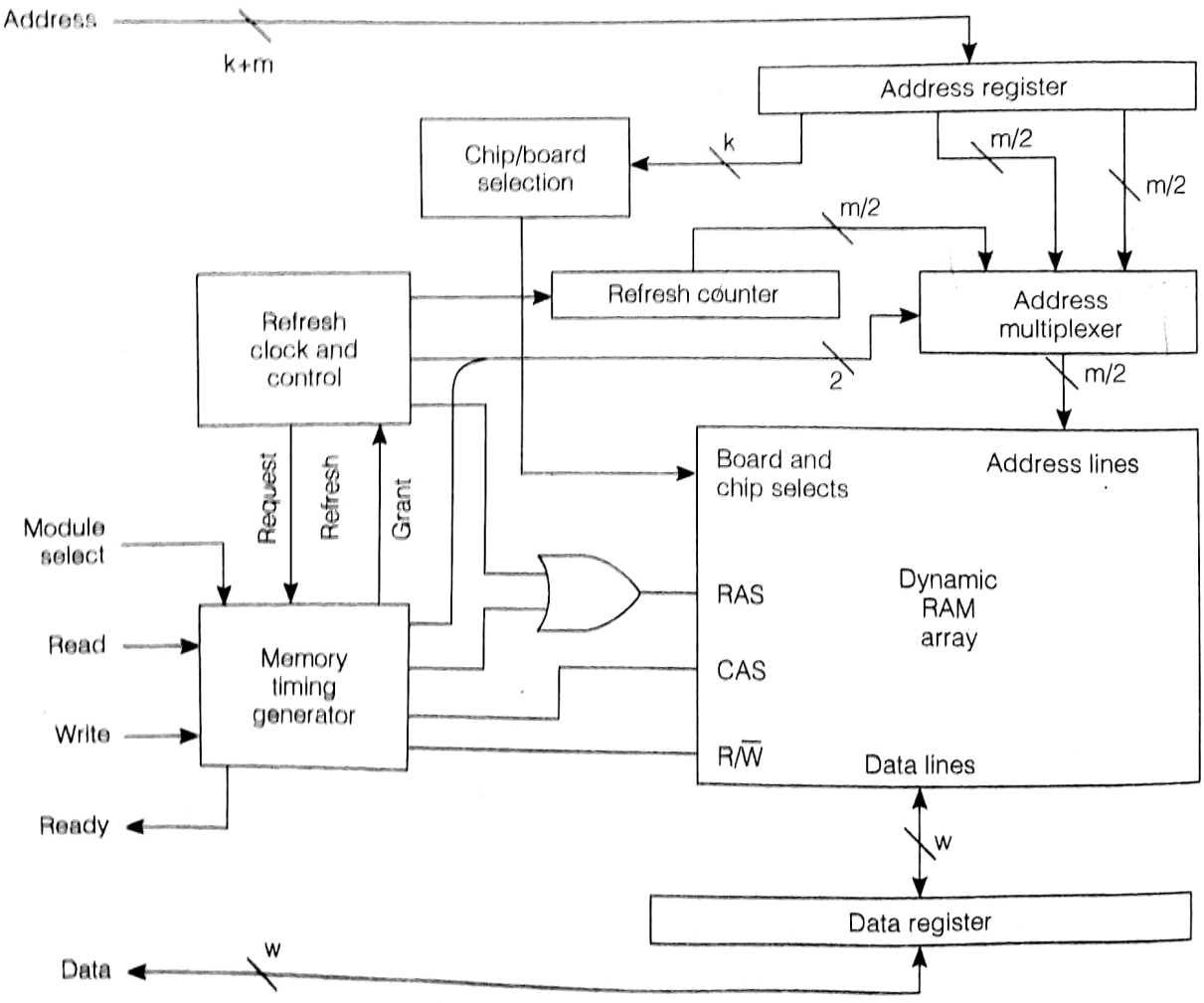


Fig. 7.23 Dynamic RAM Module with Refresh Control

either by a read or write access or by the refresh control. If a refresh is requested during a read or write, the memory timing generator will not grant the request until the memory access is complete. Similarly, a memory access that arrives during a refresh cycle will not be granted until the refresh cycle is complete. Priority is given to refresh in the case of simultaneous requests. Ready will not be set to true until both the refresh and the access are completed.

7.3.4 Memory Interleaving

The technique of organizing a number of memory modules to realize the memory address space of a processor is called memory mapping. Memory mapping can be alternatively viewed as the method of mapping different subsets of the memory address space of a processor onto individual memory modules. Memory mapping schemes distribute the memory addresses onto different modules, and therefore these schemes are also called as memory interleaving schemes. There are basically two kinds of memory interleaving schemes. They are (1) high-order memory interleaving and (2) low-order memory interleaving schemes.

In the following discussions on memory interleaving we treat memory modules as the basic building blocks of the memory system. Figure 7.24 shows the block diagram of a memory module. The module has 2^n address locations, each capable of storing a data of width d bits. T is the access time of individual memory modules. The modules work as follows: on placing the address on the address bus and asserting the read command line and the module select line, the data present in the corresponding location will be read and made available on the data bus within a maximum delay of T seconds. The write operation takes place in a similar manner.

High-Order Memory Interleaving. In the high-order memory interleaving shown in Figure 7.25a, the h high-order lines are used to generate the module select lines. The m low-order address lines are connected to the address port of the memory modules. Modules are of capacity 2^m and there are $2^h = M$ memory modules indexed from $0 \le i \le 2^h - 1$.

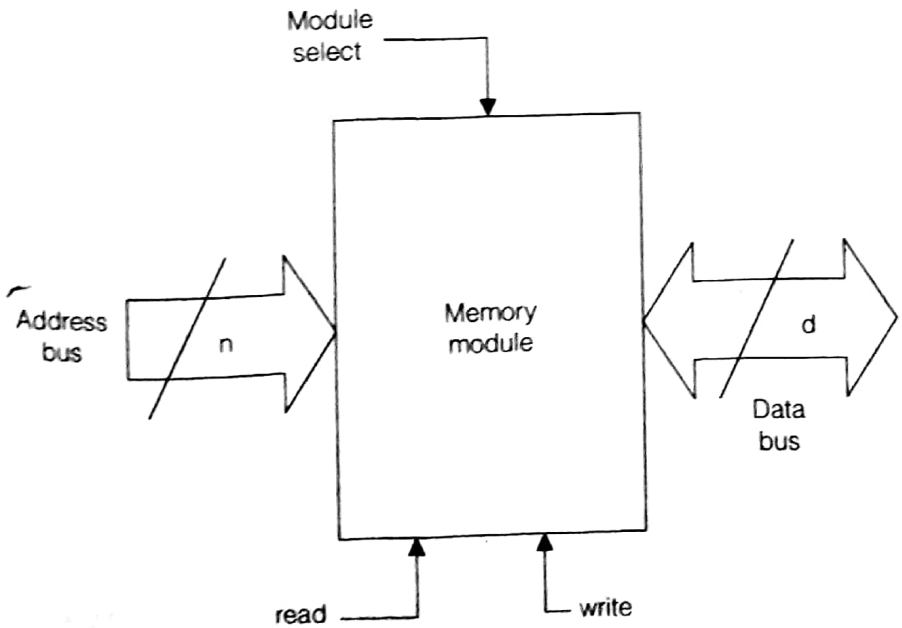


Fig. 7.24 Block Diagram of a Memory Module

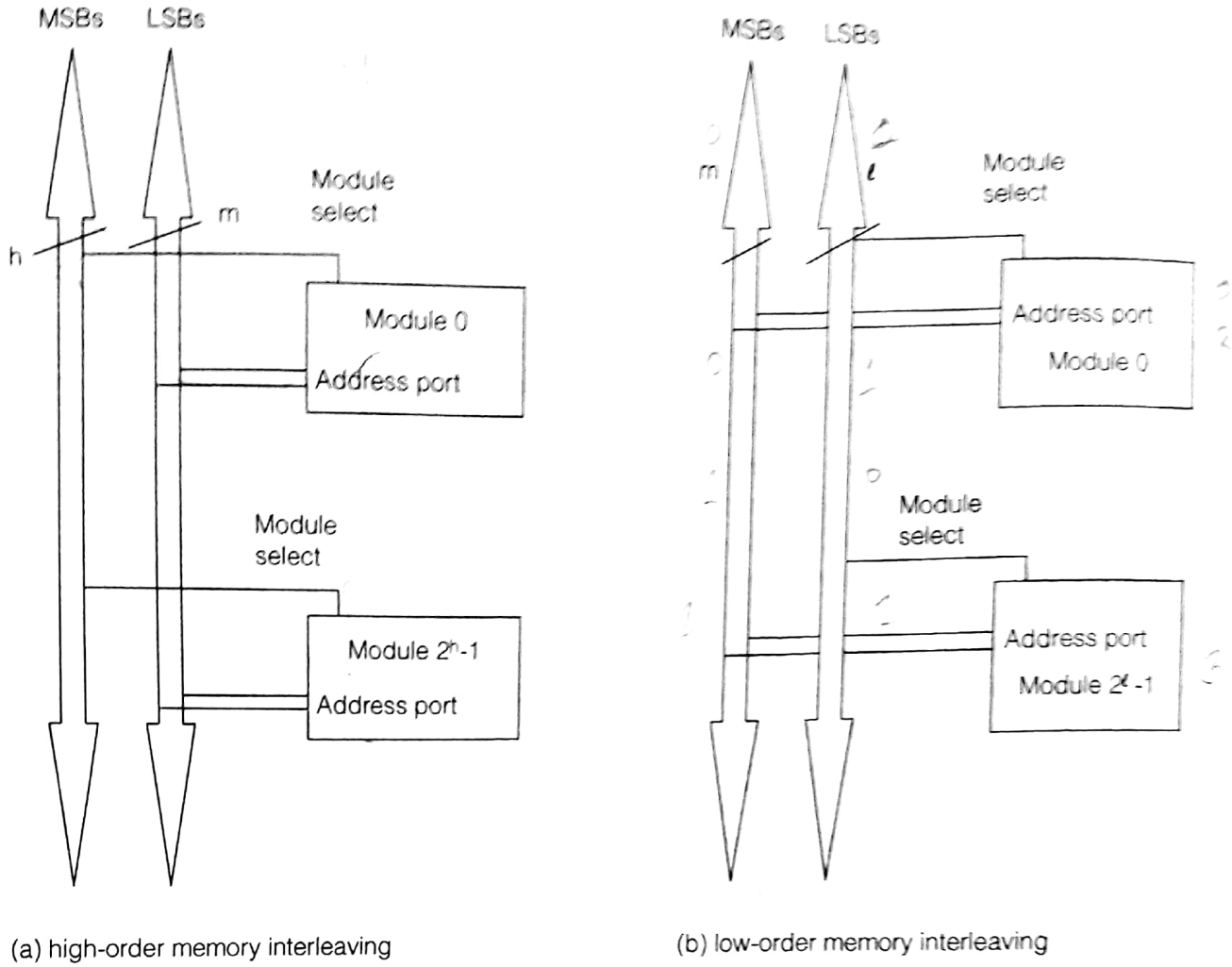


Fig. 7.25 High- and Low-Order Memory Interleaving Schemes

In the high-order memory interleaving scheme, consecutive addresses get mapped onto the same memory module. The general form of the addresses that gets mapped onto the ith module is given by

$$i.M + j; \qquad 0 \le i \le 2^h - 1 \quad \text{and} \quad 0 \le j \le 2^m - 1.$$

The high-order memory interleaving scheme is commonly used in the design of micro computer system. Based on the memory requirements of a micro computer system only a subset of the memory address space of the processor is mapped onto physical memory modules. The mapped address space can easily be expanded by adding new memory modules without changing the modules select lines of the existing modules. Successive instructions of a program reside on successive address locations and thus programs get confined to one (or more) memory module(s). If a memory module becomes faulty only those programs that reside in that module cannot be run.

Low-Order Memory Interleaving. In the low-order memory interleaving, out of the total of $m + \ell$ address lines, the ℓ low-order address lines are used for module selection, while the m high-order address lines are connected to the address ports (refer Figure 7.25b). There

are $2^t = N$ modules, each of capacity 2^m. Modules are indexed from 0 to $2^t - 1$. Addresses that get mapped onto the ith module are of the form

$$i + Nj; \qquad 0 \le i \le 2^t - 1, \qquad 0 \le j \le 2^m - 1$$

Simple Low-Order Memory Interleaving. Consider the case in which we want to access successive memory locations in sequence. In order to exploit the maximum amount of parallelism in accessing the modules, all the modules are permanently enabled as shown in Figure 7.26. The high-order address lines are connected to the address ports. If the memory read command is asserted, all the 2^t modules access the data in parallel. After T time units,

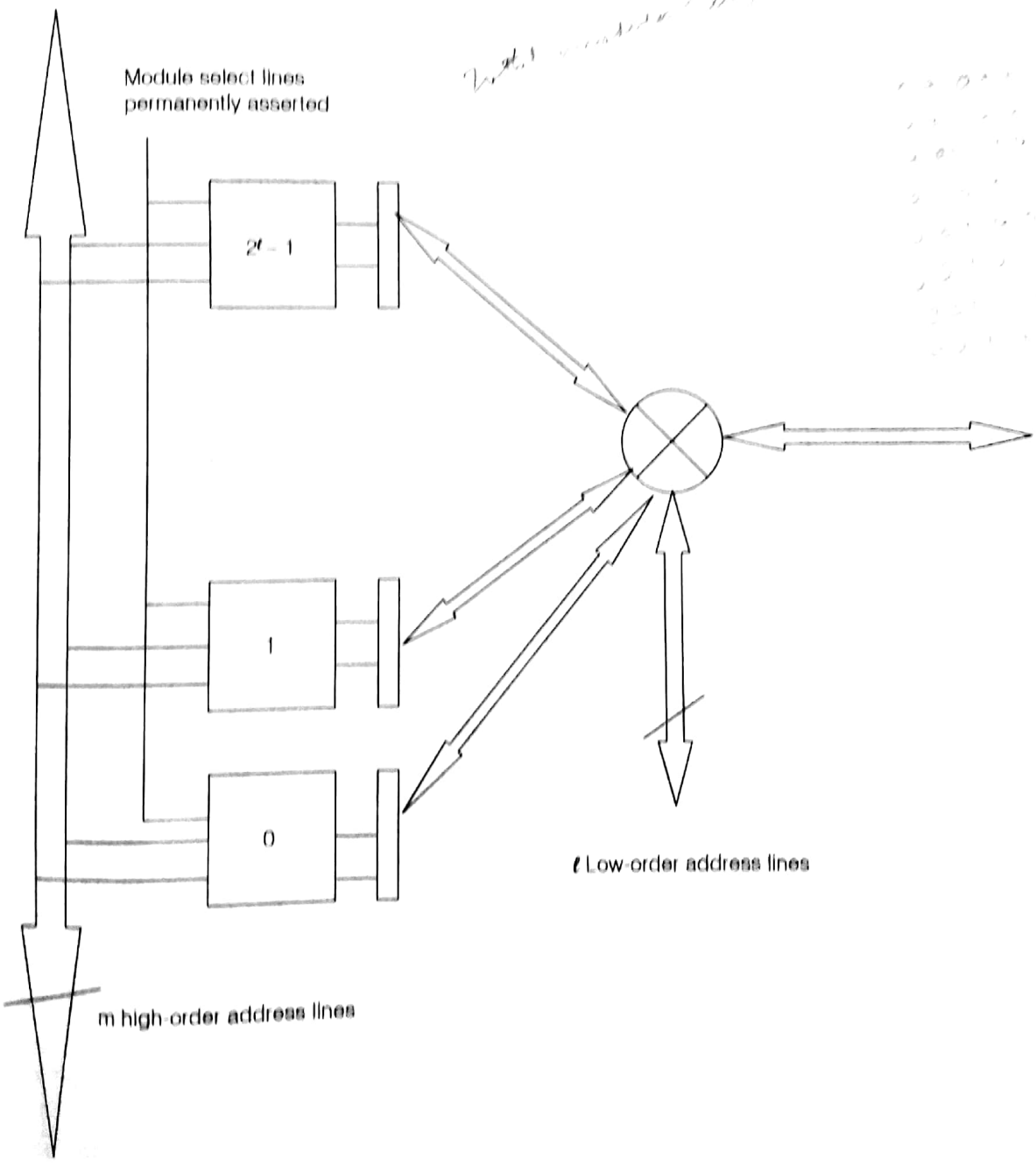


Fig. 7.26 Simplex Low-Order Memory Interleaving

If we now increment the address given to the address port of the modules, then another set of 2 data will be available after T time units.

