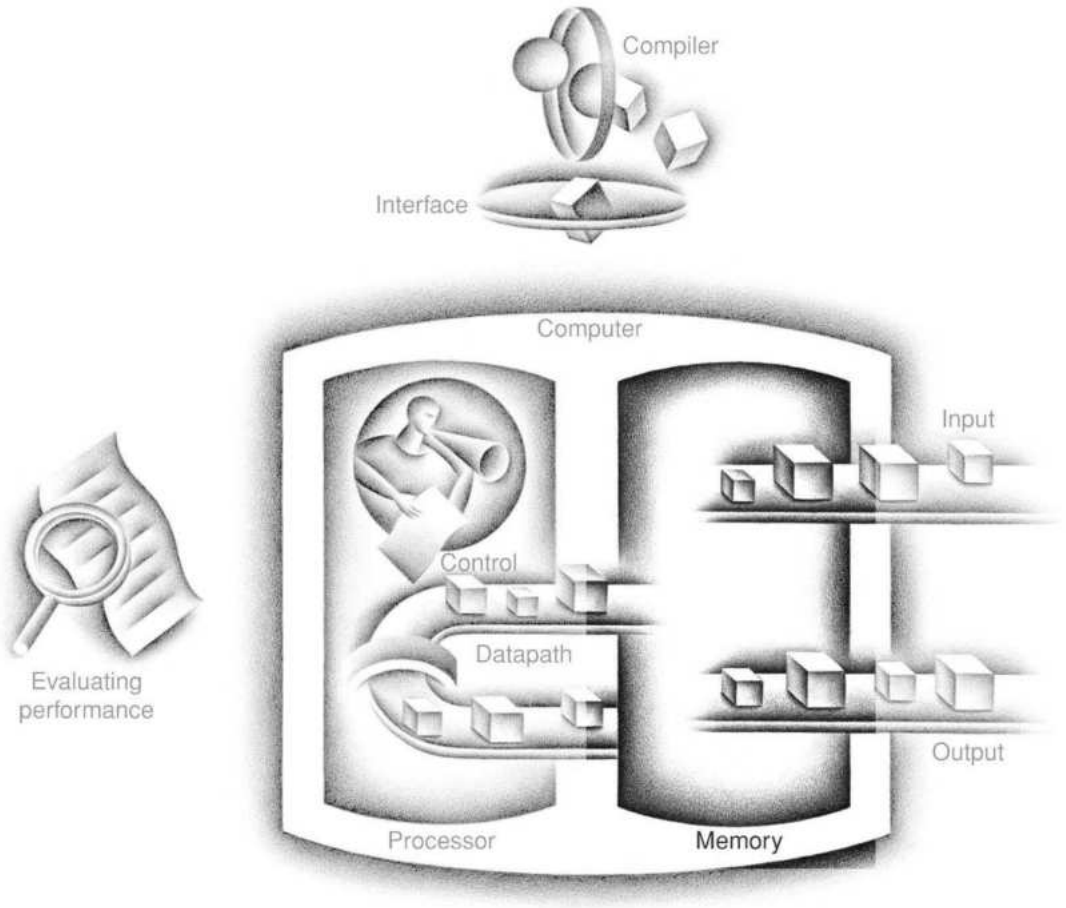


The Five Classic Components of a Computer

5.1 Introduction

From the earliest days of computing, programmers have wanted unlimited amounts of fast memory. The topics in this chapter aid programmers by creating that illusion. Before we look at creating the illusion, let's consider a simple analogy that illustrates the key principles and mechanisms that we use.

Suppose you were a student writing a term paper on important historical developments in computer hardware. You are sitting at a desk in a library with a collection of books that you have pulled from the shelves and are examining. You find that several of the important computers that you need to write about are described in the books you have, but there is nothing about the EDSAC. Therefore, you go back to the shelves and look for an additional book. You find a book on early British computers that covers the

EDSAC. Once you have a good selection of books on the desk in front of you, there is a high probability that many of the topics you need can be found in them, and you may spend most of your time just using the books on the desk without returning to the shelves. Having several books on the desk in front of you saves time compared to having only one book there and constantly having to go back to the shelves to return it and take out another.

The same principle allows us to create the illusion of a large memory that we can access as fast as a very small memory. Just as you did not need to access all the books in the library at once with equal probability, a program does not access all of its code or data at once with equal probability. Otherwise, it would be impossible to make most memory accesses fast and still have large memory in computers, just as it would be impossible for you to fit all the library books on your desk and still find what you wanted quickly.

This *principle of locality* underlies both the way in which you did your work in the library and the way that programs operate. The principle of locality states that programs access a relatively small portion of their address space at any instant of time, just as you accessed a very small portion of the library's collection. There are two different types of locality:

- **Temporal locality** (locality in time): if an item is referenced, it will tend to be referenced again soon. If you recently brought a book to your desk to look at, you will probably need to look at it again soon.
- **Spatial locality** (locality in space): if an item is referenced, items whose addresses are close by will tend to be referenced soon. For example, when you brought out the book on early English computers to learn about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you likewise brought back that book and, later on, found something useful in that book. Libraries put books on the same topic together on the same shelves to increase spatial locality. We'll see how memory hierarchies use spatial locality a little later in this chapter.

temporal locality

The locality principle stating that if a data location is referenced

then it will tend to be referenced again soon.

spatial locality

The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

Just as accesses to books on the desk naturally exhibit locality, locality in programs arises from simple and natural program structures. For example, most programs contain loops, so instructions and data are likely to be accessed repeatedly, showing large temporal locality. Since instructions are normally accessed sequentially, programs also show high spatial locality. Accesses to data also exhibit a natural spatial locality. For example, sequential accesses to elements of an array or a record will naturally have high degrees of spatial locality.

We take advantage of the principle of locality by implementing the memory of a computer as a **memory hierarchy**. A memory hierarchy consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus are smaller.

memory hierarchy

A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase.

Figure 5.1 shows the faster memory is close to the processor and the slower, less expensive memory is below it. The goal is to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.

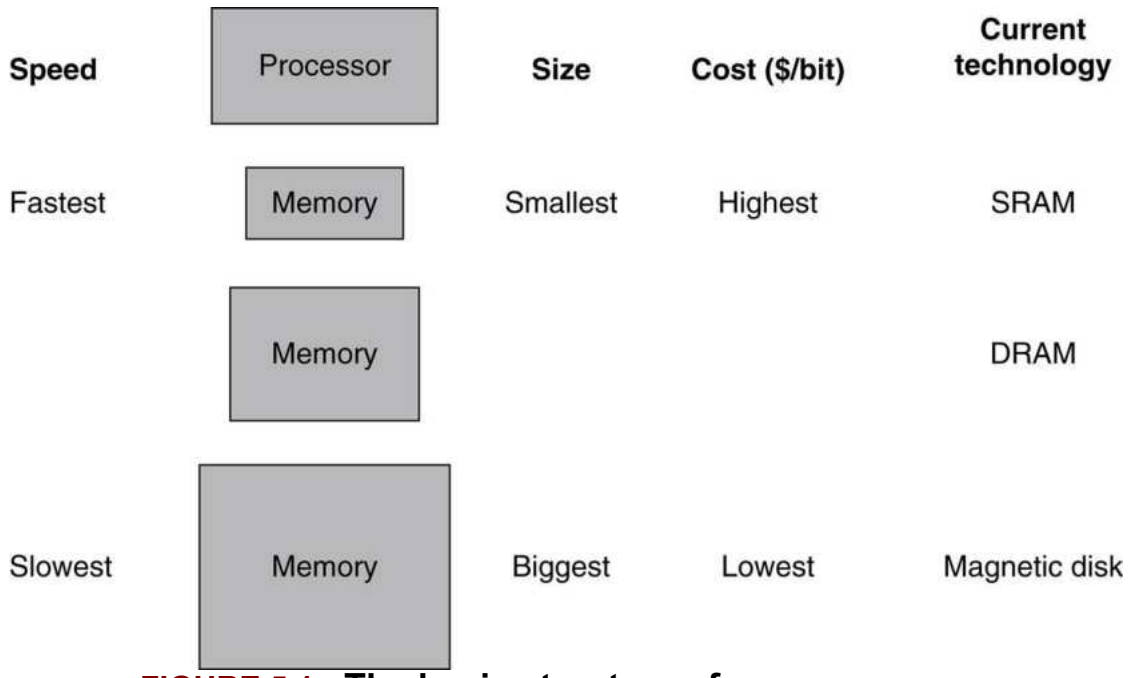


FIGURE 5.1 The basic structure of a memory hierarchy.
By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

The data are similarly hierarchical: a level closer to the processor is generally a subset of any level further away, and all the data are stored at the lowest level. By analogy, the books on your desk form a subset of the library you are working in, which is in turn a subset of all the libraries on campus. Furthermore, as we move away from the processor, the levels take progressively longer to access, just as we might encounter in a hierarchy of campus libraries.

A memory hierarchy can consist of multiple levels, but data are copied between only two adjacent levels at a time, so we can focus our attention on just two levels. The upper level—the one closer to the processor—is smaller and faster than the lower level, since the upper level uses technology that is more expensive. Figure 5.2 shows that the minimum unit of information that can be either present or not present in the two-level hierarchy is called a **block** or a **line**; in our library analogy, a block of information is one book.

block (or line)

The minimum unit of information that can be either present or not present in a cache.

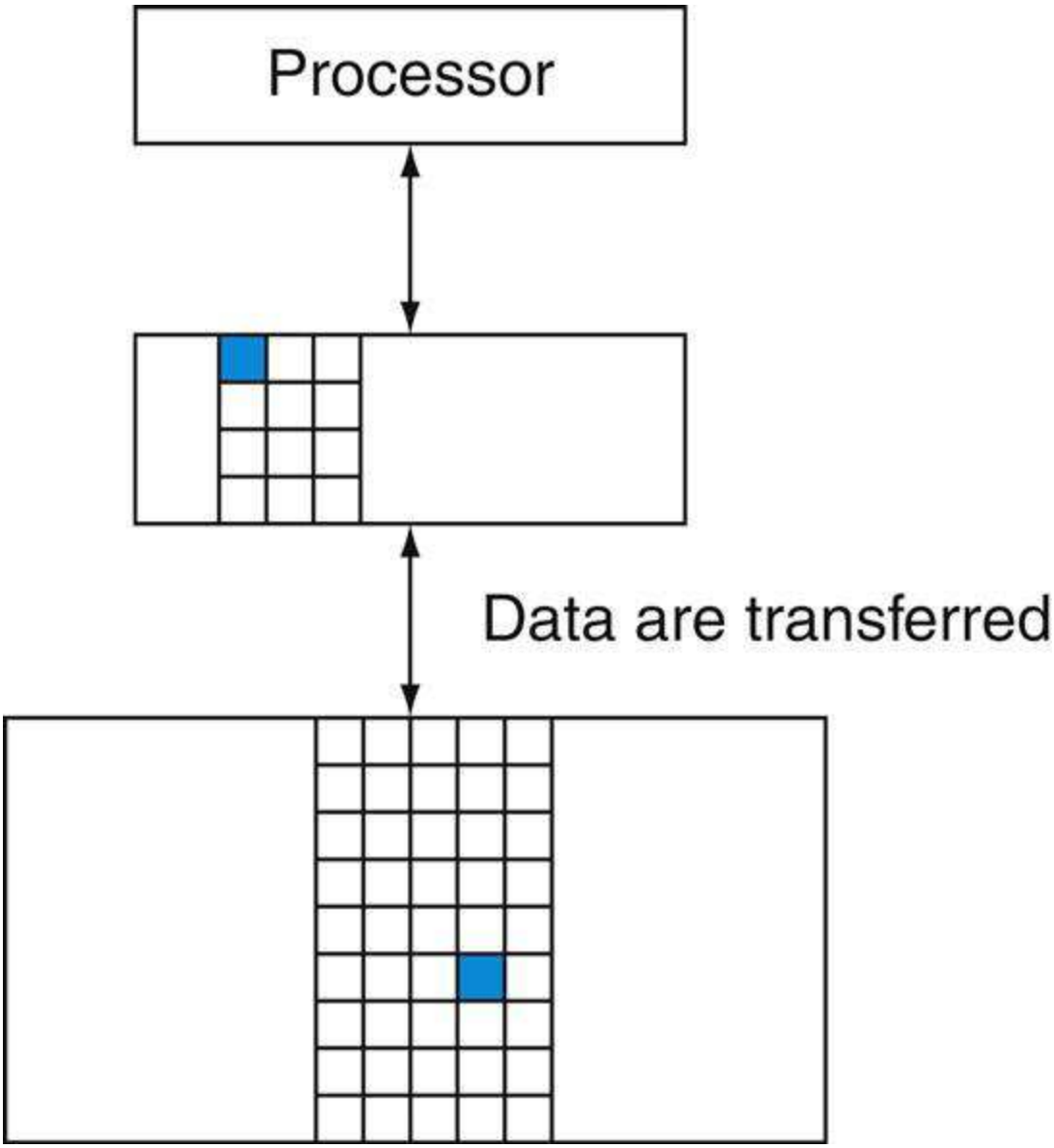


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level.
Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

If the data requested by the processor appear in some block in the upper level, this is called a *hit* (analogous to your finding the

information in one of the books on your desk). If the data are not found in the upper level, the request is called a *miss*. The lower level in the hierarchy is then accessed to retrieve the block containing the requested data. (Continuing our analogy, you go from your desk to the shelves to find the desired book.) The **hit rate**, or *hit ratio*, is the fraction of memory accesses found in the upper level; it is often used as a measure of the performance of the memory hierarchy. The **miss rate** (1−hit rate) is the fraction of memory accesses not found in the upper level.

hit rate

The fraction of memory accesses found in a level of the memory hierarchy.

miss rate

The fraction of memory accesses not found in a level of the memory hierarchy.

Since performance is the major reason for having a memory hierarchy, the time to service hits and misses is important. **Hit time** is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss (that is, the time needed to look through the books on the desk). The **miss penalty** is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor (or the time to get another book from the shelves and place it on the desk). Because the upper level is smaller and built using faster memory parts, the hit time will be much smaller than the time to access the next level in the hierarchy, which is the major component of the miss penalty. (The time to examine the books on the desk is much smaller than the time to get up and get a new book from the shelves.)

hit time

The time required to access a level of the memory hierarchy, including the time needed to determine whether the access is a hit or a miss.

miss penalty

The time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced the miss, and then pass the block to the requestor.

As we will see in this chapter, the concepts used to build memory systems affect many other aspects of a computer, including how the operating system manages memory and I/O, how compilers generate code, and even how applications use the computer. Of course, because all programs spend much of their time accessing memory, the memory system is necessarily a major factor in determining performance. The reliance on memory hierarchies to achieve performance has meant that programmers, who used to be able to think of memory as a flat, random access storage device, now need to understand that memory is a hierarchy to get good performance. We show how important this understanding is in later examples, such as Figure 5.18 on page 400, and Section 5.14, which shows how to double matrix multiply performance.

Since memory systems are critical to performance, computer designers devote a great deal of attention to these systems and develop sophisticated mechanisms for improving the performance of the memory system. In this chapter, we discuss the major conceptual ideas, although we use many simplifications and abstractions to keep the material manageable in length and complexity.

The BIG Picture

Programs exhibit both temporal locality, the tendency to reuse recently accessed data items, and spatial locality, the tendency to reference data items that are close to other recently accessed items. Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy.

Figure 5.3 shows that a memory hierarchy uses smaller and faster memory technologies close to the processor. Thus, accesses

that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy, which are larger but slower. If the hit rate is high enough, the memory hierarchy has an effective access time close to that of the highest (and fastest) level and a size equal to that of the lowest (and largest) level.

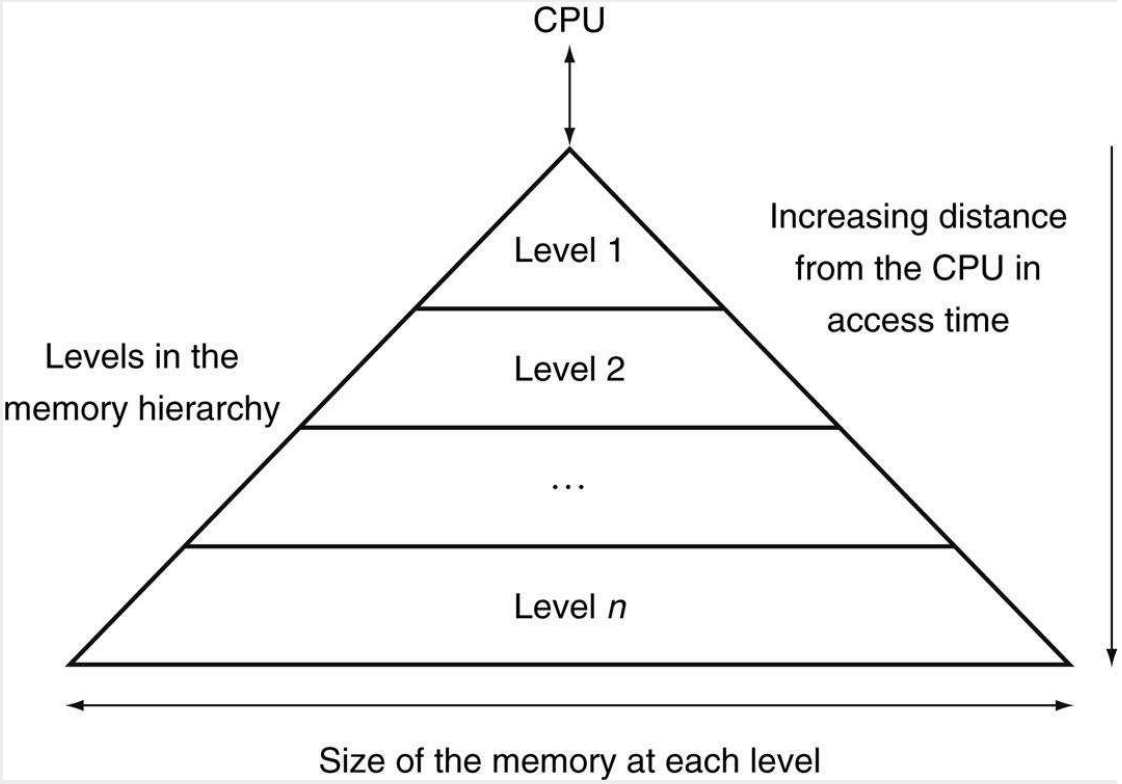


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size.
This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

In most systems, the memory is a true hierarchy, meaning that data cannot be present in level i unless they are also present in level $i + 1$.

Check Yourself

Which of the following statements are generally true?

1. Memory hierarchies take advantage of temporal locality.
2. On a read, the value returned depends on which blocks are in the cache.
3. Most of the cost of the memory hierarchy is at the highest level.
4. Most of the capacity of the memory hierarchy is at the lowest level.

5.2 Memory Technologies

There are four primary technologies used today in memory hierarchies. Main memory is implemented from DRAM (*dynamic random access memory*), while levels closer to the processor (caches) use SRAM (*static random access memory*). DRAM is less costly per bit than SRAM, although it is substantially slower. The price difference arises because DRAM uses significantly less area per bit of memory, and DRAMs thus have larger capacity for the same amount of silicon; the speed difference arises from several factors described in Section A.9 of Appendix A. The third technology is flash memory. This nonvolatile memory is the secondary memory in Personal Mobile Devices. The fourth technology, used to implement the largest and slowest level in the hierarchy in servers, is magnetic disk. The access time and price per bit vary widely among these technologies, as the table below shows, using typical values for 2012.

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

We describe each memory technology in the remainder of this section.

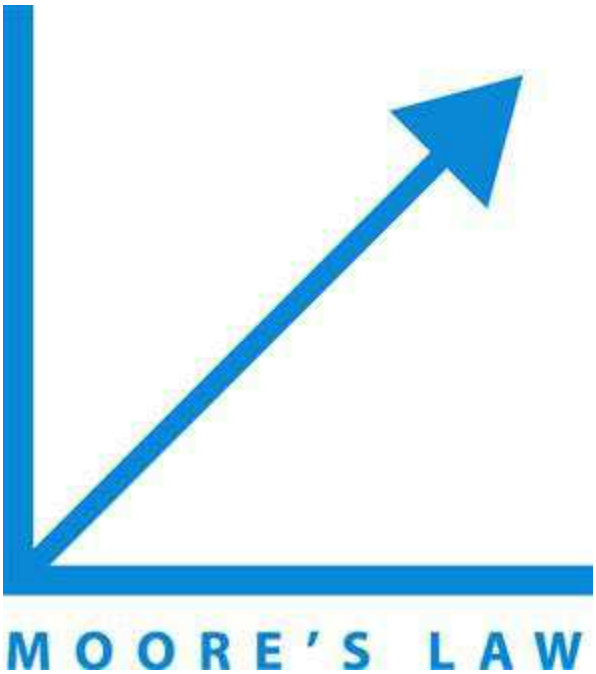
SRAM Technology

SRAMs are simply integrated circuits that are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time to any datum, though the

read and write access times may differ.

SRAMs don't need to refresh and so the access time is very close to the cycle time. SRAMs typically use six to eight transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In the past, most PCs and server systems used separate SRAM chips for either their primary, secondary, or even tertiary caches. Today, thanks to **Moore's Law**, all levels of caches are integrated onto the processor chip, so the market for independent SRAM chips has nearly evaporated.



DRAM Technology

In a SRAM, as long as power is applied, the value can be kept indefinitely. In a *dynamic RAM* (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only one transistor per bit of storage, they are much denser and cheaper per bit than SRAM. As DRAMs store the charge on a capacitor, it cannot be kept indefinitely and must periodically be refreshed. That is why this memory structure is called dynamic, in contrast to the static storage

in an SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds. If every bit had to be read out of the DRAM and then written back individually, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs use a two-level decoding structure, and this allows us to refresh an entire *row* (which shares a word line) with a read cycle followed immediately by a write cycle.

Figure 5.4 shows the internal organization of a DRAM, and Figure 5.5 shows how the density, cost, and access time of DRAMs have changed over the years.

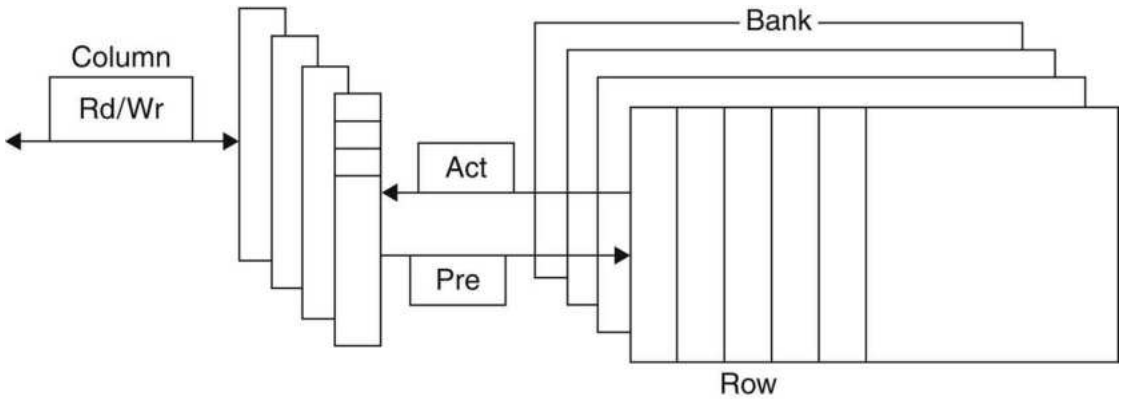


FIGURE 5.4 Internal organization of a DRAM. Modern DRAMs are organized in banks, typically four for DDR3. Each bank consists of a series of rows. Sending a PRE (precharge) command opens or closes a bank. A row address is sent with an Act (activate), which causes the row to transfer to a buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR3) or by specifying a block transfer and the starting address. Each command, as well as block transfers, is synchronized with a clock.

Year introduced	Chip size	$ per GiB	Total access time to a new row/column	Average column access time to existing row
1980	64 Kibibit	$1,500,000	250 ns	150 ns
1983	256 Kibibit	$500,000	185 ns	100 ns
1985	1 Mebibit	$200,000	135 ns	40 ns
1989	4 Mebibit	$50,000	110 ns	40 ns
1992	16 Mebibit	$15,000	90 ns	30 ns
1996	64 Mebibit	$10,000	60 ns	12 ns
1998	128 Mebibit	$4,000	60 ns	10 ns
2000	256 Mebibit	$1,000	55 ns	7 ns
2004	512 Mebibit	$250	50 ns	5 ns
2007	1 Gibibit	$50	45 ns	1.25 ns
2010	2 Gibibit	$30	40 ns	1 ns
2012	4 Gibibit	$1	35 ns	0.8 ns

FIGURE 5.5 DRAM size increased by multiples of four approximately once every 3 years until 1996, and thereafter considerably slower.
The improvements in access time have been slower but continuous, and cost roughly tracks density improvements, although cost is often affected by other issues, such as availability and demand. The cost per gibibyte is not adjusted for inflation.

The row organization that helps with refresh also helps with performance. To improve performance, DRAMs buffer rows for repeated access. The buffer acts like an SRAM; by changing the address, random bits can be accessed in the buffer until the next row access. This capability improves the access time significantly, since the access time to bits in the row is much lower. Making the chip wider also improves the memory bandwidth of the chip. When the row is in the buffer, it can be transferred by successive addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits), or by specifying a block transfer and the starting address within the buffer.

To improve the interface to processors further, DRAMs added clocks and are properly called synchronous DRAMs or SDRAMs. The advantage of SDRAMs is that the use of a clock eliminates the time for the memory and processor to synchronize. The speed advantage of synchronous DRAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, the clock transfers the successive bits in a burst. The fastest version is called *Double Data Rate (DDR)* SDRAM.

The name means data transfers on both the rising *and* falling edge of the clock, thereby getting twice as much bandwidth as you might expect based on the clock rate and the data width. The latest version of this technology is called DDR4. A DDR4-3200 DRAM can do 3200 million transfers per second, which means it has a 1600- MHz clock.

Sustaining that much bandwidth requires clever organization *inside* the DRAM. Instead of just a faster row buffer, the DRAM can be internally organized to read or write from multiple *banks*, with each having its own row buffer. Sending an address to several banks permits them all to read or write simultaneously. For example, with four banks, there is just one access time and then accesses rotate between the four banks to supply four times the bandwidth. This rotating access scheme is called *address interleaving*.

Although personal mobile devices like the iPad (see Chapter 1) use individual DRAMs, memory for servers is commonly sold on small boards called *dual inline memory modules* (DIMMs). DIMMs typically contain 4–16 DRAMs, and they are normally organized to be 8 bytes wide for server systems. A DIMM using DDR4-3200 SDRAMs could transfer at 8 ×3200 =25,600 megabytes per second. Such DIMMs are named after their bandwidth: PC25600. Since a DIMM can have so many DRAM chips that only a portion of them are used for a particular transfer, we need a term to refer to the subset of chips in a DIMM that share common address lines. To avoid confusion with the internal DRAM names of row and banks, we use the term *memory rank* for such a subset of chips in a DIMM.

Elaboration

One way to measure the performance of the memory system behind the caches is the Stream benchmark [McCalpin, 1995]. It measures the performance of long vector operations. They have no temporal locality and they access arrays that are larger than the cache of the computer being tested.

Flash Memory

Flash memory is a type of *electrically erasable programmable read-only memory* (EEPROM).

Unlike disks and DRAM, but like other EEPROM technologies,

writes can wear out flash memory bits. To cope with such limits, most flash products include a controller to spread the writes by remapping blocks that have been written many times to less trodden blocks. This technique is called *wear leveling*. With wear leveling, personal mobile devices are very unlikely to exceed the write limits in the flash. Such wear leveling lowers the potential performance of flash, but it is needed unless higher-level software monitors block wear. Flash controllers that perform wear leveling can also improve yield by mapping out memory cells that were manufactured incorrectly.

Disk Memory

As Figure 5.6 shows, a magnetic hard disk consists of a collection of platters, which rotate on a spindle at 5400 to 15,000 revolutions per minute. The metal platters are covered with magnetic recording material on both sides, similar to the material found on a cassette or videotape. To read and write information on a hard disk, a movable *arm* containing a small electromagnetic coil called a *read-write head* is located just above each surface. The entire drive is permanently sealed to control the environment inside the drive, which, in turn, allows the disk heads to be much closer to the drive surface.

FIGURE 5.6 A disk showing 10 disk platters and the read/write heads.
The diameter of today's disks is 2.5 or 3.5 inches, and there are typically one or two platters per drive today.

Each disk surface is divided into concentric circles, called **tracks**. There are typically tens of thousands of tracks per surface. Each track is in turn divided into **sectors** that contain the information; each track may have thousands of sectors. Sectors are typically 512

to 4096 bytes in size. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code (see Section 5.5), a gap, the sector number of the next sector, and so on.

track

One of thousands of concentric circles that make up the surface of a magnetic disk.

sector

One of the segments that make up a track on a magnetic disk; a sector is the smallest amount of information that is read or written on a disk.

The disk heads for each surface are connected together and move in conjunction, so that every head is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the heads at a given point on all surfaces.

To access data, the operating system must direct the disk through a three-stage process. The first step is to position the head over the proper track. This operation is called a **seek**, and the time to move the head to the desired track is called the *seek time*.

seek

The process of positioning a read/write head over the proper track on a disk.

Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average is open to wide interpretation because it depends on the seek distance. The industry calculates average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are usually advertised as 3 ms to 13 ms, but, depending on the application and scheduling of disk requests, the actual average seek time may be only 25% to 33% of the advertised number because of the locality of disk references. This locality arises both because of successive

accesses to the same file and because the operating system tries to schedule such accesses together.

Once the head has reached the correct track, we must wait for the desired sector to rotate under the read/write head. This time is called the **rotational latency** or **rotational delay**. The average latency to the desired information is halfway around the disk. Disks rotate at 5400 RPM to 15,000 RPM. The average rotational latency at 5400 RPM is

rotational latency

Also called **rotational delay**. The time required for the desired sector of a disk to rotate under the read/write head; usually assumed to be half the rotation time.

$$\text{Average rotational latency} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}} = \frac{0.5 \text{ rotation}}{5400 \text{ RPM}/\left(60 \dfrac{\text{seconds}}{\text{minute}}\right)}$$
$$= 0.0056 \text{ seconds} = 5.6 \text{ ms}$$

The last component of a disk access, *transfer time*, is the time to transfer a block of bits. The transfer time is a function of the sector size, the rotation speed, and the recording density of a track. Transfer rates in 2012 were between 100 and 200 MB/sec.

One complication is that most disk controllers have a built-in cache that stores sectors as they are passed over; transfer rates from the cache are typically higher, and were up to 750 MB/sec (6 Gbit/sec) in 2012.

Alas, where block numbers are located is no longer intuitive. The assumptions of the sector-track-cylinder model above are that nearby blocks are on the same track, blocks in the same cylinder take less time to access since there is no seek time, and some tracks are closer than others. The reason for the change was the raising of the level of the disk interfaces. To speed-up sequential transfers, these higher-level interfaces organize disks more like tapes than like random access devices. The logical blocks are ordered in serpentine fashion across a single surface, trying to capture all the sectors that are recorded at the same bit density to try to get best performance. Hence, sequential blocks may be on different tracks.

In summary, the two primary differences between magnetic disks and semiconductor memory technologies are that disks have a slower access time because they are mechanical devices—flash is 1000 times as fast and DRAM is 100,000 times as fast—yet they are cheaper per bit because they have very high storage capacity at a modest cost—disks are 10 to 100 times cheaper. Magnetic disks are nonvolatile like flash, but unlike flash there is no write wear-out problem. However, flash is much more rugged and hence a better match to the jostling inherent in personal mobile devices.

5.3 The Basics of Caches

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of the American Language, Third College Edition, 1988

In our library example, the desk acted as a cache—a safe place to store things (books) that we needed to examine. *Cache* was the name chosen to represent the level of the memory hierarchy between the processor and main memory in the first commercial computer to have this extra level. The memories in the datapath in Chapter 4 are simply replaced by caches. Today, although this remains the dominant use of the word *cache*, the term is also used to refer to any storage managed to take advantage of locality of access. Caches first appeared in research computers in the early 1960s and in production computers later in that same decade; every general-purpose computer built now from servers to low-power embedded processors, includes caches.

In this section, we begin by looking at a very simple cache in which the processor requests are each one word, and the blocks also consist of a single word. (Readers already familiar with cache basics may want to skip to Section 5.4.) Figure 5.7 shows such a simple cache, before and after requesting a data item that is not initially in the cache. Before the request, the cache contains a collection of recent references $X_1, X_2, \ldots, X_{n-1}$, and the processor requests a word X_n that is not in the cache. This request results in a miss, and the word X_n is brought from memory into the cache.

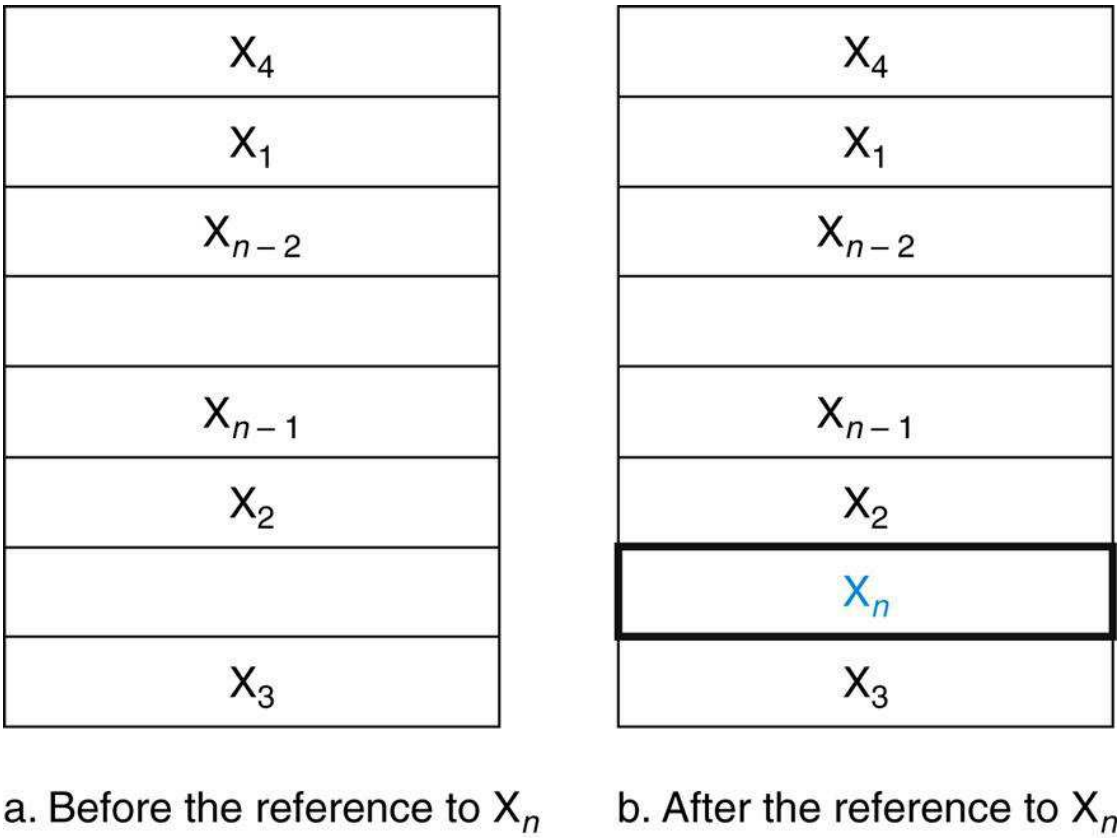


FIGURE 5.7 The cache just before and just after a reference to a word X_n that is not initially in the cache.
This reference causes a miss that forces the cache to fetch X_n from memory and insert it into the cache.

In looking at the scenario in Figure 5.7, there are two questions to answer: How do we know if a data item is in the cache? Moreover, if it is, how do we find it? The answers are related. If each word can go in exactly one place in the cache, then it is straightforward to find the word if it is in the cache. The simplest way to assign a location in the cache for each word in memory is to assign the cache location based on the *address* of the word in memory. This cache structure is called **direct mapped**, since each memory location is mapped directly to exactly one location in the cache. The typical mapping between addresses and cache locations for a direct-mapped cache is usually simple. For example, almost all direct-mapped caches use this mapping to find a block:

direct-mapped cache

A cache structure in which each memory location is mapped to

exactly one location in the cache.

(Block address) modulo (Number of blocks in the cache)

If the number of entries in the cache is a power of 2, then modulo can be computed simply by using the low-order $\log_2$ (cache size in blocks) bits of the address. Thus, an 8-block cache uses the three lowest bits ($8=2^3$) of the block address. For example, Figure 5.8 shows how the memory addresses between 1_{ten} (00001_{two}) and 29_{ten} (11101_{two}) map to locations 1_{ten} (001_{two}) and 5_{ten} (101_{two}) in a direct-mapped cache of eight words.

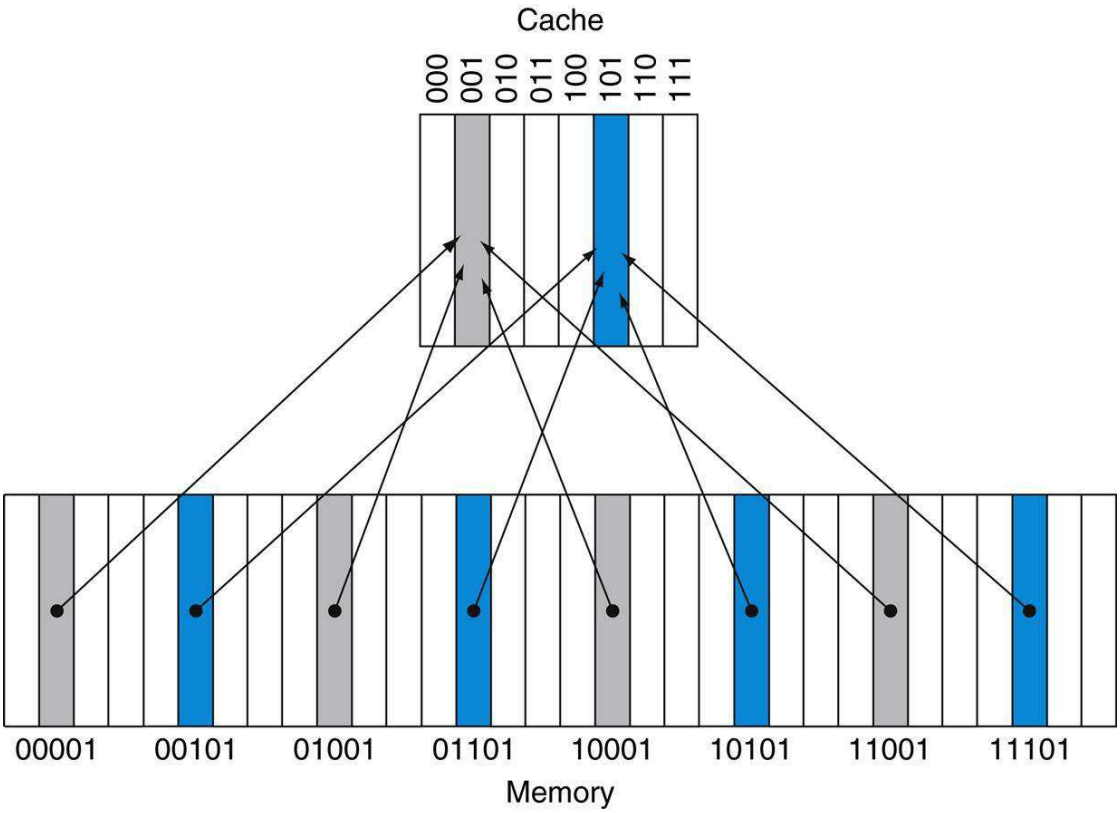


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations.

Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8)=3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{two}, and 11001_{two} all map to entry 001_{two}

of the cache, while addresses 00101_{two}, 01101_{two}, 10101_{two}, and 11101_{two} all map to entry 101_{two} of the cache.

Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to the cache. The tags contain the address information required to identify whether a word in the cache corresponds to the requested word. The tag needs just to contain the upper portion of the address, corresponding to the bits that are not used as an index into the cache. For example, in Figure 5.8 we need only have the upper two of the five address bits in the tag, since the lower 3-bit index field of the address selects the block. Architects omit the index bits because they are redundant, since by definition, the index field of any address of a cache block must be that block number.

tag

A field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word.

We also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty, as in Figure 5.7. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a **valid bit** to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

valid bit

A field in the tables of a memory hierarchy that indicates that the associated block in the hierarchy contains valid data.

For the rest of this section, we will focus on explaining how a cache deals with reads. In general, handling reads is a little simpler than handling writes, since reads do not have to change the contents of the cache. After seeing the basics of how reads work and how cache misses can be handled, we'll examine the cache designs for real computers and detail how these caches handle writes.

The BIG Picture

Caching is perhaps the most important example of the big idea of **prediction**. It relies on the principle of locality to try to find the desired data in the higher levels of the memory hierarchy, and provides mechanisms to ensure that when the prediction is wrong it finds and uses the proper data from the lower levels of the memory hierarchy. The hit rates of the cache prediction on modern computers are often above 95% (see Figure 5.46).

Accessing a Cache

Below is a sequence of nine memory references to an empty eight-block cache, including the action for each reference. Figure 5.9 shows how the contents of the cache change on each miss. Since there are eight blocks in the cache, the low-order 3 bits of an address give the block number:

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.9b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (5.9c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (5.9d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (5.9e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (5.9f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

FIGURE 5.9 The cache contents are shown after each reference request that misses, with the index and tag fields shown in binary for the sequence of addresses on page 379.

The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110_{two} (miss), 11010_{two} (miss), 10110_{two} (hit), 11010_{two} (hit), 10000_{two} (miss), 00011_{two} (miss), 10000_{two} (hit), 10010_{two} (miss), and 10000_{two} (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010_{two} (18) is referenced, the entry for address 11010_{two} (26) must be replaced, and a reference to 11010_{two} will cause a subsequent miss. The tag field will contain only the upper portion of the address. The full address of a word contained in cache block *i* with tag field *j* for this cache is $j \times 8 + i$, or equivalently the concatenation of the tag field *j* and the index *i*. For example, in cache *f* above, index 010_{two}

has tag 10_{two} and corresponds to address 10010_{two}.

Since the cache is empty, several of the first references are misses; the caption of Figure 5.9 describes the actions for each memory reference. On the eighth reference we have conflicting demands for a block. The word at address 18 (10010_{two}) should be brought into cache block 2 (010_{two}). Hence, it must replace the word at address 26 (11010_{two}), which is already in cache block 2 (010_{two}). This behavior allows a cache to take advantage of temporal locality: recently referenced words replace less recently referenced words.

This situation is directly analogous to needing a book from the shelves and having no more space on your desk—some book already on your desk must be returned to the shelves. In a direct-mapped cache, there is only one place to put the newly requested item and hence just one choice of what to replace.

We know where to look in the cache for each possible address: the low-order bits of an address can be used to find the unique cache entry to which the address could map. Figure 5.10 shows how a referenced address is divided into

- A *tag field*, which is used to compare with the value of the tag field of the cache
- A *cache index*, which is used to select the block

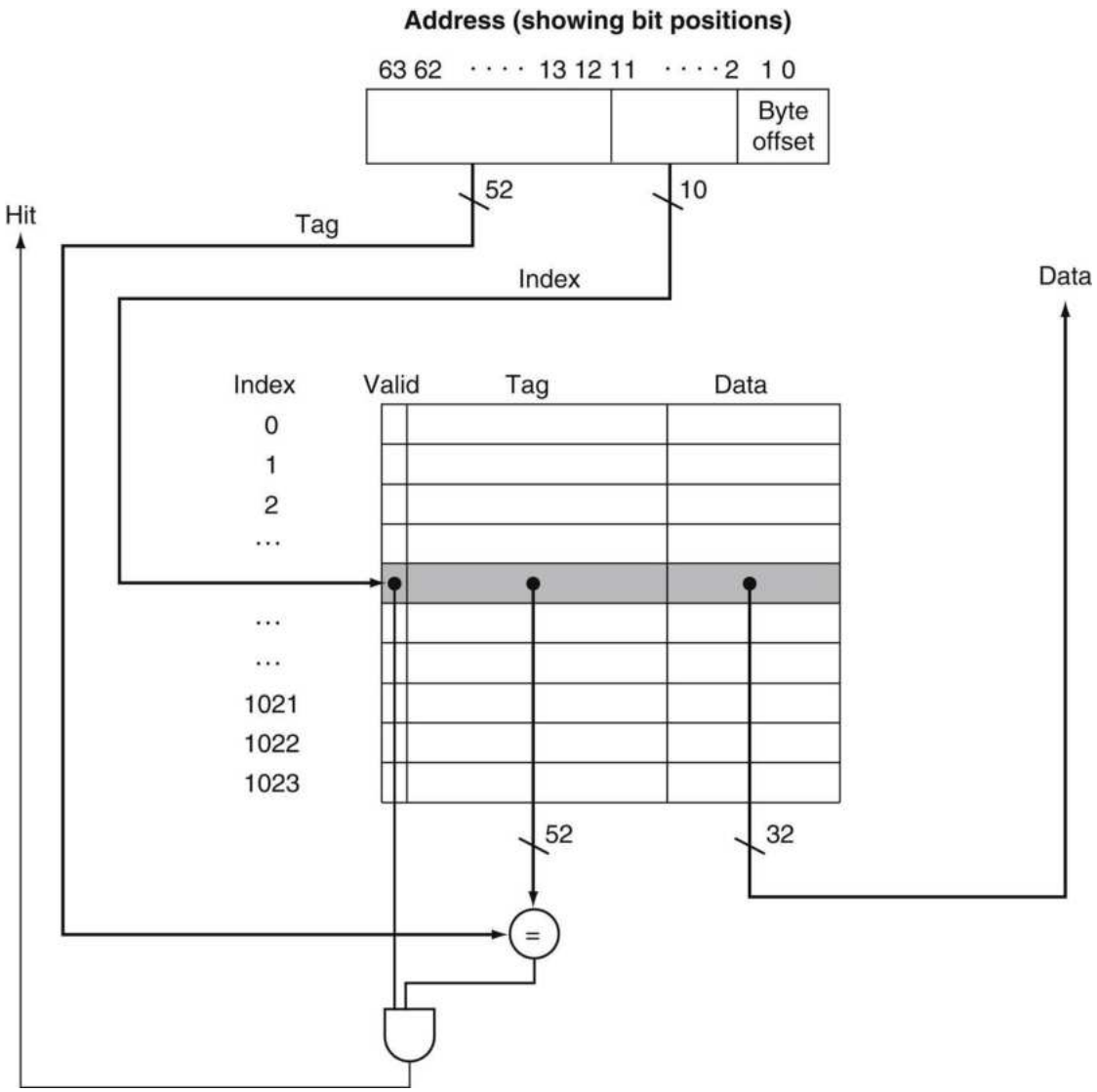


FIGURE 5.10 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag.
This cache holds 1024 words or 4 KiB. Unless noted otherwise, we assume 64-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving 64 −10 −2 =52 bits to be compared against the tag. If the tag and upper 52 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

The index of a cache block, together with the tag contents of that

block, uniquely specifies the memory address of the word contained in the cache block. Because the index field is used as an address to reference the cache, and because an n-bit field has 2^n values, the total number of entries in a direct-mapped cache must be a power of 2. Since words are aligned to multiples of four bytes, the least significant two bits of every address specify a byte within a word. Hence, if the words are aligned in memory, the least significant two bits can be ignored when selecting a word in the block. For this chapter, we'll assume that data are aligned in memory, and discuss how to handle unaligned cache accesses in an Elaboration.

The total number of bits needed for a cache is a function of the cache size and the address size, because the cache includes both the storage for the data and the tags. The size of the block above was one word (4 bytes), but normally it is several. For the following situation:

- 64-bit addresses
- A direct-mapped cache
- The cache size is 2^n blocks, so n bits are used for the index
- The block size is 2^m words (2^{m+2} bytes), so m bits are used for the word within the block, and two bits are used for the byte part of the address

The size of the tag field is

$$64 - (n + m + 2).$$

The total number of bits in a direct-mapped cache is

$$2^n \times (\text{block size} + \text{tag size} + \text{valid field size}).$$

Since the block size is 2^m words (2^{m+5} bits), and we need 1 bit for the valid field, the number of bits in such a cache is

$$2^n \times (2^m \times 32 + (64 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 63 - n - m).$$

Although this is the actual size in bits, the naming convention is to exclude the size of the tag and valid field and to count only the

size of the data. Thus, the cache in Figure 5.10 is called a 4 KiB cache.

Bits in a Cache

Example

How many total bits are required for a direct-mapped cache with 16 KiB of data and four-word blocks, assuming a 64-bit address?

Answer

We know that 16 KiB is 4096 (2^{12}) words. With a block size of four words (2^2), there are 1024 (2^{10}) blocks. Each block has 4 ×32 or 128 bits of data plus a tag, which is 64 −10 −2 −2 bits, plus a valid bit. Thus, the complete cache size is

$$2^{10} \times (4 \times 32 + (64 - 10 - 2 - 2) + 1) = 2^{10} \times 179 = 179 \text{ Kibibits}$$

or 22.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.4 times as many as needed just for the storage of the data.

Mapping an Address to a Multiword Cache Block

Example

Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

Answer

We saw the formula on page 376. The block is given by

$$(\text{Block address}) \text{ modulo } (\text{Number of blocks in the cache})$$

where the address of the block is

$$\frac{\text{Byte address}}{\text{Bytes per block}}$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

which maps to cache block number (75 modulo 64)=11. In fact, this block maps all addresses between 1200 and 1215.

Larger blocks exploit spatial locality to lower miss rates. As Figure 5.11 shows, increasing the block size usually decreases the miss rate. The miss rate may go up eventually if the block size becomes a significant fraction of the cache size, because the number of blocks that can be held in the cache will become small, and there will be a great deal of competition for those blocks. As a result, a block will be bumped out of the cache before many of its words are accessed. Stated alternatively, spatial locality among the words in a block decreases with a very large block; consequently, the benefits to the miss rate become smaller.

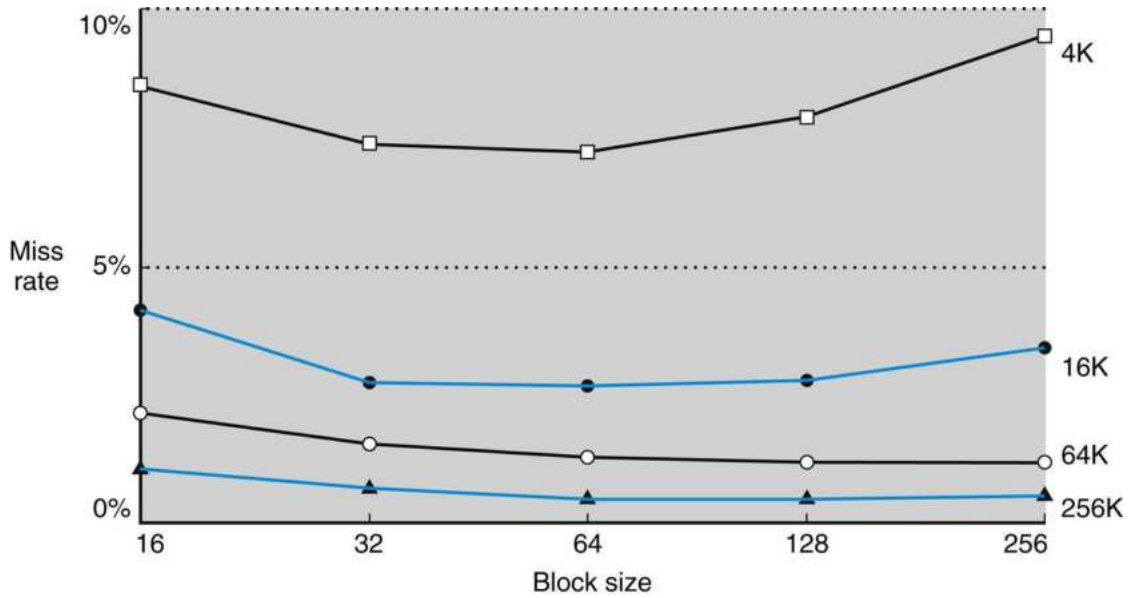


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so these data are based on SPEC92.

A more serious problem associated with just increasing the block size is that the cost of a miss rises. The miss penalty is determined by the time required to fetch the block from the next lower level of the hierarchy and load it into the cache. The time to fetch the block has two parts: the latency to the first word and the transfer time for the rest of the block. Clearly, unless we change the memory system, the transfer time—and hence the miss penalty—will likely increase as the block size expands. Furthermore, the improvement in the miss rate starts to decrease as the blocks become larger. The result is that the increase in the miss penalty overwhelms the decrease in the miss rate for blocks that are too large, and cache performance thus decreases. Of course, if we design the memory to transfer larger blocks more efficiently, we can increase the block size and obtain further improvements in cache performance. We discuss this topic in the next section.

Elaboration

Although it is hard to do anything about the longer latency

component of the miss penalty for large blocks, we may be able to hide some of the transfer time so that the miss penalty is effectively smaller. The easiest method for doing this, called *early restart*, is simply to resume execution as soon as the requested word of the block is returned, rather than wait for the entire block. Many processors use this technique for instruction access, where it works best. Instruction accesses are largely sequential, so if the memory system can deliver a word every clock cycle, the processor may be able to restart operation when the requested word is returned, with the memory system delivering new instruction words just in time. This technique is usually less effective for data caches because it is likely that the words will be requested from the block in a less predictable way, and the probability that the processor will need another word from a different cache block before the transfer completes is high. If the processor cannot access the data cache because a transfer is ongoing, then it must stall.

An even more sophisticated scheme is to organize the memory so that the requested word is transferred from the memory to the cache first. The remainder of the block is then transferred, starting with the address after the requested word and wrapping around to the beginning of the block. This technique, called *requested word first* or *critical word first*, can be slightly faster than early restart, but it is limited by the same properties that restrain early restart.

Handling Cache Misses

Before we look at the cache of a real system, let's see how the control unit deals with **cache misses**. (We describe a cache controller in detail in Section 5.9.) The control unit must detect a miss and process the miss by fetching the requested data from memory (or, as we shall see, a lower-level cache). If the cache reports a hit, the computer continues using the data as if nothing happened.

cache miss

A request for data from the cache that cannot be filled because the data are not present in the cache.

Modifying the control of a processor to handle a hit is trivial;

misses, however, require some extra work. The cache miss handling is done in collaboration with the processor control unit and with a separate controller that initiates the memory access and refills the cache. The processing of a cache miss creates a pipeline stall (Chapter 4) in contrast to an exception or interrupt, which would require saving the state of all registers. For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory. More sophisticated out-of-order processors can allow execution of instructions while waiting for a cache miss, but we'll assume in-order processors that stall on cache misses in this section.

Let's look a little more closely at how instruction misses are handled; the same approach can be easily extended to handle data misses. If an instruction access results in a miss, then the content of the Instruction register is invalid. To get the proper instruction into the cache, we must be able to tell the lower level in the memory hierarchy to perform a read. Since the program counter is incremented in the first clock cycle of execution, the address of the instruction that generates an instruction cache miss is equal to the value of the program counter minus 4. Once we have the address, we need to instruct the main memory to perform a read. We wait for the memory to respond (since the access will take multiple clock cycles), and then write the words containing the desired instruction into the cache.

We can now define the steps to be taken on an instruction cache miss:

1. Send the original PC value to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address (from the ALU) into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refetch the instruction, this time finding it in the cache.

The control of the cache on a data access is essentially identical: on a miss, we simply stall the processor until the memory responds with the data.

Handling Writes

Writes work somewhat differently. Suppose on a store instruction, we wrote the data into only the data cache (without changing main memory); then, after the write into the cache, memory would have a different value from that in the cache. In such a case, the cache and memory are said to be *inconsistent*. The simplest way to keep the main memory and the cache consistent is always to write the data into both the memory and the cache. This scheme is called **write-through**.

write-through

A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data are always consistent between the two.

The other key aspect of writes is what occurs on a write miss. We first fetch the words of the block from memory. After the block is fetched and placed into the cache, we can overwrite the word that caused the miss into the cache block. We also write the word to main memory using the full address.

Although this design handles writes very simply, it would not provide good performance. With a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For example, suppose 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of 1.0 +100 ×10%=11, reducing performance by more than a factor of 10.

One solution to this problem is to use a **write buffer**. A write buffer stores the data while they are waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an empty position in the write buffer. Of course, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help, because writes are being generated faster than

the memory system can accept them.

write buffer

A queue that holds data while the data are waiting to be written to memory.

The rate at which writes are generated may also be *less* than the rate at which the memory can accept them, and yet stalls may still occur. This can happen when the writes occur in bursts. To reduce the occurrence of such stalls, processors usually increase the depth of the write buffer beyond a single entry.

The alternative to a write-through scheme is a scheme called **write-back**. In a write-back scheme, when a write occurs, the new value is written only to the block in the cache. The modified block is written to the lower level of the hierarchy when it is replaced. Write-back schemes can improve performance, especially when processors can generate writes as fast or faster than the writes can be handled by main memory; a write-back scheme is, however, more complex to implement than write-through.

write-back

A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced.

In the rest of this section, we describe caches from real processors, and we examine how they handle both reads and writes. In Section 5.8, we will describe the handling of writes in more detail.

Elaboration

Writes introduce several complications into caches that are not present for reads. Here we discuss two of them: the policy on write misses and efficient implementation of writes in write-back caches.

Consider a miss in a write-through cache. The most common strategy is to allocate a block in the cache, called *write allocate*. The block is fetched from memory and then the appropriate portion of

the block is overwritten. An alternative strategy is to update the portion of the block in memory but not put it in the cache, called *no write allocate*. The motivation is that sometimes programs write entire blocks of data, such as when the operating system zeros a page of memory. In such cases, the fetch associated with the initial write miss may be unnecessary. Some computers allow the write allocation policy to be changed on a per-page basis.

Actually implementing stores efficiently in a cache that uses a write-back strategy is more complex than in a write-through cache. A write-through cache can write the data into the cache and read the tag; if the tag mismatches, then a miss occurs. Because the cache is write-through, the overwriting of the block in the cache is not catastrophic, since memory has the correct value. In a write-back cache, we must first write the block back to memory if the data in the cache are modified and we have a cache miss. If we simply overwrote the block on a store instruction before we knew whether the store had hit in the cache (as we could for a write-through cache), we would destroy the contents of the block, which is not backed up in the next lower level of the memory hierarchy.

In a write-back cache, because we cannot overwrite the block, stores either require two cycles (a cycle to check for a hit followed by a cycle to actually perform the write) or require a write buffer to hold that data—effectively allowing the store to take only one cycle by pipelining it. When a store buffer is used, the processor does the cache lookup and places the data in the store buffer during the normal cache access cycle. Assuming a cache hit, the new data are written from the store buffer into the cache on the next unused cache access cycle.

By comparison, in a write-through cache, writes can always be done in one cycle. We read the tag and write the data portion of the selected block. If the tag matches the address of the block being written, the processor can continue normally, since the correct block has been updated. If the tag does not match, the processor generates a write miss to fetch the rest of the block corresponding to that address.

Many write-back caches also include write buffers that are used to reduce the miss penalty when a miss replaces a modified block. In such a case, the modified block is moved to a write-back buffer associated with the cache while the requested block is read from

memory. The write-back buffer is later written back to memory. Assuming another miss does not occur immediately, this technique halves the miss penalty when a dirty block must be replaced.

An Example Cache: The Intrinsity FastMATH Processor

The Intrinsity FastMATH is an embedded microprocessor that uses the MIPS architecture and a simple cache implementation. Near the end of the chapter, we will examine the more complex cache designs of ARM and Intel microprocessors, but we start with this simple, yet real, example for pedagogical reasons. Figure 5.12 shows the organization of the Intrinsity FastMATH data cache. Note that the address size for this computer is just 32 bits, not 64 as in the rest of the book.

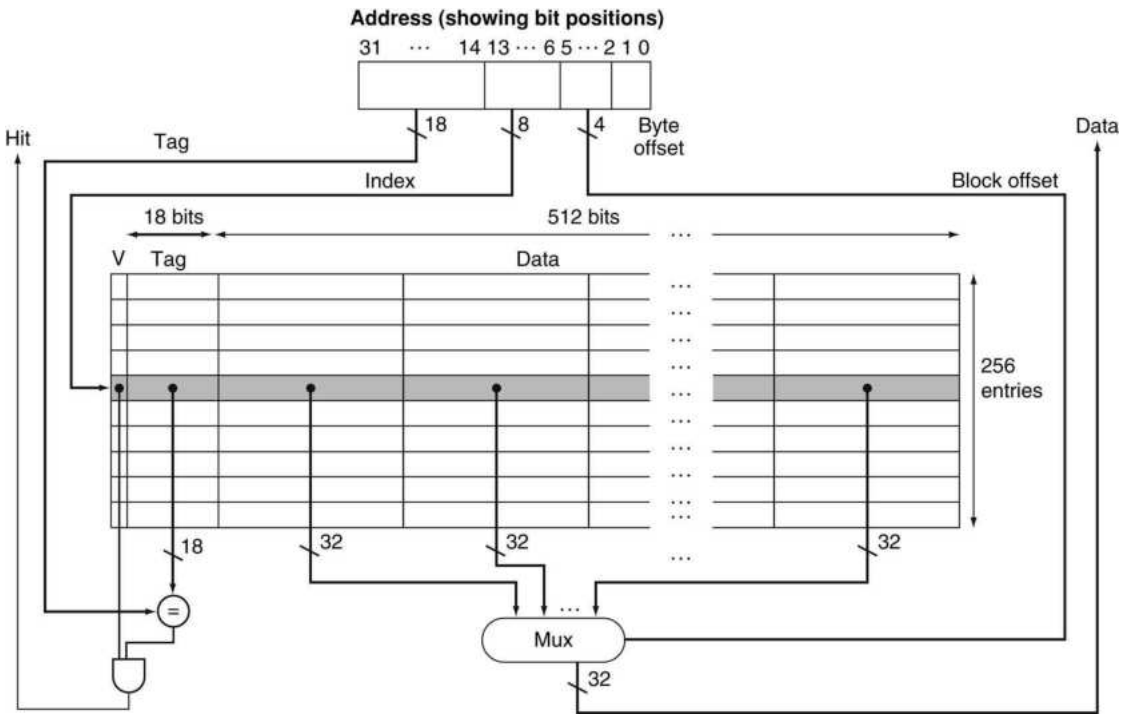


FIGURE 5.12 The 16 KiB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block.
Note that the address size for this computer is just 32 bits. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache.

This processor has a 12-stage pipeline. When operating at peak speed, the processor can request both an instruction word and a data word on every clock. To satisfy the demands of the pipeline without stalling, separate instruction and data caches are used. Each cache is 16 KiB, or 4096 words, with 16-word blocks.

Read requests for the cache are straightforward. Because there are separate data and instruction caches, we need separate control signals to read and write each cache. (Remember that we need to update the instruction cache when a miss occurs.) Thus, the steps for a read request to either cache are as follows:

1. Send the address to the appropriate cache. The address comes either from the PC (for an instruction) or from the ALU (for data).
2. If the cache signals hit, the requested word is available on the data lines. Since there are 16 words in the desired block, we need to select the right one. A block index field is used to control the multiplexor (shown at the bottom of the figure), which selects the requested word from the 16 words in the indexed block.
3. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request.

For writes, the Intrinsity FastMATH offers both write-through and write-back, leaving it up to the operating system to decide which strategy to use for an application. It has a one-entry write buffer.

What cache miss rates are attained with a cache structure like that used by the Intrinsity FastMATH? Figure 5.13 shows the miss rates for the instruction and data caches. The combined miss rate is the effective miss rate per reference for each program after accounting for the differing frequency of instruction and data accesses.

Instruction miss rate	Data miss rate	Effective combined miss rate
0.4%	11.4%	3.2%

FIGURE 5.13 Approximate instruction and data miss rates for the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks.
The combined miss rate is the effective miss rate seen for the combination of the 16 KiB instruction cache and 16 KiB data cache. It is obtained by weighting the instruction and data individual miss rates by the frequency of instruction and data references.

Although miss rate is an important characteristic of cache designs, the ultimate measure will be the effect of the memory system on program execution time; we'll see how miss rate and execution time are related shortly.

Elaboration

A combined cache with a total size equal to the sum of the two

split caches will usually have a better hit rate. This higher rate occurs because the combined cache does not rigidly divide the number of entries that may be used by instructions from those that may be used by data. Nonetheless, almost all processors today use split instruction and data caches to increase cache *bandwidth* to match what modern pipelines expect. (There may also be fewer conflict misses; see Section 5.8.)

Here are miss rates for caches the size of those found in the Intrinsity FastMATH processor, and for a combined cache whose size is equal to the sum of the two caches:

- Total cache size: 32 KiB
- Split cache effective miss rate: 3.24%
- Combined cache miss rate: 3.18%

The miss rate of the split cache is only slightly worse.

The advantage of doubling the cache bandwidth, by supporting both an instruction and data access simultaneously, easily overcomes the disadvantage of a slightly increased miss rate. This observation cautions us that we cannot use miss rate as the sole measure of cache performance, as Section 5.4 shows.

split cache

A scheme in which a level of the memory hierarchy is composed of two independent caches that operate in parallel with each other, with one handling instructions and one handling data.

Summary

We began the previous section by examining the simplest of caches: a direct-mapped cache with a one-word block. In such a cache, both hits and misses are simple, since a word can go in exactly one location and there is a separate tag for every word. To keep the cache and memory consistent, a write-through scheme can be used, so that every write into the cache also causes memory to be updated. The alternative to write-through is a write-back scheme that copies a block back to memory when it is replaced; we'll discuss this scheme further in upcoming sections.

To take advantage of spatial locality, a cache must have a block size larger than one word. The use of a bigger block decreases the miss rate and improves the efficiency of the cache by reducing the

amount of tag storage relative to the amount of data storage in the cache. Although a larger block size decreases the miss rate, it can also increase the miss penalty. If the miss penalty increased linearly with the block size, larger blocks could easily lead to lower performance.

To avoid performance loss, the bandwidth of main memory is increased to transfer cache blocks more efficiently. Common methods for increasing bandwidth external to the DRAM are making the memory wider and interleaving. DRAM designers have steadily improved the interface between the processor and memory to increase the bandwidth of burst mode transfers to reduce the cost of larger cache block sizes.

Check Yourself

The speed of the memory system affects the designer's decision on the size of the cache block. Which of the following cache designer guidelines is generally valid?

1. The shorter the memory latency, the smaller the cache block
2. The shorter the memory latency, the larger the cache block
3. The higher the memory bandwidth, the smaller the cache block
4. The higher the memory bandwidth, the larger the cache block

5.4 Measuring and Improving Cache Performance

In this section, we begin by examining ways to measure and analyze cache performance. We then explore two different techniques for improving cache performance. One focuses on reducing the miss rate by reducing the probability that two distinct memory blocks will contend for the same cache location. The second technique reduces the miss penalty by adding an additional level to the hierarchy. This technique, called *multilevel caching*, first appeared in high-end computers selling for more than $100,000 in 1990; since then it has become common on personal mobile devices selling for a few hundred dollars!

CPU time can be divided into the clock cycles that the CPU spends executing the program and the clock cycles that the CPU

spends waiting for the memory system. Normally, we assume that the costs of cache accesses that are hits are part of the normal CPU execution cycles. Thus,

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

The memory-stall clock cycles come primarily from cache misses, and we make that assumption here. We also restrict the discussion to a simplified model of the memory system. In real processors, the stalls generated by reads and writes can be quite complex, and accurate performance prediction usually requires very detailed simulations of the processor and memory system.

Memory-stall clock cycles can be defined as the sum of the stall cycles coming from reads plus those coming from writes:

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

The read-stall cycles can be defined in terms of the number of read accesses per program, the miss penalty in clock cycles for a read, and the read miss rate:

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

Writes are more complicated. For a write-through scheme, we have two sources of stalls: write misses, which usually require that we fetch the block before continuing the write (see the *Elaboration* on page 386 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write happens. Thus, the cycles stalled for writes equal the sum of these two:

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

Because the write buffer stalls depend on the proximity of writes, and not just the frequency, it is impossible to give a simple equation to compute such stalls. Fortunately, in systems with a reasonable write buffer depth (e.g., four or more words) and a memory capable of accepting writes at a rate that significantly exceeds the average write frequency in programs (e.g., by a factor of 2), the write buffer stalls will be small, and we can safely ignore them. If a system did not meet these criteria, it would not be well designed; instead, the designer should have used either a deeper write buffer or a write-back organization.

Write-back schemes also have potential additional stalls arising from the need to write a cache block back to memory when the block is replaced. We will discuss this more in Section 5.8.

In most write-through cache organizations, the read and write miss penalties are the same (the time to fetch the block from memory). If we assume that the write buffer stalls are negligible, we can combine the reads and writes by using a single miss rate and the miss penalty:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

We can also factor this as

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Let's consider a simple example to help us understand the impact of cache performance on processor performance.

Calculating Cache Performance

Example

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect

cache that never missed. Assume the frequency of all loads and stores is 36%.

Answer

The number of memory miss cycles for instructions in terms of the Instruction count (I) is

$$\text{Instruction miss cycles} = I \times 2\% \times 100 = 2.00 \times I$$

As the frequency of all loads and stores is 36%, we can find the number of memory miss cycles for data references:

$$\text{Data miss cycles} = I \times 36\% \times 4\% \times 100 = 1.44 \times I$$

The total number of memory-stall cycles is 2.00 I +1.44 I =3.44 I. This is more than three cycles of memory stall per instruction. Accordingly, the total CPI including memory stalls is 2 +3.44 =5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{I \times CPI_{stall} \times \text{Clock cycle}}{I \times CPI_{perfect} \times \text{Clock cycle}} = \frac{CPI_{stall}}{CPI_{perfect}} = \frac{5.44}{2}$$

The performance with the perfect cache is better by $\frac{5.44}{2} = 2.72$.

What happens if the processor is made faster, but the memory system is not? The amount of time spent on memory stalls will take up an increasing fraction of the execution time; Amdahl's Law, which we examined in Chapter 1, reminds us of this fact. A few simple examples show how serious this problem can be. Suppose we speed-up the computer in the previous example by reducing its CPI from 2 to 1 without changing the clock rate, which might be done with an improved pipeline. The system with cache misses

would then have a CPI of 1 +3.44 =4.44, and the system with the perfect cache would be

$$\frac{4.44}{1} = 4.44 \text{ times as fast.}$$

The amount of execution time spent on memory stalls would have risen from

$$\frac{3.44}{5.44} = 63\%$$

to

$$\frac{3.44}{4.44} = 77\%$$

Similarly, increasing the clock rate without changing the memory system also increases the performance lost due to cache misses.

The previous examples and equations assume that the hit time is not a factor in determining cache performance. Clearly, if the hit time increases, the total time to access a word from the memory system will increase, possibly causing an increase in the processor cycle time. Although we will see additional examples of what can raise hit time shortly, one example is increasing the cache size. A larger cache could clearly have a bigger access time, just as, if your desk in the library was very large (say, 3 square meters), it would take longer to locate a book on the desk. An increase in hit time likely adds another stage to the pipeline, since it may take multiple cycles for a cache hit. Although it is more complex to calculate the performance impact of a deeper pipeline, at some point the increase in hit time for a larger cache could dominate the improvement in hit rate, leading to a decrease in processor performance.

To capture the fact that the time to access data for both hits and misses affects performance, designers sometime use *average memory access time* (AMAT) as a way to examine alternative cache designs. Average memory access time is the average time to access memory considering both hits and misses and the frequency of different

accesses; it is equal to the following:

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

Calculating Average Memory Access Time

Example

Find the AMAT for a processor with a 1 ns clock cycle time, a miss penalty of 20 clock cycles, a miss rate of 0.05 misses per instruction, and a cache access time (including hit detection) of 1 clock cycle. Assume that the read and write miss penalties are the same and ignore other write stalls.

Answer

The average memory access time per instruction is

$$\begin{aligned}\text{AMAT} &= \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}\\ &= 1 + 0.05 \times 20\\ &= 2 \text{ clock cycles}\end{aligned}$$

or 2 ns.

The next subsection discusses alternative cache organizations that decrease miss rate but may sometimes increase hit time; additional examples appear in Section 5.16.

Reducing Cache Misses by More Flexible Placement of Blocks

So far, when we put a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. However, there is actually a whole range of schemes for placing blocks. Direct mapped, where a block can be placed in exactly one location, is at one extreme.

At the other extreme is a scheme where a block can be placed in *any* location in the cache. Such a scheme is called **fully associative**, because a block in memory may be associated with any entry in the cache. To find a given block in a fully associative cache, all the entries in the cache must be searched because a block can be placed in any one. To make the search practical, it is done in parallel with a comparator associated with each cache entry. These comparators significantly increase the hardware cost, effectively making fully associative placement practical only for caches with small numbers of blocks.

fully associative cache

A cache structure in which a block can be placed in any location in the cache.

The middle range of designs between direct mapped and fully associative is called **set associative**. In a set-associative cache, there are a fixed number of locations where each block can be placed. A set-associative cache with n locations for a block is called an n-way set-associative cache. An n-way set-associative cache consists of a number of sets, each of which consists of n blocks. Each block in the memory maps to a unique *set* in the cache given by the index field, and a block can be placed in *any* element of that set. Thus, a set-associative placement combines direct-mapped placement and fully associative placement: a block is directly mapped into a set, and then all the blocks in the set are searched for a match. For example, Figure 5.14 shows where block 12 may be put in a cache with eight blocks total, according to the three block placement policies.

set-associative cache

A cache that has a fixed number of locations (at least two) where each block can be placed.

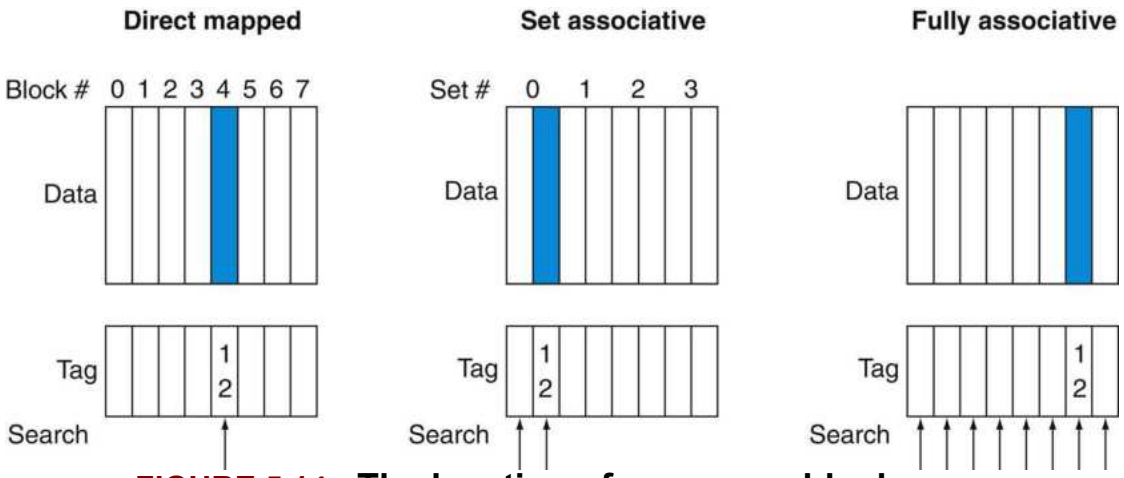


FIGURE 5.14 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement.
In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8)=4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4)=0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Remember that in a direct-mapped cache, the position of a memory block is given by

(Block number) modulo (Number of *blocks* in the cache)

In a set-associative cache, the set containing a memory block is given by

(Block number) modulo (Number of *sets* in the cache)

Since the block may be placed in any element of the set, *all the tags of all the elements of the set* must be searched. In a fully associative cache, the block can go anywhere, and *all tags of all the blocks in the cache* must be searched.

We can also think of all block placement strategies as a variation on set associativity. Figure 5.15 shows the possible associativity

structures for an eight-block cache. A direct-mapped cache is just a one-way set-associative cache: each cache entry holds one block and each set has one element. A fully associative cache with m entries is simply an m-way set-associative cache; it has one set with m blocks, and an entry can reside in any block within that set.