The data accessed are stored in the latches connected to the data port of the modules. The latch is controlled by a clock having latching pulse that occurs at regular interval of T time units. A multiplexer is used to route the data from the data port of the modules into a single data bus. The lower order address bus is used to control the multiplexer. The multiplexer switches between the data ports at a rate of $T/2$. The effective memory access time for the low order interleaved memory is $T/2$, while fetching data stored in successive memory locations.

The primary advantage of the low-order memory interleaving is that parallel access of data stored in successive addresses is possible leading to increased memory bandwidth. However, the disadvantage is that if a module fails, then all those programs which have a part of them present in that failed module cannot run.

Figure 7.27 shows a simple low-order interleaved memory for the case of $\ell = 3$. For the case of $\ell = 3$, there would be 8 modules and addresses that get mapped onto the ith module $(0 \leq i \leq 7)$ would be of the form $i, i + 8, i + 16, \ldots$ as shown in Fig 7.27.

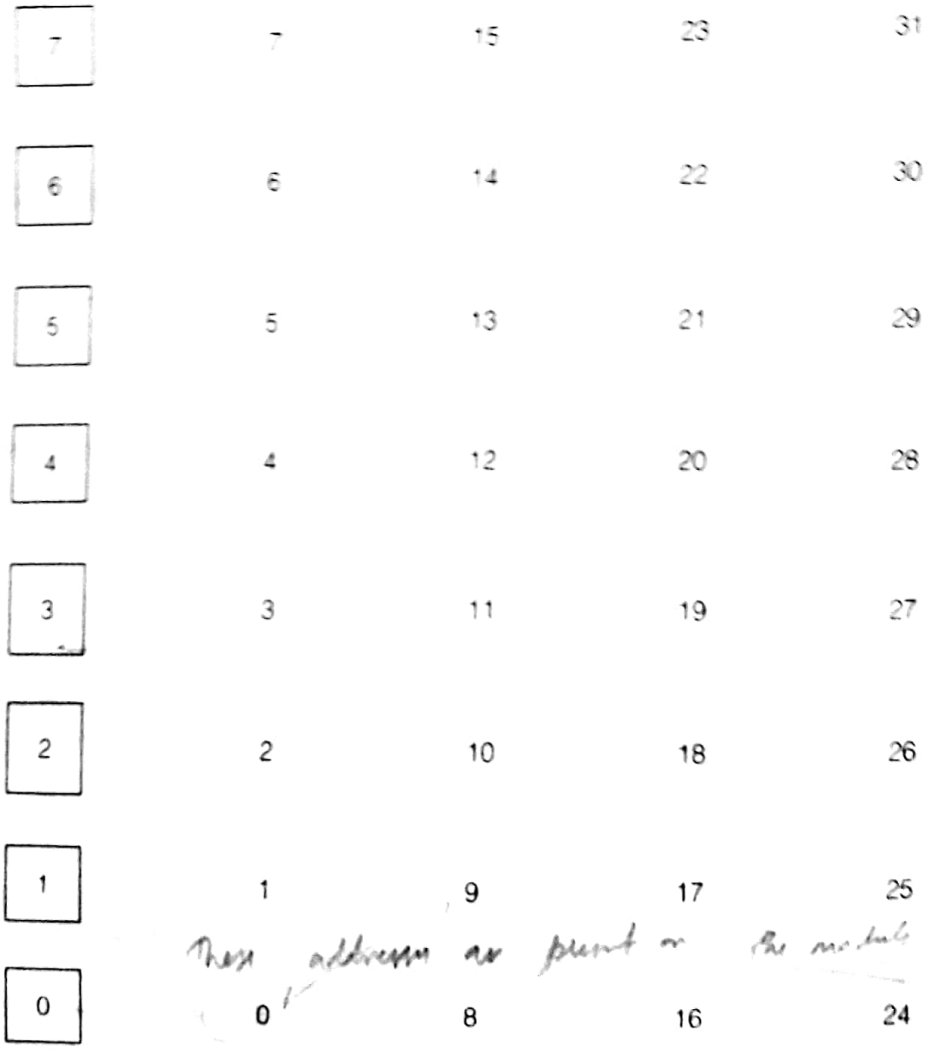


Fig. 7.27 Mapping of Address for $\ell = 3$ in Simple Low-Order Interleaved Memory

Often need arises to access data which are stored in addresses which are separated by a constant skip distance. If the skip distance is d, the pattern of addresses to be accessed will look like i, $i + d$, $i + 2d$, $i + 3d$, ... Example situations wherein data has to be accessed with constant skip distance are as follows. Suppose we want to access higher precession data such as accessing a 32-bit data in a byte-addressable memory, then the skip distance would be four. Consider the access to multidimensional array, for example we have stored a matrix row-wise, that is, first row followed by second row and so on. If we want to access the matrix elements column-wise, this skip distance would be equal to the number of columns in the matrix.

Let us analyze the performance of the simplex low-order interleaved memory design while accessing addresses with a constant skip distance. Figure 7.27 shows the mapping of first 32 addresses onto 8 modules. For a skip distance of $d = 2$ the sequence of addresses that are accessed would be i, $i + 2$, $i + 4$,.... If i falls on the 0th module, the sequence of modules that needs to be accessed would be 0, 2, 4, 6, 0, 2, 4, 6,.... If i falls on an odd (even) numbered module, then for d = 2, all further access will only get mapped onto odd (even) numbered modules. This means that only 4 modules can be accessed in parallel, and the effective memory access time is given by,

$$T_{\text{eff}} = \left(\frac{T}{8} \times 2\right) = \frac{T}{4}.$$

For the case of $i = 0$, $\ell = 3$ and $d = 3$ the address sequence would be 0, 3, 6, 9, 12, 15, 18, 21,... and the corresponding sequence of modules onto which the addresses get mapped would be 0, 3, 6, 1, 4, 7, 2, 5,.... We find that over a sequence of eight successive references, the addresses gets mapped onto different modules. From the first look it would appear that these eight references can be made parallel. In simple low-order interleaving (Figure 7.26) all modules get the same address. However, it can be seen that for the above address sequence the value of the higher order m address lines would be the same only for the first three accesses (that is, 0, 3, 6) and would change to a different value for the next three accesses (that is, 9, 12,15) and so on. Therefore the effective memory access time for simplex low-order memory interleaving is given by the formula

$$T_{\text{eff}} = \left(\frac{T}{N} \times s\right) \quad \text{for } s < N \quad \text{and} \quad T_{\text{eff}} = T \quad \text{for } s \geq N,$$

where $N = 2^{\ell}$ and s is the skip distance.

Complex Low-Order Memory Interleaving. In the simplex low-order interleaving we saw that even though a sequence of successive references fall on to different memory modules, it may not be possible to perform parallel access from the different memory modules, because the common address that goes to the address ports of the module may be different for different references. The complex low-order memory interleaving removes the above disadvantage by providing latches before the address ports also so that different addresses can be stored in these address latches for enabling parallel access.

Figure 7.28 shows the block diagram of the complex low-order memory interleaving scheme. A line from each memory module to the controller indicates whether the module is currently busy or idle. Memory reference made by the processor are given to a unit called memory controller. Memory controller finds the module on to which the referred memory address is mapped. If the memory module is free, the high-order address is placed on the address bus and is latched by the address latch of that memory module. The read command

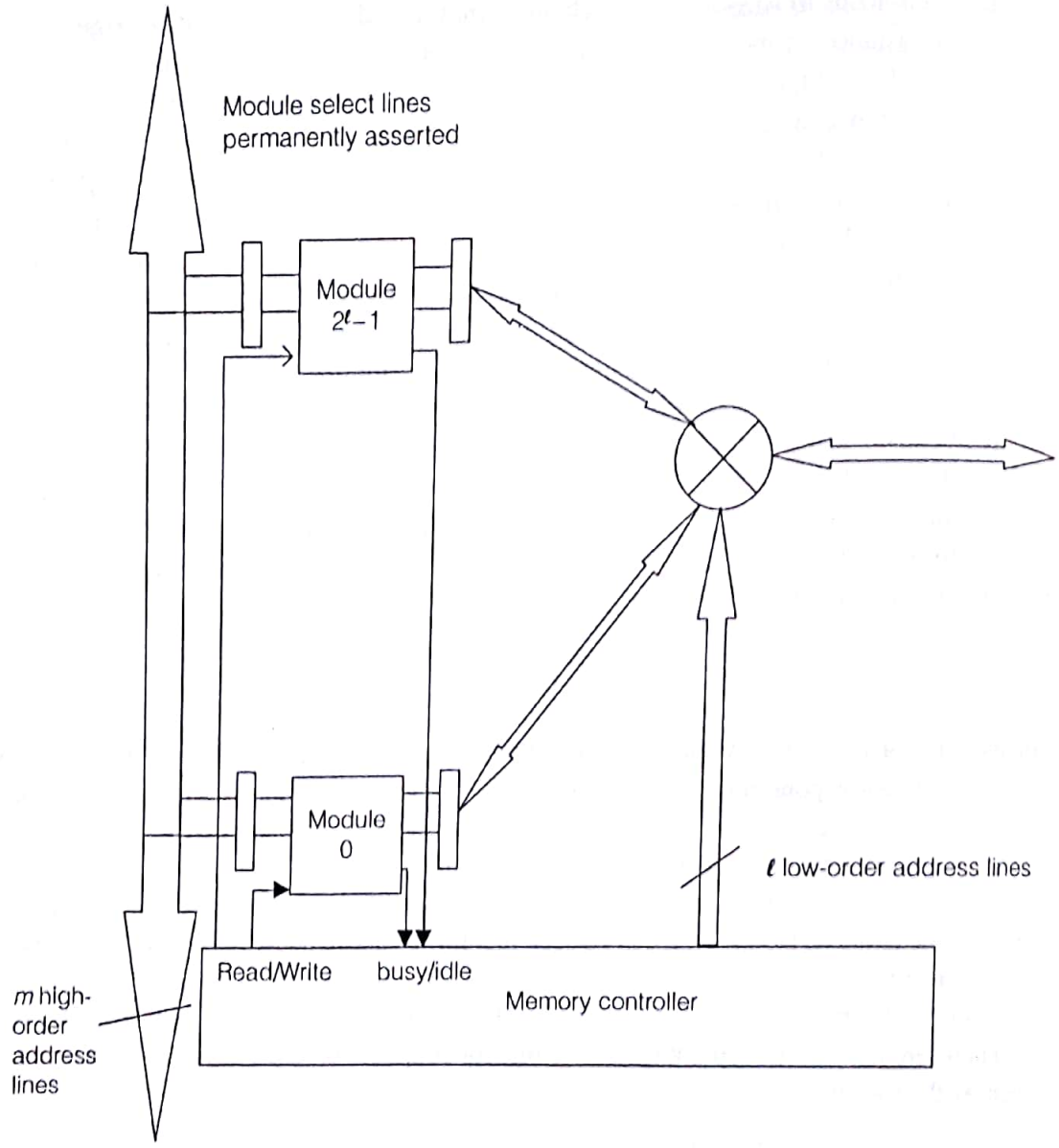


Fig. 7.28 Complex Low-Order Memory Interleaving

is given to that module. The module makes the module status line busy during the access and idle after the access is over. On detecting the change in the module status, the memory controller routes the data to the common data bus by controlling the multiplexer with appropriate low-order bits.

If a reference falls on a module that is currently busy, then the above access is deferred till the module becomes free. In the case of a memory write after latching the appropriate high-order address on the address latch the data to be stored is latched into the data latch of the module from the common data bus by controlling the multiplexer, and then the write command is given.

It can be seen that $T_{eff} = T/N$ if N and s are relative prime (that is, 1 is the only common prime factor of N and s). This is the condition that in a sequence of 'N' addresses, all those address fall on different memory module. More generally the effective access time of the high-order memory interleaving scheme is given by T/N times the product of common prime factors of N and s.

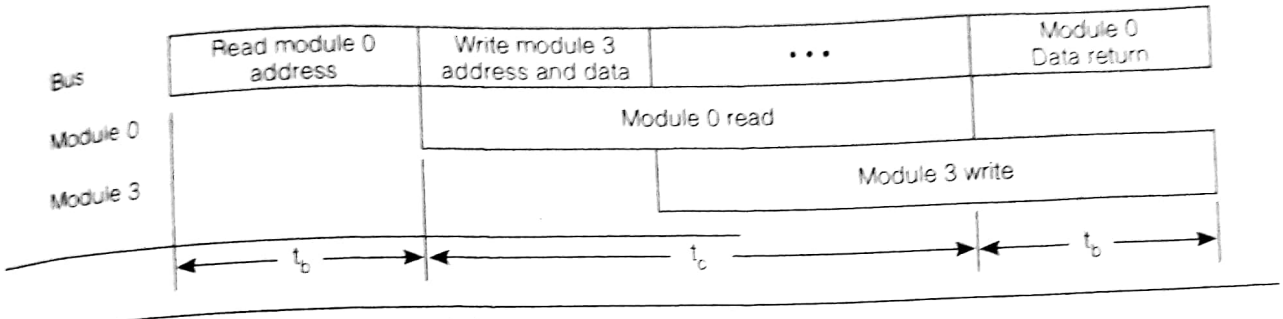
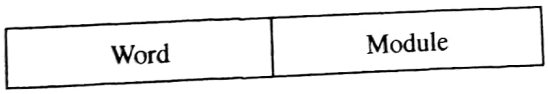


Fig. 7.29 Timing of Multiple Modules on a Bus

Multiple Modules and Interleaving. Because the memory module bus interface is separated from the internal workings of the memory system, it may have a separate timing system. Suppose t_b is the time required to transmit information (address, data, or command) over the bus to or from the memory module. Then, if the module cycle time t_c is larger than t_b, it may be useful to attach the bus to several modules and time-multiplex information transmission for all of them. Figure 7.29 shows possible activity timing. There is no benefit to having multiple modules on a bus if successive accesses are to the same module, except for clever tricks like sending the address for the next read while the data from the current one is being returned. The principle of locality says that it is more likely for consecutive addresses to be accessed at nearly the same time than widely separated ones. The probability of accessing different modules successively will thus be higher if successive words are in different modules.

Module Interleaving. With module interleaving, the memory address will be of the form:



With 2^k modules and $t_b < t_c/2^k$, it is possible to get up to a 2^k-fold increase in memory bandwidth, provided the device using the memory does not have to wait for completion of one access before starting another. Input/output using DMA usually satisfies this requirement, for example. In practice, if $t_c = q \cdot t_b$, more than q modules are interleaved on a bus. This makes it more likely that successive accesses will be to different modules, even though it does not improve the best-case speed.

Module interleaving is not consistent with the need to furnish machines with different amounts of installed memory. Since the memory module is the largest unit of structure, it is convenient to expand memory by installing more modules. But installing fewer than the maximum number of modules requires that the module be selected by the high-order address bits, so words in installed memory are numbered together. A simple, program-controlled, or even mechanically switched, *address mapping* function can solve this problem. Consider a memory system that can have up to 16 modules, selected by bits m_i, $i = 0, \ldots, 3$. Each module has 2^{28} words, selected by word-address bits w_i, $i = 0, \ldots, 27$. Table 7.4 shows the correspondences to be made between the bits of a 32-bit address presented to the memory system a_i, the word-address bits in module w_i, and the module selection bits m_i for different amounts of installed memory. The mapping is particularly simple if all of the words of a module are installed, for then the order of the word-address bits does not matter, and

variable degree of interleaving

Table 7.4 Address Mapping for Multiple Modules

System Address	a_{31}	a_{30}	a_{29}	a_{28}	...	a_3	a_2	a_1	a_0	
1 module installed	m_3	m_2	m_1	m_0	...	w_3	w_2	w_1	w_0	No interleave
2 modules installed	m_3	m_2	m_1	w_0	...	w_3	w_2	w_1	m_0	2-way
4 modules installed	m_3	m_2	w_1	w_0	...	w_3	w_2	m_1	m_0	4-way
8 modules installed	m_3	w_2	w_1	w_0	...	w_3	m_2	m_1	m_0	8-way

only bits at the two ends of the address need to be mapped, as shown in the table. For example, with four modules installed, the low-order 2 bits of the address, a_1 and a_0, are connected to the module address lines, m_1 and m_0; a_3 through a_{29} form the word address in the module, and the upper 2 address bits, which are zero for a four-module system, connect to m_3 and m_2. In the scheme shown, address bits a_4 through a_{27} always connect to w_4 through w_{27} for any configuration. The mapping can be done by a few switches or jumpers on the memory modules. We will see much more elaborate address mapping schemes in connection with the memory hierarchy.

7.3.5 Performance Trade-Offs in Memory System Design

Given the variety of possible implementations, and the several layers of hardware structure that must be analyzed, the design trade-offs in memory system design are considerable. In this section we focus on two design considerations: access-time performance and cost.

Performance and Design: Access Time. As you can see from the design of memories and modules, several levels of hardware structure must be analyzed to determine system performance. There are also several things taking place during a memory access that must be considered in determining its speed. The number of steps that can be done in parallel also depends on whether the memory is based on static or dynamic RAM. At a high level, the following things happen:

For any access:

- Transmission of address to memory
- Transmission of command (read/write) to memory
- Decoding of address by memory

For read:

- Return of data from memory
- Completion signal

For write:

- Transmission of data to memory
- Storing of data into selected cell
- Completion signal

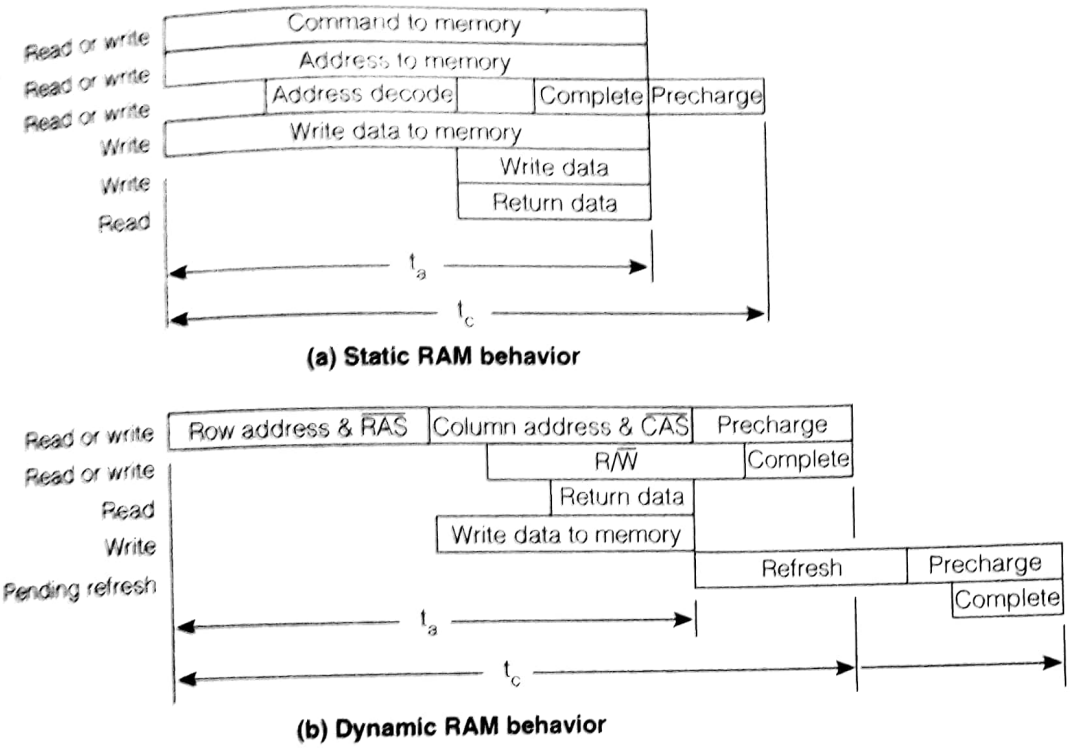


Fig. 7.30 Sequence of Steps in Accessing Memory

overlap in memory

Figure 7.30a and b show how the above operations can fit into the period of the memory cycle, what can overlap, and what must be done in sequence. It includes time for refresh of dynamic RAM and for precharge, which may occur in static RAM also. The worst-case chain of events that must occur in sequence determines access time t_a and cycle time t_c in each case. Two cycle times are possible with dynamic RAM, depending on whether a refresh is pending when the read or write is finished. If the memory module has address and data registers, separate steps to load the registers and read them out will occur at the ends of the access periods.

To get an accurate idea of the speed of a large memory system, we must look at the steps involved in a memory access in more detail. Each level of organization adds time to the information flow to and from the cells on memory chips. Figure 7.31 shows terms that must be included for a complete access- and cycle-time analysis of a static RAM memory system. It shows which steps can overlap by arranging them in one horizontal row. An optional precharge time is shown after a write. If it is required after a read, it may be overlapped with the return of data.

Illustrative Example: 70 ns SRAMs Do Not Make a 70 ns Memory Module. In this example, we compute the time to perform a read from a hypothetical memory module composed of 70 ns, 4 M static RAM chips. Assume that address bus drivers at the CPU have a 40 ns delay, and assume 10 ns for bus propagation; thus the request arrives at the memory module at 50 ns. With board select decoder chip delay of 20 ns, extra driver time of 30 ns necessary to send the select to another board, and a 20 ns delay of the decoder that selects the memory chip, the request has reached the chip at 120 ns elapsed time.

All Access	Read	Write
Command and address drivers		Data driver turn-on
Bus propagation time		Data bus delay
Board select decoder		Board data fan-out
Chip select decoder		Chip data fan-out
Cell select decoder		Cell data fan-out
	Chip read delay	Chip write time
	Chip multiplex delay	(Precharge time)
	Board multiplex delay	
	Bus time for data return	

Fig. 7.31 Timing Considerations in a Memory System Timing Analysis

Let us calculate how the 70 ns chip read access time arises. Assume that our chip is designed as a $2\frac{1}{2}$ D memory with a 6-bit decoder for row selection and a 6-bit multiplexer for bit selection, built with base 4 tree decoders, each with three logic levels. Assuming a 10 ns delay per logic level, the 70 ns access time is composed of 60 ns of gate delays and 10 ns of read time.

With a 70 ns read access time for the chip, valid data will reach the chip outputs at 190 ns. With a 30 ns chip-to-board-driver delay, data will reach the data bus drivers after an elapsed time of 220 ns. The main memory bus drivers and bus propagation would take the same 50 ns assumed to send the address to memory. Thus the total cycle time for this memory would be 270 ns, considerably greater than the 70 ns RAM chip access time!

Illustrative Example: Improving Memory Bandwidth. Consider two ways of improving the bandwidth of memories with w-bit words and a cycle time of t_c. One way is to multiplex k modules on a fast bus, and the other is to widen the word to $k \times w$ bits and have the device that is using the memory access a *superword* in time t_c. For the right kind of memory use, say a DMA device doing a block transfer, either technique can increase the bandwidth by a factor of k. The latency is still t_c after a read request before any bits appear.

superword

The performance of the two memory systems will depend on the way they are accessed. For the multiple-module system, the average cycle time $\hat{t}_c$ depends on the probability of an access being a read or a write and on the average wait time $\hat{w}$ for the referenced module to become ready:

$$\hat{t}_c = \text{p(read)} \cdot 2t_b + \text{p(write)} \cdot t_b + \hat{w}$$

The important parameters for the superword memory are the average number of words out of a k-word access that can be used on a read $\hat{k}_r$ or a write $\hat{k}_w$. Using these parameters, the average cycle time is

$$\hat{t}_c = \text{p(read)} \cdot (2t_b + t_c) \cdot \hat{k}_r + \text{p(write)} \cdot (t_b + t_c) \cdot \hat{k}_w$$

If the memory accesses are not as uniform as they are in a block transfer, the multiple-module memory probably has better performance. There is no gain from the superword design unless the next access is to one of three specific words. In the multiple-module approach, a next access to any word of a fraction $(k-1)/k$ of the memory will have a shorter access time.

Let us now estimate the costs of building the two different systems. Of course, an accurate cost calculation requires complete electronic, mechanical, and packaging specifications, identifying suppliers, and so on, but what a designer needs to decide a trade-off is an approximate calculation that treats the two designs consistently. Taking costs as proportional to the number of gates is one way to do this. A VLSI gate is not worth the same amount as a TTL decoder chip gate, so we will estimate separately for each level of structure. Doing levels separately can account for wire cost too. Each gate output is associated with a connection, and within a given level the cost of connection is probably about the same. Treating on-chip connections the same as bus wires would be wrong, but by increasing the cost of a bus driver by its output connections, we can estimate both.

The multiple-module system will have a cost of the form Cost_1 in the following equation.

$$\begin{array}{lcccccccccc} & & \textbf{A bus} & & \textbf{A reg.} & \textbf{Decode} & & \textbf{Cells} & \textbf{Data I/O} & \textbf{D bus} & \textbf{Refresh} \\ \text{Cost}_1 & = & C_1 m & + & kC_2 m & + & kC_3 2^m & + & kC_4 w 2^m & - & kC_5 w & + & C_6 w & + & kC_7 m/2 \end{array}$$

The address and data registers, the decoders, the memory cells, and the refresh circuits appear in each module, so their cost is multiplied by k. The address and data buses have m and k bits, respectively. The constants, C_i, are proportional to the costs for the different kinds of hardware required for the different structures. The superword system has only one module, but a wider word, so only the number of cells and the data register and bus costs have a factor of k, as shown in the following equation for Cost_2.

$$\begin{array}{lcccccccccc} & & \textbf{A bus} & & \textbf{A reg.} & \textbf{Decode} & & \textbf{Cells} & \textbf{Data I/O} & \textbf{D bus} & \textbf{Refresh} \\ \text{Cost}_2 & = & C_1 m & + & C_2 m & + & C_3 2^m & + & C_4 kw 2^m & + & C_5 kw & + & C_6 kw & + & C_7 m/2 \end{array}$$

Taking the difference between the two cost functions gives the following expression for $\text{Cost}_1 - \text{Cost}_2$

$$\begin{array}{lcccccccc} & & & & \textbf{A reg.} & \textbf{Decode} & & \textbf{D bus} & & \textbf{Refresh} \\ \text{Cost}_1 - \text{Cost}_2 & = & & & (k-1)C_2 m & + & (k-1)C_3 2^m & - & (k-1)C_6 w & + & (k-1)C_7 m/2 \end{array}$$

which will favor the superword memory if it is positive and the multiple-module design if negative.

If constants were all about the same size, the lower decoder cost for the superword memory would dominate, making it the better choice if performance is ignored. But constants are important in practice. Bus wires, connector pins, drivers for long, high-fan-out wires, and bus receivers to reduce errors can make C_6 very large, while VLSI integration can make C_2, C_3, and C_7 very small. Thus the cost of the wider bus alone could make the superword memory more expensive.

We have seen in this section that many layers of structure are possible in assembling RAM chips into the main memory of a computer. The next section shows that the overall memory used by a computer may not be just a complex assembly of many chips of the same type.

7.4 Memory Hierarchy

The problem of designing memory system involves three primary parameters, (1) the cost of the memory, (2) the size of the memory, and (3) the access time. It is obvious that the design will aim for a large-sized, high-speed (small access time) memory for a given cost.

Memory devices built using different technologies such as semiconductor (SRAM and DRAM), magnetic and optical storage devices, and tape drives, exhibit widely varying values for the above parameters. Therefore, a monolithic memory system built using any one of the above technology will not meet the desired design requirements of today's computer. The solution to this problem is offered by the hierarchical memory design.

The objectives of a memory hierarchy is to arrange different types of memory at various hierarchical levels such that the overall memory is large sized, but its effective access time being very small, and it is of reasonable cost. A typical hierarchy places a small-sized high-speed and costly memory close to the processor, and as levels increase the memory progressively, becomes cheap, slow, and voluminous.

We now formally define the memory hierarchy and quantify the above design criteria. A memory hierarchy having n levels is structured so that memory at level the i for $1 \le i \le n$, is characterized by the following specifications. Let c_i, t_i, s_i, b_i and g_i denote respectively the cost per bit, average access time in seconds, the size of the memory in bits, memory access bandwidth in bits/seconds, and the granularity of transfer from level i to level $i-1$. Then the following relationships hold good between levels i and $i+1$:

granularity (frequency) amount of transfer of data between 2 adjacent levels.

$$c_i > c_{i+1},\ t_i < t_{i+1},\ s_i < s_{i+1},\ b_i > b_{i+1},\ \text{and}\ g_i < g_{i+1}.$$

Inclusion Property. The mapping of the address space of the processor onto the various levels of the memory hierarchy obeys a property called as the inclusion property. The inclusion property states that the address space in level i is a subset of that in level $i+1$. All data in level i should also exist in level $i+1$. However it does not necessarily mean that the data stored in a given address A in level i is the same as that of what is stored in address A in level $i+1$.