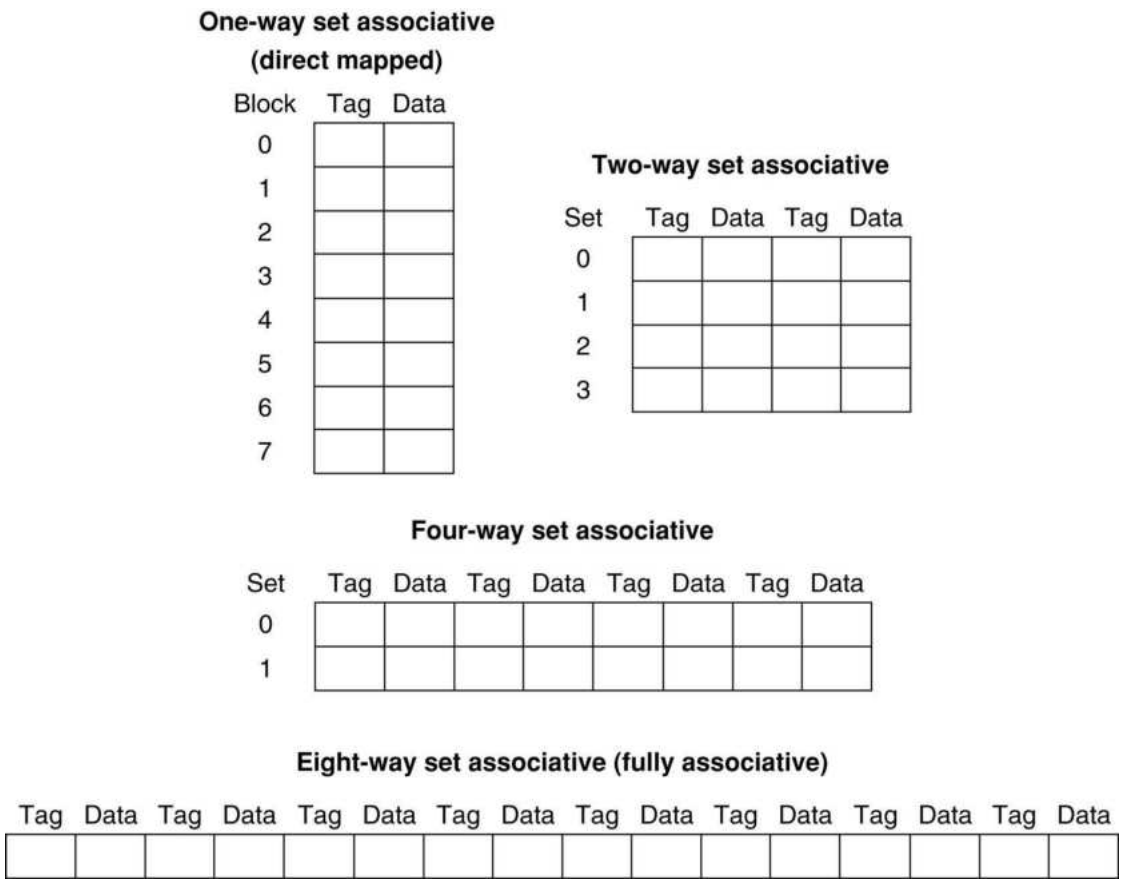


FIGURE 5.15 An eight-block cache configured as direct-mapped, two-way set associative, four-way set associative, and fully associative.
The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

The advantage of increasing the degree of associativity is that it usually decreases the miss rate, as the next example shows. The main disadvantage, which we discuss in more detail shortly, is a potential increase in the hit time.

Misses and Associativity in Caches

Example

Assume there are three small caches, each consisting of four one-word blocks. One cache is fully associative, a second is two-way set

associative, and the third is direct-mapped. Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, and 8.

Answer

The direct-mapped case is easiest. First, let's determine to which cache block each block address maps:

Block address	Cache block
0	(0 modulo 4)=0
6	(6 modulo 4)=2
8	(8 modulo 4)=0

Now we can fill in the cache contents after each reference, using a blank entry to mean that the block is invalid, colored text to show a new entry added to the cache for the associated reference, and plain text to show an old entry in the cache:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

The direct-mapped cache generates five misses for the five accesses.

The set-associative cache has two sets (with indices 0 and 1) with two elements per set. Let's first determine to which set each block address maps:

Block address	Cache set
0	(0 modulo 2)=0
6	(6 modulo 2)=0
8	(8 modulo 2)=0

Because we have a choice of which entry in a set to replace on a miss, we need a replacement rule. Set-associative caches usually replace the least recently used block within a set; that is, the block that was used furthest in the past is replaced. (We will discuss other replacement rules in more detail shortly.) Using this

replacement rule, the contents of the set-associative cache after each reference look like this:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Notice that when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0. The two-way set-associative cache has four misses, one less than the direct-mapped cache.

The fully associative cache has four cache blocks (in a single set); any memory block can be stored in any cache block. The fully associative cache has the best performance, with only three misses:

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

For this series of references, three misses is the best we can do, because three unique block addresses are accessed. Notice that if we had eight blocks in the cache, there would be no replacements in the two-way set-associative cache (check this for yourself), and it would have the same number of misses as the fully associative cache. Similarly, if we had 16 blocks, all three caches would have the identical number of misses. Even this trivial example shows that cache size and associativity are not independent in determining cache performance.

How much of a reduction in the miss rate is achieved by

associativity? Figure 5.16 shows the improvement for a 64 KiB data cache with a 16-word block, and associativity ranging from direct-mapped to eight-way. Going from one-way to two-way associativity decreases the miss rate by about 15%, but there is little further improvement in going to higher associativity.

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

FIGURE 5.16 The data cache miss rates for an organization like the Intrinsity FastMATH processor for SPEC CPU2000 benchmarks with associativity varying from one-way to eight-way. These results for 10 SPEC CPU2000 programs are from Hennessy and Patterson (2003).

Locating a Block in the Cache

Now, let's consider the task of finding a block in a cache that is set associative. Just as in a direct-mapped cache, each block in a set-associative cache includes an address tag that gives the block address. The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor. Figure 5.17 decomposes the address. The index value is used to select the set containing the address of interest, and the tags of all the blocks in the set must be searched. Because speed is of the essence, all the tags in the selected set are searched in parallel. As in a fully associative cache, a sequential search would make the hit time of a set-associative cache too slow.

Tag	Index	Block offset

FIGURE 5.17 The three portions of an address in a set-associative or direct-mapped cache. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in

the selected set. The block offset is the address of the desired data within the block.

If the total cache size is kept the same, increasing the associativity raises the number of blocks per set, which is the number of simultaneous compares needed to perform the search in parallel: each increase by a factor of 2 in associativity doubles the number of blocks per set and halves the number of sets. Accordingly, each factor-of-2 increase in associativity decreases the size of the index by 1 bit and expands the size of the tag by 1 bit. In a fully associative cache, there is effectively only one set, and all the blocks must be checked in parallel. Thus, there is no index, and the entire address, excluding the block offset, is compared against the tag of every block. In other words, we search the full cache without any indexing.

In a direct-mapped cache, only a single comparator is needed, because the entry can be in only one block, and we access the cache simply by indexing. Figure 5.18 shows that in a four-way set-associative cache, four comparators are needed, together with a 4-to-1 multiplexor to choose among the four potential members of the selected set. The cache access consists of indexing the appropriate set and then searching the tags of the set. The costs of an associative cache are the extra comparators and any delay imposed by having to do the compare and select from among the elements of the set.

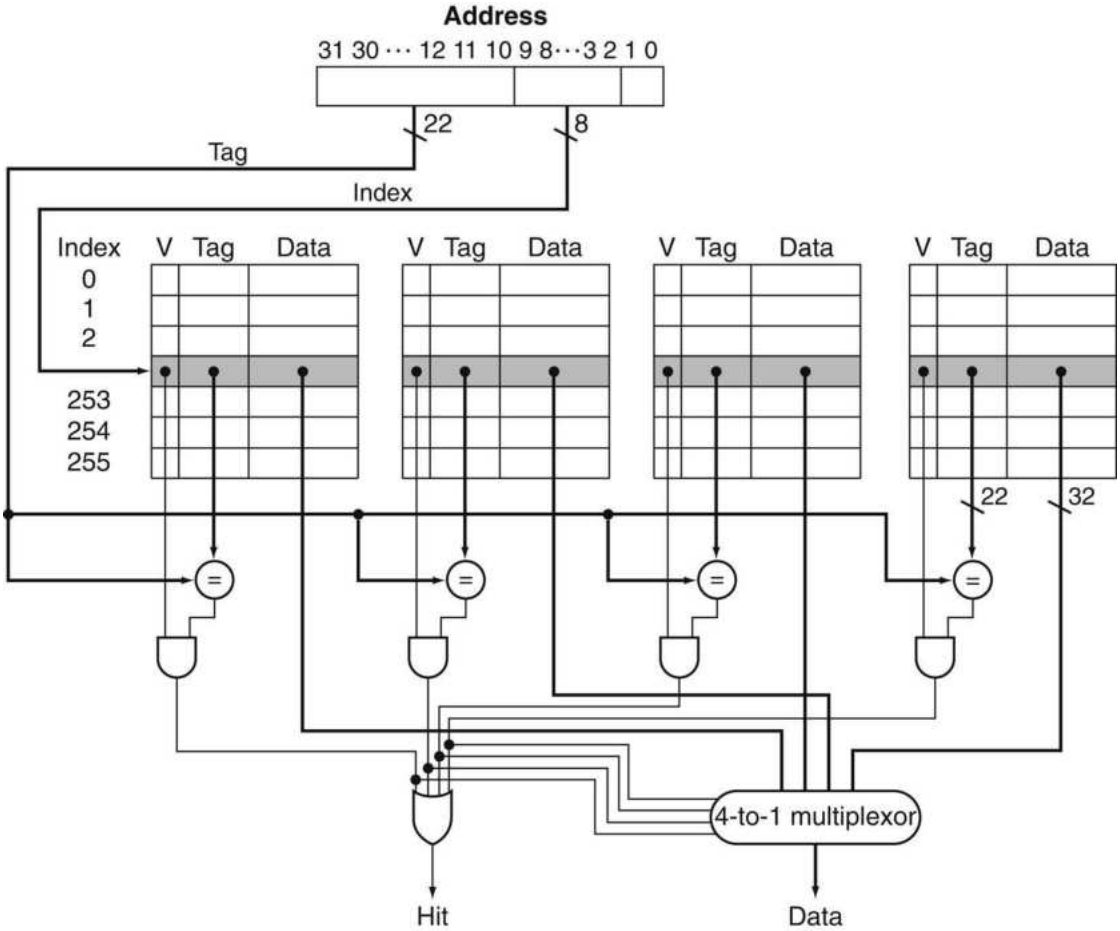


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor.
The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware.

Elaboration

A *Content Addressable Memory* (CAM) is a circuit that combines comparison and storage in a single device. Instead of supplying an address and reading a word like a RAM, you send the data and the CAM looks to see if it has a copy and returns the index of the matching row. CAMs mean that cache designers can afford to implement much higher set associativity than if they needed to build the hardware out of SRAMs and comparators. In 2013, the greater size and power of CAM generally leads to two-way and four-way set associativity being built from standard SRAMs and comparators, with eight-way and above built using CAMs.

Choosing Which Block to Replace

When a miss occurs in a direct-mapped cache, the requested block can go in exactly one position, and the block occupying that position must be replaced. In an associative cache, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set.

The most commonly used scheme is **least recently used (LRU)**, which we used in the previous example. In an LRU scheme, the block replaced is the one that has been unused for the longest time. The set-associative example on page 397 uses LRU, which is why we replaced Memory(0) instead of Memory(6).

least recently used (LRU)

A replacement scheme in which the block replaced is the one that has been unused for the longest time.

LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. As associativity increases, implementing LRU gets harder; in Section 5.8, we will see an alternative scheme

for replacement.

Size of Tags versus Set Associativity

Example

Increasing associativity requires more comparators and more tag bits per cache block. Assuming a cache of 4096 blocks, a four-word block size, and a 64-bit address, find the total number of sets and the total number of tag bits for caches that are direct-mapped, two-way and four-way set associative, and fully associative.

Answer

Since there are 16 ($=2^4$) bytes per block, a 64-bit address yields $64-4=60$ bits to be used for index and tag. The direct-mapped cache has the same number of sets as blocks, and hence 12 bits of index, since $\log_2(4096)=12$; hence, the total number is $(60-12)\times 4096=48\times 4096=197$ K tag bits.

Each degree of associativity decreases the number of sets by a factor of 2 and thus decreases the number of bits used to index the cache by 1 and increases the number of bits in the tag by 1. Thus, for a two-way set-associative cache, there are 2048 sets, and the total number of tag bits is $(60-11)\times 2\times 2048=98\times 2048=401$ Kbits. For a four-way set-associative cache, the total number of sets is 1024, and the total number is $(60-10)\times 4\times 1024=100\times 1024=205$ K tag bits.

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 60 bits, leading to $60\times 4096\times 1=246$ K tag bits.

Reducing the Miss Penalty Using Multilevel Caches

All modern computers make use of caches. To close the gap further between the fast clock rates of modern processors and the increasingly long time required to access DRAMs, most microprocessors support an additional level of caching. This second-level cache is normally on the same chip and is accessed whenever a miss occurs in the primary cache. If the second-level cache contains the desired data, the miss penalty for the first-level

cache will be essentially the access time of the second-level cache, which will be much less than the access time of main memory. If neither the primary nor the secondary cache contains the data, a main memory access is required, and a larger miss penalty is incurred.

How significant is the performance improvement from the use of a secondary cache? The next example shows us.

Performance of Multilevel Caches

Example

Suppose we have a processor with a base CPI of 1.0, assuming all references hit in the primary cache, and a clock rate of 4 GHz. Assume a main memory access time of 100 ns, including all the miss handling. Suppose the miss rate per instruction at the primary cache is 2%. How much faster will the processor be if we add a secondary cache that has a 5- ns access time for either a hit or a miss and is large enough to reduce the miss rate to main memory to 0.5%?

Answer

The miss penalty to main memory is

$$\frac{100\text{ ns}}{0.25\dfrac{\text{ns}}{\text{clock cycle}}} = 400 \text{ clock cycles}$$

The effective CPI with one level of caching is given by

$$\text{Total CPI} = \text{Base CPI} + \text{Memory-stall cycles per instruction}$$

For the processor with one level of caching,

$$\text{Total CPI} = 1.0 + \text{Memory-stall cycles per instruction} = 1.0 + 2\% \times 400 = 9$$

With two levels of caching, a miss in the primary (or first-level) cache can be satisfied either by the secondary cache or by main

memory. The miss penalty for an access to the second-level cache is

$$\frac{5\,\text{ns}}{0.25\,\dfrac{\text{ns}}{\text{clock cycle}}} = 20 \text{ clock cycles}$$

If the miss is satisfied in the secondary cache, then this is the entire miss penalty. If the miss needs to go to main memory, then the total miss penalty is the sum of the secondary cache access time and the main memory access time.

Thus, for a two-level cache, total CPI is the sum of the stall cycles from both levels of cache and the base CPI:

$$\begin{aligned}\text{Total CPI} &= 1 + \text{Primary stalls per instruction} + \text{Secondary stalls per instruction}\\ &= 1 + 2\% \times 20 + 0.5\% \times 400 = 1 + 0.4 + 2.0 = 3.4\end{aligned}$$

Thus, the processor with the secondary cache is faster by

$$\frac{9.0}{3.4} = 2.6$$

Alternatively, we could have computed the stall cycles by summing the stall cycles of those references that hit in the secondary cache ((2%−0.5%)×20 =0.3). Those references that go to main memory, which must include the cost to access the secondary cache as well as the main memory access time, are (0.5%×(20 +400)=2.1). The sum, 1.0 +0.3 +2.1, is again 3.4.

The design considerations for a primary and secondary cache are significantly different, because the presence of the other cache changes the best choice versus a single-level cache. In particular, a two-level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle or fewer pipeline stages, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times.

The effect of these changes on the two caches can be seen by comparing each cache to the optimal design for a single level of

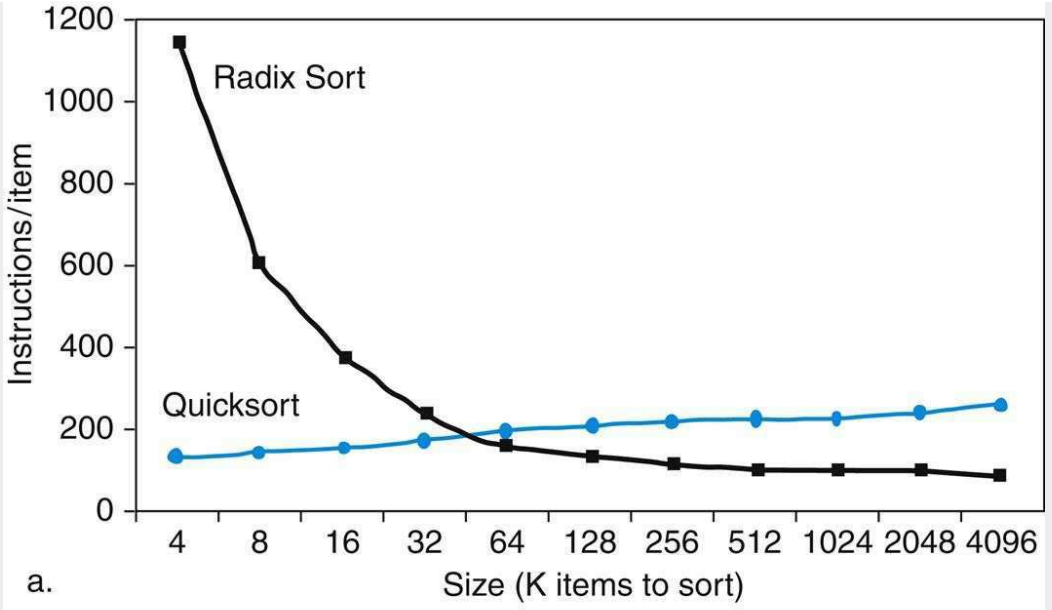
cache. In comparison to a single-level cache, the primary cache of a **multilevel cache** is often smaller. Furthermore, the primary cache may use a smaller block size, to go with the smaller cache size and also to reduce the miss penalty. In comparison, the secondary cache will be much larger than in a single-level cache, since the access time of the secondary cache is less critical. With a larger total size, the secondary cache may use a larger block size than appropriate with a single-level cache. It often uses higher associativity than the primary cache given the focus of reducing miss rates.

multilevel cache

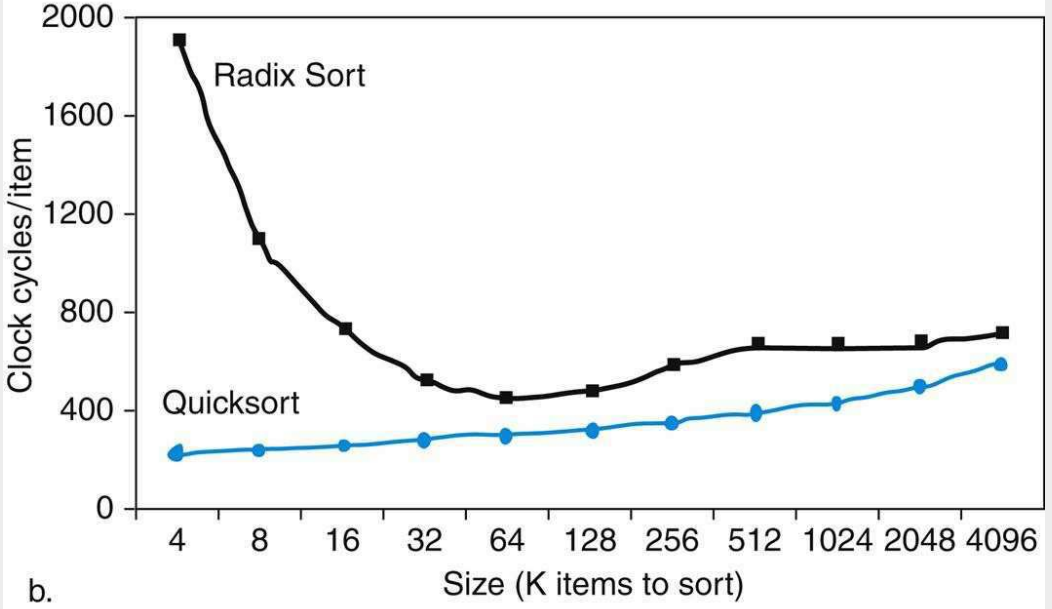
A memory hierarchy with multiple levels of caches, rather than just a cache and main memory.

Understanding Program Performance

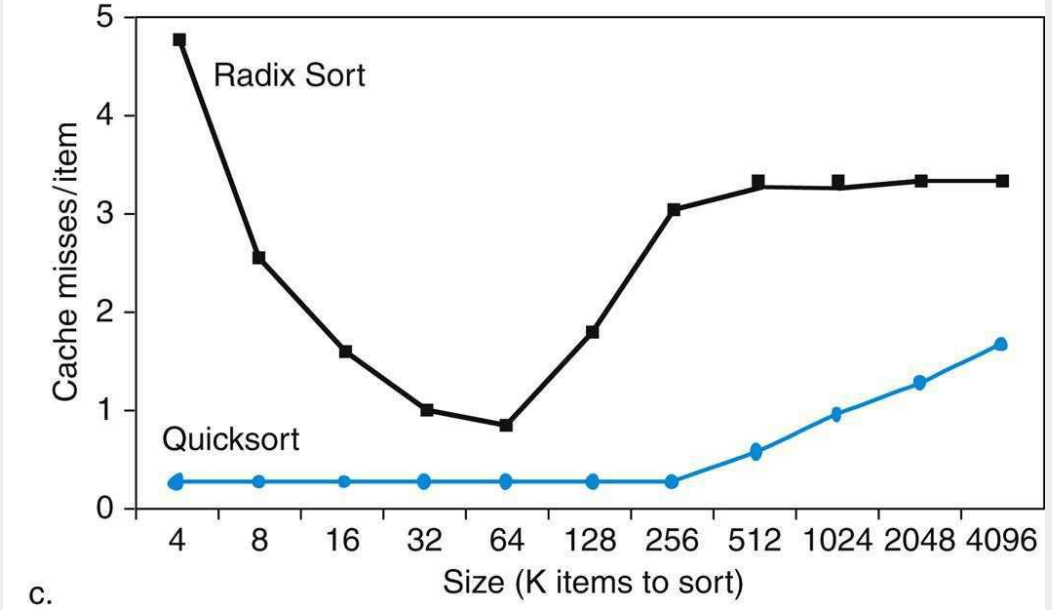
Sorting has been exhaustively analyzed to find better algorithms: Bubble Sort, Quicksort, Radix Sort, and so on. Figure 5.19(a) shows instructions executed by item searched for Radix Sort versus Quicksort. As expected, for large arrays, Radix Sort has an algorithmic advantage over Quicksort in terms of number of operations. Figure 5.19(b) shows time per key instead of instructions executed. We see that the lines start on the same trajectory as in Figure 5.19(a), but then the Radix Sort line diverges as the data to sort increase. What is going on? Figure 5.19(c) answers by looking at the cache misses per item sorted: Quicksort consistently has many fewer misses per item to be sorted.


1200
1000
800
600
400
200
0
Instructions/item
Radix Sort
Quicksort
4
8
16
32
64
128
256
512
1024
2048
4096
Size (K items to sort)

a.


2000
1600
1200
800
400
0
Clock cycles/item
Radix Sort
Quicksort
4
8
16
32
64
128
256
512
1024
2048
4096
Size (K items to sort)

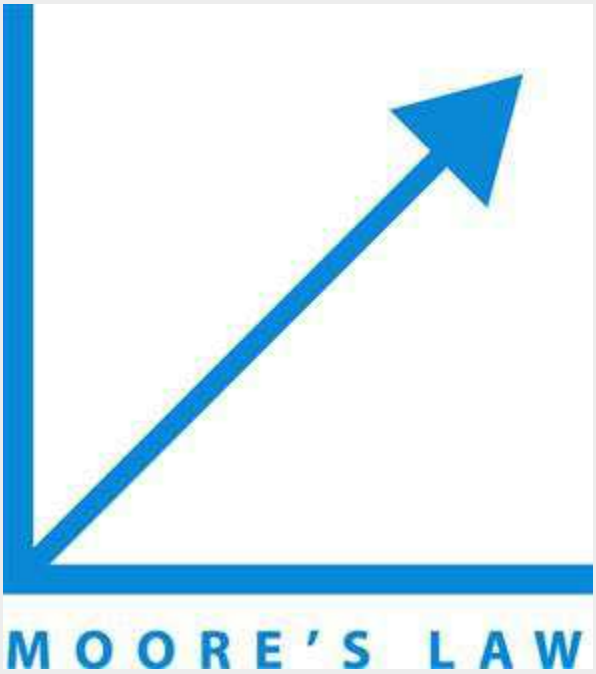
b.


5
4
3
2
1
0
Cache misses/item
Radix Sort
Quicksort
4
8
16
32
64
128
256
512
1024
2048
4096
Size (K items to sort)

c.

FIGURE 5.19 Comparing Quicksort and Radix Sort by (a) instructions executed per item sorted, (b) time per item sorted, and (c) cache misses per item sorted.
These data are from a paper by LaMarca and Ladner [1996]. Due to such results, new versions of Radix Sort have been invented that take memory hierarchy into account, to regain its algorithmic advantages (see Section 5.15). The basic idea of cache optimizations is to use all the data in a block repeatedly before they are replaced on a miss.

Alas, standard algorithmic analysis often ignores the impact of the memory hierarchy. As faster clock rates and **Moore's Law** allow architects to squeeze all the performance out of a stream of instructions, using the memory hierarchy well is vital to high performance. As we said in the introduction, understanding the behavior of the memory hierarchy is critical to understanding the performance of programs on today's computers.



Software Optimization via Blocking

Given the importance of the memory hierarchy to program performance, not surprisingly many software optimizations were invented that can dramatically improve performance by reusing data within the cache and hence lower miss rates due to improved temporal locality.

When dealing with arrays, we can get good performance from the memory system if we store the array in memory so that accesses to the array are sequential in memory. Suppose that we are dealing with multiple arrays, however, with some arrays accessed by rows and some by columns. Storing the arrays row-by-row (called *row major order*) or column-by-column (*column major order*) does not solve the problem because both rows and columns are used in every loop iteration.

Instead of operating on entire rows or columns of an array, *blocked* algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced; that is, improve temporal locality to reduce cache misses.

For example, the inner loops of DGEMM (lines 4 through 9 of Figure 3.22 in Chapter 3) are

```
for (int j = 0; j < n; ++j)
 {
  double cij = C[i+j*n]; /* cij = C[i][j] */
  for( int k = 0; k < n; k++ )
   cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
  C[i+j*n] = cij; /* C[i][j] = cij */
  }
}
```

It reads all N-by-N elements of B, reads the same N elements in what corresponds to one row of A repeatedly, and writes what corresponds to one row of N elements of C. (The comments make the rows and columns of the matrices easier to identify.) Figure 5.20 gives a snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

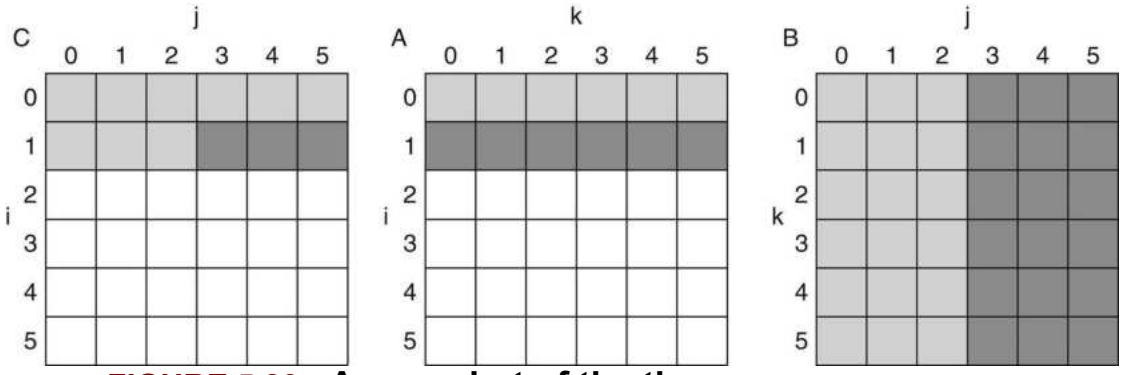


FIGURE 5.20 A snapshot of the three arrays C, A, and B when N = 6 and i = 1.
The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. Compared to Figure 5.22, elements of A and B are read repeatedly to calculate new elements of C. The variables i, j, and k are shown along the rows or columns used to access the arrays.

The number of capacity misses clearly depends on N and the size of the cache. If it can hold all three N-by-N matrices, then all is well, provided there are no cache conflicts. We purposely picked the matrix size to be 32 by 32 in DGEMM for Chapters 3 and 4 so that this would be the case. Each matrix is 32 ×32 =1024 elements and each element is 8 bytes, so the three matrices occupy 24 KiB, which comfortably fit in the 32 KiB data cache of the Intel Core i7 (Sandy Bridge).

If the cache can hold one N-by-N matrix and one row of N, then at least the ith row of A and the array B may stay in the cache. Less than that and misses may occur for both B and C. In the worst case, there would be $2\,N^3+N^2$ memory words accessed for N^3 operations.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix. Hence, we essentially invoke the version of DGEMM from Figure 4.78 in Chapter 4 repeatedly on matrices of size BLOCKSIZE by BLOCKSIZE. BLOCKSIZE is called the *blocking factor*.

Figure 5.21 shows the blocked version of DGEMM. The function do_block is DGEMM from Figure 3.22 with three new parameters si, sj, and sk to specify the starting position of each submatrix of A, B, and C. The two inner loops of the do_block now compute in steps of size BLOCKSIZE rather than the full length of B and C. The gcc

optimizer removes any function call overhead by "inlining" the function; that is, it inserts the code directly to avoid the conventional parameter passing and return address bookkeeping instructions.

```
1  #define BLOCKSIZE 32
2  void do_block (int n, int si, int sj, int sk, double *A, double
3  *B, double *C)
4  {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6        for (int j = sj; j < sj+BLOCKSIZE; ++j)
7           {
8              double cij = C[i+j*n];/* cij = C[i][j] */
9              for( int k = sk; k < sk+BLOCKSIZE; k++ )
10                cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
11             C[i+j*n] = cij;/* C[i][j] = cij */
12            }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16    for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17       for ( int si = 0; si < n; si += BLOCKSIZE )
18          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19             do_block(n, si, sj, sk, A, B, C);
20 }
```

FIGURE 5.21 Cache blocked version of DGEMM in Figure 3.22.
Assume `C` is initialized to zero. The `do_block` function is basically DGEMM from Chapter 3 with new parameters to specify the starting positions of the submatrices of `BLOCKSIZE`. The gcc optimizer can remove the function overhead instructions by inlining the `do_block` function.

Figure 5.22 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2\ N^3/\texttt{BLOCKSIZE} + N^2$. This total is an improvement by about a factor of `BLOCKSIZE`. Hence, blocking exploits a combination of spatial and temporal locality, since `A` benefits from spatial locality and `B` benefits from temporal locality.

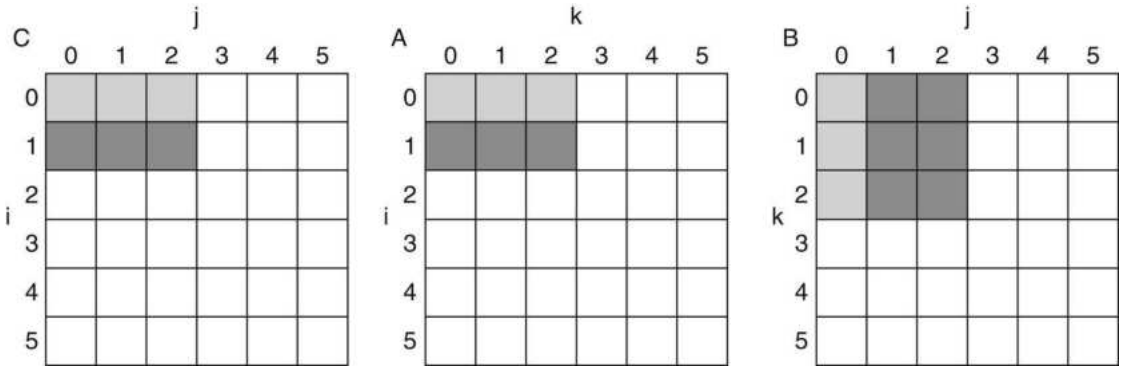


FIGURE 5.22 The age of accesses to the arrays C, A, and B when *BLOCKSIZE* = 3.
Note that, in contrast to Figure 5.20, fewer elements are accessed.

Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size, such that the block can be held in registers, we can minimize the number of loads and stores in the program, which again improves performance.

Figure 5.23 shows the impact of cache blocking on the performance of the unoptimized DGEMM as we increase the matrix size beyond where all three matrices fit in the cache. The unoptimized performance is halved for the largest matrix. The cache-blocked version is less than 10% slower even at matrices that are 960 ×960, or 900 times larger than the 32 ×32 matrices in Chapters 3 and 4.

Elaboration

Multilevel caches create many complications. First, there are now several different types of misses and corresponding miss rates. In the example on pages 402–403, we saw the primary cache miss rate and the **global miss rate**—the fraction of references that missed in all cache levels. There is also a miss rate for the secondary cache, which is the ratio of all misses in the secondary cache divided by the number of accesses to it. This miss rate is called the **local miss rate** of the secondary cache. Because the primary cache filters accesses, especially those with good spatial and temporal locality, the local miss rate of the secondary cache is much higher than the global miss rate. For the example on pages 402–403, we can compute the local miss rate of the secondary cache as

0.5%/2%=25%! Luckily, the global miss rate dictates how often we must access the main memory.

global miss rate

The fraction of references that miss in all levels of a multilevel cache.

local miss rate

The fraction of references to one level of a cache that miss; used in multilevel hierarchies.

Elaboration

With out-of-order processors (see Chapter 4), performance is more complex, since they execute instructions during the miss penalty. Instead of instruction miss rates and data miss rates, we use misses per instruction, and this formula:

$$\frac{\text{Memory} - \text{stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

There is no general way to calculate overlapped miss latency, so evaluations of memory hierarchies for out-of-order processors inevitably require simulation of the processor and the memory hierarchy. Only by seeing the execution of the processor during each miss can we see if the processor stalls waiting for data or simply finds other work to do. A guideline is that the processor often hides the miss penalty for an L1 cache miss that hits in the L2 cache, but it rarely hides a miss to the L2 cache.

Elaboration

The performance challenge for algorithms is that the memory hierarchy varies between different implementations of the same architecture in cache size, associativity, block size, and number of caches. To cope with such variability, some recent numerical libraries parameterize their algorithms and then search the parameter space at runtime to find the best combination for a

particular computer. This approach is called *autotuning*.

Check Yourself

Which of the following is generally true about a design with multiple levels of caches?

1. First-level caches are more concerned about hit time, and second-level caches are more concerned about miss rate.
2. First-level caches are more concerned about miss rate, and second-level caches are more concerned about hit time.

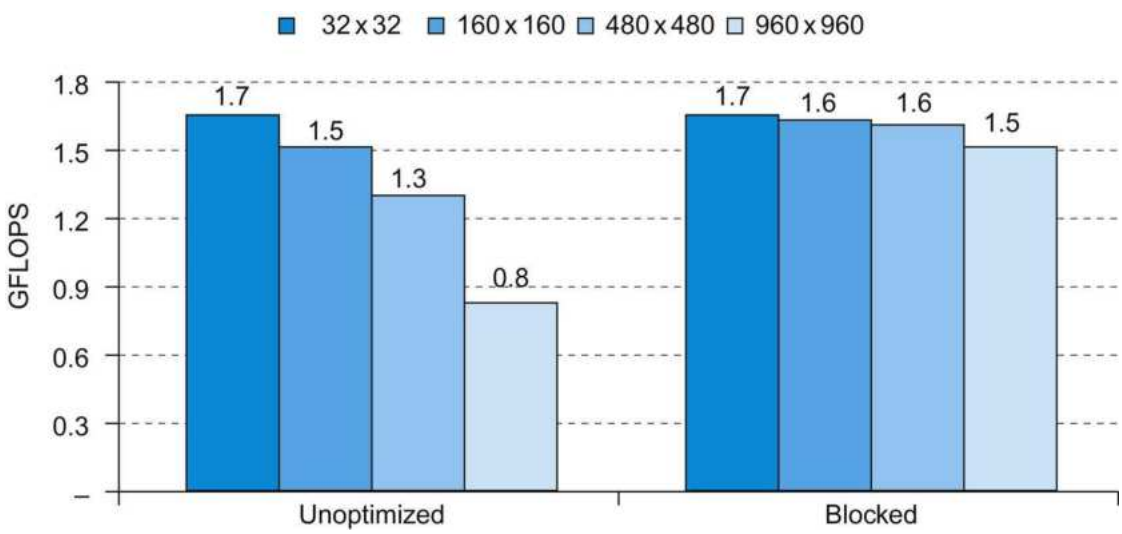


FIGURE 5.23 Performance of unoptimized DGEMM (Figure 3.22) versus cache blocked DGEMM (Figure 5.21) as the matrix dimension varies from 32 ×32 (where all three matrices fit in the cache) to 960 ×960.

Summary

In this section, we focused on four topics: cache performance, using associativity to reduce miss rates, the use of multilevel cache hierarchies to reduce miss penalties, and software optimizations to improve effectiveness of caches.

The memory system has a significant effect on program execution time. The number of memory-stall cycles depends on both the miss rate and the miss penalty. The challenge, as we will see in Section 5.8, is to reduce one of these factors without significantly affecting

other critical factors in the memory hierarchy.

To reduce the miss rate, we examined the use of associative placement schemes. Such schemes can reduce the miss rate of a cache by allowing more flexible placement of blocks within the cache. Fully associative schemes allow blocks to be placed anywhere, but also require that every block in the cache be searched to satisfy a request. The higher costs make large fully associative caches impractical. Set-associative caches are a practical alternative, since we need only search among the elements of a unique set that is chosen by indexing. Set-associative caches have higher miss rates but are faster to access. The amount of associativity that yields the best performance depends on both the technology and the details of the implementation.

We looked at multilevel caches as a technique to reduce the miss penalty by allowing a larger secondary cache to handle misses to the primary cache. Second-level caches have become commonplace as designers find that limited silicon and the goals of high clock rates prevent primary caches from becoming large. The secondary cache, which is often 10 or more times larger than the primary cache, handles many accesses that miss in the primary cache. In such cases, the miss penalty is that of the access time to the secondary cache (typically <10 processor cycles) versus the access time to memory (typically >100 processor cycles). As with associativity, the design tradeoffs between the size of the secondary cache and its access time depend on a number of aspects of the implementation.

Finally, given the importance of the memory hierarchy in performance, we looked at how to change algorithms to improve cache behavior, with blocking being an important technique when dealing with large arrays.

5.5 Dependable Memory Hierarchy

Implicit in all the prior discussion is that the memory hierarchy doesn't forget. Fast but undependable is not very attractive. As we learned in Chapter 1, the one great idea for **dependability** is redundancy. In this section we'll first go over the terms to define terms and measures associated with failure, and then show how redundancy can make nearly unforgettable memories.

Defining Failure

We start with an assumption that you have a specification of proper service. Users can then see a system alternating between two states of delivered service with respect to the service specification:

1. *Service accomplishment*, where the service is delivered as specified
2. *Service interruption*, where the delivered service is different from the specified service

Transitions from state 1 to state 2 are caused by *failures*, and transitions from state 2 to state 1 are called *restorations*. Failures can be permanent or intermittent. The latter is the more difficult case; it is harder to diagnose the problem when a system oscillates between the two states. Permanent failures are far easier to diagnose.

This definition leads to two related terms: reliability and availability.

Reliability is a measure of the continuous service accomplishment —or, equivalently, of the time to failure—from a reference point. Hence, *mean time to failure* (MTTF) is a reliability measure. A related term is *annual failure rate* (AFR), which is just the percentage of devices that would be expected to fail in a year for a given MTTF. When MTTF gets large it can be misleading, while AFR leads to better intuition.

MTTF vs. AFR of Disks

Example

Some disks today are quoted to have a 1,000,000-hour MTTF. As 1,000,000 hours is 1,000,000/(365 ×24)=114 years, it would seem like they practically never fail. Warehouse-scale computers that run Internet services such as Search might have 50,000 servers. Assume

each server has two disks. Use AFR to calculate how many disks we would expect to fail per year.

Answer

One year is 365 ×24 =8760 hours. A 1,000,000-hour MTTF means an AFR of 8760/1,000,000 =0.876%. With 100,000 disks, we would expect 876 disks to fail per year, or on average more than two disk failures per day!

Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF +MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. *Availability* is then a measure of service accomplishment with respect to the alternation between the two states of accomplishment and interruption. Availability is statistically quantified as

$$\text{Availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are actually quantifiable measures, rather than just synonyms for dependability. Shrinking MTTR can help availability as much as increasing MTTF. For example, tools for fault detection, diagnosis, and repair can help reduce the time to repair faults and thereby improve availability.

We want availability to be very high. One shorthand is to quote the number of "nines of availability" per year. For instance, a very good Internet service today offers 4 or 5 nines of availability. Given 365 days per year, which is 365 ×24 ×60 =526,000 minutes, then the shorthand is decoded as follows:

One nine: 90% =>36.5 days of repair/year
Two nines: 99% =>3.65 days of repair/year
Three nines: 99.9% =>526 minutes of repair/year
Four nines: 99.99% =>52.6 minutes of repair/year
Five nines: 99.999% =>5.26 minutes of repair/year

and so on.

To increase MTTF, you can improve the quality of the components or design systems to continue operation in the

presence of components that have failed. Hence, failure needs to be defined with respect to a context, as failure of a component may not lead to a failure of the system. To make this distinction clear, the term *fault* is used to mean failure of a component. Here are three ways to improve MTTF:

1. *Fault avoidance:* Preventing fault occurrence by construction.
2. *Fault tolerance:* Using redundancy to allow the service to comply with the service specification despite faults occurring.
3. *Fault forecasting:* **Predicting** the presence and creation of faults, allowing the component to be replaced *before* it fails.

PREDICTION

The Hamming Single Error Correcting, Double Error Detecting Code (SEC/DED)

Richard Hamming invented a popular redundancy scheme for memory, for which he received the Turing Award in 1968. To invent redundant codes, it is helpful to talk about how "close" correct bit patterns can be. What we call the *Hamming distance* is just

the minimum number of bits that are different between any two correct bit patterns. For example, the distance between 011011 and 001111 is two. What happens if the minimum distance between members of a code is two, and we get a one-bit error? It will turn a valid pattern in a code to an invalid one. Thus, if we can detect whether members of a code are accurate or not, we can detect single bit errors, and can say we have a single bit **error detection code**.

error detection code

A code that enables the detection of an error in data, but not the precise location and, hence, correction of the error.

Hamming used a *parity code* for error detection. In a parity code, the number of 1 s in a word is counted; the word has odd parity if the number of 1 s is odd and even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for even). That is, the parity of the N+1 bit word should always be even. Then, when the word is read out, the parity bit is read and checked. If the parity of the memory word and the stored parity bit do not match, an error has occurred.

Example

Calculate the parity of a byte with the value 31_{ten} and show the pattern stored to memory. Assume the parity bit is on the right. Suppose the most significant bit was inverted in memory, and then you read it back. Did you detect the error? What happens if the two most significant bits are inverted?

Answer

31_{ten} is 00011111_{two}, which has five 1 s. To make parity even, we need to write a 1 in the parity bit, or 000111111_{two}. If the most significant bit is inverted when we read it back, we would see 100111111_{two} which has seven 1 s. Since we expect even parity and calculated odd parity, we would signal an error. If the *two* most significant bits are inverted, we would see 110111111_{two} which has eight 1 s or even parity, and we would *not* signal an error.

If there are 2 bits of error, then a 1-bit parity scheme will not detect any errors, since the parity will match the data with two errors. (Actually, a 1-bit parity scheme can detect any odd number of errors; however, the probability of having three errors is much lower than the probability of having two, so, in practice, a 1-bit parity code is limited to detecting a single bit of error.)

Of course, a parity code cannot correct errors, which Hamming wanted to do as well as detect them. If we used a code that had a minimum distance of 3, then any single bit error would be closer to the correct pattern than to any other valid pattern. He came up with an easy to understand mapping of data into a distance 3 code that we call *Hamming Error Correction Code* (ECC) in his honor. We use extra parity bits to allow the position identification of a single error. Here are the steps to calculate Hamming ECC

1. Start numbering bits from 1 on the left, contrary to the traditional numbering of the rightmost bit being 0.
2. Mark all bit positions that are powers of 2 as parity bits (positions 1, 2, 4, 8, 16, …).
3. All other bit positions are used for data bits (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …).
4. The position of parity bit determines sequence of data bits that it checks (Figure 5.24 shows this coverage graphically) is:

- Bit 1 (0001_{two}) checks bits (1,3,5,7,9,11,...), which are bits where rightmost bit of address is 1 (0001_{two}, 0011_{two}, 0101_{two}, 0111_{two}, 1001_{two}, 1011_{two},…).
- Bit 2 (0010_{two}) checks bits (2,3,6,7,10,11,14,15,…), which are the bits where the second bit to the right in the address is 1.
- Bit 4 (0100_{two}) checks bits (4–7, 12–15, 20–23,…), which are the bits where the third bit to the right in the address is 1.
- Bit 8 (1000_{two}) checks bits (8–15, 24–31, 40–47,...), which are the bits where the fourth bit to the right in the address is 1.

Note that each data bit is covered by two or more parity bits.

5. Set parity bits to create even parity for each group.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

FIGURE 5.24 Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits.

In what seems like a magic trick, you can determine whether bits are incorrect by looking at the parity bits. Using the 12 bit code in Figure 5.24, if the value of the four parity calculations (p8,p4,p2,p1) was 0000, then there was no error. However, if the pattern was, say, 1010, which is 10_{ten}, then Hamming ECC tells us that bit 10 (d6) is an error. Since the number is binary, we can correct the error just by inverting the value of bit 10.

Example

Assume one byte data value is 10011010_{two}. First show the Hamming ECC code for that byte, and then invert bit 10 and show that the ECC code finds and corrects the single bit error.

Leaving spaces for the parity bits, the 12 bit pattern is__ 1_ 0 0 1_ 1 0 1 0.

Answer

Position 1 checks bits 1,3,5,7,9, and 11, which we highlight:__ **1**_ **0** 0 **1**_ **1** 0 **1** 0. To make the group even parity, we should set bit 1 to 0.

Position 2 checks bits 2,3,6,7,10,11, which is 0_ **1**_ 0 **0** **1**_ 1 **0** **1** 0 or odd parity, so we set position 2 to a 1.

Position 4 checks bits 4,5,6,7,12, which is 0 1 1_ **0** **0** **1**_ 1 0 1, so we set it to a 1.

Position 8 checks bits 8,9,10,11,12, which is 0 1 1 1 0 0 1_ **1** **0** **1** **0**, so we set it to a 0.

The final code word is 011100101010. Inverting bit 10 changes it to 011100101**1**10.

Parity bit 1 is 0 (**0**1**1**1**0**0**1**0**1**1**1**0 is four 1 s, so even parity; this group is OK).

Parity bit 2 is 1 (0**1**1**1**0**0**1**0**1**1**1**0 is five 1 s, so odd parity; there is an error somewhere).

Parity bit 4 is 1 (011**100**1011**1**0 is two 1 s, so even parity; this group is OK).

Parity bit 8 is 1 (0111001**01110** is three 1 s, so odd parity; there is an error somewhere).

Parity bits 2 and 8 are incorrect. As 2 +8 =10, bit 10 must be wrong. Hence, we can correct the error by inverting bit 10: 011100101**0**10. Voila!

Hamming did not stop at single bit error correction code. At the cost of one more bit, we can make the minimum Hamming distance in a code be 4. This means we can correct single bit errors *and detect double bit errors*. The idea is to add a parity bit that is calculated over the whole word. Let's use a 4-bit data word as an example, which would only need 7 bits for single bit error detection. Hamming parity bits H (p1 p2 p3) are computed (even parity as usual) plus the even parity over the entire word, p4:

1 2 3 4 5 6 7 **8**

p_1 p_2 d_1 p_3 d_2 d_3 d_4 $\mathbf{p}_4$

Then the algorithm to correct one error and detect two is just to calculate parity over the ECC groups (H) as before plus one more over the whole group (p_4). There are four cases:

1. H is even and p_4 is even, so no error occurred.
2. H is odd and p_4 is odd, so a correctable single error occurred. (p_4 should calculate odd parity if one error occurred.)
3. H is even and p_4 is odd, a single error occurred in p_4 bit, not in the rest of the word, so correct the p_4 bit.
4. H is odd and p_4 is even, a double error occurred. (p_4 should calculate even parity if two errors occurred.)

Single Error Correcting/Double Error Detecting (SEC/DED) is common in memory for servers today. Conveniently, 8-byte data blocks can get SEC/DED with just one more byte, which is why many DIMMs are 72 bits wide.

Elaboration

To calculate how many bits are needed for SEC, let p be total number of parity bits and d number of data bits in $p+d$ bit word. If p error correction bits are to point to error bit ($p+d$ cases) plus one case to indicate that no error exists, we need:

$$2^p \geq p + d + 1 \text{ bits, and thus } p \geq \log(p + d + 1).$$

For example, for 8 bits data means $d = 8$ and $2^p \geq p + 8 + 1$, so $p = 4$. Similarly, $p = 5$ for 16 bits of data, 6 for 32 bits, 7 for 64 bits, and so on.

Elaboration

In very large systems, the possibility of multiple errors as well as complete failure of a single wide memory chip becomes significant. IBM introduced *chipkill* to solve this problem, and many big systems use this technology. (Intel calls their version SDDC.)

Similar in nature to the RAID approach used for disks (see Section 5.11), Chipkill distributes the data and ECC information, so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Assuming a 10,000-processor cluster with 4 GiB per processor, IBM calculated the following rates of *unrecoverable* memory errors in 3 years of operation:

- Parity only—about 90,000, or one unrecoverable (or undetected) failure every 17 minutes.
- SEC/DED only—about 3500, or about one undetected or unrecoverable failure every 7.5 hours.
- Chipkill—6, or about one undetected or unrecoverable failure every 2 months.

  Hence, Chipkill is a requirement for warehouse-scale computers.

Elaboration

While single or double bit errors are typical for memory systems, networks can have bursts of bit errors. One solution is called *Cyclic Redundancy Check*. For a block of k bits, a transmitter generates an

n-k bit frame check sequence. It transmits *n* bits exactly divisible by some number. The receiver divides the frame by that number. If there is no remainder, it assumes there is no error. If there is, the receiver rejects the message, and asks the transmitter to send again. As you might guess from Chapter 3, it is easy to calculate division for some binary numbers with a shift register, which made CRC codes popular even when hardware was more precious. Going even further, Reed-Solomon codes use Galois fields to *correct* multibit transmission errors, but now data are considered coefficients of a polynomial and the code space is values of a polynomial. The Reed-Solomon calculation is considerably more complicated than binary division!

5.6 Virtual Machines

Virtual machines (VM) were first developed in the mid-1960s, and they have remained an important part of mainframe computing over the years. Although largely ignored in the single-user PC era in the 1980s and 1990s, they have recently gained popularity due to

- The increasing importance of isolation and security in modern systems
- The failures in security and reliability of standard operating systems
- The sharing of a single computer among many unrelated users, in particular for Cloud computing
- The dramatic increases in raw speed of processors over the decades, which makes the overhead of VMs more acceptable

The broadest definition of VMs includes basically all emulation methods that provide a standard software interface, such as the Java VM. In this section, we are interested in VMs that provide a complete system-level environment at the binary *instruction set architecture* (ISA) level. Although some VMs run different ISAs in the VM from the native hardware, we assume they always match the hardware. Such VMs are called (Operating) *System Virtual Machines*. IBM VM/370, VirtualBox, VMware ESX Server, and Xen are examples.

System virtual machines present the illusion that the users have an entire computer to themselves, including a copy of the operating system. A single computer runs multiple VMs and can support a

number of different *operating systems* (OSes). On a conventional platform, a single OS "owns" all the hardware resources, but with a VM, multiple OSes all share the hardware resources.

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor*; the VMM is the heart of virtual machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. The VMM determines how to map virtual resources to physical resources: a physical resource may be time-shared, partitioned, or even emulated in software. The VMM is much smaller than a traditional OS; the isolation portion of a VMM is perhaps only 10,000 lines of code.

Although our interest here is in VMs for improving protection, VMs provide two other benefits that are commercially significant:

1. *Managing software*. VMs provide an abstraction that can run the complete software stack, even including old operating systems like DOS. A typical deployment might be some VMs running legacy OSes, many running the current stable OS release, and a few testing the next OS release.
2. *Managing hardware*. One reason for multiple servers is to have each application running with the compatible version of the operating system on separate computers, as this separation can improve dependability. VMs allow these separate software stacks to run independently yet share hardware, thereby consolidating the number of servers. Another example is that some VMMs support migration of a running VM to a different computer, either to balance load or to evacuate from failing hardware.

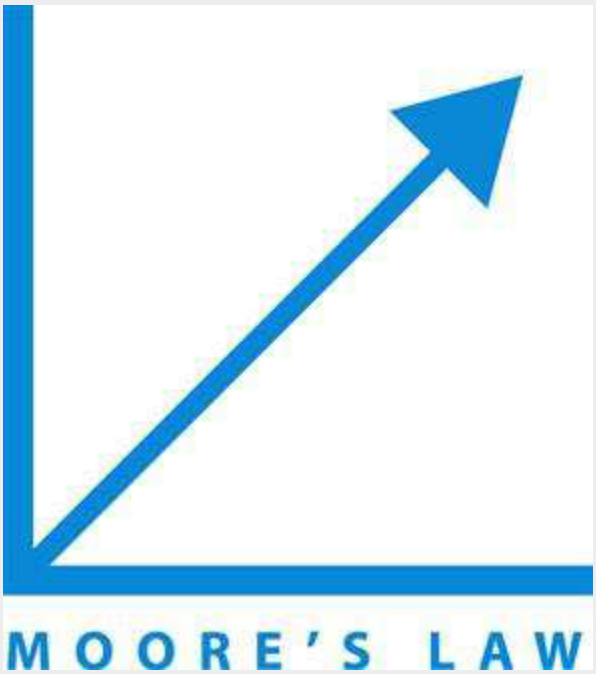
Hardware/Software Interface

Amazon Web Services (AWS) uses the virtual machines in its Cloud computing offering EC2 for five reasons:

1. It allows AWS to protect users from each other while sharing the same server.
2. It simplifies software distribution within a warehouse-scale computer. A customer installs a virtual machine image configured with the appropriate software, and AWS distributes it to all the instances a customer wants to use.
3. Customers (and AWS) can reliably "kill" a VM to control

resource usage when customers complete their work.

4. Virtual machines hide the identity of the hardware on which the customer is running, which means AWS can keep using old servers *and* introduce new, more efficient servers. The customer expects performance for instances to match their ratings in "EC2 Compute Units," which AWS defines: to "provide the equivalent CPU capacity of a 1.0–1.2 GHz 2007 AMD Opteron or 2007 Intel Xeon processor." Thanks to **Moore's Law**, newer servers clearly offer more EC2 Compute Units than older ones, but AWS can keep renting old servers as long as they are economical.



5. Virtual machine monitors can control the rate that a VM uses the processor, the network, and disk space, which allows AWS to offer many price points of instances of different types running on the same underlying servers. For example, in 2012 AWS offered 14 instance types, from small standard instances at $0.08 per hour to high I/O quadruple extra-large instances at $3.10 per hour.

In general, the cost of processor virtualization depends on the workload. User-level processor-bound programs have zero virtualization overhead, because the OS is rarely invoked, so everything runs at native speeds. I/O-intensive workloads are generally also OS-intensive, executing many system calls and

privileged instructions that can result in high virtualization overhead. On the other hand, if the I/O-intensive workload is also *I/O-bound*, the cost of processor virtualization can be completely hidden, since the processor is often idle waiting for I/O.

The overhead is determined by both the number of instructions that must be emulated by the VMM and by how much time each takes to emulate. Hence, when the guest VMs run the same ISA as the host, as we assume here, the goal of the architecture and the VMM is to run almost all instructions directly on the native hardware.

Requirements of a Virtual Machine Monitor

What must a VM monitor do? It presents a software interface to guest software, it must isolate the state of guests from each other, and it must protect itself from guest software (including guest OSes). The qualitative requirements are:

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to change the allocation of real system resources directly.

To "virtualize" the processor, the VMM must control just about everything—access to privileged state, I/O, exceptions, and interrupts—even though the guest VM and OS presently running are temporarily using them.

For example, in the case of a timer interrupt, the VMM would suspend the currently running guest VM, save its state, handle the interrupt, determine which guest VM to run next, and then load its state. Guest VMs that rely on a timer interrupt are provided with a virtual timer and an emulated timer interrupt by the VMM.

To be in charge, the VMM must be at a higher privilege level than the guest VM, which generally runs in user mode; this also ensures that the execution of any privileged instruction will be handled by the VMM. The basic system requirements to support VMMs are:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode; all system

resources must be controllable just via these instructions.

(Lack of) Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it's relatively easy to reduce both the number of instructions that must be executed by a VMM and improve their emulation speed. An architecture that allows the VM to execute directly on the hardware earns the title *virtualizable*, and the IBM 370 and the RISC-V architectures proudly bear that label.

Alas, since VMs have been considered for PC and server applications only fairly recently, most instruction sets were created without virtualization in mind. These culprits include x86 and most RISC architectures, including ARMv7 and MIPS.

Because the VMM must ensure that the guest system only interacts with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction—for example, reading or writing a status bit that enables interrupts—it will trap to the VMM. The VMM can then affect the appropriate changes to corresponding real resources.

Hence, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information, as the guest OS expects.

In the absence of such support, other measures must be taken. A VMM must take special precautions to locate all problematic instructions and ensure that they behave correctly when executed by a guest OS, thereby increasing the complexity of the VMM and reducing the performance of running the VM.

Protection and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular.

For example, the x86 instruction POPF loads the flag registers

from the top of the stack in memory. One of the flags is the *Interrupt Enable* (IE) flag. If you run the POPF instruction in user mode, rather than trap it, it simply changes all the flags except IE. In system mode, it does change the IE. Since a guest OS runs in user mode inside a VM, this is a problem, as it expects to see a changed IE.

Historically, IBM mainframe hardware and VMM took three steps to improve the performance of virtual machines:

1. Reduce the cost of processor virtualization.
2. Reduce interrupt overhead cost due to the virtualization.
3. Reduce interrupt cost by steering interrupts to the proper VM without invoking VMM.

AMD and Intel tried to address the first point in 2006 by reducing the cost of processor virtualization. It will be interesting to see how many generations of architecture and VMM modifications it will take to address all three points, and how long before virtual machines of the 21st century for x86 will be as efficient as the IBM mainframes and VMMs of the 1970s.

Elaboration

RISC-V traps all privileged instructions when running in user mode, so it supports *classical virtualization*, wherein the guest OS runs in user mode and the VMM runs in supervisor mode.

5.7 Virtual Memory

> *… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.*
>
> *Kilburn et al., One-level storage system, 1962*

In earlier sections, we saw how caches provided fast access to recently-used portions of a program's code and data. Similarly, the main memory can act as a "cache" for the secondary storage, traditionally implemented with magnetic disks. This technique is called **virtual memory**. Historically, there were two major motivations for virtual memory: to allow efficient and safe sharing

of memory among several programs, such as for the memory needed by multiple virtual machines for Cloud computing, and to remove the programming burdens of a small, limited amount of main memory. Five decades after its invention, it's the former reason that reigns today.

virtual memory

A technique that uses main memory as a "cache" for secondary storage.

Of course, to allow multiple virtual machines to share the same memory, we must be able to protect the virtual machines from each other, ensuring that a program can just read and write the portions of main memory that have been assigned to it. Main memory need contain only the active portions of the many virtual machines, just as a cache contains only the active portion of one program. Thus, the principle of locality enables virtual memory as well as caches, and virtual memory allows us to share the processor efficiently as well as the main memory.

We cannot know which virtual machines will share the memory with other virtual machines when we compile them. In fact, the virtual machines sharing the memory change dynamically while they are running. Because of this dynamic interaction, we would like to compile each program into its own *address space*—a separate range of memory locations accessible only to this program. Virtual memory implements the translation of a program's address space to **physical addresses**. This translation process enforces **protection** of a program's address space from other virtual machines.

physical address

An address in main memory.

protection

A set of mechanisms for ensuring that multiple processes sharing the processor, memory, or I/O devices cannot interfere, intentionally or unintentionally, with one another by reading or writing each other's data. These mechanisms also isolate the

operating system from a user process.

The second motivation for virtual memory is to allow a single-user program to exceed the size of primary memory. Formerly, if a program became too large for memory, it was up to the programmer to make it fit. Programmers divided programs into pieces and then identified the pieces that were mutually exclusive. These *overlays* were loaded or unloaded under user program control during execution, with the programmer ensuring that the program at no time tried to access an overlay that was not loaded and that the overlays loaded never exceeded the total size of the memory. Overlays were traditionally organized as modules, each containing both code and data. Calls between procedures in different modules would lead to overlaying of one module with another.

As you can well imagine, this responsibility was a substantial burden on programmers. Virtual memory, which was invented to relieve programmers of this difficulty, automatically manages the two levels of the memory hierarchy represented by main memory (sometimes called *physical memory* to distinguish it from virtual memory) and secondary storage.

Although the concepts at work in virtual memory and in caches are the same, their differing historical roots have led to the use of different terminology. A virtual memory block is called a *page*, and a virtual memory miss is called a **page fault**. With virtual memory, the processor produces a **virtual address**, which is translated by a combination of hardware and software to a *physical address*, which in turn can be used to access main memory. Figure 5.25 shows the virtually addressed memory with pages mapped to main memory. This process is called *address mapping* or **address translation**. Today, the two memory hierarchy levels controlled by virtual memory are usually DRAMs and flash memory in personal mobile devices and DRAMs and magnetic disks in servers (see Section 5.2). If we return to our library analogy, we can think of a virtual address as the title of a book and a physical address as the location of that book in the library, such as might be given by the Library of Congress call number.

page fault

An event that occurs when an accessed page is not present in main memory.

virtual address

An address that corresponds to a location in virtual space and is translated by address mapping to a physical address when memory is accessed.

address translation

Also called **address mapping**. The process by which a virtual address is mapped to an address used to access memory.

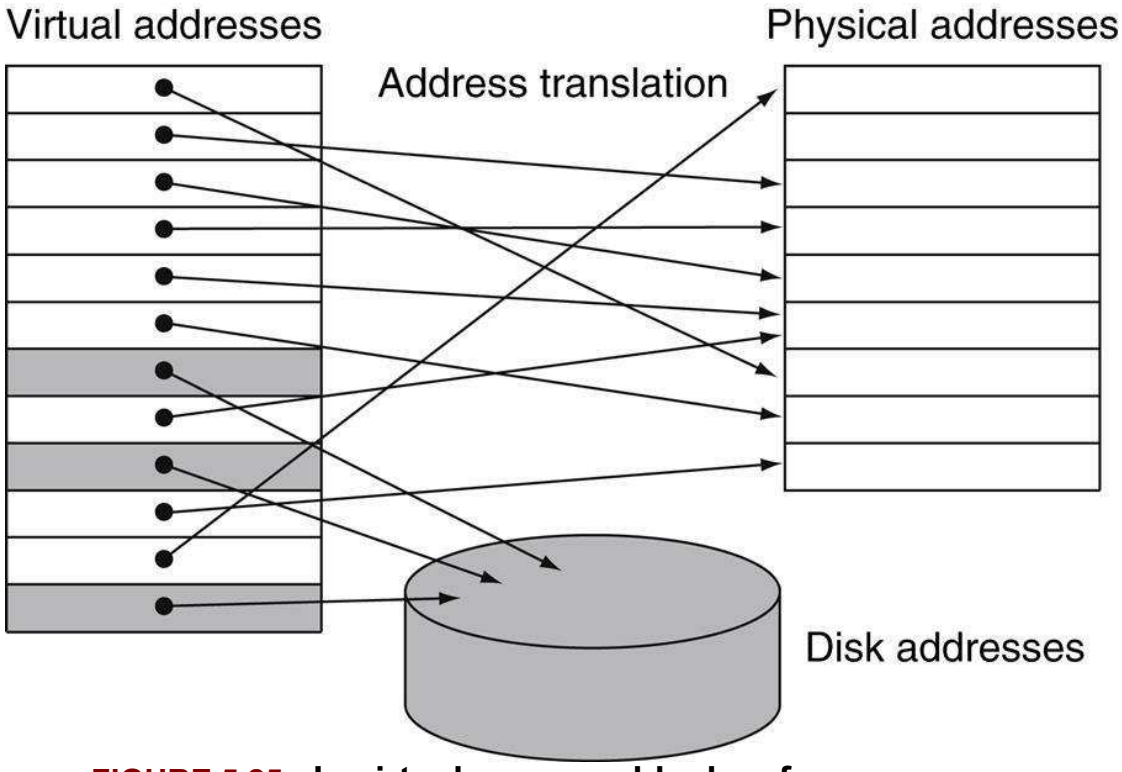


FIGURE 5.25 In virtual memory, blocks of memory (called *pages*) are mapped from one set of addresses (called *virtual addresses*) to another set (called *physical addresses*).
The processor generates virtual addresses while the memory is accessed using physical addresses. Both the virtual memory and the physical memory are broken into pages, so that a virtual page is mapped to a physical page. Of course, it is also possible for a

virtual page to be absent from main memory and not be mapped to a physical address; in that case, the page resides on disk. Physical pages can be shared by having two virtual addresses point to the same physical address. This capability is used to allow two different programs to share data or code.

Virtual memory also simplifies loading the program for execution by providing *relocation*. Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory. This relocation allows us to load the program anywhere in main memory. Furthermore, all virtual memory systems in use today relocate the program as a set of fixed-size blocks (pages), thereby eliminating the need to find a contiguous block of memory to allocate to a program; instead, the operating system needs only to find enough pages in main memory.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 5.26 shows the translation of the virtual page number to a *physical page number*. While RISC-V has a 64-bit address, the upper 16 bits are not used, so the address to be mapped is 48 bits. This figure assumes the physical memory is 1 TiB, or 2^{40} bytes, which needs a 40-bit address. The physical page number constitutes the upper portion of the physical address, while the page offset, which is not changed, constitutes the lower portion. The number of bits in the page offset field determines the page size. The number of pages addressable with the virtual address can be different than the number of pages addressable with the physical address. Having a larger number of virtual pages than physical pages is the basis for the illusion of an essentially unbounded amount of virtual memory.

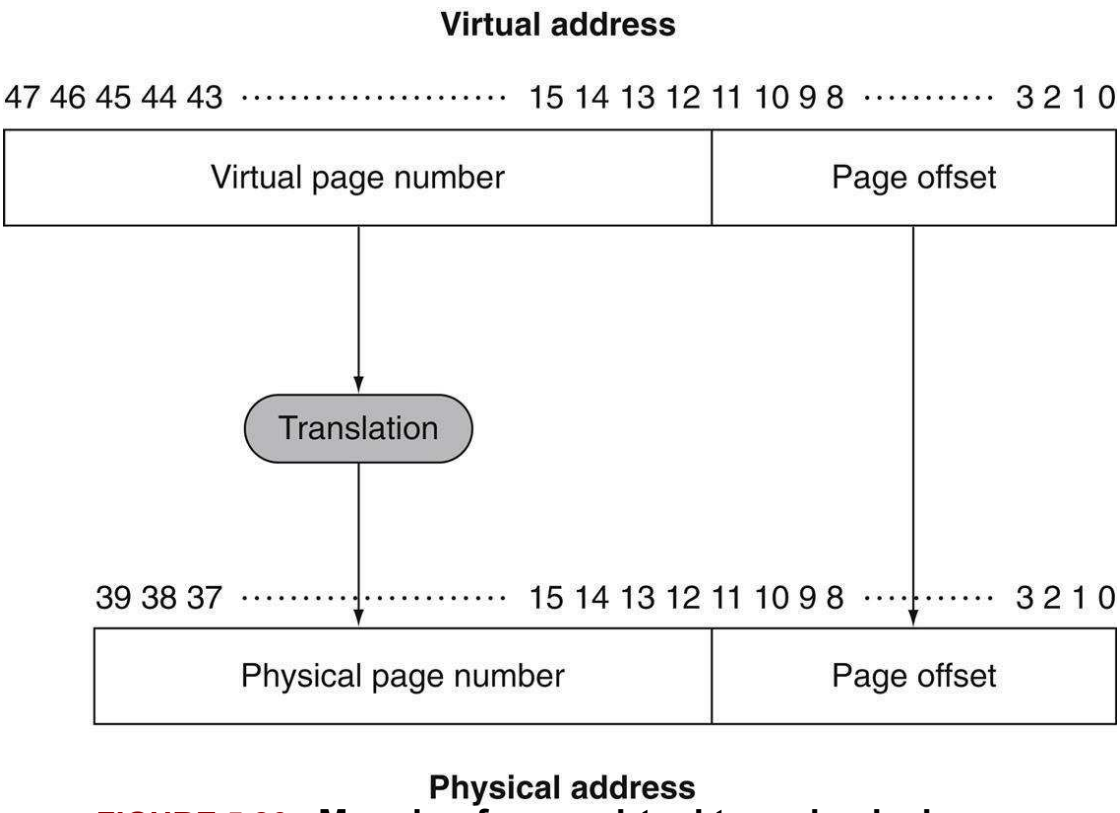


FIGURE 5.26 Mapping from a virtual to a physical address.
The page size is 2^{12} = 4 KiB. The number of physical pages allowed in memory is 2^{28}, since the physical page number has 28 bits in it. Thus, main memory can have at most 1 TiB, while the virtual address space is 256 TiB. RISC-V allows physical memory to be up to 1 PiB; we chose 1 TiB because it is ample for many computers in 2016.

Many design choices in virtual memory systems are motivated by the high cost of a page fault. A page fault to disk will take millions of clock cycles to process. (The table on page 372 shows that main memory latency is about 100,000 times quicker than disk.) This enormous miss penalty, dominated by the time to get the first word for typical page sizes, leads to several key decisions in designing virtual memory systems:

- Pages should be large enough to try to amortize the high access time. Sizes from 4 KiB to 64 KiB are typical today. New desktop and server systems are being developed to support 32 KiB and 64 KiB pages, but new embedded systems are going in the other direction, to 1 KiB pages.
- Organizations that reduce the page fault rate are attractive. The

primary technique used here is to allow fully associative placement of pages in memory.

- Page faults can be handled in software because the overhead will be small compared to the disk access time. In addition, software can afford to use clever algorithms for choosing how to place pages because even little reductions in the miss rate will pay for the cost of such algorithms.
- Write-through will not work for virtual memory, since writes take too long. Instead, virtual memory systems use write-back.

The next few subsections address these factors in virtual memory design.

Elaboration

We present the motivation for virtual memory as many virtual machines sharing the same memory, but virtual memory was originally invented so that many programs could share a computer as part of a timesharing system. Since many readers today have no experience with time-sharing systems, we use virtual machines to motivate this section.

Elaboration

RISC-V supports a variety of virtual memory configurations. In addition to the 48-bit virtual address scheme, which is a good fit for large servers in 2016, the architecture can support 39-bit and 57-bit virtual address spaces. All of these configurations use a page size of 4 Kibibytes.

Elaboration

For servers and even PCs, 32-bit address processors are problematic. Although we normally think of virtual addresses as much larger than physical addresses, the opposite can occur when the processor address size is small relative to the state of the memory technology. No single program or virtual machine can benefit, but a collection of programs or virtual machines running at the same time can benefit from not having to be swapped out of main memory or by running on parallel processors.

Elaboration

The discussion of virtual memory in this book focuses on paging, which uses fixed-size blocks. There is also a variable-size block scheme called **segmentation**. In segmentation, an address consists of two parts: a segment number and a segment offset. The segment number is mapped to a physical address, and the offset is *added* to find the actual physical address. Because the segment can vary in size, a bounds check is also needed to make sure that the offset is within the segment. The major use of segmentation is to support more powerful methods of protection and sharing in an address space. Most operating system textbooks contain extensive discussions of segmentation compared to paging and of the use of segmentation to share the address space logically. The major disadvantage of segmentation is that it splits the address space into logically separate pieces that must be manipulated as a two-part address: the segment number and the offset. Paging, in contrast, makes the boundary between page number and offset invisible to programmers and compilers.

Segments have also been used as a method to extend the address space without changing the word size of the computer. Such attempts have been unsuccessful because of the awkwardness and performance penalties inherent in a two-part address, of which programmers and compilers must be aware.

Many architectures divide the address space into large fixed-size blocks that simplify protection between the operating system and user programs and increase the efficiency of implementing paging. Although these divisions are often called "segments," this mechanism is much simpler than variable block size segmentation and is not visible to user programs; we discuss it in more detail shortly.

segmentation

A variable-size address mapping scheme in which an address consists of two parts: a segment number, which is mapped to a physical address, and a segment offset.

Placing a Page and Finding It Again

Because of the incredibly high penalty for a page fault, designers reduce page fault frequency by optimizing page placement. If we allow a virtual page to be mapped to any physical page, the operating system can then choose to replace any page it wants when a page fault occurs. For example, the operating system can use a sophisticated algorithm and complex data structures that track page usage to try to choose a page that will not be needed for a long time. The ability to use a clever and flexible replacement scheme reduces the page fault rate and simplifies the use of fully associative placement of pages.

As mentioned in Section 5.4, the difficulty in using fully associative placement is in locating an entry, since it can be anywhere in the upper level of the hierarchy. A full search is impractical. In virtual memory systems, we locate pages by using a table that indexes the main memory; this structure is called a **page table**, and it resides in main memory. A page table is indexed by the page number from the virtual address to discover the corresponding physical page number. Each program has its own page table, which maps the virtual address space of that program to main memory. In our library analogy, the page table corresponds to a mapping between book titles and library locations. Just as the card catalog may contain entries for books in another library on campus rather than the local branch library, we will see that the page table may contain entries for pages not present in memory. To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table; we call this the *page table register*. Assume for now that the page table is in a fixed and contiguous area of memory.

page table

The table containing the virtual to physical address translations in a virtual memory system. The table, which is stored in memory, is typically indexed by the virtual page number; each entry in the table contains the physical page number for that virtual page if the page is currently in memory.

Hardware/Software Interface

The page table, together with the program counter and the

registers, specifies the *state* of a virtual machine. If we want to allow another virtual machine to use the processor, we must save this state. Later, after restoring this state, the virtual machine can continue execution. We often refer to this state as a *process*. The process is considered *active* when it is in possession of the processor; otherwise, it is considered *inactive*. The operating system can make a process active by loading the process's state, including the program counter, which will initiate execution at the value of the saved program counter.

The process's address space, and hence all the data it can access in memory, is defined by its page table, which resides in memory. Rather than save the entire page table, the operating system simply loads the page table register to point to the page table of the process it wants to make active. Each process has its own page table, since different processes use the same virtual addresses. The operating system is responsible for allocating the physical memory and updating the page tables, so that the virtual address spaces of distinct processes do not collide. As we will see shortly, the use of separate page tables also provides protection of one process from another.

Figure 5.27 uses the page table register, the virtual address, and the indicated page table to show how the hardware can form a physical address. A valid bit is used in each page table entry, just as we did in a cache. If the bit is off, the page is not present in main memory and a page fault occurs. If the bit is on, the page is in memory and the entry contains the physical page number.

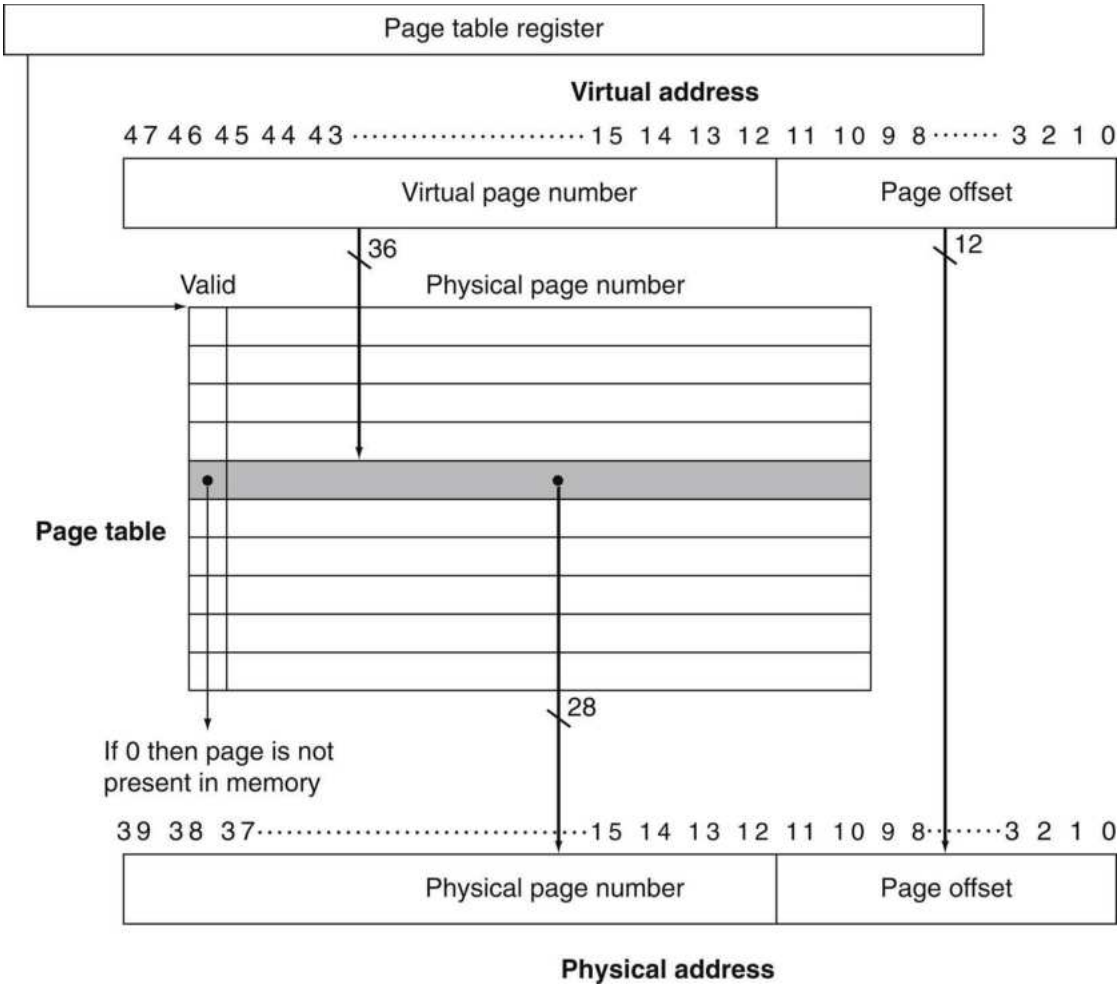


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address.
We assume a 48-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{48} bytes, or 256 TiB, and the physical address space is 2^{40} bytes, which allows main memory of up to 1 TiB. If RISC-V used a single page table as shown in this figure, the number of entries in the page table would be 2^{36}, or about 64 billion entries. (We'll see what RISC-V does to reduce the number of entries shortly.) The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 29 bits wide, it would typically be rounded up to a power of 2 bits for ease of indexing. The page table entries in RISC-V are 64 bits. The extra bits would be used to store additional information that needs to be kept on a per-page basis,

such as protection.

Because the page table contains a mapping for every possible virtual page, no tags are required. In cache terminology, the index that is used to access the page table consists of the full block address, which in this case is the virtual page number.

Page Faults

If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This transfer is done with the exception mechanism, which we saw in Chapter 4 and will discuss again later in this section. Once the operating system gets control, it must find the page in the next level of the hierarchy (usually flash memory or magnetic disk) and decide where to place the requested page in the main memory.

The virtual address alone does not immediately tell us where the page is in secondary memory. Returning to our library analogy, we cannot find the location of a library book on the shelves just by knowing its title. Instead, we go to the catalog and look up the book, obtaining an address for the location on the shelves, such as the Library of Congress call number. Likewise, in a virtual memory system, we must keep track of the location in secondary memory of each page in virtual address space.

Because we do not know ahead of time when a page in memory will be replaced, the operating system usually creates the space on flash memory or disk for all the pages of a process when it creates the process. This space is called the **swap space**. At that time, it also creates a data structure to record where each virtual page is stored on disk. This data structure may be part of the page table or may be an auxiliary data structure indexed in the same way as the page table. Figure 5.28 shows the organization when a single table holds either the physical page number or the secondary memory address.

swap space

The space on the disk reserved for the full virtual memory space of a process.

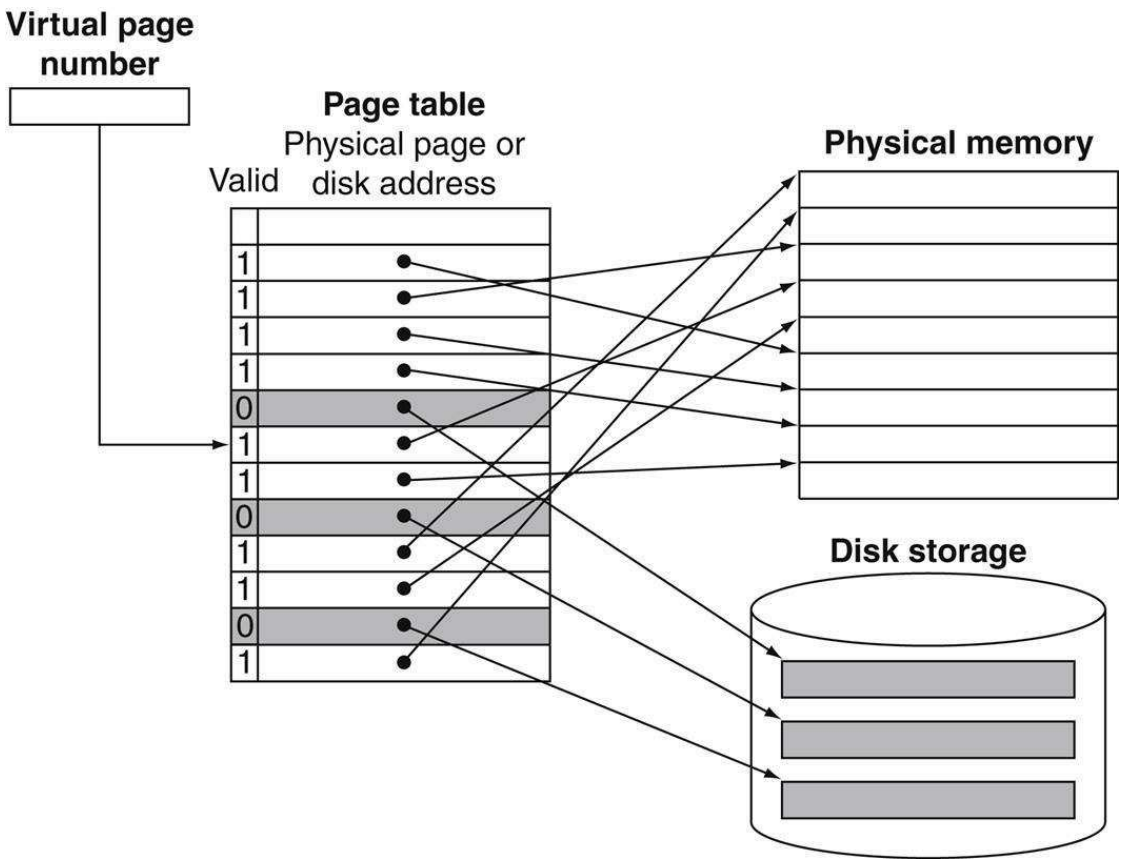


FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy.

The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

The operating system also creates a data structure that tracks which processes and which virtual addresses use each physical page. When a page fault occurs, if all the pages in main memory are in use, the operating system must choose a page to replace. Because we want to minimize the number of page faults, most operating

systems try to choose a page that they hypothesize will not be needed soon. Using the past to predict the future, operating systems follow the *least recently used* (LRU) replacement scheme, which we mentioned in Section 5.4. The operating system searches for the least recently used page, assuming that a page that has not been used in a long time is less likely to be needed than a more recently accessed page. The replaced pages are written to swap space in secondary memory. In case you are wondering, the operating system is just another process, and these tables controlling memory are in memory; the details of this seeming contradiction will be explained shortly.

Hardware/Software Interface

Implementing a completely accurate LRU scheme is too expensive, since it requires updating a data structure on *every* memory reference. Thus, most operating systems approximate LRU by keeping track of which pages have and which pages have not been recently used. To help the operating system estimate the LRU pages, RISC-V computers provide a **reference bit**, sometimes called a **use bit** or **access bit**, which is set whenever a page is accessed. The operating system periodically clears the reference bits and later records them so it can determine which pages were touched during a particular time period. With this usage information, the operating system can select a page that is among the least recently referenced (detected by having its reference bit off). If this bit is not provided by the hardware, the operating system must find another way to estimate which pages have been accessed.

reference bit

Also called **use bit** or **access bit**. A field that is set whenever a page is accessed and that is used to implement LRU or other replacement schemes.

Virtual Memory for Large Virtual Addresses

The caption in Figure 5.27 points out that with a single level page table for a 48-bit address with 4 KiB pages, we need 64 billion table

entries. As each page table entry is 8 bytes for RISC-V, it would require 0.5 TiB just to map the virtual addresses to physical addresses! Moreover, there could be hundreds of processes running, each with its own page table. That much memory for translation would be unaffordable even for the largest systems.

A range of techniques is used to reduce the amount of storage required for the page table. The five techniques below aim at reducing the total maximum storage required as well as minimizing the main memory dedicated to page tables:

1. The simplest technique is to keep a limit register that restricts the size of the page table for a given process. If the virtual page number becomes larger than the contents of the limit register, entries must be added to the page table. This technique allows the page table to grow as a process consumes more space. Thus, the page table will only be large if the process is using many pages of virtual address space. This technique requires that the address space expand in just one direction.
2. Allowing growth in only one direction is not sufficient, since most languages require two areas whose size is expandable: one area holds the stack, and the other area holds the heap. Because of this duality, it is convenient to divide the page table and let it grow from the highest address down, as well as from the lowest address up. This means that there will be two separate page tables and two separate limits. The use of two page tables breaks the address space into two segments. The high-order bit of an address usually determines which segment and thus which page table to use for that address. Since the high-order address bit specifies the segment, each segment can be as large as one-half of the address space. A limit register for each segment specifies the current size of the segment, which grows in units of pages. Unlike the type of segmentation discussed in the second elaboration on page 423, this form of segmentation is invisible to the application program, although not to the operating system. The major disadvantage of this scheme is that it does not work well when the address space is used in a sparse fashion rather than as a contiguous set of virtual addresses.
3. Another approach to reducing the page table size is to apply a hashing function to the virtual address so that the page table need be only the size of the number of *physical* pages in main memory.

Such a structure is called an *inverted page table*. Of course, the lookup process is slightly more complex with an inverted page table, because we can no longer just index the page table.

4. To reduce the actual main memory tied up in page tables, most modern systems also allow the page tables to be paged. Although this sounds tricky, it works by using the same basic ideas of virtual memory and simply allowing the page tables to reside in the virtual address space. In addition, there are some small but critical problems, such as a never-ending series of page faults, which must be avoided. How these problems are overcome is both very detailed and typically highly processor-specific. In brief, these problems are avoided by placing all the page tables in the address space of the operating system and placing at least some of the page tables for the operating system in a portion of main memory that is physically addressed and is always present and thus never in secondary memory.

5. Multiple levels of page tables can also be used to reduce the total amount of page table storage, and this is the solution that RISC-V uses to reduce the memory footprint of address translation. Figure 5.29 above shows the four levels of address translation to go from a 48-bit virtual address to a 40-bit physical address of a 4 KiB page. Address translation happens by first looking in the level 0 table, using the highest-order bits of the address. If the address in this table is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry, and so on. Thus, the level 0 table maps the virtual address to a 512 GB (2^{39} bytes) region. The level 1 table in turn maps the virtual address to a 1 GB (2^{30}) region. The next level maps this down to a 2 MB (2^{21}) region. The final table maps the virtual address to the 4 KiB (2^{12}) memory page. This scheme allows the address space to be used in a sparse fashion (multiple noncontiguous segments can be active) without having to allocate the entire page table. Such schemes are particularly useful with very large address spaces and in software systems that require noncontiguous allocation. The primary disadvantage of this multi-level mapping is the more complex process for address translation.

What about Writes?

The difference between the access time to the cache and main memory is tens to hundreds of cycles, and write-through schemes can be used, although we need a write buffer to hide the latency of the write from the processor. In a virtual memory system, writes to the next level of the hierarchy (disk) can take millions of processor clock cycles; therefore, building a write buffer to allow the system to write-through to disk would be completely impractical. Instead, virtual memory systems must use write-back, performing the individual writes into the page in memory, and copying the page back to secondary memory when it is replaced in the main memory.

Hardware/Software Interface

A write-back scheme has another major advantage in a virtual memory system. Because the disk transfer time is small compared with its access time, copying back an entire page is much more efficient than writing individual words back to the disk. A write-back operation, although faster than transferring separate words, is still costly. Thus, we would like to know whether a page *needs* to be copied back when we choose to replace it. To track whether a page has been written since it was read into the memory, a *dirty bit* is added to the page table. The dirty bit is set when any word in a page is written. If the operating system chooses to replace the page, the dirty bit indicates whether the page needs to be written out before its location in memory can be given to another page. Hence, a modified page is often called a *dirty* page.

Making Address Translation Fast: the TLB

Since the page tables are stored in main memory, every memory access by a program can take at least twice as long: one memory access to obtain the physical address and a second access to get the data. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again soon, because the references to the words on that page have both temporal and spatial locality.

Accordingly, modern processors include a special cache that keeps track of recently used translations. This special address translation cache is traditionally referred to as a **translation-**

lookaside buffer (TLB), although it would be more accurate to call it a translation cache. The TLB corresponds to that little piece of paper we typically use to record the location of a set of books we look up in the card catalog; rather than continually searching the entire catalog, we record the location of several books and use the scrap of paper as a cache of Library of Congress call numbers.

translation-lookaside buffer (TLB)

A cache that keeps track of recently used address mappings to try to avoid an access to the page table.

Figure 5.30 shows that each tag entry in the TLB holds a portion of the virtual page number, and each data entry of the TLB holds a physical page number. Because we access the TLB instead of the page table on every reference, the TLB will need to include other status bits, such as the dirty and the reference bits. Although Figure 5.30 shows a single page table, TLBs work fine with multi-level page tables as well. The TLB simply loads the physical address and protection tags from the last level page table.

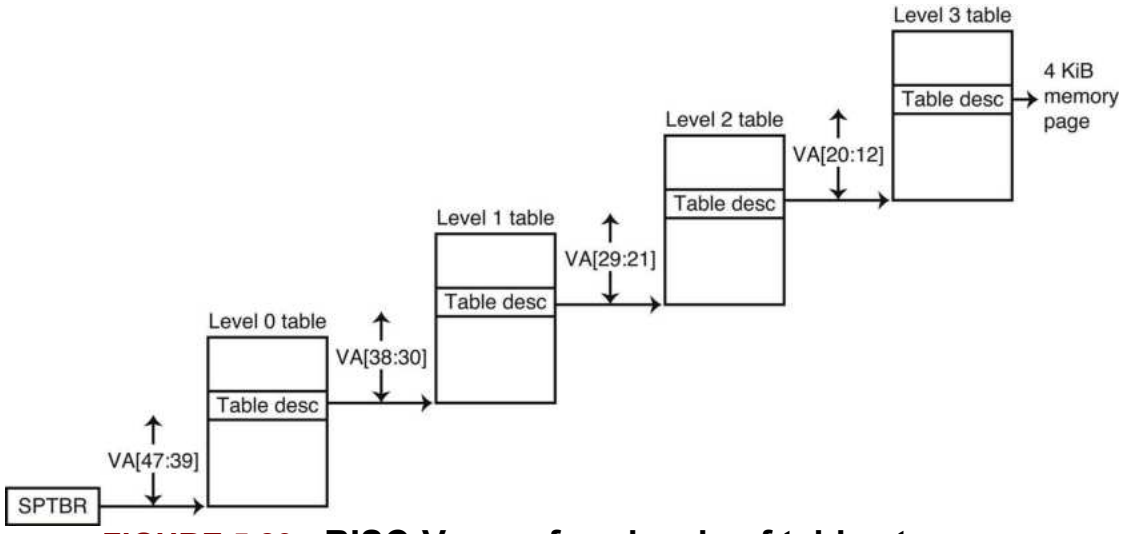


FIGURE 5.29 RISC-V uses four levels of tables to translate a 48-bit virtual address into a 40-bit physical address.
Rather than needing 64 billion page table entries for the single page table in Figure 5.27, this hierarchical approach needs just a tiny fraction. Each step of the translation uses 9 bits of the virtual address to find the

next level table, until the upper 36 bits of the virtual address are mapped to the physical address of the desired 4 KiB page. Each RISC-V page table entry is 8 bytes, so the 512 entries of a table fill a single 4 KiB page. The *Supervisor Page Table Base Register* (SPTBR) gives the starting address of the first page table.

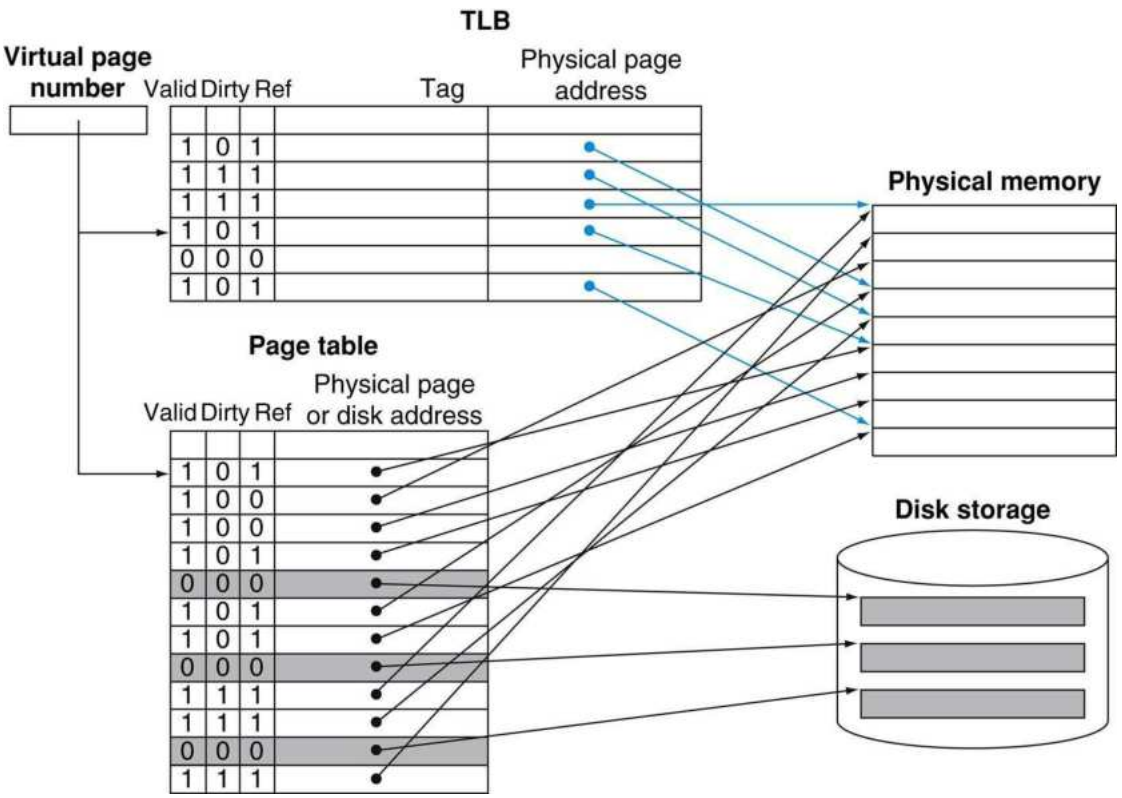


FIGURE 5.30 The TLB acts as a cache of the page table for the entries that map to physical pages only.
The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

On every reference, we look up the virtual page number in the TLB. If we get a hit, the physical page number is used to form the address, and the corresponding reference bit is turned on. If the processor is performing a write, the dirty bit is also turned on. If a miss in the TLB occurs, we must determine whether it is a page fault or merely a TLB miss. If the page exists in memory, then the TLB miss indicates only that the translation is missing. In such

cases, the processor can handle the TLB miss by loading the translation from the (last-level) page table into the TLB and then trying the reference again. If the page is not present in memory, then the TLB miss indicates a true page fault. In this case, the processor invokes the operating system using an exception. Because the TLB has many fewer entries than the number of pages in main memory, TLB misses will be much more frequent than true page faults.

TLB misses can be handled either in hardware or in software. In practice, with care there can be little performance difference between the two approaches, because the basic operations are the same in either case.

After a TLB miss occurs and the missing translation has been retrieved from the page table, we will need to select a TLB entry to replace. Because the reference and dirty bits are contained in the TLB entry, we need to copy these bits back to the page table entry when we replace an entry. These bits are the only portion of the TLB entry that can be changed. Using write-back—that is, copying these entries back at miss time rather than when they are written—is very efficient, since we expect the TLB miss rate to be small. Some systems use other techniques to approximate the reference and dirty bits, eliminating the need to write into the TLB except to load a new table entry on a miss.

Some typical values for a TLB might be

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10–100 clock cycles
- Miss rate: 0.01%–1%

Designers have used a wide variety of associativities in TLBs. Some systems use small, fully associative TLBs because a fully associative mapping has a lower miss rate; furthermore, since the TLB is small, the cost of a fully associative mapping is not too high. Other systems use large TLBs, often with small associativity. With a fully associative mapping, choosing the entry to replace becomes tricky since implementing a hardware LRU scheme is too expensive. Furthermore, since TLB misses are much more frequent than page faults and thus must be handled more cheaply, we cannot afford an expensive software algorithm, as we can for page

faults. As a result, many systems provide some support for randomly choosing an entry to replace. We'll examine replacement schemes in a little more detail in Section 5.8.

The Intrinsity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsity FastMATH. The memory system uses 4 KiB pages and just a 32-bit address space; thus, the virtual page number is 20 bits long. The physical address is the same size as the virtual address. The TLB contains 16 entries, it is fully associative, and it is shared between the instruction and data references. Each entry is 64 bits wide and contains a 20-bit tag (which is the virtual page number for that TLB entry), the corresponding physical page number (also 20 bits), a valid bit, a dirty bit, and other bookkeeping bits. Like most MIPS systems, it uses software to handle TLB misses.

Figure 5.31 shows the TLB and one of the caches, while Figure 5.32 shows the steps in processing a read or write request. When a TLB miss occurs, the hardware saves the page number of the reference in a special register and generates an exception. The exception invokes the operating system, which handles the miss in software. To find the physical address for the missing page, a TLB miss indexes the page table using the page number of the virtual address and the page table register, which indicates the starting address of the active process page table. Using a special set of system instructions that can update the TLB, the operating system places the physical address from the page table into the TLB. A TLB miss takes about 13 clock cycles, assuming the code and the page table entry are in the instruction cache and data cache, respectively. A true page fault occurs if the page table entry does not have a valid physical address. The hardware maintains an index that indicates the recommended entry to replace; it is chosen randomly.

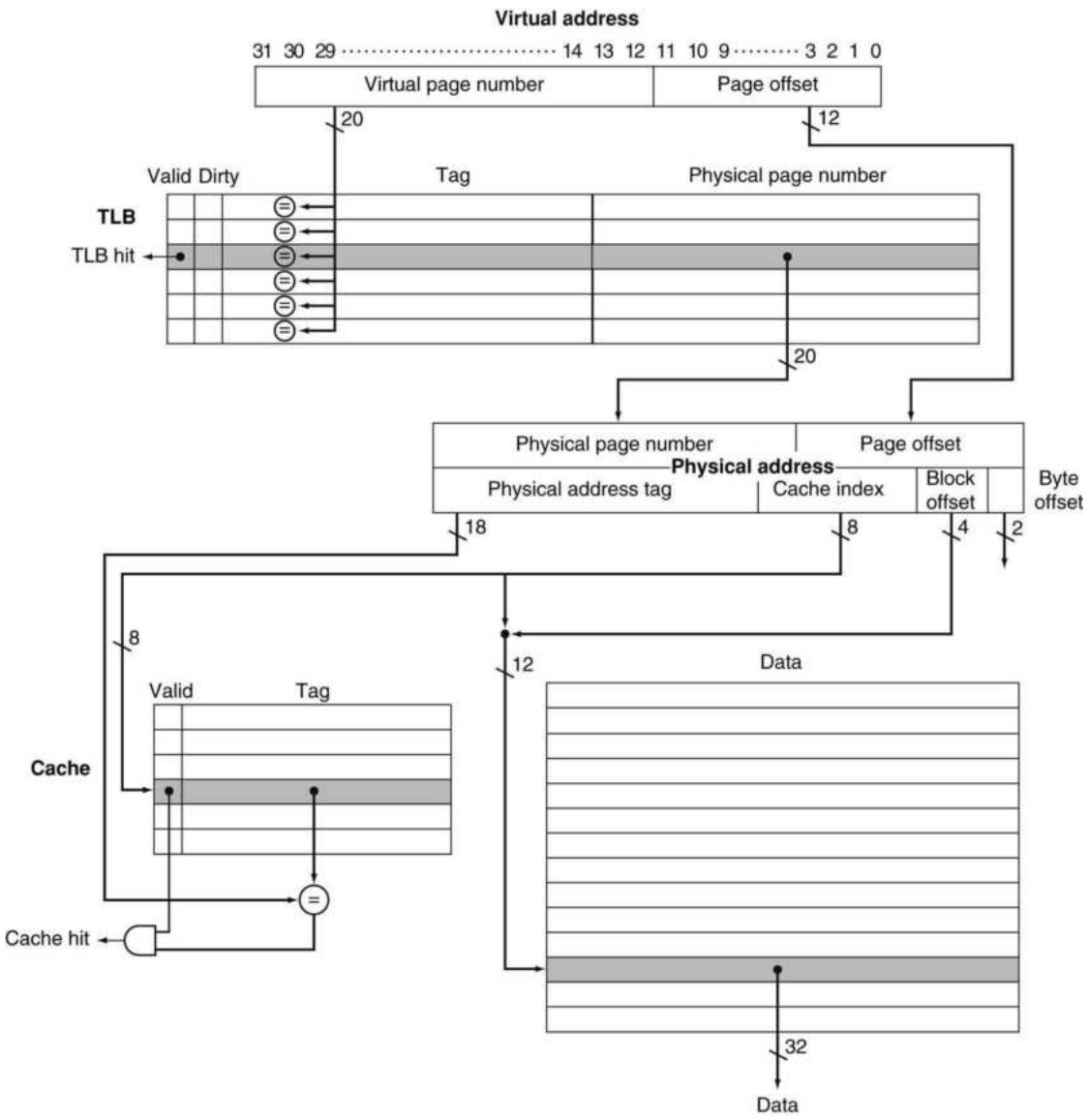


FIGURE 5.31 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity FastMATH.
This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. Note that the address size for this computer is just 32 bits. This diagram focuses on a read; Figure 5.32 describes how to handle writes. Note that unlike Figure 5.12, the tag and data RAMs are split. By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in

the TLB. (See content addressable memories in the *Elaboration* on page 400.) If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.

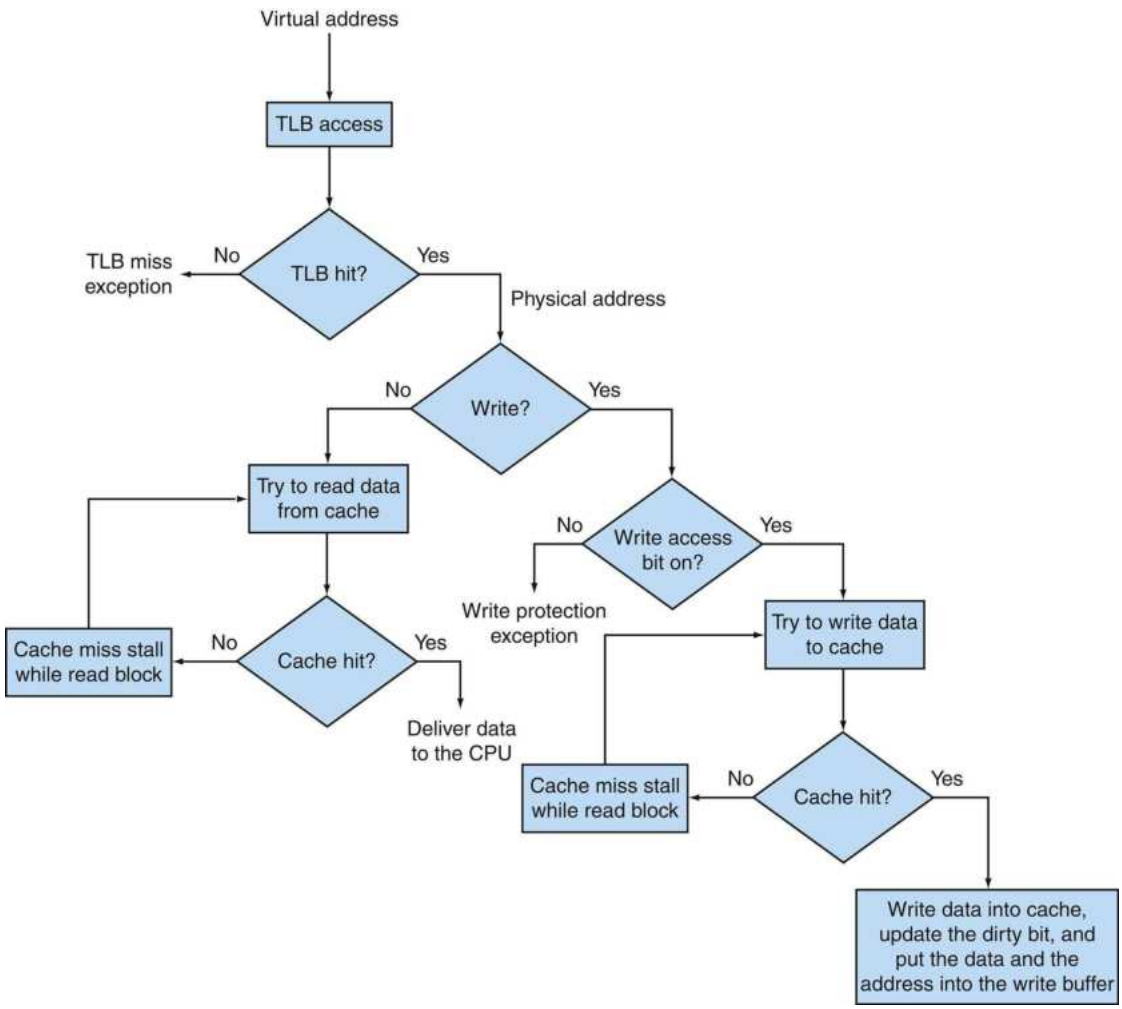


FIGURE 5.32 Processing a read or a write-through in the Intrinsity FastMATH TLB and cache. If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall while the data are brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data are sent to the write buffer if we assume write-through. A write miss is just like a read miss except that the block is modified after it is read from memory. Write-back requires writes to set a dirty bit for the cache block, and a write buffer is loaded with the whole block only on a read miss or write miss if the block to be replaced is dirty. Notice that a TLB hit and a cache hit are independent events, but a cache hit can only occur after a TLB hit occurs, which means that the data must be present in memory. The relationship between TLB misses and cache misses is examined further in the following example and the exercises at the end of this chapter. Note that

the address size for this computer is just 32 bits.

There is an extra complication for write requests: namely, the write access bit in the TLB must be checked. This bit prevents the program from writing into pages for which it has only read access. If the program attempts a write and the write access bit is off, an exception is generated. The write access bit forms part of the protection mechanism, which we will discuss shortly.

Integrating Virtual Memory, TLBs, and Caches

Our virtual memory and cache systems work together as a hierarchy, so that data cannot be in the cache unless it is present in main memory. The operating system helps maintain this hierarchy by flushing the contents of any page from the cache when it decides to migrate that page to secondary memory. At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault.

Under the best of circumstances, a virtual address is translated by the TLB and sent to the cache where the appropriate data are found, retrieved, and sent back to the processor. In the worst case, a reference can miss in all three components of the memory hierarchy: the TLB, the page table, and the cache. The following example illustrates these interactions in more detail.

Overall Operation of a Memory Hierarchy

Example

In a memory hierarchy like that of Figure 5.31, which includes a TLB and a cache organized as shown, a memory reference can encounter three different types of misses: a TLB miss, a page fault, and a cache miss. Consider all the combinations of these three events with one or more occurring (seven possibilities). For each possibility, state whether this event can actually occur and under what circumstances.

Answer

Figure 5.33 shows all combinations and whether each is possible in

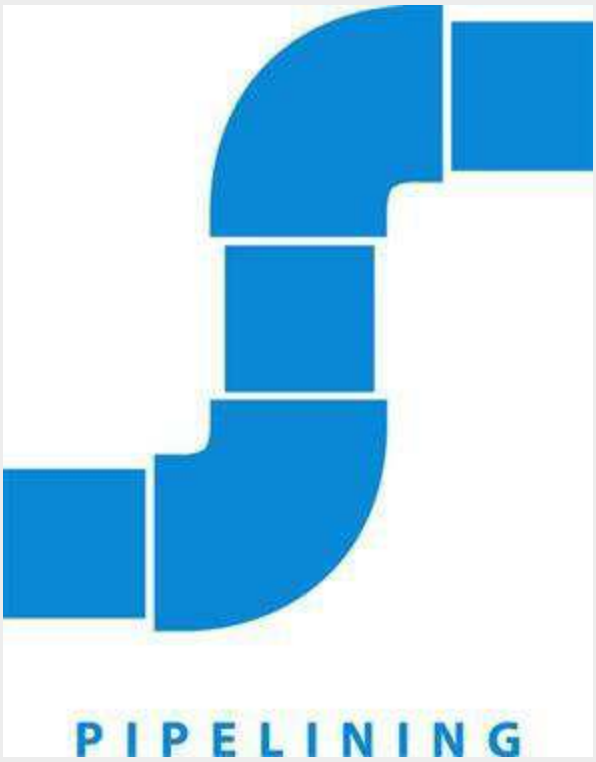
practice.

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.33 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, page table hit, cache miss) but never detected.

Elaboration

Figure 5.33 assumes that all memory addresses are translated to physical addresses before the cache is accessed. In this organization, the cache is *physically indexed* and *physically tagged* (both the cache index and tag are physical, rather than virtual, addresses). In such a system, the amount of time to access memory, assuming a cache hit, must accommodate both a TLB access and a cache access; of course, these accesses can be **pipelined**.



Alternatively, the processor can index the cache with an address that is completely or partially virtual. This is called a **virtually addressed cache**, and it uses tags that are virtual addresses; hence, such a cache is *virtually indexed* and *virtually tagged*. In such caches, the address translation hardware (TLB) is unused during the normal cache access, since the cache is accessed with a virtual address that has not been translated to a physical address. This takes the TLB out of the critical path, reducing cache latency. When a cache miss occurs, however, the processor needs to translate the address to a physical address so that it can fetch the cache block from main memory.

virtually addressed cache

A cache that is accessed with a virtual address rather than a physical address.

When the cache is accessed with a virtual address and pages are shared between processes (which may access them with different virtual addresses), there is the possibility of **aliasing**. Aliasing occurs when the same object has two names—in this case, two virtual addresses for the same page. This ambiguity creates a problem, because a word on such a page may be cached in two

different locations, each corresponding to distinct virtual addresses. This ambiguity would allow one program to write the data without the other program being aware that the data had changed. Completely virtually addressed caches either introduce design limitations on the cache and TLB to reduce aliases or require the operating system, and possibly the user, to take steps to ensure that aliases do not occur.

aliasing

A situation in which two addresses access the same object; it can occur in virtual memory when there are two virtual addresses for the same physical page.

A common compromise between these two design points is caches that are virtually indexed—sometimes using just the page-offset portion of the address, which is really a physical address since it is not translated—but use physical tags. These designs, which are *virtually indexed but physically tagged*, attempt to achieve the performance advantages of virtually indexed caches with the architecturally simpler advantages of a **physically addressed cache**. For example, there is no alias problem in this case. Figure 5.31 assumed a 4 KiB page size, but it's really 16 KiB, so the Intrinsity FastMATH can use this trick. To pull it off, there must be careful coordination between the minimum page size, the cache size, and associativity. RISC-V requires caches to *behave as though* physically tagged and indexed, but it does not mandate this implementation. For example, virtually indexed, physically tagged data caches could use additional logic to ensure that software cannot tell the difference.

physically addressed cache

A cache that is addressed by a physical address.

Implementing Protection with Virtual Memory

Perhaps the most important function of virtual memory today is to allow sharing of a single main memory by multiple processes, while providing memory protection among these processes and the operating system. The protection mechanism must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process or into the operating system either intentionally or unintentionally. The write access bit in the TLB can protect a page from being written. Without this level of protection,

computer viruses would be even more widespread.

Hardware/Software Interface

To enable the operating system to implement protection in the virtual memory system, the hardware must provide at least the three basic capabilities summarized below. Note that the first two are the same requirements as needed for virtual machines (Section 5.6).

1. Support at least two modes that indicate whether the running process is a user process or an operating system process, variously called a **supervisor** process, a **kernel** process, or an *executive* process.
2. Provide a portion of the processor state that a user process can read but not write. This state includes the user/supervisor mode bit, which dictates whether the processor is in user or supervisor mode, the page table pointer, and the TLB. To write these elements, the operating system uses special instructions that are only available in supervisor mode.
3. Provide mechanisms whereby the processor can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a **system call** exception, implemented as a special instruction (`ecall` in the RISC-V instruction set) that transfers control to a dedicated location in supervisor code space. As with any other exception, the program counter from the point of the system call is saved in the *supervisor exception program counter* (SEPC), and the processor is placed in supervisor mode. To return to user mode from the exception, use the *supervisor exception return* (`sret`) instruction, which resets to user mode and jumps to the address in SEPC.

By using these mechanisms and storing the page tables in the operating system's address space, the operating system can change the page tables while preventing a user process from changing them, ensuring that a user process can access only the storage provided to it by the operating system.

supervisor mode

Also called **kernel mode**. A mode indicating that a running process is an operating system process.

system call

A special instruction that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process.

We also want to prevent a process from reading the data of another process. For example, we wouldn't want a student program to read the teacher's grades while they were in the processor's memory. Once we begin sharing main memory, we must provide the ability for a process to protect its data from both reading and writing by another process; otherwise, sharing the main memory will be a mixed blessing!

Remember that each process has its own virtual address space. Thus, if the operating system keeps the page tables organized so that the independent virtual pages map to disjoint physical pages, one process will not be able to access another's data. Of course, this also requires that a user process be unable to change the page table mapping. The operating system can assure safety if it prevents the user process from modifying its own page tables. However, the operating system must be able to modify the page tables. Placing the page tables in the protected address space of the operating system satisfies both requirements.

When processes want to share information in a limited way, the operating system must assist them, since accessing the information of another process requires changing the page table of the accessing process. The write access bit can be used to restrict the sharing to just read sharing, and, like the rest of the page table, this bit can be changed only by the operating system. To allow another process, say, P1, to read a page owned by process P2, P2 would ask the operating system to create a page table entry for a virtual page in P1's address space that points to the same physical page that P2 wants to share. The operating system could use the write protection bit to prevent P1 from writing the data, if that was P2's wish. Any bits that determine the access rights for a page must be included in both the page table and the TLB, because the page table is accessed only on a TLB *miss*.

Elaboration

When the operating system decides to change from running process P1 to running process P2 (called a **context switch** or *process switch*), it must ensure that P2 cannot get access to the page tables of P1 because that would compromise protection. If there is no TLB, it suffices to change the page table register to point to P2's page table (rather than to P1's); with a TLB, we must clear the TLB entries that belong to P1—both to protect the data of P1 and to force the TLB to load the entries for P2. If the process switch rate were high, this could be quite inefficient. For example, P2 might load only a few TLB entries before the operating system switched back to P1. Unfortunately, P1 would then find that all its TLB entries were gone and would have to pay TLB misses to reload them. This problem arises because the virtual addresses used by P1 and P2 are the same, and we must clear out the TLB to avoid confusing these addresses.

A common alternative is to extend the virtual address space by adding a *process identifier* or *task identifier*. The Intrinsity FastMATH has an 8-bit *address space ID* (ASID) field for this purpose. This small field identifies the currently running process; it is kept in a register loaded by the operating system when it switches processes. RISC-V also offers ASID to reduce TLB flushes on context switches. The process identifier is concatenated to the tag portion of the TLB, so that a TLB hit occurs only if both the page number *and* the process identifier match. This combination eliminates the need to clear the TLB, except on rare occasions.

Similar problems can occur for a cache, since on a process switch, the cache will contain data from the running process. These problems arise in different ways for physically addressed and virtually addressed caches, and a variety of solutions, such as process identifiers, are used to ensure that a process gets its own data.

context switch

A changing of the internal state of the processor to allow a different process to use the processor that includes saving the state needed to return to the currently executing process.

Handling TLB Misses and Page Faults

Although the translation of virtual to physical addresses with a TLB is straightforward when we get a TLB hit, as we saw earlier, handling TLB misses and page faults is more complex. A TLB miss occurs when no entry in the TLB matches a virtual address. Recall that a TLB miss can indicate one of two possibilities:

1. The page is present in memory, and we need only create the missing TLB entry.
2. The page is not present in memory, and we need to transfer control to the operating system to deal with a page fault.

Handling a TLB miss or a page fault requires using the exception mechanism to interrupt the active process, transferring control to the operating system, and later resuming execution of the interrupted process. A page fault will be recognized sometime during the clock cycle used to access memory. To restart the instruction after the page fault is handled, the program counter of the instruction that caused the page fault must be saved. The *supervisor exception program counter* (SEPC) register is used to hold this value.

In addition, a TLB miss or page fault exception must be asserted by the end of the same clock cycle that the memory access occurs, so that the next clock cycle will begin exception processing rather than continue normal instruction execution. If the page fault was not recognized in this clock cycle, a load instruction could overwrite a register, and this could be disastrous when we try to restart the instruction. For example, consider the instruction `lb x10, 0(x10)`: the computer must be able to prevent the write pipeline stage from occurring; otherwise, it could not properly restart the instruction, since the contents of `x10` would have been destroyed. A similar complication arises on stores. We must prevent the write into memory from actually completing when there is a page fault; this is usually done by deasserting the write control line to the memory.

Hardware/Software Interface

Between the time we begin executing the exception handler in the operating system and the time that the operating system has saved all the state of the process, the operating system is particularly vulnerable. For instance, if another exception occurred when we

were processing the first exception in the operating system, the control unit would overwrite the exception link register, making it impossible to return to the instruction that caused the page fault! We can avoid this disaster by providing the ability to **disable** and **enable exceptions**. When an exception first occurs, the processor sets a bit that disables all other exceptions; this could happen at the same time the processor sets the supervisor mode bit. The operating system will then save just enough state to allow it to recover if another exception occurs—namely, the *supervisor exception rogram counter* (SEPC) and the *supervisor exception cause* (SCAUSE) registers, which as we saw in Chapter 4 records the reason for the exception. SEPC and SCAUSE in RISC-V are two of the special control registers that help with exceptions, TLB misses, and page faults. The operating system can then re-enable exceptions. These steps make sure that exceptions will not cause the processor to lose any state and thereby be unable to restart execution of the interrupting instruction.

exception enable

Also called interrupt enable. A signal or action that controls whether the process responds to an exception or not; necessary for preventing the occurrence of exceptions during intervals before the processor has safely saved the state needed to restart.

Once the operating system knows the virtual address that caused the page fault, it must complete three steps:

1. Look up the page table entry using the virtual address and find the location of the referenced page in secondary memory.
2. Choose a physical page to replace; if the chosen page is dirty, it must be written out to secondary memory before we can bring a new virtual page into this physical page.
3. Start a read to bring the referenced page from secondary memory into the chosen physical page.

Of course, this last step will take millions of processor clock cycles for disks (so will the second if the replaced page is dirty); accordingly, the operating system will usually select another process to execute in the processor until the disk access completes. Because the operating system has saved the state of the process, it

can freely give control of the processor to another process.

When the read of the page from secondary memory is complete, the operating system can restore the state of the process that originally caused the page fault and execute the instruction that returns from the exception. This instruction will reset the processor from kernel to user mode, as well as restore the program counter. The user process then re-executes the instruction that faulted, accesses the requested page successfully, and continues execution.

Page fault exceptions for data accesses are difficult to implement properly in a processor because of a combination of three characteristics:

1. They occur in the middle of instructions, unlike instruction page faults.
2. The instruction cannot be completed before handling the exception.
3. After handling the exception, the instruction must be restarted as if nothing had occurred.

Making instructions **restartable**, so that the exception can be handled and the instruction later continued, is relatively easy in an architecture like the RISC-V. Because each instruction writes only one data item and this write occurs at the end of the instruction cycle, we can simply prevent the instruction from completing (by not writing) and restart the instruction at the beginning.

restartable instruction

An instruction that can resume execution after an exception is resolved without the exception's affecting the result of the instruction.

Elaboration

For processors with more complex instructions that can touch many memory locations and write many data items, making instructions restartable is much harder. Processing one instruction may generate a number of page faults in the middle of the instruction. For example, x86 processors have block move instructions that touch thousands of data words. In such processors, instructions often cannot be restarted from the beginning, as we do for RISC-V instructions. Instead, the

instruction must be interrupted and later continued midstream in its execution. Resuming an instruction in the middle of its execution usually requires saving some special state, processing the exception, and restoring that special state. Making this work properly requires careful and detailed coordination between the exception-handling code in the operating system and the hardware.

Elaboration

Rather than pay an extra level of indirection on every memory access, the Virtual Memory Monitor (Section 5.6) maintains a *shadow page table* that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Hence, the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As noted above, the latter happens naturally if accessing the page table pointer is a privileged operation.

Elaboration

The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the expanding diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

Elaboration

In addition to virtualizing the instruction set for a virtual machine, another challenge is virtualization of virtual memory, as each guest OS in every virtual machine manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* (which are often treated synonymously), and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms *virtual memory*, *physical memory*, and *machine memory* to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guest's real memory to physical memory. The virtual memory architecture is typically specified via page tables, as in IBM VM/370, the x86, and RISC-V.

Summary

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and secondary memory. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, virtual memory supports sharing of the main memory among multiple, simultaneously active processes, in a protected manner.

Managing the memory hierarchy between main memory and disk is challenging because of the high cost of page faults. Several techniques are used to reduce the miss rate:

1. Pages are made large to take advantage of spatial locality and to reduce the miss rate.
2. The mapping between virtual addresses and physical addresses, which is implemented with a page table, is made fully associative so that a virtual page can be placed anywhere in main memory.
3. The operating system uses techniques, such as LRU and a reference bit, to choose which pages to replace.

Writes to secondary memory are expensive, so virtual memory uses a write-back scheme and also tracks whether a page is unchanged (using a dirty bit) to avoid writing clean pages.

The virtual memory mechanism provides address translation from a virtual address used by the program to the physical address space used for accessing memory. This address translation allows protected sharing of the main memory and provides several

additional benefits, such as simplifying memory allocation. Ensuring that processes are protected from each other requires that only the operating system can change the address translations, which is implemented by preventing user programs from altering the page tables. Controlled sharing of pages between processes can be implemented with the help of the operating system and access bits in the page table that indicate whether the user program has read or write access to a page.

If a processor had to access a page table resident in memory to translate every access, virtual memory would be too expensive, as caches would be pointless! Instead, a TLB acts as a cache for translations from the page table. Addresses are then translated from virtual to physical using the translations in the TLB.

Caches, virtual memory, and TLBs all rely on a common set of principles and policies. The next section discusses this common framework.

Understanding Program Performance

Although virtual memory was invented to enable a small memory to act as a large one, the performance difference between secondary memory and main memory means that if a program routinely accesses more virtual memory than it has physical memory, it will run very slowly. Such a program would be continuously swapping pages between main memory and secondary memory, called *thrashing*. Thrashing is a disaster if it occurs, but it is rare. If your program thrashes, the easiest solution is to run it on a computer with more memory or buy more memory for your computer. A more complex choice is to re-examine your algorithm and data structures to see if you can change the locality and thereby reduce the number of pages that your program uses simultaneously. This set of popular pages is informally called the *working set*.

A more common performance problem is TLB misses. Since a TLB might handle only 32–64 page entries at a time, a program could easily see a high TLB miss rate, as the processor may access less than a quarter mebibyte directly: 64 × 4 KiB =0.25 MiB. For example, TLB misses are often a challenge for Radix Sort. To try to alleviate this problem, most computer architectures now offer support for larger page sizes. For instance, in addition to the

minimum 4 KiB page, RISC-V hardware supports 2 MiB and 1 GiB pages. Hence, if a program uses large page sizes, it can access more memory directly without TLB misses.

The practical challenge is getting the operating system to allow programs to select these larger page sizes. Once again, the more complex solution to reducing TLB misses is to re-examine the algorithm and data structures to reduce the working set of pages; given the importance of memory accesses to performance and the frequency of TLB misses, some programs with large working sets have been redesigned with that goal.

Elaboration

RISC-V supports the larger page sizes via the multi-level page table of Figure 5.29. In addition to pointing at the next level page table in levels 1 and 2, it allows a *superpage translation* to map the virtual address to a 1 GiB physical address (if the block translation is in level 1) or a 2 MiB physical address (if the block translation is in level 2).

5.8 A Common Framework for Memory Hierarchy

By now, you've recognized that the different types of memory hierarchies have a great deal in common. Although many of the aspects of memory hierarchies differ quantitatively, many of the policies and features that determine how a hierarchy functions are similar qualitatively. Figure 5.34 shows how some of the quantitative characteristics of memory hierarchies can differ. In the rest of this section, we will discuss the common operational alternatives for memory hierarchies, and how these determine their behavior. We will examine these policies in a series of four questions that apply between any two levels of a memory hierarchy, although for simplicity, we will primarily use terminology for caches.

Feature	Typical values for L1 caches	Typical values for L2 caches	Typical values for paged memory	Typical values for a TLB
Total size in blocks	250–2000	2500–25,000	16,000–250,000	40–1024
Total size in kilobytes	16–64	125–2000	1,000,000–1,000,000,000	0.25–16
Block size in bytes	16–64	64–128	4000–64,000	4–32
Miss penalty in clocks	10–25	100–1000	10,000,000–100,000,000	10–1000
Miss rates (global for L2)	2%–5%	0.1%–2%	0.00001%–0.0001%	0.01%–2%

FIGURE 5.34 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer.
These are typical values for these levels as of 2012. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; for example, as caches become larger to overcome larger miss penalties, block sizes also grow. While not shown, server microprocessors today also have L3 caches, which can be 2 to 8 MiB and contain many more blocks than L2 caches. L3 caches lower the L2 miss penalty to 30 to 40 clock cycles.

Question 1: Where Can a Block Be Placed?

We have seen that block placement in the upper level of the hierarchy can use a range of schemes, from direct mapped to set associative to fully associative. As mentioned above, this entire range of schemes can be thought of as variations on a set-associative scheme where the number of sets and the number of blocks per set varies:

Scheme name	Number of sets	Blocks per set
Direct mapped	Number of blocks in cache	1
Set associative	$\frac{\text{Number of blocks in the cache}}{\text{Associativity}}$	Associativity (typically 2–16)
Fully associative	1	Number of blocks in the cache

The advantage of increasing the degree of associativity is that it usually decreases the miss rate. The improvement in miss rate comes from reducing misses that compete for the same location. We will examine these in more detail shortly. First, let's look at how much improvement is gained. Figure 5.35 shows the miss rates for several cache sizes as associativity varies from direct mapped to

eight-way set associative. The largest gains are obtained in going from direct mapped to two-way set associative, which yields between a 20% and 30% reduction in the miss rate. As cache sizes grow, the relative improvement from associativity increases only slightly; since the overall miss rate of a larger cache is lower, the opportunity for improving the miss rate decreases and the absolute improvement in the miss rate from associativity shrinks significantly. The potential disadvantages of associativity, as we mentioned earlier, are increased cost and slower access time.

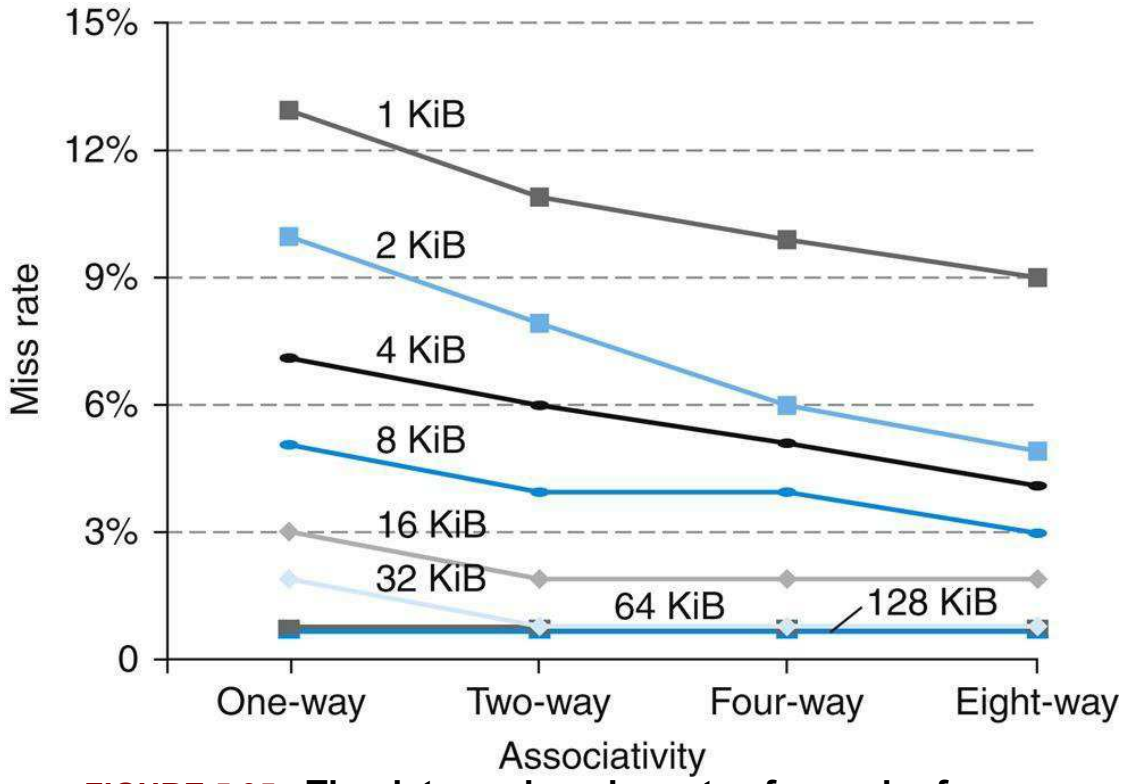


FIGURE 5.35 The data cache miss rates for each of eight cache sizes improve as the associativity increases.

While the benefit of going from one-way (direct mapped) to two-way set associative is significant, the benefits of further associativity are smaller (e.g., 1–10% improvement going from two-way to four-way versus 20–30% improvement going from one-way to two-way). There is even less improvement in going from four-way to eight-way set associative, which, in turn, comes very close to the miss rates of a fully associative cache. Smaller caches obtain a significantly larger absolute benefit from associativity

because the base miss rate of a small cache is larger. Figure 5.16 explains how these data were collected.

Question 2: How Is a Block Found?

The choice of how we locate a block depends on the block placement scheme, since that dictates the number of possible locations. We can summarize the schemes as follows:

Associativity	Location method	Comparisons required
Direct mapped	Index	1
Set associative	Index the set, search among elements	Degree of associativity
Full	Search all cache entries	Size of the cache
	Separate lookup table	0

The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware. Including the L2 cache on the chip enables much higher associativity, because the hit times are not as critical and the designer does not have to rely on standard SRAM chips as the building blocks. Fully associative caches are prohibitive except for small sizes, where the cost of the comparators is not overwhelming and where the absolute miss rate improvements are greatest.

In virtual memory systems, a separate mapping table—the page table—is kept to index the memory. In addition to the storage needed for the table, using an index table requires an extra memory access. The choice of full associativity for page placement and the extra table is motivated by these facts:

1. Full associativity is beneficial, since misses are very expensive.
2. Full associativity allows software to use sophisticated replacement schemes that are designed to reduce the miss rate.
3. The full map can be easily indexed with no extra hardware and no searching required.

Therefore, virtual memory systems almost always use fully associative placement.

Set-associative placement is often used for caches and TLBs, where the access combines indexing and the search of a small set. A few systems have used direct-mapped caches because of their advantage in access time and simplicity. The advantage in access time occurs because finding the requested block does not depend on a comparison. Such design choices depend on many details of the implementation, such as whether the cache is on-chip, the technology used for implementing the cache, and the critical role of cache access time in determining the processor cycle time.

Question 3: Which Block Should Be Replaced on a Cache Miss?

When a miss occurs in an associative cache, we must decide which block to replace. In a fully associative cache, all blocks are candidates for replacement. If the cache is set associative, we must choose among the blocks in the set. Of course, replacement is easy in a direct-mapped cache because there is only one candidate.

There are the two primary strategies for replacement in set-associative or fully associative caches:

- *Random:* Candidate blocks are randomly selected, possibly using some hardware assistance.
- *Least recently used* (LRU): The block replaced is the one that has been unused for the longest time.

In practice, LRU is too costly to implement for hierarchies with more than a small degree of associativity (two to four, typically), since tracking the usage information is expensive. Even for four-way set associativity, LRU is often approximated—for example, by keeping track of which pair of blocks is LRU (which requires 1 bit), and then tracking which block in each pair is LRU (which requires 1 bit per pair).

For larger associativity, either LRU is approximated or random replacement is used. In caches, the replacement algorithm is in hardware, which means that the scheme should be easy to implement. Random replacement is simple to build in hardware, and for a two-way set-associative cache, random replacement has a miss rate about 1.1 times higher than LRU replacement. As the caches become larger, the miss rate for both replacement strategies falls, and the absolute difference becomes small. In fact, random

replacement can sometimes be better than the simple LRU approximations that are easily implemented in hardware.

In virtual memory, some form of LRU is always approximated, since even a tiny reduction in the miss rate can be important when the cost of a miss is enormous. Reference bits or equivalent functionality are often provided to make it easier for the operating system to track a set of less recently used pages. Because misses are so expensive and relatively infrequent, approximating this information primarily in software is acceptable.

Question 4: What Happens on a Write?

A key characteristic of any memory hierarchy is how it deals with writes. We have already seen the two basic options:

- *Write-through:* The information is written to both the block in the cache and the block in the lower level of the memory hierarchy (main memory for a cache). The caches in Section 5.3 used this scheme.
- *Write-back:* The information is written just to the block in the cache. The modified block is written to the lower level of the hierarchy only when it is replaced. Virtual memory systems always use write-back, for the reasons discussed in Section 5.7.

Both write-back and write-through have their advantages. The key advantages of write-back are the following:

- Individual words can be written by the processor at the rate that the cache, rather than the memory, can accept them.
- Multiple writes within a block require only one write to the lower level in the hierarchy.
- When blocks are written back, the system can make effective use of a high-bandwidth transfer, since the entire block is written.

Write-through has these advantages:

- Misses are simpler and cheaper because they never require a block to be written back to the lower level.
- Write-through is easier to implement than write-back, although to be realistic, a write-through cache will still need to use a write buffer.

The BIG Picture

Caches, TLBs, and virtual memory may initially look very

different, but they rely on the same two principles of locality, and they can be understood by their answers to four questions:

Question 1:	Where can a block be placed?
Answer:	One place (direct mapped), a few places (set associative), or any place (fully associative).
Question 2:	How is a block found?
Answer:	There are four methods: indexing (as in a direct-mapped cache), limited search (as in a set-associative cache), full search (as in a fully associative cache), and a separate lookup table (as in a page table).
Question 3:	What block is replaced on a miss?
Answer:	Typically, either the least recently used or a random block.
Question 4:	How are writes handled?
Answer:	Each level in the hierarchy can use either write-through or write-back.

In virtual memory systems, only a write-back policy is practical because of the long latency of a write to the lower level of the hierarchy. The rate at which writes are generated by a processor generally exceeds the rate at which the memory system can process them, even allowing for physically and logically wider memories and burst modes for DRAM. Consequently, today lowest-level caches typically use write-back.

The Three Cs: An Intuitive Model for Understanding the Behavior of Memory Hierarchies

In this subsection, we look at a model that provides insight into the sources of misses in a memory hierarchy and how the misses will be affected by changes in the hierarchy. We will explain the ideas in terms of caches, although the ideas carry over directly to any other level in the hierarchy. In this model, all misses are classified into one of three categories (the **three Cs**):

three Cs model

A cache model in which all cache misses are classified into one of three categories: compulsory misses, capacity misses, and conflict misses.

- **Compulsory misses**: These are cache misses caused by the first access to a block that has never been in the cache. These are also called **cold-start misses**.
- **Capacity misses**: These are cache misses caused when the cache cannot contain all the blocks needed during execution of a program. Capacity misses occur when blocks are replaced and then later retrieved.
- **Conflict misses**: These are cache misses that occur in set-associative or direct-mapped caches when multiple blocks compete for the same set. Conflict misses are those misses in a direct-mapped or set-associative cache that are eliminated in a fully associative cache of the same size. These cache misses are also called **collision misses**.

compulsory miss

Also called **cold-start miss**. A cache miss caused by the first access to a block that has never been in the cache.

capacity miss

A cache miss that occurs because the cache, even with full associativity, cannot contain all the blocks needed to satisfy the request.

conflict miss

Also called **collision miss**. A cache miss that occurs in a set-associative or direct-mapped cache when multiple blocks compete for the same set and that are eliminated in a fully associative cache of the same size.

Figure 5.36 shows how the miss rate divides into the three sources. These sources of misses can be directly attacked by changing some aspect of the cache design. Since conflict misses arise straight from contention for the same cache block, increasing associativity reduces conflict misses. Associativity, however, may slow access time, leading to lower overall performance.

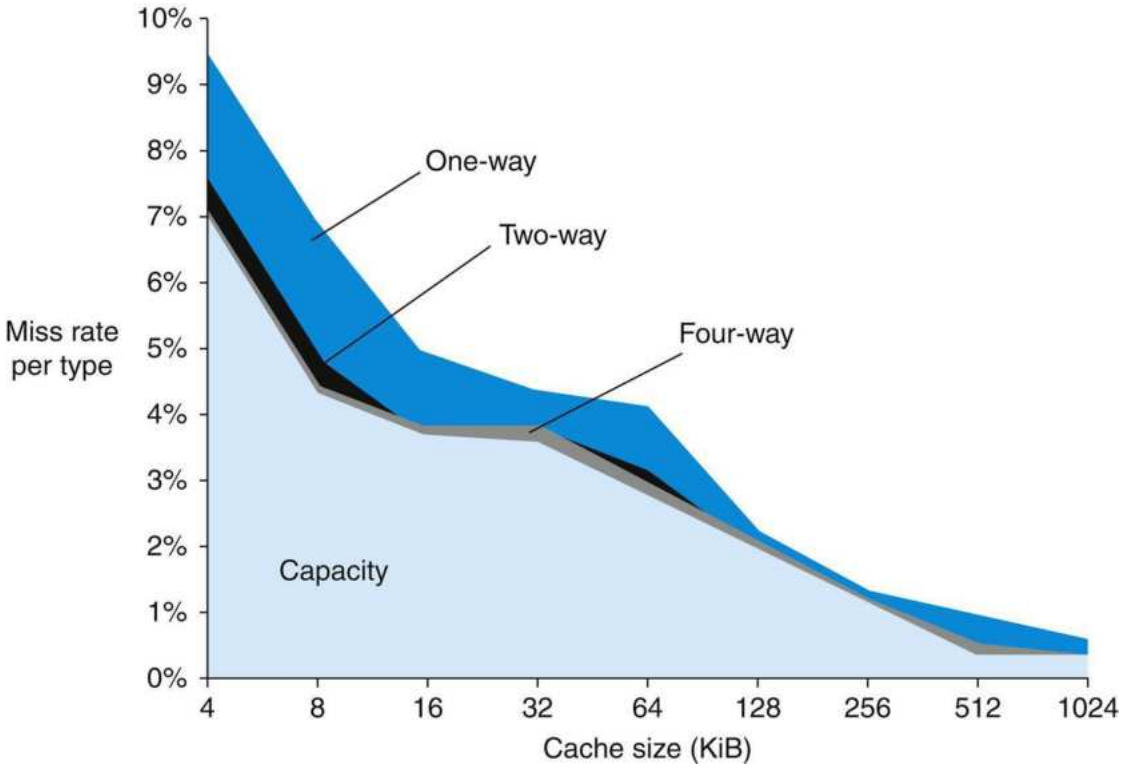


FIGURE 5.36 The miss rate can be broken into three sources of misses.
This graph shows the total miss rate and its components for a range of cache sizes. These data are for the SPEC CPU2000 integer and floating-point benchmarks and are from the same source as the data in Figure 5.35. The compulsory miss component is 0.006% and cannot be seen in this graph. The next component is the capacity miss rate, which depends on cache size. The conflict portion, which depends both on associativity and on cache size, is shown for a range of associativities from one-way to eight-way. In each case, the labeled section corresponds to the increase in the miss rate that occurs when the associativity is changed from the next higher degree to the labeled degree of associativity. For example, the section labeled *two-way* indicates the additional misses arising when the cache has associativity of two rather than four. Thus, the difference in the miss rate incurred by a direct-mapped cache versus a fully associative cache of the same size is given by the sum of the sections marked *four-way, two-way*, and *one-way*. The difference between eight-way and four-way is so small that it is difficult to see on this graph.

Capacity misses can easily be reduced by enlarging the cache;

indeed, second-level caches have been growing steadily bigger for many years. Of course, when we make the cache larger, we must also be careful about increasing the access time, which could lead to lower overall performance. Thus, first-level caches have been growing slowly, if at all.

Because compulsory misses are generated by the first reference to a block, the primary way for the cache system to reduce the number of compulsory misses is to increase the block size. This will reduce the number of references required to touch each block of the program once, because the program will consist of fewer cache blocks. As mentioned above, increasing the block size too much can have a negative effect on performance because of the increase in the miss penalty.

The BIG Picture

The challenge in designing memory hierarchies is that every change that potentially improves the miss rate can also negatively affect overall performance, as Figure 5.37 summarizes. This combination of positive and negative effects is what makes the design of a memory hierarchy interesting.

Design change	Effect on miss rate	Possible negative performance effect
Increases cache size	Decreases capacity misses	May increase access time
Increases associativity	Decreases miss rate due to conflict misses	May increase access time
Increases block size	Decreases miss rate for a wide range of block sizes due to spatial locality	Increases miss penalty. Very large block could increase miss rate

FIGURE 5.37 Memory hierarchy design challenges.

The decomposition of misses into the three Cs is a useful qualitative model. In real cache designs, many of the design choices interact, and changing one cache characteristic will often affect several components of the miss rate. Despite such shortcomings, this model is a useful way to gain insight into the performance of cache designs.

Check Yourself

Which of the following statements (if any) is generally true?
1. There is no way to reduce compulsory misses.
2. Fully associative caches have no conflict misses.
3. In reducing misses, associativity is more important than capacity.

5.9 Using a Finite-State Machine to Control a Simple Cache

We can now build control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in Chapter 4. This section starts with a definition of a simple cache and then a description of *finite-state machines* (FSMs). It finishes with the FSM of a controller for this simple cache. Section 5.12 goes into more depth, showing the cache and controller in a new hardware description language.

A Simple Cache

We're going to design a controller for a straightforward cache. Here are the key characteristics of the cache:

- Direct-mapped cache
- Write-back using write allocate
- Block size is four words (16 bytes or 128 bits)
- Cache size is 16 KiB, so it holds 1024 blocks
- 32-bit addresses
- The cache includes a valid bit and dirty bit per block

From Section 5.3, we can now calculate the fields of an address for the cache:

- Cache index is 10 bits
- Block offset is 4 bits
- Tag size is 32−(10+4) or 18 bits

The signals between the processor to the cache are

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a cache operation or not
- 32-bit address
- 32-bit data from processor to cache

- 32-bit data from cache to processor
- 1-bit Ready signal, saying the cache operation is complete

The interface between the memory and the cache has the same fields as between the processor and the cache, except that the data fields are now 128 bits wide. The extra memory width is generally found in microprocessors today, which deal with either 32-bit or 64-bit words in the processor while the DRAM controller is often 128 bits. Making the cache block match the width of the DRAM simplified the design. Here are the signals:

- 1-bit Read or Write signal
- 1-bit Valid signal, saying whether there is a memory operation or not
- 32-bit address
- 128-bit data from cache to memory
- 128-bit data from memory to cache
- 1-bit Ready signal, saying the memory operation is complete

Note that the interface to memory is not a fixed number of cycles. We assume a memory controller that will notify the cache via the Ready signal when the memory read or write is finished.

Before describing the cache controller, we need to review finite-state machines, which allow us to control an operation that can take multiple clock cycles.

Finite-State Machines

To design the control unit for the single-cycle datapath, we used truth tables that specified the setting of the control signals based on the instruction class. For a cache, the control is more complex because the operation can be a series of steps. The control for a cache must specify both the signals to be set in any step and the next step in the sequence.

The most common multistep control method is based on **finite-state machines**, which are usually represented graphically. A finite-state machine consists of a set of states and directions on how to change states. The directions are defined by a **next-state function**, which maps the current state and the inputs to a new state. When we use a finite-state machine for control, each state also specifies a set of outputs that are asserted when the machine is in that state. The implementation of a finite-state machine usually

assumes that all outputs that are not explicitly asserted are deasserted. Similarly, the correct operation of the datapath depends on the fact that a signal that is not explicitly asserted is deasserted, rather than acting as a don't care.

finite-state machine

A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function

A combinational function that, given the inputs and the current state, determines the next state of a finite-state machine.

Multiplexor controls are slightly different, since they select one of the inputs, whether they are 0 or 1. Thus, in the finite-state machine, we always specify the setting of all the multiplexor controls that we care about. When we implement the finite-state machine with logic, setting a control to 0 may be the default and therefore may not require any gates. A simple example of a finite-state machine appears in Appendix A, and if you are unfamiliar with the concept of a finite-state machine, you may want to examine Appendix A before proceeding.

A finite-state machine can be implemented with a temporary register that holds the current state and a block of combinational logic that determines both the data-path signals to be asserted and the next state. Figure 5.38 shows how such an implementation might look. Appendix C describes in detail how the finite-state machine is implemented using this structure. In Section A.3, the combinational control logic for a finite-state machine is implemented both with either a ROM (*read-only memory*) or a PLA (*programmable logic array*). (Also see Appendix A for a description of these logic elements.)

Elaboration

Note that this simple design is called a *blocking* cache, in that the processor must wait until the cache has finished the request. Section 5.12 describes the alternative, which is called a *nonblocking* cache.

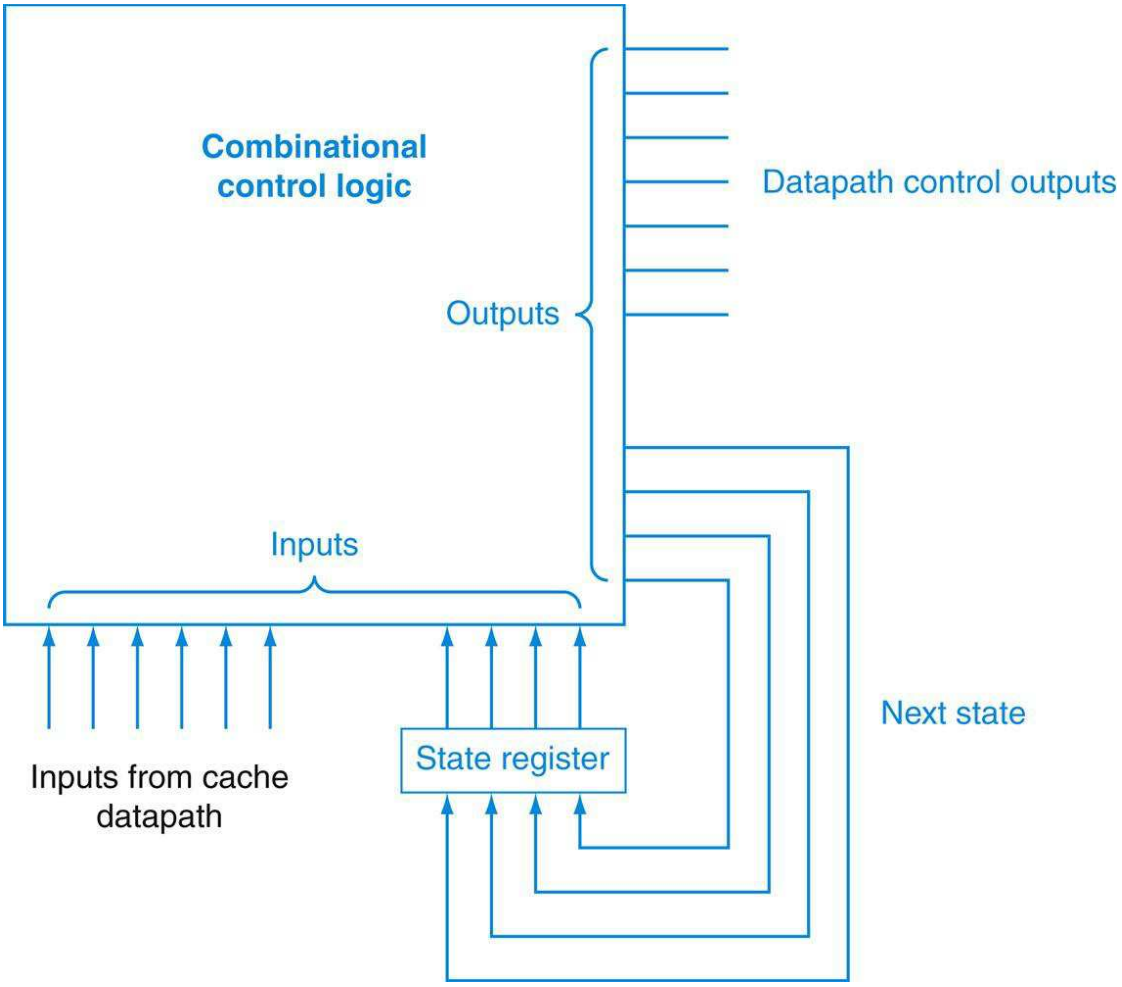


FIGURE 5.38 Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state.
The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. We use color to indicate that these are control lines and logic versus data lines and logic. The *Elaboration* below explains this in more detail.

Elaboration

The style of finite-state machine in this book is called a Moore machine, after Edward Moore. Its identifying characteristic is that

the output depends only on the current state. For a Moore machine, the box labeled combinational control logic can be split into two pieces. One piece has the control output and only the state input, while the other has just the next-state output.

An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix A, when the implementation of this finite-state machine is taken down to logic gates, the size advantage can be clearly seen. The potential disadvantage of a Moore machine is that it may require additional states. For example, in situations where there is a one-state difference between two sequences of states, the Mealy machine may unify the states by making the outputs depend on the inputs.

FSM for a Simple Cache Controller

Figure 5.39 shows the four states of our simple cache controller:

- *Idle:* This state waits for a valid read or write request from the processor, which moves the FSM to the Compare Tag state.
- *Compare Tag:* As the name suggests, this state tests to see if the requested read or write is a hit or a miss. The index portion of the address selects the tag to be compared. If the data in the cache block referred to by the index portion of the address are valid, and the tag portion of the address matches the tag, then it is a hit. Either the data are read from the selected word if it is a load or written to the selected word if it is a store. The Cache Ready signal is then set. If it is a write, the dirty bit is set to 1. Note that a write hit also sets the valid bit and the tag field; while it seems unnecessary, it is included because the tag is a single memory, so to change the dirty bit we likewise need to change the valid and tag fields. If it is a hit and the block is valid, the FSM returns to the idle state. A miss first updates the cache tag and then goes either to the Write-Back state, if the block at this location has dirty bit value of 1, or to the Allocate state if it is 0.
- *Write-Back:* This state writes the 128-bit block to memory using the

address composed from the tag and cache index. We remain in this state waiting for the Ready signal from memory. When the memory write is complete, the FSM goes to the Allocate state.

- *Allocate:* The new block is fetched from memory. We remain in this state waiting for the Ready signal from memory. When the memory read is complete, the FSM goes to the Compare Tag state. Although we could have gone to a new state to complete the operation instead of reusing the Compare Tag state, there is a good deal of overlap, including the update of the appropriate word in the block if the access was a write.

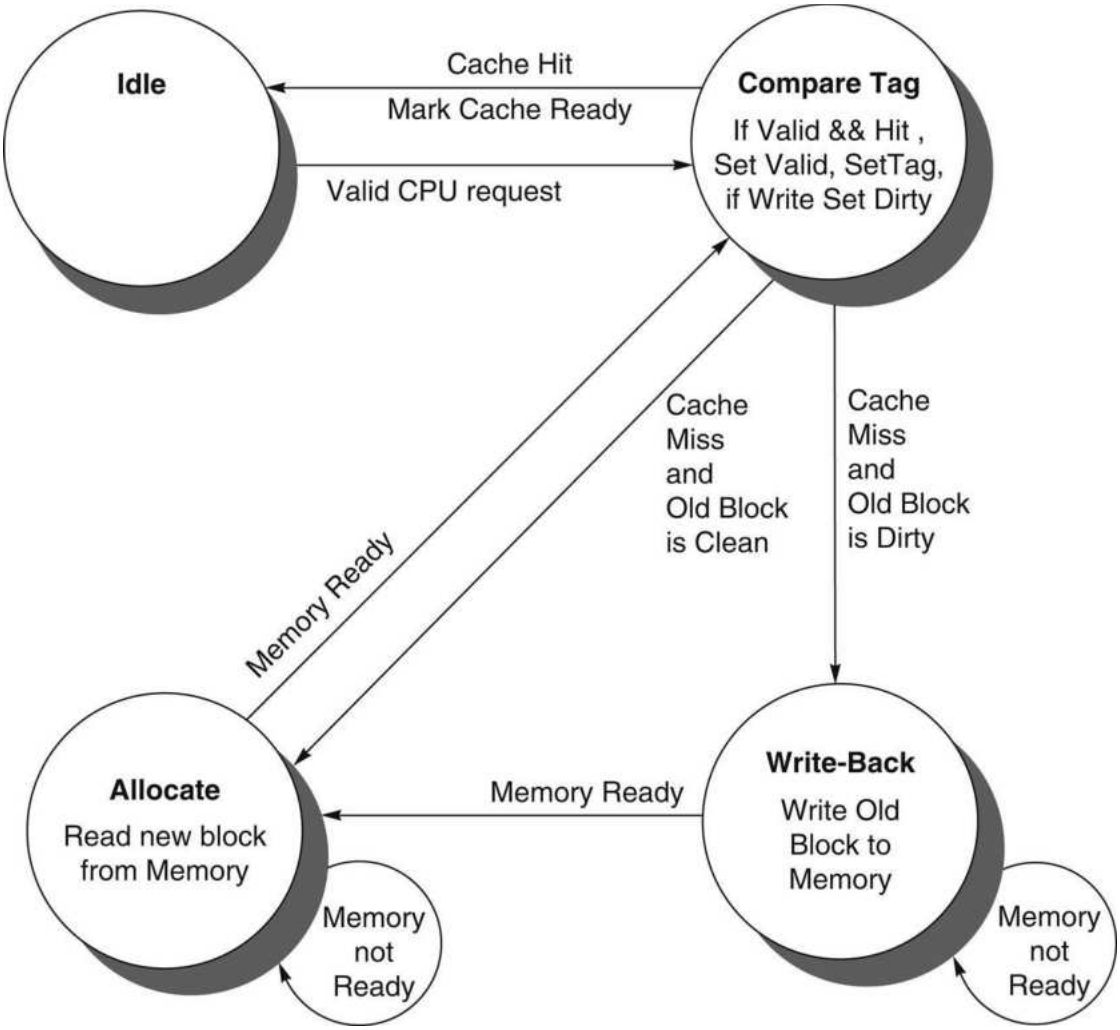


FIGURE 5.39 Four states of the simple controller.