When the processor makes a request to read from an address (belonging to the address space of the processor) the memory system first checks for the availability of the corresponding data in level 1. If the data is present in the level 1, it is accessed and given to the processor, if not the next level is checked and so on.

Coherence Property. Due to inclusion property of the memory address space there is replication of certain data at various levels of the hierarchy. Naturally one would expect that different copies of the same data present at various levels of hierarchy to be identical, in which case we say that the data is consistent or coherent. However, when the processor makes a reference to a data for reading or writing, the memory controller searches for the data in the ascending order of the levels. Therefore, the copy of a given data may be updated at lower levels, while the higher levels may have older versions of the data. Similarly, a DMA controller

data consistency

may update the copies of data at higher levels without modifying them in the lower levels. As a result, different copies of the same data may have different values at various levels. When a data is accessed by the processor or DMA controller, they would read different values for the same data. This creates the data inconsistency or coherence problem. In general, if more than one entity can access/modify data of a memory hierarchy and if the order of search used by these entities differ than we have a potential coherence problem. Unlike in a uniprocessor system, the data coherence problem is severe in the multiprocessor system.

The coherence property of the memory hierarchy demands that copies of the same data at different memory levels be identical or consistent. To maintain the data consistency, memory hierarchy generally adopts one of the following two schemes. In the first scheme, called write-through (WT), if a word is modified in the lowest level, copies of that data is immediately updated at all higher levels. The second method, called the write-back (WB), delays the update of copies at higher levels until the data being modified in the lower level is replaced or removed from that level.

Memory Hierarchy Design Problem. Hit ratio h_i of the level i is defined as the probability that an arbitrary data requested by the processor is present in the ith memory level. The miss ratio of the ith memory level is $1 - h_i$. The access frequency f_i of the ith level is the fraction of the time an arbitrary memory request made by the processor is actually satisfied by the ith level. Data is accessed from the ith level only when there are misses in level 1 to $i - 1$ and there is a hit in level i. It then follows that,

$$f_i = (1 - h_1)(1 - h_2) \ldots (1 - h_{i-1})h_i$$

The effective access time T_i of the ith level is the time it takes to access a data from the ith level. It is the sum of the individual access times t_k of all levels from 1 to i,

$$T_i = \sum_{k=1}^{i} t_k$$

In general, t_k includes the time wasted due to memory conflicts at level k and the delay involved in switching from level $k - 1$ to level k. Finally, the effective access time of the n-level memory hierarchy is given by

$$T = \sum_{i=1}^{n} f_i T_i$$

The total cost of the memory hierarchy is

$$C = \sum_{i=1}^{n} c_i s_i$$

The effective size of the hierarchy is just s_n, the size of the last memory, because the levels lower than n simply store subsets of the data stored in level n. The design problem of the memory hierarchy can now be stated as minimization of the effective access time of the hierarchy T, subject to a given memory system cost C and size s_n.

Program locality. The sequence of references in terms of the memory addresses generated by a program during its execution is called a reference string. A reference string consisting of n successive references of a program is represented as $R(n) = \{r(1), r(2), \ldots r(n)\}$ where $r(i)$, $1 \le i \le n$, is an address belonging to the virtual address space of the program. It is well

reference string

known that the sequences of references made by a program are not randomly distributed in the virtual address space but exhibits certain pattern. By understanding this pattern, it is possible to predict the future references that are likely to be made, and bring the corresponding data to levels closer to the processor. The above technique increases the hit ratios of lower levels leading to reduction in effective access time.

Locality of Reference. The characteristics of a typical program that leads to a predictable access pattern is referred to as the locality of reference. The locality of reference can be classified into three categories. They are (1) the temporal locality of reference, (2) the spatial locality of reference and (3) the sequential locality of reference.

temporal locality

The temporal locality of reference states that there is a tendency for a process to repeatedly refer certain addresses over a small period of time. If we plot the accesses made by a program to a particular memory location in the time axis, then the markings would be clustered. In other words, at a given point of time, the references made in the recent past are likely to be repeated in the near future. Programming language constructs that lead to the temporal locality of reference are loops, temporary variables, and access to stacks.

spatial locality

The spatial locality of reference abstracts the tendency of a process to make future references in the neighborhood of the last reference. Modular programming technique and program characteristics such as loops and array data structures lead to spatial locality of reference. There is a special kind of spatial locality of reference that is often strongly exhibited by the programs which is called as the sequential locality of reference.

sequential locality

The principle of sequential locality of reference states that the next reference to be made by a program is more likely to be the address that is next to the previously referred address. The normal flow of control of a program (that is instructions being fetched one after another and executed) and traversals of a sequential data structures such as arrays enforce spatial and sequential locality of reference.

working set

The principle of locality states that the references made by a program are localized in time and address space and hence restricted to a small subset of the virtual address space which is called as the working set of the program. The working set of program depends upon various parameters that includes (1) the level i in which the working set is defined, (2) the time t at which the working set is defined, (3) the time interval Δt over which the working set is defined, (4) the size s_i of the working set, and (5) the hit ratio h_i of the working set at the level i.

Formally the working set denoted as $W(i, t, \Delta t, s_i, h_i)$ denote a subset of virtual address space such that it is of size s_i and if this subset is brought to level i, the hit radio of level i would be h_i for memory references made in the time interval $(t, t + \Delta t)$. It is obvious W is a function of t. It is to be noted that the parameters of the working set are all not independent. For example, for a given i, fixing t, Δt and s_i would automatically fix h_i. Similarly fixing Δt and h_i would demand a certain value for s_i. It is the role of the memory management polices to identify at any given time t, a working set for a reasonable value of Δt such that the size of the working set is s_i and it is the one that maximizes h_i. Further the memory management policy should constantly update the level i with the current working set.

The spatial locality of reference helps us to determine the size of the unit of transfer (such as cache block, page) between levels. The temporal locality aids in identifying the number of units to be prefetched from one level to another level. The sequential locality of reference can be used to distribute successive units to concurrently operating devices at a certain level of the hierarchy to exploit parallel access of data.

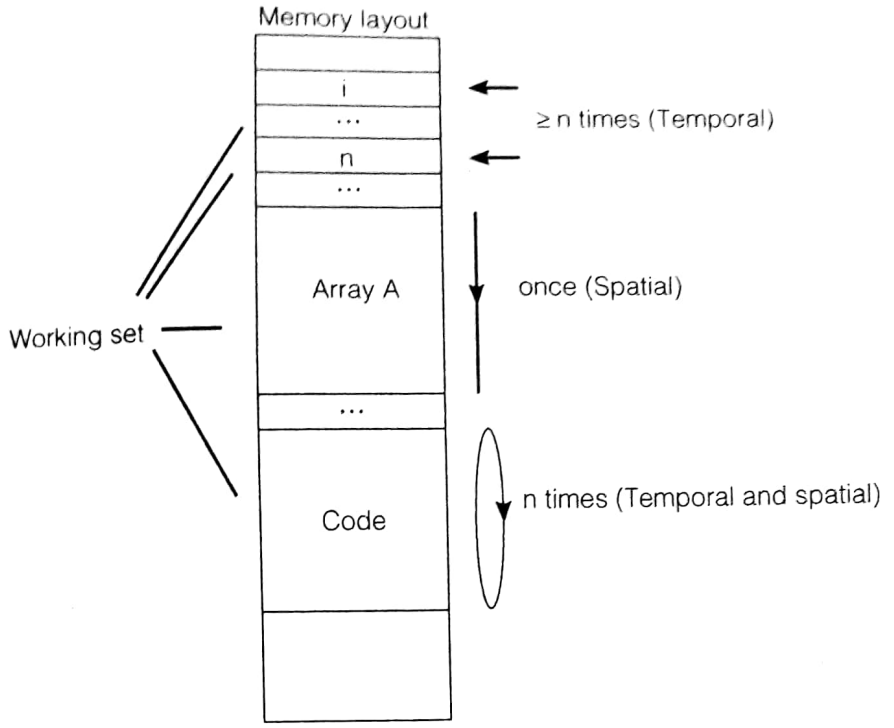


Fig. 7.32 Temporal and Spatial Locality of the **for** Loop

Illustrative Example: Temporal and Spatial Locality in a Code Fragment. Both temporal and spatial locality can be found in various combinations in many fragments of code. Consider the following C code that initializes the first n elements of the array A to zero:

```
for (i = 0; i < n; i++)
A[i] = 0;
```

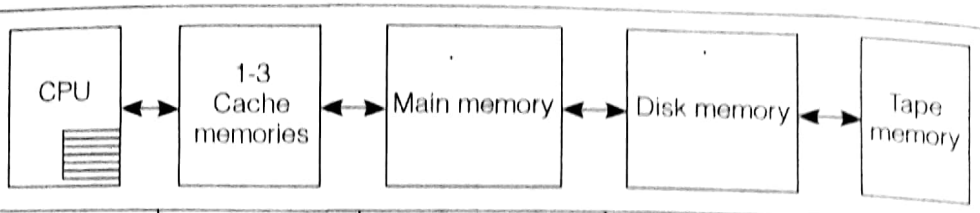
In this statement, variables i and n exhibit temporal locality, as each one of them is accessed multiple times as the loop is performed. The array A exhibits spatial locality, as each element is accessed in turn. The code body itself exhibits both temporal and spatial locality: it is executed n times (temporal) and the instruction sequence of the code is executed one instruction after the other (spatial). Figure 7.32 shows a memory layout of the loop. Notice how i and n are "hit" n or more (in the case of i) times during execution, while A is scanned once, and the code sequence executed n times. During the execution of the statement the working set is composed of i, n, A, and the code body.

7.4.1 A Typical Memory Hierarchy

Table 7.5 shows how a typical memory hierarchy might be organized, as well as some approximate values for capacity, latency, bandwidth, and cost.

The registers, which are internal to the CPU, are the first level of memory. One step outward in the hierarchy comes RAM. Many high-performance systems employ two or three levels of cache memory in addition to the conventional main memory. Beyond the main memory the table shows a disk system, and beyond that a tape system. Conventionally, high-speed memory is referred to as *primary memory*, disk drives as *secondary*

Table 7.5 The Memory Hierarchy, Cost, and Performance

Component					
Access type	Random access	Random access	Random access	Direct access	Sequential access
Capacity, bytes	64–1024	8 KB–4 MB	64 MB–2 GB	10–200 GB	1 TB
Latency	.4–10 ns	0.4–20 ns	10–50 ns	10 ms	10 ms–10 s
Block size	1 word	16 words	16 words	4 KB	4 KB
Bandwidth	System clock rate	system clock rate - 80 MB/s	10–4000 MB/s	50 MB/s	1 MB/s
Cost/MB	High	$10	$0.25	$0.002	$0.01

primary, secondary, and tertiary memory

memory, and tape, if it exists, as *tertiary memory*. Observe the large differences in capacity, bandwidth, latency, and cost between primary and secondary memory.

Conceptually, each adjacent pair in the hierarchy will have interfaces like those in Figure 7.1, with the exception that the actual physical addresses and data passed between any two interfaces may be of a different form than the format of addresses and data used between CPU and main memory. For example, the disk level of the hierarchy may require addresses specified as track and sector numbers on the disk, and may transmit data in 4–8 K blocks. Considering that block sizes may range from 1 byte to several KB, while access times may range from a few nanoseconds to a few seconds—a factor of 10^9—issues involved in understanding and managing the trade-offs are of considerable interest.

block transfer in memory hierachy

Since the capacity of each level is likely to be smaller than that of the next level farther out in the hierarchy, instructions and data must be continually passed from level to level during the course of processing. This transfer of blocks back and forth in the hierarchy takes place transparently to the user's program. Since access times to cache memory are close to the CPU clock period, the cache control mechanism must be implemented in hardware. In contrast, more leisurely disk access times, usually measured in tens of thousands of clock cycles mean that the control mechanism can be implemented partially in software, by routines within the operating system.

Different technologies for secondary-level memory can lead to very different miss response times. Table 7.5 shows how different the sizes and speeds of different levels of the hierarchy may be. Two different regimes, in which miss response time differs by three to four orders of magnitude, lead to very different trade-offs in the determination of things like information placement, replacement, and what to do with the processor during miss response. In the ***cache regime***, secondary-level latency is only about 10 times as

cache regime

long as primary-level access time, making it possible for the computer to wait for miss responses if they occur frequently enough. In the *disk regime*, secondary latency is 10,000 to 100,000 times as long as primary access, making it essential that the computer system not be kept idle waiting for the miss response.

disk regime

Disks and Virtual Memory in the Memory Hierarchy. Historically, the first two-level memory hierarchy was magnetic core memory and magnetic drum, with an access-time difference of about a factor of 10,000. This evolved into today's semiconductor main memory paired with magnetic disk, which has an access-time ratio of about 100,000. A hierarchy consisting of main memory and disk is known as *virtual memory*, which is covered in more detail in Section 7.6.

virtual memory

The most important thing about accessing the disk is that the computer cannot be kept idle for tens of thousands of instruction times waiting for miss response. The processor must be reassigned to programs other than the one causing the miss, if only to run disk driver software. Two types of work are assigned to the processor. The processor assists in miss response, using programs which are guaranteed to be stored at the primary level, and it is used to do *multiprogramming*, which is sharing the processor among several independent programs that simultaneously occupy memory. When a miss occurs for one multiprogrammed job, the processor is given to another one that is ready to run.

multiprogramming

Processor assistance in miss response is not limited to disk I/O. Software can also be used to make placement and replacement decisions and compute disk addresses. As a result, there can be great flexibility in primary memory placement in the disk regime, provided the placement scheme does not interfere with fast translation to a primary address on a hit.

The Cache Regime. A difference of only about a factor of 10 in access times makes it possible for the computer to wait for the secondary level to respond to a miss, provided the miss ratio is small. A small, fast, semiconductor memory, possibly static RAM, connected to a larger and slower memory of a different semiconductor technology, probably dynamic RAM, fits this regime and is called a *cache*. Cache is usually thought of as the lowest level of the hierarchy, so its access time matches the processor speed. Where there are two cache levels between the processor and main memory, the faster level is called the *primary cache* and the slower level is called the *secondary cache*.

primary cache
secondary cache

Since the processor waits for a cache miss, software cannot be used in miss response, and everything must be done by hardware. This means that placement and replacement decisions and secondary address formation must be kept simple, so that they can be done quickly with limited hardware. For this reason, there is usually less placement flexibility in cache than in the primary level of a virtual memory system. Because cache memory is so fast, it is essential that primary address formation on a hit be extremely fast. Also, because primary- and secondary-level speeds are not very different, it may be feasible for certain types of access to bypass the cache and to go directly between the processor and secondary level.

The Cache Miss versus the Main Memory Miss. The cache hit time is typically 1–2 clock cycles, and the miss penalty is only a few 10's of clock cycles to bring the desired block into the cache from main memory. A main memory miss penalty typically runs to the hundreds of thousands of clock cycles to bring the block in from disk. As a result, cache miss rates of 1–2% are tolerable, whereas main memory miss rates must be 0.001% or less to avoid serious effects on system performance.

7.5 The Cache

Cache operations are transparent to the running program. The program issues effective addresses and read or write requests, and these requests are satisfied by memory. Whether it is the cache or main memory that satisfies the request is unknown to the program. Traffic to and from the CPU is in the form of words. Traffic between the cache and main memory is in the form of blocks. These cache blocks are sometimes referred to as *cache lines*.

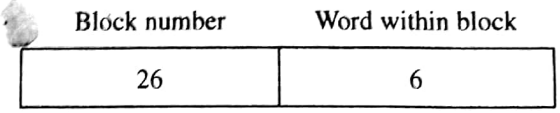
cache lines

The Mapping Function. Figure 7.33 shows a schematic view of the cache mapping function. The mapping function is responsible for all cache operations. It is implemented in hardware, because of the required high-speed operation. The mapping function determines the following:

- Placement strategies—where to place an incoming block in the cache.
- Replacement strategies—which block to replace when a cache miss occurs.
- Read and write policies—how to handle reads and writes upon cache hits and misses.

Three different types of mapping functions are in common use: associative, direct-mapped, and block-set-associative. The latter function is actually a combination of the first two, and is sometimes referred to as set-associative. We discuss each mapping method in more detail after first defining memory fields that are used in selecting the cache block and the word within that block.

Memory Fields. We divide main memory addresses into fields for this discussion. These fields partition the main memory address into blocks and words within the blocks.



In the following example, a 32-bit main memory address is partitioned into two fields, a low-order field specifying the word in a 64-byte block, and a high-order field specifying the block number. Thus there are 2^{26} or 64 M 64-byte blocks in the example. The block

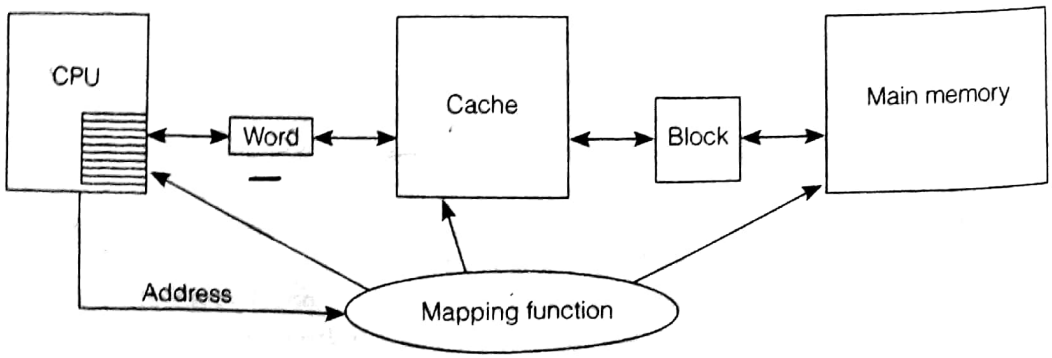


Fig.

number field may be further partitioned for purposes of finding the block in the cache. *In the following discussion of caches, we equate bytes and words, for simplicity's sake.* We also use 16-bit main memory addresses and 8-byte blocks in the following discussions to make the examples easier to comprehend, although such a small main memory address space is unrealistic in practice.

Associative-Mapped Caches. Associative-mapped caches are the simplest to understand, so we discuss them first. In *associative mapping*, any block from main memory can be placed anywhere in the cache. After being placed in the cache, a given block is identified uniquely by its main memory block number, referred to as the *tag*, which is stored inside a separate tag memory in the cache.

associative mapping

tag

Regardless of the kind of cache, a given block in the cache may or may not contain valid information. For example, when the system has just been powered up, and before the cache has had any blocks loaded into it, all the information there is invalid. The cache maintains a *valid bit* for each block, to keep track of whether the information in the corresponding block is valid.

valid bit

Figure 7.34 shows the various memory structures in an associative cache. There is the cache itself, containing 256 8-byte blocks, or lines, a 256 × 13-bit tag memory for holding the tags of the blocks currently stored in the cache, and a 256 × 1-bit memory for storing the valid bits. Main memory contains 8,192 8-byte blocks. The figure indicates that main memory address references are partitioned into two fields, a 3-bit word field describing the location of the desired word in the cache line, and a 13-bit tag field describing the main memory block number desired. The 3-bit word field becomes essentially a "cache address," specifying where to find the word if indeed it is in the cache. The remaining 13 bits must be compared against every 13-bit tag in the tag memory to see if the desired word is present. In the figure, main memory block [illegible] has been stored in cache block 255, and so tag entry 255 is 2. Main memory block 119 h[illegible]n stored in the second

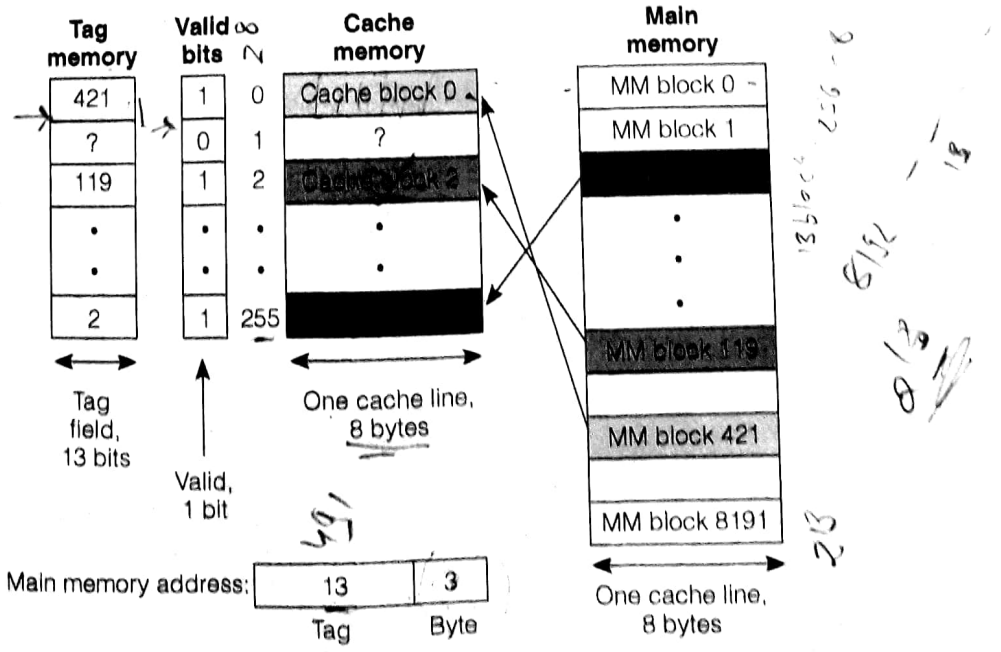


Fig. 7.34 Associative Cache

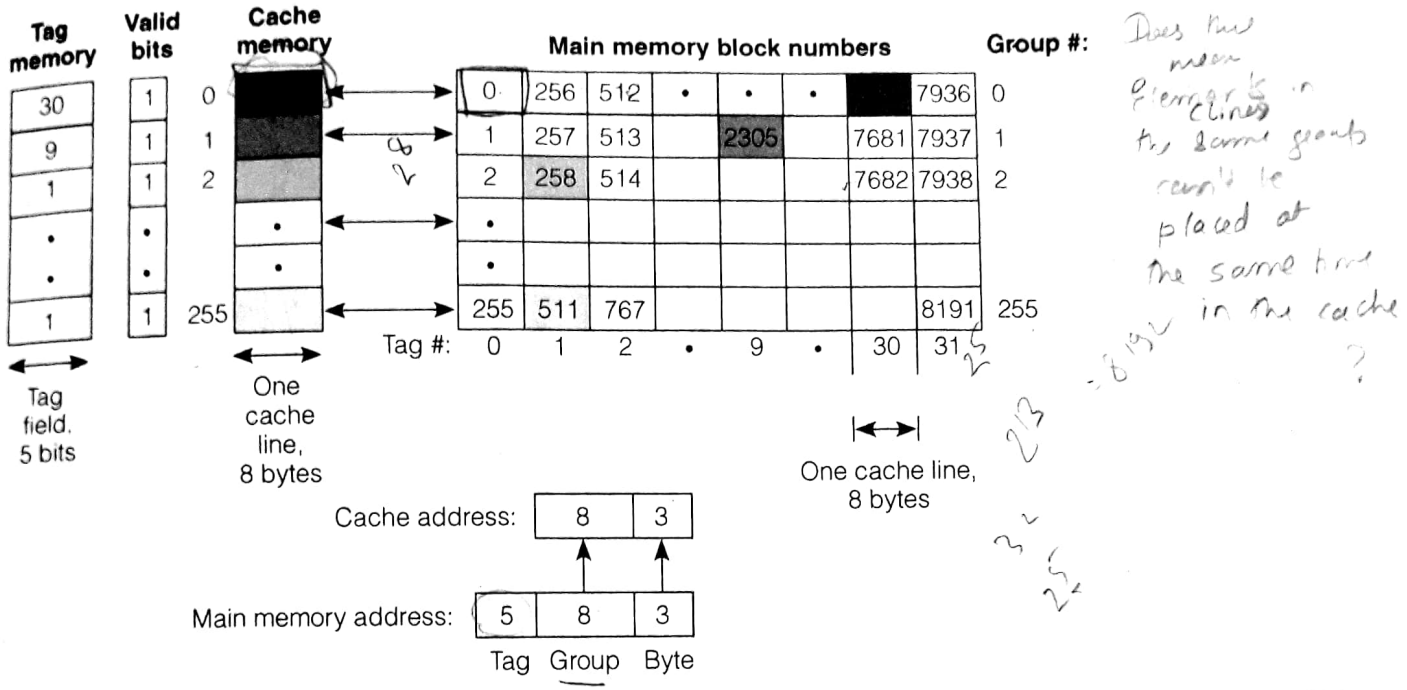


Fig. 7.36 Direct-Mapped Cache

Direct-Mapped Caches. *Direct-mapped* caches form the other extreme, where a given main memory block can be placed in one and only one place in the cache. Figure 7.36 shows an example of a direct-mapped cache. For simplicity, the example again uses a 256 block × 8-byte cache, and a 16-bit main memory address. Main memory has been laid out in a rectangular array in the figure to illustrate block placement in the cache. The main memory in the figure has 256 rows, or groups, by 32 columns, still yielding $256 \times 32 = 8,192 = 2^{13}$ total blocks, as before. The cache in the example contains 256 blocks, 1 for each group. The blocks in a given row can be placed *only* in the corresponding (1-block) row of the cache. Notice that the main memory address is partitioned into three fields. The word field still specifies the word in the block. The group field specifies which of the 256 cache locations the block will be in, if it is indeed in the cache. The tag field specifies which of the 32 blocks from main memory is actually present in the cache. Now the cache address is composed of the group field, which specifies the address of the block location in the cache, and the word field, which specifies the address of the word in the block. As before, there is also a valid bit specifying whether the information in the selected block is valid. The figure shows block 7680, from group 0 of main memory placed in block location 0 of the cache (as it must be) and the corresponding tag set to 30. In like manner, main memory block 258 is in main memory group 2, column 1 in main memory, so it is placed in block location 2 of the cache, and the corresponding tag memory entry is 1.

The tasks required of the direct-mapped cache in servicing a memory request are shown in Figure 7.37. The figure shows the group field of the memory address being decoded, ①, and used to select the tag of the one cache block location in which the block must be stored if it is in the cache. If the valid bit for that block location is set, ②, then that tag is gated out, ③, and compared with the tag of the incoming memory address, ④. A cache hit gates the cache block out, ⑤, and the word field selects the specified word from the block, ⑥. Only one tag needs to be compared, resulting in considerably less hardware than in the associative memory case.

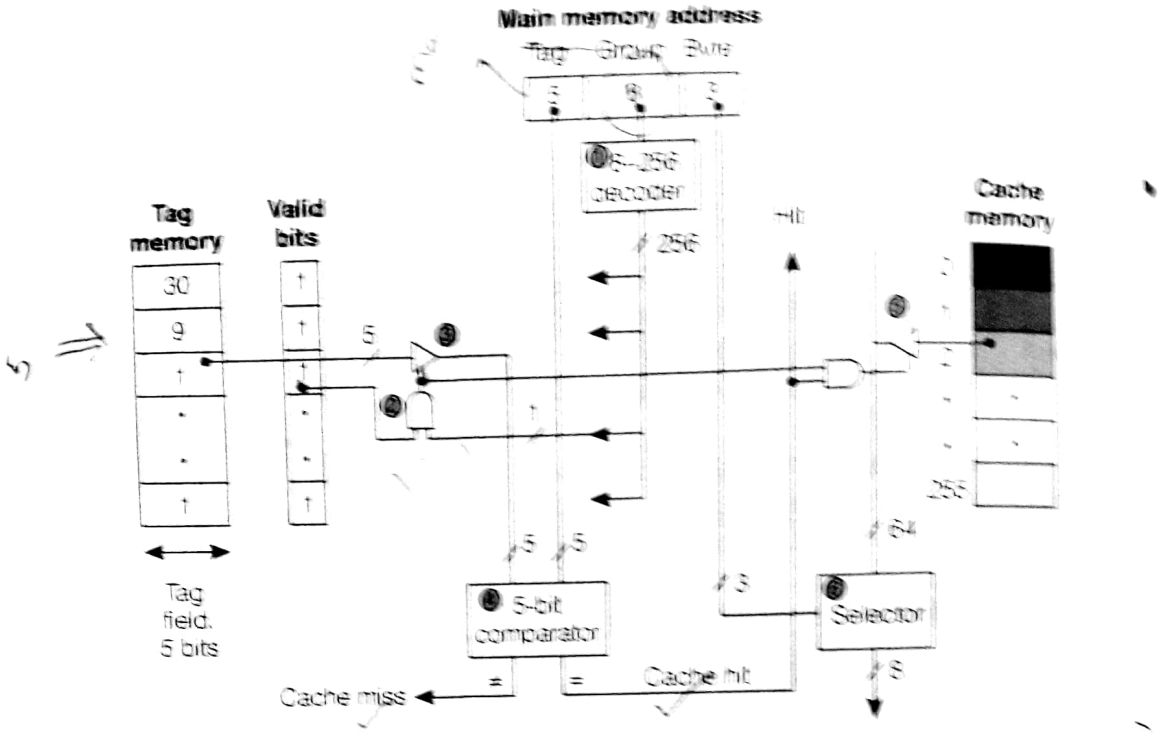


Fig. 7.37 Direct-Mapped Cache Operation

The direct-mapped cache imposes a considerable amount of rigidity on the cache organization. It relies on the principle of locality of reference for its success. Examining Figure 7.33, we see that as long as the working set of the executing program consists of not more than 256 contiguous blocks, every reference will be a cache hit. If any program reference strays out of a $256 \times 8 = 2$ K byte cacheful of memory, a cache miss will occur.

The direct-mapped cache has the advantage of simplicity, but the obvious disadvantage that only a single block from a given group can be present in the cache at any given time. If two blocks from the same group are frequently referenced, perhaps by being part of the same loop, for example, *thrashing*—the repeated moving of the two blocks in and out of the cache—will occur. An obvious improvement in the direct-mapped cache would be to compromise and allow more than one block from a given group to be stored in the cache, in a *block set* of blocks that is associatively searched. This is known as a *block-set-associative cache*.

thrashing

block set

Block-Set-Associative Caches. Block-set-associative caches share properties of both of the previous mapping functions. The set-associative cache is similar to the direct-mapped cache, but now more than one block from a given group in main memory can occupy the same group in the cache. Assume the same main memory and block structure as before, but with the cache being twice as large, so that a set of two main memory blocks from the same group can occupy a given cache group.