This simple model could easily be extended with more states to try to improve performance. For example, the Compare Tag state does both the compare and the read or write of the cache data in a single clock cycle. Often the compare and cache access are done in separate states to try to improve the clock cycle time. Another optimization would be to add a write buffer so that we could save the dirty block and then read the new block first so that the processor doesn't have to wait for two memory accesses on a dirty miss. The cache would next write the dirty block from the write buffer while the processor is operating on the requested data.

Section 5.12 goes into more detail about the FSM, showing the full controller in a hardware description language and a block diagram of this simple cache.

5.10 Parallelism and Memory Hierarchy: Cache Coherence

Given that a multicore multiprocessor means multiple processors on a single chip, these processors very likely share a common physical address space. Caching shared data introduces a new problem, because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two distinct values. Figure 5.40 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty is generally referred to as the *cache coherence problem*.

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

FIGURE 5.40 The cache coherence problem for a single memory location (X), read and written by two processors (A and B).
We initially assume that neither cache contains the variable and that X has the value 0. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 0!

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared memory programs. The first aspect, called *coherence*, defines *what values* can be returned by a read. The second aspect, called

consistency, determines *when* a written value will be returned by a read.

Let's look at coherence first. A memory system is coherent if

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P. Thus, in Figure 5.40, if CPU A were to read X after time step 3, it should see the value 1.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses. Thus, in Figure 5.40, we need a mechanism so that the value 0 in the cache of CPU B is replaced by the value 1 after CPU A stores 1 into memory at address X in time step 3.
3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if CPU B stores 2 into memory at address X after time step 3, processors can never read the value at location X as 2 and then later read it as 1.

The first property simply preserves program order—we certainly expect this property to be true in uniprocessors, for instance. The second property defines the notion of what it means to have a coherent view of memory: if a processor could continuously read an old data value, we would clearly say that memory was incoherent.

The need for *write serialization* is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the identical order, which we call *write serialization*.

Basic Schemes for Enforcing Coherence

In a cache coherent multiprocessor, the caches provide both

migration and *replication* of shared data items:

- *Migration:* A data item can be moved to a local cache and used there in a transparent fashion. Migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.
- *Replication:* When shared data are being simultaneously read, the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

Supporting migration and replication is critical to performance in accessing shared data, so many multiprocessors introduce a hardware protocol to maintain coherent caches. The protocols to maintain coherence for multiple processors are called *cache coherence protocols*. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block.

The most popular cache coherence protocol is *snooping*. Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or network), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

In the following section we explain snooping-based cache coherence as implemented with a shared bus, but any communication medium that broadcasts cache misses to all processors can be used to implement a snooping-based coherence scheme. This broadcasting to all caches makes snooping protocols simple to implement but also limits their scalability.

Snooping Protocols

One method of enforcing coherence is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a *write invalidate protocol* because it invalidates copies in other caches on a write. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

Figure 5.41 shows an example of an invalidation protocol for a snooping bus with write-back caches in action. To see how this

protocol ensures coherence, consider a write followed by a read by another processor: since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache, and the cache is forced to fetch a new copy of the data. For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data at the same time, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol also enforces write serialization.

Hardware/Software Interface

One insight is that block size plays an important role in cache coherency. For example, take the case of snooping on a cache with a block size of eight words, with a single word alternatively written and read by two processors. Most protocols exchange full blocks between processors, thereby increasing coherency bandwidth demands.

Large blocks can also cause what is called **false sharing**: when two unrelated shared variables are located in the same cache block, the whole block is exchanged between processors even though the processors are accessing different variables. Programmers and compilers should lay out data carefully to avoid false sharing.

false sharing

When two unrelated shared variables are located in the same cache block and the full block is exchanged between processors even though the processors are accessing different variables.

Elaboration

Although the three properties on page 455 are sufficient to ensure coherence, the question of when a written value is seen is also important. To see why, observe that we cannot require that a read of X in Figure 5.40 instantaneously sees the value written for X by

some other processor. If, for example, a write of X on one processor precedes a read of X on another processor very shortly beforehand, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*.

We make the following two assumptions. First, a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write. Second, the processor does not change the order of any write with respect to any other memory access. These two conditions mean that if a processor writes location X followed by location Y, any processor that sees the new value of Y must also see the new value of X. These restrictions allow the processor to reorder reads, but force the processor to finish a write in program order.

Elaboration

Since input can change memory behind the caches, and since output could need the latest value in a write back cache, there is also a cache coherency problem for I/O with the caches of a single processor as well as just between caches of multiple processors. The cache coherence problem for multiprocessors and I/O (see Chapter 6), although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches.

Elaboration

In addition to the snooping cache coherence protocol where the status of shared blocks is distributed, a *directory-based* cache coherence protocol keeps the sharing status of a block of physical memory in just one location, called the *directory*. Directory-based coherence has slightly higher implementation overhead than snooping, but it can reduce traffic between caches and thus scale to larger processor counts.

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 5.41 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.
We assume that neither cache initially holds X and that the value of X in memory is 0. The CPU and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, CPU A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it.

5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks

This online section describes how using many disks in conjunction can offer much higher throughput, which was the original inspiration of *Redundant Arrays of Inexpensive Disks* (RAID). The real popularity of RAID, however, was due more to the considerably greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and **dependability** between the RAID levels.

5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks

Amdahl's law in Chapter 1 reminds us that neglecting I/O in this parallel revolution is foolhardy. A simple example demonstrates this.

Impact of I/O on System Performance

Example

Suppose we have a benchmark that executes in 100 seconds of elapsed time, of which 90 seconds is CPU time, and the rest is I/O time. Suppose the number of processors doubles every 2 years, but the processors remain at the same speed, and I/O time doesn't improve. How much faster will our program run at the end of 6 years?

Answer

We know that

$$\begin{aligned} \text{Elapsed time} &= \text{CPU time} + \text{I/O time} \\ 100 &= 90 + \text{I/O time} \\ \text{I/O time} &= 10 \text{ seconds} \end{aligned}$$

The new CPU times and the resulting elapsed times are computed in the following table.

After *n* years	CPU time	I/O time	Elapsed time	% I/O time
0 years	90 seconds	10 seconds	100 seconds	10%
2 years	$\frac{90}{2} = 45$ seconds	10 seconds	55 seconds	18%
4 years	$\frac{45}{2} = 23$ seconds	10 seconds	33 seconds	31%
6 years	$\frac{23}{2} = 11$ seconds	10 seconds	21 seconds	47%

The improvement in CPU performance after 6 years is

$$\frac{90}{11} = 8$$

However, the improvement in elapsed time is only

$$\frac{100}{21} = 4.7$$

and the I/O time has increased from 10% to 47% of the elapsed time.

Hence, the parallel revolution needs to come to I/O as well as to computation, or the effort spent in parallelizing could be squandered whenever programs do I/O, which they all must do.

Accelerating I/O performance was the original motivation of disk arrays. In the late 1980s, the high-performance storage of choice was

large, expensive disks. The argument was that by replacing a few big disks with many small disks, performance would improve because there would be more read heads. This shift is a good match for multiple processors as well, since many read/write heads mean the storage system could support many more independent accesses as well as large transfers spread across many disks. That is, you could get both high I/Os per second and high data transfer rates. In addition to higher performance, there could be advantages in cost, power, and floor space, since smaller disks are generally more efficient per gigabyte than larger disks.

The flaw in the argument was that disk arrays could make reliability much worse. These smaller, inexpensive drives had lower MTTF ratings than the large drives, but more importantly, by replacing a single drive with, say, 50 small drives, the failure rate would go up by at least a factor of 50.

The solution was to add redundancy so that the system could cope with disk failures without losing information. By having many little disks, the cost of extra redundancy to improve dependability is small, relative to the solutions for a few large disks. Thus, **dependability** was more affordable if you constructed a redundant array of inexpensive disks. This observation led to its name: **redundant arrays of inexpensive disks**, abbreviated **RAID**.

redundant arrays of inexpensive disks (RAID)

An organization of disks that uses an array of small and inexpensive disks so as to increase both performance and reliability.

In retrospect, although its invention was motivated by performance, dependability was the key reason for the widespread popularity of RAID. The parallel revolution has resurfaced the original performance side of the argument for RAID. The rest of this section surveys the options for dependability and their impacts on cost and performance.

How much redundancy do you need? Do you need extra information to find the faults? Does it matter how you organize the data and the additional check information on these disks? The paper that coined the term gave an evolutionary answer to these questions, starting with the simplest but most expensive solution. Figure e5.11.1 shows the evolution and example cost in the number of extra check disks. To keep track of the evolution, the authors numbered the stages of RAID, and they are still used today.

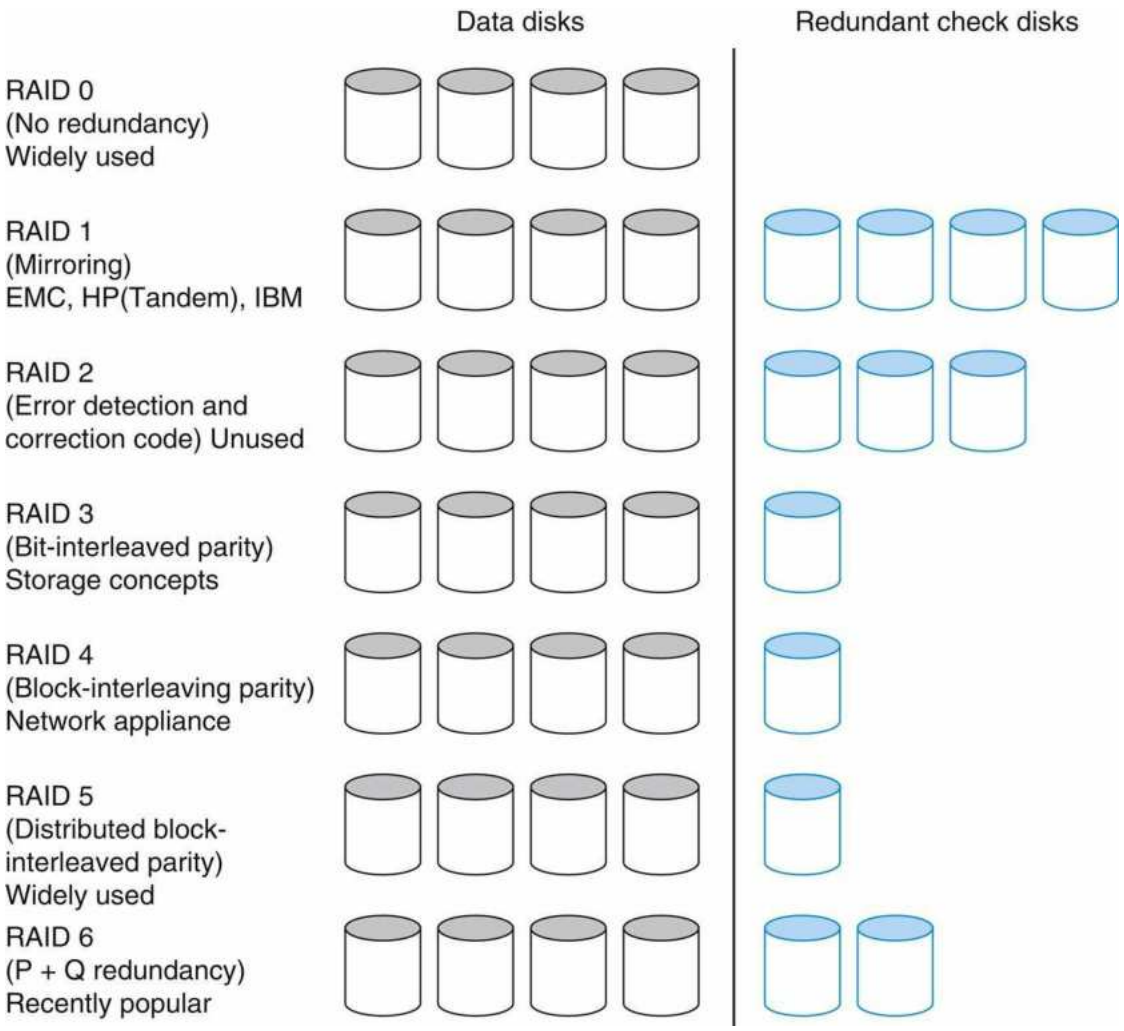


FIGURE E5.11.1 RAID for an example of four data disks showing extra check disks per RAID level and companies that use each level.
Figures e5.11.2 and e5.11.3 explain the difference between RAID 3, RAID 4, and RAID 5.

No Redundancy (RAID 0)

Simply spreading data over multiple disks, called **striping**, automatically forces accesses to several disks. Striping across a set of disks makes the collection appear to software as a single large disk, which simplifies storage management. It also improves performance for large accesses, since many disks can operate at once. Video-editing systems, for example, frequently stripe their data and may not worry about dependability as much as, say, databases.

striping

Allocation of logically sequential blocks to separate disks to allow higher performance than a single disk can deliver.

RAID 0 is something of a misnomer, as there is no redundancy. However, RAID levels are often left to the operator to set when creating a storage system, and RAID 0 is often listed as one of the options. Hence, the term RAID 0 has become widely used.

Mirroring (RAID 1)

This traditional scheme for tolerating disk failure, called **mirroring** or *shadowing*, uses twice as many disks as does RAID 0. Whenever data are written to one disk, that data are also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" and reads its contents to get the desired information. Mirroring is the most expensive RAID solution, since it requires the most disks.

mirroring

Writing identical data to multiple disks to increase data availability.

Error Detecting and Correcting Code (RAID 2)

RAID 2 borrows an error detection and correction scheme most often used for memories (see Section 5.5). Since RAID 2 has fallen into disuse, we'll not describe it here.

Bit-Interleaved Parity (RAID 3)

The cost of higher availability can be reduced to $1/n$, where n is the number of disks in a **protection group**. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure. RAID 3

is popular in applications with large data sets, such as multimedia and some scientific codes.

protection group

The group of data disks or blocks that share a common check disk or block.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo two.

Unlike RAID 1, many disks must be read to determine the missing data. The assumption behind this technique is that taking longer to recover from failure but spending less on redundant storage is a good tradeoff.

Block-Interleaved Parity (RAID 4)

RAID 4 uses the same ratio of data disks and check disks as RAID 3, but they access data differently. The parity is stored as blocks and associated with a set of data blocks.

In RAID 3, every access went to all disks. However, some applications prefer smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the RAID levels 4 to 7. Since error detection information in each sector is checked on reads to see if the data are correct, such "small reads" to each disk can occur independently as long as the minimum access is one sector. In the RAID context, a small access goes to just one disk in a protection group while a large access goes to all the disks in a protection group.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in the left in Figure e5.11.2. A "small write" would require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.

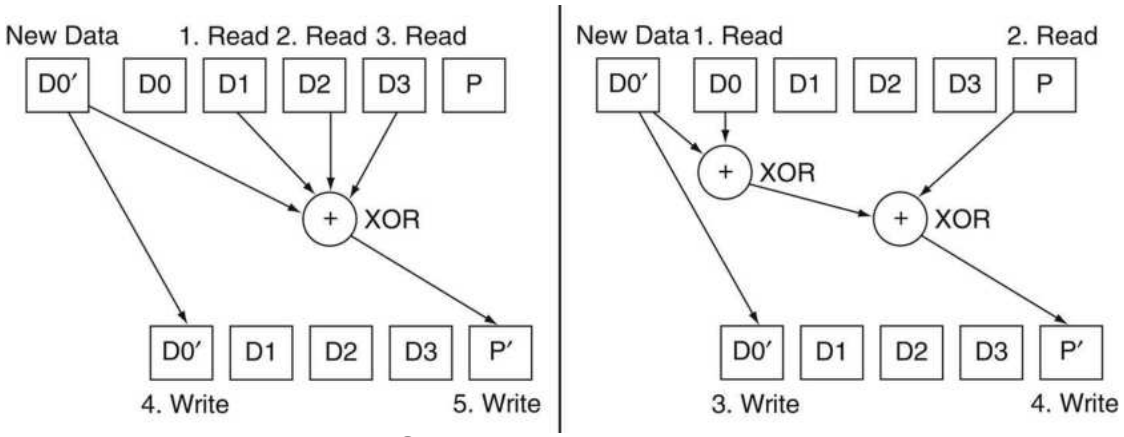


FIGURE E5.11.2 Small write update on RAID 4. This optimization for small writes reduces the number of disk accesses as well as the number of disks occupied. This figure assumes we have four blocks of data and one block of parity. The naive RAID 4 parity calculation in the left of the figure reads blocks D1, D2, and D3 before adding block D0? to calculate the new parity P? (In case you were wondering, the new data D0? comes directly from the CPU, so disks are not involved in reading it.) The RAID 4 shortcut on the right reads the old value D0 and compares it to the new value D0? to see which bits will change. You next read the old parity P and then change the corresponding bits to form P? The logical function exclusive OR does exactly what we want. This example replaces three disk reads (D1, D2, D3) and two disk writes (D0?, P?) involving all the disks for two disk reads (D0, P) and two disk writes (D0?, P?), which involve just two disks. Enlarging the size of the parity group increases the savings of the shortcut. RAID 5 uses the same shortcut.

The key insight to reduce this overhead is that parity is simply a sum of information; by watching which bits change when we write the new information, we need only change the corresponding bits on the parity disk. The right of Figure e5.11.2 shows the shortcut. We must read the old data from the disk being written, compare old data to the new data to see which bits change, read the old parity, change the corresponding bits, and then write the new data and new parity. Thus, the small write involves four disk accesses to two disks instead of accessing all disks. This organization is RAID 4.

Distributed Block-Interleaved Parity (RAID 5)

RAID 4 efficiently supports a mixture of large reads, large writes, and small reads, plus it allows small writes. One drawback to the system is that the parity disk must be updated on every write, so the parity disk is the bottleneck for back-to-back writes.

To fix the parity-write bottleneck, the parity information can be spread throughout all the disks so that there is no single bottleneck for writes. The distributed parity organization is RAID 5.

Figure e5.11.3 shows how data are distributed in RAID 4 versus RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the parity blocks are not located on the same disk. For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur concurrently with the write to block 8. Those same writes to the organization on the left result in changes to blocks P1 and P2, both on the fifth disk, which is a bottleneck.

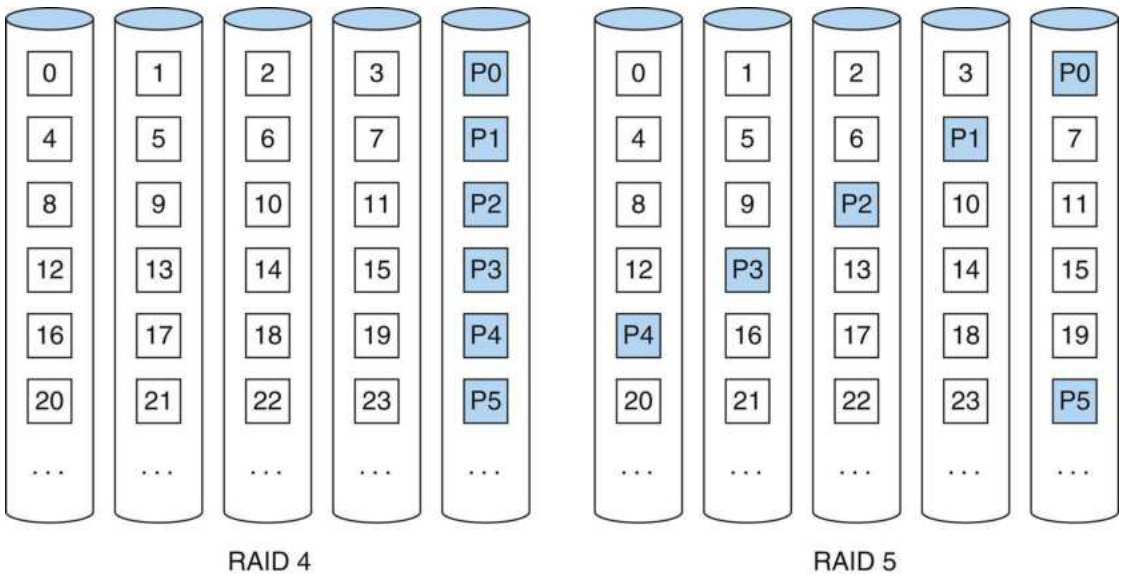


FIGURE E5.11.3 Block-interleaved parity (RAID 4) versus distributed block-interleaved parity (RAID 5).

By distributing parity blocks to all disks, some small writes can be performed in parallel.

P+Q Redundancy (RAID 6)

Parity-based schemes protect against a single self-identifying failure. When a single failure correction is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. This second check block allows recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure e5.11.2 works as well, except now there are six disk accesses instead of four to update both P and Q information.

RAID Summary

RAID 1 and RAID 5 are widely used in servers; one estimate is that 80% of disks in servers are found in a RAID organization.

One weakness of the RAID systems is repair. First, to avoid making the data unavailable during repair, the array must be designed to allow the failed disks to be replaced without having to turn off the system. RAIDs have enough redundancy to allow continuous operation, but **hot-swapping** disks place demands on the physical and electrical design of the array and the disk interfaces. Second, another failure could occur during repair, so the repair time affects the chances of losing data: the longer the repair time, the greater the chances of another failure that will lose data. Rather than having to wait for the operator to bring in a good disk, some systems include **standby spares** so that the data can be reconstructed instantly upon discovery of the failure. The operator can then replace the failed disks in a more leisurely fashion. Note that a human operator ultimately determines which disks to remove. Operators are only human, so they occasionally remove the good disk instead of the broken disk, leading to an unrecoverable disk failure.

hot-swapping

Replacing a hardware component while the system is running.

standby spares

Reserve hardware resources that can immediately take the place of a failed component.

In addition to designing the RAID system for repair, there are questions about how disk technology changes over time. Although disk manufacturers quote very high MTTF for their products, those numbers are under nominal conditions. If a particular disk array has been subject to temperature cycles due to, say, the failure of the air-conditioning system, or to shaking due to a poor rack design, construction, or installation, the failure rates can be three to six times higher (see the fallacy on page 470). The calculation of RAID reliability assumes independence between disk failures, but disk failures could be correlated, because such damage due to the environment would likely happen to all the disks in the array. Another concern is that since disk bandwidth is growing more slowly than disk capacity, the time to repair a disk in a RAID system is increasing, which in turn enhances the chances of a second failure. For example, a 3-TB disk could take almost 9 hours to read sequentially, assuming no interference. Given that the damaged RAID is likely to continue to serve data, reconstruction could be stretched considerably. Besides increasing that time, another concern is that reading much more data during reconstruction means increasing the chance of an uncorrectable read media failure, which would result in data loss. Other arguments for concern about simultaneous multiple failures are the increasing number of disks in arrays and the use of higher-capacity disks.

Hence, these trends have led to a growing interest in protecting against more than one failure, and so RAID 6 is increasingly being offered as an option and being used in the field.

Check Yourself

Which of the following are true about RAID levels 1, 3, 4, 5, and 6?

1. RAID systems rely on redundancy to achieve high availability.
2. RAID 1 (mirroring) has the highest check disk overhead.
3. For small writes, RAID 3 (bit-interleaved parity) has the worst throughput.
4. For large writes, RAID 3, 4, and 5 have the same throughput.

Elaboration

One issue is how mirroring interacts with striping. Suppose you had, say, four disks' worth of data to store and eight physical disks to use. Would you create four pairs of disks—each organized as RAID 1—and then stripe data across the four RAID 1 pairs? Alternatively, would you create two sets of four disks—each organized as RAID 0—and then mirror writes to both RAID 0 sets? The RAID terminology has evolved to call the former RAID 1+0 or RAID 10 ("striped mirrors") and the latter RAID 0+1 or RAID 01 ("mirrored stripes").

Advanced Material: Implementing Cache Controllers

This online section shows how to implement control for a cache, just as we implemented control for the single-cycle and pipelined datapaths in Chapter 4. This section starts with a description of finite-state machines and the implementation of a cache controller for a simple data cache, including a description of the cache controller in a hardware description language. It then goes into details of an example cache coherence protocol and the difficulties in implementing such a protocol.

5.12 Advanced Material: Implementing Cache Controllers

The section starts with the SystemVerilog of the cache controller from Section 5.9 in eight figures. It then goes into details of an example cache coherency protocol and the difficulties in

implementing such a protocol.

SystemVerilog of a Simple Cache Controller

The hardware description language we are using in this section is SystemVerilog. The biggest change from prior versions of Verilog is that it borrows structures from C to make the code easier to read. Figures e5.12.1 through e5.12.8 show the SystemVerilog description of the cache controller.

```
package cache_def;
  // data structures for cache tag & data

  parameter int TAGMSB = 31;     //tag msb
  parameter int TAGLSB = 14;     //tag lsb

  //data structure for cache tag
  typedef struct packed {
    bit    valid;                //valid bit
    bit    dirty;                //dirty bit
    bit [TAGMSB:TAGLSB]tag;      //tag bits
  }cache_tag_type;

  //data structure for cache memory request
  typedef struct {
    bit [9:0]index;              //10-bit index
    bit    we;                   //write enable
  }cache_req_type;

  //128-bit cache line data
  typedef bit [127:0]cache_data_type;
```

FIGURE E5.12.1 Type declarations in SystemVerilog for the cache tags and data.
The tag field is 18 bits wide and the index field is 10 bits wide, while a 2-bit field (bits 3–2) is used to index the block and select the word from the block. The rest of the type declaration is found in the following figure.

```
  // data structures for CPU<->Cache controller interface

  // CPU request (CPU->cache controller)
  typedef struct {
    bit [31:0]addr;             //32-bit request addr
    bit [31:0]data;             //32-bit request data (used when write)
    bit rw;                     //request type : 0 = read, 1 = write
    bit valid;                  //request is valid
  }cpu_req_type;

  // Cache result (cache controller->cpu)
  typedef struct {
    bit [31:0]data;             //32-bit data
    bit ready;                  //result is ready
  }cpu_result_type;

  //-----------------------------------------------------------------------
  // data structures for cache controller<->memory interface

  // memory request (cache controller->memory)
  typedef struct {
    bit [31:0]addr;             //request byte addr
    bit [127:0]data;            //128-bit request data (used when write)
    bit rw;                     //request type : 0 = read, 1 = write
    bit valid;                  //request is valid
  }mem_req_type;

  // memory controller response (memory -> cache controller)
  typedef struct {
    cache_data_type data;       //128-bit read back data
    bit    ready;               //data is ready
  }mem_data_type;

endpackage
```

FIGURE E5.12.2 Type declarations in SystemVerilog for the CPU-cache and cache-memory interfaces. These are nearly identical except that the data are 32 bits wide between the CPU and cache and are 128 bits wide between the cache and memory.

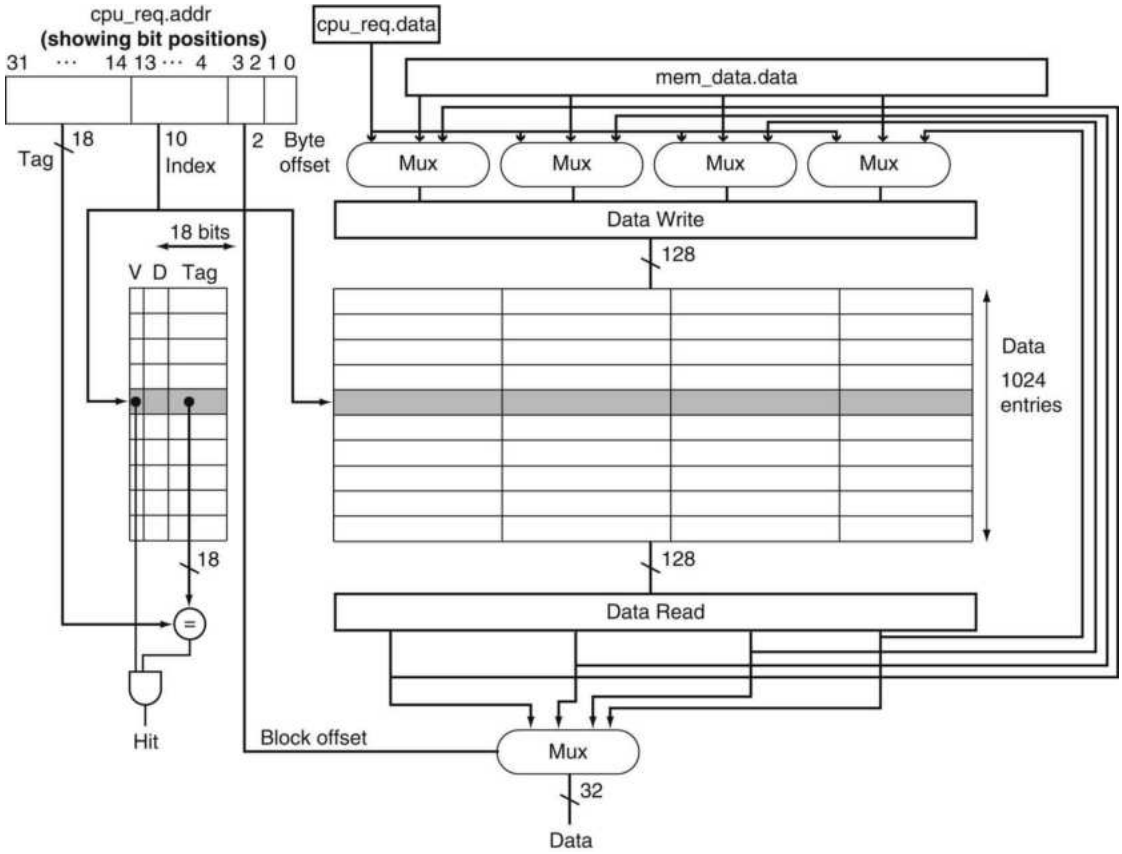


FIGURE E5.12.3 Block diagram of the simple cache using the Verilog names.
Not shown are the write enables for the cache tag memory and for the cache data memory, or the control signals for multiplexors that supply data for the Data Write variable. Rather than have separate write enables on every word of the cache data block, the Verilog reads the old value of the block into Data Write and then updates the word in that variable on a write. It then writes the whole 128-bit block.

```
/*cache: data memory, single port, 1024 blocks*/
module dm_cache_data(input  bit clk,
     input  cache_req_type  data_req,//data request/command, e.g. RW, valid
     input  cache_data_type data_write, //write port (128-bit line)
     output cache_data_type data_read); //read port
  timeunit 1ns; timeprecision 1ps;

  cache_data_typedata_mem[0:1023];

  initial begin
    for (int i=0; i<1024; i++)
         data_mem[i] = '0;
  end

  assign data_read = data_mem[data_req.index];

  always_ff @(posedge(clk)) begin
    if (data_req.we)
      data_mem[data_req.index] <= data_write;
  end
endmodule

/*cache: tag memory, single port, 1024 blocks*/
module dm_cache_tag(input  bit clk, //write clock
     input  cache_req_type tag_req, //tag request/command, e.g. RW, valid
     input  cache_tag_type tag_write,//write port
     output cache_tag_type tag_read);//read port
  timeunit 1ns; timeprecision 1ps;

  cache_tag_typetag_mem[0:1023];

  initial begin
     for (int i=0; i<1024; i++)
     tag_mem[i] = '0;
  end

  assign tag_read = tag_mem[tag_req.index];

  always_ff @(posedge(clk)) begin
    if (tag_req.we)
      tag_mem[tag_req.index] <= tag_write;
  end

endmodule
```

FIGURE E5.12.4 Cache data and tag modules in SystemVerilog.
These are nearly identical except that the data are 32 bits wide between the CPU and cache and are 128 bits wide between the cache and memory. Both only write on positive clock edges if the write enable is set.

```
/*cache finite state machine*/

module dm_cache_fsm(input  bit clk, input  bit rst,
        input  cpu_req_type   cpu_req,      //CPU request input (CPU->cache)
        input  mem_data_type  mem_data,     //memory response (memory->cache)
        output mem_req_type   mem_req,      //memory request (cache->memory)
        output cpu_result_type cpu_res      //cache result (cache->CPU)
    );

  timeunit 1ns;
  timeprecision 1ps;

  /*write clock*/
  typedef enum {idle, compare_tag, allocate, write_back} cache_state_type;

  /*FSM state register*/
  cache_state_typevstate, rstate;

  /*interface signals to tag memory*/
  cache_tag_typetag_read;                   //tag read result
  cache_tag_typetag_write;                  //tag write data
  cache_req_typetag_req;                    //tag request

  /*interface signals to cache data memory*/
  cache_data_typedata_read;                 //cache line read data
  cache_data_typedata_write;                //cache line write data
  cache_req_typedata_req;                   //data req


  /*temporary variable for cache controller result*/
  cpu_result_typev_cpu_res;

  /*temporary variable for memory controller request*/
  mem_req_typev_mem_req;

  assign mem_req = v_mem_req;               //connect to output ports
  assign cpu_res = v_cpu_res;
```

FIGURE E5.12.5 FSM in SystemVerilog, part I.
These modules instantiate the memories according to the type definitions in the previous figure.

```
always_comb begin

    /*-------------------------default values for all signals------------*/
    /*no state change by default*/
    vstate = rstate;
    v_cpu_res = '{0, 0}; tag_write = '{0, 0, 0};

    /*read tag by default*/
    tag_req.we = '0;
    /*direct map index for tag*/
    tag_req.index = cpu_req.addr[13:4];

    /*read current cache line by default*/
    data_req.we = '0;
    /*direct map index for cache data*/
    data_req.index = cpu_req.addr[13:4];

    /*modify correct word (32-bit) based on address*/
    data_write = data_read;
    case(cpu_req.addr[3:2])
    2'b00:data_write[31:0] = cpu_req.data;
    2'b01:data_write[63:32] = cpu_req.data;
    2'b10:data_write[95:64] = cpu_req.data;
    2'b11:data_write[127:96] = cpu_req.data;
    endcase

    /*read out correct word(32-bit) from cache (to CPU)*/
    case(cpu_req.addr[3:2])
    2'b00:v_cpu_res.data = data_read[31:0];
    2'b01:v_cpu_res.data = data_read[63:32];
    2'b10:v_cpu_res.data = data_read[95:64];
    2'b11:v_cpu_res.data = data_read[127:96];
    endcase

    /*memory request address (sampled from CPU request)*/
    v_mem_req.addr = cpu_req.addr;
    /*memory request data (used in write)*/
    v_mem_req.data = data_read;
    v_mem_req.rw = '0;
```

FIGURE E5.12.6 FSM in SystemVerilog, part II.
This section describes the default value of all signals. The following figures will set these values for one clock cycle, and this Verilog will reset it to these values for the following clock cycle.

```
//-----------------------------------Cache FSM-------------------------
case(rstate)
/*idle state*/
idle : begin
    /*If there is a CPU request, then compare cache tag*/
    if (cpu_req.valid)
       vstate = compare_tag;
       end
/*compare_tag state*/
compare_tag : begin
          /*cache hit (tag match and cache entry is valid)*/
    if (cpu_req.addr[TAGMSB:TAGLSB] == tag_read.tag && tag_read.valid) begin
          v_cpu_res.ready = '1;

       /*write hit*/
       if (cpu_req.rw) begin
          /*read/modify cache line*/
          tag_req.we = '1; data_req.we = '1;

          /*no change in tag*/
          tag_write.tag = tag_read.tag;
          tag_write.valid = '1;
          /*cache line is dirty*/
          tag_write.dirty = '1;
       end

          /*xaction is finished*/
          vstate = idle;
    end
    /*cache miss*/
    else begin
      /*generate new tag*/
      tag_req.we = '1;
      tag_write.valid = '1;
      /*new tag*/
      tag_write.tag = cpu_req.addr[TAGMSB:TAGLSB];
      /*cache line is dirty if write*/
      tag_write.dirty = cpu_req.rw;

      /*generate memory request on miss*/
      v_mem_req.valid = '1;
      /*compulsory miss or miss with clean block*/
      if (tag_read.valid == 1'b0 || tag_read.dirty == 1'b0)
         /*wait till a new block is allocated*/
         vstate = allocate;
```

FIGURE E5.12.7 FSM in SystemVerilog, part III. Actual FSM states via case statement in this figure and the next. This figure has the Idle state and most of the Compare Tag state.

```
      else begin
      /*miss with dirty line*/
         /*write back address*/
         v_mem_req.addr = {tag_read.tag, cpu_req.addr[TAGLSB-1:0]};
         v_mem_req.rw = '1;
         /*wait till write is completed*/
         vstate = write_back;
      end
     end
 end
 /*wait for allocating a new cache line*/
 allocate: begin
     /*memory controller has responded*/
     if (mem_data.ready) begin
        /*re-compare tag for write miss (need modify correct word)*/
        vstate = compare_tag;
        data_write = mem_data.data;
        /*update cache line data*/
        data_req.we = '1;
     end
      end
 /*wait for writing back dirty cache line*/
 write_back : begin
     /*write back is completed*/
     if (mem_data.ready) begin
        /*issue new memory request (allocating a new line)*/
        v_mem_req.valid = '1;
        v_mem_req.rw = '0;

        vstate = allocate;
     end
   end
 endcase
end

always_ff @(posedge(clk)) begin
 if (rst)
   rstate <= idle;        //reset to idle state
 else
   rstate <= vstate;
end
/*connect cache tag/data memory*/
dm_cache_tag  ctag(.*);
dm_cache_data cdata(.*);
endmodule
```

FIGURE E5.12.8 FSM in SystemVerilog, part IV. Actual FSM states via the case statement in the prior figure and this one. This figure has the last part of the Compare Tag state, plus Allocate and Write-Back states.

Figures e5.12.1 and e5.12.2 declare the structures that are used in the definition of the cache in the following figures. For example, the cache tag structure (`cache_tag_type`) contains a valid bit (`valid`), a dirty bit (`dirty`), and an 18-bit tag field (`[TAGMSB:TAGLSB] tag`). Figure e5.12.3 shows the block diagram of the cache using the names from the Verilog description.

Figure e5.12.4 defines modules for the cache data (`dm_cache_data`) and cache tag (`dm_cache_tag`). These memories can be read at any time, but writes only occur on the positive clock edge (`posedge(clk)`) and only if write enable is a 1 (`data_req.we` or `tag_req.we`).

Figure e5.12.5 defines the inputs, outputs, and states of the FSM. The inputs are the requests from the CPU (`cpu_req`) and responses from memory (`mem_data`), and the outputs are responses to the CPU (`cpu_res`) and requests to memory (`mem_req`). The figure also declares the internal variables needed by the FSM. For example, the current state and next state registers of the FSM are `rstate` and `vstate`, respectively.

Figure e5.12.6 lists the default values of the control signals, including the word to be read or written from a block, setting the cache write enables to 0, and so on. These values are set every clock cycle, so the write enable for a portion of the cache—for example, `tag_req.we`—would be set to 1 for one clock cycle in the figures below and then would be reset to 0 according to the Verilog in this figure.

The last two figures show the FSM as a large case statement (`case(rstate)`), with the four states split across the two figures. Figure e5.12.7 starts with the Idle state (`idle`), which simply goes to the Compare Tag state (`compare_tag`) if the CPU makes a valid request. It then describes most of the Compare Tag state. The Compare Tag state checks to see if the tags match and the entry is valid. If so, then it first sets the Cache Ready signal (`v_cpu_res.ready`). If the request is a write, it sets the tag field, the valid bit, and the dirty bit. The next state is Idle. If it is a miss, then the state prepares to change the tag entry and valid and dirty bits. If the block to be replaced is clean or invalid, the next state is Allocate.

Figure e5.12.8 continues the Compare Tag state. If the block to be replaced is dirty, then the next state is Write-Back. The figure shows

the Allocate state (`allocate`) next, which simply reads the new block. It keeps looping until the memory is ready; when it is, it goes to the Compare Tag state. This is followed in the figure by the Write-Back state (`write_back`). As the figure shows, the Write-Back state merely writes the dirty block to memory, once again looping until memory is ready. When memory is ready, indicating the write is complete, we go to the Allocate state.

The code at the end sets the current state from the next state or resets the FSM to the Idle state on the next clock edge, depending on a reset signal (`rst`).

The online material includes a Test Case module that will be useful to check the code in these figures. This SystemVerilog could be used to create a cache and cache controller in an FPGA.

Basic Coherent Cache Implementation Techniques

The key to implementing an invalidate protocol is the use of the bus, or another broadcast medium, to perform invalidates. To invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors try to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes. One implication of this scheme is that a write to a shared data item cannot actually complete until it obtains bus access. All coherence schemes require some method of serializing accesses to the same cache block, by serializing access either to the communication medium or another shared structure.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a

cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item, since all written data are unfailingly sent to the memory, from which the most-recent value of a data item can always be fetched. In a design with adequate memory bandwidth to support the write traffic from the processors, using write-through simplifies the implementation of cache coherence.

For a write-back cache, finding the most-recent data value is more difficult, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for cache misses and for writes: each processor snoops all addresses placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted. The increased complexity comes from having to retrieve the cache block from a processor's cache, which can often take longer than retrieving it from the shared memory if the processors are in separate chips. Since write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors and have been the approach chosen in most multiprocessors, despite the additional complexity of maintaining coherence. Therefore, we will examine the implementation of coherence with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward, since they simply rely on the snooping capability. For writes, we'd like to know whether any other copies of the block are cached, because if there are no other cached copies, the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

To track whether or not a cache block is shared, we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*. No further invalidations will be sent by that processor for that block. The processor with the sole copy of a cache block is

normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor, and the state should be made shared.

Every bus transaction must check the cache-address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags. The interference can also be reduced in a multilevel cache by directing the snoop requests to the L2 cache, which the processor uses only when it has a miss in the L1 cache. For this scheme to work, every entry in the L1 cache must be present in the L2 cache, a property called the *inclusion property*. If the snoop gets a hit in the L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor. Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the processor and the snooping activity.

An Example Cache Coherency Protocol

A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node. This controller responds to requests from the processor and from the bus (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Logically, you can think of a separate controller being associated with each block; that is, snooping operations or cache requests for different blocks can proceed independently. In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion (that is, one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). Also, remember that although we refer to a bus in the following description, any interconnection network that supports a broadcast to all the coherence controllers and their associated caches can be used to implement snooping.

The simple protocol we consider has three states: invalid, shared, and modified. The shared state indicates that the block is potentially shared, while the modified state indicates that the block has been updated in the cache; note that the modified state *implies* that the block is exclusive. Figure e5.12.9 shows the requests generated by the processor-cache module in a node (in the first nine rows of the table) as well as those coming from the bus (in the last five rows of the table). This protocol is for a write-back cache, but it can be easily changed to work for a write-through cache by reinterpreting the modified state as an exclusive state and updating the cache on writes in the normal fashion for a write-through cache. The most common extension of this basic protocol is the addition of an exclusive state, which describes a block that is unmodified but held in only one cache; the caption of Figure e5.12.9 describes this state and its addition in more detail.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	processor	shared or modified	normal hit	Read data in cache.
Read miss	processor	invalid	normal miss	Place read miss on bus.
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.
Read miss	processor	modified	replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	processor	modified	normal hit	Write data in cache.
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	processor	invalid	normal miss	Place write miss on bus.
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.
Write miss	processor	modified	replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	bus	shared	no action	Allow memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write-back the cache block and make its state invalid.

FIGURE E5.12.9 The cache coherence mechanism receives requests from both the processor and the bus and responds to these based on the type of request, whether it hits or misses in the cache, and the state of the cache block specified in the request.

The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor

cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read misses, write misses, or invalidates snooped from the bus, an action is required only if the read or write addresses match a block in the cache and the block is valid. Some protocols also introduce a state to designate when a block is exclusively in one cache but has not yet been written. This state can arise if a write access is broken into two pieces: getting the block exclusively in one cache and then subsequently updating it; in such a protocol this "exclusive unmodified state" is transient, ending as soon as the write is completed. Other protocols use and maintain an exclusive state for an unmodified block. In a snooping protocol, this state can be entered when a processor reads a block that is not resident in any other cache. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same processor need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a block is modified. The popular MESI protocol, which is named for the four states it includes (modified, exclusive, shared, and invalid), uses this structure. The MOESI protocol introduces another extension: the "owned" state.

When an invalidate or a write miss is placed on the bus, any processors with copies of the cache block invalidate it. For a write-through cache, the data for a write miss can always be retrieved from the memory. For a write miss in a writeback cache, if the block is exclusive in just one cache, that cache also writes back the block; otherwise, the data can be read from memory.

Figure e5.12.10 shows a finite-state transition diagram for a single cache block using a write invalidation protocol and a write-back

cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on processor requests (on the left, which corresponds to the top half of the table in Figure e5.12.9), contrary to transitions based on bus requests (on the right, which corresponds to the last five rows of the table in Figure e5.12.9). Boldface type is used to distinguish the bus actions, in contrast to the conditions on which a state transition depends. The state in each node represents the state of the selected cache block specified by the processor or bus request.

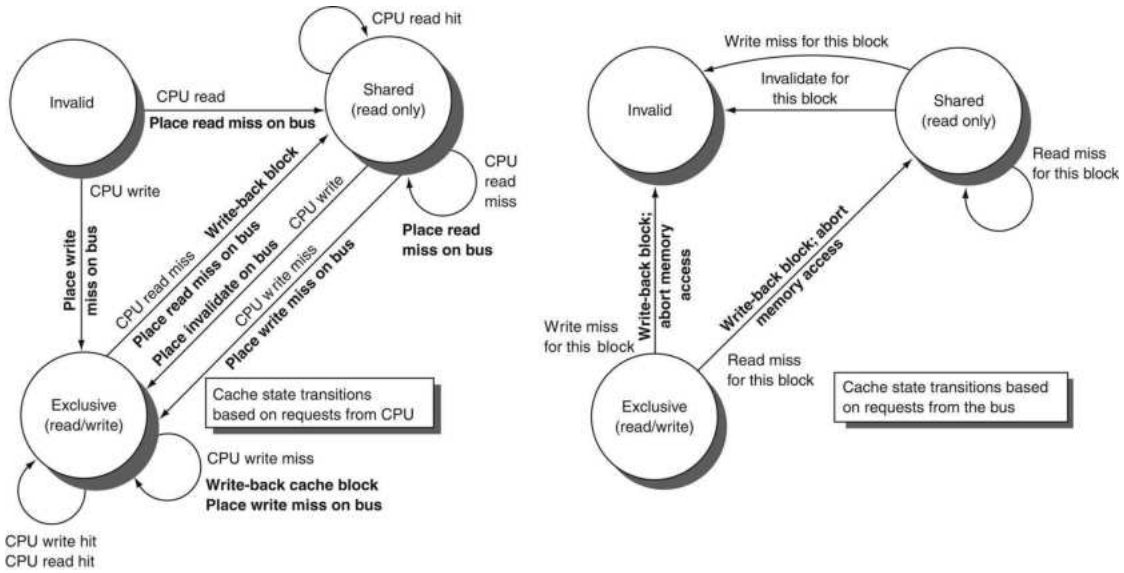


FIGURE E5.12.10 A write-invalidate, cache-coherence protocol for a write-back cache, showing the states and state transitions for each block in the cache.

The cache states are shown in circles, with any access permitted by the processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in bold. The stimulus actions apply to a block in the cache, not to a specific address in the cache. Hence, a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read

miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory provides data on a read miss for a block that is clean in all caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty states. Most of the state changes indicated by arcs in the left half of Figure e5.12.10 would be needed in a write-back uniprocessor cache, with the exception being the invalidate on a write hit to a shared block. The state changes represented by the arcs in the right half of Figure e5.12.10 are needed only for coherence and would not appear at all in an uniprocessor cache controller.

As mentioned earlier, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. Figure e5.12.11 shows how the state transitions in the right half of Figure e5.12.10 are combined with those in the left half of the figure to form a single state diagram for each cache block.

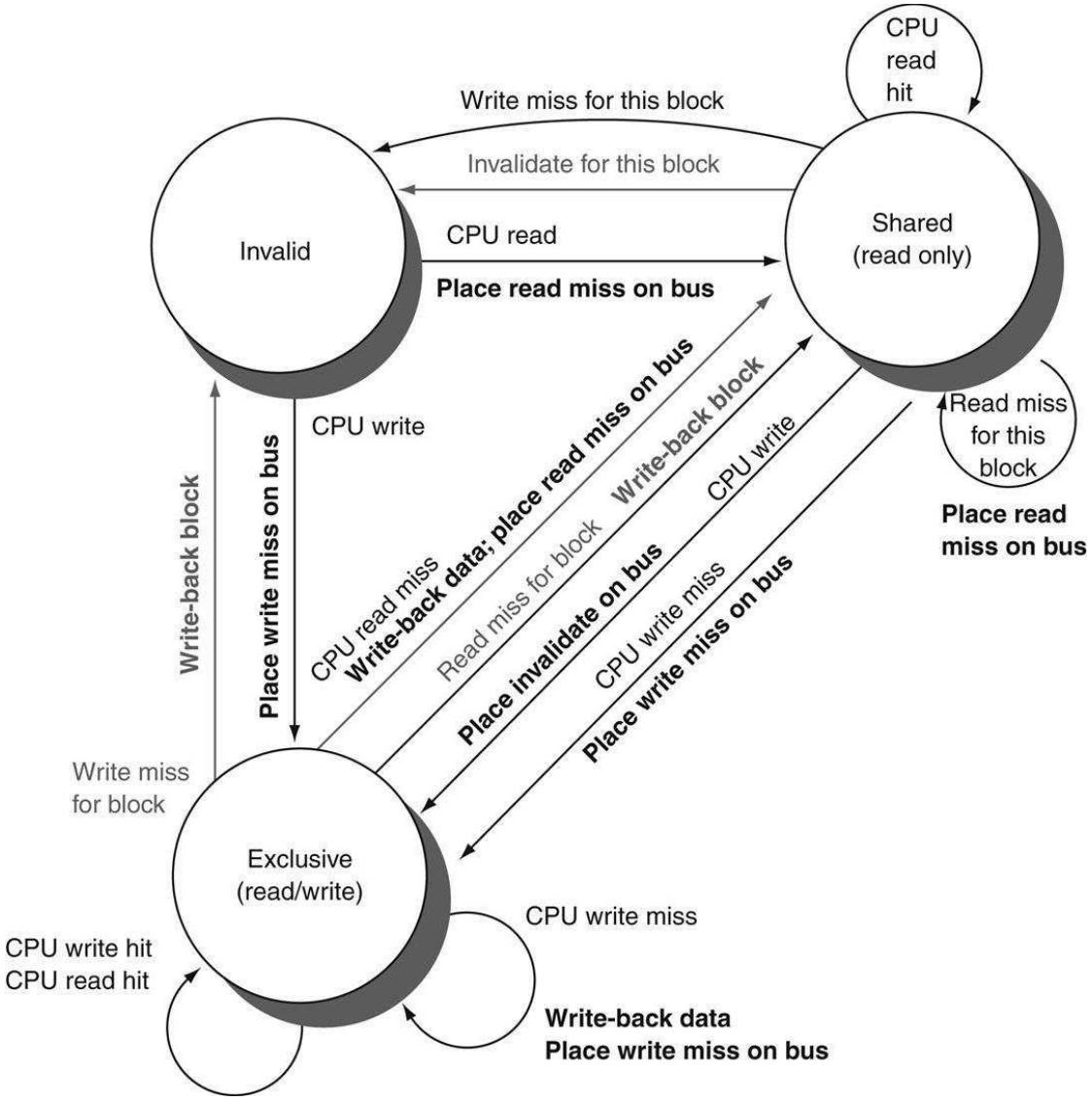


FIGURE E5.12.11 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray.
As in Figure e5.12.10, the activities on a transition are shown in bold.

To understand why this protocol works, observe that any valid cache block is either in the shared state in one or more caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires an invalidate or write miss to be placed on the bus, causing all caches to make the block invalid. In addition, if some other cache had the block in the exclusive state, that cache generates a write back, which supplies the block containing the desired

address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the cache with the exclusive copy changes its state to shared.

The actions in gray in Figure e5.12.11, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the memory, which simplifies the implementation.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality, this is not true. Similarly, if we used a switch, as all recent multiprocessors do, then even read misses would also not be atomic.

Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. On the next page, we will discuss how these protocols are implemented without a bus.

Constructing small-scale (two to four processors) multiprocessors has become very easy. For example, the Intel Nehalem and AMD Opteron processors are designed for use in cache-coherent multiprocessors and have an external interface that supports snooping and allows two to four processors to be directly connected. They also have larger on-chip caches to reduce bus utilization. In the case of the Opteron processors, the support for interconnecting multiple processors is integrated onto the processor chip, as are the memory interfaces. In the case of the Intel design, a two-processor system can be built with only a few additional external chips to interface with the memory system and I/O. Although these designs cannot be easily scaled to larger processor counts, they offer an extremely cost-effective solution for two to four processors.

The devil is in the details.

Classic proverb.

Implementing Snoopy Cache Coherence

As we said earlier, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss; communicating with the other processors and memory; getting the most-recent value for a write miss and ensuring that any invalidates are processed; and updating the cache cannot be done as if they took a single cycle.

In a simple single-bus system, these steps can be made effectively atomic by arbitrating for the bus first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In most bus-based multiprocessors, a single line is used to signal when all necessary invalidates have been received and are being processed. Following that signal, the processor that generated the miss can release the bus, knowing that any required actions will be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively makes the individual steps atomic.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: one write is processed before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a snoopy system, ensuring that a race has only one winner is accomplished by using broadcast for all misses, as well as some basic properties of the interconnection network. These properties, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snoopy cache coherence without a bus.

5.13 Real Stuff: The ARM Cortex-A53 and Intel Core i7 Memory Hierarchies

In this section, we will look at the memory hierarchy of the same two microprocessors described in Chapter 4: the ARM Cortex-A53 and Intel Core i7. This section is in part based on Section 2.6 of *Computer Architecture: A Quantitative Approach*, 5th edition.

Figure 5.42 summarizes the address sizes and TLBs of the two processors. Note that the Cortex-A53 has two 10-entry fully associative micro-TLBs backed by a shared 512-entry four-way set associative main TLB with a 48-bit virtual address space and a 40-bit physical address space. The Core i7 has three TLBs with a 48-bit virtual address and a 44-bit physical address. Although the 64-bit registers of these processors could hold a larger virtual address, there was no software need for such a large space, and 48-bit virtual addresses shrinks both the page table memory footprint and the TLB hardware.

Characteristic	ARM Cortex-A53	Intel Core i7
Virtual address	48 bits	48 bits
Physical address	40 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 2 MiB, 1 GiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both micro TLBs are fully associative, with 10 entries, round robin replacement 64-entry, four-way set-associative TLBs TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, seven per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

FIGURE 5.42 Address translation and TLB hardware for the ARM Cortex-A53 and Intel Core i7 920.
Both processors provide support for large pages, which are used for things like the operating system or mapping a frame buffer. The large-page scheme avoids using a large number of entries to map a single object that is always present.

Figure 5.43 shows their caches. The Cortex-A53 has between one and four processors or cores while the Core i7 is fixed at four. Cortex-A53 has a 16 to 64 KiB, two-way L1 instruction cache (per core) and the Core i7 has a 32 KiB, four-way set associative, L1 instruction cache (per core). Both use 64 byte blocks. The Cortex-A53 increases the associativity to four-way for the data cache, other variables remain the same. Similarly, the Core i7 keeps everything the same except the associativity, which it increases to eight-way. The Core i7 provides a 256 KiB, eight-way set associative unified L2 cache (per core) with 64 byte blocks. In contrast, the Cortex-A53 provides a L2 cache that is shared between one and four cores. This cache is 16-way set associative with 64 byte blocks and between 128 KiB and 2 MiB in size. As the Core i7 is used for servers, it also offers an L3 cache shared by all the cores on the chip. Its size varies

depending on the number of cores. With four cores, as in this case, the size is 8 MiB.

Characteristic	ARM Cortex-A53	Intel Core i7
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	Configurable 16 to 64 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	Two-way (I), four-way (D) set associative	Four-way (I), eight-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, variable allocation policies (default is Write-allocate)	Write-back, No-write-allocate
L1 hit time (load-use)	Two clock cycles	Four clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 2 MiB	256 KiB (0.25 MiB)
L2 cache associativity	16-way set associative	8-way set associative
L2 replacement	Approximated LRU	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate	Write-back, Write-allocate
L2 hit time	12 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

FIGURE 5.43 Caches in the ARM Cortex-A53 and Intel Core i7 920.

A significant challenge facing cache designers is to support processors like the Cortex-A53 and the Core i7 that can execute more than one memory instruction per clock cycle. A popular technique is to break the cache into banks and allow multiple, independent, **parallel** accesses, provided the accesses are to different banks. The technique is similar to interleaved DRAM

banks (see Section 5.2).

The Cortex-A53 and the Core i7 have additional optimizations that allow them to reduce the miss penalty. The first of these is the return of the requested word first on a miss. They also continue to execute instructions that access the data cache during a cache miss. Designers who are attempting to hide the cache miss latency commonly use this technique, called a **nonblocking cache.** They implement two flavors of nonblocking. *Hit under miss* allows additional cache hits during a miss, while *miss under miss* allows multiple outstanding cache misses. The aim of the first of these two is hiding some miss latency with other work, while the aim of the second is overlapping the latency of two different misses.

nonblocking cache

A cache that allows the processor to make references to the cache while the cache is handling an earlier miss.

Overlapping a large fraction of miss times for multiple outstanding misses requires a high-bandwidth memory system capable of handling multiple misses in parallel. In a personal mobile device, the memory system below it can often pipeline, merge, reorder, or prioritize requests appropriately. Large servers

and multiprocessors typically have memory systems capable of handling several outstanding misses in parallel.

The Cortex-A53 and the Core i7 have prefetch mechanisms for data accesses. They look at a pattern of data misses and uses this information to try to predict the next address to start fetching the data before the miss occurs. Such techniques generally work best when accessing arrays in loops.

The sophisticated memory hierarchies of these chips and the large fraction of the dies dedicated to caches and TLBs show the significant design effort expended to try to close the gap between processor cycle times and memory latency.

Performance of the Cortex-A53 and Core i7 Memory Hierarchies

The memory hierarchy of the Cortex-A53 was measured using a 32 KiB two-way set associative L1 instruction cache, a 32 KiB four-way set associative L1 data cache, and a 1 MiB 16-way set associative L2 cache running the integer SPEC2006 benchmarks.

The Cortex-A53 instruction cache miss rates for these benchmarks are very small. Figure 5.44 shows the data cache results for the Cortex-A53, which have significant L1 and L2 miss rates. The L1 data cache miss rates go from 0.5% to 37.3%, with a mean of 6.4% and a median of 2.4%. The (global) L2 cache miss rates vary from 0.1% to 9.0%, with a mean of 1.3% and a median of 0.3%. The L1 miss penalty for a 1 GHz Cortex-A53 is 12 clock cycles, while the L2 miss penalty is 124 clock cycles. Using these miss penalties, Figure 5.45 shows the average miss penalty per data access. When these low miss rates are multiplied by their high miss penalties, you can see that they can represent a significant fraction of the CPI for 5 of the 12 SPEC2006 programs.

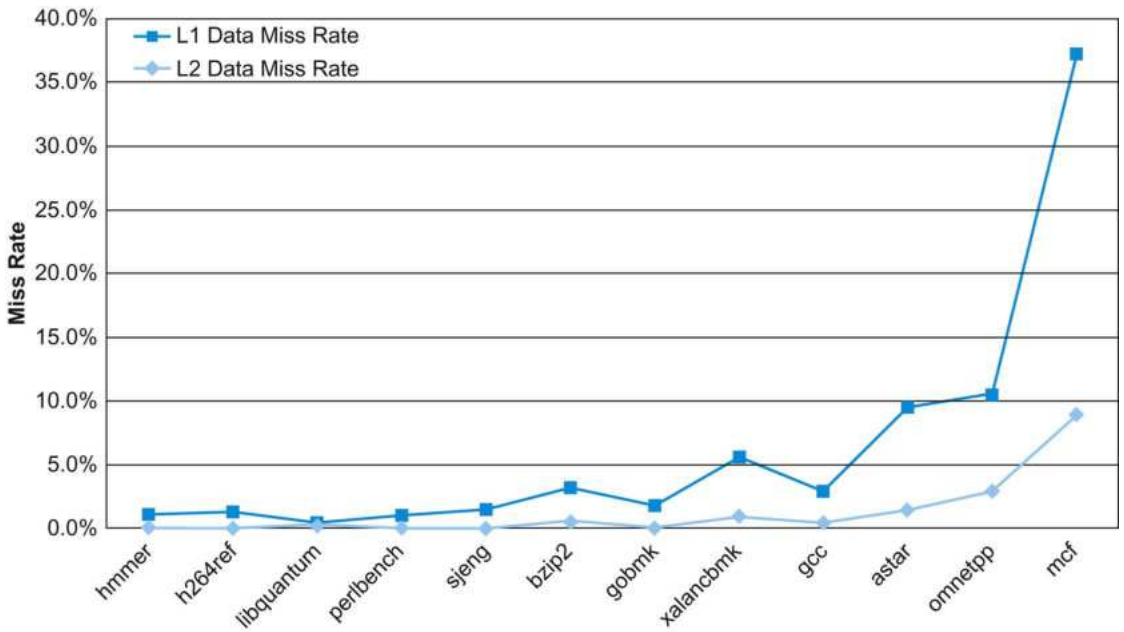


FIGURE 5.44 Data cache miss rates for ARM Cortex-A53 when running SPEC2006int. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate; that is, counting all references, including those that hit in L1. (See the *Elaboration* in Section 5.4.) mcf is known as a cache buster. Note that this figure is for the same systems and benchmarks as Figure 4.74 in Chapter 4.

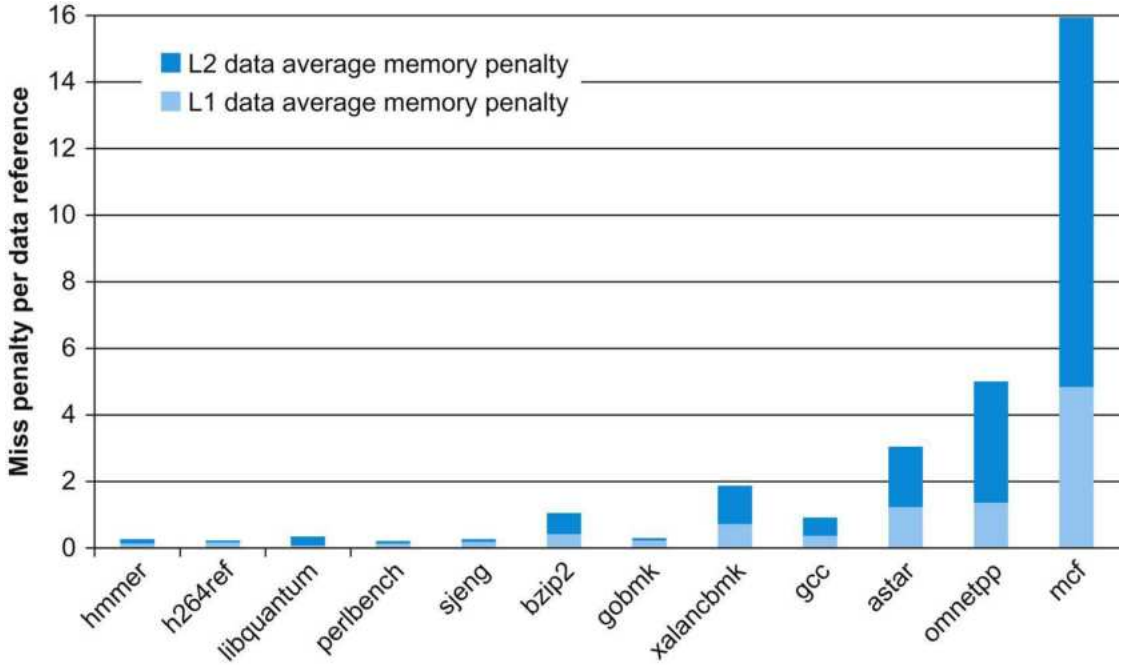


FIGURE 5.45 The average memory access penalty

in clock cycles per data memory reference coming from L1 and L2 is shown for the ARM processor when running SPEC2006int.
Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

Figure 5.46 shows the miss rates for the caches of the Core i7 using the SPEC2006 benchmarks. The L1 instruction cache miss rate varies from 0.1% to 1.8%, averaging just over 0.4%. This rate is in keeping with other studies of instruction cache behavior for the SPECCPU2006 benchmarks, which show low instruction cache miss rates. With L1 data cache miss rates running 5% to 10%, and sometimes higher, the importance of the L2 and L3 caches should be obvious. Since the cost for a miss to memory is over 100 cycles, and the average data miss rate in L2 is 4%, L3 is obviously critical. Assuming about half the instructions is loads or stores, without L3 the L2 cache misses could add two cycles per instruction to the CPI! In comparison, the average L3 data miss rate of 1% is still significant but four times lower than the L2 miss rate and six times less than the L1 miss rate.

Elaboration

Because speculation may sometimes be wrong (see Chapter 4), there are references to the L1 data cache that do not correspond to loads or stores that eventually complete execution. The data in Figure 5.44 are measured against all data requests, including some that are cancelled. The miss rate when measured against only completed data accesses is 1.6 times higher (an average of 9.5% versus 5.9% for L1 Dcache misses).

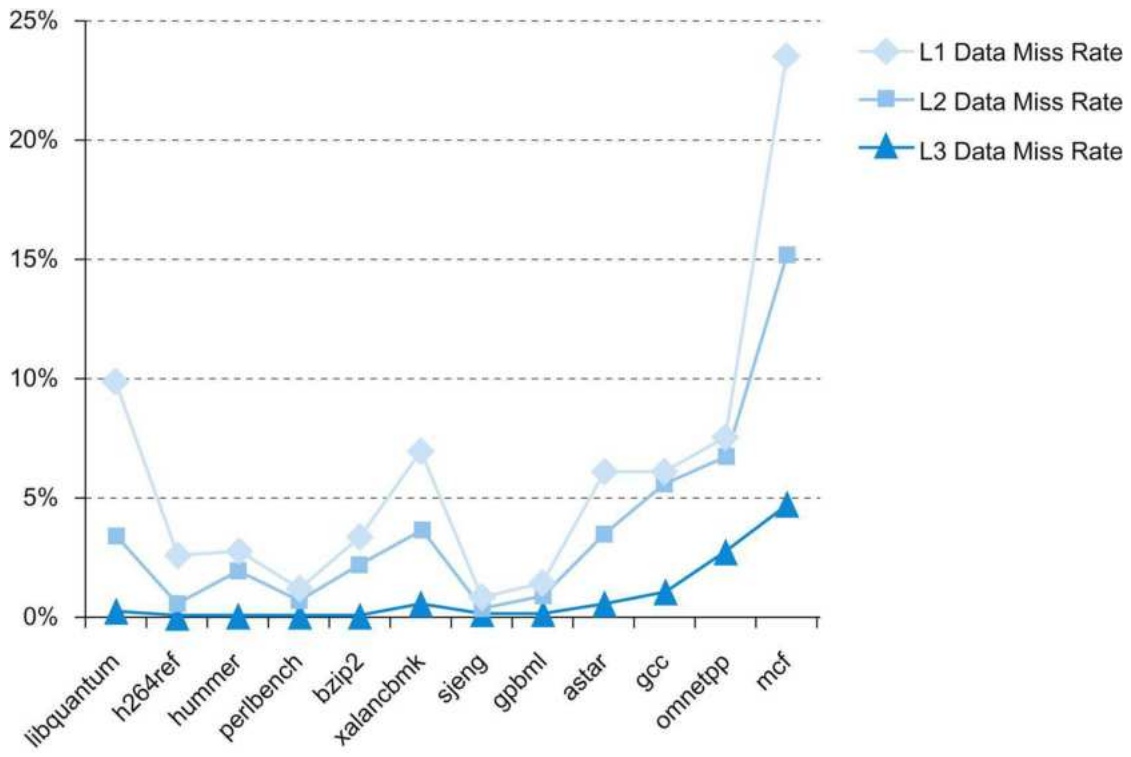


FIGURE 5.46 The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full integer SPECCPU2006 benchmarks.

5.14 Real Stuff: The Rest of the RISC-V System and Special Instructions

Figure 5.48 lists the 13 remaining RISC-V instructions in the special purpose and systems category.

The fence instructions provide synchronization barriers for instructions (**`fence.i`**), data (**`fence`**), and address translations (**`sfence.vma`**). The first, `fence.i`, informs the processor that software has modified instruction memory, so that it can guarantee that instruction fetch will reflect the updated instructions. The second, `fence`, affects data memory access ordering for multiprocessing and I/O. The third, `sfence.vma`, informs the processor that software has modified the page tables, so that it can guarantee that address translations will reflect the updates.

The six *control and status register* (CSR) access instructions move data between general-purpose registers and CSRs. The `csrrwi` instruction (CSR read/write immediate) copies a CSR to an integer

register, then overwrites the CSR with an immediate. `csrrsi` (CSR read/set immediate) copies a CSR to an integer register, and overwrites the CSR with the bitwise OR of the CSR and an immediate. `csrrci` (CSR read/clear) is like `csrrsi`, but clears bits instead of setting them. The `csrrw`, `csrrs`, and `csrrc` instructions use a register operand instead of an immediate, but otherwise do the same thing.

Two instructions' only purpose is to generate exceptions: `ecall` generates an environment call exception to invoke the OS, and `ebreak` generates a breakpoint exception to invoke the debugger. The supervisor exception-return instruction (`sret`), naturally enough, allows the program to return from an exception handler.

Finally, the wait-for-interrupt instruction, `wfi`, informs the processor that it may enter an idle state until an interrupt occurs.

5.15 Going Faster: Cache Blocking and Matrix Multiply

Our next step in the continuing saga of improving performance of DGEMM by tailoring it to the underlying hardware is to add cache blocking to the subword parallelism and instruction level parallelism optimizations of Chapters 3 and 4. Figure 5.47 shows the blocked version of DGEMM from Figure 4.78. The changes are the same as was made earlier in going from unoptimized DGEMM in Figure 3.22 to blocked DGEMM in Figure 5.21 above. This time we take the unrolled version of DGEMM from Chapter 4 and invoke it many times on the submatrices of `A`, `B`, and `C`. Indeed, lines 28–34 and lines 7–8 in Figure 5.47 mirror lines 14–20 and lines 5–6 in Figure 5.21, except for incrementing the for loop in line 7 by the amount unrolled.

```
 1 #include <x86intrin.h>
 2 #define UNROLL (4)
 3 #define BLOCKSIZE 32
 4 void do_block (int n, int si, int sj, int sk,
 5                double *A, double *B, double *C)
 6 {
 7   for ( int i = si; i < si+BLOCKSIZE; i+=UNROLL*4 )
 8     for ( int j = sj; j < sj+BLOCKSIZE; j++ ) {
 9       __m256d c[4];
10       for ( int x = 0; x < UNROLL; x++ )
11         c[x] = _mm256_load_pd(C+i+x*4+j*n);
12      /* c[x] = C[i][j] */
13       for( int k = sk; k < sk+BLOCKSIZE; k++ )
14       {
15         __m256d b = _mm256_broadcast_sd(B+k+j*n);
16      /* b = B[k][j] */
17         for (int x = 0; x < UNROLL; x++)
18           c[x] = _mm256_add_pd(c[x], /* c[x]+=A[i][k]*b */
19                  _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
20       }
21
22
23       for ( int x = 0; x < UNROLL; x++ )
24         _mm256_store_pd(C+i+x*4+j*n, c[x]);
           /* C[i][j] = c[x] */
25     }
26 }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
31     for ( int si = 0; si < n; si += BLOCKSIZE )
32       for ( int sk = 0; sk < n; sk += BLOCKSIZE )
33         do_block(n, si, sj, sk, A, B, C);
34 }
```

FIGURE 5.47 Optimized C version of DGEMM from Figure 4.78 using cache blocking. These changes are the same ones found in Figure 5.21. The assembly language produced by the compiler for the `do_block` function is nearly identical to Figure 4.79. Once again, there is no overhead to call the `do_block` because the compiler inlines the function call.

Unlike the earlier chapters, we do not show the resulting x86 code because the inner loop code is nearly identical to Figure 4.79, as the blocking does not affect the computation, just the order that it accesses data in memory. What does change is the bookkeeping integer instructions to implement the loops. It expands from 14

instructions before the inner loop and eight after the loop for Figure 4.78 to 40 and 28 instructions respectively for the bookkeeping code generated for Figure 5.47. Nevertheless, the extra instructions executed pale in comparison to the performance improvement of reducing cache misses. Figure 5.49 compares unoptimized to optimized for subword parallelism, instruction level parallelism, and caches. Blocking improves performance over unrolled AVX code by factors of 2 to 2.5 for the larger matrices. When we compare unoptimized code to the code with all three optimizations, the performance improvement is factors of 8 to 15, with the largest increase for the largest matrix.

Elaboration

As mentioned in the *Elaboration* in Section 3.9, these results are with Turbo mode turned off. As in Chapters 3 and 4, when we turn it on, we improve all the results by the temporary increase in the clock rate of 3.3/2.6 =1.27. Turbo mode works particularly well in this case because it is using only a single core of an eight-core chip. However, if we want to run fast we should use all cores, which we'll see in Chapter 6.

Type	Mnemonic	Name
Mem. Ordering	FENCE.I	Instruction Fence
	FENCE	Fence
	SFENCE.VMA	Address Translation Fence
CSR Access	CSRRWI	CSR Read/Write Immediate
	CSRRSI	CSR Read/Set Immediate
	CSRRCI	CSR Read/Clear Immediate
	CSRRW	CSR Read/Write
	CSRRS	CSR Read/Set
	CSRRC	CSR Read/Clear
System	ECALL	Environment Call
	EBREAK	Environment Breakpoint
	SRET	Supervisor Exception Return
	WFI	Wait for Interrupt

FIGURE 5.48 The list of assembly language instructions for the systems and special operations in the full RISC-V instruction set.

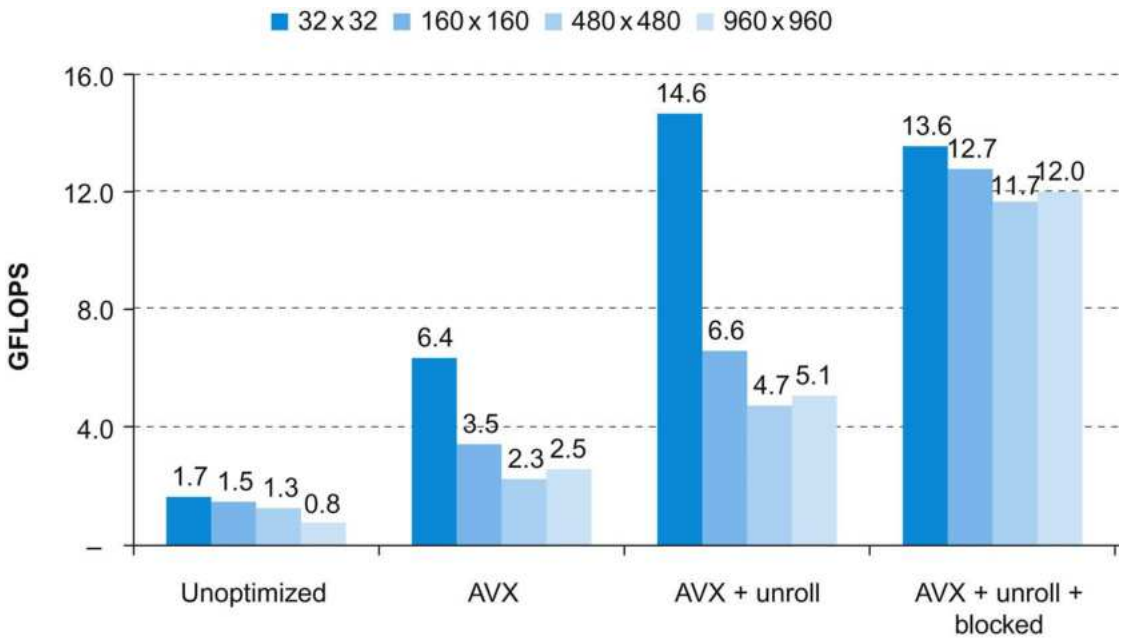


FIGURE 5.49 Performance of four versions of DGEMM from matrix dimensions 32 ×32 to 960 ×960.
The fully optimized code for the largest matrix is almost 15 times as fast the unoptimized version in Figure 3.22 in Chapter 3.

5.16 Fallacies and Pitfalls

As one of the most naturally quantitative aspects of computer architecture, the memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Not only have there been many fallacies propagated and pitfalls encountered, but some have led to major negative outcomes. We start with a pitfall that often traps students in exercises and exams.

Pitfall: Ignoring memory system behavior when writing programs or when generating code in a compiler.

This could be rewritten as a fallacy: "Programmers can ignore

memory hierarchies in writing code." The evaluation of sort in Figure 5.19 and of cache blocking in Section 5.14 demonstrate that programmers can easily double performance if they factor the behavior of the memory system into the design of their algorithms.

Pitfall: Forgetting to account for byte addressing or the cache block size in simulating a cache.

When simulating a cache (by hand or by computer), we need to make sure we account for the effect of byte addressing and multiword blocks in determining into which cache block a given address maps. For example, if we have a 32-byte direct-mapped cache with a block size of 4 bytes, the byte address 36 maps into block 1 of the cache, since byte address 36 is block address 9 and (9 modulo 8)=1. On the other hand, if address 36 is a word address, then it maps into block (36 mod 8)=4. Make sure the problem clearly states the base of the address.