Figure 7.38 shows a two-way-set-associative cache that is similar to the direct-mapped cache in the previous example, but with twice as many blocks in the cache, arranged so that a set of any two blocks from each main memory group can be stored in the cache. The main memory address is still partitioned into an 8-bit set field and a 5-bit tag field, but now there are two possible places in which a given block can reside, and both must be searched associatively.

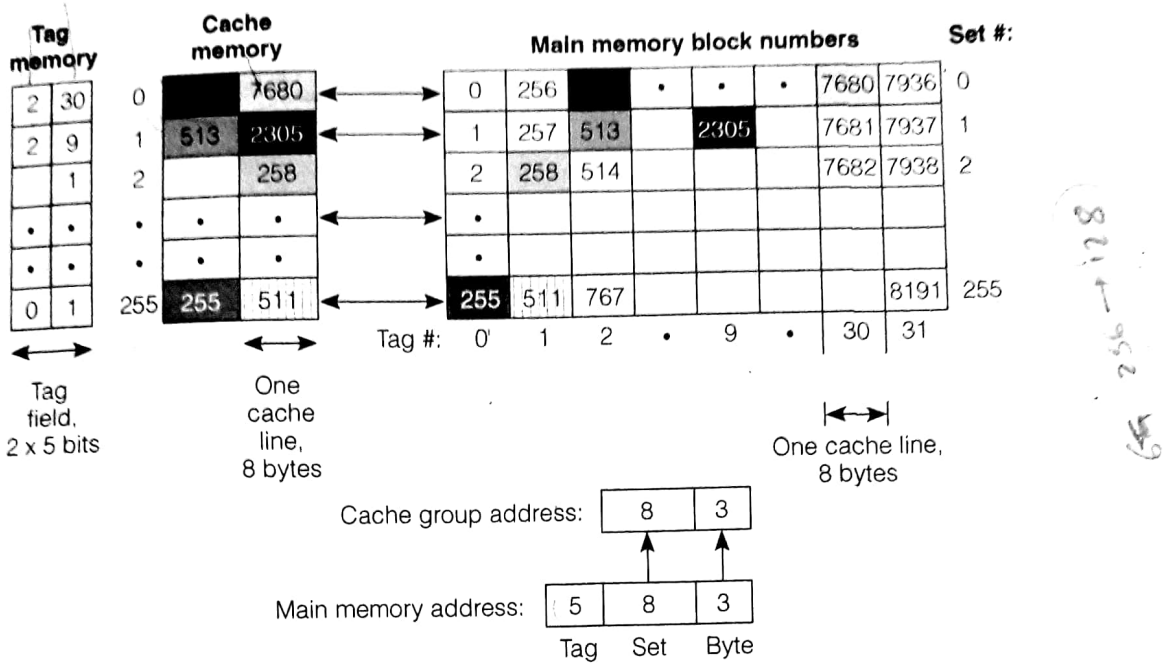


Fig. 7.38 2-Way Set-Associative Cache

The cache group address is the same as that of the direct-mapped cache—an 8-bit block location and a 3-bit word address. The figure shows that the cache entries corresponding to the second group contain blocks 513 and 2304, for example. The group field, now called the set field, is again decoded, and directs the search to the correct group, and now only the tags in the selected group must be searched. So instead of 256 compares, the cache only needs to do 2. For simplicity, the valid bits are not shown in the figure, but they must be present, 1 bit for each cache block. The cache hardware would be similar to that shown in Figure 7.36, but there would be two simultaneous comparisons of the two blocks in the set.

Sector-Mapped Cache. In sector-mapped cache organization scheme, the cache is divided into a number of sectors, each sector having one or more cache blocks. The number of blocks in a sector is usually an integral power of 2. The cache memory is divided in terms of sector frames. Each sector frame consists of a number of block frames. The size of the block frame is equal to the size of the block. The size of the sector frame is equal to the size of the sector. The placement policy we use is that any sector from the main memory can be mapped onto any sector frame in the cache. A tag field associated with the sector frame identifies the sector currently held by the sector frame. Block frames in the sector frame are automatically reserved for the corresponding blocks in the sector for which the sector frame is reserved. Figure 7.39 is used to illustrate the working of the sector mapping.

To understand the working of the sector mapping, consider that as a result of a cache miss, a block is to be brought from the main memory. We first identify the sector to which this block belongs. Let us assume that this sector is not resident in the main memory. The desired sector can be placed in any sector frame. Therefore, we choose one of the available unused sector frames in the main memory, and if none is available we use a replacement policy to identify a sector frame. The chosen sector frame is allocated for the desired sector by filling the tag field of the sector frame with the sector number of the sector which

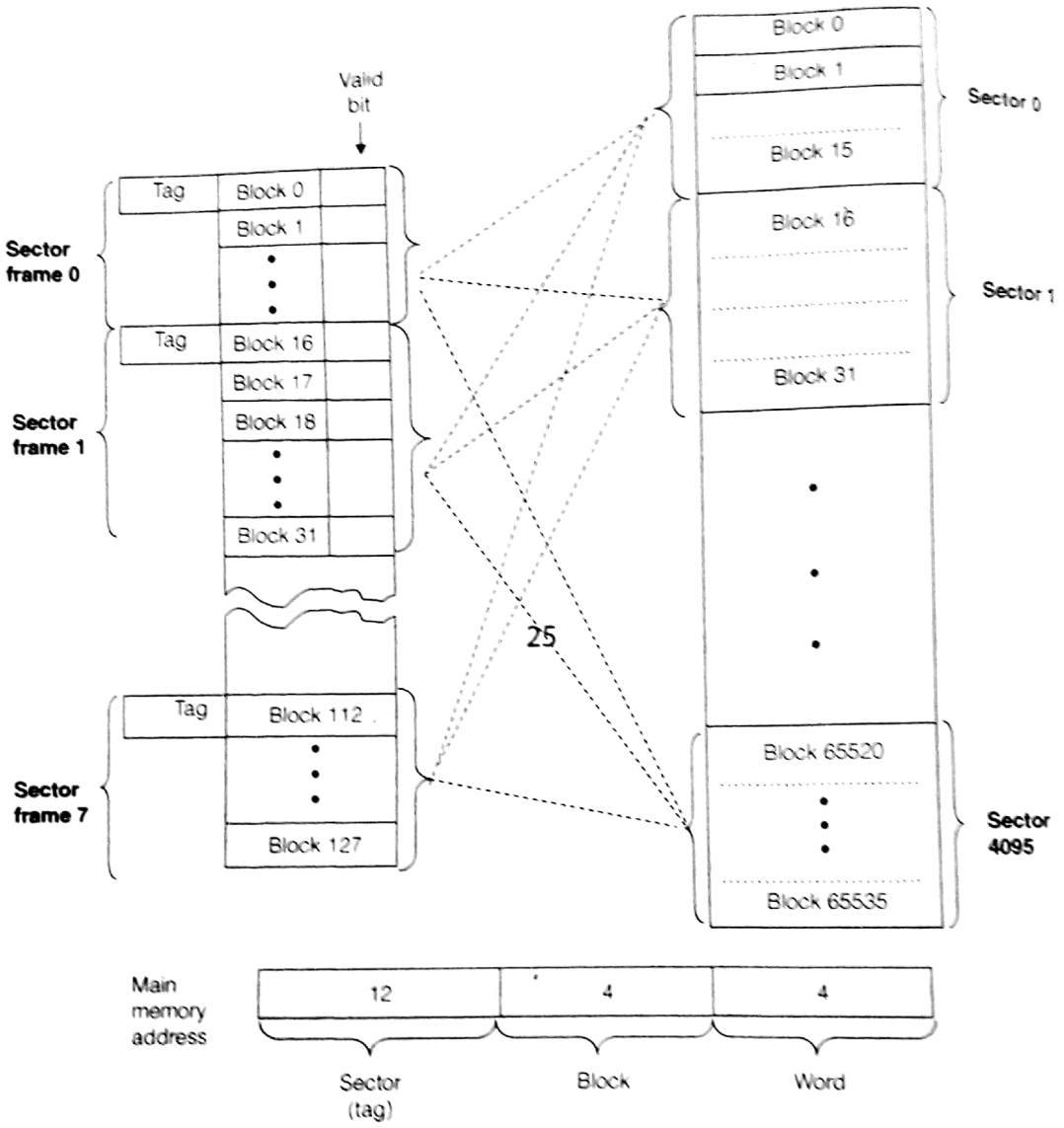


Fig. 7.39 Sector-Mapped Cache Organization

is allocated. Since corresponding blocks in the sector are automatically allocated to the corresponding block frames in the sector frame, there is no need for separate tag fields in every block frame of that sector frame. The above procedure automatically allocates a block frame for the desired block.

Having allocated a block frame, the desired block is brought from the main memory and loaded into this block frame. However, we do not bring all the other blocks of this sector because they are not demanded by the program. Prefetching these blocks might result in a wastage of the bandwidth. Blocks are selectively brought into the cache, only if there is a miss on them. In order to distinguish which block frames are loaded with the corresponding

block (that is, having valid data
that is set on loading the block
Searching for a data in se
block number where the addr
resides is found out (Figure 7
sector is not assigned, then it
whether the desired block fra
block frame is not valid but
the data is forwarded to the

GETTING SPECIFIC: THE I
separate instruction and data
in size. Each cache line is 3
sets, for a 7-bit set field. Ad
20 bits:

Tag
20

Cache Hardware Requi
cache form the two extre
between the two. If there
only one block per set, th
Let us compare the
shows a comparison of
the remainder of the cac
the Pentium cache and
tag, G as the number of
the number of bits in th
of lines in the cache.
The Pentium exam
tive results in a savings
in the comparison hard

Cache Replacement
schemes can be used

Table 7.6 Compari

Cache Type
Associative
Direct-mapped
Block-set-associativ
Intel Pentium—bloc
Hypothetical associ

block (that is, having valid data) and which are not, we use a valid bit with every block frame that is set on loading the block frame.

Searching for a data in sector mapping goes as follows. Given a physical address, the block number where the address lies is found out and then the sector in which the block resides is found out (Figure 7.39). This sector is searched associatively in the cache. If the sector is not assigned, then it is a cache miss. If the sector frame is assigned, we then see whether the desired block frame is valid. If so the data is retrieved from the cache. If the block frame is not valid but only reserved, then the block is loaded into the cache and then the data is forwarded to the processor.

GETTING SPECIFIC: THE INTEL PENTIUM CACHE STRUCTURE The Pentium chip has separate instruction and data caches. Each is two-way, block-set-associative, 8 K = 2^{13} bytes in size. Each cache line is 32 bytes, giving a 5-bit byte field. There must be $2^{13}/(2 \times 2^5) = 2^7$ sets, for a 7-bit set field. Addresses are 32 bits. This means the tag field must be 32 – 12 or 20 bits:

Tag	Set	Byte
20	7	5

■

Cache Hardware Requirements. The fully associative cache and the direct-mapped cache form the two extremes of a continuum, with the block-set-associative cache being between the two. If there is only a single group, the cache is fully associative. If there is only one block per set, the cache is direct-mapped.

Let us compare the amount of cache hardware required for the three cases. Table 7.6 shows a comparison of tag memory sizes and the amount of comparison hardware, since the remainder of the cache hardware is roughly the same for all three cases. It also shows the Pentium cache and a hypothetical Pentium cache. Define T as the number of bits in a tag, G as the number of bits in the group field, K as the number of lines in a set, B = T + G as the number of bits in the entire block field of a main memory address, and C as the number of lines in the cache.

The Pentium example shows that moving from fully associative to block-set-associative results in a savings of over half in hardware requirements, with the main savings being in the comparison hardware.

Cache Replacement Policies. The cache placement policies or the cache organization schemes can be used to choose which is (are) the possible cache block frame(s) that can

Table 7.6 Comparison of Cache Structures and Cache Hardware Requirements

Cache Type	No. of Tag Bits	Tag Memory Size, Latches	Compare Hardware, XORs
Associative	T = B	T × C	T × C
Direct-mapped	T = B – G	(B – G) × 2G	B – G
Block-set-associative	T = B – G	(B – G) × 2G × K	(B – G) × K
Intel Pentium—block-set-associative	20 = 27 – 7	20 × 27 × 2 = 5120	20 × 2 = 40
Hypothetical associative Pentium cache	27	27 × 256 = 6,912	27 × 256 = 6,912

be used for placing the block that is fetched from the main memory. Placement is possible only if atleast one of the possible candidate block frames is vacant. If all the candidate block frames are full, we use a replacement policy to replace one of the existing blocks to make room for the incoming block. The question then is, which block should be chosen for replacement?

Obviously, a good replacement policy should choose the block that is unlikely to be referred in the future. Since the future references are unknown, different cache replacement policies make use of the program characteristics to guess the future references and choose the block that is least likely to be referred in future. The cache replacement policies depend upon the cache placement policy that is in use in the system.

In the case of direct-mapped cache, the replacement policy is trivial. Each block in the main memory is mapped onto a unique block frame in the cache. Therefore, whatever block that is existing in that block frame has to be replaced and we are left with no choice.

For the case of fully associative cache organization, any block frame in the cache can be used for placement and therefore a good replacement policy is crucial for the cache performance. In the case of set associative memory, the replacement policy has to be applied only among the blocks existing in the set onto which the candidate block is mapped. The following are some of the important cache replacement policies that are used in practice.

First in First Out (FIFO). FIFO algorithm chooses for replacement the block that has been brought into the cache first among the blocks that are resident in the cache. In other words, the block having the largest resident time is replaced. However, the disadvantage of the FIFO algorithm is that oldest block may still be present in the working set of the program, but FIFO algorithm replaces it.

Least Frequently Used (LFU). In comparison to the resident time, the access frequency of a block is a better indicator of the usefulness of a block. This algorithm keeps track of the frequency of access of the different blocks that are resident in the cache and chooses for replacement the block with the least frequency of access. The disadvantage of LFU algorithm is that it may replace a newly brought block because its frequency of access is small while that block may be needed for future.

Least Recently Used (LRU). The LRU algorithm relies upon the temporal locality of reference property of the program. It assumes that at any given time, blocks that are referred in the recent past are likely to be referred in the near future. Therefore, it chooses the block that has been used least recently for replacement. LRU algorithm can be formally defined as follows. Given a reference string

$$RS(n) = \{r(1), r(2), \ldots, r(n)\}$$

We define the backward reference distance $b(n)$ of the ith cache block after the nth reference as follows. Each address present in the reference string falls in a particular cache block. Therefore, we can arrive at a string of blocks referred by the program and denote it as

$$BS(n) = \{b(1), b(2), \ldots, b(n)\}.$$

$$b(n) = \begin{cases} k & \text{if } b(n-k) \text{ is the last occurrence of } i \text{ in } BS(n) \\ \infty & \text{if } i \text{ does not appear in } BS(n) \end{cases}$$

If a cache miss occurs after the nth reference, the LRU algorithm chooses from the resident blocks that block with the largest backward reference distance. The LRU is the most commonly used replacement policy. The LRU algorithm is implemented in hardware by maintaining a counter for each block. The value of the counter represents how recently the corresponding block is accessed, and the counters are manipulated as follows.