In like fashion, we must account for the block size. Suppose we have a cache with 256 bytes and a block size of 32 bytes. Into which block does the byte address 300 fall? If we break the address 300 into fields, we can see the answer:

63	62	61	...	...	...	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	...	...	...	0	0	0	1	0	0	1	0	1	1	0	0
										Cache block number			Block offset				

Block address

Byte address 300 is block address

$$\left\lfloor \frac{300}{32} \right\rfloor = 9$$

The number of blocks in the cache is

$$\left\lceil \frac{256}{32} \right\rceil = 8$$

Block number 9 falls into cache block number (9 modulo 8)=1.

This mistake catches many people, including the authors (in earlier drafts) and instructors who forget whether they intended the addresses to be in doublewords, words, bytes, or block numbers. Remember this pitfall when you tackle the exercises.

Pitfall: Having less set associativity for a shared cache than the number of cores or threads sharing that cache.

Without extra care, a **parallel** program running on 2^n processors or threads can easily allocate data structures to addresses that would map to the same set of a shared L2 cache. If the cache is at least 2^n-way associative, then these accidental conflicts are hidden by the hardware from the program. If not, programmers could face apparently mysterious performance bugs—actually due to L2 conflict misses—when migrating from, say, a 16-core design to 32-core design if both use 16-way associative L2 caches.

PARALLELISM

Pitfall: Using average memory access time to evaluate the memory hierarchy of an out-of-order processor.

If a processor stalls during a cache miss, then you can separately calculate the memory-stall time and the processor execution time, and hence evaluate the memory hierarchy independently using average memory access time (see page 391).

If the processor continues to execute instructions, and may even sustain more cache misses during a cache miss, then the only accurate assessment of the memory hierarchy is to simulate the out-of-order processor along with the memory hierarchy.

Pitfall: Extending an address space by adding segments on top of an unsegmented address space.

During the 1970s, many programs grew so large that not all the code and data could be addressed with just a 16-bit address. Computers were then revised to offer 32-bit addresses, either through an unsegmented 32-bit address space (also called a *flat address space*) or by adding 16 bits of segment to the existing 16-bit address. From a marketing point of view, adding segments that were programmer-visible and that forced the programmer and compiler to decompose programs into segments could solve the addressing problem. Unfortunately, there is trouble any time a programming language wants an address that is larger than one segment, such as indices for large arrays, unrestricted pointers, or reference parameters. Moreover, adding segments can turn every address into two words—one for the segment number and one for the segment offset—causing problems in the use of addresses in registers.

Fallacy: Disk failure rates in the field match their specifications.

Two recent studies evaluated large collections of disks to check the relationship between results in the field compared to specifications. One study was of almost 100,000 disks that had quoted MTTF of 1,000,000 to 1,500,000 hours, or AFR of 0.6% to 0.8%. They found AFRs of 2% to 4% to be common, often three to five times higher than the specified rates [Schroeder and Gibson, 2007]. A second study of more than 100,000 disks at Google, which had a quoted AFR of about 1.5%, saw failure rates of 1.7% for drives in their first year rise to 8.6% for drives in their third year, or about

five to six times the declared rate [Pinheiro, Weber, and Barroso, 2007].

Fallacy: Operating systems are the best place to schedule disk accesses.

As mentioned in Section 5.2, higher-level disk interfaces offer logical block addresses to the host operating system. Given this high-level abstraction, the best an OS can do to try to help performance is to sort the logical block addresses into increasing order. However, since the disk knows the actual mapping of the logical addresses onto the physical geometry of sectors, tracks, and surfaces, it can reduce the rotational and seek latencies by rescheduling.

For example, suppose the workload is four reads [Anderson, 2003]:

Operation	Starting LBA	Length
Read	724	8
Read	100	16
Read	9987	1
Read	26	128

The host might reorder the four reads into logical block order:

Operation	Starting LBA	Length
Read	26	128
Read	100	16
Read	724	8
Read	9987	1

Depending on the relative location of the data on the disk, reordering could make it worse, as Figure 5.50 shows. The disk-scheduled reads complete in three-quarters of a disk revolution, but the OS-scheduled reads take three revolutions.

Pitfall: Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.

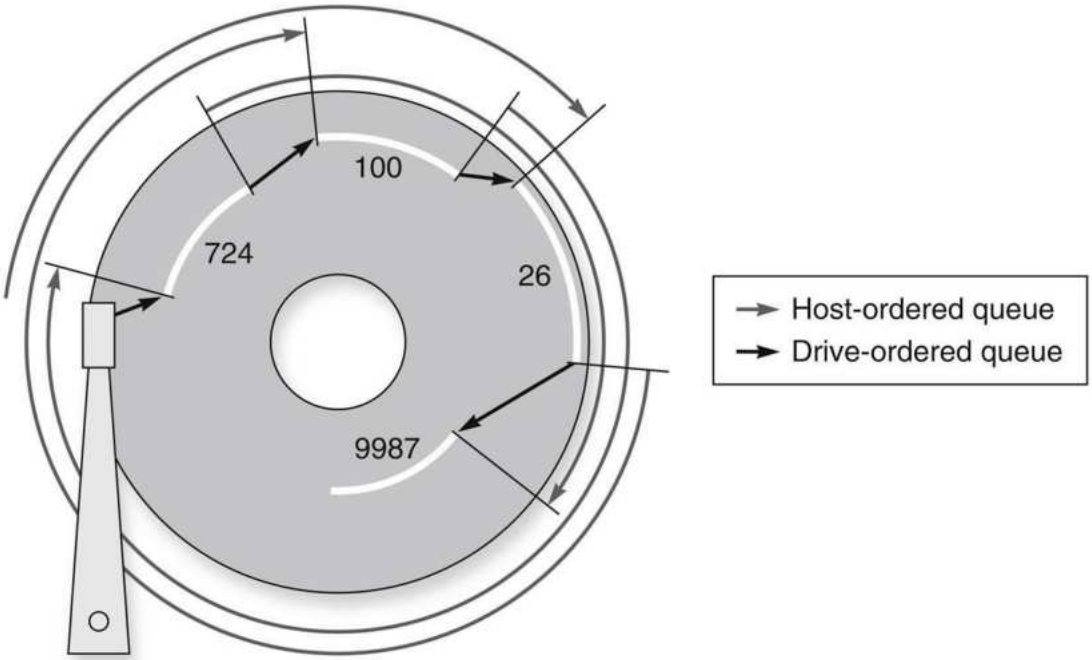


FIGURE 5.50 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered.
The former takes three revolutions to complete the four reads, while the latter completes them in just three-fourths of a revolution. From Anderson [2003].

Many architects in the 1970s and 1980s weren't careful to make sure that all instructions reading or writing information related to hardware resource information were privileged. This *laissez-faire* attitude causes problems for VMMs for all of these architectures, including the x86, which we use here as an example.

Figure 5.51 describes the 18 instructions that cause problems for virtualization [Robin and Irvine, 2000]. The two broad classes are instructions that

- Read control registers in user mode that reveals that the guest operating system is running in a virtual machine (such as POPF, mentioned earlier)
- Check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level

Problem category	Problem x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (SGDT) Store local descriptor table register (SLDT) Store interrupt descriptor table register (SIDT) Store machine status word (SMSW) Push flags (PUSHF, PUSHFD) Pop flags (POPF, POPFD)
When accessing virtual memory mechanisms in user mode, instructions fail the x86 protection checks	Load access rights from segment descriptor (LAR) Load segment limit from segment descriptor (LSL) Verify if segment descriptor is readable (VERR) Verify if segment descriptor is writable (VERW) Pop to segment register (POP CS, POP SS, . . .) Push segment register (PUSH CS, PUSH SS, . . .) Far call to different privilege level (CALL) Far return to different privilege level (RET) Far jump to different privilege level (JMP) Software interrupt (INT) Store segment selector register (STR) Move to/from segment registers (MOVE)

FIGURE 5.51 Summary of 18 x86 instructions that cause problems for virtualization [Robin and Irvine, 2000].

The first five instructions in the top group allow a program in user mode to read a control register, such as descriptor table registers, without causing a trap. The pop flags instruction modifies a control register with sensitive information but fails silently when in user mode. The protection checking of the segmented architecture of the x86 is the downfall of the bottom group, as each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the Move to segment register tries to modify control state, and protection checking foils it as well.

To simplify implementations of VMMs on the x86, both AMD and Intel have proposed extensions to the architecture via a new mode. Intel's VT-x provides a new execution mode for running VMs, an architected definition of the VM state, instructions to swap VMs rapidly, and a large set of parameters to select the circumstances where a VMM must be invoked. Altogether, VT-x adds 11 new instructions for the x86. AMD's Pacifica makes similar proposals.

An alternative to modifying the hardware is to make small

changes to the operating system to avoid using the troublesome pieces of the architecture. This technique is called *paravirtualization*, and the open source Xen VMM is a good example. The Xen VMM provides a guest OS with a virtual machine abstraction that uses only the easy-to-virtualize parts of the physical x86 hardware on which the VMM runs.

5.17 Concluding Remarks

The difficulty of building a memory system to keep pace with faster processors is underscored by the fact that the raw material for main memory, DRAMs, is essentially the same in the fastest computers as it is in the slowest and cheapest.

It is the principle of locality that gives us a chance to overcome the long latency of memory access—and the soundness of this strategy is demonstrated at all levels of the **memory hierarchy**. Although these levels of the hierarchy look quite different in quantitative terms, they follow similar strategies in their operation and exploit the same properties of locality.

Multilevel caches make it possible to use more cache optimizations more easily for two reasons. First, the design parameters of a lower-level cache are different from a first-level

cache. For example, because a lower-level cache will be much larger, it is possible to use bigger block sizes. Second, a lower-level cache is not constantly being used by the processor, as a first-level cache is. This allows us to consider having the lower-level cache do something when it is idle that may be useful in preventing future misses.

Another trend is to seek software help. Efficiently managing the memory hierarchy using a variety of program transformations and hardware facilities is a major focus of compiler enhancements. Two different ideas are being explored. One idea is to reorganize the program to enhance its spatial and temporal locality. This approach focuses on loop-oriented programs that use sizable arrays as the major data structure; large linear algebra problems are a typical example, such as DGEMM. By restructuring the loops that access the arrays, substantially improved locality—and, therefore, cache performance—can be obtained.

PREDICTION

Another approach is **prefetching**. In prefetching, a block of data is brought into the cache before it is actually referenced. Many

microprocessors use hardware prefetching to try to *predict* accesses that may be difficult for software to notice.

prefetching
A technique in which data blocks needed in the future are brought into the cache early by using special instructions that specify the address of the block.

A third approach is special cache-aware instructions that optimize memory transfer. For example, the microprocessors in Section 6.10 in Chapter 6 use an optimization that does not fetch the contents of a block from memory on a write miss because the program is going to write the full block. This optimization significantly reduces memory traffic for one kernel.

As we will see in Chapter 6, memory systems are a central design issue for parallel processors. The growing significance of the memory hierarchy in determining system performance means that this important area will continue to be a focus for both designers and researchers for some years to come.

5.18 Historical Perspective and Further Reading

This section, which appears online, gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy, protection mechanisms, and virtual machines, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

5.18 Historical Perspective and Further Reading

> *…the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory…. Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large.*
>
> *Maurice Wilkes, Memoirs of a Computer Pioneer, 1985*

This history section gives an overview of memory technologies, from mercury delay lines to DRAM, the invention of the memory hierarchy, and protection mechanisms, and concludes with a brief history of operating systems, including CTSS, MULTICS, UNIX, BSD UNIX, MS-DOS, Windows, and Linux.

The developments of most of the concepts in this chapter have been driven by revolutionary advances in the technology we use for memory. Before we discuss how memory hierarchies were evolved, let's take a brief tour of the development of memory technology.

The ENIAC had only a small number of registers (about 20) for its storage and implemented these with the same basic vacuum tube technology that it used for building logic circuitry. However, the vacuum tube technology was far too expensive to be used to build a larger memory capacity. Eckert came up with the idea of developing a new technology based on mercury delay lines. In this technology, electrical signals were converted into vibrations that were sent down a tube of mercury, reaching the other end, where they were read out and recirculated. One mercury delay line could store about 0.5 Kbits. Although these bits were accessed serially, the mercury delay line was about a hundred times more cost-effective than vacuum tube memory. The first known working mercury delay lines were developed at Cambridge for the EDSAC. Figure e5.17.1 shows the mercury delay lines of the EDSAC, which had 32 tanks and 512 36-bit words.

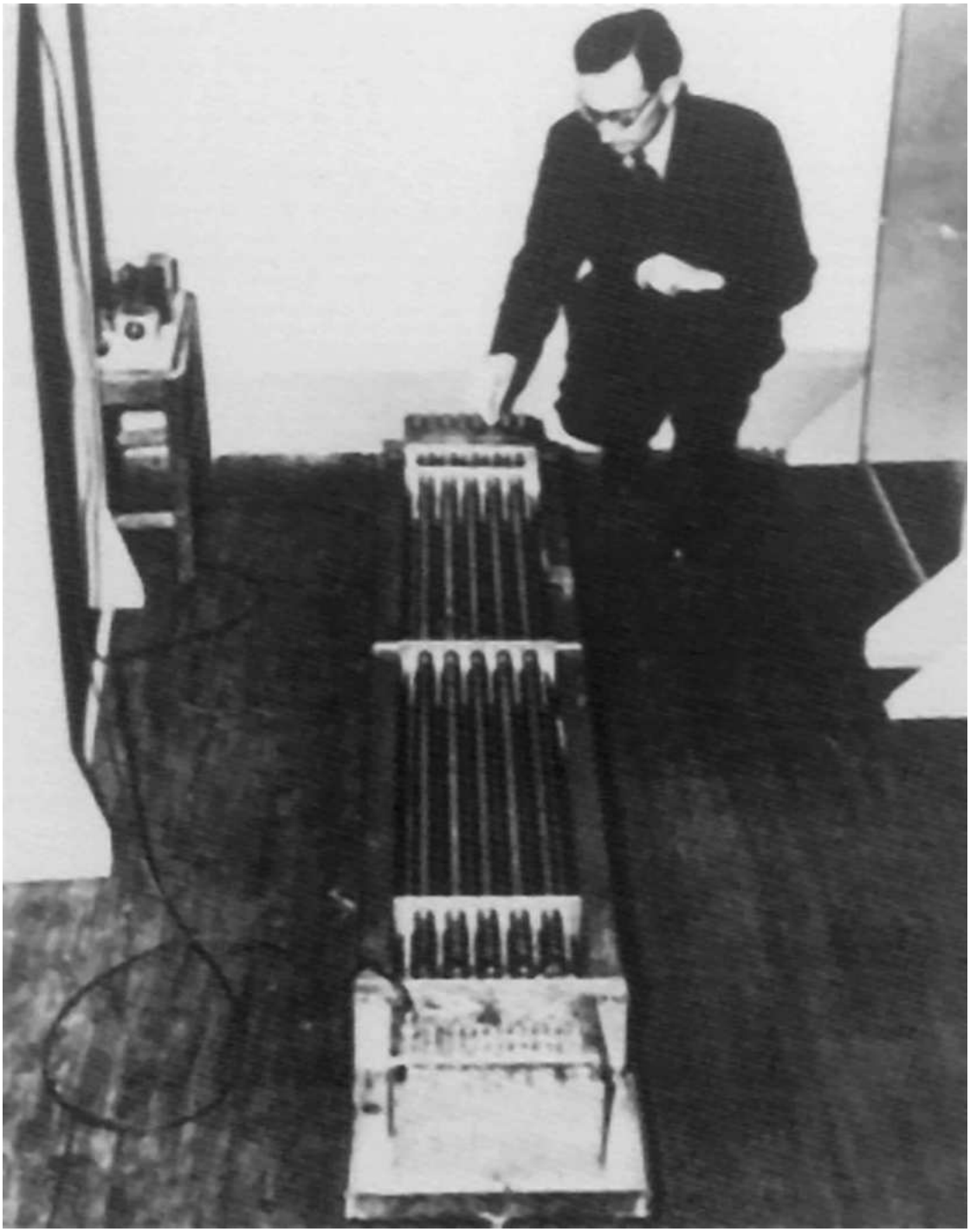


FIGURE E5.17.1 The mercury delay lines in the EDSAC.
This technology made it possible to build the first stored-program computer. The young engineer in this photograph is none other than Maurice Wilkes, the lead architect of the EDSAC.

Despite the tremendous advance offered by the mercury delay lines, they were terribly unreliable and still rather expensive. The

breakthrough came with the invention of core memory by J. Forrester at MIT as part of the Whirlwind project in the early 1950s (see Figure e5.17.2). Core memory uses a ferrite core, which can be magnetized, and once magnetized, it acts as a store (just as a magnetic recording tape stores information). A set of wires running through the center of the core, which had a dimension of 0.1–1.0 millimeters, makes it possible to read the value stored on any ferrite core. The Whirlwind eventually included a core memory with 2048 16-bit words, or 32 Kbits. Core memory was a tremendous advance: it was cheaper, faster, considerably more reliable, and had higher density. Core memory was so much better than the alternatives that it became the dominant memory technology only a few years after its invention and remained so for nearly 20 years.

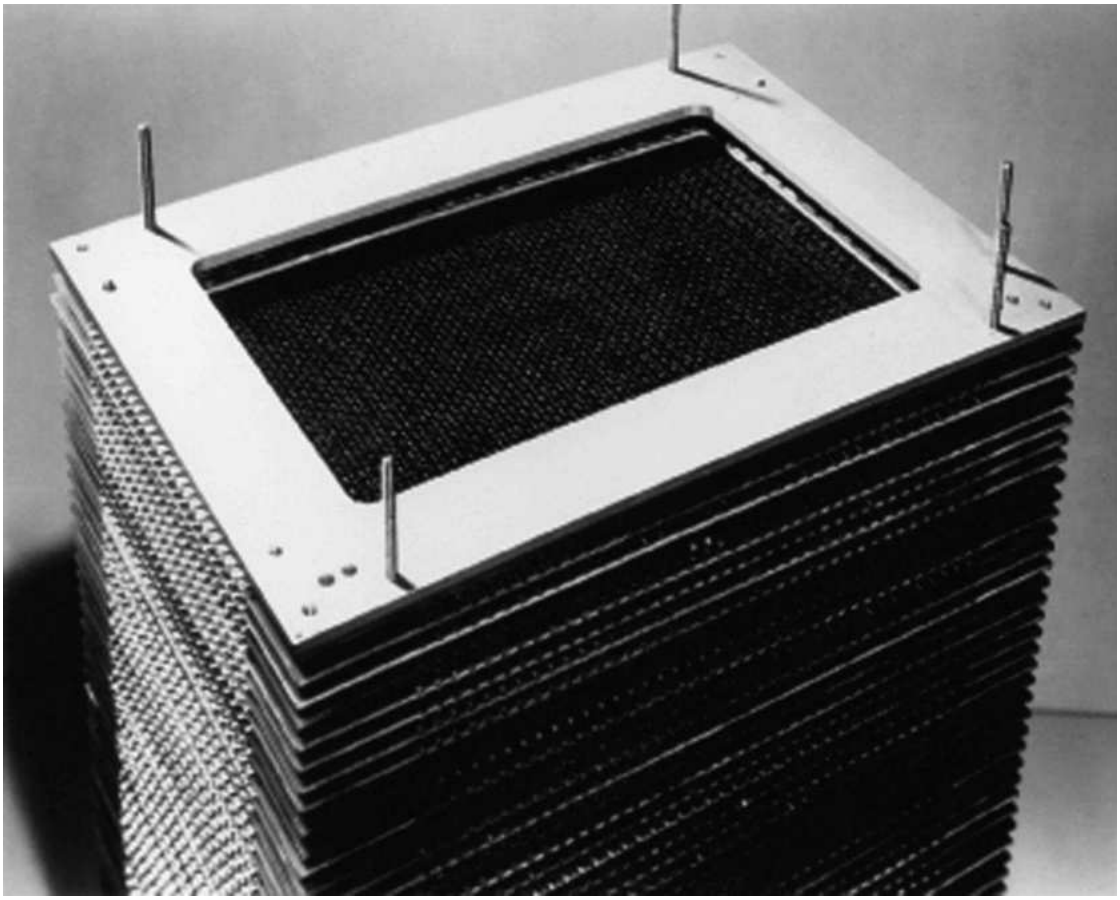


FIGURE E5.17.2 A core memory plane from the Whirlwind containing 256 cores arranged in a 16×16 array.
Core memory was invented for the Whirlwind, which was used for air defense problems, and is now on display at the Smithsonian. (Incidentally, Ken Olsen, the founder of Digital and its president for 20 years, built the computer that tested these core memories; it was his first computer.)

The technology that replaced core memory was the same one that we now use both for logic and for memory: the integrated circuit. While registers were built out of transistorized memory in the 1960s, and IBM computers used transistorized memory for microcode store and caches in 1970, building main memory out of transistors remained prohibitively expensive until the development of the integrated circuit. With the integrated circuit, it became possible to build a DRAM (dynamic random access memory—see Appendix A for a description). The first DRAMs were built at Intel in 1970, and the computers using DRAM memories (as a high-speed

option to core) came shortly thereafter; they used 1 Kbit DRAMs. In fact, computer folklore says that Intel developed the microprocessor partly to help sell more DRAM. Figure e5.17.3 shows an early DRAM board. By the late 1970s, core memory had become a historical curiosity. Just as core memory technology had allowed a tremendous expansion in memory size, DRAM technology allowed a comparable expansion. In the 1990s, many personal computers had as much memory as the largest computers using core memory ever had.

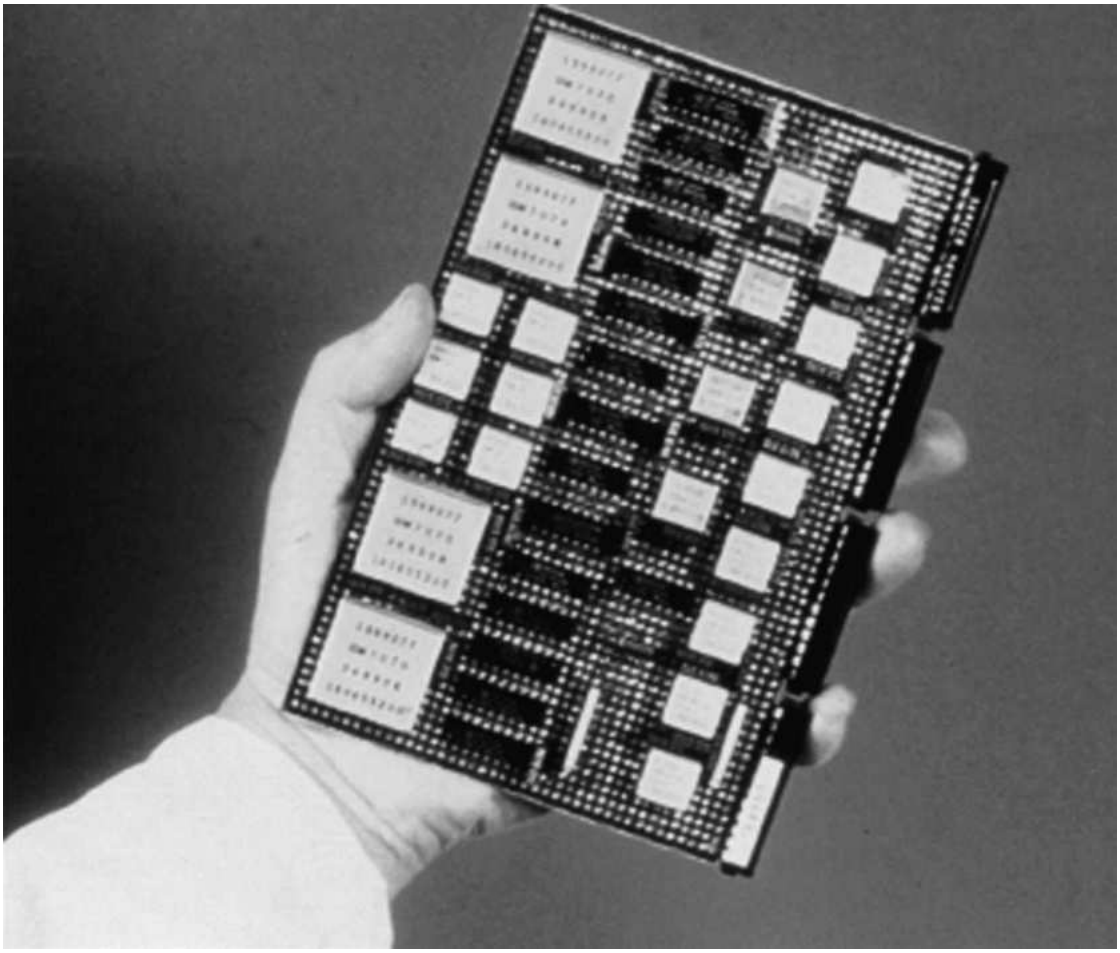


FIGURE E5.17.3 An early DRAM board. This board uses 18 Kbit chips.

Nowadays, DRAMs are typically packaged with multiple chips on a little board called a DIMM (dual inline memory module). The SIMM (single inline memory module) shown in Figure e5.17.4 contains a total of 1 MB and sold for about $5 in 1997. As of 2004, DIMMs were available with up to 1024 MB and sold for about $100. While DRAMs will remain the dominant memory technology for some time to come, innovations in the packaging of DRAMs to provide both higher bandwidth and greater density are ongoing.

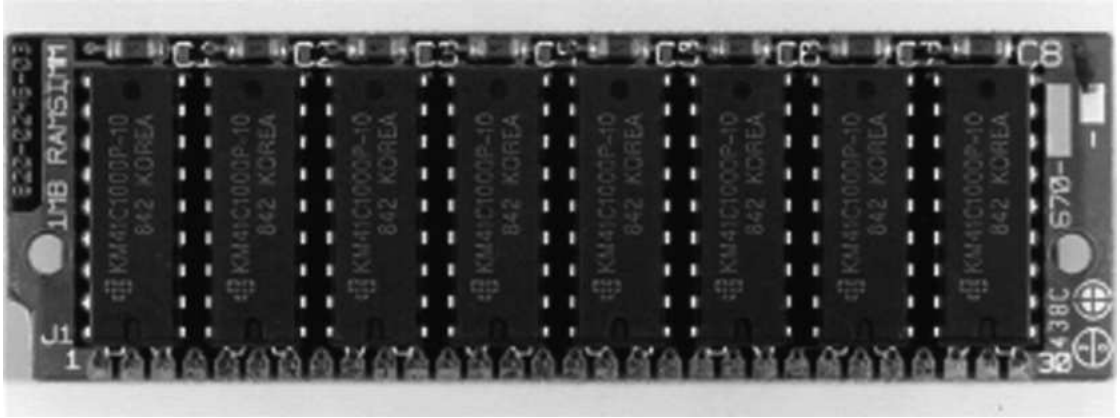
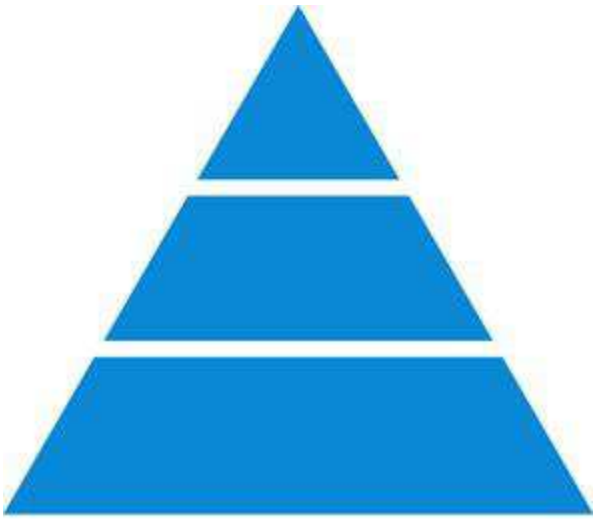


FIGURE E5.17.4 A 1 MB SIMM, built in 1986, using 1 Mbit chips.
This SIMM sold for about $5/MB in 1997. As of 2006, most main memory is packed in DIMMs similar to this, though using much higher-density memory chips (1 Gbit).

The Development of Memory Hierarchies

Although the pioneers of computing foresaw the need for a **memory hierarchy** and coined the term, the automatic management of two levels was first proposed by Kilburn and his colleagues and demonstrated at the University of Manchester with the Atlas computer, which implemented virtual memory. This was the year *before* the IBM 360 was announced. IBM planned to include virtual memory with the next generation (System/370), but the OS/360 operating system wasn't up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this computer that the term *translation-lookaside buffer* was coined. All but some embedded computers use virtual memory today.



HIERARCHY

The problems of inadequate address space have plagued designers repeatedly. The architects of the PDP-11 identified a small address space as the only architectural mistake from which it is difficult to recover. When the PDP-11 was designed, core memory densities were increasing at a very slow rate, and the competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go through the 16-bit datapath twice—hence, the decision to add just 4 more address bits than the predecessor of the PDP-11, to 16 from 12. The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses. Unfortunately, the expansion effort was greatly complicated by programmers who stored extra information in the upper 8 "unused" address bits. The wider address lasted until 2000, when IBM expanded the architecture to 64 bits in the z-series.

Running out of address space has often been the cause of death for an architecture, while other architectures have managed to make the transition to a larger address space. For example, the PDP-11, a 16-bit computer, was replaced by the 32-bit VAX. The 80386 extended the 80286 architecture from a segmented 24-bit address space to a flat 32-bit address space in 1985. In the 1990s, several RISC instruction sets made the transition from 32-bit addressing to 64-bit addressing by providing a compatible

extension of their instruction sets. MIPS was the first to do so. A decade later, Intel and HP announced the IA-64 in large part to provide a 64-bit address successor to the 32-bit Intel IA-32 and HP Precision architectures. The evolutionary AMD64 won that battle versus the revolutionary IA-64, and all but a few thousand of the 64-bit address computers from Intel are based on the x86.

Many of the early ideas in memory hierarchies originated in England. Just a few years after the Atlas paper, Wilkes [1965] published the first paper describing the concept of a cache, calling it a "slave":

> *The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.*

This two-page paper describes a direct-mapped cache. Although this was the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge by Scarrott and described at the 1965 IFIP Congress. It was based on tunnel diode memory, the fastest form of memory available at the time.

Subsequent to that publication, IBM started a project that led to the first commercial computer with a cache, the IBM 360/85. Gibson at IBM recognized that memory-accessing behavior would have a significant impact on performance. He described how to measure program behavior and cache behavior and showed that the miss rate varies between programs. Using a sample of 20 programs (each with 3 million references—an incredible number for that time), Gibson analyzed the effectiveness of caches using average memory access time as the metric. Conti, Gibson, and Pitowsky described the resulting performance of the 360/85 in the first paper to use the term *cache* in 1968. Since this early work, it has become clear that caches are one of the most important ideas not only in computer architecture but in software systems as well. The idea of caching has found applications in operating systems, networking systems, databases, and compilers, to name a few. There are thousands of papers on the topic of caching, and it continues to be a popular area of research.

One of the first papers on nonblocking caches was by Kroft in 1981, who may have coined the term. He later explained that he was the first to design a computer with a cache at Control Data Corporation, and when using old concepts for new mechanisms, he hit upon the idea of allowing his two-ported cache to continue to service other accesses on a miss.

Multilevel caches were the inevitable resolution to the lack of improvement in main memory latency and the higher clock rates of microprocessors. Only those in the field for a while are surprised by the size of some second- or third-level caches, as they are larger than main memories of past machines. The other surprise is that the number of levels is continually increasing, even on a single-chip microprocessor.

Disk Storage

In 1956, IBM developed the first disk storage system with both moving heads and multiple disk surfaces in San Jose, helping to seed the birth of the magnetic storage industry in the southern end of Silicon Valley. Reynold B. Johnson led the development of the IBM 305 RAMAC (Random Access Method of Accounting and Control). It could store 5 million characters (5 MB) of data on 50 disks, each 24 inches in diameter. The RAMAC is shown in Figures e5.17.5 and e5.17.6. Although the disk pioneers would be amazed at the size, cost, and capacity of modern disks, the basic mechanical design is the same as the RAMAC.

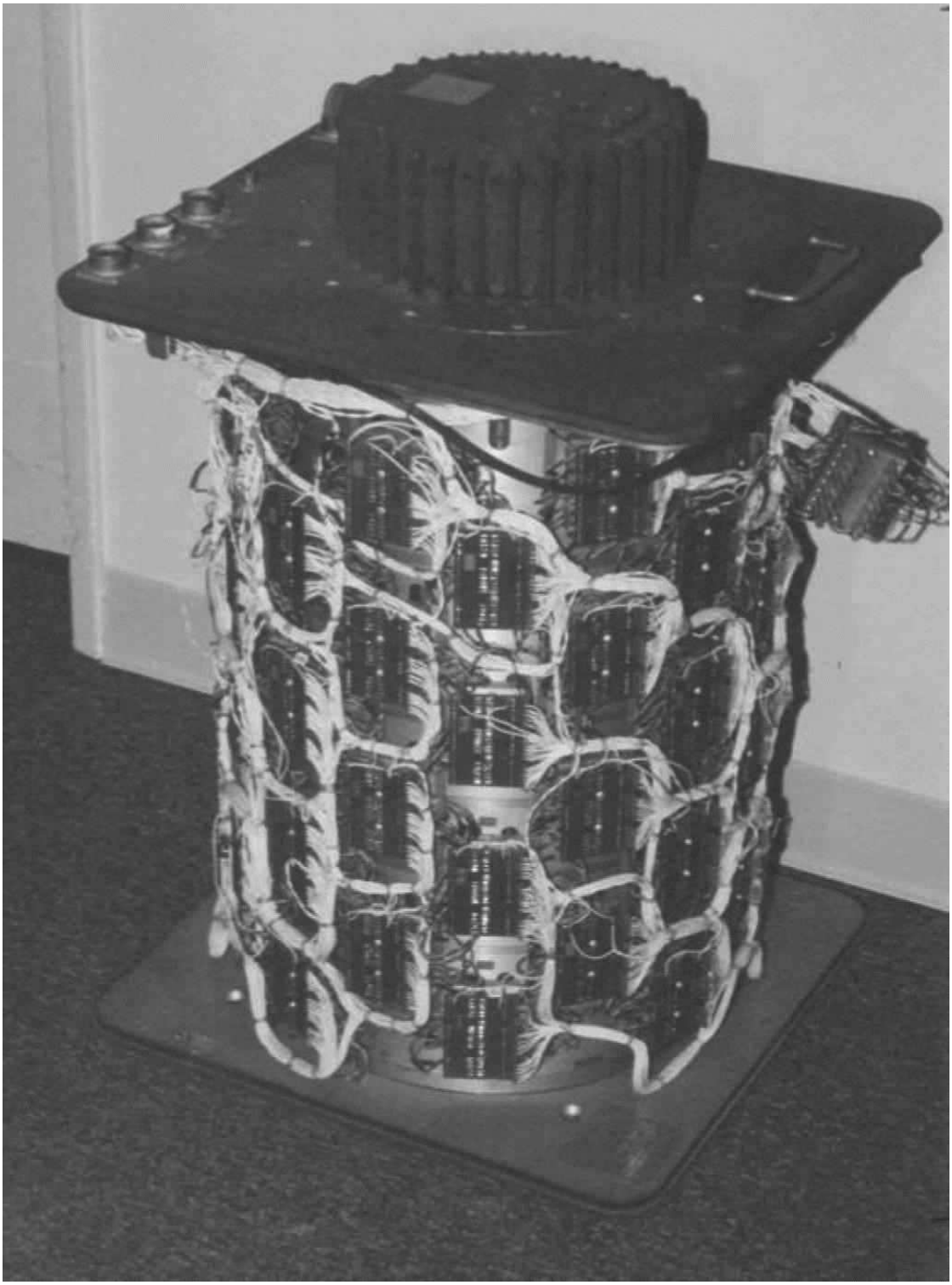


FIGURE E5.17.5 A magnetic drum made by Digital Development Corporation in the 1960s and used on a CDC machine.
The electronics supporting the read/write heads can be seen on the outside of the drum.

FIGURE E5.17.6 The RAMAC disk drive from IBM, made in 1956, was the first disk drive with a moving head and the first with multiple platters.
The IBM storage technology Web site has a discussion of IBM's major contributions to storage technology.

Moving-head disks quickly became the dominant high-speed magnetic storage, though their high cost meant that magnetic tape continued to be used extensively until the 1970s. The next key milestone for hard disks was the removable hard disk drive developed by IBM in 1962; this made it possible to share the expensive drive electronics and helped disks overtake tapes as the preferred storage medium. Figure e5.17.7 shows a removable disk drive and the multiplatter disk used in the drive. IBM also invented the floppy disk drive in 1970, originally to hold microcode for the IBM 370 series. Floppy disks became popular with the PC about 10 years later.

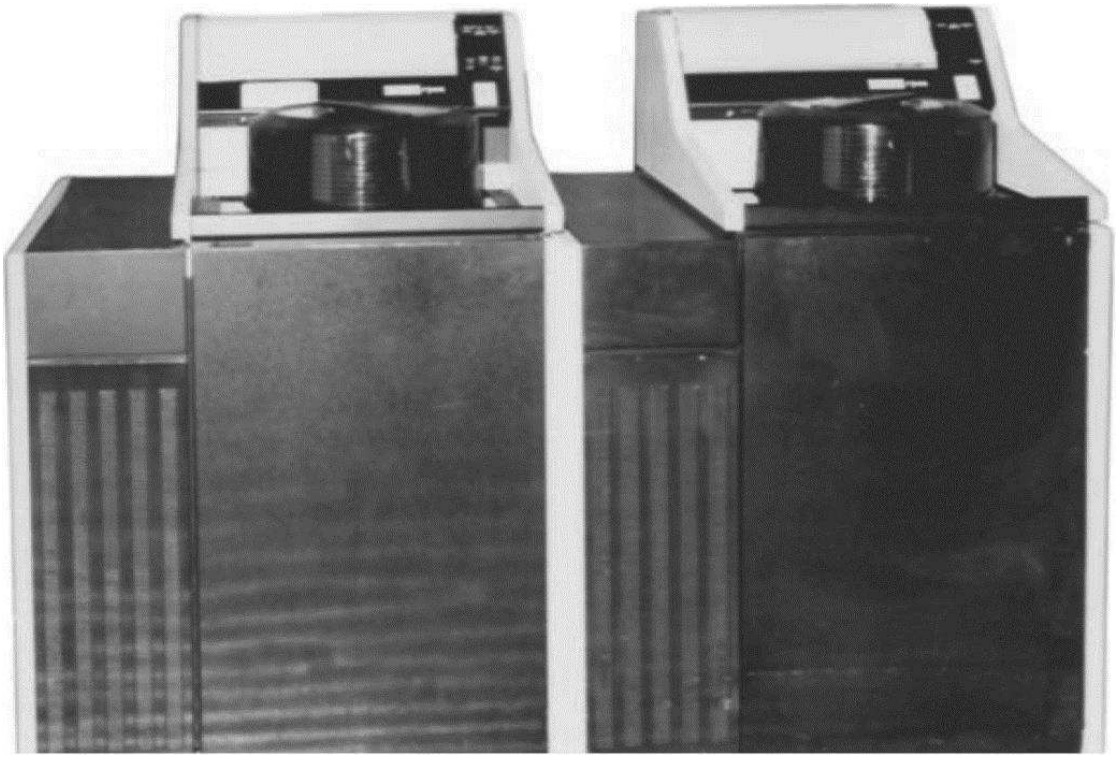


FIGURE E5.17.7 This is a DEC disk drive and the removable pack.
These disks became popular starting in the mid-1960s and dominated disk technology until Winchester drives in the late 1970s. This drive was made in the mid-1970s; each disk pack in this drive could hold 80 MB.

The sealed Winchester disk, which was developed by IBM in 1973, completely dominates disk technology today. Winchester disks benefited from two related properties. First, reductions in the cost of the disk electronics made it unnecessary to share the electronics and thus made nonremovable disks economical. Since the disk was fixed and could be in a sealed enclosure, both the environmental and control problems were greatly reduced, allowing significant gains in density. The first disk that IBM shipped had two spindles, each with a 30 MB disk; the moniker "30-30" for the disk led to the name Winchester. Winchester disks grew rapidly in popularity in the 1980s, completely replacing removable disks by the middle of that decade.

The historic role of IBM in the disk industry came to an end in 2002, when IBM sold its disk storage division to Hitachi. IBM continues to make storage subsystems, but it purchases its disk drives from others.

A Very Brief History of Flash Memory

Flash memory was invented by researchers at Toshiba in the 1980s. They invented both the NOR-based Flash memory in 1984 and the denser NAND-based Flash memory in 1989. The first use was in digital cameras, starting with the CompactFlash form factor for NOR Flash memory and the SmartMedia form factor for NAND Flash memory. Today, all digital cameras, cell phones, music players, and tablets rely on Flash memory, and an increasing fraction of laptops use flash memory instead of disk.

A Brief History of Databases

Although there had been data stores of punch cards and later magnetic tapes, the emergence of the magnetic disk led to modern databases.

In 1961, Charles Bachman at General Electric created a pioneering database management system called Integrated Data Store (IDS) to take advantage of the new magnetic disks. In 1971, Bachman and others published standards on how to manage databases using Cobol programs, named the Codasyl approach after the standards committee on which they served. Many companies offered Codasyl-compatible databases, but not IBM. IBM had introduced IMS in 1968, which was derived from IBM's work on the NASA Apollo project. Both Codasyl databases and IMS are classified as navigational databases because programs had to navigate through the data.

Ted Codd, a researcher at IBM, thought the navigational approach was wrong-headed. He recalled that people didn't write programs when dealing with the old punch card databases. Instead, they set up data flows through series of punch card machines that would perform simple functions like copy or sort. Once the card machines were set up, you just pushed all the cards through to get your results. In his view, users should only declare the type of data they were looking for and leave it up to computers to process it. In 1970, he published a new way to organize and access data called the relational model. It was based on set theory; data were independent of the implementation and users described what they were looking for in a declarative, nonprocedural language.

This paper led to considerable controversy within IBM, because it

already had a database product. Codd even arranged a public debate between him and Bachman, which led to internal criticism at IBM that Codd was undermining IMS. The good news was that the debate led researchers at IBM and U.C. Berkeley to try to demonstrate the viability of relational databases by building System R and Ingres.

System R in 1974–79 demonstrated its feasibility and, perhaps more importantly, created the Structured Query Language (SQL) that is still widely used today. However, these results were not sufficient to convince IBM, and some of the researchers left IBM to build relational databases for other companies.

Mike Stonebraker and Gene Wong were interested in geographic data systems, and in 1973 they decided to pursue relational databases. Rather than build on IBM mainframes, the Ingres project was built on DEC minicomputers and Unix. Ingres was important because it led to a company that tried to commercialize the ideas, because 1000 copies of its source code were openly distributed, and because it trained a generation of database developers and researchers. The code and people led to many other companies, including Sybase. Larry Ellison started Oracle by first reading the papers from the System R and Ingres groups and then by hiring people who worked on those projects. Microsoft later purchased a copy of Sybase sources that became the foundation of its SQL Server product.

Relational databases matured in the 1980s, with IBM developing its own relational databases, including DB2. The 1990s saw both the development of object-oriented databases, to address the impedance mismatch between databases and programming, and the evolution of parallel databases for analytic processing and data mining.

ACM showered awards on this community. The ACM Turing Award went to Charles Bachman in 1973 for his contributions via IDS and the Codasyl group. Codd won it in 1980 for the relational model. In 1988, the developers of System R (Donald Chamberlin, Jim Gray, Raymond Lorie, Gianfranco Putzolu, Patricia Selinger, and Irving Traiger) shared the ACM Systems Software Award with the developers of Ingres (Gerald Held, Michael Stonebraker, and Eugene Wong). Jim Gray won the Turing Award in 1998 for his contributions to transaction processing and databases. Stonebraker

won it in 2014 for contributions to the concepts and practices underlying modern database systems. Finally, the first two ACM SIGMOD Innovations Awards went to Stonebraker and Gray, and the 2002 and 2003 editions went to Selinger and Chamberlin.

RAID

The small-form-factor hard disks for PCs in the mid-1980s led a group at Berkeley to propose redundant arrays of inexpensive disks (RAID). This group had worked on the reduced instruction set computer effort and so expected much faster processors to become available. Their two questions were: What could be done with the small disks that accompanied their PCs? What could be done in the area of I/O to keep up with much faster processors? They argued to replace one large mainframe drive with 50 small drives, as you could get much greater performance with that many independent arms. The many small drives even offered savings in power consumption and floor space.

The downside of many disks was much lower MTTF. Hence, on their own they reasoned out the advantages of redundant disks and rotating parity to address how to get greater performance with many small drives yet have reliability as high as that of a single mainframe disk.

The problem they experienced when explaining their ideas was that some researchers had heard of disk arrays with some form of redundancy, and they didn't understand the Berkeley proposal. Hence, the first RAID paper [Patterson, Gibson, and Katz 1987] is not only a case for arrays of small-form-factor disk drives, but also something of a tutorial and classification of existing work on disk arrays. Mirroring (RAID 1) had long been used in fault-tolerant computers such as those sold by Tandem. Thinking Machines had arrays with 32 data disks and seven check disks using ECC for correction (RAID 2) in 1987, and Honeywell Bull had a RAID 2 product even earlier. Also, disk arrays with a single parity disk had been used in scientific computers in the same time frame (RAID 3). Their paper then described a single parity disk with support for sector accesses (RAID 4) and rotated parity (RAID 5). Chen et al. [1994] survey the original RAID ideas, commercial products, and other developments.

Unknown to the Berkeley group, engineers at IBM working on the AS/400 computer also came up with rotated parity to give greater reliability for a collection of large disks. IBM filed a patent on RAID 5 shortly before the Berkeley group submitted their paper. Patents for RAID 1, RAID 2, and RAID 3 from several companies predate the IBM RAID 5 patent, which has led to plenty of courtroom action.

EMC had been a supplier of DRAM boards for IBM computers, but around 1988 new policies from IBM made it nearly impossible for EMC to continue to sell IBM memory boards. The Berkeley paper crossed the desks of EMC executives, and so they decided to go after the market dominated by IBM disk storage products. As the paper advocated, their model was to use many small drives to compete with mainframe drives, and EMC announced a RAID product in 1990. It relied on mirroring (RAID 1) for reliability; RAID 5 products came much later for EMC. Over the next year, Micropolis offered a RAID 3 product; Compaq offered a RAID 4 product; and Data General, IBM, and NCR offered RAID 5 products.

The RAID ideas soon spread to the rest of the workstation and server industry. An article explaining RAID in *Byte* magazine led to RAID products being offered on desktop PCs, which was something of a surprise to the Berkeley group. They had focused on performance with good availability, but higher availability was attractive to the PC market.

Another surprise was the cost of the disk arrays. With redundant power supplies and fans, the ability to "hot-swap" a disk drive, the RAID hardware controller itself, the redundant disks, and so on, the first disk arrays cost many times the cost of the disks. Perhaps as a result, the "inexpensive" in RAID morphed into "independent." Many marketing departments and technical writers today know of RAID only as "redundant arrays of independent disks."

In 2004, more than 80% of the nondesktop drive sales were found in RAIDs. In recognition of their role, in 1999 Garth Gibson, Randy Katz, and David Patterson received the IEEE Reynold B. Johnson Information Storage Award "for the development of Redundant Arrays of Inexpensive Disks (RAID)."

Protection Mechanisms

Architectural support for protection has varied greatly over the past 20 years. In early computers, before virtual memory, protection was very simple at best. In the 1960s, more sophisticated mechanisms that supported different protection levels (called *rings*) were invented. In the late 1970s and early 1980s, very elaborate mechanisms for protection were devised and later built; these mechanisms supported a variety of powerful protection schemes that allowed controlled instances of sharing, in such a way that a process could share data while controlling exactly what was done to the data. The most powerful method, called *capabilities*, created a data object that described the access rights to some portion of memory. These capabilities could then be passed to other processes, thus granting access to the object described by the capability. Supporting this sophisticated protection mechanism was both complex and costly, because creation, copying, and manipulation of capabilities required a combination of operating system and hardware support. Recent computers all support a simpler protection scheme based on virtual memory, similar to that discussed in Section 5.7. Given current concerns about computer security due to the costs of worms and viruses, perhaps we will see a renaissance in protection research, potentially renewing interest in 20-year-old publications.

As mentioned in the text, system virtual machines were pioneered at IBM as part of its investigation into virtual memory. IBM's first computer with virtual memory was the IBM 360/67, introduced in 1967. IBM researchers wrote the program CP-67, which created the illusion of several independent 360 computers. They then wrote an interactive, single-user operating system called CMS that ran on these virtual machines. CP-67 led to the product VM/370, and today IBM sells z/VM for its mainframe computers.

A Brief History of Modern Operating Systems

MIT developed the first timesharing system, CTSS (Compatible Time-Sharing System), in 1961. John McCarthy is generally given credit for the idea of timesharing, but Fernando Corbato was the systems person who realized the concept in the form of the CTSS. CTSS allowed three people to share a machine, and its response time of minutes or seconds was a dramatic improvement over the

batch processing system it replaced. Moreover, it demonstrated the value of interactive computing.

Flush with the success of their first system, this group launched into their second system, MULTICS (Multiplexed Information and Computing Service). They included many innovations, such as strong protection, controlled sharing, and dynamic libraries. However, it suffered from the "second system effect." Fred Brooks, Jr. described the second system effect in his classic book about lessons learned from developing an operating system for the IBM mainframe, *The Mythical Man Month:*

> *When one is designing the successor to a relatively small, elegant, and successful system, there is a tendency to become grandiose in one's success and design an elephantine feature-laden monstrosity.*

MULTICS took sharing to a logical extreme to discover the issues, including that it was too extreme. MIT, General Electric, and later Bell Labs all tried to build an economical and useful system. Despite a great deal of time and money, they failed.

UC Berkeley was building its own timesharing system, Cal TSS. ("Cal" is a nickname for University of California.) The people leading that project included Peter Deutsch, Butler Lampson, Chuck Thacker, and Ken Thompson. They added paging virtual memory hardware to an SDS 920 and wrote an operating system for it. SDS sold this computer as the SDS-930, and it was the first commercially available timesharing system to have operational hardware and software. Thompson graduated and joined Bell Labs. The others founded Berkeley Computer Corporation (BCC), with the goal of selling time-sharing hardware and software. We'll pick up BCC later in the story, but for now let's follow Thompson.

At Bell Labs in 1971, Thompson led the development of a simple timesharing system that had some of the good ideas of MULTICS but left out many of the complex features. To demonstrate the contrast, it was first called UNICS. As they were joined by others at Bell Labs who had been burned from the MULTICS experience, it was renamed UNIX, with the *x* coming from Phoenix, the legendary bird that rose from the ashes.

Their result was the most elegant operating system ever built. Forced to live in the 16-bit address space of the DEC

minicomputers, it had an amazing amount of functionality per line of code. Major contributions were pipes, a uniform file system, a uniform process model, and the shell user interface that allowed users to connect programs together using pipes and files.

Dennis Ritchie joined the UNIX team in 1973 from MIT, where he had experience in MULTICS, which was written in a high-level language. Like prior operating systems, UNIX had been written in assembly language. Ritchie designed a language for system implementation called C, and it was used to make UNIX portable.

Between 1971 and 1976, Bell released six editions of the UNIX timesharing system. Thompson took a sabbatical at his alma mater and brought UNIX with him. Berkeley and many other universities began to use UNIX on the popular PDP-11 minicomputer.

When DEC announced the VAX, a 32-bit virtual address successor to the PDP-11, the question arose as to what operating system should be run. UNIX became the first operating system to be migrated to a different computer when it was ported to the VAX.

Students at Berkeley had one of the first VAXes, and they were soon adding features to UNIX for the VAX, such as paging and a very efficient implementation of the TCP/IP protocol. The Berkeley implementation of TCP/IP was notable not just because it was fast. It was essentially the *only* implementation of TCP/IP for years, since early implementations in most other operating systems consisted of copying the Berkeley code verbatim, with minimal changes to integrate into the local system.

The Advanced Research Project Agency (ARPA), which funded computer science research, asked a Stanford professor, Forrest Basket, to recommend which system the academic community should use: the DEC operating system VMS, led by David Cutler, or the Berkeley version of UNIX, led by a graduate student named Bill Joy. He recommended the latter, and Berkeley UNIX soon became the academic standard bearer.

The Berkeley Software Distribution (BSD) of UNIX, first released in 1978, was essentially one of the first open source movements. The sources were shipped with the tapes, and systems developers around the world learned their craft by studying the UNIX code.

BSD was also the first split of UNIX, because AT&T Bell Labs continued to develop UNIX on its own. This eventually led to a forest of UNIXes, as each company compiled the UNIX source code

for their architecture. Bill Joy graduated from Berkeley and helped found Sun Microsystems, so naturally Sun OS was based on BSD UNIX. Among the many UNIX flavors were Santa Cruz Operation UNIX, HP-UX, and IBM's AIX. AT&T and Sun attempted to unify UNIX by striking a deal whereby AT&T and Sun would combine forces and jointly develop AT&T UNIX. This led to an adverse reaction from HP, IBM, and others, because they did not want a competitor supplying their code, so they created the Open Source Foundation as a competing organization.

In addition to the UNIX variants from companies, public domain versions also proliferated. The BSD team at Berkeley rewrote substantial portions of UNIX so that they could distribute it without needing a license from AT&T. This eventually led to a lawsuit, which Berkeley won. BSD UNIX soon split into FreeBSD, NetBSD, and OpenBSD, provided by competing camps of developers. Apple's current operating system, OS X, is based on Free BSD.

Let's go back to Berkeley Computer Corporation. Alas, this effort was not commercially viable. About the same time as BCC was getting in trouble, Xerox hired Robert Taylor to build the computer science division of the new Xerox Palo Alto Research Center (PARC) in 1970. He had just returned from a tour of duty at ARPA, where he had funded the Berkeley research. He recruited Deutsch, Lampson, and Thacker from BCC to form the core of PARC's team: 11 of the initial 20 employees were from BCC, and they decided to build small computers for individuals rather than large computers for groups. This first personal computer, called the Alto, was built from the same technology as minicomputers, but it had a keyboard, mouse, graphical display, and windows. It popularized windows and led to many inventions, including client-server computing, the Ethernet, and print servers. It directly inspired the Macintosh, which was the successor to the popular Apple II.

IBM had long been interested in selling to the home, so the success of the Apple II led IBM to start a competing project. In contrast to its tradition, for this project IBM designed everything from components outside of the company. They selected the new 16-bit microprocessor from Intel, the 8086. (To lower costs, they started with the version with the 8-bit bus, called the 8088.) They visited Microsoft to see if this small company would be willing to sell their popular Basic interpreter and asked for recommendations

for an operating system. Gates volunteered that Microsoft could deliver both an interpreter and an operating system, as long as they were paid a royalty fee of between $10 and $50 for each copy rather than a flat fee. IBM agreed, provided Microsoft could meet their deadlines. Microsoft didn't have an operating system, nor the time and resources to build one, but Gates knew that a Seattle company had developed an operating system for the Intel 8086. Microsoft purchased QDOS (Quick and Dirty Operating System) for $15,000, made a small change and relabeled it MS-DOS. MS-DOS was a simple operating system without any modern features—no protection, no processes, and no virtual memory—in part because they believed it wasn't necessary for a personal computer.

Announced in 1980, the IBM PC became a tremendous success for IBM and the companies it relied upon. Microsoft sold 500,000 copies of MS-DOS by 1983, and the $10 million income allowed Microsoft to start new software projects.

After seeing a version of the Macintosh under development, Microsoft hired some people from PARC to lead its reply. The Macintosh was announced in 1984, and Windows was available on PCs the following year. It was originally an application that ran on top of DOS, but was later integrated with DOS and renamed Windows 2.0. Microsoft hired Cutler from DEC to lead the development of Windows NT, a new operating system. NT was a modern operating system with protection, processors, and so on and has much in common with DEC's VMS. Today's PC operating systems are more sophisticated than any of the timesharing systems of 20 years ago, yet they still suffer from the need to maintain compatibility with the crippled first PC operating systems such as MS-DOS.

The popularity of the PC led to a desire for a UNIX that ran on it. Many tried to develop one, but the most successful was written from scratch in 1991 by Linus Torvalds. In addition to making the source code available, like BSD, he allowed everyone to make changes and submit them for inclusion in his next release. Linux popularized open source development as we know it today, with such software getting hundreds of volunteers to test releases and add new features.

Many people in this story won awards for their roles in the development of modern operating systems. McCarthy received an

ACM Turing Award in 1971 in part for his contributions to timesharing. In 1983, Thompson and Ritchie received one for UNIX. The announcement said that "the genius of the UNIX system is its framework, which enables programmers to stand on the work of others." In 1990, Corbato received the Turing Award for his contributions to CTSS and MULTICS. Two years later, Lampson won it in part for his work on personal computing and operating systems.

Further Reading

Brooks, F.P. [1975]. *The mythical man-month*. Reading: Addison-Wesley

The classic book that explains the challenge of software engineering using IBM OS development as the example.

Cantin, J.F. and M.D. Hill [2001]. "Cache performance for selected SPEC CPU2000 benchmarks", *SIGARCH Computer Architecture News* 29:4 þ (September), 13–18.

A reference paper of cache miss rates for many cache sizes for the SPEC2000 benchmarks.

Chen, P.M., E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson [1994]. "RAID: High-performance, reliable secondary storage", *ACM Computing Surveys* 26:2 (June) 145–88.

A tutorial covering disk arrays and the advantages of such an organization.

Conti, C., D.H. Gibson, and S.H. Pitowsky [1968]. "Structural aspects of the System/360 Model 85, part I: General organization", *IBM Systems J.* 7:1, 2–14.

A classic paper that describes the first commercial computer to use a cache and its resulting performance.

Hennessy, J. and D. Patterson [2003]. Chapter 5 in *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers, San Francisco.

For more in-depth coverage of a variety of topics including protection, cache performance of out-of-order processors, virtually addressed caches, multilevel caches, compiler optimizations, additional latency tolerance mechanisms, and cache coherency.

Kilburn, T., D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner [1962]. "One-level storage system", *IRE Transactions on Electronic*

Computers EC-11(April), 223–335. Also appears in D.P. Siewiorek, C.G. Bell, and A. Newell [1982], Computer Structures: Principles and Examples, McGraw-Hill, New York, 135–48.

This classic paper is the first proposal for virtual memory.

LaMarca, A. and R.E. Ladner [1996]. "The influence of caches on the performance of heaps", *ACM J. of Experimental Algorithmics*, Vol. 1.

This paper shows the difference between complexity analysis of an algorithm, instruction count performance, and memory hierarchy for four sorting algorithms.

McCalpin, J.D. [1995]. *"STREAM: Sustainable Memory Bandwidth in High Performance Computers"*, https://www.cs.virginia.edu/stream/.

A widely used microbenchmark that measures the performance of the memory system behind the caches.

Patterson, D., G. Gibson, and R. Katz [1988]. *"A case for redundant arrays of inexpensive disks (RAID)"*, SIGMOD Conference, 109–116.

A classic paper that advocates arrays of smaller disks and introduces RAID levels.

Przybylski, S.A. [1990]. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann Publishers, San Francisco.

A thorough exploration of multilevel memory hierarchies and their performance.

Ritchie, D. [1984]. "The evolution of the UNIX time-sharing system", *AT&T Bell Laboratories Technical Journal* 1984, 1577–1593.

The history of UNIX from one of its inventors.

Ritchie, D.M. and K. Thompson [1978]. "The UNIX time-sharing system", *Bell System Technical Journal (August)*, 1991–2019.

A paper describing the most elegant operating system ever invented.

Silberschatz, A., P. Galvin, and G. Grange [2003]. *Operating System Concepts*, sixth edition, Addison-Wesley, Reading, MA.

An operating systems textbook with a thorough discussion of virtual memory, processes and process management, and protection issues.

Smith, A.J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

The classic survey paper on caches. This paper defined the terminology for the field and has served as a reference for many computer designers.

Smith, D.K. and R.C. Alexander [1988]. *Fumbling the Future: How*

Xerox Invented, Then Ignored, the First Personal Computer, Morrow, New York.

A popular book that explains the role of Xerox PARC in laying the foundation for today's computing, but which Xerox did not substantially benefit from.

Tanenbaum, A. [2001]. *Modern Operating Systems*, second edition, Upper Saddle River, Prentice Hall, NJ.

An operating system textbook with a good discussion of virtual memory.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation", *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–71.

The first classic paper on caches.

5.19 Exercises

Assume memory is byte addressable and words are 64 bits, unless specified otherwise.

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 64-bit integer.

```
for (I=0; I<8; I++)
 for (J=0; J<8000; J++)
  A[I][J]=B[I][0]+A[J][I];
```

5.1.1 [5] <§5.1> How many 64-bit integers can be stored in a 16-byte cache block?

5.1.2 [5] <§5.1> Which variable references exhibit temporal locality?

5.1.3 [5] <§5.1> Which variable references exhibit spatial locality? Locality is affected by both the reference order and data layout. The same computation can also be written below in Matlab, which differs from C in that it stores matrix elements within the same column contiguously in memory.

```
for I=1:8
 for J=1:8000
  A(I,J)=B(I,0)+A(J,I);
 end
end
```

5.1.4 [5] <§5.1> Which variable references exhibit temporal

locality?

5.1.5 [5] <§5.1> Which variable references exhibit spatial locality?

5.1.6 [15] <§5.1> How many 16-byte cache blocks are needed to store all 64-bit matrix elements being referenced using Matlab's matrix storage? How many using C's matrix storage? (Assume each row contains more than one element.)

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 64-bit memory address references, given as word addresses.

```
0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c,
0xba, 0xfd
```

5.2.1 [10] <§5.3> For each of these references, identify the binary word address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list whether each reference is a hit or a miss, assuming the cache is initially empty.

5.2.2 [10] <§5.3> For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

5.2.3 [20] <§§5.3, 5.4> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of eight words of data:

- C1 has 1-word blocks,
- C2 has 2-word blocks, and
- C3 has 4-word blocks.

5.3 By convention, a cache is named according to the amount of data it contains (i.e., a 4 KiB cache can hold 4 KiB of data); however, caches also require SRAM to store metadata such as tags and valid bits. For this exercise, you will examine how a cache's configuration affects the total amount of SRAM needed to implement it as well as the performance of the cache. For all parts, assume that the caches are byte addressable, and that addresses and words are 64 bits.

5.3.1 [10] <§5.3> Calculate the total number of bits required to implement a 32 KiB cache with two-word blocks.

5.3.2 [10] <§5.3> Calculate the total number of bits required to implement a 64 KiB cache with 16-word blocks. How much

bigger is this cache than the 32 KiB cache described in Exercise 5.3.1? (Notice that, by changing the block size, we doubled the amount of data without doubling the total size of the cache.)

5.3.3 [5] <§5.3> Explain why this 64 KiB cache, despite its larger data size, might provide slower performance than the first cache.

5.3.4 [10] <§§5.3, 5.4> Generate a series of read requests that have a lower miss rate on a 32 KiB two-way set associative cache than on the cache described in Exercise 5.3.1.

5.4 [15] <§5.3> Section 5.3 shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache). Assuming a 64-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address[63:54] XOR Block address[53:44]). Is it possible to use this to index a direct-mapped cache? If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why.

5.5 For a direct-mapped cache design with a 64-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
63–10	9–5	4–0

5.5.1 [5] <§5.3> What is the cache block size (in words)?

5.5.2 [5] <§5.3> How many blocks does the cache have?

5.5.3 [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?

Beginning from power on, the following byte-addressed cache references are recorded.

Address												
Hex	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
Dec	0	4	16	132	232	160	1024	30	140	3100	180	2180

5.5.4 [20] <§5.3> For each reference, list (1) its tag, index, and offset, (2) whether it is a hit or a miss, and (3) which bytes were replaced (if any).

5.5.5 [5] <§5.3> What is the hit ratio?

5.5.6 [5] <§5.3> List the final state of the cache, with each valid

entry represented as a record of <index, tag, data>. For example,

```
<0, 3, Mem[0xC00]-Mem[0xC1F]>
```

5.6 Recall that we have two write policies and two write allocation policies, and their combinations can be implemented either in L1 or L2 cache. Assume the following choices for L1 and L2 caches:

L1	L2
Write through, non-write allocate	Write back, write allocate

5.6.1 [5] <§§5.3, 5.8> Buffers are employed between different levels of memory hierarchy to reduce access latency. For this given configuration, list the possible buffers needed between L1 and L2 caches, as well as L2 cache and memory.

5.6.2 [20] <§§5.3, 5.8> Describe the procedure of handling an L1 write-miss, considering the components involved and the possibility of replacing a dirty block.

5.6.3 [20] <§§5.3, 5.8> For a multilevel exclusive cache configuration (a block can only reside in one of the L1 and L2 caches), describe the procedures of handling an L1 write-miss and an L1 read-miss, considering the components involved and the possibility of replacing a dirty block.

5.7 Consider the following program and cache behaviors.

Data Reads per 1000 Instructions	Data Writes per 1000 Instructions	Instruction Cache Miss Rate	Data Cache Miss Rate	Block Size (bytes)
250	100	0.30%	2%	64

5.7.1 [10] <§§5.3, 5.8> Suppose a CPU with a write-through, write-allocate cache achieves a CPI of 2. What are the read and write bandwidths (measured by bytes per cycle) between RAM and the cache? (Assume each miss generates a request for one block.)

5.7.2 [10] <§§5.3, 5.8> For a write-back, write-allocate cache, assuming 30% of replaced data cache blocks are dirty, what are the read and write bandwidths needed for a CPI of 2?

5.8 Media applications that play audio or video files are part of a class of workloads called "streaming" workloads (i.e., they bring in large amounts of data but do not reuse much of it). Consider a video streaming workload that accesses a 512 KiB working set

sequentially with the following word address stream:
`0, 1, 2, 3, 4, 5, 6, 7, 8, 9 …`

5.8.1 [10] <§§5.4, 5.8> Assume a 64 KiB direct-mapped cache with a 32-byte block. What is the miss rate for the address stream above? How is this miss rate sensitive to the size of the cache or the working set? How would you categorize the misses this workload is experiencing, based on the 3C model?

5.8.2 [5] <§§5.1, 5.8> Re-compute the miss rate when the cache block size is 16 bytes, 64 bytes, and 128 bytes. What kind of locality is this workload exploiting?

5.8.3 [10] <§5.13> "Prefetching" is a technique that leverages predictable address patterns to speculatively bring in additional cache blocks when a particular cache block is accessed. One example of prefetching is a stream buffer that prefetches sequentially adjacent cache blocks into a separate buffer when a particular cache block is brought in. If the data are found in the prefetch buffer, it is considered as a hit, moved into the cache, and the next cache block is prefetched. Assume a two-entry stream buffer; and, assume that the cache latency is such that a cache block can be loaded before the computation on the previous cache block is completed. What is the miss rate for the address stream above?

5.9 Cache block size (B) can affect both miss rate and miss latency. Assuming a machine with a base CPI of 1, and an average of 1.35 references (both instruction and data) per instruction, find the block size that minimizes the total miss latency given the following miss rates for various block sizes.

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%

5.9.1 [10] <§5.3> What is the optimal block size for a miss latency of 20 ×B cycles?

5.9.2 [10] <§5.3> What is the optimal block size for a miss latency of 24 +B cycles?

5.9.3 [10] <§5.3> For constant miss latency, what is the optimal block size?

5.10 In this exercise, we will look at the different ways capacity affects overall performance. In general, cache access time is

proportional to capacity. Assume that main memory accesses take 70 ns and that 36% of all instructions access data memory. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

5.10.1 [5] <§5.4> Assuming that the L1 hit time determines the cycle times for P1 and P2, what are their respective clock rates?

5.10.2 [10] <§5.4> What is the Average Memory Access Time for P1 and P2 (in cycles)?

5.10.3 [5] <§5.4> Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2? Which processor is faster? (When we say a "base CPI of 1.0", we mean that instructions complete in one cycle, unless either the instruction access or the data access causes a cache miss.)

For the next three problems, we will consider the addition of an L2 cache to P1 (to presumably make up for its limited L1 cache capacity). Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its local miss rate.

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

5.10.4 [10] <§5.4> What is the AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

5.10.5 [5] <§5.4> Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 with the addition of an L2 cache?

5.10.6 [10] <§5.4> What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P1 without an L2 cache?

5.10.7 [15] <§5.4> What would the L2 miss rate need to be in order for P1 with an L2 cache to be faster than P2 without an L2 cache?

5.11 This exercise examines the effect of different cache designs, specifically comparing associative caches to the direct-mapped caches from Section 5.4. For these exercises, refer to the sequence of word address shown below.

```
0x03, 0xb4, 0x2b, 0x02, 0xbe, 0x58, 0xbf, 0x0e, 0x1f, 0xb5,
0xbf, 0xba, 0x2e, 0xce
```

5.11.1 [10] <§5.4> Sketch the organization of a three-way set associative cache with two-word blocks and a total size of 48 words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.2 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.1. Assume a true LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset
- whether the reference is a hit or a miss, and
- which tags are in each way of the cache after the reference has been handled.

5.11.3 [5] <§5.4> Sketch the organization of a fully associative cache with one-word blocks and a total size of eight words. Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.4 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.3. Assume a true LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset,
- whether the reference is a hit or a miss, and
- the contents of the cache after each reference has been handled.

5.11.5 [5] <§5.4> Sketch the organization of a fully associative cache with two-word blocks and a total size of eight words.

Your sketch should have a style similar to Figure 5.18, but clearly show the width of the tag and data fields.

5.11.6 [10] <§5.4> Trace the behavior of the cache from Exercise 5.11.5. Assume an LRU replacement policy. For each reference, identify

- the binary word address,
- the tag,
- the index,
- the offset,
- whether the reference is a hit or a miss, and
- the contents of the cache after each reference has been handled.

5.11.7 [10] <§5.4> Repeat Exercise 5.11.6 using MRU (*most recently used*) replacement.

5.11.8 [15] <§5.4> Repeat Exercise 5.11.6 using the optimal replacement policy (i.e., the one that gives the lowest miss rate).

5.12 Multilevel caching is an important technique to overcome the limited amount of space that a first-level cache can provide while still maintaining its speed. Consider a processor with the following parameters:

Base CPI, No Memory Stalls	Processor Speed	Main Memory Access Time	First-Level Cache Miss Rate per Instruction**	Second-Level Cache, Direct-Mapped Speed	Miss Rate with Second-Level Cache, Direct-Mapped	Second-Level Cache, Eight-Way Set Associative Speed	Miss Rate with Second-Level Cache, Eight-Way Set Associative
1.5	2 GHz	100 ns	7%	12 cycles	3.5%	28 cycles	1.5%

**First Level Cache miss rate is per instruction. Assume the total number of L1 cache misses (instruction and data combined) is equal to 7% of the number of instructions.

5.12.1 [10] <§5.4> Calculate the CPI for the processor in the table using: 1) only a first-level cache, 2) a second-level direct-mapped cache, and 3) a second-level eight-way set associative cache. How do these numbers change if main memory access time doubles? (Give each change as both an absolute CPI and a percent change.) Notice the extent to which an L2 cache can hide the effects of a slow memory.

5.12.2 [10] <§5.4> It is possible to have an even greater cache hierarchy than two levels? Given the processor above with a second-level, direct-mapped cache, a designer wants to add a third-level cache that takes 50 cycles to access and will have a 13% miss rate. Would this provide better performance? In general, what are the advantages and disadvantages of adding

a third-level cache?

5.12.3 [20] <§5.4> In older processors, such as the Intel Pentium or Alpha 21264, the second level of cache was external (located on a different chip) from the main processor and the first-level cache. While this allowed for large second-level caches, the latency to access the cache was much higher, and the bandwidth was typically lower because the second-level cache ran at a lower frequency. Assume a 512 KiB off-chip second-level cache has a miss rate of 4%. If each additional 512 KiB of cache lowered miss rates by 0.7%, and the cache had a total access time of 50 cycles, how big would the cache have to be to match the performance of the second-level direct-mapped cache listed above?

5.13 *Mean time between failures* (MTBF), *mean time to replacement* (MTTR), and *mean time to failure* (MTTF) are useful metrics for evaluating the reliability and availability of a storage resource. Explore these concepts by answering the questions about a device with the following metrics:

MTTF	MTTR
3 Years	1 Day

5.13.1 [5] <§5.5> Calculate the MTBF for such a device.

5.13.2 [5] <§5.5> Calculate the availability for such a device.

5.13.3 [5] <§5.5> What happens to availability as the MTTR approaches 0? Is this a realistic situation?

5.13.4 [5] <§5.5> What happens to availability as the MTTR gets very high, i.e., a device is difficult to repair? Does this imply the device has low availability?

5.14 This exercise examines the *single error correcting, double error detecting* (SEC/DED) Hamming code.

5.14.1 [5] <§5.5> What is the minimum number of parity bits required to protect a 128-bit word using the SEC/DED code?

5.14.2 [5] <§5.5> Section 5.5 states that modern server memory modules (DIMMs) employ SEC/DED ECC to protect each 64 bits with 8 parity bits. Compute the cost/performance ratio of this code to the code from Exercise 5.14.1. In this case, cost is the relative number of parity bits needed while performance is the relative number of errors that can be corrected. Which is better?

5.14.3 [5] <§5.5> Consider a SEC code that protects 8 bit words with 4 parity bits. If we read the value 0x375, is there an error? If so, correct the error.

5.15 For a high-performance system such as a B-tree index for a database, the page size is determined mainly by the data size and disk performance. Assume that, on average, a B-tree index page is 70% full with fix-sized entries. The utility of a page is its B-tree depth, calculated as $\log_2$(entries). The following table shows that for 16-byte entries, and a 10-year-old disk with a 10 ms latency and 10 MB/s transfer rate, the optimal page size is 16 K.

Page Size (KiB)	Page Utility or B-Tree Depth (Number of Disk Accesses Saved)	Index Page Access Cost (ms)	Utility/Cost
2	6.49 (or $\log_2(2048/16 \times 0.7)$)	10.2	0.64
4	7.49	10.4	0.72
8	8.49	10.8	0.79
16	9.49	11.6	0.82
32	10.49	13.2	0.79
64	11.49	16.4	0.70
128	12.49	22.8	0.55
256	13.49	35.6	0.38

5.15.1 [10] <§5.7> What is the best page size if entries now become 128 bytes?

5.15.2 [10] <§5.7> Based on Exercise 5.15.1, what is the best page size if pages are half full?

5.15.3 [20] <§5.7> Based on Exercise 5.15.2, what is the best page size if using a modern disk with a 3 ms latency and 100 MB/s transfer rate? Explain why future servers are likely to have larger pages.

Keeping "frequently used" (or "hot") pages in DRAM can save disk accesses, but how do we determine the exact meaning of "frequently used" for a given system? Data engineers use the cost ratio between DRAM and disk access to quantify the reuse time threshold for hot pages. The cost of a disk access is $Disk/accesses_per_sec, while the cost to keep a page in DRAM is $DRAM_MiB/page_size. The typical DRAM and disk costs and typical database page sizes at several time points are listed below:

Year	DRAM Cost ($/MiB)	Page Size (KiB)	Disk Cost ($/disk)	Disk Access Rate (access/sec)
1987	5000	1	15,000	15
1997	15	8	2000	64
2007	0.05	64	80	83

5.15.4 [20] <§5.7> What other factors can be changed to keep using the same page size (thus avoiding software rewrite)? Discuss their likeliness with current technology and cost trends.

5.16 As described in Section 5.7, virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. This exercise shows how this table must be updated as addresses are accessed. The following data constitute a stream of virtual byte addresses as seen on a system. Assume 4 KiB pages, a four-entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the next largest page number.

Decimal	4669	2227	13916	34587	48870	12608	49225
hex	0x123d	0x08b3	0x365c	0x871b	0xbee6	0x3140	0xc049

TLB

Valid	Tag	Physical Page Number	Time Since Last Access
1	0xb	12	4
1	0x7	4	1
1	0x3	6	3
0	0x4	9	7

Page table

Index	Valid	Physical Page or in Disk

0	1	5
1	0	Disk
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
a	1	3
b	1	12

5.16.1 [10] <§5.7> For each access shown above, list
- whether the access is a hit or miss in the TLB,
- whether the access is a hit or miss in the page table,
- whether the access is a page fault,
- the updated state of the TLB.

5.16.2 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 16 KiB pages instead of 4 KiB pages. What would be some of the advantages of having a larger page size? What are some of the disadvantages?

5.16.3 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 4 KiB pages and a two-way set associative TLB.

5.16.4 [15] <§5.7> Repeat Exercise 5.16.1, but this time use 4 KiB pages and a direct mapped TLB.

5.16.5 [10] <§§5.4, 5.7> Discuss why a CPU must have a TLB for high performance. How would virtual memory accesses be handled if there were no TLB?

5.17 There are several parameters that affect the overall size of the page table. Listed below are key page table parameters.

Virtual Address Size	Page Size	Page Table Entry Size
32 bits	8 KiB	4 bytes

5.17.1 [5] <§5.7> Given the parameters shown above, calculate the maximum possible page table size for a system running five processes.

5.17.2 [10] <§5.7> Given the parameters shown above, calculate the total page table size for a system running five applications that each utilize half of the virtual memory available, given a two-level page table approach with up to 256 entries at the 1st level. Assume each entry of the main page table is 6 bytes.

Calculate the minimum and maximum amount of memory required for this page table.

5.17.3 [10] <§5.7> A cache designer wants to increase the size of a 4 KiB virtually indexed, physically tagged cache. Given the page size shown above, is it possible to make a 16 KiB direct-mapped cache, assuming two 64-bit words per block? How would the designer increase the data size of the cache?

5.18 In this exercise, we will examine space/time optimizations for page tables. The following list provides parameters of a virtual memory system.

Virtual Address (bits)	Physical DRAM Installed	Page Size	PTE Size (byte)
43	16 GiB	4 KiB	4

5.18.1 [10] <§5.7> For a single-level page table, how many *page table entries* (PTEs) are needed? How much physical memory is needed for storing the page table?

5.18.2 [10] <§5.7> Using a multi-level page table can reduce the physical memory consumption of page tables by only keeping active PTEs in physical memory. How many levels of page tables will be needed if the segment tables (the upper-level page tables) are allowed to be of unlimited size? How many memory references are needed for address translation if missing in TLB?

5.18.3 [10] <§5.7> Suppose the segments are limited to the 4 KiB page size (so that they can be paged). Is 4 bytes large enough for all page table entries (including those in the segment tables?

5.18.4 [10] <§5.7> How many levels of page tables are needed if the segments are limited to the 4 KiB page size?

5.18.5 [15] <§5.7> An inverted page table can be used to further optimize space and time. How many PTEs are needed to store the page table? Assuming a hash table implementation, what are the common case and worst case numbers of memory references needed for servicing a TLB miss?

5.19 The following table shows the contents of a four-entry TLB.

Entry-ID	Valid	VA Page	Modified	Protection	PA Page
1	1	140	1	RW	30
2	0	40	0	RX	34
3	1	200	1	RO	32
4	1	280	0	RW	31

5.19.1 [5] <§5.7> Under what scenarios would entry 3's valid bit be set to zero?

5.19.2 [5] <§5.7> What happens when an instruction writes to VA page 30? When would a software managed TLB be faster than a hardware managed TLB?

5.19.3 [5] <§5.7> What happens when an instruction writes to VA page 200?

5.20 In this exercise, we will examine how replacement policies affect miss rate. Assume a two-way set associative cache with four one-word blocks. Consider the following word address sequence: 0, 1, 2, 3, 4, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0. Consider the following address sequence: 0, 2, 4, 8, 10, 12, 14, 16, 0

5.20.1 [5] <§§5.4, 5.8> Assuming an LRU replacement policy, which accesses are hits?

5.20.2 [5] <§§5.4, 5.8> Assuming an MRU (*most recently used*) replacement policy, which accesses are hits?

5.20.3 [5] <§§5.4, 5.8> Simulate a random replacement policy by flipping a coin. For example, "heads" means to evict the first block in a set and "tails" means to evict the second block in a set. How many hits does this address sequence exhibit?

5.20.4 [10] <§§5.4, 5.8> Describe an optimal replacement policy for this sequence. Which accesses are hits using this policy?

5.20.5 [10] <§§5.4, 5.8> Describe why it is difficult to implement a cache replacement policy that is optimal for all address sequences.

5.20.6 [10] <§§5.4, 5.8> Assume you could make a decision upon each memory reference whether or not you want the requested

address to be cached. What effect could this have on miss rate?

5.21 One of the biggest impediments to widespread use of virtual machines is the performance overhead incurred by running a virtual machine. Listed below are various performance parameters and application behavior.

Base CPI	Privileged O/S accesses per 10,000 instructions	Overhead to trap to the guest O/S	Overhead to trap to VMM	I/O access per 10,000 instructions	I/O access time (includes time to trap to guest O/S)
1.5	120	15 cycles	175 cycles	30	1100 cycles

5.21.1 [10] <§5.6> Calculate the CPI for the system listed above assuming that there are no accesses to I/O. What is the CPI if the VMM overhead doubles? If it is cut in half? If a virtual machine software company wishes to limit the performance degradation to 10%, what is the longest possible penalty to trap to the VMM?

5.21.2 [15] <§5.6> I/O accesses often have a large effect on overall system performance. Calculate the CPI of a machine using the performance characteristics above, assuming a non-virtualized system. Calculate the CPI again, this time using a virtualized system. How do these CPIs change if the system has half the I/O accesses?

5.22 [15] <§§5.6, 5.7> Compare and contrast the ideas of virtual memory and virtual machines. How do the goals of each compare? What are the pros and cons of each? List a few cases where virtual memory is desired, and a few cases where virtual machines are desired.

5.23 [10] <§5.6> Section 5.6 discusses virtualization under the assumption that the virtualized system is running the same ISA as the underlying hardware. However, one possible use of virtualization is to emulate non-native ISAs. An example of this is QEMU, which emulates a variety of ISAs such as MIPS, SPARC, and PowerPC. What are some of the difficulties involved in this kind of virtualization? Is it possible for an emulated system to run faster than on its native ISA?

5.24 In this exercise, we will explore the control unit for a cache controller for a processor with a write buffer. Use the finite state machine found in Figure 5.39 as a starting point for designing your own finite state machines. Assume that the cache controller is for the simple direct-mapped cache described on page 453 (Figure 5.39 in Section 5.9), but you will add a write buffer with a

capacity of one block.

Recall that the purpose of a write buffer is to serve as temporary storage so that the processor doesn't have to wait for two memory accesses on a dirty miss. Rather than writing back the dirty block before reading the new block, it buffers the dirty block and immediately begins reading the new block. The dirty block can then be written to main memory while the processor is working.

5.24.1 [10] <§§5.8, 5.9> What should happen if the processor issues a request that *hits* in the cache while a block is being written back to main memory from the write buffer?

5.24.2 [10] <§§5.8, 5.9> What should happen if the processor issues a request that *misses* in the cache while a block is being written back to main memory from the write buffer?

5.24.3 [30] <§§5.8, 5.9> Design a finite state machine to enable the use of a write buffer.

5.25 Cache coherence concerns the views of multiple processors on a given cache block. The following data show two processors and their read/write operations on two different words of a cache block X (initially X[0]=X[1]=0).

P1	P2
`X[0] ++; X[1] = 3;`	`X[0] = 5; X[1] +=2;`

5.25.1 [15] <§5.10> List the possible values of the given cache block for a correct cache coherence protocol implementation. List at least one more possible value of the block if the protocol doesn't ensure cache coherency.

5.25.2 [15] <§5.10> For a snooping protocol, list a valid operation sequence on each processor/cache to finish the above read/write operations.

5.25.3 [10] <§5.10> What are the best-case and worst-case numbers of cache misses needed to execute the listed read/write instructions?

Memory consistency concerns the views of multiple data items. The following data show two processors and their read/write operations on different cache blocks (A and B initially 0).

P1	P2
`A = 1; B = 2; A+=2; B++;`	`C = B; D = A;`

5.25.4 [15] <§5.10> List the possible values of C and D for all

implementations that ensure both consistency assumptions on page 455.

5.25.5 [15] <§5.10> List at least one more possible pair of values for C and D if such assumptions are not maintained.

5.25.6 [15] <§§5.3, 5.10> For various combinations of write policies and write allocation policies, which combinations make the protocol implementation simpler?

5.26 Chip multiprocessors (CMPs) have multiple cores and their caches on a single chip. CMP on-chip L2 cache design has interesting trade-offs. The following table shows the miss rates and hit latencies for two benchmarks with private vs. shared L2 cache designs. Assume the L1 cache has a 3% miss rate and a 1-cycle access time.

	Private	Shared
Benchmark A miss rate	10%	4%
Benchmark B miss rate	2%	1%

Assume the following hit latencies:

Private Cache	Shared Cache	Memory
5	20	180

5.26.1 [15] <§5.13> Which cache design is better for each of these benchmarks? Use data to support your conclusion.

5.26.2 [15] <§5.13> Off-chip bandwidth becomes the bottleneck as the number of CMP cores increases. How does this bottleneck affect private and shared cache systems differently? Choose the best design if the latency of the first off-chip link doubles.

5.26.3 [10] <§5.13> Discuss the pros and cons of shared vs. private L2 caches for both single-threaded, multi-threaded, and multiprogrammed workloads, and reconsider them if having on-chip L3 caches.

5.26.4 [10] <§5.13> Would a non-blocking L2 cache produce more improvement on a CMP with a shared L2 cache or a private L2 cache? Why?

5.26.5 [10] <§5.13> Assume new generations of processors double the number of cores every 18 months. To maintain the same level of per-core performance, how much more off-chip memory bandwidth is needed for a processor released in three

years?

5.26.6 [15] <§5.13> Consider the entire memory hierarchy. What kinds of optimizations can improve the number of concurrent misses?

5.27 In this exercise we show the definition of a web server log and examine code optimizations to improve log processing speed. The data structure for the log is defined as follows:

```
struct entry {
 int srcIP; // remote IP address
 char URL[128]; // request URL (e.g., "GET index.html")
 long long refTime; // reference time
 int status; // connection status
 char browser[64]; // client browser name
} log [NUM_ENTRIES];
```

Assume the following processing function for the log:

```
topK_sourceIP (int hour);
```

This function determines the most frequently observed source IPs during the given hour.

5.27.1 [5] <§5.15> Which fields in a log entry will be accessed for the given log processing function? Assuming 64-byte cache blocks and no prefetching, how many cache misses per entry does the given function incur on average?

5.27.2 [5] <§5.15> How can you reorganize the data structure to improve cache utilization and access locality?

5.27.3 [10] <§5.15> Give an example of another log processing function that would prefer a different data structure layout. If both functions are important, how would you rewrite the program to improve the overall performance? Supplement the discussion with code snippet and data.

5.28 For the problems below, use data from "Cache Performance for SPEC CPU2000 Benchmarks" (http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/) for the pairs of benchmarks shown in the following table.

a.	Mesa/gcc
b.	mcf/swim

5.28.1 [10] <§5.15> For 64 KiB data caches with varying set associativities, what are the miss rates broken down by miss types (cold, capacity, and conflict misses) for each benchmark?

5.28.2 [10] <§5.15> Select the set associativity to be used by a 64 KiB L1 data cache shared by both benchmarks. If the L1 cache has to be directly mapped, select the set associativity for the 1 MiB L2 cache.

5.28.3 [20] <§5.15> Give an example in the miss rate table where higher set associativity actually increases miss rate. Construct a cache configuration and reference stream to demonstrate this.

5.29 To support multiple virtual machines, two levels of memory virtualization are needed. Each virtual machine still controls the mapping of *virtual address* (VA) to *physical address* (PA), while the hypervisor maps the *physical address* (PA) of each virtual machine to the actual *machine address* (MA). To accelerate such mappings, a software approach called "shadow paging" duplicates each virtual machine's page tables in the hypervisor, and intercepts VA to PA mapping changes to keep both copies consistent. To remove the complexity of shadow page tables, a hardware approach called *nested page table* (NPT) explicitly supports two classes of page tables (VA⇒PA and PA⇒MA) and can walk such tables purely in hardware.

Consider the following sequence of operations: (1) Create process; (2) TLB miss; (3) page fault; (4) context switch;

5.29.1 [10] <§§5.6, 5.7> What would happen for the given operation sequence for shadow page table and nested page table, respectively?

5.29.2 [10] <§§5.6, 5.7> Assuming an x86-based four-level page table in both guest and nested page table, how many memory references are needed to service a TLB miss for native vs. nested page table?

5.29.3 [15] <§§5.6, 5.7> Among TLB miss rate, TLB miss latency, page fault rate, and page fault handler latency, which metrics are more important for shadow page table? Which are important for nested page table?

Assume the following parameters for a shadow paging system.

TLB Misses per 1000 Instructions	NPT TLB Miss Latency	Page Faults per 1000 Instructions	Shadowing Page Fault Overhead
0.2	200 cycles	0.001	30,000 cycles

5.29.4 [10] <§5.6> For a benchmark with native execution CPI of 1, what are the CPI numbers if using shadow page tables vs. NPT (assuming only page table virtualization overhead)?

5.29.5 [10] <§5.6> What techniques can be used to reduce page table shadowing induced overhead?

5.29.6 [10] <§5.6> What techniques can be used to reduce NPT induced overhead?

Answers to Check Yourself

§5.1, page 369: 1 and 4. (3 is false because the cost of the memory hierarchy varies per computer, but in 2016 the highest cost is usually the DRAM.)

§5.3, page 390: 1 and 4: A lower miss penalty can enable smaller blocks, since you don't have that much latency to amortize, yet higher memory bandwidth usually leads to larger blocks, since the miss penalty is only slightly larger.

§5.4, page 409: 1.

§5.8, page 449: 2. (Both large block sizes and prefetching may reduce compulsory misses, so 1 is false.)

CHAPTER 4

Cache Memory

4.1 Computer Memory System Overview
- Characteristics of Memory Systems
- The Memory Hierarchy

4.2 Cache Memory Principles

4.3 Elements of Cache Design
- Cache Addresses
- Cache Size
- Mapping Function
- Replacement Algorithms
- Write Policy
- Line Size
- Number of Caches

4.4 Pentium 4 Cache Organization

4.5 Key Terms, Review Questions, and Problems

Appendix 4A Performance Characteristics of Two-Level Memories
- Locality
- Operation of Two-Level Memory
- Performance

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Present an overview of the main characteristics of computer memory systems and the use of a memory hierarchy.
- Describe the basic concepts and intent of cache memory.
- Discuss the key elements of cache design.
- Distinguish among direct mapping, associative mapping, and set-associative mapping.
- Explain the reasons for using multiple levels of cache.
- Understand the performance implications of multiple levels of memory.

Although seemingly simple in concept, computer memory exhibits perhaps the widest range of type, technology, organization, performance, and cost of any feature of a computer system. No single technology is optimal in satisfying the memory requirements for a computer system. As a consequence, the typical computer system is equipped with a hierarchy of memory subsystems, some internal to the system (directly accessible by the processor) and some external (accessible by the processor via an I/O module).

This chapter and the next focus on internal memory elements, while Chapter 6 is devoted to external memory. To begin, the first section examines key characteristics of computer memories. The remainder of the chapter examines an essential element of all modern computer systems: cache memory.

4.1 COMPUTER MEMORY SYSTEM OVERVIEW

Characteristics of Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.

The term **location** in Table 4.1 refers to whether memory is internal or external to the computer. Internal memory is often equated with main memory, but there are other forms of internal memory. The processor requires its own local memory, in the form of registers (e.g., see Figure 2.3). Further, as we will see, the control unit portion of the processor may also require its own internal memory. We will defer discussion of these latter two types of internal memory to later chapters. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

Table 4.1 Key Characteristics of Computer Memory Systems

Location	**Performance**
Internal (e.g., processor registers, cache, main memory)	Access time
External (e.g., optical disks, magnetic disks, tapes)	Cycle time
Capacity	Transfer rate
Number of words	**Physical Type**
Number of bytes	Semiconductor
Unit of Transfer	Magnetic
Word	Optical
Block	Magneto-optical
Access Method	**Physical Characteristics**
Sequential	Volatile/nonvolatile
Direct	Erasable/nonerasable
Random	**Organization**
Associative	Memory modules

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

- **Word:** The "natural" unit of organization of memory. The size of a word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation. The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.
- **Addressable units:** In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.
- **Unit of transfer:** For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an addressable unit. For external memory, data are often transferred in much larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

- **Sequential access:** Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units, discussed in Chapter 6, are sequential access.
- **Direct access:** As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique

address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units, discussed in Chapter 6, are direct access.

- **Random access:** Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- **Associative:** This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

- **Access time (latency):** For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
- **Memory cycle time:** This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.
- **Transfer rate:** This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time). For non-random-access memory, the following relationship holds:

$$T_n = T_A + \frac{n}{R} \tag{4.1}$$

where

T_n = Average time to read or write n bits

T_A = Average access time

n = Number of bits

R = Transfer rate, in bits per second (bps)

A variety of **physical types** of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several **physical characteristics** of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory (memory on integrated circuits) may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as *read-only memory* (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

For random-access memory, the **organization** is a key design issue. In this context, *organization* refers to the physical arrangement of bits to form words. The obvious arrangement is not always used, as is explained in Chapter 5.

The Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit;
- Greater capacity, smaller cost per bit;
- Greater capacity, slower access time.

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit;

b. Increasing capacity;

c. Increasing access time;

d. Decreasing frequency of access of the memory by the processor.

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization

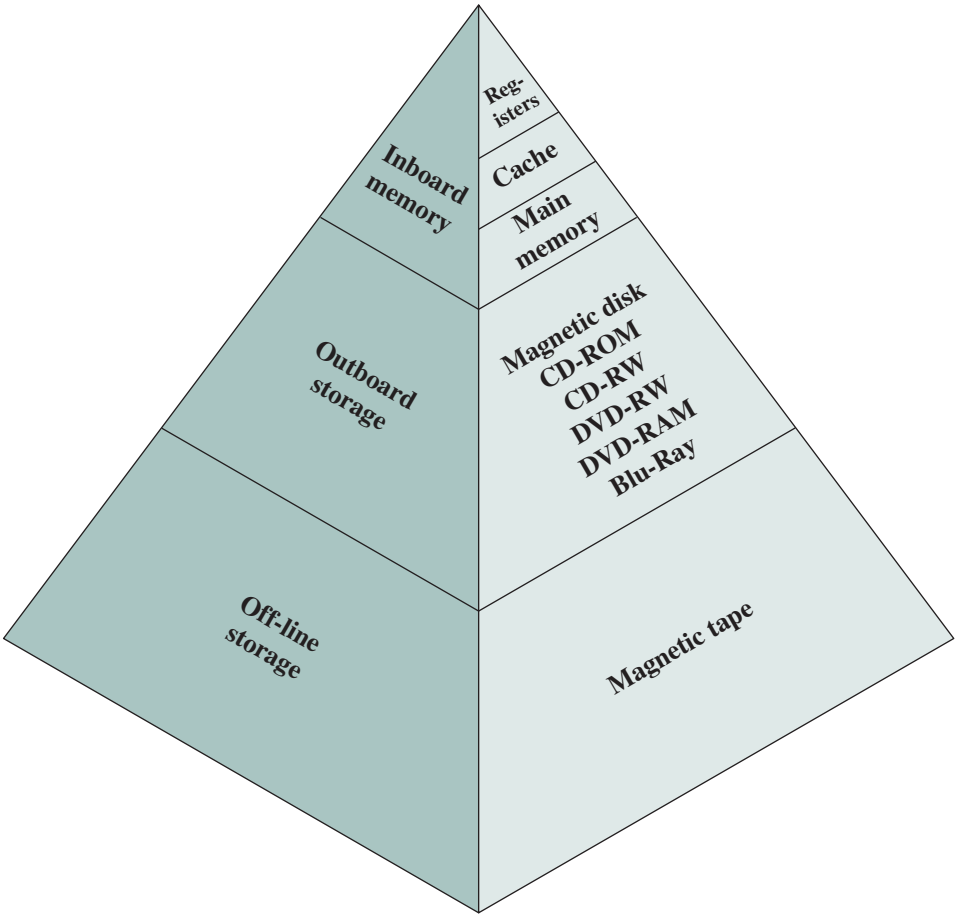


Figure 4.1 The Memory Hierarchy

is item (d): decreasing frequency of access. We examine this concept in greater detail when we discuss the cache, later in this chapter, and virtual memory in Chapter 8. A brief explanation is provided at this point.

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference** [DENN68]. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

EXAMPLE 4.1 Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of 0.01 μs; level 2 contains 100,000 words and has an access time of 0.1 μs. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Figure 4.2 shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2.[1] As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in level 1. Then the average time to access a word can be expressed as

$$(0.95)(0.01\ \mu s) + (0.05)(0.01\ \mu s + 0.1\ \mu s) = 0.0095 + 0.0055 = 0.015\ \mu s$$

The average access time is much closer to 0.01 μs than to 0.1 μs, as desired.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2

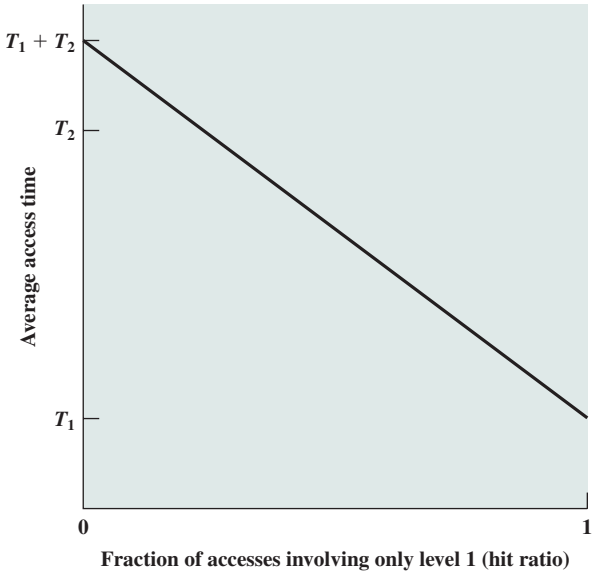


Figure 4.2 Performance of Accesses Involving only Level 1 (hit ratio)

[1]If the accessed word is found in the faster memory, that is defined as a **hit**. A **miss** occurs if the accessed word is not found in the faster memory.

memory contain all program instructions and data. The current clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 4.1. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory known as virtual memory, which is discussed in Chapter 8.

Other forms of memory may be included in the hierarchy. For example, large IBM mainframes include a form of internal memory known as expanded storage. This uses a semiconductor technology that is slower and less expensive than that of main memory. Strictly speaking, this memory does not fit into the hierarchy but is a side branch: Data can be moved between main memory and expanded storage but not between expanded storage and external memory. Other forms of secondary memory include optical and magneto-optical disks. Finally, additional levels can be effectively added to the hierarchy in software. A portion of main memory can be used as a buffer to hold data temporarily that is to be read out to disk. Such a technique, sometimes referred to as a disk cache,[2] improves performance in two ways:

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimizes processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

Appendix 4A examines the performance implications of multilevel memory structures.

[2] Disk cache is generally a purely software technique and is not examined in this book. See [STAL15] for a discussion.

4.2 CACHE MEMORY PRINCIPLES

Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory. The concept is illustrated in Figure 4.3a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

Figure 4.3b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

Figure 4.4 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = 2^n/K$ blocks in main memory. The cache consists of m blocks, called **lines**.[3] Each line contains K words,

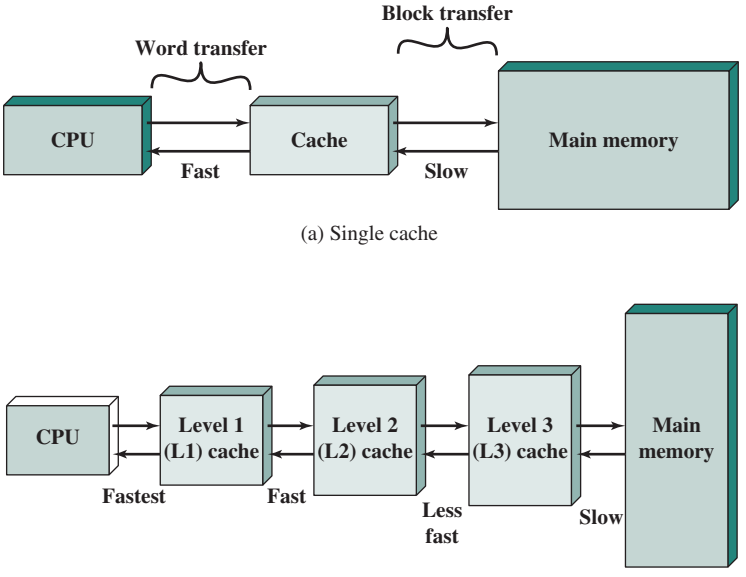


Figure 4.3 Cache and Main Memory

[3]In referring to the basic unit of the cache, the term *line* is used, rather than the term *block*, for two reasons: (1) to avoid confusion with a main memory block, which contains the same number of data words as a cache line; and (2) because a cache line includes not only K words of data, just as a main memory block, but also includes tag and control bits.

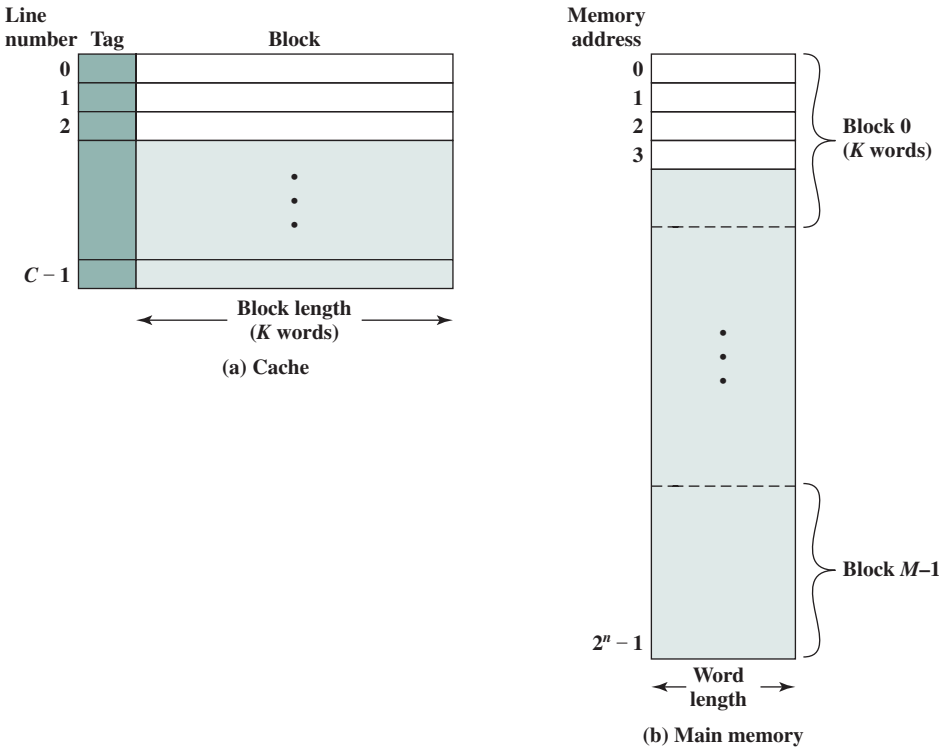


Figure 4.4 Cache/Main Memory Structure

plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each "word" being a single byte; in this case the line size is 4 bytes. The number of lines is considerably less than the number of main memory blocks ($m \ll M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

Figure 4.5 illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor. Figure 4.5 shows these last two operations occurring in parallel and reflects the organization shown in Figure 4.6, which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from

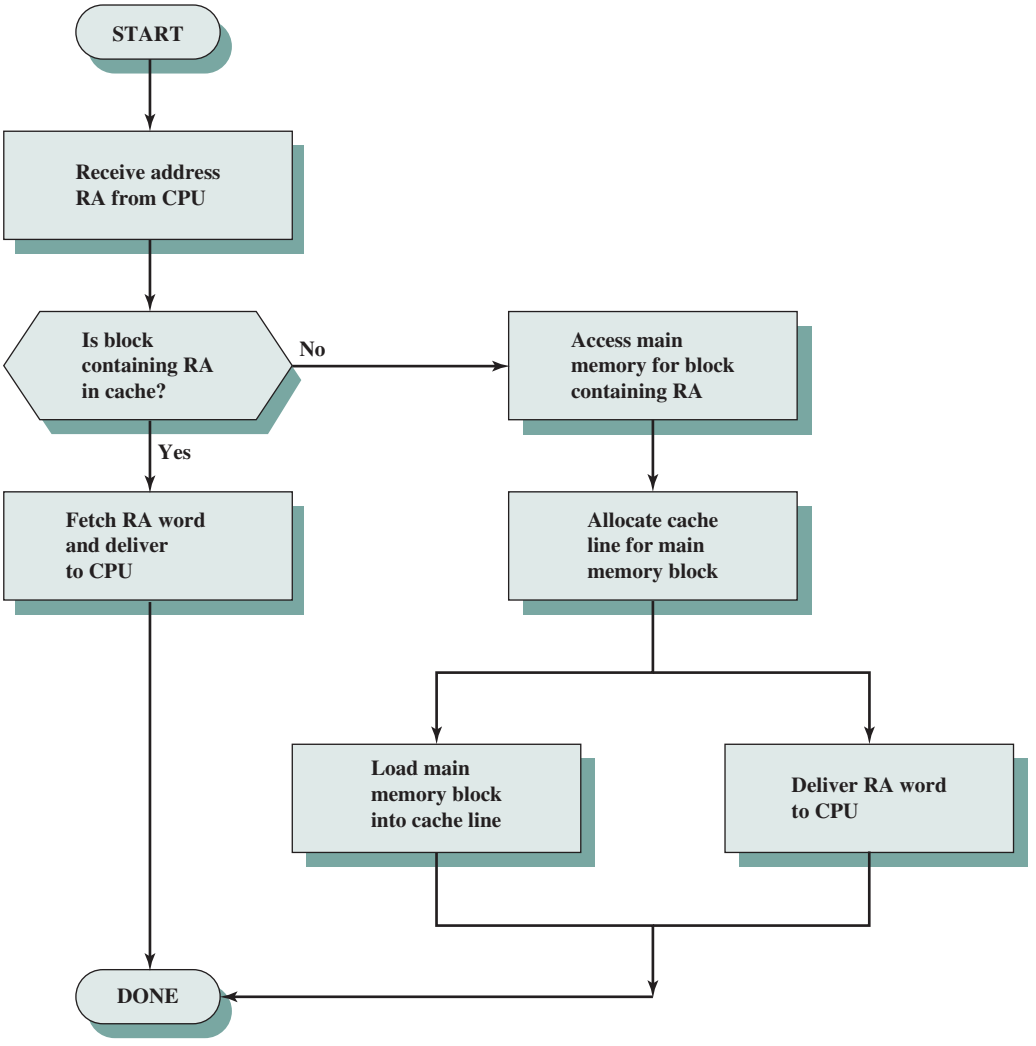


Figure 4.5 Cache Read Operation

which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor. In other organizations, the cache is physically interposed between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

A discussion of the performance parameters related to cache use is contained in Appendix 4A.

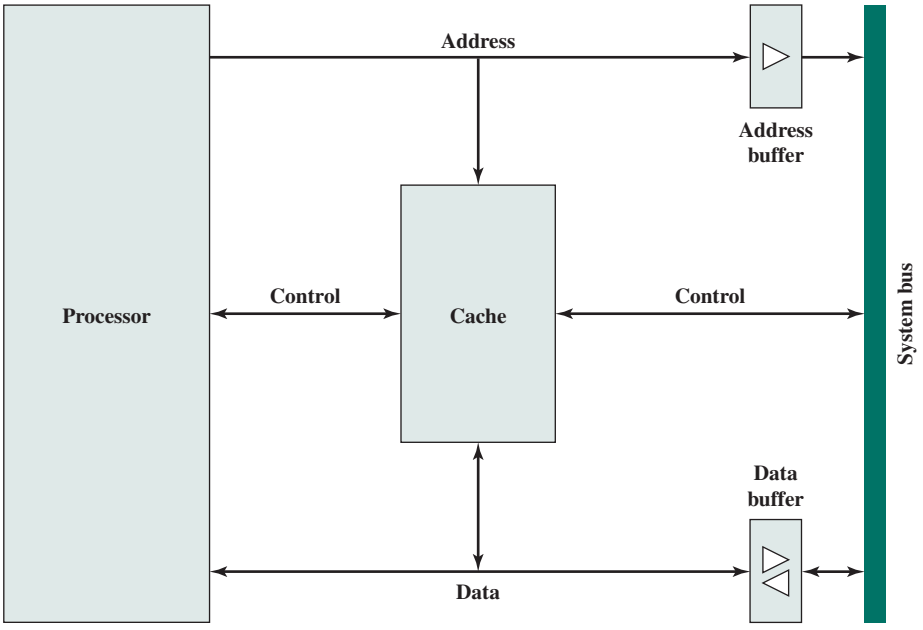


Figure 4.6 Typical Cache Organization

4.3 ELEMENTS OF CACHE DESIGN

This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in **high-performance computing (HPC)**. HPC deals with supercomputers and their software, especially for scientific applications that involve large amounts of data, vector and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches [BAIL93]. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache [WANG99, PRES01].[4]

Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. Table 4.2 lists key elements.

Cache Addresses

Almost all nonembedded processors, and many embedded processors, support virtual memory, a concept discussed in Chapter 8. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads

[4]For a general discussion of HPC, see [DOWD98].

Table 4.2 Elements of Cache Design

Cache Addresses	**Write Policy**
Logical	Write through
Physical	Write back
Cache Size	**Line Size**
Mapping Function	**Number of Caches**
Direct	Single or two level
Associative	Unified or split
Set associative	
Replacement Algorithm	
Least recently used (LRU)	
First in first out (FIFO)	
Least frequently used (LFU)	
Random	

to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory (Figure 4.7). A **logical cache**, also known as a **virtual cache**, stores data using

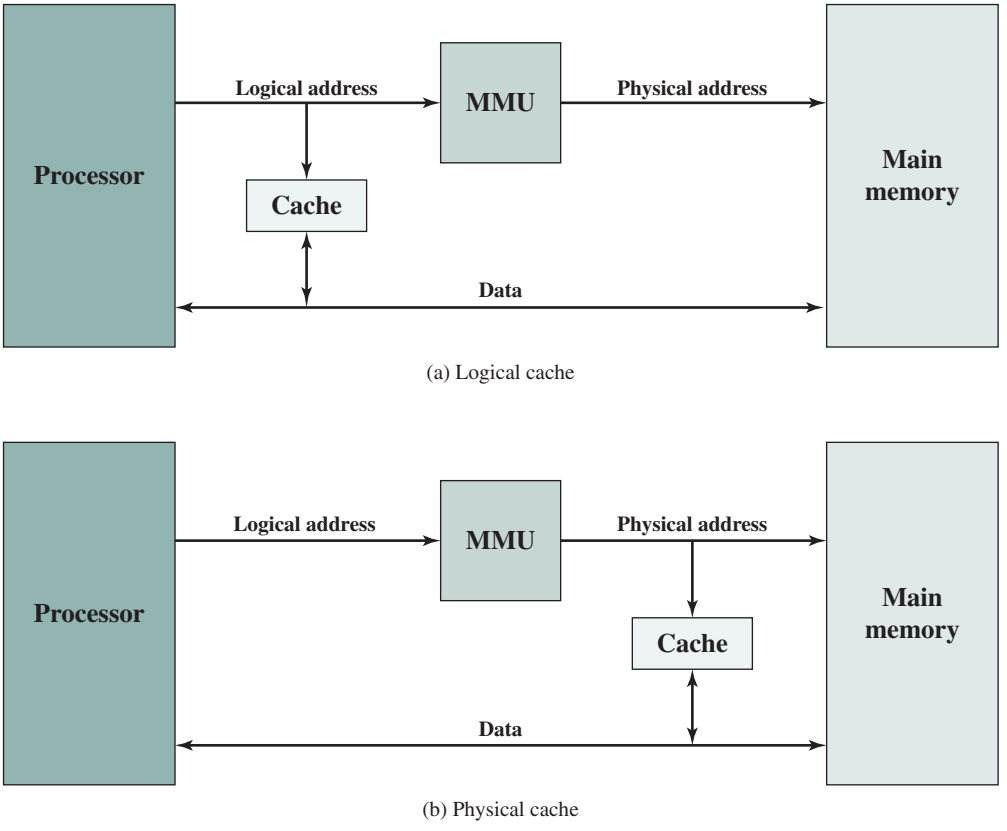


Figure 4.7 Logical and Physical Caches

virtual addresses. The processor accesses the cache directly, without going through the MMU. A **physical cache** stores data using main memory **physical addresses**.

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

The subject of logical versus physical cache is a complex one, and beyond the scope of this book. For a more in-depth discussion, see [CEKL97] and [JACO08].

Cache Size

The second item in Table 4.2, cache size, has already been discussed. We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single "optimum" cache size. Table 4.3 lists the cache sizes of some current and past processors.

Mapping Function

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set-associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example.

EXAMPLE 4.2 For all three cases, the example includes the following elements:

- The cache can hold 64 kB.
- Data are transferred between main memory and the cache in blocks of 4 bytes each. This means that the cache is organized as $16K = 2^{14}$ lines of 4 bytes each.
- The main memory consists of 16 MB, with each byte directly addressable by a 24-bit address ($2^{24} = 16M$). Thus, for mapping purposes, we can consider main memory to consist of 4M blocks of 4 bytes each.

Table 4.3 Cache Sizes of Some Processors

Processor	Type	Year of Introduction	L1 Cache[a]	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16–32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128–256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256–512 kB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 kB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTA[b]	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6 × 32 kB/ 32 kB	1.5 MB	12 MB
IBM zEnterprise 196	Mainframe/ server	2011	24 × 64 kB/ 128 kB	24 × 1.5 MB	24 MB L3 192 MB L4

Notes: [a] Two values separated by a slash refer to instruction and data caches. [b] Both caches are instruction only; no data caches.

DIRECT MAPPING The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

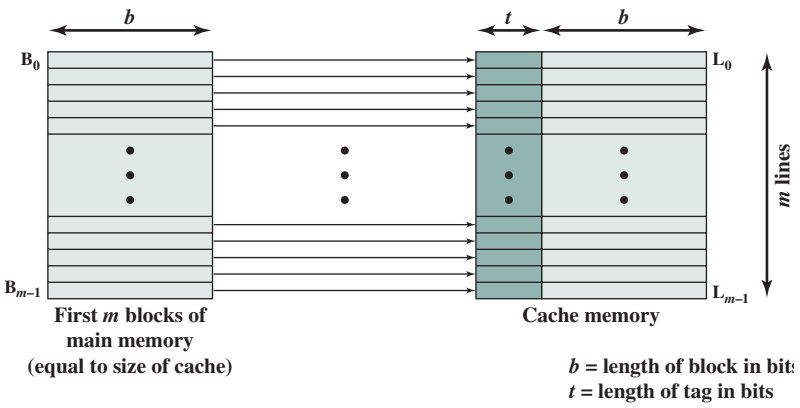
where

i = cache line number

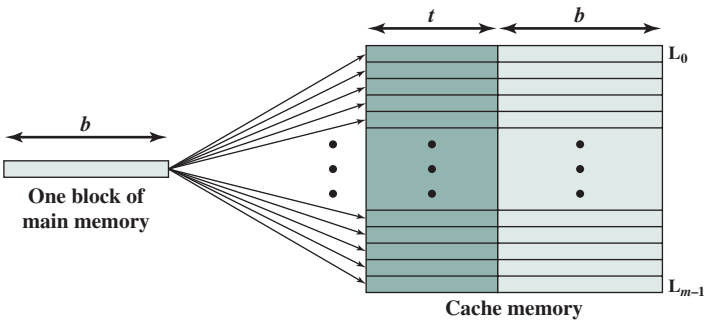
j = main memory block number

m = number of lines in the cache

Figure 4.8a shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks



(a) Direct mapping



(b) Associative mapping

Figure 4.8 Mapping from Main Memory to Cache: Direct and Associative

of main memory map into the cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1, and so on.

The mapping function is easily implemented using the main memory address. Figure 4.9 illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of $s - r$ bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \dfrac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache $= m = 2^r$
- Size of cache $= 2^{r+w}$ words or bytes
- Size of tag $= (s - r)$ bits

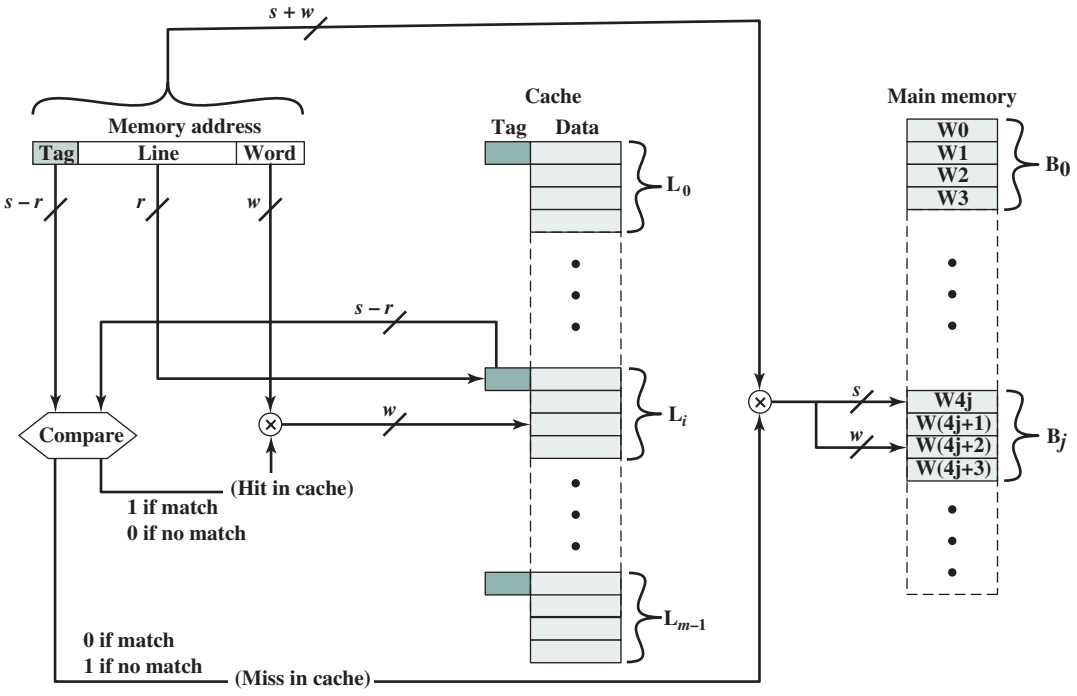


Figure 4.9 Direct-Mapping Cache Organization

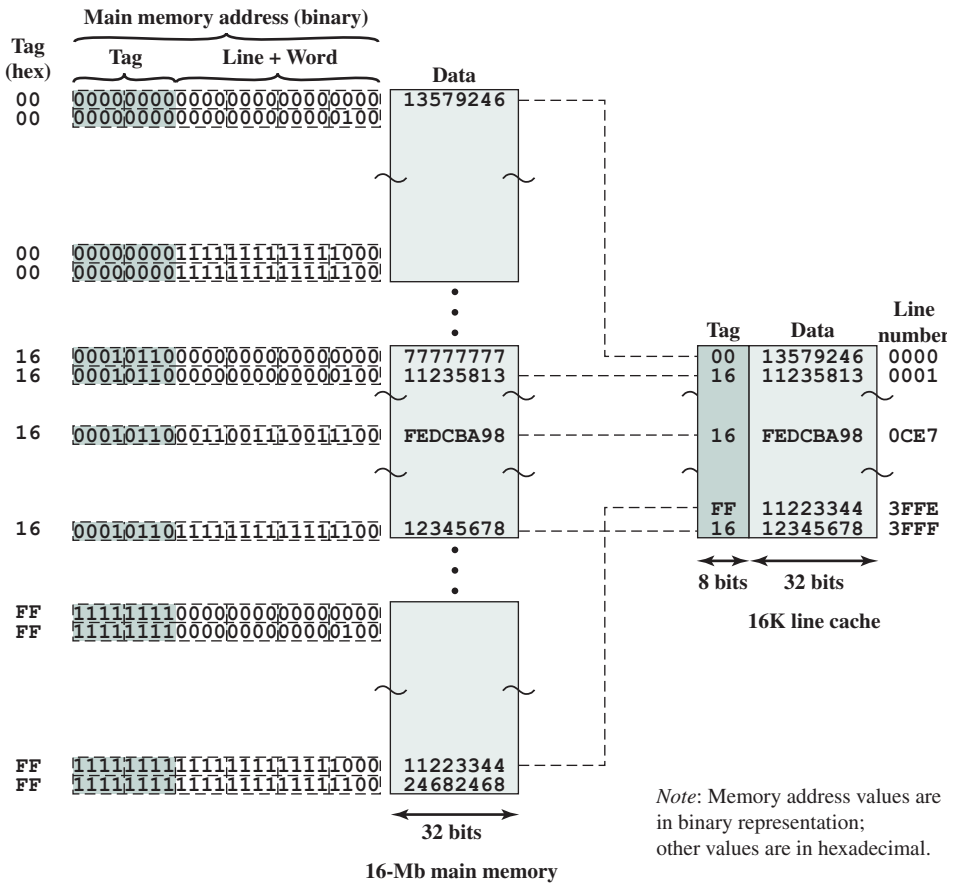
EXAMPLE 4.2a Figure 4.10 shows our example system using direct mapping.[5] In the example, $m = 16K = 2^{14}$ and $i = j$ modulo 2^{14}. The mapping becomes

Cache Line	Starting Memory Address of Block
0	000000, 010000, …, FF0000
1	000004, 010004, …, FF0004
⋮	⋮
$2^{14} - 1$	00FFFC, 01FFFC, …, FFFFFC

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, …, FF0000 have tag numbers 00, 01, …, FF, respectively.

Referring back to Figure 4.5, a read operation works as follows. The cache system is presented with a 24-bit address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a block from main memory. The actual address that is used for the fetch is the 22-bit tag-plus-line concatenated with two 0 bits, so that 4 bytes are fetched starting on a block boundary.

[5]In this and subsequent figures, memory values are represented in hexadecimal notation. See Chapter 9 for a basic refresher on number systems (decimal, binary, hexadecimal).



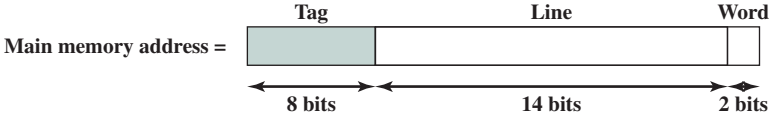


Figure 4.10 Direct Mapping Example

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m + 1, 2m + 1, \ldots, 2^s - m + 1$
$\vdots$	$\vdots$
$m - 1$	$m - 1, 2m - 1, 3m - 1, \ldots, 2^s - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually

read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as *thrashing*).

Selective Victim Cache Simulator

One approach to lower the **miss** penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a victim cache. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory. This concept is explored in Appendix F.

ASSOCIATIVE MAPPING Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 4.8b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match. Figure 4.11 illustrates the logic.

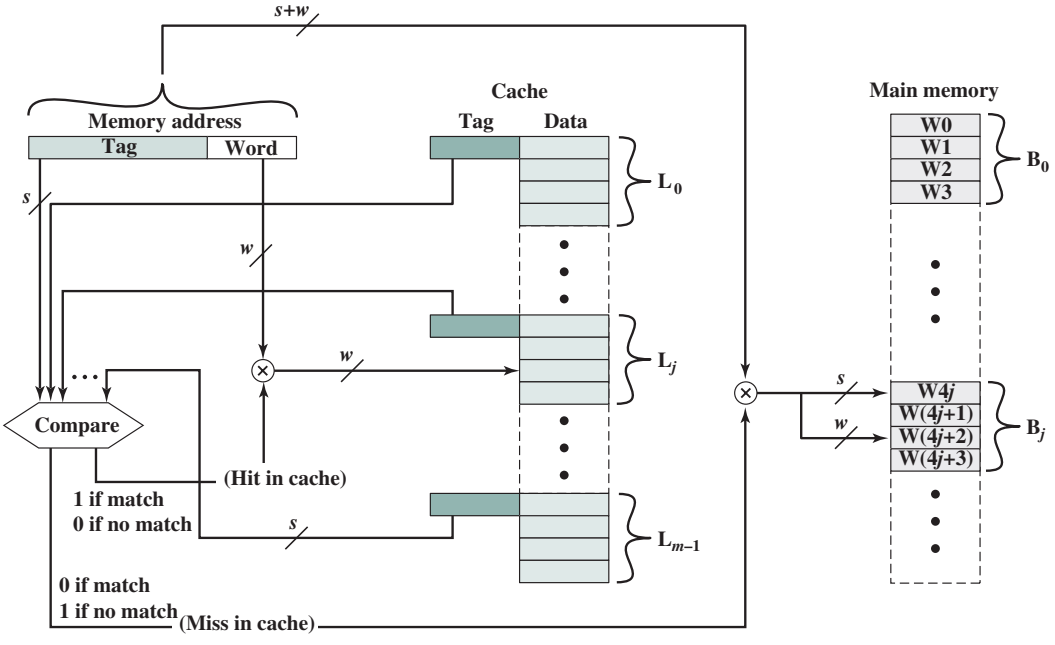


Figure 4.11 Fully Associative Cache Organization

EXAMPLE 4.2b Figure 4.12 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7. This is easily seen in binary notation:

Memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
Tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

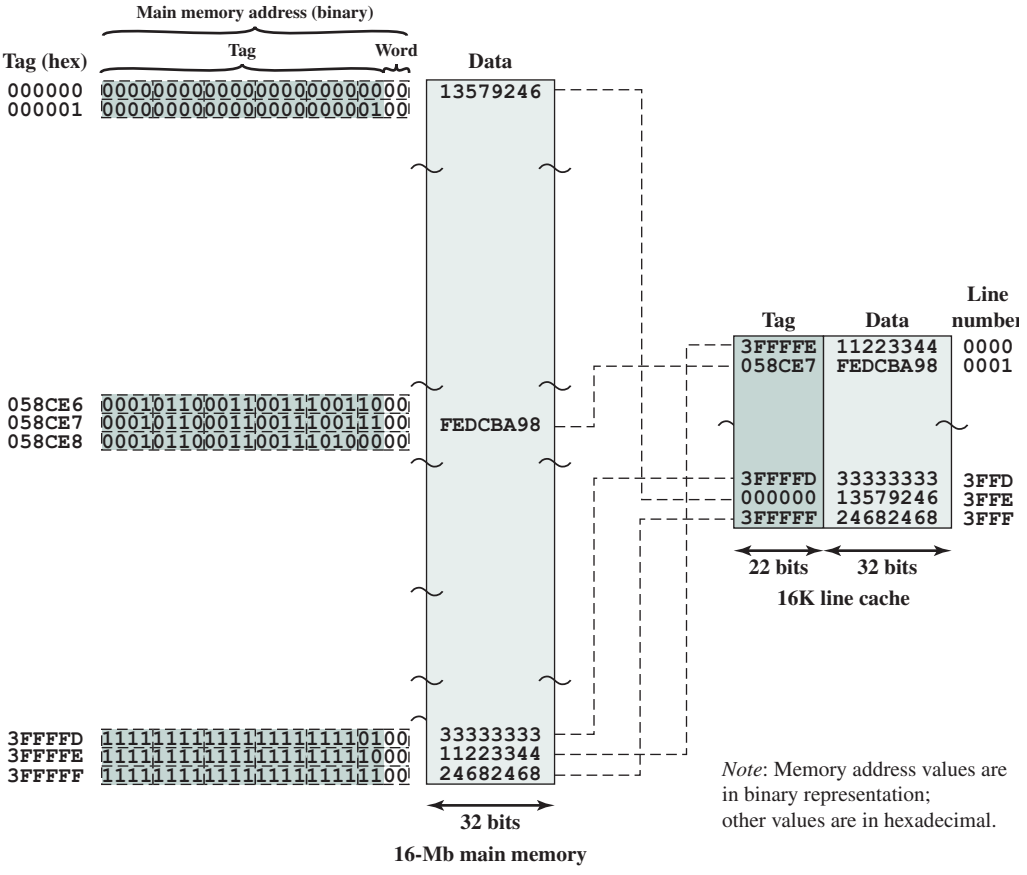


Figure 4.12 Associative Mapping Example

Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size = line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \dfrac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache = undetermined
- Size of tag $= s$ bits

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel.

Cache Time Analysis Simulator

SET-ASSOCIATIVE MAPPING Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

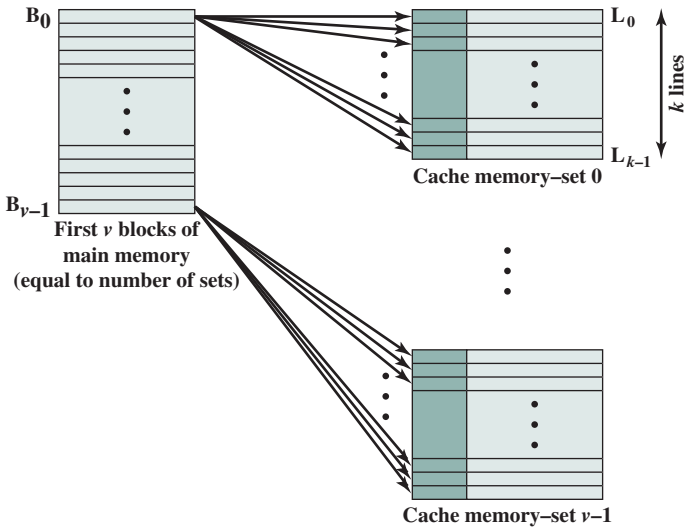
In this case, the cache consists of number sets, each of which consists of a number of lines. The relationships are

$$m = v \times k$$
$$i = j \text{ modulo } v$$

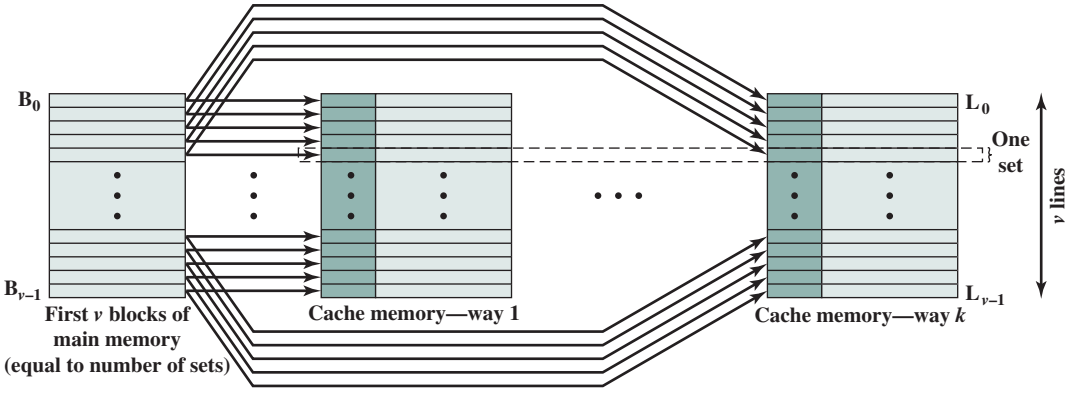
where

i = cache set number
j = main memory block number
m = number of lines in the cache
v = number of sets
k = number of lines in each set

This is referred to as k-way set-associative mapping. With set-associative mapping, block B_j can be mapped into any of the lines of set j. Figure 4.13a illustrates this mapping for the first v blocks of main memory. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as v associative caches. It is also possible to implement the set-associative cache as k direct mapping caches, as shown in Figure 4.13b. Each direct-mapped cache is referred to as a *way,* consisting of v lines. The first v lines of main memory are direct mapped into the v lines of each way; the next group of v lines of main memory are similarly mapped, and so on. The direct-mapped implementation is typically used



(a) v associative–mapped caches



(b) k direct–mapped caches

Figure 4.13 Mapping from Main Memory to Cache: k-Way Set Associative

for small degrees of associativity (small values of k) while the associative-mapped implementation is typically used for higher degrees of associativity [JACO08].

For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of $v = 2^d$ sets. The s bits of the Tag and Set fields specify one of the 2^s blocks of main memory. Figure 4.14 illustrates the cache control logic. With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes

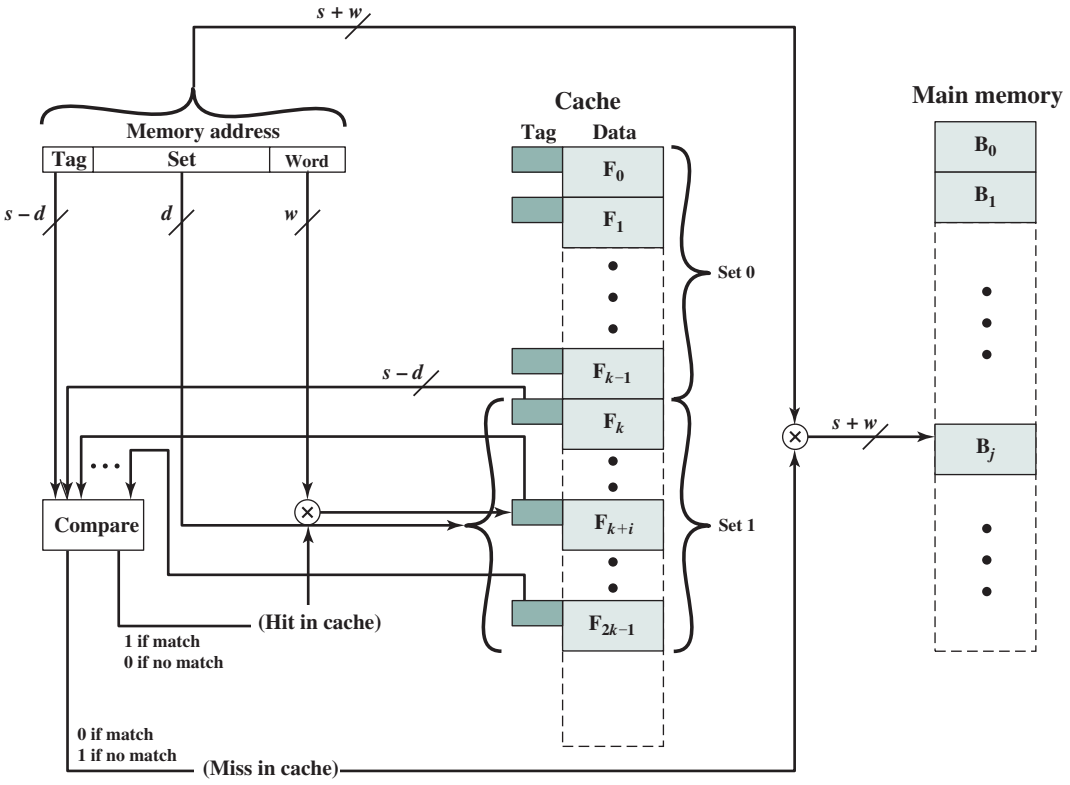


Figure 4.14 *k*-Way Set-Associative Cache Organization

- Block size = line size = 2^w words or bytes
- Number of blocks in main memory $= \dfrac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set $= k$
- Number of sets $= v = 2^d$
- Number of lines in cache $= m = kv = k \times 2^d$
- Size of cache $= k \times 2^{d+w}$ words or bytes
- Size of tag $= (s - d)$ bits

EXAMPLE 4.2c Figure 4.15 shows our example using set-associative mapping with two lines in each set, referred to as two-way set-associative. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^{13}. This determines the mapping of blocks into lines. Thus, blocks 000000, 008000, …, FF8000 of main memory map into cache set 0. Any of those blocks can be loaded into either of the two lines in the set. Note that no two blocks that map into the same cache set have the same tag number. For a read operation, the 13-bit set number is used to determine which set of two lines is to be examined. Both lines in the set are examined for a match with the tag number of the address to be accessed.

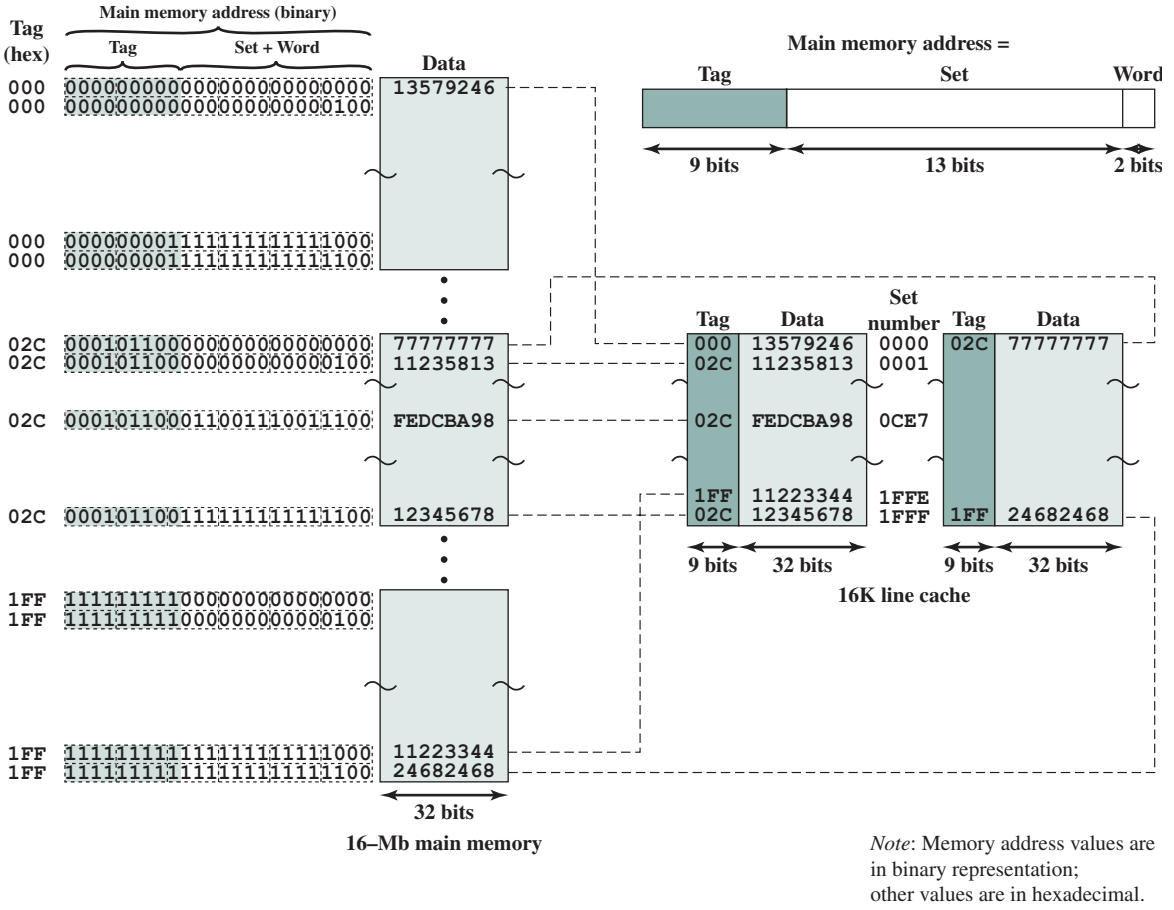


Figure 4.15 Two-Way Set-Associative Mapping Example

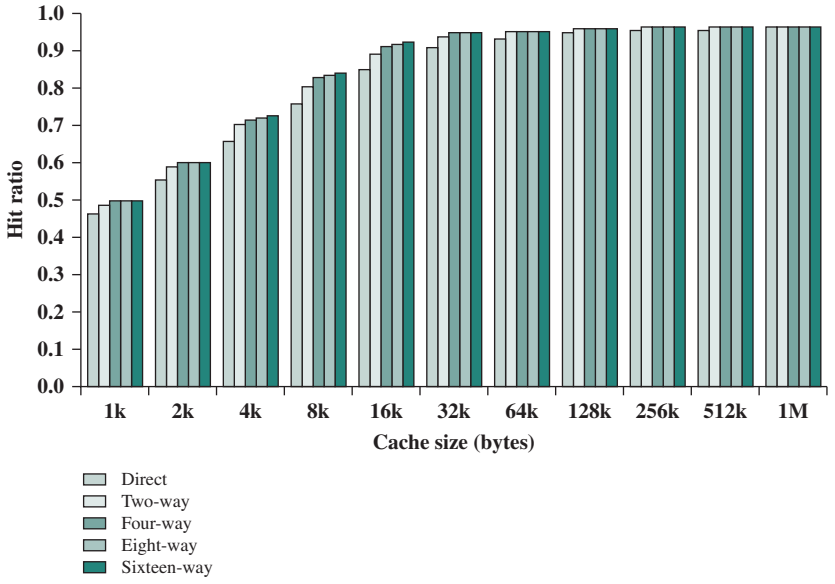


Figure 4.16 Varying Associativity over Cache Size

In the extreme case of $v = m$, $k = 1$, the set-associative technique reduces to direct mapping, and for $v = 1$, $k = m$, it reduces to associative mapping. The use of two lines per set ($v = m/2$, $k = 2$) is the most common set-associative organization. It significantly improves the hit ratio over direct mapping. Four-way set associative ($v = m/4$, $k = 4$) makes a modest additional improvement for a relatively small additional cost [MAYB84, HILL89]. Further increases in the number of lines per set have little effect.

Figure 4.16 shows the results of one simulation study of set-associative cache performance as a function of cache size [GENU04]. The difference in performance between direct and two-way set associative is significant up to at least a cache size of 64 kB. Note also that the difference between two-way and four-way at 4 kB is much less than the difference in going from for 4 kB to 8 kB in cache size. The complexity of the cache increases in proportion to the associativity, and in this case would not be justifiable against increasing cache size to 8 or even 16 kB. A final point to note is that beyond about 32 kB, increase in cache size brings no significant increase in performance.

The results of Figure 4.16 are based on simulating the execution of a GCC compiler. Different applications may yield different results. For example, [CANT01] reports on the results for cache performance using many of the CPU2000 SPEC benchmarks. The results of [CANT01] in comparing hit ratio to cache size follow the same pattern as Figure 4.16, but the specific values are somewhat different.

Cache Simulator
Multitask Cache Simulator

Replacement Algorithms

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is **least recently used (LRU)**: Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0. When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage [SMIT82].

Write Policy

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage

of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15% [SMIT82]. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

EXAMPLE 4.3 Consider a cache with a line size of 32 bytes and a main memory that requires 30 ns to transfer a 4-byte word. For any line that is written at least once before being swapped out of the cache, what is the average number of times that the line must be written before being swapped out for a write-back cache to be more efficient that a write-through cache?

For the write-back case, each dirty line is written back once, at swap-out time, taking $8 \times 30 = 240$ ns. For the write-through case, each update of the line requires that one word be written out to main memory, taking 30 ns. Therefore, if the average line that gets written at least once gets written more than 8 times before swap out, then write back is more efficient.

In a bus organization in which more than one device (typically a processor) has a cache and main memory is shared, a new problem is introduced. If data in one cache are altered, this invalidates not only the corresponding word in main memory, but also that same word in other caches (if any other cache happens to have that same word). Even if a write-through policy is used, the other caches may contain invalid data. A system that prevents this problem is said to maintain cache coherency. Possible approaches to cache coherency include the following:

- **Bus watching with write through:** Each cache controller monitors the address lines to detect write operations to memory by other bus masters. If another master writes to a location in shared memory that also resides in the cache memory, the cache controller invalidates that cache entry. This strategy depends on the use of a write-through policy by all cache controllers.
- **Hardware transparency:** Additional hardware is used to ensure that all updates to main memory via cache are reflected in all caches. Thus, if one processor modifies a word in its cache, this update is written to main memory. In addition, any matching words in other caches are similarly updated.
- **Noncacheable memory:** Only a portion of main memory is shared by more than one processor, and this is designated as noncacheable. In such a system, all accesses to shared memory are cache misses, because the shared memory is never copied into the cache. The noncacheable memory can be identified using chip-select logic or high-address bits.

Cache coherency is an active field of research. This topic is explored further in Part Five.

Line Size

Another design element is the line size. When a block of data is retrieved and placed in the cache, not only the desired word but also some number of adjacent words are retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality, which states that data in the vicinity of a referenced word are likely to be referenced in the near future. As the block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced. Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache. Because each block fetch overwrites older cache contents, a small number of blocks results in data being overwritten shortly after they are fetched.
- As a block becomes larger, each additional word is farther from the requested word and therefore less likely to be needed in the near future.

The relationship between block size and hit ratio is complex, depending on the locality characteristics of a particular program, and no definitive optimum value has been found. A size of from 8 to 64 bytes seems reasonably close to optimum [SMIT87, PRZY88, PRZY90, HAND98]. For HPC systems, 64- and 128-byte cache line sizes are most frequently used.

Number of Caches

When caches were originally introduced, the typical system had a single cache. More recently, the use of multiple caches has become the norm. Two aspects of this design issue concern the number of levels of caches and the use of unified versus split caches.

MULTILEVEL CACHES As logic density has increased, it has become possible to have a cache on the same chip as the processor: the on-chip cache. Compared with a cache reachable via an external bus, the on-chip cache reduces the processor's external bus activity and therefore speeds up execution times and increases overall system performance. When the requested instruction or data is found in the on-chip cache, the bus access is eliminated. Because of the short data paths internal to the processor, compared with bus lengths, on-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles. Furthermore, during this period the bus is free to support other transfers.

The inclusion of an on-chip cache leaves open the question of whether an off-chip, or external, cache is still desirable. Typically, the answer is yes, and most contemporary designs include both on-chip and external caches. The simplest such organization is known as a two-level cache, with the internal level 1 (L1) and the external cache designated as level 2 (L2). The reason for including an L2 cache is the following: If there is no L2 cache and the processor makes an access request for a memory location not in the L1 cache, then the processor must access DRAM or

ROM memory across the bus. Due to the typically slow bus speed and slow memory access time, this results in poor performance. On the other hand, if an L2 SRAM (static RAM) cache is used, then frequently the missing information can be quickly retrieved. If the SRAM is fast enough to match the bus speed, then the data can be accessed using a zero-wait state transaction, the fastest type of bus transfer.

Two features of contemporary cache design for multilevel caches are noteworthy. First, for an off-chip L2 cache, many designs do not use the system bus as the path for transfer between the L2 cache and the processor, but use a separate data path, so as to reduce the burden on the system bus. Second, with the continued shrinkage of processor components, a number of processors now incorporate the L2 cache on the processor chip, improving performance.

The potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches. Several studies have shown that, in general, the use of a second-level cache does improve performance (e.g., see [AZIM92], [NOVI93], [HAND98]). However, the use of multilevel caches does complicate all of the design issues related to caches, including size, replacement algorithm, and write policy; see [HAND98] and [PEIR99] for discussions.

Figure 4.17 shows the results of one simulation study of two-level cache performance as a function of cache size [GENU04]. The figure assumes that both caches have the same line size and shows the total hit ratio. That is, a hit is counted if the desired data appears in either the L1 or the L2 cache. The figure shows the impact of L2 on total hits with respect to L1 size. L2 has little effect on the total number of cache hits until it is at least double the L1 cache size. Note that the steepest part of the slope for an L1 cache of 8 kB is for an L2 cache of 16 kB. Again for an L1 cache of 16 kB, the steepest part of the curve is for an L2 cache size of 32 kB. Prior to that point, the L2 cache has little, if any, impact on total cache performance. The need for the L2 cache to be larger than

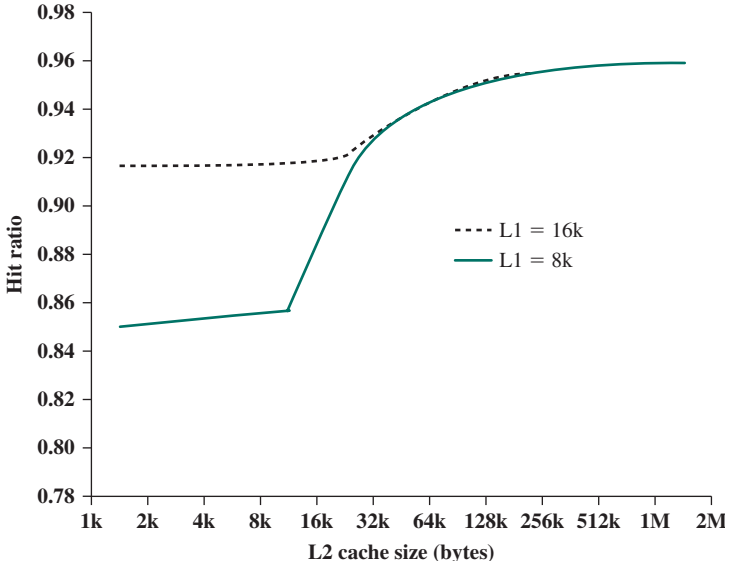


Figure 4.17 Total Hit Ratio (L1 and L2) for 8-kB and 16-kB L1

the L1 cache to affect performance makes sense. If the L2 cache has the same line size and capacity as the L1 cache, its contents will more or less mirror those of the L1 cache.

With the increasing availability of on-chip area available for cache, most contemporary microprocessors have moved the L2 cache onto the processor chip and added an L3 cache. Originally, the L3 cache was accessible over the external bus. More recently, most microprocessors have incorporated an on-chip L3 cache. In either case, there appears to be a performance advantage to adding the third level (e.g., see [GHAI98]). Further, large systems, such as the IBM mainframe zEnterprise systems, now incorporate 3 on-chip cache levels and a fourth level of cache shared across multiple chips [CURR11].

UNIFIED VERSUS SPLIT CACHES When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data. These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

There are two potential advantages of a unified cache:

- For a given cache size, a unified cache has a higher hit rate than split caches because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
- Only one cache needs to be designed and implemented.

The trend is toward split caches at the L1 and unified caches for higher levels, particularly for superscalar machines, which emphasize parallel instruction execution and the prefetching of predicted future instructions. The key advantage of the split cache design is that it eliminates contention for the cache between the instruction fetch/decode unit and the execution unit. This is important in any design that relies on the pipelining of instructions. Typically, the processor will fetch instructions ahead of time and fill a buffer, or pipeline, with instructions to be executed. Suppose now that we have a unified instruction/data cache. When the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the instruction prefetcher issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction. This cache contention can degrade performance by interfering with efficient use of the instruction pipeline. The split cache structure overcomes this difficulty.

4.4 PENTIUM 4 CACHE ORGANIZATION

The evolution of cache organization is seen clearly in the evolution of Intel microprocessors (Table 4.4). The 80386 does not include an on-chip cache. The 80486 includes a single on-chip cache of 8 kB, using a line size of 16 bytes and a four-way

Table 4.4 Intel Cache Evolution

Problem	Solution	Processor on Which Feature First Appears
External memory slower than the system bus.	Add external cache using faster memory technology.	386
Increased processor speed results in external bus becoming a bottleneck for cache access.	Move external cache on-chip, operating at the same speed as the processor.	486
Internal cache is rather small, due to limited space on chip.	Add external L2 cache using faster technology than main memory.	486
Contention occurs when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.	Create separate data and instruction caches.	Pentium
Increased processor speed results in external bus becoming a bottleneck for L2 cache access.	Create separate back-side bus that runs at higher speed than the main (front-side) external bus. The BSB is dedicated to the L2 cache.	Pentium Pro
	Move L2 cache on to the processor chip.	Pentium II
Some applications deal with massive databases and must have rapid access to large amounts of data. The on-chip caches are too small.	Add external L3 cache.	Pentium III
	Move L3 cache on-chip.	Pentium 4

set-associative organization. All of the Pentium processors include two on-chip L1 caches, one for data and one for instructions. For the Pentium 4, the L1 data cache is 16 kB, using a line size of 64 bytes and a four-way set-associative organization. The Pentium 4 instruction cache is described subsequently. The Pentium II also includes an L2 cache that feeds both of the L1 caches. The L2 cache is eight-way set associative with a size of 512 kB and a line size of 128 bytes. An L3 cache was added for the Pentium III and became on-chip with high-end versions of the Pentium 4.

Figure 4.18 provides a simplified view of the Pentium 4 organization, highlighting the placement of the three caches. The processor core consists of four major components:

- **Fetch/decode unit:** Fetches program instructions in order from the L2 cache, decodes these into a series of micro-operations, and stores the results in the L1 instruction cache.
- **Out-of-order execution logic:** Schedules execution of the micro-operations subject to data dependencies and resource availability; thus, micro-operations may be scheduled for execution in a different order than they were fetched from the instruction stream. As time permits, this unit schedules speculative execution of micro-operations that may be required in the future.

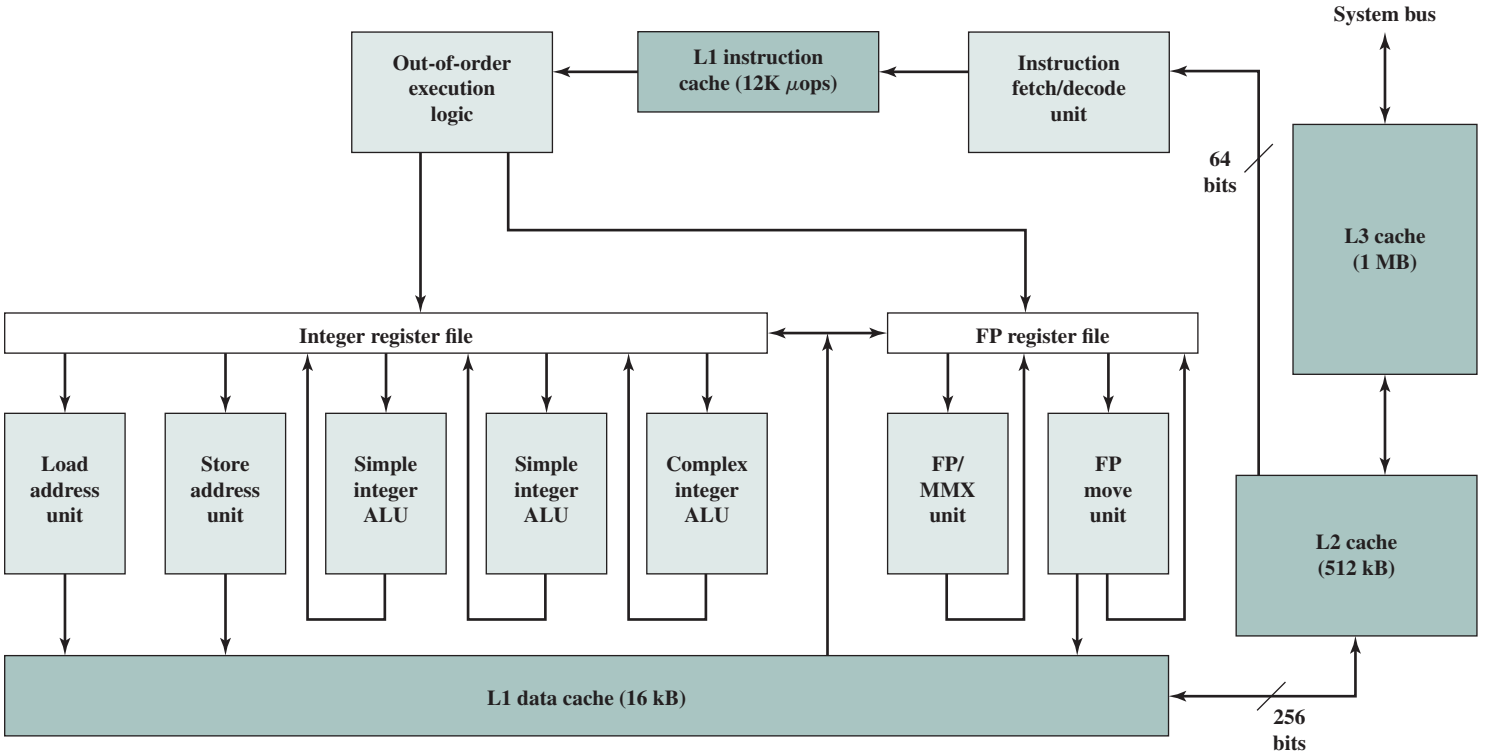


Figure 4.18 Pentium 4 Block Diagram

Table 4.5 Pentium 4 Cache Operating Modes

Control Bits		Operating Mode		
CD	**NW**	**Cache Fills**	**Write Throughs**	**Invalidates**
0	0	Enabled	Enabled	Enabled
1	0	Disabled	Enabled	Enabled
1	1	Disabled	Disabled	Disabled

Note: CD = 0; NW = 1 is an invalid combination.