When a cache hit occurs

1. The counter of the referred block is set to 0.
2. Blocks having counter value originally less than the referred block are incremented.
3. Blocks having counter value originally more than the referred block are unaltered.

When a cache miss occurs and the cache (or set) is not full

1. The referred block is loaded and its counter is set to 0.
2. All other counters are incremented.

When a cache miss occurs and the cache (or set) is full

1. Block with the largest value of the counter is replaced by the new block and its counter value is set to 0.
2. All other counters are incremented.

For fully associative cache memories or set-associative cache memories with large number of blocks per set, it is costly to implement the above LRU algorithm. Therefore, approximate versions of LRU algorithm are usually implemented. In the case of a four-way set-associative mapping, an approximate LRU scheme might use one bit to indicate which pair of blocks is LRU, and use another bit to record which block in each pair is LRU.

Random Replacement. In random replacement, a random or pseudorandom technique is used to select a block for replacement. Surprisingly, measurements show that even the random replacement policy is quite effective given current cache sizes.

In all replacement policies, the replaced block may have to be stored in the main memory before being overwritten by the newly brought in block. An improvement can be achieved if the block to be replaced is first stored in a fast register and then updated in the main memory later. As a result the block fetched from the main memory can be quickly written in the cache and it need not wait for the main memory update of the replaced block. We will consider more on this technique in our discussions on cache write buffer.

Cache Read and Write Policies. When the processor makes a memory reference, it may be for a memory read or for a memory write. In each of these two cases, there can be a hit or a miss in the cache. Accordingly, we have the following four different cases.

1. cache hit on a memory read
2. cache hit on a memory write
3. cache miss on a memory read
4. cache miss on a memory write

Note that write policies are needed only for the data cache and not for the instruction caches, as instruction caches are not supposed to be modified. The cache read and write policies are arrived at in order to implement a closely related policy called as the main memory update policy which is explained in the following.

Whenever the content of the cache is changed by the processor through a write operation, the contents of the main memory have to be updated subsequently to avoid the data inconsistency or the cache coherence problem. The main memory update policy is concerned with when and how to update the main memory such that the cache coherence problem is solved and at the same time the memory bandwidth is efficiently used. We now discuss the cache read and write policies for the above four cases.

Cache Hit on a Memory Read. This is the simplest of the four cases. The referred data is accessed from the cache and delivered to the processor. There is no need to update main memory on memory read operation.

Cache Hit on a Memory Write. Whenever a write operation performs, there is a potential cache coherence problem which needs to be solved using a main memory update policy. There are two main memory update policies in use, they are the write-through and the write-back policies.

1. Write-Through Policy. The write-through policy updates both the cache and main memory upon each write. The copy of data in the main memory is never different from that of the copy in the cache. Thus the cache coherence problem is solved. This will be a winning strategy, if there are a few writes to a given block, but it will be disadvantageous if there are multiple writes, as might be the case in updating a loop counter for example.
2. Write-Back Policy. The write-back policy writes only to the cache and postpones updating main memory until the block is replaced in the cache. A policy called as the simple write-back always updates the main memory whenever a block is replaced in the cache. The simple write-back policy may perform many redundant updates. This would be the case, for example, when a block is brought in to the cache and is read many times but not written even once.

A better scheme called flagged write-back attaches a bit called dirty bit with all cache blocks. When a new block is brought and loaded in a block frame, its dirty bit is reset. The dirty bit is set the first time, a value is written to the block. When a block in the cache is to be replaced, its dirty bit is examined, and if it has been set, the block is written back to main memory, otherwise the block is simply overwritten.

An advantage of the write-through policy is that during replacement the replaced block can simply be over written by the fetched block, and there is no need to associate dirty bits to the cache blocks.

Cache Miss on a Memory Read. If there is a cache miss on a memory read operation, the requested block is fetched from the main memory and loaded into the cache. Then, the desired data is read from the cache and given to the processor. While loading the fetched block in the cache, there can be two cases.

1. One or more candidate block frames (as dictated by the placement policy) for the fetched block is available, in which case the fetched block is loaded in one of those candidate block frames.
2. The cache is full in the sense that all candidate block frames for the fetched block are occupied, in which case a replacement policy has to be used to make room

for the fetched block. In case the write-through policy is used or if the write-back policy is used, and the dirty bit of the replaced block is reset, the fetched block can be simply overwritten on the replaced block. Otherwise, the fetched block has to wait for the replaced block to be updated in the main memory.

Cache Miss on a Memory Write. First let us consider the case wherein the write-back policy is used during cache hits. If a cache miss on a memory write occurs, then the missed block is fetched from the main memory and loaded in to the cache so that future cache misses are avoided. The copy of the block loaded in the cache is written.

Next, let us consider the write-through policy. In the case of write-through policy whenever cache is written, the main memory is also updated. If it is assumed that a write operation is more likely to be followed by write operations, then there is no need to load the block into the cache as a cache hit or a miss is not going to affect the time taken for a write operation in write-through policy. In case if it is assumed that a write operation can be followed by either a read or a write operation, then it is worth, while bringing the block into the cache as a subsequent read operation would be faster.

Therefore, there are two possible variations of the write-through policies that can be adopted. They are the write-through with write-allocate policy, and the write-through with no-write-allocate policy. In the case of write-through with write-allocate policy, the missed block is always fetched from the main memory (both for cache miss on a write, and for cache miss on a read) and loaded into the cache. This is followed by a write operation on the copy of the blocks in the main memory and in the cache. In the case of write-through with no-write-allocate policy, during a cache miss on a write operation, only the copy of the block in the main memory is written and the block is not loaded into the cache, whereas the block is allocated to the cache on a read miss.

Supposing in a cache miss, if the block is fetched from the main memory and loaded into the cache before accessing the data from the cache block, then the processor has to wait till the block is loaded into the cache. To speed up this process, a technique called load through or read through can be adopted. Here the fetched block is first loaded into a high speed register. The desired word is accessed from this register and forwarded to the processor while, the loading of the block onto the cache takes place simultaneously.

Cache Fetch Policies. The cache fetch policies are concerned about when to fetch blocks from the main memory, whether to prefetch blocks and if so when, how many, and which blocks to prefetch. The simplest scheme is called demand fetch that fetches only that block for which a cache miss has occurred and does no prefetching.

Prefetching schemes, in addition to fetching blocks on demand, prefetches blocks from main memory to reduce future cache misses. Aggressive prefetching of blocks may have detoriating effect. It is not only that the prefetched data or instruction may not be needed in the future, but prefetched blocks may replace useful blocks and also lead to bus contention due to increased traffic.

Therefore, a practical prefetch scheme called as the "one blocklook ahead always prefetch scheme" makes use of the temporal locality of reference and works as follows. Whenever block i is referred, the scheme prefetches block $i + 1$ if the block $i + 1$ is not already present in the cache. The scheme is called "always prefetch" because it prefetches $i + 1$ whenever block i is referred, irrespective of whether there is a hit or a miss on block i. An alternate scheme

called as "one block lookahead prefetch on a miss" scheme prefetches $i + 1$, only under the condition that there is miss on block i. The scheme is based on the following assumption. If there is a miss on i, then block i needs to be anyhow fetched. Therefore prefetching $i + 1$ along with i will only increase the memory bandwidth requirement marginally.

Optimal Size of Cache Block. With an increase in the size of the block, the hit ratio at first improves due to the spatial locality of reference. However, with continued increase in block size the effect of spatial locality in improving hit ratio saturates, however the number of block frames becomes less for a given size of the cache resulting in frequent replacement and drop in hit ratio.

Levels of Cache. The current trend, is to use two or three levels of caches between the processor and the main memory. Level 1 cache also called as the primary cache, is usually located within the processor's core and it operates at the same clock speed of the processor. The level 2 cache, also called as the secondary cache, is a larger cache placed after the primary cache. Usually level 2 caches are off-chip cache, but some modern processors have integrated them inside.

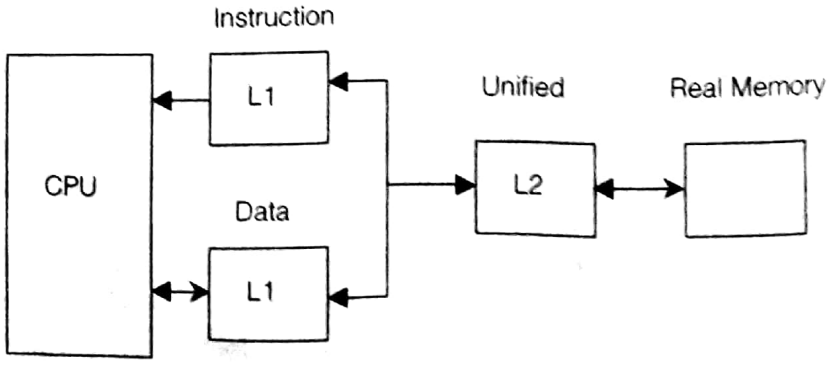
unified cache

split cache

Unified and Split Cache. If there is a single cache at a given level that holds both data and instructions, then it is called as the *unified cache*. If the cache at a given level is split into two caches, one for storing the instructions and another for storing the data, then this approach is called as the *split cache*. For a given overall size, a unified cache is more flexible and offers higher hit ratio than the split caches. The reason being the working set of a program may have a larger fraction of data than instruction or vice versa. A unified cache can flexibly accommodate either data or instruction.

The main advantage of the split cache is that it avoids the cache contention. With a split cache, the instruction fetch stage and the operand fetch/result store stage of the pipeline can simultaneously access respectively, the instruction and the data cache. Recent trend is to use split cache in level 1 and a large unified cache at level 2 as shown in Figure 7.40.

Burst Mode Cache. In the case of asynchronous SRAMs, the processor has to provide an address for each cache access and then has to wait for the memory access to be completed. On the other hand, the burst mode cache reads or writes a contiguous sequence of addresses in a quick burst. Burst mode in a cache eliminates the need to send a separate address for each memory read or write operation. The synchronous burst SRAM uses an



internal clock to increment the address after each memory operation. In order to make the memory operations synchronous, the cache runs at the same speed as the processor. Hence synchronous burst SRAMs are expensive than conventional asynchronous cache designs.

Pipelined burst SRAM eliminates the need for a synchronous internal clock. This cache includes an internal pipelined register that is loaded by the cache through burst operation. The registers are later read by the processor.

Cache Write Buffer. A block from the cache may have to be written into the main memory frequently or infrequently depending upon the cache write policy. In the case of write back protocol on a cache miss, the block that is chosen for replacement should be written back if it is dirty. In the case of write-through policy every block that is written in the cache should also be updated in the main memory. If the processor has to wait for main memory update, then it slows the computation.

In order to remove this bottleneck, a temporary storage called the write buffer is introduced between the cache and the main memory. Whenever a main memory update has to be done, the processor simply stores the update information (main memory address and the data) in the write buffer and proceeds to execute other instructions. The write buffer is a pipeline that can store a number of write requests from the processor. Whenever the main memory is not read by the processor, the write buffer uses those unused memory cycles to complete the write requests that it has stored. This technique relieves the processor from waiting for main memory updates to be complete. However, it is assumed that the main memory address that is to be updated eventually by the write buffer may not be read till the updating is over. If the processor generates a memory read operation on an address that is not yet updated (still in the write buffer), then care should be taken to provide the updated data.

The problem shows up in the following case. Suppose the write-back policy is used, a block is chosen for replacement and placed in the write buffer. A cache read miss occurs on the block that is still in the write buffer. To maintain data consistency in the presence of write buffer on all memory read a check is made whether the data is present in the write buffer. This is done by comparing (associatively) the addresses of data to be read from the memory with the addresses of the data in the write buffer. In case there is a match in the address, the data is read form the write buffer.

Virtual-and Real-Address Caches. We saw that when the processor generates the virtual address, it is first translated into the physical or real address. The real address is used to search in the cache based on the cache organization scheme. The disadvantage with this scheme is that there is a latency involved in translating from virtual address to real address and the cache access can begin only after the latency period.

The other alternative is to initiate the cache search during the address translation. Figure 7.41 shows the working of virtual address cache. The displacement field of the virtual address (word offset within page) is not used for translating from the virtual page number to the physical page number. Therefore, the cache set index and displacement (word offset within the block) fields are extracted from the virtual-address displacement and they are used by the cache controller to access the cache block and the tags. Simultaneously the virtual page name is being translated to the page frame address. The actual tag is extracted from the page frame address and then compared with the tags accessed from the cache. If a hit occurs, the corresponding cache block and the displacement are used to read the word and is given to the processor. If the tags do not compare, then it indicates

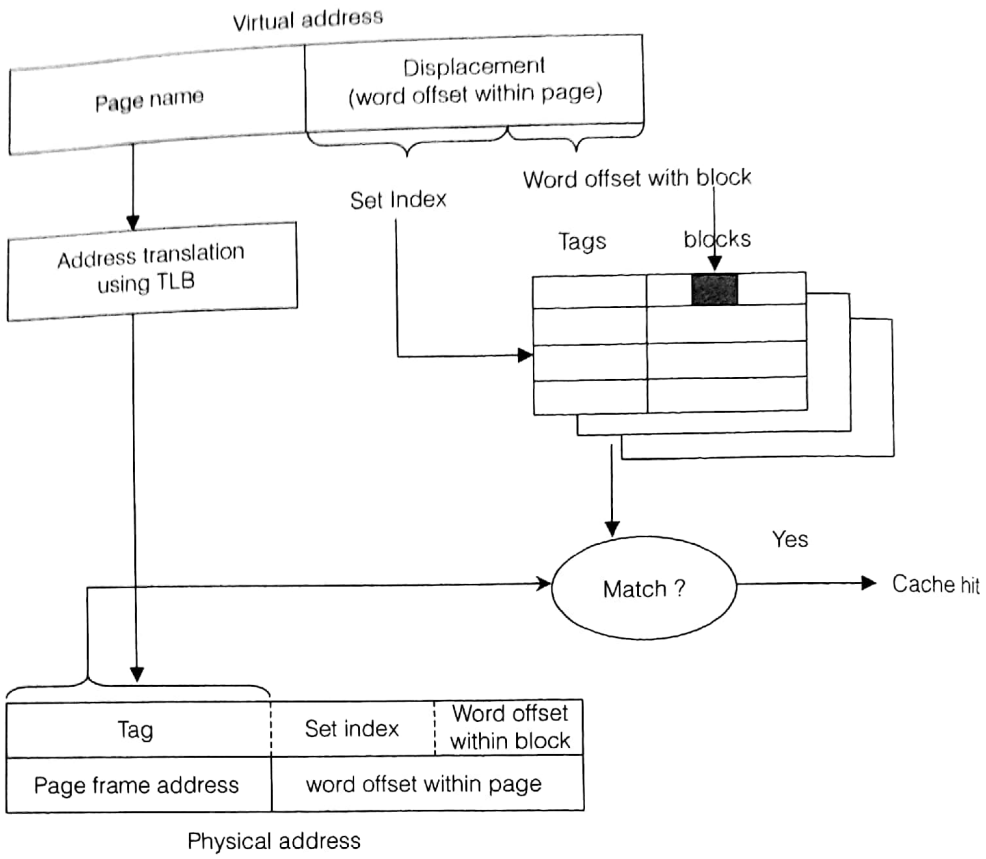


Fig. 7.41 Virtual Address Cache (For the case of cache and page size being same)

a cache miss condition. If a set associative cache is used, then all tags corresponding to the set index have to be compared with the tag field of the physical address. This scheme assumes that the set index and the word offset within block fields fall within the virtual address displacement (word offset within page) field. Only if this condition is met, address translation using TLB can go on in parallel with the cache access. For the direct-mapped cache, the above condition implies that the size of the cache cannot be larger than the size of the page. However, if a set-associative cache is used, the size of the cache can be larger and it is given by the formula.