- **Execution units:** These units execute micro-operations, fetching the required data from the L1 data cache and temporarily storing results in registers.
- **Memory subsystem:** This unit includes the L2 and L3 caches and the system bus, which is used to access main memory when the L1 and L2 caches have a cache miss and to access the system I/O resources.

Unlike the organization used in all previous Pentium models, and in most other processors, the Pentium 4 instruction cache sits between the instruction decode logic and the execution core. The reasoning behind this design decision is as follows: As discussed more fully in Chapter 16, the Pentium process decodes, or translates, Pentium machine instructions into simple RISC-like instructions called micro-operations. The use of simple, fixed-length micro-operations enables the use of superscalar pipelining and scheduling techniques that enhance performance. However, the Pentium machine instructions are cumbersome to decode; they have a variable number of bytes and many different options. It turns out that performance is enhanced if this decoding is done independently of the scheduling and pipelining logic. We return to this topic in Chapter 16.

The data cache employs a write-back policy: Data are written to main memory only when they are removed from the cache and there has been an update. The Pentium 4 processor can be dynamically configured to support write-through caching.

The L1 data cache is controlled by two bits in one of the control registers, labeled the CD (cache disable) and NW (not write-through) bits (Table 4.5). There are also two Pentium 4 instructions that can be used to control the data cache: INVD invalidates (flushes) the internal cache memory and signals the external cache (if any) to invalidate. WBINVD writes back and invalidates internal cache and then writes back and invalidates external cache.

Both the L2 and L3 caches are eight-way set-associative with a line size of 128 bytes.

4.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

access time	cache line	cache set
associative mapping	cache memory	data cache
cache hit	cache miss	direct access

direct mapping	logical cache	spatial locality
high-performance computing (HPC)	memory hierarchy	split cache
hit	miss	tag
hit ratio	multilevel cache	temporal locality
instruction cache	physical address	unified cache
L1 cache	physical cache	virtual address
L2 cache	random access	virtual cache
L3 cache	replacement algorithm	write back
line	secondary memory	write through
locality	sequential access	
	set-associative mapping	

Review Questions

4.1 What are the differences among sequential access, direct access, and random access?

4.2 What is the general relationship among access time, memory cost, and capacity?

4.3 How does the principle of locality relate to the use of multiple memory levels?

4.4 What are the differences among direct mapping, associative mapping, and set-associative mapping?

4.5 For a direct-mapped cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

4.6 For an associative cache, a main memory address is viewed as consisting of two fields. List and define the two fields.

4.7 For a set-associative cache, a main memory address is viewed as consisting of three fields. List and define the three fields.

4.8 What is the distinction between spatial locality and temporal locality?

4.9 In general, what are the strategies for exploiting spatial locality and temporal locality?

Problems

4.1 A set-associative cache consists of 64 lines, or slots, divided into four-line sets. Main memory contains 4K blocks of 128 words each. Show the format of main memory addresses.

4.2 A two-way set-associative cache has lines of 16 bytes and a total size of 8 kB. The 64-MB main memory is byte addressable. Show the format of main memory addresses.

4.3 For the hexadecimal main memory addresses 111111, 666666, BBBBBB, show the following information, in hexadecimal format:

- **a.** Tag, Line, and Word values for a direct-mapped cache, using the format of Figure 4.10
- **b.** Tag and Word values for an associative cache, using the format of Figure 4.12
- **c.** Tag, Set, and Word values for a two-way set-associative cache, using the format of Figure 4.15

4.4 List the following values:

- **a.** For the direct cache example of Figure 4.10: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag
- **b.** For the associative cache example of Figure 4.12: address length, number of addressable units, block size, number of blocks in main memory, number of lines in cache, size of tag

c. For the two-way set-associative cache example of Figure 4.15: address length, number of addressable units, block size, number of blocks in main memory, number of lines in set, number of sets, number of lines in cache, size of tag

4.5 Consider a 32-bit microprocessor that has an on-chip 16-kB four-way set-associative cache. Assume that the cache has a line size of four 32-bit words. Draw a block diagram of this cache showing its organization and how the different address fields are used to determine a cache hit/miss. Where in the cache is the word from memory location ABCDE8F8 mapped?

4.6 Given the following specifications for an external cache memory: four-way set associative; line size of two 16-bit words; able to accommodate a total of 4K 32-bit words from main memory; used with a 16-bit processor that issues 24-bit addresses. Design the cache structure with all pertinent information and show how it interprets the processor's addresses.

4.7 The Intel 80486 has an on-chip, unified cache. It contains 8 kB and has a four-way set-associative organization and a block length of four 32-bit words. The cache is organized into 128 sets. There is a single "line valid bit" and three bits, B0, B1, and B2 (the "LRU" bits), per line. On a cache miss, the 80486 reads a 16-byte line from main memory in a bus memory read burst. Draw a simplified diagram of the cache and show how the different fields of the address are interpreted.

4.8 Consider a machine with a byte addressable main memory of 2^{16} bytes and block size of 8 bytes. Assume that a direct mapped cache consisting of 32 lines is used with this machine.

a. How is a 16-bit memory address divided into tag, line number, and byte number?

b. Into what line would bytes with each of the following addresses be stored?

0001	0001	0001	1011
1100	0011	0011	0100
1101	0000	0001	1101
1010	1010	1010	1010

c. Suppose the byte with address 0001 1010 0001 1010 is stored in the cache. What are the addresses of the other bytes stored along with it?

d. How many total bytes of memory can be stored in the cache?

e. Why is the tag also stored in the cache?

4.9 For its on-chip cache, the Intel 80486 uses a replacement algorithm referred to as **pseudo least recently used**. Associated with each of the 128 sets of four lines (labeled L0, L1, L2, L3) are three bits B0, B1, and B2. The replacement algorithm works as follows: When a line must be replaced, the cache will first determine whether the most recent use was from L0 and L1 or L2 and L3. Then the cache will determine which of the pair of blocks was least recently used and mark it for replacement. Figure 4.19 illustrates the logic.

a. Specify how the bits B0, B1, and B2 are set and then describe in words how they are used in the replacement algorithm depicted in Figure 4.19.

b. Show that the 80486 algorithm approximates a true LRU algorithm. *Hint:* Consider the case in which the most recent order of usage is L0, L2, L3, L1.

c. Demonstrate that a true LRU algorithm would require 6 bits per set.

4.10 A set-associative cache has a block size of four 16-bit words and a set size of 2. The cache can accommodate a total of 4096 words. The main memory size that is cacheable is 64K $\times$ 32 bits. Design the cache structure and show how the processor's addresses are interpreted.

4.11 Consider a memory system that uses a 32-bit address to address at the byte level, plus a cache that uses a 64-byte line size.

a. Assume a direct mapped cache with a tag field in the address of 20 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.

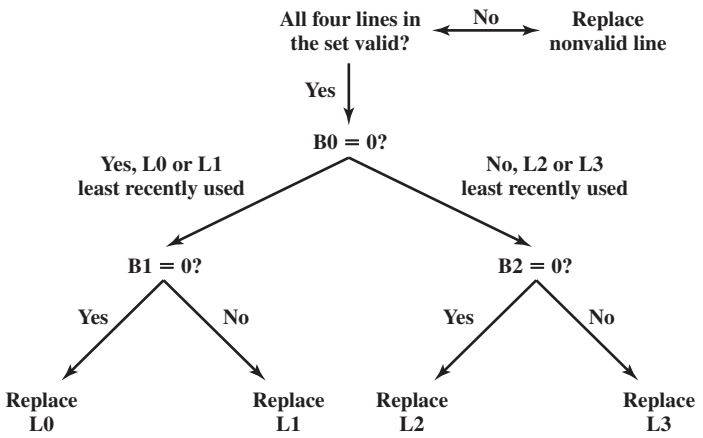


Figure 4.19 Intel 80486 On-Chip Cache Replacement Strategy

b. Assume an associative cache. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in cache, size of tag.

c. Assume a four-way set-associative cache with a tag field in the address of 9 bits. Show the address format and determine the following parameters: number of addressable units, number of blocks in main memory, number of lines in set, number of sets in cache, number of lines in cache, size of tag.

4.12 Consider a computer with the following characteristics: total of 1 MB of main memory; word size of 1 byte; block size of 16 bytes; and cache size of 64 kB.

a. For the main memory addresses of F0010, 01234, and CABBE, give the corresponding tag, cache line address, and word offsets for a direct-mapped cache.

b. Give any two main memory addresses with different tags that map to the same cache slot for a direct-mapped cache.

c. For the main memory addresses of F0010 and CABBE, give the corresponding tag and offset values for a fully-associative cache.

d. For the main memory addresses of F0010 and CABBE, give the corresponding tag, cache set, and offset values for a two-way set-associative cache.

4.13 Describe a simple technique for implementing an LRU replacement algorithm in a four-way set-associative cache.

4.14 Consider again Example 4.3. How does the answer change if the main memory uses a block transfer capability that has a first-word access time of 30 ns and an access time of 5 ns for each word thereafter?

4.15 Consider the following code:

```
for (i = 0; i < 20; i++)
  for (j = 0; j < 10; j++)
    a[i] = a[i]*j
```

a. Give one example of the spatial locality in the code.

b. Give one example of the temporal locality in the code.

4.16 Generalize Equations (4.2) and (4.3), in Appendix 4A, to *N*-level memory hierarchies.

4.17 A computer system contains a main memory of 32K 16-bit words. It also has a 4K word cache divided into four-line sets with 64 words per line. Assume that the cache is initially empty. The processor fetches words from locations 0, 1, 2, . . . , 4351 in that

order. It then repeats this fetch sequence nine more times. The cache is 10 times faster than main memory. Estimate the improvement resulting from the use of the cache. Assume an LRU policy for block replacement.

4.18 Consider a cache of 4 lines of 16 bytes each. Main memory is divided into blocks of 16 bytes each. That is, block 0 has bytes with addresses 0 through 15, and so on. Now consider a program that accesses memory in the following sequence of addresses:
Once: 63 through 70.
Loop ten times: 15 through 32; 80 through 95.

a. Suppose the cache is organized as direct mapped. Memory blocks 0, 4, and so on are assigned to line 1; blocks 1, 5, and so on to line 2; and so on. Compute the hit ratio.

b. Suppose the cache is organized as two-way set associative, with two sets of two lines each. Even-numbered blocks are assigned to set 0 and odd-numbered blocks are assigned to set 1. Compute the hit ratio for the two-way set-associative cache using the least recently used replacement scheme.

4.19 Consider a memory system with the following parameters:

$$T_c = 100 \text{ ns} \qquad C_c = 10^{-4} \text{ \$/bit}$$

$$T_m = 1200 \text{ ns} \qquad C_m = 10^{-5} \text{ \$/bit}$$

a. What is the cost of 1 MB of main memory?

b. What is the cost of 1 MB of main memory using cache memory technology?

c. If the effective access time is 10% greater than the cache access time, what is the hit ratio H?

4.20 **a.** Consider an L1 cache with an access time of 1 ns and a hit ratio of $H = 0.95$. Suppose that we can change the cache design (size of cache, cache organization) such that we increase H to 0.97, but increase access time to 1.5 ns. What conditions must be met for this change to result in improved performance?

d. Explain why this result makes intuitive sense.

4.21 Consider a single-level cache with an access time of 2.5 ns, a line size of 64 bytes, and a hit ratio of $H = 0.95$. Main memory uses a block transfer capability that has a first-word (4 bytes) access time of 50 ns and an access time of 5 ns for each word thereafter.

a. What is the access time when there is a cache miss? Assume that the cache waits until the line has been fetched from main memory and then re-executes for a hit.

b. Suppose that increasing the line size to 128 bytes increases the H to 0.97. Does this reduce the average memory access time?

4.22 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main memory hit ratio is 0.6. What is the average time in nanoseconds required to access a referenced word on this system?

4.23 Consider a cache with a line size of 64 bytes. Assume that on average 30% of the lines in the cache are dirty. A word consists of 8 bytes.

a. Assume there is a 3% miss rate (0.97 hit ratio). Compute the amount of main memory traffic, in terms of bytes per instruction for both write-through and write-back policies. Memory is read into cache one line at a time. However, for write back, a single word can be written from cache to main memory.

b. Repeat part a for a 5% rate.

c. Repeat part a for a 7% rate.

d. What conclusion can you draw from these results?

4.24 On the Motorola 68020 microprocessor, a cache access takes two clock cycles. Data access from main memory over the bus to the processor takes three clock cycles in the

case of no wait state insertion; the data are delivered to the processor in parallel with delivery to the cache.

a. Calculate the effective length of a memory cycle given a hit ratio of 0.9 and a clocking rate of 16.67 MHz.

b. Repeat the calculations assuming insertion of two wait states of one cycle each per memory cycle. What conclusion can you draw from the results?

4.25 Assume a processor having a memory cycle time of 300 ns and an instruction processing rate of 1 MIPS. On average, each instruction requires one bus memory cycle for instruction fetch and one for the operand it involves.

a. Calculate the utilization of the bus by the processor.

b. Suppose the processor is equipped with an instruction cache and the associated hit ratio is 0.5. Determine the impact on bus utilization.

4.26 The performance of a single-level cache system for a read operation can be characterized by the following equation:

$$T_a = T_c + (1 - H)T_m$$

where T_a is the average access time, T_c is the cache access time, T_m is the memory access time (memory to processor register), and H is the hit ratio. For simplicity, we assume that the word in question is loaded into the cache in parallel with the load to processor register. This is the same form as Equation (4.2).

a. Define T_b = time to transfer a line between cache and main memory, and W = fraction of write references. Revise the preceding equation to account for writes as well as reads, using a write-through policy.

b. Define W_b as the probability that a line in the cache has been altered. Provide an equation for T_a for the write-back policy.

4.27 For a system with two levels of cache, define T_{c_1} = first − level cache access time; T_{c_2} = second − level cache access time; T_m = memory access time; H_1 = first − level cache hit ratio; H_2 = combined first/second level cache hit ratio. Provide an equation for T_a for a read operation.

4.28 Assume the following performance characteristics on a cache read miss: one clock cycle to send an address to main memory and four clock cycles to access a 32-bit word from main memory and transfer it to the processor and cache.

a. If the cache line size is one word, what is the miss penalty (i.e., additional time required for a read in the event of a read miss)?

b. What is the miss penalty if the cache line size is four words and a multiple, nonburst transfer is executed?

c. What is the miss penalty if the cache line size is four words and a transfer is executed, with one clock cycle per word transfer?

4.29 For the cache design of the preceding problem, suppose that increasing the line size from one word to four words results in a decrease of the read miss rate from 3.2% to 1.1%. For both the nonburst transfer and the burst transfer case, what is the average miss penalty, averaged over all reads, for the two different line sizes?

APPENDIX 4A PERFORMANCE CHARACTERISTICS OF TWO-LEVEL MEMORIES

In this chapter, reference is made to a cache that acts as a buffer between main memory and processor, creating a two-level internal memory. This two-level architecture exploits a property known as locality to provide improved performance over a comparable one-level memory.

The main memory cache mechanism is part of the computer architecture, implemented in hardware and typically invisible to the operating system. There are two other instances of a two-level memory approach that also exploit locality and that are, at least partially, implemented in the operating system: virtual memory and the disk cache (Table 4.6). Virtual memory is explored in Chapter 8; disk cache is beyond the scope of this book but is examined in [STAL15]. In this appendix, we look at some of the performance characteristics of two-level memories that are common to all three approaches.

Locality

The basis for the performance advantage of a two-level memory is a principle known as **locality of reference** [DENN68]. This principle states that memory references tend to cluster. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Intuitively, the principle of locality makes sense. Consider the following line of reasoning:

1. Except for branch and call instructions, which constitute only a small fraction of all program instructions, program execution is sequential. Hence, in most cases, the next instruction to be fetched immediately follows the last instruction fetched.
2. It is rare to have a long uninterrupted sequence of procedure calls followed by the corresponding sequence of returns. Rather, a program remains confined to a rather narrow window of procedure-invocation depth. Thus, over a short period of time references to instructions tend to be localized to a few procedures.
3. Most iterative constructs consist of a relatively small number of instructions repeated many times. For the duration of the iteration, computation is therefore confined to a small contiguous portion of a program.
4. In many programs, much of the computation involves processing data structures, such as arrays or sequences of records. In many cases, successive references to these data structures will be to closely located data items.

Table 4.6 Characteristics of Two-Level Memories

	Main Memory Cache	Virtual Memory (paging)	Disk Cache
Typical access time ratios	5:1 (main memory vs. cache)	10^6:1 (main memory vs. disk)	10^6:1 (main memory vs. disk)
Memory management system	Implemented by special hardware	Combination of hardware and system software	System software
Typical block or page size	4 to 128 bytes (cache block)	64 to 4096 bytes (virtual memory page)	64 to 4096 bytes (disk block or pages)
Access of processor to second level	Direct access	Indirect access	Indirect access

Table 4.7 Relative Dynamic Frequency of High-Level Language Operations

Study	[HUCK83]	[KNUT71]	[PATT82a]		[TANE78]
Language	Pascal	FORTRAN	Pascal	C	SAL
Workload	Scientific	Student	System	System	System
Assign	74	67	45	38	42
Loop	4	3	5	3	4
Call	1	3	15	12	12
IF	20	11	29	43	36
GOTO	2	9	—	3	—
Other	—	7	6	1	6

This line of reasoning has been confirmed in many studies. With reference to point 1, a variety of studies have analyzed the behavior of high-level language programs. Table 4.7 includes key results, measuring the appearance of various statement types during execution, from the following studies. The earliest study of programming language behavior, performed by Knuth [KNUT71], examined a collection of FORTRAN programs used as student exercises. Tanenbaum [TANE78] published measurements collected from over 300 procedures used in operating-system programs and written in a language that supports structured programming (SAL). Patterson and Sequein [PATT82a] analyzed a set of measurements taken from compilers and programs for typesetting, computer-aided design (CAD), sorting, and file comparison. The programming languages C and Pascal were studied. Huck [HUCK83] analyzed four programs intended to represent a mix of general-purpose scientific computing, including fast Fourier transform and the integration of systems of differential equations. There is good agreement in the results of this mixture of languages and applications that branching and call instructions represent only a fraction of statements executed during the lifetime of a program. Thus, these studies confirm assertion 1.

With respect to assertion 2, studies reported in [PATT85a] provide confirmation. This is illustrated in Figure 4.20, which shows call-return behavior. Each call is represented by the line moving down and to the right, and each return by the line moving up and to the right. In the figure, a *window* with depth equal to 5 is defined. Only a sequence of calls and returns with a net movement of 6 in either direction causes the window to move. As can be seen, the executing program can remain within a stationary window for long periods of time. A study by the same analysts of C and Pascal programs showed that a window of depth 8 will need to shift only on less than 1% of the calls or returns [TAMI83].

A distinction is made in the literature between spatial locality and temporal locality. **Spatial locality** refers to the tendency of execution to involve a number of memory locations that are clustered. This reflects the tendency of a processor to access instructions sequentially. Spatial location also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data. **Temporal locality** refers to the tendency for a processor to access memory locations that have been used recently. For example, when an iteration loop is executed, the processor executes the same set of instructions repeatedly.

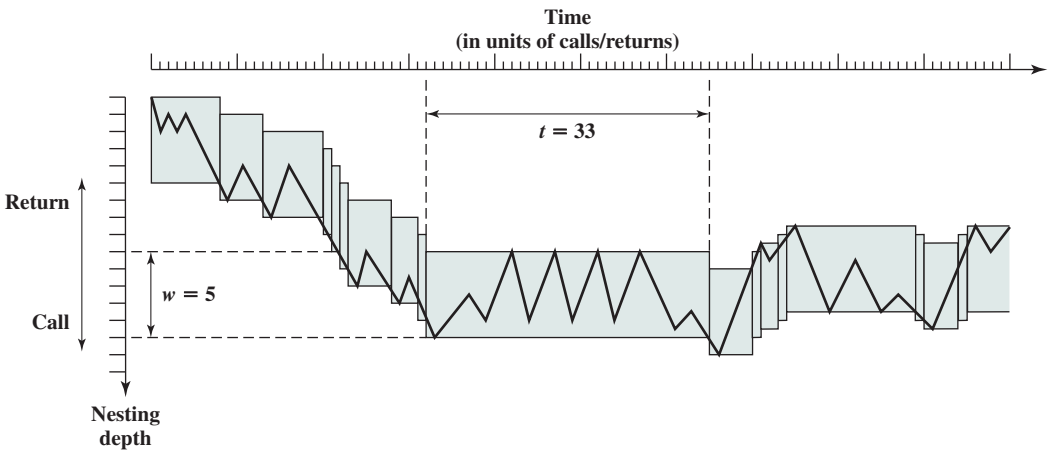


Figure 4.20 Example Call-Return Behavior of a Program

Traditionally, temporal locality is exploited by keeping recently used instruction and data values in cache memory and by exploiting a cache hierarchy. Spatial locality is generally exploited by using larger cache blocks and by incorporating prefetching mechanisms (fetching items of anticipated use) into the cache control logic. Recently, there has been considerable research on refining these techniques to achieve greater performance, but the basic strategies remain the same.

Operation of Two-Level Memory

The locality property can be exploited in the formation of a two-level memory. The upper-level memory (M1) is smaller, faster, and more expensive (per bit) than the lower-level memory (M2). M1 is used as a temporary store for part of the contents of the larger M2. When a memory reference is made, an attempt is made to access the item in M1. If this succeeds, then a quick access is made. If not, then a block of memory locations is copied from M2 to M1 and the access then takes place via M1. Because of locality, once a block is brought into M1, there should be a number of accesses to locations in that block, resulting in fast overall service.

To express the average time to access an item, we must consider not only the speeds of the two levels of memory, but also the probability that a given reference can be found in M1. We have

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{4.2}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.*g*., *cache*, *disk cache*)
T_2 = access time of M2 (e.*g*., *main memory*, *disk*)
H = hit ratio (fraction of time reference is found in M1)

Figure 4.2 shows average access time as a function of hit ratio. As can be seen, for a high percentage of hits, the average total access time is much closer to that of M1 than M2.

Performance

Let us look at some of the parameters relevant to an assessment of a two-level memory mechanism. First consider cost. We have

$$C_s = \frac{C_1S_1 + C_2S_2}{S_1 + S_2} \tag{4.3}$$

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

We would like $C_s \approx C_2$. Given that $C_1 \gg C_2$, this requires $S_1 < S_2$. Figure 4.21 shows the relationship.

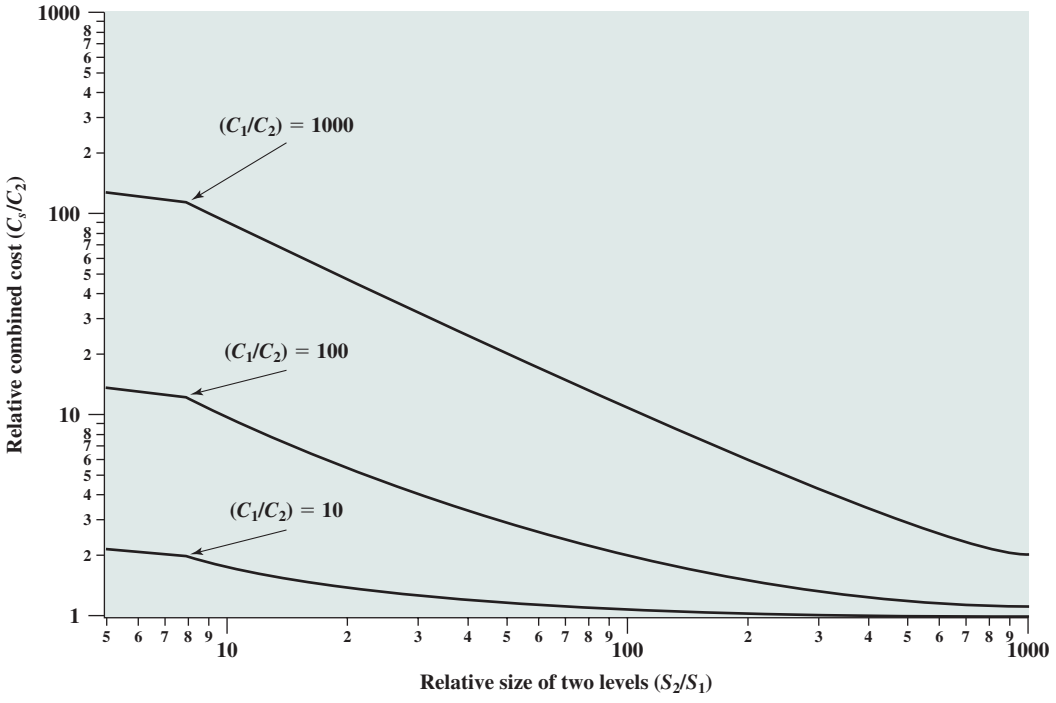


Figure 4.21 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

Next, consider access time. For a two-level memory to provide a significant performance improvement, we need to have T_s approximately equal to $T_1 (T_s \approx T_1)$. Given that T_1 is much less than $T_2 (T_1 << T_2)$, a hit ratio of close to 1 is needed.

So we would like M1 to be small to hold down cost, and large to improve the hit ratio and therefore the performance. Is there a size of M1 that satisfies both requirements to a reasonable extent? We can answer this question with a series of subquestions:

- What value of hit ratio is needed so that $T_s \approx T_1$?
- What size of M1 will assure the needed hit ratio?
- Does this size satisfy the cost requirement?

To get at this, consider the quantity T_1/T_s, which is referred to as the *access efficiency*. It is a measure of how close average access time (T_s) is to M1 access time (T_1). From Equation (4.2),

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\dfrac{T_2}{T_1}} \tag{4.4}$$

Figure 4.22 plots T_1/T_s as a function of the hit ratio H, with the quantity T_2/T_1 as a parameter. Typically, on-chip cache access time is about 25 to 50 times faster than main memory access time (i.e., T_2/T_1 is 25 to 50), off-chip cache access time

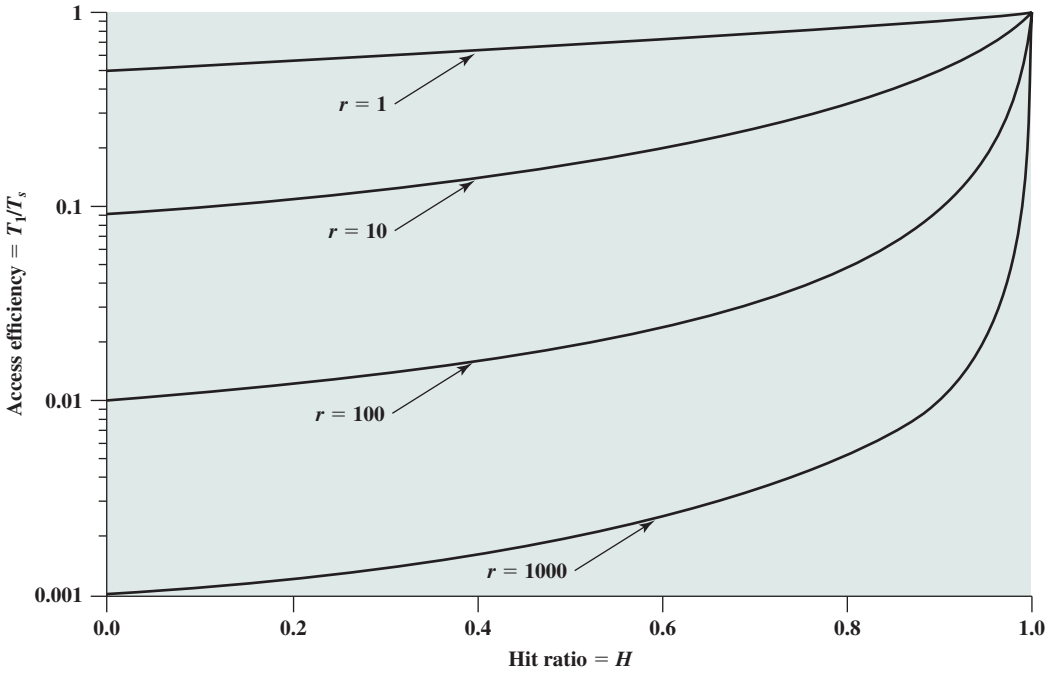


Figure 4.22 Access Efficiency as a Function of Hit Ratio ($r = T_2/T_1$)

is about 5 to 15 times faster than main memory access time (i.e., T_2/T_1 is 5 to 15), and main memory access time is about 1000 times faster than disk access time ($T_2/T_1 = 1000$). Thus, a hit ratio in the range of near 0.9 would seem to be needed to satisfy the performance requirement.

We can now phrase the question about relative memory size more exactly. Is a hit ratio of, say, 0.8 or better reasonable for $S_1 << S_2$? This will depend on a number of factors, including the nature of the software being executed and the details of the design of the two-level memory. The main determinant is, of course, the degree of locality. Figure 4.23 suggests the effect that locality has on the hit ratio. Clearly, if M1 is the same size as M2, then the hit ratio will be 1.0: All of the items in M2 are always also stored in M1. Now suppose that there is no locality; that is, references are completely random. In that case the hit ratio should be a strictly linear function of the relative memory size. For example, if M1 is half the size of M2, then at any time half of the items from M2 are also in M1 and the hit ratio will be 0.5. In practice, however, there is some degree of locality in the references. The effects of moderate and strong locality are indicated in the figure. Note that Figure 4.23 is not derived from any specific data or model; the figure suggests the type of performance that is seen with various degrees of locality.

So if there is strong locality, it is possible to achieve high values of hit ratio even with relatively small upper-level memory size. For example, numerous studies have shown that rather small cache sizes will yield a hit ratio above 0.75 *regardless of the size of main memory* (e.g., [AGAR89], [PRZY88], [STRE83], and [SMIT82]). A cache in the range of 1K to 128K words is generally adequate, whereas main

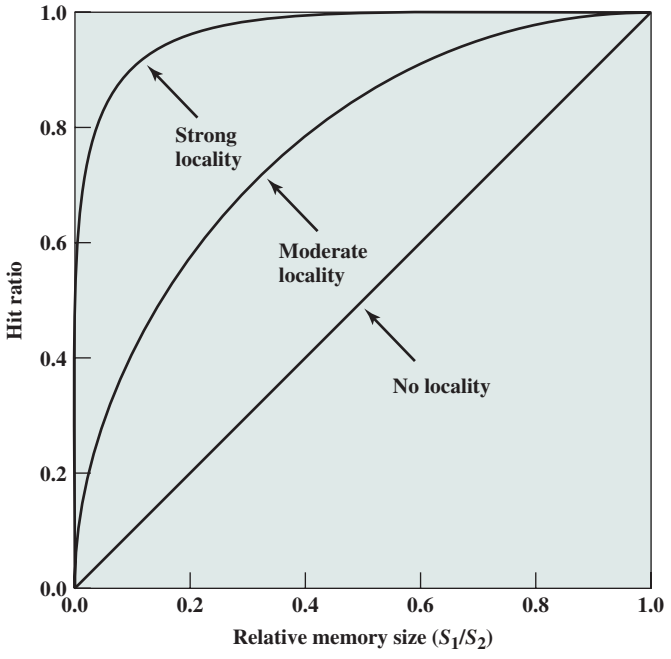


Figure 4.23 Hit Ratio as a Function of Relative Memory Size

memory is now typically in the gigabyte range. When we consider virtual memory and disk cache, we will cite other studies that confirm the same phenomenon, namely that a relatively small M1 yields a high value of hit ratio because of locality.

This brings us to the last question listed earlier: Does the relative size of the two memories satisfy the cost requirement? The answer is clearly yes. If we need only a relatively small upper-level memory to achieve good performance, then the average cost per bit of the two levels of memory will approach that of the cheaper lower-level memory.

Please note that with L2 cache, or even L2 and L3 caches, involved, analysis is much more complex. See [PEIR99] and [HAND98] for discussions.

CHAPTER 5

Internal Memory

5.1 Semiconductor Main Memory
- Organization
- DRAM and SRAM
- Types of ROM
- Chip Logic
- Chip Packaging
- Module Organization
- Interleaved Memory

5.2 Error Correction

5.3 DDR DRAM
- Synchronous DRAM
- DDR SDRAM

5.4 Flash Memory
- Operation
- NOR and NAND Flash Memory

5.5 Newer Nonvolatile Solid-State Memory Technologies
- STT-RAM
- PCRAM
- ReRAM

5.6 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Present an overview of the principle types of semiconductor main memory.
- Understand the operation of a basic code that can detect and correct single-bit errors in 8-bit words.
- Summarize the properties of contemporary **DDR DRAM** organizations.
- Understand the difference between **NOR** and **NAND** flash memory.
- Present an overview of the newer nonvolatile solid-state memory technologies.

We begin this chapter with a survey of semiconductor main memory subsystems, including ROM, DRAM, and SRAM memories. Then we look at error control techniques used to enhance memory reliability. Following this, we look at more advanced DRAM architectures.

5.1 SEMICONDUCTOR MAIN MEMORY

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as *cores*. Hence, main memory was often referred to as *core*, a term that persists to this day. The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. Key aspects of this technology are explored in this section.

Organization

The basic element of a **semiconductor memory** is the memory cell. Although a variety of electronic technologies are used, all semiconductor memory cells share certain properties:

- They exhibit two stable (or semistable) states, which can be used to represent binary 1 and 0.
- They are capable of being written into (at least once), to set the state.
- They are capable of being read to sense the state.

Figure 5.1 depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state. The details of the internal organization, functioning, and timing of the memory cell depend on the specific integrated circuit technology used and are beyond the scope of this book, except for a brief summary. For our purposes, we will take it as given that individual cells can be selected for reading and writing operations.

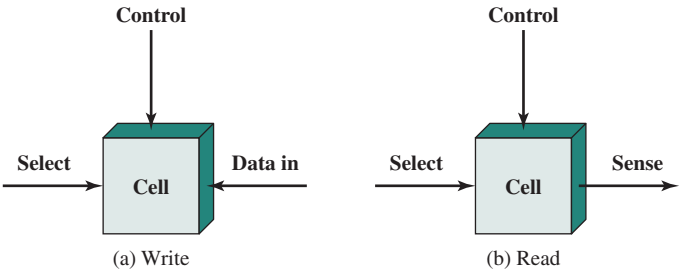


Figure 5.1 Memory Cell Operation

DRAM and SRAM

All of the memory types that we will explore in this chapter are random access. That is, individual words of memory are directly accessed through wired-in addressing logic.

Table 5.1 lists the major types of semiconductor memory. The most common is referred to as **random-access memory (RAM)**. This is, in fact, a misuse of the term, because all of the types listed in the table are random access. One distinguishing characteristic of memory that is designated as RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals.

The other distinguishing characteristic of traditional RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are DRAM and SRAM. Newer forms of RAM, discussed in Section 5.5, are nonvolatile.

DYNAMIC RAM RAM technology is divided into two technologies: dynamic and static. A **dynamic RAM (DRAM)** is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge refreshing to maintain data storage. The term

Table 5.1 Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

dynamic refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure 5.2a is a typical DRAM structure for an individual cell that stores one bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.

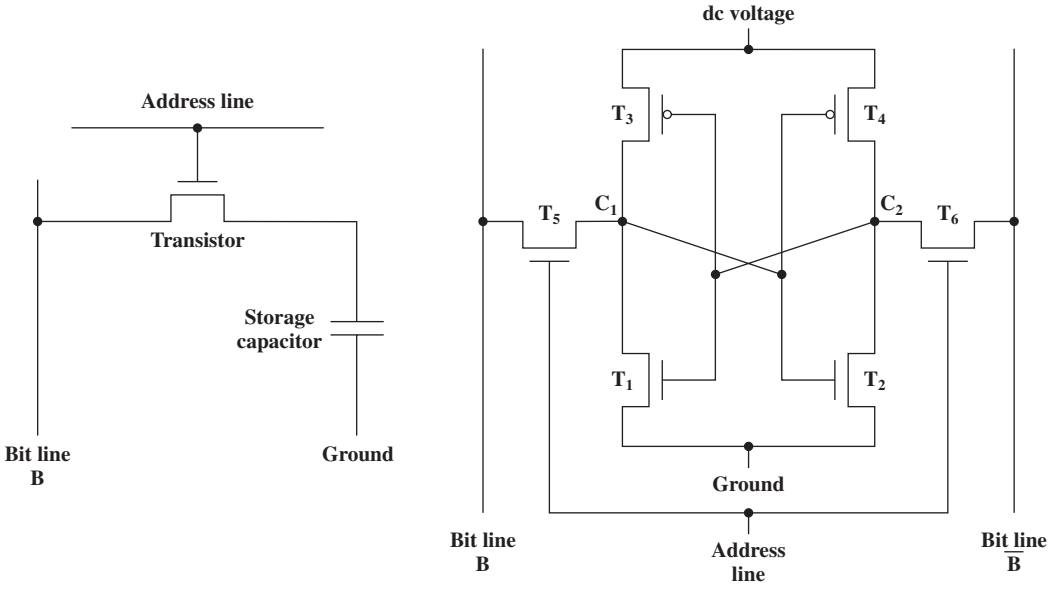
For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains a logic 1 or a logic 0. The readout from the cell discharges the capacitor, which must be restored to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

STATIC RAM In contrast, a **static RAM (SRAM)** is a digital device that uses the same logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations (see Chapter 11 for a description of flip-flops). A static RAM will hold its data as long as power is supplied to it.

Figure 5.2b is a typical SRAM structure for an individual cell. Four transistors (T_1, T_2, T_3, T_4) are cross connected in an arrangement that produces a stable logic



(a) Dynamic RAM (DRAM) cell

(b) Static RAM (SRAM) cell

Figure 5.2 Typical Memory Cell Structures

state. In logic state 1, point C_1 is high and point C_2 is low; in this state, T_1 and T_4 are off and T_2 and T_3 are on.[1] In logic state 0, point C_1 is low and point C_2 is high; in this state, T_1 and T_4 are on and T_2 and T_3 are off. Both states are stable as long as the direct current (dc) voltage is applied. Unlike the DRAM, no refresh is needed to retain data.

As in the DRAM, the SRAM address line is used to open or close a switch. The address line controls two transistors (T_5 and T_6). When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation. For a write operation, the desired bit value is applied to line B, while its complement is applied to line $\overline{B}$. This forces the four transistors (T_1, T_2, T_3, T_4) into the proper state. For a read operation, the bit value is read from line B.

SRAM VERSUS DRAM Both static and dynamic RAMs are volatile; that is, power must be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense (smaller cells = more cells per unit area) and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the smaller variable cost of DRAM cells. Thus, DRAMs tend to be favored for large memory requirements. A final point is that SRAMs are somewhat faster than DRAMs. Because of these relative characteristics, SRAM is used for cache memory (both on and off chip), and DRAM is used for main memory.

Types of ROM

As the name suggests, a **read-only memory (ROM)** contains a permanent pattern of data that cannot be changed. A ROM is nonvolatile; that is, no power source is required to maintain the bit values in memory. While it is possible to read a ROM, it is not possible to write new data into it. An important application of ROMs is microprogramming, discussed in Part Four. Other potential applications include

- Library subroutines for frequently wanted functions
- System programs
- Function tables

For a modest-sized requirement, the advantage of ROM is that the data or program is permanently in main memory and need never be loaded from a secondary storage device.

A ROM is created like any other integrated circuit chip, with the data actually wired into the chip as part of the fabrication process. This presents two problems:

- The data insertion step includes a relatively large fixed cost, whether one or thousands of copies of a particular ROM are fabricated.
- There is no room for error. If one bit is wrong, the whole batch of ROMs must be thrown out.

When only a small number of ROMs with a particular memory content is needed, a less expensive alternative is the **programmable ROM (PROM)**. Like the

[1]The circles associated with T_3 and T_4 in Figure 5.2b indicate signal negation.

ROM, the PROM is **nonvolatile** and may be written into only once. For the PROM, the writing process is performed electrically and may be performed by a supplier or customer at a time later than the original chip fabrication. Special equipment is required for the writing or "programming" process. PROMs provide flexibility and convenience. The ROM remains attractive for high-volume production runs.

Another variation on read-only memory is the **read-mostly memory**, which is useful for applications in which read operations are far more frequent than write operations but for which nonvolatile storage is required. There are three common forms of read-mostly memory: EPROM, EEPROM, and flash memory.

The optically **erasable programmable read-only memory (EPROM)** is read and written electrically, as with PROM. However, before a write operation, all the storage cells must be erased to the same initial state by exposure of the packaged chip to ultraviolet radiation. Erasure is performed by shining an intense ultraviolet light through a window that is designed into the memory chip. This erasure process can be performed repeatedly; each erasure can take as much as 20 minutes to perform. Thus, the EPROM can be altered multiple times and, like the ROM and PROM, holds its data virtually indefinitely. For comparable amounts of storage, the EPROM is more expensive than PROM, but it has the advantage of the multiple update capability.

A more attractive form of read-mostly memory is **electrically erasable programmable read-only memory (EEPROM)**. This is a read-mostly memory that can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation, on the order of several hundred microseconds per byte. The EEPROM combines the advantage of nonvolatility with the flexibility of being updatable in place, using ordinary bus control, address, and data lines. EEPROM is more expensive than EPROM and also is less dense, supporting fewer bits per chip.

Another form of semiconductor memory is **flash memory** (so named because of the speed with which it can be reprogrammed). First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or "flash." However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

Chip Logic

As with other integrated circuit products, semiconductor memory comes in packaged chips (Figure 1.11). Each chip contains an array of memory cells.

In the memory hierarchy as a whole, we saw that there are trade-offs among speed, density, and cost. These trade-offs also exist when we consider the organization of memory cells and functional logic on a chip. For semiconductor memories, one of the key design issues is the number of bits of data that may be read/written at a time. At one extreme is an organization in which the physical arrangement of cells in the array is the same as the logical arrangement (as perceived by the processor) of words in memory. The array is organized into W words of B bits each.

For example, a 16-Mbit chip could be organized as 1M 16-bit words. At the other extreme is the so-called 1-bit-per-chip organization, in which data are read/written one bit at a time. We will illustrate memory chip organization with a DRAM; ROM organization is similar, though simpler.

Figure 5.3 shows a typical organization of a 16-Mbit DRAM. In this case, 4 bits are read or written at a time. Logically, the memory array is organized as four square arrays of 2048 by 2048 elements. Various physical arrangements are possible. In any case, the elements of the array are connected by both horizontal (row) and vertical (column) lines. Each horizontal line connects to the Select terminal of each cell in its row; each vertical line connects to the Data-In/Sense terminal of each cell in its column.

Address lines supply the address of the word to be selected. A total of $\log_2 W$ lines are needed. In our example, 11 address lines are needed to select one of 2048 rows. These 11 lines are fed into a row decoder, which has 11 lines of input and 2048 lines for output. The logic of the decoder activates a single one of the 2048 outputs depending on the bit pattern on the 11 input lines ($2^{11} = 2048$).

An additional 11 address lines select one of 2048 columns of 4 bits per column. Four data lines are used for the input and output of 4 bits to and from a data buffer. On input (write), the bit driver of each bit line is activated for a 1 or 0 according to the value of the corresponding data line. On output (read), the value of each bit line is passed through a sense amplifier and presented to the data lines. The row line selects which row of cells is used for reading or writing.

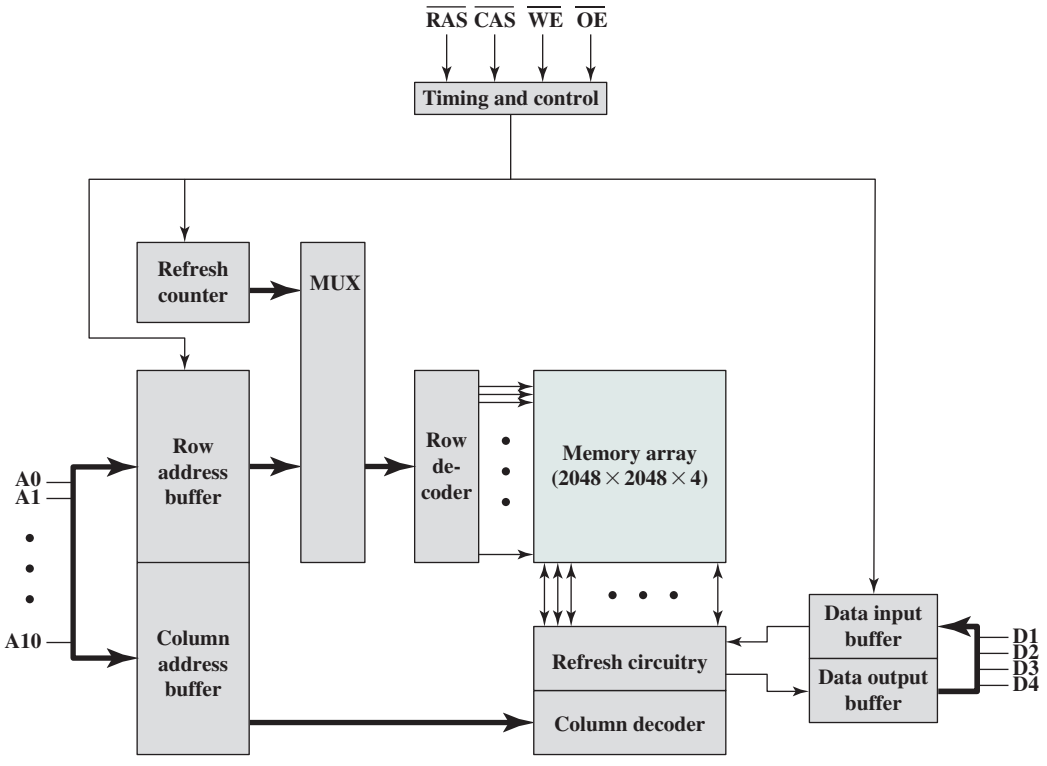


Figure 5.3 Typical 16-Mbit DRAM (4M × 4)

Because only 4 bits are read/written to this DRAM, there must be multiple DRAMs connected to the memory controller to read/write a word of data to the bus.

Note that there are only 11 address lines (A0–A10), half the number you would expect for a 2048 × 2048 array. This is done to save on the number of pins. The 22 required address lines are passed through select logic external to the chip and multiplexed onto the 11 address lines. First, 11 address signals are passed to the chip to define the row address of the array, and then the other 11 address signals are presented for the column address. These signals are accompanied by row address select ($\overline{RAS}$) and column address select ($\overline{CAS}$) signals to provide timing to the chip.

The write enable ($\overline{WE}$) and output enable ($\overline{OE}$) pins determine whether a write or read operation is performed. Two other pins, not shown in Figure 5.3, are ground (Vss) and a voltage source (Vcc).

As an aside, multiplexed addressing plus the use of square arrays result in a quadrupling of memory size with each new generation of memory chips. One more pin devoted to addressing doubles the number of rows and columns, and so the size of the chip memory grows by a factor of 4.

Figure 5.3 also indicates the inclusion of refresh circuitry. All DRAMs require a refresh operation. A simple technique for refreshing is, in effect, to disable the DRAM chip while all data cells are refreshed. The refresh counter steps through all of the row values. For each row, the output lines from the refresh counter are supplied to the row decoder and the RAS line is activated. The data are read out and written back into the same location. This causes each cell in the row to be refreshed.

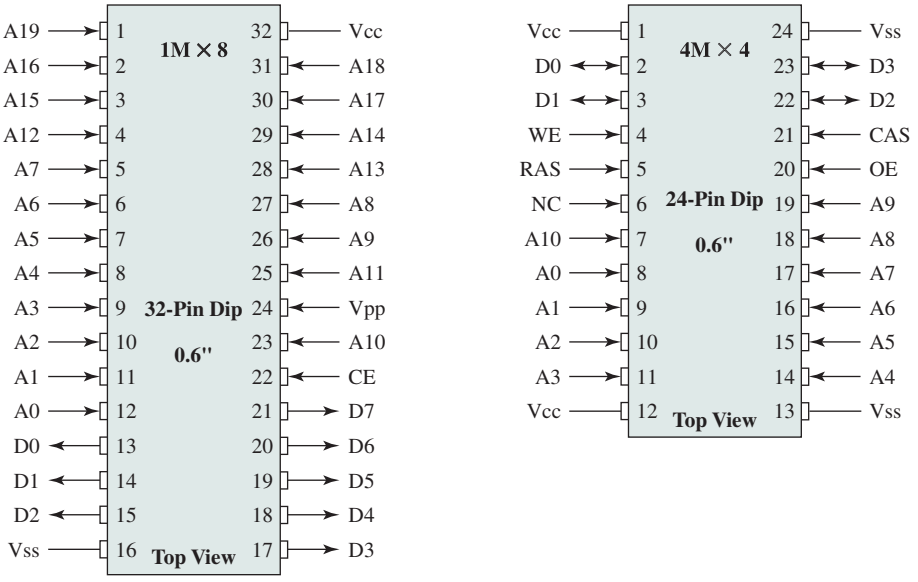
Chip Packaging

As was mentioned in Chapter 2, an integrated circuit is mounted on a package that contains pins for connection to the outside world.

Figure 5.4a shows an example EPROM package, which is an 8-Mbit chip organized as 1M × 8. In this case, the organization is treated as a one-word-per-chip package. The package includes 32 pins, which is one of the standard chip package sizes. The pins support the following signal lines:

- The address of the word being accessed. For 1M words, a total of 20 ($2^{20} = 1M$) pins are needed (A0–A19).
- The data to be read out, consisting of 8 lines (D0–D7).
- The power supply to the chip (V_{cc}).
- A ground pin (V_{ss}).
- A chip enable (CE) pin. Because there may be more than one memory chip, each of which is connected to the same address bus, the CE pin is used to indicate whether or not the address is valid for this chip. The CE pin is activated by logic connected to the higher-order bits of the address bus (i.e., address bits above A19). The use of this signal is illustrated presently.
- A program voltage (V_{pp}) that is supplied during programming (write operations).

A typical DRAM pin configuration is shown in Figure 5.4b, for a 16-Mbit chip organized as 4M × 4. There are several differences from a ROM chip. Because a RAM can be updated, the data pins are input/output. The write enable (WE) and output enable (OE) pins indicate whether this is a write or read operation.



(a) 8-Mbit EPROM (b) 16-Mbit DRAM

Figure 5.4 Typical Memory Package Pins and Signals

Because the DRAM is accessed by row and column, and the address is multiplexed, only 11 address pins are needed to specify the 4M row/column combinations ($2^{11} \times 2^{11} = 2^{22} = 4M$). The functions of the row address select (RAS) and column address select (CAS) pins were discussed previously. Finally, the no connect (NC) pin is provided so that there are an even number of pins.

Module Organization

If a RAM chip contains only one bit per word, then clearly we will need at least a number of chips equal to the number of bits per word. As an example, Figure 5.5 shows how a memory module consisting of 256K 8-bit words could be organized. For 256K words, an 18-bit address is needed and is supplied to the module from some external source (e.g., the address lines of a bus to which the module is attached). The address is presented to 8 256K × 1-bit chips, each of which provides the input/output of one bit.

This organization works as long as the size of memory equals the number of bits per chip. In the case in which larger memory is required, an array of chips is needed. Figure 5.6 shows the possible organization of a memory consisting of 1M word by 8 bits per word. In this case, we have four columns of chips, each column containing 256K words arranged as in Figure 5.5. For 1M word, 20 address lines are needed. The 18 least significant bits are routed to all 32 modules. The high-order 2 bits are input to a group select logic module that sends a chip enable signal to one of the four columns of modules.

Interleaved Memory

Main memory is composed of a collection of DRAM memory chips. A number of chips can be grouped together to form a *memory bank*. It is possible to organize the memory

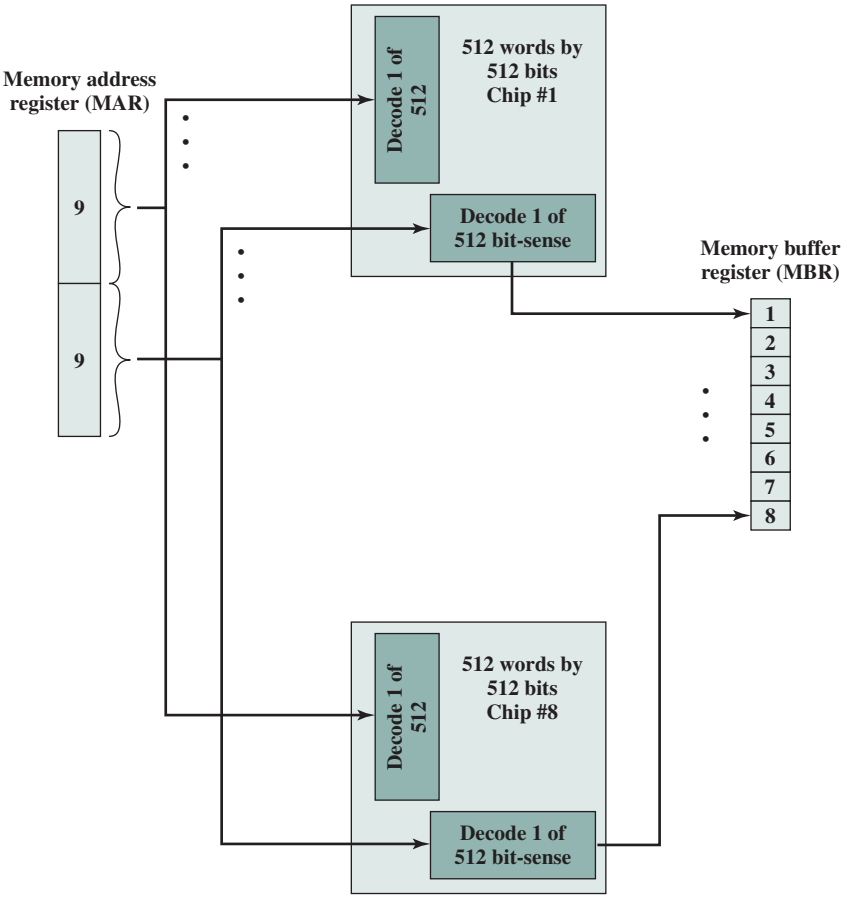


Figure 5.5 256-KByte Memory Organization

banks in a way known as interleaved memory. Each bank is independently able to service a memory read or write request, so that a system with K banks can service K requests simultaneously, increasing memory read or write rates by a factor of K. If consecutive words of memory are stored in different banks, then the transfer of a block of memory is speeded up. Appendix G explores the topic of interleaved memory.

Interleaved Memory Simulator

5.2 ERROR CORRECTION

A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A **hard failure** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or

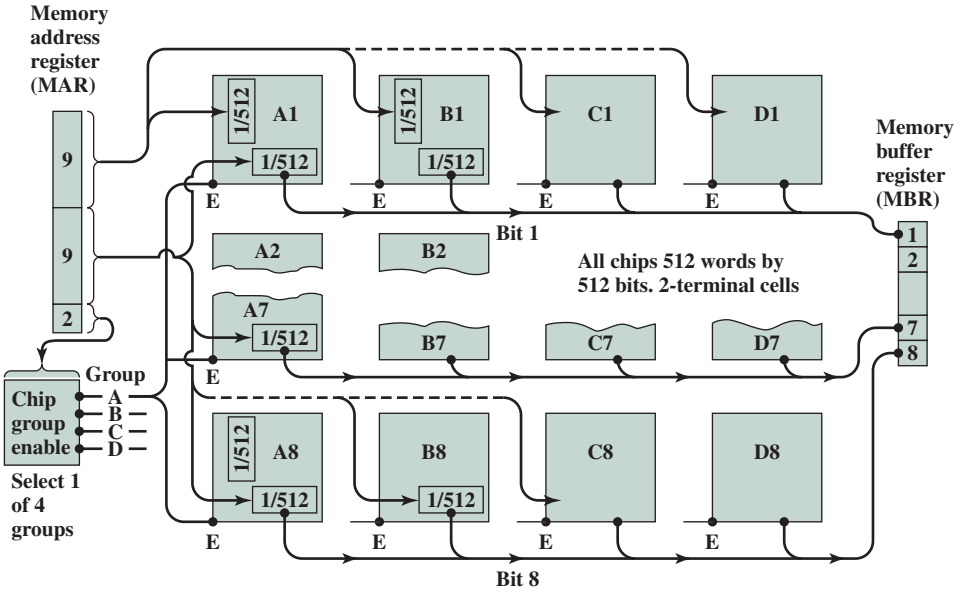


Figure 5.6 1-MB Memory Organization

switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A **soft error** is a random, nondestructive event that alters the contents of one or more memory cells without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials. Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 5.7 illustrates in general terms how the process is carried out. When data are to be written into memory, a calculation, depicted as a function f, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an M-bit word of data is to be stored and the code is of length K bits, then the actual size of the stored word is $M + K$ bits.

When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of K code bits is generated from the M data bits and compared with the fetched code bits. The comparison yields one of three results:

- No errors are detected. The fetched data bits are sent out.
- An error is detected, and it is possible to correct the error. The data bits plus **error correction** bits are fed into a corrector, which produces a corrected set of M bits to be sent out.
- An error is detected, but it is not possible to correct it. This condition is reported.

Codes that operate in this fashion are referred to as **error-correcting codes**. A code is characterized by the number of bit errors in a word that it can correct and detect.

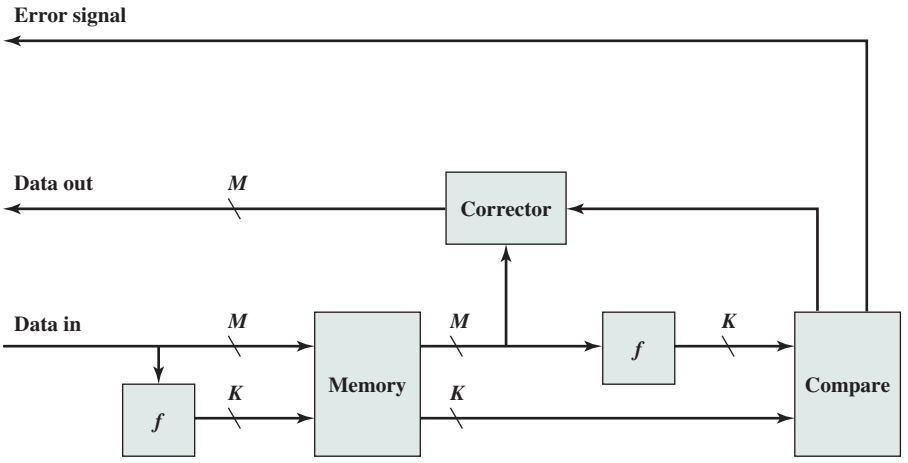


Figure 5.7 Error-Correcting Code Function

The simplest of the error-correcting codes is the **Hamming code** devised by Richard Hamming at Bell Laboratories. Figure 5.8 uses Venn diagrams to illustrate the use of this code on 4-bit words ($M = 4$). With three intersecting circles, there are seven compartments. We assign the 4 data bits to the inner compartments (Figure 5.8a). The remaining compartments are filled with what are called *parity bits.* Each parity bit is chosen so that the total number of 1s in its circle is even (Figure 5.8b). Thus, because circle A includes three data 1s, the parity bit in that circle is set to 1. Now, if an error changes one of the data bits (Figure 5.8c), it is easily found. By checking the parity bits, discrepancies are found in circle A and circle C but not in circle B. Only one of the seven compartments is in A and C but not B (Figure 5.8d). The error can therefore be corrected by changing that bit.

To clarify the concepts involved, we will develop a code that can detect and correct single-bit errors in 8-bit words.

To start, let us determine how long the code must be. Referring to Figure 5.7, the comparison logic receives as input two K-bit values. A bit-by-bit comparison is done by taking the exclusive-OR of the two inputs. The result is called the *syndrome word.* Thus, each bit of the **syndrome** is 0 or 1 according to if there is or is not a match in that bit position for the two inputs.

The syndrome word is therefore K bits wide and has a range between 0 and 2^K-1. The value 0 indicates that no error was detected, leaving 2^K-1 values to indicate, if there is an error, which bit was in error. Now, because an error could occur on any of the M data bits or K check bits, we must have

$$2^K-1 \geq M + K$$

This inequality gives the number of bits needed to correct a single bit error in a word containing M data bits. For example, for a word of 8 data bits ($M = 8$), we have

- $K = 3$: $2^3-1 < 8 + 3$
- $K = 4$: $2^4-1 > 8 + 4$

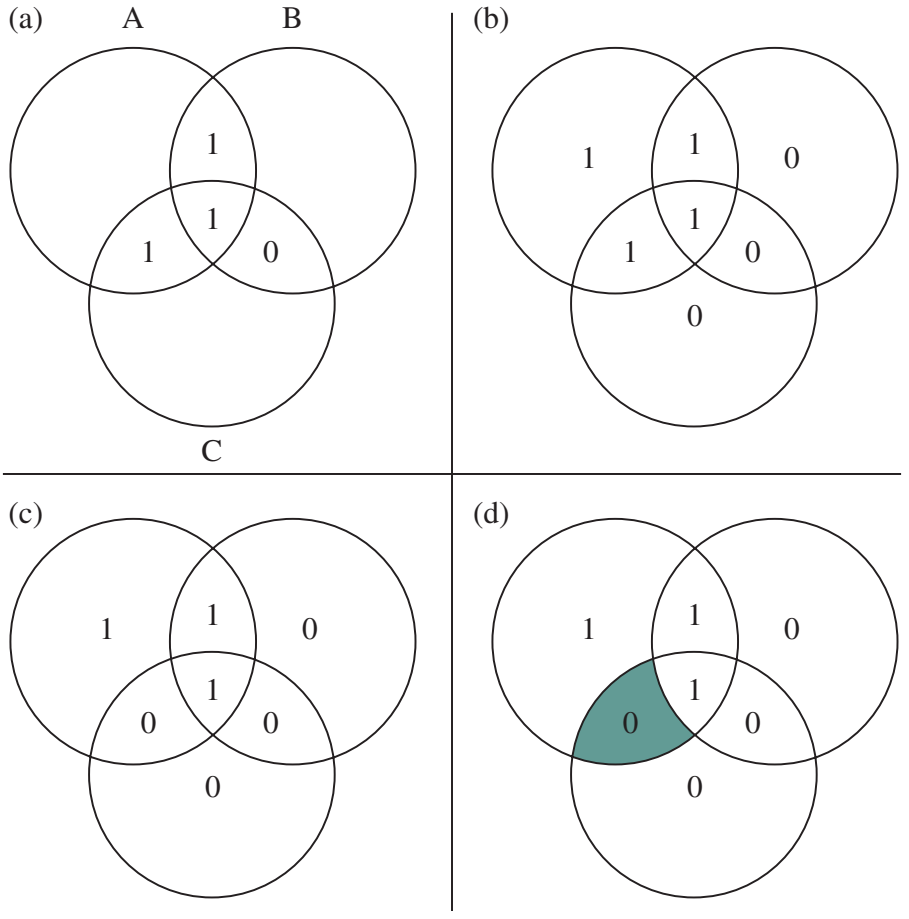


Figure 5.8 Hamming Error-Correcting Code

Thus, eight data bits require four check bits. The first three columns of Table 5.2 lists the number of check bits required for various data word lengths.

For convenience, we would like to generate a 4-bit syndrome for an 8-bit data word with the following characteristics:

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in Figure 5.9. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check

Table 5.2 Increase in Word Length with Error Correction

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	**Check Bits**	**% Increase**	**Check Bits**	**% Increase**
8	4	50.0	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

bits. The check bits are calculated as follows, where the symbol $\oplus$ designates the exclusive-OR operation:

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$$
$$C2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$$
$$C4 = D2 \oplus D3 \oplus D4 \oplus D8$$
$$C8 = \oplus D5 \oplus D6 \oplus D7 \oplus D8$$

Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit. Thus, data bit positions 3, 5, 7, 9, and 11 (D1, D2, D4, D5, D7) all contain a 1 in the least significant bit of their position number as does C1; bit positions 3, 6, 7, 10, and 11 all contain a 1 in the second bit position, as does C2; and so on. Looked at another way, bit position n is checked by those bits C_i such that $\sum_i = n$. For example, position 7 is checked by bits in position 4, 2, and 1; and $7 = 4 + 2 + 1$.

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Figure 5.9 Layout of Data Bits and Check Bits

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C4 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

When the new check bits are compared with the old check bits, the syndrome word is formed:

$$\begin{array}{rcccc} & C8 & C4 & C2 & C1 \\ & 0 & 1 & 1 & 1 \\ \oplus & 0 & 0 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.

Figure 5.10 illustrates the preceding calculation. The data and check bits are positioned properly in the 12-bit word. Four of the data bits have a value 1 (shaded in the table), and their bit position values are XORed to produce the Hamming code 0111, which forms the four check digits. The entire block that is stored is 001101001111. Suppose now that data bit 3, in bit position 6, sustains an error and is changed from 0 to 1. The resulting block is 001101101111, with a Hamming code of 0001. An XOR of the Hamming code and all of the bit position values for nonzero data bits results in 0110. The nonzero result detects an error and indicates that the error is in bit position 6.

The code just described is known as a **single-error-correcting (SEC) code**. More commonly, semiconductor memory is equipped with a **single-error-correcting, double-error-detecting (SEC-DED) code**. As Table 5.2 shows, such codes require one additional bit compared with SEC codes.

Figure 5.11 illustrates how such a code works, again with a 4-bit data word. The sequence shows that if two errors occur (Figure 5.11c), the checking procedure goes astray (d) and worsens the problem by creating a third error (e). To overcome

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1
Word stored as	0	0	1	1	0	1	0	0	1	1	1	1
Word fetched as	0	0	1	1	0	1	1	0	1	1	1	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Check bit					0				0		0	1

Figure 5.10 Check Bit Calculation

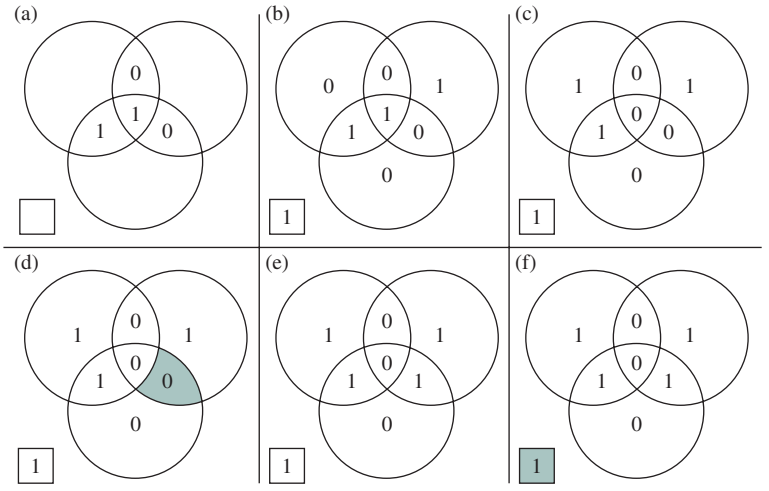


Figure 5.11 Hamming SEC-DEC Code

the problem, an eighth bit is added that is set so that the total number of 1s in the diagram is even. The extra parity bit catches the error (f).

An error-correcting code enhances the reliability of the memory at the cost of added complexity. With a 1-bit-per-chip organization, an SEC-DED code is generally considered adequate. For example, the IBM 30xx implementations used an 8-bit SEC-DED code for each 64 bits of data in main memory. Thus, the size of main memory is actually about 12% larger than is apparent to the user. The VAX computers used a 7-bit SEC-DED for each 32 bits of memory, for a 22% overhead. Contemporary DRAM systems may have anywhere from 7% to 20% overhead [SHAR03].

5.3 DDR DRAM

As discussed in Chapter 1, one of the most critical system bottlenecks when using high-performance processors is the interface to internal main memory. This interface is the most important pathway in the entire computer system. The basic building block of main memory remains the DRAM chip, as it has for decades; until recently, there had been no significant changes in DRAM architecture since the early 1970s. The traditional DRAM chip is constrained both by its internal architecture and by its interface to the processor's memory bus.

We have seen that one attack on the performance problem of DRAM main memory has been to insert one or more levels of high-speed SRAM cache between the DRAM main memory and the processor. But SRAM is much costlier than DRAM, and expanding cache size beyond a certain point yields diminishing returns.

In recent years, a number of enhancements to the basic DRAM architecture have been explored. The schemes that currently dominate the market are SDRAM and DDR-DRAM. We examine each of these in turn.

Synchronous DRAM

One of the most widely used forms of DRAM is the **synchronous DRAM (SDRAM)**. Unlike the traditional DRAM, which is asynchronous, the SDRAM exchanges data with the processor synchronized to an external clock signal and running at the full speed of the processor/memory bus without imposing wait states.

In a typical DRAM, the processor presents addresses and control levels to the memory, indicating that a set of data at a particular location in memory should be either read from or written into the DRAM. After a delay, the access time, the DRAM either writes or reads the data. During the access-time delay, the DRAM performs various internal functions, such as activating the high capacitance of the row and column lines, sensing the data, and routing the data out through the output buffers. The processor must simply wait through this delay, slowing system performance.

With synchronous access, the DRAM moves data in and out under control of the system clock. The processor or other master issues the instruction and address information, which is latched by the DRAM. The DRAM then responds after a set number of clock cycles. Meanwhile, the master can safely do other tasks while the SDRAM is processing the request.

Figure 5.12 shows the internal logic of a typical 256-Mb SDRAM typical of SDRAM organization, and Table 5.3 defines the various pin assignments. The

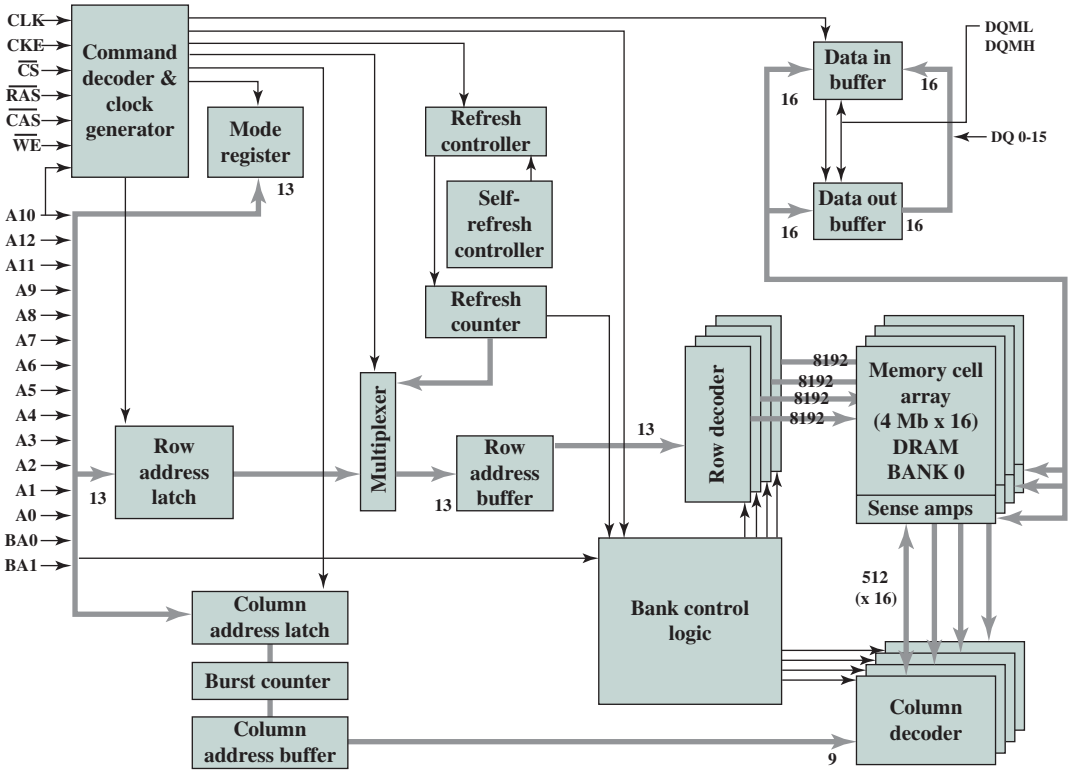


Figure 5.12 256-Mb Synchronous Dynamic RAM (SDRAM)

Table 5.3 SDRAM Pin Assignments

A0 to A13	Address inputs
BA0, BA1	Bank address lines
CLK	Clock input
CKE	Clock enable
$\overline{CS}$	Chip select
$\overline{RAS}$	Row address strobe
$\overline{CAS}$	Column address strobe
$\overline{WE}$	Write enable
DQ0 to DQ7	Data input/output
DQM	Data mask

SDRAM employs a burst mode to eliminate the address setup time and row and column line precharge time after the first access. In burst mode, a series of data bits can be clocked out rapidly after the first bit has been accessed. This mode is useful when all the bits to be accessed are in sequence and in the same row of the array as the initial access. In addition, the SDRAM has a multiple-bank internal architecture that improves opportunities for on-chip parallelism.

The mode register and associated control logic is another key feature differentiating SDRAMs from conventional DRAMs. It provides a mechanism to customize the SDRAM to suit specific system needs. The mode register specifies the burst length, which is the number of separate units of data synchronously fed onto the bus. The register also allows the programmer to adjust the latency between receipt of a read request and the beginning of data transfer.

The SDRAM performs best when it is transferring large blocks of data sequentially, such as for applications like word processing, spreadsheets, and multimedia.

Figure 5.13 shows an example of SDRAM operation. In this case, the burst length is 4 and the latency is 2. The burst read command is initiated by having $\overline{CS}$ and $\overline{CAS}$ low while holding $\overline{RAS}$ and $\overline{WE}$ high at the rising edge of the clock. The address inputs determine the starting column address for the burst, and the mode register sets the type of burst (sequential or interleave) and the burst length (1, 2, 4, 8, full page). The delay from the start of the command to when the data from the first cell appears on the outputs is equal to the value of the $\overline{CAS}$ latency that is set in the mode register.

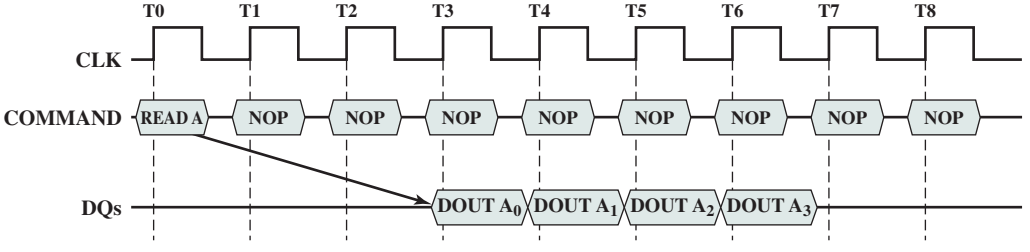


Figure 5.13 SDRAM Read Timing (burst length = 4, $\overline{CAS}$ latency = 2)

DDR SDRAM

Although SDRAM is a significant improvement on asynchronous RAM, it still has shortcomings that unnecessarily limit that I/O data rate that can be achieved. To address these shortcomings a newer version of SDRAM, referred to as double-data-rate DRAM (DDR DRAM) provides several features that dramatically increase the data rate. DDR DRAM was developed by the JEDEC Solid State Technology Association, the Electronic Industries Alliance's semiconductor-engineering-standardization body. Numerous companies make DDR chips, which are widely used in desktop computers and servers.

DDR achieves higher data rates in three ways. First, the data transfer is synchronized to both the rising and falling edge of the clock, rather than just the rising edge. This doubles the data rate; hence the term *double data rate*. Second, DDR uses higher clock rate on the bus to increase the transfer rate. Third, a buffering scheme is used, as explained subsequently.

JEDEC has thus far defined four generations of the DDR technology (Table 5.4). The initial DDR version makes use of a 2-bit prefetch buffer. The prefetch buffer is a memory cache located on the SDRAM chip. It enables the SDRAM chip to preposition bits to be placed on the data bus as rapidly as possible. The DDR I/O bus uses the same clock rate as the memory chip, but because it can handle two bits per cycle, it achieves a data rate that is double the clock rate. The 2-bit prefetch buffer enables the SDRAM chip to keep up with the I/O bus.

To understand the operation of the prefetch buffer, we need to look at it from the point of view of a word transfer. The prefetch buffer size determines how many words of data are fetched (across multiple SDRAM chips) every time a column command is performed with DDR memories. Because the core of the DRAM is much slower than the interface, the difference is bridged by accessing information in parallel and then serializing it out the interface through a multiplexor (MUX). Thus, DDR prefetches two words, which means that every time a read or a write operation is performed, it is performed on two words of data, and bursts out of, or into, the SDRAM over one clock cycle on both clock edges for a total of two consecutive operations. As a result, the DDR I/O interface is twice as fast as the SDRAM core.