Maximum cache size = Size of page X Degree of set associativity

If a cache larger than the size given by the above condition is used, then a problem called synonym or alias occurs. More than one virtual address can refer to a cache address since the number of virtual address bits used is less than the actual number needed to address the cache.

UltraSPARC-1 Address Translation Mechanism. UltraSPARC-1 has 64-bit logical address having a 31-bit virtual page number field and a 13-bit field for displacement within page. The UltraSPARC has a virtually indexed level 2 instruction cache of size 16 Kbytes.

It is pseudo two-way set-associative cache (the most significant bit of the cache index is predicted). Therefore, only the page offset field (13-bits) is needed to index the cache and to identify the word within the block. The instruction cache is effectively indexed by the physical address, and there are no cache synonym problems. However the direct-mapped 16-Kbyte of level 1 data cache is twice the page size (2^{13}). Therefore the synonym problem can occur. This problem is solved at the OS level.

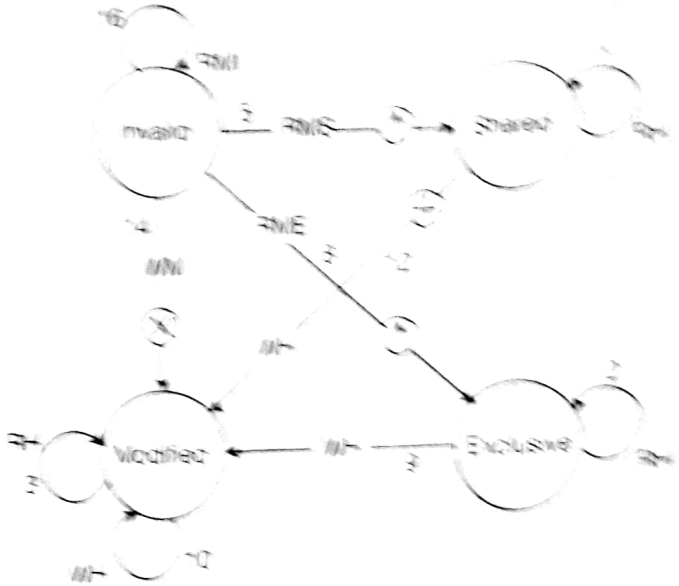
MESI Cache Coherence Protocol. In this case of a shared memory multiprocessor system, part of the memory address space is being shared by multiple processors. The address space is accessed by all the processors using a shared bus. However, to improve the performance, each processor may have one or more levels of local cache memory. Therefore multiple copies of the same data are present in different cache memories. If each processor is allowed to independently modify its local copy, then the data becomes inconsistent. Different programs running on the processors and sharing the data would produce results that are unpredictable and not reproducible. The MESI protocol is a cache coherence protocol to maintain the data consistency in a hierarchical memory of a multiprocessor system.

MESI protocol uses the bus snooping technique. In the bus snooping technique different cache controllers can exchange information over a bus. In particular (1) a cache controller can broadcast messages to other cache controllers over the bus and (2) all cache controllers monitor the bus to update the state of its cache blocks.

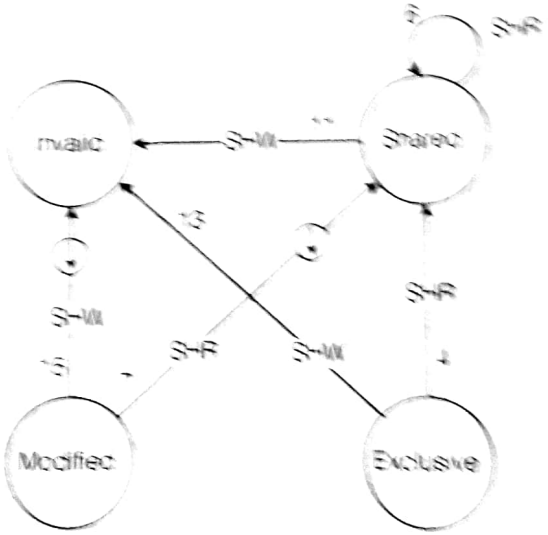
In the MESI protocol, every cache block is associated with two bits that represent the state of the block to be one among the following four states: modified, exclusive, shared and invalid. Following is the semantics associated with the states.

- **Invalid.** Indicates that the data in the cache block is invalid. It may be because the copy of the data in that local cache is not the most resent one, and therefore it is invalid to use it.
- **Shared.** Indicates that the copy of the cache block may be present in other caches of the system, but in any case the version of the data in the block is most recent and hence valid for usage.
- **Exclusive.** The block is cached only by this cache and it is not present in other caches. The copy of the cache block is the same as the one present in the main memory.
- **Modified.** The cache block is present only in the current cache and it has been modified. To elaborate, the cache block is exclusively present only this cache, but it has been modified and therefore the copy in the main memory is old.

Figure 7.42 shows the state transition diagram of the MESI protocol. The state diagram is split for easy understanding in such a manner that Figure 7.42a shows those transition that occur in the state of the cache blocks present in the local cache of the processor that initiates the memory read or write. Figure 7.42b shows those transition that change the state of the cache blocks in caches belonging to processors which have not initiated the memory operation but perform the change in the state as a result of snooping the bus. Certain state transitions are accompanied by bus operations which are shown in Figure 7.42. The state diagram is explained by considering various scenarios.



(a) State diagram for the state of the block in cache at initiating processor



(b) State diagram for the state of the block in the snooping cache

Event initiating state transition

RH	Read hit
RMS	Readmiss, shared
RME	Read miss, exclusive
RMI	Read miss, invalid
WH	Write hit
WM	Write miss
SHR	Snoop hit on read
SHW	Snoop hit on write or Read with intent to modify

Action performed during state transition

- Dirty block is updated in main memory
- Invalidate transaction
- Read with intent to modify
- Cache block is read from main memory

Fig. 7.42 Cache Coherence Protocol State Transition Diagram

Cache Read Operation. In this case a processor performs a memory read to its local cache which can be either a hit or a miss.

1. **Cache Read Hit.** A cache read hit is possible only when the cache block is in one of the following states: shared or modified or exclusive state. The cache read operation takes place on that block and the state remains as it is. The reason being as long as the block is in either one of the following states (shared or exclusive or modified), then the block is recent and hence it is valid to read the same (Transitions 1,2,3). In case the block is shared, all shared blocks are consistent and memory read will not modify the block content. Hence there is no need to modify the state of the shared blocks present in other memories.
2. **Cache Read Miss.** In case the block is not present in the cache, we will invoke a replacement policy, reserve a block frame and set its state as invalid. Therefore a miss in cache read is similar to the case wherein the block is in the invalid state. The invalid block must be fetched from the main memory to the shared or exclusive states to satisfy a read. Before doing so, the processor broadcasts a message called "snoop hit on read" (SHR) on the snoop bus. The other caches snoop the message and take one of the following operations based on the state of its cache block.

   - **Exclusive State.** A cache block in one of local cache is in exclusive state (note that multiple cache blocks cannot be in exclusive state). On hearing the SHR, the processor (having that block in exclusive state) responds back indicating that it shares the block and the processor changes the state of its block from exclusive to the shared state (Transition 4). This will be followed by a block read operation by the initiating processor and the state of the block is changed from invalid to shared state (Transition 5).
   - **Shared State.** One or more caches have copy of the block in the shared state. All those processors, on hearing the SHR, send signal back to the initiating processor that they share the block (Transition 6). Since the main memory copy is valid, the initiating processor reads the block and is changed from invalid to shared state (Transition 5).
   - **Modified State.** A cache block in one of local cache is in modified state (note that multiple cache blocks cannot be in modified state). On hearing the SHR, the responding processor sends a signal to the initiating processor to retry the read operation later. The initiating processor remains in the invalid state and retries the block read operation later. Meanwhile the responding processor updates the main memory copy of the block and changes its block state from modified to shared (Transition 7). On the next retrial (Transition 5) will be invoked. The above operation ensures that the most recent copy of the block is read by the initiating processor.
   - **None of the Above Cases.** No other processor has a copy of this block and therefore no signal returns back for the SHR (Transition 16). The initiating processor reads the block and updates the state to exclusive state (Transition 8).

Cache Write Operation. A processor attempts to write a block. It broadcasts a snoop hit on write (SHW) signal over the snoop bus and gets the reply. There can be a hit or miss.

1. **Cache Write Hit.** A cache write hit can happen when the block is available in the cache and it is in a state other than the invalid state. Following are the three possible states the cache block can be.
   - **Exclusive.** Only the initiating processor has an exclusive copy of the block. The initiating processor simply writes the block and changes the state from exclusive to modified state (Transition 9).
   - **Modified.** Only the initiating processor has an exclusive copy of the block but it is modified. The initiating processor simply writes the block and maintains the state as the modified state (Transition 10).
   - **Shared State.** One or more shared copies of the block exist. The initiating processor sends a signal called invalidate transaction. On getting this signal, processors having the shared block come to know the intention of a processor in modifying the data. In anticipation of the modification, those processors invalidate their blocks (Transition 11). There is no need for a response. The initiating processor modifies the block and changes the state to modified state (Transition 12).

2. **Write Miss.** If the block is not available in the local cache, it has to be read from the main memory and then modify it (Transition 14). Therefore it sends the signal read-with-intend-to-modify (RWITM). Other processor responds based on their state as given below.
   - **Shared or Exclusive State.** No response is given back. However, the other caches invalidate their blocks as they are going to be modified by the initiating processor (Transition 11,13).
   - **Modified State.** The responding processor writes back its block into the main memory and then invalidates it (Transition 15). The initiating processor then reads the block, performs a write operation and then marks the block as modified (Transition 14).

The reader may wonder why in the last case the block has to be written into the main memory by the responding processor as it is anyhow going to be overwritten by the initiating processor. Note that it may so happen that a processor broadcasts the RWITM signal, but it is not followed by a cache write operation on that block (this may occur if the process exits after sending RWITM). In the above-mentioned situation, if the responding processor invalidates its block, then although the main memory would have an older version of the block, the most recent copy of the block would not be present in the system. To avoid this problem, the responding processor updates the main memory before invalidating its block. Finally note that the case of a process exiting after sending RWITM would not matter if the responding processor changes the state of its block from shared state or exclusive state to invalid state. In this case the main memory would have the most recent copy.

This completes our discussion on the MESI protocol state machine. The Pentium processor implements the MESI cache coherence protocol.

Performance and Design: Cache. The direct-mapped and block-set-associative caches may suffer from minor time penalties due to the need to decode the group field. Decoding the group field will result in the same penalties that were discussed in the section on matrix and tree decoders; that is, high fan-in requirements may result in the need to add one or

more additional gate layers,
a direct effect on cycle time
direct-mapped or set-associ
required for the associative
with a corresponding increa

A quantitative analysis
ing research, but we can get
sis. Let h be the hit ratio, the
of memory references, and
access time upon a cache hi
the average time T_{AV} to acce

This equation holds f
memory to the CPU direct
loaded before initiating a
is replaced by the cache li
not only on the particular
work load. As an example
mapped cache to a set-ass

Example 7.1 Spe
direct-mapped to two-w
of benchmarks to impro
speedup, assuming no ch

Answer: $T_{without}$
T_{with}

so $S = 24.5/22.0 = 1.11$

GETTING SPECIFIC: T
of the PowerPC G4 C
with a block size of 16
convenience in the up
operations of 4 words
the cache for other us
to a write-back policy
cache can also be dis
again on a per-line ba

The ARM710T p
each block having 16
replacement policy.

more additional gate layers, with a corresponding increase in delay. This increase will have a direct effect on cycle time. On the other hand, given an equivalent amount of hardware, direct-mapped or set-associative caches will have more capacity because the gates not required for the associative memory can be used to increase the number of cache lines, with a corresponding increase in hit ratio.

A quantitative analysis of cache performance is quite complex and is the subject of ongoing research, but we can get an idea of the factors involved with a simple mathematical analysis. Let h be the hit ratio, the ratio of memory references that hit in the cache to the total number of memory references, and $(1 - h)$ be the corresponding miss ratio. We define T_C as the cache access time upon a cache hit, and T_M as the time to access the value upon a cache miss. Then the average time T_{AV} to access memory during a particular machine execution interval is

cache access time

$$T_{AV} = h \times T_C + (1 - h) \times T_M \qquad \text{[Eq. 7.2]}$$

This equation holds for a read-through cache, where the value is forwarded from main memory to the CPU directly as it is received, without waiting for the entire cache line to be loaded before initiating another cache read to forward the value. If this is not the case, T_M is replaced by the cache line fill time. We emphasize that specific values for h will depend not only on the particular cache structure, but also on the access pattern of the particular work load. As an example, let us calculate the *speedup*, S, resulting from changing a direct-mapped cache to a set-associative one:

speed due to the cache

$$S = \frac{T_{\text{without}}}{T_{\text{with}}} \qquad \text{[Eq. 7.3]}$$

Example 7.1 Speeding up the Cache Changing a certain cache from direct-mapped to two-way set-associative caused the hit ratio measured for a certain set of benchmarks to improve from 0.91 to 0.96, with $T_C = 20$ ns, and $T_M = 70$ ns. What is the speedup, assuming no change in T_C from the set-associative multiplexer?

Answer:
$$T_{\text{without}} = 0.91 \times 20 + (1 - 0.91) \times 70 = 24.5 \text{ ns};$$
$$T_{\text{with}} = 0.96 \times 20 + (1 - 0.96) \times 70 = 22.0 \text{ ns};$$

so $S = 24.5/22.0 = 1.11$, an 11% improvement in execution time. ■

Getting Specific: The PowerPC G4 Cache Figure 7.43 shows the organization of the PowerPC G4 Cache. The on-chip 32 KB cache is 64 × 8 block-set-associative, with a block size of 16 words of 4 bytes organized as two independent 8-word sectors for convenience in the updating process. A cache block can be updated in two single-cycle operations of 4 words each. Cache requests are buffered so that a cache miss does not block the cache for other uses during the next cycle. Although the cache was designed to adhere to a write-back policy, write-through can be implemented on a per-cache-line basis. The cache can also be disabled, thus enforcing a read no-allocate or write no-allocate policy, again on a per-line basis. The cache uses an LRU replacement policy. ■

The ARM710T processor has an unified cache which is a four-way set associative with each block having 16 bytes. It uses a write-through memory update policy and a random replacement policy.