Although each new generation of SDRAM results is much greater capacity, the core speed of the SDRAM has not changed significantly from generation to generation. To achieve greater data rates than those afforded by the rather modest increases in SDRAM clock rate, JEDEC increased the buffer size. For DDR2, a 4-bit buffer is used, allowing for words to be transferred in parallel, increasing the effective data rate by a factor of 4. For DDR3, an 8-bit buffer is used and a factor of 8 speedup is achieved (Figure 5.14).

Table 5.4 DDR Characteristics

	DDR1	DDR2	DDR3	DDR4
Prefetch buffer (bits)	2	4	8	8
Voltage level (V)	2.5	1.8	1.5	1.2
Front side bus data rates (Mbps)	200—400	400—1066	800—2133	2133—4266

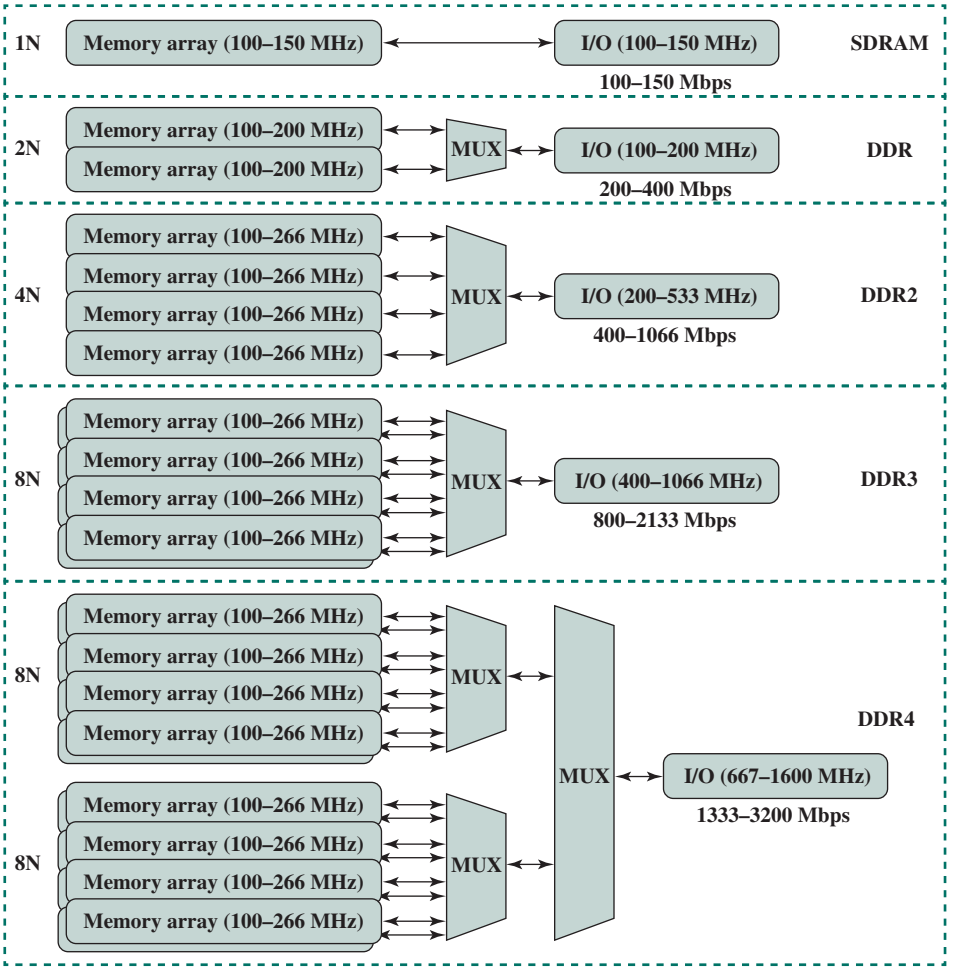


Figure 5.14 DDR Generations

The downside to the prefetch is that it effectively determines the minimum burst length for the SDRAMs. For example, it is very difficult to have an efficient burst length of four words with DDR3's prefetch of eight. Accordingly, the JEDEC designers chose not to increase the buffer size to 16 bits for DDR4, but rather to introduce the concept of a **bank group** [ALLA13]. Bank groups are separate entities such that they allow a column cycle to complete within a bank group, but that column cycle does not impact what is happening in another bank group. Thus, two prefetches of eight can be operating in parallel in the two bank groups. This arrangement keeps the prefetch buffer size the same as for DDR3, while increasing performance as if the prefetch is larger.

Figure 5.14 shows a configuration with two bank groups. With DDR4, up to 4 bank groups can be used.

5.4 FLASH MEMORY

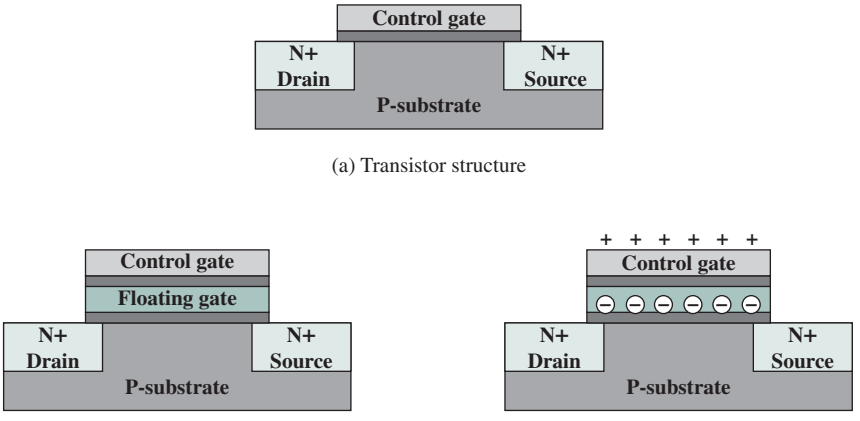
Another form of semiconductor memory is flash memory. Flash memory is used both for internal memory and external memory applications. Here, we provide a technical overview and look at its use for internal memory.

First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or "flash." However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

Operation

Figure 5.15 illustrates the basic operation of a flash memory. For comparison, Figure 5.15a depicts the operation of a transistor. Transistors exploit the properties of semiconductors so that a small voltage applied to the gate can be used to control the flow of a large current between the source and the drain.

In a flash memory cell, a second gate—called a floating gate, because it is insulated by a thin oxide layer—is added to the transistor. Initially, the floating gate does not interfere with the operation of the transistor (Figure 5.15b). In this state, the cell is deemed to represent binary 1. Applying a large voltage across the oxide layer causes electrons to tunnel through it and become trapped on the floating gate, where they remain even if the power is disconnected (Figure 5.15c). In this state, the cell is deemed to represent binary 0. The state of the cell can be read by using external circuitry to test whether the transistor is working or not. Applying a large voltage in the opposite direction removes the electrons from the floating gate, returning to a state of binary 1.



(a) Transistor structure

(b) Flash memory cell in one state

(c) Flash memory cell in zero state

Figure 5.15 Flash Memory Operation

An important characteristic of flash memory is that it is persistent memory, which means that it retains data when there is no power applied to the memory. Thus, it is useful for secondary (external) storage, and as an alternative to random access memory in computers.

NOR and NAND Flash Memory

There are two distinctive types of flash memory, designated as NOR and NAND (Figure 5.16). In **NOR flash memory**, the basic unit of access is a bit, referred to as a *memory cell*. Cells in NOR flash are connected in parallel to the bit lines so that each cell can be read/write/erased individually. If any memory cell of the device is turned on by the corresponding word line, the bit line goes low. This is similar in function to a NOR logic gate.[2]

NAND flash memory is organized in transistor arrays with 16 or 32 transistors in series. The bit line goes low only if all the transistors in the corresponding word lines are turned on. This is similar in function to a NAND logic gate.

Although the specific quantitative values of various characteristics of NOR and NAND are changing year by year, the relative differences between the two types has remained stable. These differences are usefully illustrated by the Kiviat graphs[3] shown in Figure 5.17.

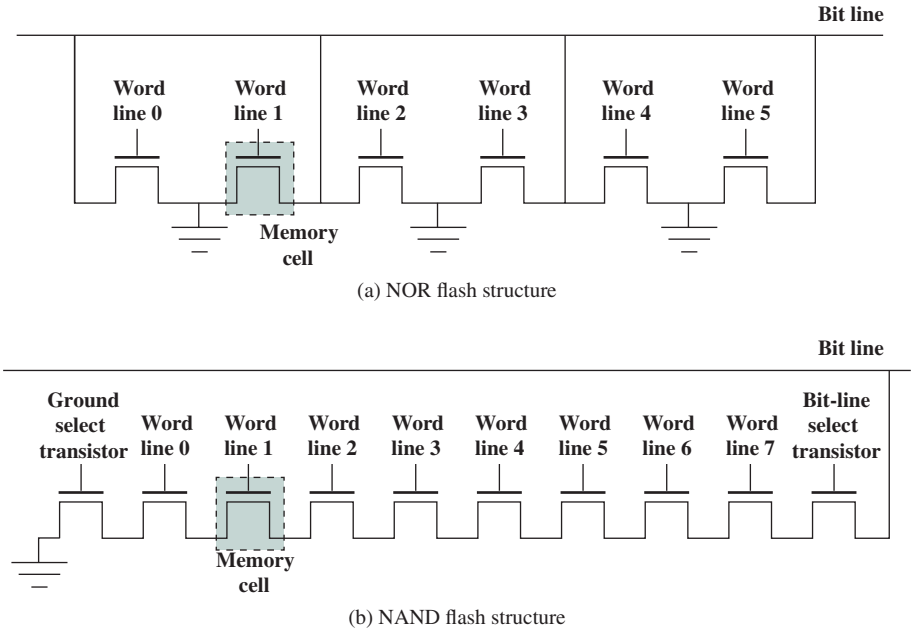


Figure 5.16 Flash Memory Structures

[2]The circles associated with and in Figure 5.2b indicate signal negation.

[3]A Kiviat graph provides a pictorial means of comparing systems along multiple variables [MORR74]. The variables are laid out at as lines of equal angular intervals within a circle, each line going from the center of the circle to the circumference. A given system is defined by one point on each line; the closer to the circumference, the better the value. The points are connected to yield a shape that is characteristic of that system. The more area enclosed in the shape, the "better" is the system.

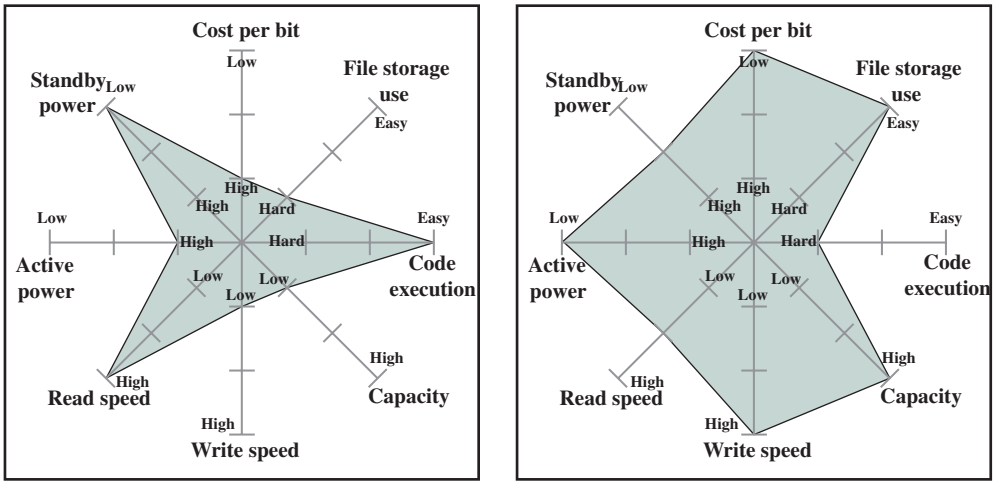


Figure 5.17 Kiviat Graphs for Flash Memory

NOR flash memory provides high-speed random access. It can read and write data to specific locations, and can reference and retrieve a single byte. NAND reads and writes in small blocks. NAND provides higher bit density than NOR and greater write speed. NAND flash does not provide a random-access external address bus so the data must be read on a blockwise basis (also known as page access), where each block holds hundreds to thousands of bits.

For internal memory in embedded systems, NOR flash memory has traditionally been preferred. NAND memory has made some inroads, but NOR remains the dominant technology for internal memory. It is ideally suited for microcontrollers where the amount of program code is relatively small and a certain amount of application data does not vary. For example, the flash memory in Figure 1.16 is NOR memory.

NAND memory is better suited for external memory, such as USB flash drives, memory cards (in digital cameras, MP3 players, etc.), and in what are known as solid-state disks (SSDs). We discuss SSDs in Chapter 6.

5.5 NEWER NONVOLATILE SOLID-STATE MEMORY TECHNOLOGIES

The traditional memory hierarchy has consisted of three levels (Figure 5.18):

- **Static RAM (SRAM):** SRAM provides rapid access time, but is the most expensive and the least dense (bit density). SRAM is suitable for cache memory.
- **Dynamic RAM (DRAM):** Cheaper, denser, and slower than SRAM, DRAM has traditionally been the choice off-chip main memory.
- **Hard disk:** A magnetic disk provides very high bit density and very low cost per bit, with relatively slow access times. It is the traditional choice for external storage as part of the memory hierarchy.

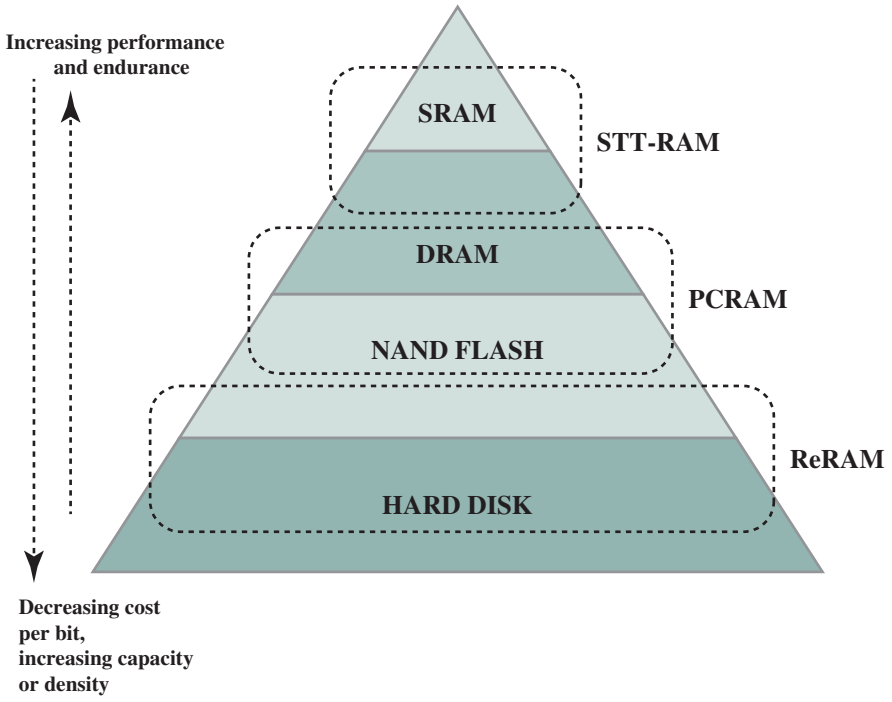


Figure 5.18 Nonvolatile RAM within the Memory Hierarchy

Into this mix, as we have seen, as been added flash memory. Flash memory has the advantage over traditional memory that it is nonvolatile. NOR flash is best suited to storing programs and static application data in embedded systems, while NAND flash has characteristics intermediate between DRAM and hard disks.

Over time, each of these technologies has seen improvements in scaling: higher bit density, higher speed, lower power consumption, and lower cost. However, for semiconductor memory, it is becoming increasingly difficult to continue the pace of improvement [ITRS14].

Recently, there have been breakthroughs in developing new forms of nonvolatile semiconductor memory that continue scaling beyond flash memory. The most promising technologies are spin-transfer torque RAM (STT-RAM), phase-change RAM (PCRAM), and resistive RAM (ReRAM) ([ITRS14], [GOER12]). All of these are in volume production. However, because NAND Flash and to some extent NOR Flash are still dominating the applications, these emerging memories have been used in specialty applications and have not yet fulfilled their original promise to become dominating mainstream high-density nonvolatile memory. This is likely to change in the next few years.

Figure 5.18 shows how these three technologies are likely to fit into the memory hierarchy.

STT-RAM

STT-RAM is a new type of **magnetic RAM (MRAM)**, which features non-volatility, fast writing/reading speed (< 10 ns), and high programming endurance ($> 10^{15}$ cycles) and zero standby power [KULT13]. The storage capability or programmability of MRAM arises from magnetic tunneling junction (MTJ), in which a thin tunneling dielectric is sandwiched between two ferromagnetic layers. One ferromagnetic layer (pinned or reference layer) is designed to have its magnetization pinned, while the magnetization of the other layer (free layer) can be flipped by a write event. An MTJ has a low (high) resistance if the magnetizations of the free layer and the pinned layer are parallel (anti-parallel). In first-generation MRAM design, the magnetization of the free layer is changed by the current-induced magnetic field. In STT-RAM, a new write mechanism, called *polarization-current-induced magnetization switching*, is introduced. For STT-RAM, the magnetization of the free layer is flipped by the electrical current directly. Because the current required to switch an MTJ resistance state is proportional to the MTJ cell area, STT-RAM is believed to have a better scaling property than the first-generation MRAM. Figure 5.19a illustrates the general configuration.

STT-RAM is a good candidate for either cache or main memory.

PCRAM

Phase-change RAM (PCRAM) is the most mature or the new technologies, with an extensive technical literature ([RAOU09], [ZHOU09], [LEE10]).

PCRAM technology is based on a chalcogenide alloy material, which is similar to those commonly used in optical storage media (compact discs and digital versatile discs). The data storage capability is achieved from the resistance differences between an amorphous (high-resistance) and a crystalline (low-resistance) phase of the chalcogenide-based material. In SET operation, the phase change material is crystallized by applying an electrical pulse that heats a significant portion of the cell above its crystallization temperature. In RESET operation, a larger electrical current is applied and then abruptly cut off in order to melt and then quench the material, leaving it in the amorphous state. Figure 5.19b illustrates the general configuration.

PCRAM is a good candidate to replace or supplement DRAM for main memory.

ReRAM

ReRAM (also known as RRAM) works by creating resistance rather than directly storing charge. An electric current is applied to a material, changing the resistance of that material. The resistance state can then be measured and a 1 or 0 is read as the result. Much of the work done on ReRAM to date has focused on finding appropriate materials and measuring the resistance state of the cells. ReRAM designs are low voltage, endurance is far superior to flash memory, and the cells are much smaller—at least in theory. Figure 5.19c shows one ReRam configuration.

ReRAM is a good candidate to replace or supplement both secondary storage and main memory.

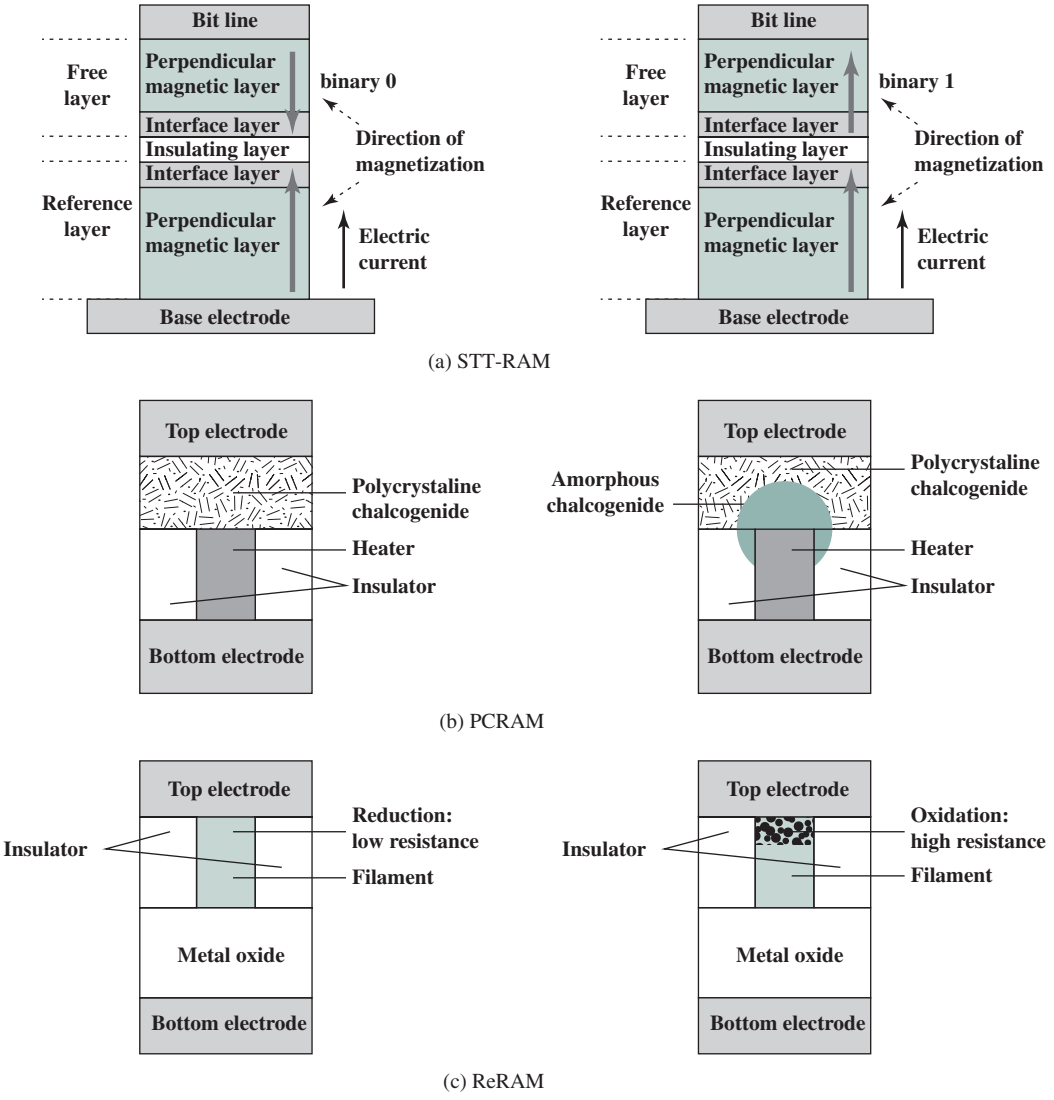


Figure 5.19 Nonvolatile RAM Technologies

5.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

bank group double data rate DRAM (DDR DRAM) dynamic RAM (DRAM)	electrically erasable programmable ROM (EEPROM) erasable programmable ROM (EPROM)	error correcting code (ECC) error correction flash memory Hamming code hard failure

magnetic RAM (MRAM)	read-mostly memory	soft error
NAND flash memory	read-only memory (ROM)	spin-transfer torque RAM (STT-RAM)
nonvolatile memory	resistive RAM (ReRAM)	static RAM (SRAM)
NOR flash memory	semiconductor memory	synchronous DRAM (SDRAM)
phase-change RAM (PCRAM)	single-error-correcting (SEC) code	syndrome
programmable ROM (PROM)	single-error-correcting, double-error-detecting (SEC-DED) code	volatile memory
random access memory (RAM)		

Review Questions

5.1 What are the key properties of semiconductor memory?

5.2 What are two interpretations of the term *random-access memory*?

5.3 What is the difference between DRAM and SRAM in terms of application?

5.4 What is the difference between DRAM and SRAM in terms of characteristics such as speed, size, and cost?

5.5 Explain why one type of RAM is considered to be analog and the other digital.

5.6 What are some applications for ROM?

5.7 What are the differences among EPROM, EEPROM, and flash memory?

5.8 Explain the function of each pin in Figure 5.4b.

5.9 What is a parity bit?

5.10 How is the syndrome for the Hamming code interpreted?

5.11 How does SDRAM differ from ordinary DRAM?

5.12 What is DDR RAM?

5.13 What is the difference between NAND and NOR flash memory?

5.14 List and briefly define three newer nonvolatile solid-state memory technologies.

Problems

5.1 Suggest reasons why RAMs traditionally have been organized as only one bit per chip whereas ROMs are usually organized with multiple bits per chip.

5.2 Consider a dynamic RAM that must be given a refresh cycle 64 times per ms. Each refresh operation requires 150 ns; a memory cycle requires 250 ns. What percentage of the memory's total operating time must be given to refreshes?

5.3 Figure 5.20 shows a simplified timing diagram for a DRAM read operation over a bus. The access time is considered to last from t_1 to t_2. Then there is a recharge time, lasting from t_2 to t_3, during which the DRAM chips will have to recharge before the processor can access them again.

a. Assume that the access time is 60 ns and the recharge time is 40 ns. What is the memory cycle time? What is the maximum data rate this DRAM can sustain, assuming a 1-bit output?

b. Constructing a 32-bit wide memory system using these chips yields what data transfer rate?

5.4 Figure 5.6 indicates how to construct a module of chips that can store 1 MB based on a group of four 256-Kbyte chips. Let's say this module of chips is packaged as a single 1-MB chip, where the word size is 1 byte. Give a high-level chip diagram of how to construct an 8-MB computer memory using eight 1-MB chips. Be sure to show the address lines in your diagram and what the address lines are used for.

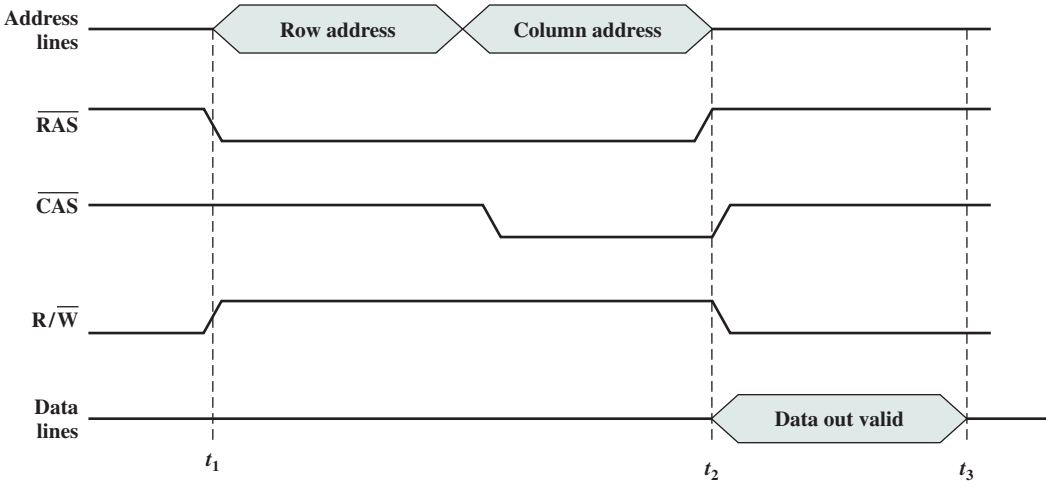


Figure 5.20 Simplified DRAM Read Timing

5.5 On a typical Intel 8086-based system, connected via system bus to DRAM memory, for a read operation, $\overline{RAS}$ is activated by the trailing edge of the Address Enable signal (Figure C.1 in Appendix C). However, due to propagation and other delays, $\overline{RAS}$ does not go active until 50 ns after Address Enable returns to a low. Assume the latter occurs in the middle of the second half of state T_1 (somewhat earlier than in Figure C.1). Data are read by the processor at the end of T_3. For timely presentation to the processor, however, data must be provided 60 ns earlier by memory. This interval accounts for propagation delays along the data paths (from memory to processor) and processor data hold time requirements. Assume a clocking rate of 10 MHz.

a. How fast (access time) should the DRAMs be if no wait states are to be inserted?

b. How many wait states do we have to insert per memory read operation if the access time of the DRAMs is 150 ns?

5.6 The memory of a particular microcomputer is built from 64K × 1 DRAMs. According to the data sheet, the cell array of the DRAM is organized into 256 rows. Each row must be refreshed at least once every 4 ms. Suppose we refresh the memory on a strictly periodic basis.

a. What is the time period between successive refresh requests?

b. How long a refresh address counter do we need?

5.7 Figure 5.21 shows one of the early SRAMs, the 16 × 4 Signetics 7489 chip, which stores 16 4-bit words.

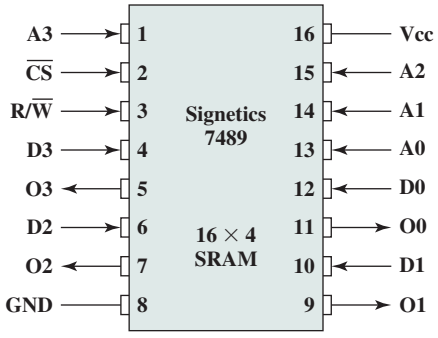
a. List the mode of operation of the chip for each $\overline{CS}$ input pulse shown in Figure 5.21c.

b. List the memory contents of word locations 0 through 6 after pulse n.

c. What is the state of the output data leads for the input pulses h through m?

5.8 Design a 16-bit memory of total capacity 8192 bits using SRAM chips of size 64 × 1 bit. Give the array configuration of the chips on the memory board showing all required input and output signals for assigning this memory to the lowest address space. The design should allow for both byte and 16-bit word accesses.

5.9 A common unit of measure for failure rates of electronic components is the **Failure unIT** (FIT), expressed as a rate of failures per billion device hours. Another well known but less used measure is **mean time between failures (MTBF),** which is the average time of operation of a particular component until it fails. Consider a 1 MB memory of a 16-bit microprocessor with 256K × 1 DRAMs. Calculate its MTBF assuming 2000 FITS for each DRAM.

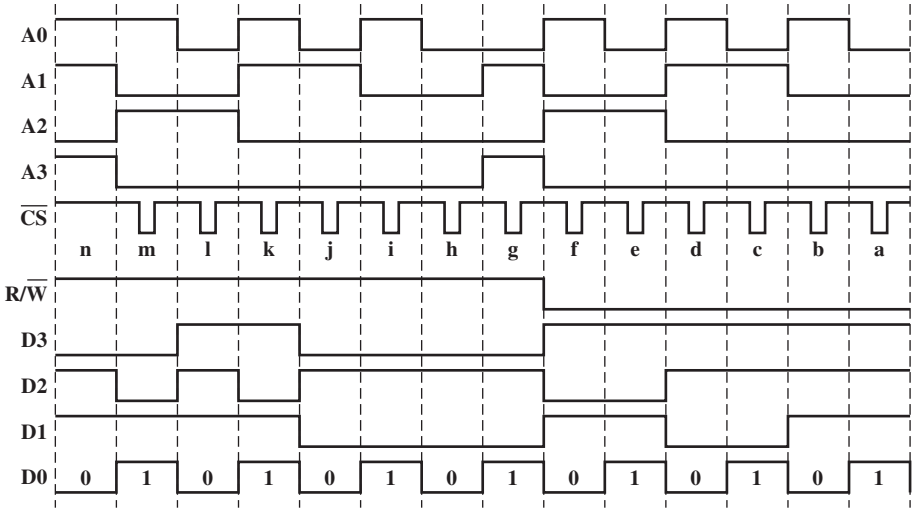


(a) Pin layout

Operating Mode	Inputs			Outputs
	$\overline{CS}$	R/$\overline{W}$	D*n*	O*n*
Write	L	L	L	L
	L	L	H	H
Read	L	H	X	Data
Inhibit writing	H	L	L	H
	H	L	H	L
Store - disable outputs	H	H	X	H

H = high voltage level
L = low voltage level
X = don't care

(b) Truth table



(c) Pulse train

Figure 5.21 The Signetics 7489 SRAM

5.10 For the Hamming code shown in Figure 5.10, show what happens when a check bit rather than a data bit is in error?

5.11 Suppose an 8-bit data word stored in memory is 11000010. Using the Hamming algorithm, determine what check bits would be stored in memory with the data word. Show how you got your answer.

5.12 For the 8-bit word 00111001, the check bits stored with it would be 0111. Suppose when the word is read from memory, the check bits are calculated to be 1101. What is the data word that was read from memory?

5.13 How many check bits are needed if the Hamming error correction code is used to detect single bit errors in a 1024-bit data word?

5.14 Develop an SEC code for a 16-bit data word. Generate the code for the data word 0101000000111001. Show that the code will correctly identify an error in data bit 5.

CHAPTER 6

External Memory

6.1 Magnetic Disk
- Magnetic Read and Write Mechanisms
- Data Organization and Formatting
- Physical Characteristics
- Disk Performance Parameters

6.2 RAID
- RAID Level 0
- RAID Level 1
- RAID Level 2
- RAID Level 3
- RAID Level 4
- RAID Level 5
- RAID Level 6

6.3 Solid State Drives
- SSD Compared to HDD
- SSD Organization
- Practical Issues

6.4 Optical Memory
- Compact Disk
- Digital Versatile Disk
- High-Definition Optical Disks

6.5 Magnetic Tape

6.6 Key Terms, Review Questions, and Problems

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the key properties of magnetic disks.
- Understand the performance issues involved in **magnetic disk** access.
- Explain the concept of **RAID** and describe the various levels.
- Compare and contrast hard disk drives and solid disk drives.
- Describe in general terms the operation of **flash memory**.
- Understand the differences among the different optical disk storage media.
- Present an overview of **magnetic tape** storage technology.

This chapter examines a range of external memory devices and systems. We begin with the most important device, the magnetic disk. Magnetic disks are the foundation of external memory on virtually all computer systems. The next section examines the use of disk arrays to achieve greater performance, looking specifically at the family of systems known as RAID (Redundant Array of Independent Disks). An increasingly important component of many computer systems is the solid state disk, which is discussed next. Then, external **optical memory** is examined. Finally, magnetic tape is described.

6.1 MAGNETIC DISK

A disk is a circular **platter** constructed of nonmagnetic material, called the **substrate**, coated with a magnetizable material. Traditionally, the substrate has been an aluminum or aluminum alloy material. More recently, glass substrates have been introduced. The glass substrate has a number of benefits, including the following:

- Improvement in the uniformity of the magnetic film surface to increase disk reliability.
- A significant reduction in overall surface defects to help reduce read-write errors.
- Ability to support lower fly heights (described subsequently).
- Better stiffness to reduce disk dynamics.
- Greater ability to withstand shock and damage.

Magnetic Read and Write Mechanisms

Data are recorded on and later retrieved from the disk via a conducting coil named the **head**; in many systems, there are two heads, a read head and a write head. During a read or write operation, the head is stationary while the platter rotates beneath it.

The write mechanism exploits the fact that electricity flowing through a coil produces a magnetic field. Electric pulses are sent to the write head, and the resulting magnetic patterns are recorded on the surface below, with different patterns for positive and negative currents. The write head itself is made of easily magnetizable

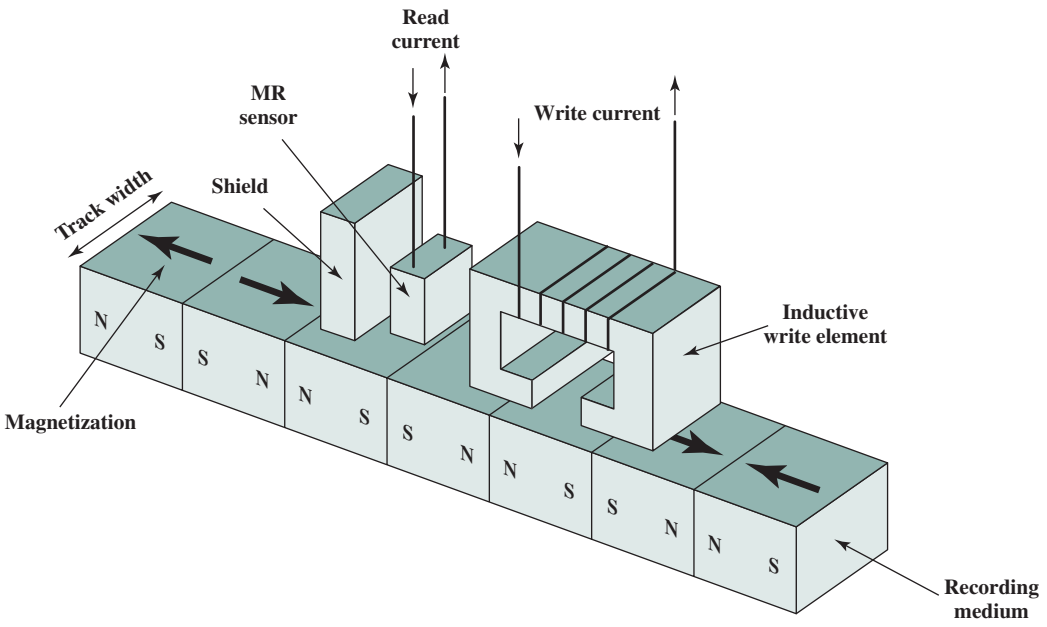


Figure 6.1 Inductive Write/Magnetoresistive Read Head

material and is in the shape of a rectangular doughnut with a gap along one side and a few turns of conducting wire along the opposite side (Figure 6.1). An electric current in the wire induces a magnetic field across the gap, which in turn magnetizes a small area of the recording medium. Reversing the direction of the current reverses the direction of the magnetization on the recording medium.

The traditional read mechanism exploits the fact that a magnetic field moving relative to a coil produces an electrical current in the coil. When the surface of the disk rotates under the head, it generates a current of the same polarity as the one already recorded. The structure of the head for reading is in this case essentially the same as for writing and therefore the same head can be used for both. Such single heads are used in floppy disk systems and in older rigid disk systems.

Contemporary rigid disk systems use a different read mechanism, requiring a separate read head, positioned for convenience close to the write head. The read head consists of a partially shielded **magnetoresistive (MR)** sensor. The MR material has an electrical resistance that depends on the direction of the magnetization of the medium moving under it. By passing a current through the MR sensor, resistance changes are detected as voltage signals. The MR design allows higher-frequency operation, which equates to greater storage densities and operating speeds.

Data Organization and Formatting

The head is a relatively small device capable of reading from or writing to a portion of the platter rotating beneath it. This gives rise to the organization of data on the platter in a concentric set of rings, called **tracks**. Each track is the same width as the head. There are thousands of tracks per surface.

Figure 6.2 depicts this data layout. Adjacent tracks are separated by **intertrack gaps**. This prevents, or at least minimizes, errors due to misalignment of the head or simply interference of magnetic fields. Data are transferred to and from the disk in **sectors**. There are typically hundreds of sectors per track, and these may be of either fixed or variable length. In most contemporary systems, fixed-length sectors are used, with 512 bytes being the nearly universal sector size. To avoid imposing unreasonable precision requirements on the system, adjacent sectors are separated by intersector gaps.

A bit near the center of a rotating disk travels past a fixed point (such as a read–write head) slower than a bit on the outside. Therefore, some way must be found to compensate for the variation in speed so that the head can read all the bits at the same rate. This can be done by defining a variable spacing between bits of information recorded in

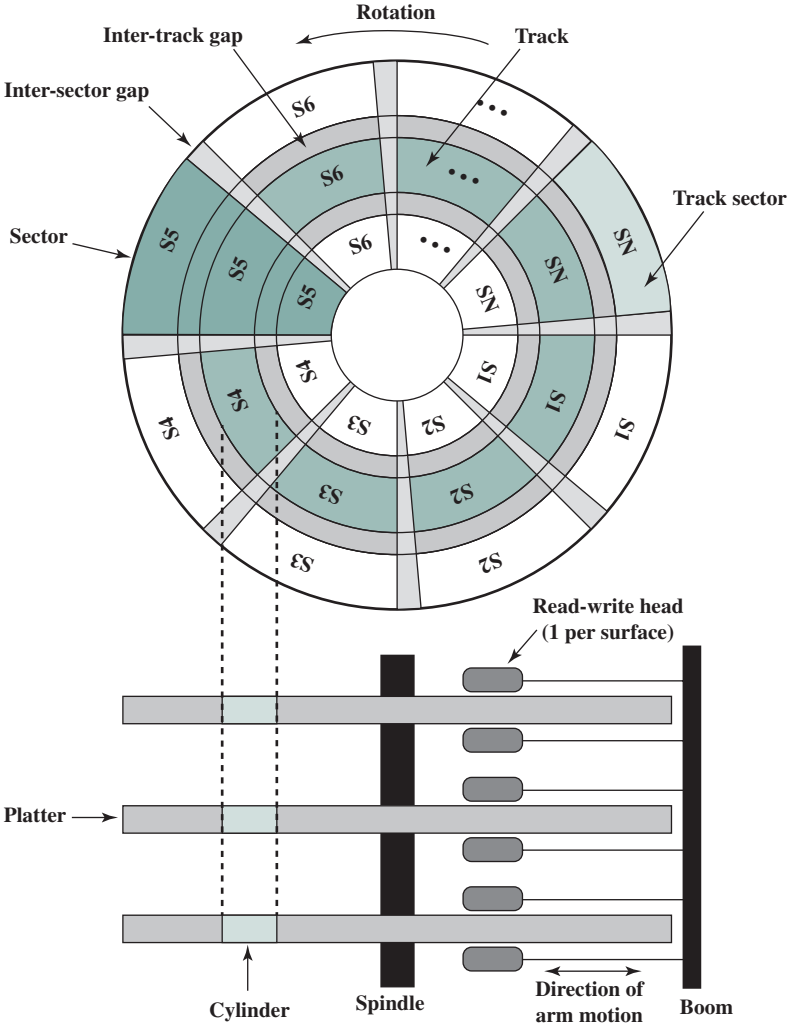
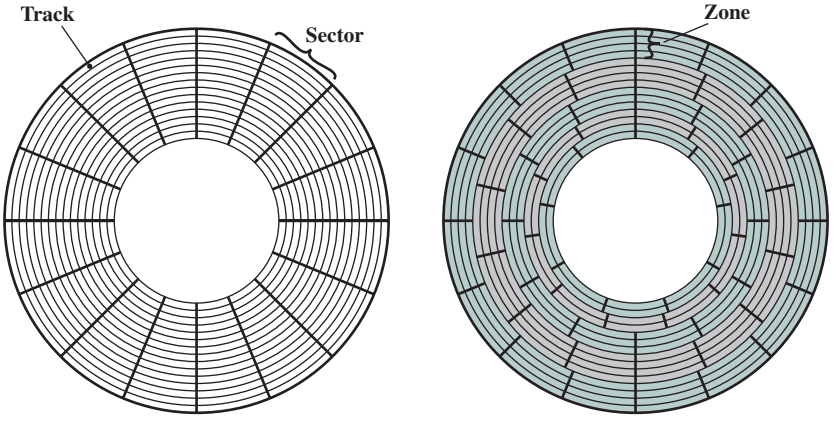


Figure 6.2 Disk Data Layout

locations on the disk, in a way that the outermost tracks has sectors with bigger spacing. The information can then be scanned at the same rate by rotating the disk at a fixed speed, known as the **constant angular velocity (CAV)**. Figure 6.3a shows the layout of a disk using CAV. The disk is divided into a number of pie-shaped sectors and into a series of concentric tracks. The advantage of using CAV is that individual blocks of data can be directly addressed by track and sector. To move the head from its current location to a specific address, it only takes a short movement of the head to a specific track and a short wait for the proper sector to spin under the head. The disadvantage of CAV is that the amount of data that can be stored on the long outer tracks is the only same as what can be stored on the short inner tracks.

Because the **density**, in bits per linear inch, increases in moving from the outermost track to the innermost track, disk storage capacity in a straightforward CAV system is limited by the maximum recording density that can be achieved on the innermost track. To maximize storage capacity, it would be preferable to have the same linear bit density on each track. This would require unacceptably complex circuitry. Modern hard disk systems use simpler technique, which approximates equal bit density per track, known as multiple zone recording (MZR), in which the surface is divided into a number of concentric zones (16 is typical). Each zone contains a number of contiguous tracks, typically in the thousands. Within a zone, the number of bits per track is constant. Zones farther from the center contain more bits (more sectors) than zones closer to the center. Zones are defined in such a way that the linear bit density is approximately the same on all tracks of the disk. MZR allows for greater overall storage capacity at the expense of somewhat more complex circuitry. As the disk head moves from one zone to another, the length (along the track) of individual bits changes, causing a change in the timing for reads and writes.

Figure 6.3b is a simplified MZR layout, with 15 tracks organized into 5 zones. The innermost two zones have two tracks each, with each track having nine sectors; the next zone has 3 tracks, each with 12 sectors; and the outermost 2 zones have 4 tracks each, with each track having 16 sectors.



(a) Constant angular velocity (b) Multiple zone recording

Figure 6.3 Comparison of Disk Layout Methods

Some means is needed to locate sector positions within a track. Clearly, there must be some starting point on the track and a way of identifying the start and end of each sector. These requirements are handled by means of control data recorded on the disk. Thus, the disk is formatted with some extra data used only by the disk drive and not accessible to the user.

An example of disk formatting is shown in Figure 6.4. In this case, each track contains 30 fixed-length sectors of 600 bytes each. Each sector holds 512 bytes of data plus control information useful to the disk controller. The ID field is a unique identifier or address used to locate a particular sector. The SYNCH byte is a special bit pattern that delimits the beginning of the field. The track number identifies a track on a surface. The head number identifies a head, because this disk has multiple surfaces (explained presently). The ID and data fields each contain an error-detecting code.

Physical Characteristics

Table 6.1 lists the major characteristics that differentiate among the various types of magnetic disks. First, the head may either be fixed or movable with respect to the radial direction of the platter. In a **fixed-head disk**, there is one read-write head per track. All of the heads are mounted on a rigid arm that extends across all tracks; such systems are rare today. In a **movable-head disk**, there is only one read-write head. Again, the head is mounted on an arm. Because the head must be able to be positioned above any track, the arm can be extended or retracted for this purpose.

The disk itself is mounted in a disk drive, which consists of the arm, a spindle that rotates the disk, and the electronics needed for input and output of binary data. A **nonremovable disk** is permanently mounted in the disk drive; the hard disk in a personal computer is a nonremovable disk. A **removable disk** can be removed and replaced with another disk. The advantage of the latter type is that unlimited amounts of data are available with a limited number of disk systems. Furthermore, such a disk may be moved from one computer system to another. Floppy disks and ZIP cartridge disks are examples of removable disks.

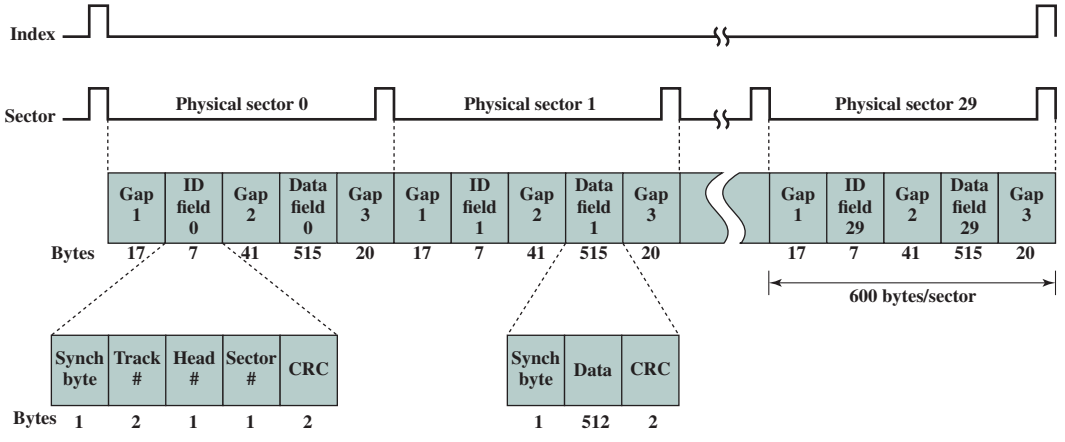


Figure 6.4 Winchester Disk Format (Seagate ST506)

Table 6.1 Physical Characteristics of Disk Systems

Head Motion	**Platters**
Fixed head (one per track)	Single platter
Movable head (one per surface)	Multiple platter
Disk Portability	**Head Mechanism**
Nonremovable disk	Contact (floppy)
Removable disk	Fixed gap
Sides	Aerodynamic gap (Winchester)
Single sided	
Double sided	

For most disks, the magnetizable coating is applied to both sides of the platter, which is then referred to as **double sided**. Some less expensive disk systems use **single-sided** disks.

Some disk drives accommodate **multiple platters** stacked vertically a fraction of an inch apart. Multiple arms are provided (Figure 6.2). Multiple–platter disks employ a movable head, with one read-write head per platter surface. All of the heads are mechanically fixed so that all are at the same distance from the center of the disk and move together. Thus, at any time, all of the heads are positioned over tracks that are of equal distance from the center of the disk. The set of all the tracks in the same relative position on the platter is referred to as a **cylinder**. This is illustrated in Figure 6.2.

Finally, the head mechanism provides a classification of disks into three types. Traditionally, the read-write head has been positioned a fixed distance above the platter, allowing an air gap. At the other extreme is a head mechanism that actually comes into physical contact with the medium during a read or write operation. This mechanism is used with the **floppy disk**, which is a small, flexible platter and the least expensive type of disk.

To understand the third type of disk, we need to comment on the relationship between data density and the size of the air gap. The head must generate or sense an electromagnetic field of sufficient magnitude to write and read properly. The narrower the head is, the closer it must be to the platter surface to function. A narrower head means narrower tracks and therefore greater data density, which is desirable. However, the closer the head is to the disk, the greater the risk of error from impurities or imperfections. To push the technology further, the Winchester disk was developed. Winchester heads are used in sealed drive assemblies that are almost free of contaminants. They are designed to operate closer to the disk's surface than conventional rigid disk heads, thus allowing greater data density. The head is actually an aerodynamic foil that rests lightly on the platter's surface when the disk is motionless. The air pressure generated by a spinning disk is enough to make the foil rise above the surface. The resulting noncontact system can be engineered to use narrower heads that operate closer to the platter's surface than conventional rigid disk heads.

Table 6.2 gives disk parameters for typical contemporary high-performance disks.

Table 6.2 Typical Hard Disk Drive Parameters

Characteristics	Seagate Enterprise	Seagate Barracuda XT	Seagate Cheetah NS	Seagate Laptop HDD
Application	Enterprise	Desktop	Network-attached storage, application servers	Laptop
Capacity	6 TB	3 TB	600 GB	2 TB
Average seek time	4.16 ms	N/A	3.9 ms read 4.2 ms write	13 ms
Spindle speed	7200 rpm	7200 rpm	10,075 rpm	5400 rpm
Average latency	4.16 ms	4.16 ms	2.98	5.6 ms
Maximum sustained transfer rate	216 MB/sec	149 MB/sec	97 MB/sec	300 MB/sec
Bytes per sector	512/4096	512	512	4096
Tracks per cylinder (number of platter surfaces)	8	10	8	4
Cache	128 MB	64 MB	16 MB	8 MB

Disk Performance Parameters

The actual details of disk I/O operation depend on the computer system, the operating system, and the nature of the I/O channel and disk controller hardware. A general timing diagram of disk I/O transfer is shown in Figure 6.5.

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time**. In either case, once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay**, or *rotational latency*. The sum of the seek time, if any, and the rotational delay equals the **access time**, which is the time it takes to get into position to read or write. Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time**.

In addition to the access time and transfer time, there are several queuing delays normally associated with a disk I/O operation. When a process issues an I/O

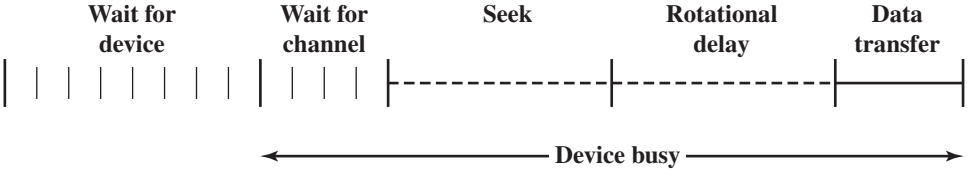


Figure 6.5 Timing of a Disk I/O Transfer

request, it must first wait in a queue for the device to be available. At that time, the device is assigned to the process. If the device shares a single I/O channel or a set of I/O channels with other disk drives, then there may be an additional wait for the channel to be available. At that point, the seek is performed to begin disk access.

In some high-end systems for servers, a technique known as rotational positional sensing (RPS) is used. This works as follows: When the seek command has been issued, the channel is released to handle other I/O operations. When the seek is completed, the device determines when the data will rotate under the head. As that sector approaches the head, the device tries to reestablish the communication path back to the host. If either the control unit or the channel is busy with another I/O, then the reconnection attempt fails and the device must rotate one whole revolution before it can attempt to reconnect, which is called an RPS miss. This is an extra delay element that must be added to the timeline of Figure 6.5.

SEEK TIME Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time, and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed. Unfortunately, the traversal time is not a linear function of the number of tracks, but includes a settling time (time after positioning the head over the target track until track identification is confirmed).

Much improvement comes from smaller and lighter disk components. Some years ago, a typical disk was 14 inches (36 cm) in diameter, whereas the most common size today is 3.5 inches (8.9 cm), reducing the distance that the arm has to travel. A typical average seek time on contemporary hard disks is under 10 ms.

ROTATIONAL DELAY Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm (for handheld devices such as digital cameras) up to, as of this writing, 20,000 rpm; at this latter speed, there is one revolution per 3 ms. Thus, on the average, the rotational delay will be 1.5 ms.

TRANSFER TIME The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

$$T = \frac{b}{rN}$$

where

T = transfer time
b = number of bytes to be transferred
N = number of bytes on a track
r = rotation speed, in revolutions per second

Thus the total average read or write time T_{total} can be expressed as

$$T_{total} = T_s + \frac{1}{2r} + \frac{b}{rN} \tag{6.1}$$

where T_s is the average seek time. Note that on a zoned drive, the number of bytes per track is variable, complicating the calculation.[1]

[1]Compare the two preceding equations to Equation (4.1).

A TIMING COMPARISON With the foregoing parameters defined, let us look at two different I/O operations that illustrate the danger of relying on average values. Consider a disk with an advertised average seek time of 4 ms, rotation speed of 15,000 rpm, and 512-byte sectors with 500 sectors per track. Suppose that we wish to read a file consisting of 2500 sectors for a total of 1.28 Mbytes. We would like to estimate the total time for the transfer.

First, let us assume that the file is stored as compactly as possible on the disk. That is, the file occupies all of the sectors on 5 adjacent tracks (5 tracks $\times$ 500 sectors/track $=$ 2500 sectors). This is known as *sequential organization*. Now, the time to read the first track is as follows:

Average seek	4 ms
Average rotational delay	2 ms
Read 500 sectors	4 ms
	10 ms

Suppose that the remaining tracks can now be read with essentially no seek time. That is, the I/O operation can keep up with the flow from the disk. Then, at most, we need to deal with rotational delay for the four remaining tracks. Thus each successive track is read in $2 + 4 = 6$ ms. To read the entire file,

$$\text{Total time} = 10 + (4 \times 6) = 34 \text{ ms} = 0.034 \text{ seconds}$$

Now let us calculate the time required to read the same data using random access rather than sequential access; that is, accesses to the sectors are distributed randomly over the disk. For each sector, we have

Average seek	4 ms
Rotational delay	2 ms
Read 1 sectors	0.008 ms
	6.008 ms

$$\text{Total time} = 2500 \times 6.008 = 15{,}020 \text{ ms} = 15.02 \text{ seconds}$$

It is clear that the order in which sectors are read from the disk has a tremendous effect on I/O performance. In the case of file access in which multiple sectors are read or written, we have some control over the way in which sectors of data are deployed. However, even in the case of a file access, in a multiprogramming environment, there will be I/O requests competing for the same disk. Thus, it is worthwhile to examine ways in which the performance of disk I/O can be improved over that achieved with purely random access to the disk. This leads to a consideration of disk scheduling algorithms, which is the province of the operating system and beyond the scope of this book (see [STAL15] for a discussion).

RAID Simulator

6.2 RAID

As discussed earlier, the rate in improvement in secondary storage performance has been considerably less than the rate for processors and main memory. This mismatch has made the disk storage system perhaps the main focus of concern in improving overall computer system performance.

As in other areas of computer performance, disk storage designers recognize that if one component can only be pushed so far, additional gains in performance are to be had by using multiple parallel components. In the case of disk storage, this leads to the development of arrays of disks that operate independently and in parallel. With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks. Further, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.

With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. This could make it difficult to develop database schemes that are usable on a number of platforms and operating systems. Fortunately, industry has agreed on a standardized scheme for multiple-disk database design, known as RAID (Redundant Array of Independent Disks). The RAID scheme consists of seven levels,[2] zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
2. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.
3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

The term *RAID* was originally coined in a paper by a group of researchers at the University of California at Berkeley [PATT88].[3] The paper outlined various RAID configurations and applications and introduced the definitions of the RAID levels that are still used. The RAID strategy employs multiple disk drives and distributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental increases in capacity.

---

[2]Additional levels have been defined by some researchers and some companies, but the seven levels described in this section are the ones universally agreed on.

[3]In that paper, the acronym RAID stood for Redundant Array of Inexpensive Disks. The term *inexpensive* was used to contrast the small relatively inexpensive disks in the RAID array to the alternative, a single large expensive disk (SLED). The SLED is essentially a thing of the past, with similar disk technology being used for both RAID and non-RAID configurations. Accordingly, the industry has adopted the term *independent* to emphasize that the RAID array creates significant performance and reliability gains.

The unique contribution of the RAID proposal is to address effectively the need for redundancy. Although allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, the use of multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

We now examine each of the RAID levels. Table 6.3 provides a rough guide to the seven levels. In the table, I/O performance is shown both in terms of data transfer capacity, or ability to move data, and I/O request rate, or ability to satisfy I/O requests, since these RAID levels inherently perform differently relative to these two metrics. Each RAID level's strong point is highlighted by darker shading. Figure 6.6 illustrates the use of the seven RAID schemes to support a data capacity requiring four disks with no redundancy. The figures highlight the layout of user data and redundant data and indicates the relative storage requirements of the various levels. We refer to these figures throughout the following discussion. Of the seven RAID levels described, only four are commonly used: RAID levels 0, 1, 5, and 6.

RAID Level 0

RAID level 0 is not a true member of the RAID family because it does not include redundancy to improve performance. However, there are a few applications, such as some on supercomputers in which performance and capacity are primary concerns and low cost is more important than improved reliability.

For RAID 0, the user and system data are distributed across all of the disks in the array. This has a notable advantage over the use of a single large disk: If two-different I/O requests are pending for two different blocks of data, then there is a good chance that the requested blocks are on different disks. Thus, the two requests can be issued in parallel, reducing the I/O queuing time.

But RAID 0, as with all of the RAID levels, goes further than simply distributing the data across a disk array: The data are *striped* across the available disks. This is best understood by considering Figure 6.7. All of the user and system data are viewed as being stored on a logical disk. The logical disk is divided into strips; these strips may be physical blocks, sectors, or some other unit. The strips are mapped round robin to consecutive physical disks in the RAID array. A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a **stripe.** In an n-disk array, the first n logical strips are physically stored as the first strip on each of the n disks, forming the first stripe; the second n strips are distributed as the second strips on each disk; and so on. The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to n strips for that request can be handled in parallel, greatly reducing the I/O transfer time.

Figure 6.7 indicates the use of array management software to map between logical and physical disk space. This software may execute either in the disk subsystem or in a host computer.

RAID 0 FOR HIGH DATA TRANSFER CAPACITY The performance of any of the RAID levels depends critically on the request patterns of the host system and on the layout of the data. These issues can be most clearly addressed in RAID 0, where the

Table 6.3 RAID Levels

Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

Note: $N =$ number of data disks; m proportional to $\log N$

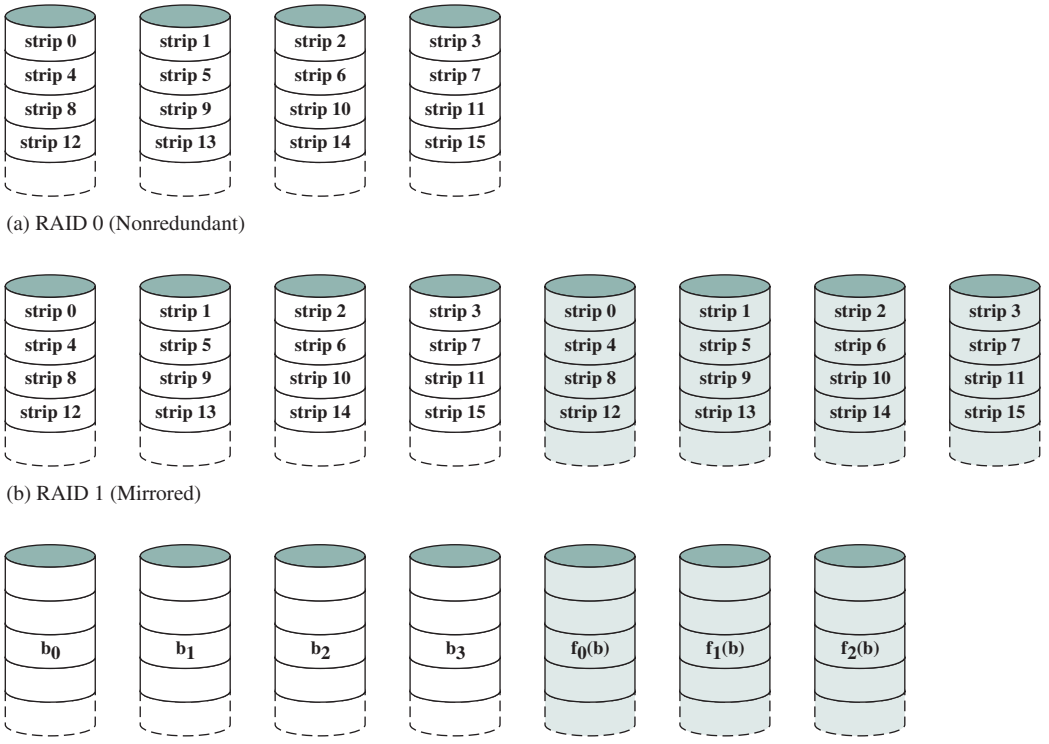


Figure 6.6 RAID Levels (*Continued*)

impact of redundancy does not interfere with the analysis. First, let us consider the use of RAID 0 to achieve a high data transfer rate. For applications to experience a high transfer rate, two requirements must be met. First, a high transfer capacity must exist along the entire path between host memory and the individual disk drives. This includes internal controller buses, host system I/O buses, I/O adapters, and host memory buses.

The second requirement is that the application must make I/O requests that drive the disk array efficiently. This requirement is met if the typical request is for large amounts of logically contiguous data, compared to the size of a strip. In this case, a single I/O request involves the parallel transfer of data from multiple disks, increasing the effective transfer rate compared to a single-disk transfer.

RAID 0 FOR HIGH I/O REQUEST RATE In a transaction-oriented environment, the user is typically more concerned with response time than with transfer rate. For an individual I/O request for a small amount of data, the I/O time is dominated by the motion of the disk heads (seek time) and the movement of the disk (rotational latency).

In a transaction environment, there may be hundreds of I/O requests per second. A disk array can provide high I/O execution rates by balancing the I/O load across multiple disks. Effective load balancing is achieved only if there are typically

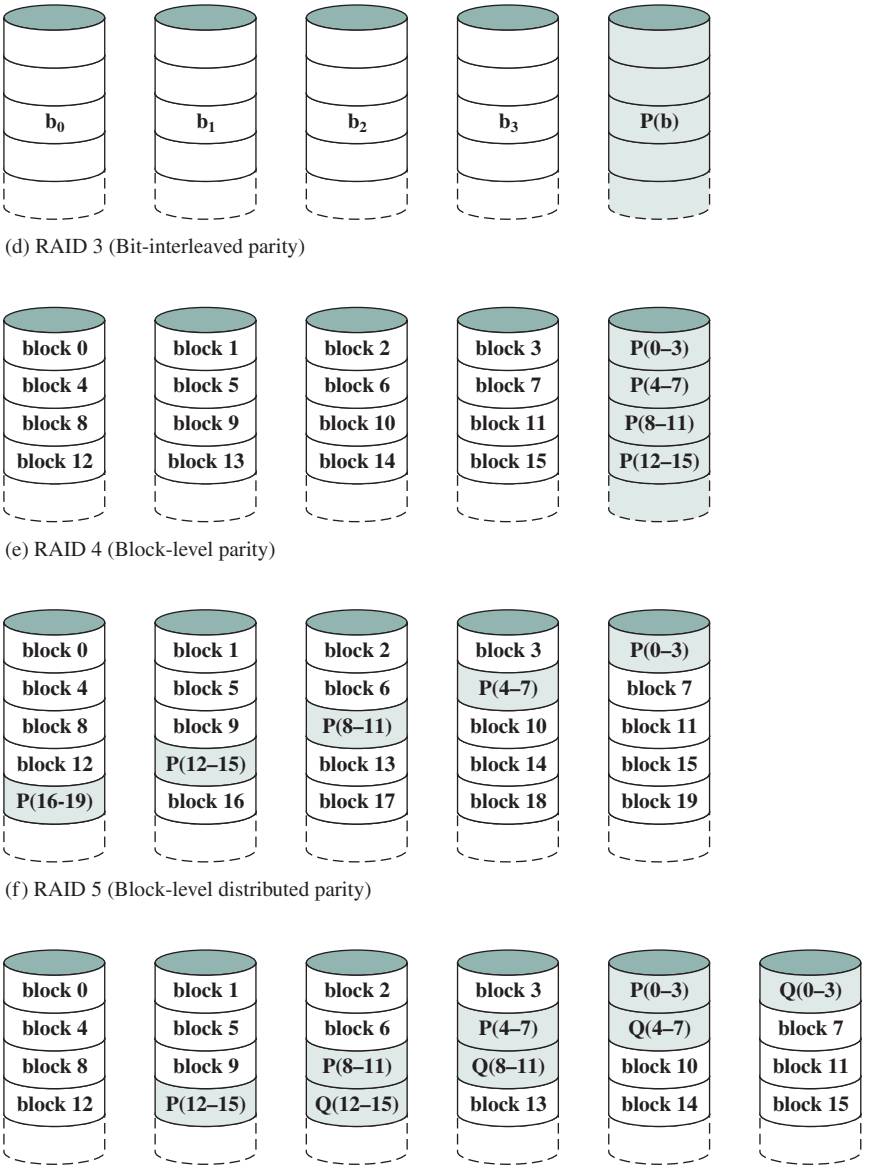


Figure 6.6 RAID Levels (*Continued*)

multiple I/O requests outstanding. This, in turn, implies that there are multiple independent applications or a single transaction-oriented application that is capable of multiple asynchronous I/O requests. The performance will also be influenced by the strip size. If the strip size is relatively large, so that a single I/O request only involves a single disk access, then multiple waiting I/O requests can be handled in parallel, reducing the queuing time for each request.

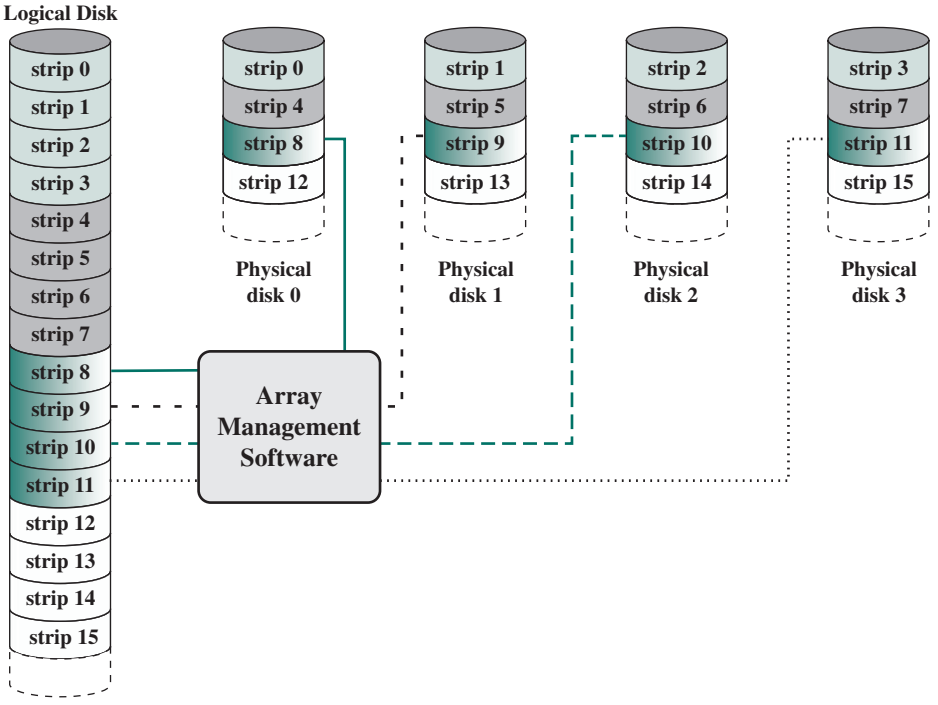


Figure 6.7 Data Mapping for a RAID Level 0 Array

RAID Level 1

RAID 1 differs from RAID levels 2 through 6 in the way in which redundancy is achieved. In these other RAID schemes, some form of parity calculation is used to introduce redundancy, whereas in RAID 1, redundancy is achieved by the simple expedient of duplicating all the data. As Figure 6.6b shows, data striping is used, as in RAID 0. But in this case, each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data. RAID 1 can also be implemented without data striping, though this is less common.

There are a number of positive aspects to the RAID 1 organization:

1. A read request can be serviced by either of the two disks that contains the requested data, whichever one involves the minimum seek time plus rotational latency.
2. A write request requires that both corresponding strips be updated, but this can be done in parallel. Thus, the write performance is dictated by the slower of the two writes (i.e., the one that involves the larger seek time plus rotational latency). However, there is no "write penalty" with RAID 1. RAID levels 2 through 6 involve the use of parity bits. Therefore, when a single strip is updated, the array management software must first compute and update the parity bits as well as updating the actual strip in question.
3. Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive.

The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the logical disk that it supports. Because of that, a RAID 1 configuration is likely to be limited to drives that store system software and data and other highly critical files. In these cases, RAID 1 provides real-time copy of all data so that in the event of a disk failure, all of the critical data are still immediately available.

In a transaction-oriented environment, RAID 1 can achieve high I/O request rates if the bulk of the requests are reads. In this situation, the performance of RAID 1 can approach double of that of RAID 0. However, if a substantial fraction of the I/O requests are write requests, then there may be no significant performance gain over RAID 0. RAID 1 may also provide improved performance over RAID 0 for data transfer intensive applications with a high percentage of reads. Improvement occurs if the application can split each read request so that both disk members participate.

RAID Level 2

RAID levels 2 and 3 make use of a parallel access technique. In a parallel access array, all member disks participate in the execution of every I/O request. Typically, the spindles of the individual drives are synchronized so that each disk head is in the same position on each disk at any given time.

As in the other RAID schemes, data striping is used. In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word. With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks. Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.

Although RAID 2 requires fewer disks than RAID 1, it is still rather costly. The number of redundant disks is proportional to the log of the number of data disks. On a single read, all disks are simultaneously accessed. The requested data and the associated error-correcting code are delivered to the array controller. If there is a single-bit error, the controller can recognize and correct the error instantly, so that the read access time is not slowed. On a single write, all data disks and parity disks must be accessed for the write operation.

RAID 2 would only be an effective choice in an environment in which many disk errors occur. Given the high reliability of individual disks and disk drives, RAID 2 is overkill and is not implemented.

RAID Level 3

RAID 3 is organized in a similar fashion to RAID 2. The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array. RAID 3 employs parallel access, with data distributed in small strips. Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

REDUNDANCY In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive and operation resumed.

Data reconstruction is simple. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. The parity for the *i*th bit is calculated as follows:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

where $\oplus$ is exclusive-OR function.

Suppose that drive X1 has failed. If we add $X4(i) \oplus X1(i)$ to both sides of the preceding equation, we get

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Thus, the contents of each strip of data on X1 can be regenerated from the contents of the corresponding strips on the remaining disks in the array. This principle is true for RAID levels 3 through 6.

In the event of a disk failure, all of the data are still available in what is referred to as reduced mode. In this mode, for reads, the missing data are regenerated on the fly using the exclusive-OR calculation. When data are written to a reduced RAID 3 array, consistency of the parity must be maintained for later regeneration. Return to full operation requires that the failed disk be replaced and the entire contents of the failed disk be regenerated on the new disk.

PERFORMANCE Because data are striped in very small strips, RAID 3 can achieve very high data transfer rates. Any I/O request will involve the parallel transfer of data from all of the data disks. For large transfers, the performance improvement is especially noticeable. On the other hand, only one I/O request can be executed at a time. Thus, in a transaction-oriented environment, performance suffers.

RAID Level 4

RAID levels 4 through 6 make use of an independent access technique. In an independent access array, each member disk operates independently, so that separate I/O requests can be satisfied in parallel. Because of this, independent access arrays are more suitable for applications that require high I/O request rates and are relatively less suited for applications that require high data transfer rates.

As in the other RAID schemes, data striping is used. In the case of RAID 4 through 6, the strips are relatively large. With RAID 4, a bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk.

RAID 4 involves a write penalty when an I/O write request of small size is performed. Each time that a write occurs, the array management software must update not only the user data but also the corresponding parity bits. Consider an array of five drives in which X0 through X3 contain data and X4 is the parity disk. Suppose that a write is performed that only involves a strip on disk X1. Initially, for each bit *i*, we have the following relationship:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \qquad \textbf{(6.2)}$$

After the update, with potentially altered bits indicated by a prime symbol:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i)X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(\mathrm{i}) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(\mathrm{i}) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

The preceding set of equations is derived as follows. The first line shows that a change in X1 will also affect the parity disk X4. In the second line, we add the terms $\oplus$ X1(i) $\oplus$ X1(i)]. Because the exclusive-OR of any quantity with itself is 0, this does not affect the equation. However, it is a convenience that is used to create the third line, by reordering. Finally, Equation (6.2) is used to replace the first four terms by X4(i).

To calculate the new parity, the array management software must read the old user strip and the old parity strip. Then it can update these two strips with the new data and the newly calculated parity. Thus, each strip write involves two reads and two writes.

In the case of a larger size I/O write that involves strips on all disk drives, parity is easily computed by calculation using only the new data bits. Thus, the parity drive can be updated in parallel with the data drives and there are no extra reads or writes.

In any case, every write operation must involve the parity disk, which therefore can become a bottleneck.

RAID Level 5

RAID 5 is organized in a similar fashion to RAID 4. The difference is that RAID 5 distributes the parity strips across all disks. A typical allocation is a round-robin scheme, as illustrated in Figure 6.6f. For an n-disk array, the parity strip is on a different disk for the first n stripes, and the pattern then repeats.

The distribution of parity strips across all drives avoids the potential I/O bottle-neck found in RAID 4.

RAID Level 6

RAID 6 was introduced in a subsequent paper by the Berkeley researchers [KATZ89]. In the RAID 6 scheme, two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require N disks consists of $N + 2$ disks.

Figure 6.6g illustrates the scheme. P and Q are two different data check algorithms. One of the two is the exclusive-OR calculation used in RAID 4 and 5. But the other is an independent data check algorithm. This makes it possible to regenerate data even if two disks containing user data fail.

The advantage of RAID 6 is that it provides extremely high data availability. Three disks would have to fail within the MTTR (mean time to repair) interval to cause data to be lost. On the other hand, RAID 6 incurs a substantial write penalty, because each write affects two parity blocks. Performance benchmarks [EISC07] show a RAID 6 controller can suffer more than a 30% drop in overall write performance compared with a RAID 5 implementation. RAID 5 and RAID 6 read performance is comparable.

Table 6.4 is a comparative summary of the seven levels.

6.3 SOLID STATE DRIVES

One of the most significant developments in computer architecture in recent years is the increasing use of solid state drives (SSDs) to complement or even replace **hard disk drives (HDDs)**, both as internal and external secondary memory. The term *solid*

Table 6.4 RAID Comparison

Level	Advantages	Disadvantages	Applications
0	I/O performance is greatly improved by spreading the I/O load across many channels and drives No parity calculation overhead is involved Very simple design Easy to implement	The failure of just one drive will result in all data in an array being lost	Video production and editing Image Editing Pre-press applications Any application requiring high bandwidth
1	100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures Simplest RAID storage subsystem design	Highest disk overhead of all RAID types (100%)—inefficient	Accounting Payroll Financial Any application requiring very high availability
2	Extremely high data transfer rates possible The higher the data transfer rate required, the better the ratio of data disks to ECC disks Relatively simple controller design compared to RAID levels 3, 4, & 5	Very high ratio of ECC disks to data disks with smaller word sizes—inefficient Entry level cost very high—requires very high transfer rate requirement to justify	No commercial implementations exist/not commercially viable
3	Very high read data transfer rate Very high write data transfer rate Disk failure has an insignificant impact on throughput Low ratio of ECC (parity) disks to data disks means high efficiency	Transaction rate equal to that of a single disk drive at best (if spindles are synchronized) Controller design is fairly complex	Video production and live streaming Image editing Video editing Prepress applications Any application requiring high throughput
4	Very high Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency	Quite complex controller design Worst write transaction rate and Write aggregate transfer rate Difficult and inefficient data rebuild in the event of disk failure	No commercial implementations exist/not commercially viable
5	Highest Read data transaction rate Low ratio of ECC (parity) disks to data disks means high efficiency Good aggregate transfer rate	Most complex controller design Difficult to rebuild in the event of a disk failure (as compared to RAID level 1)	File and application servers Database servers Web, e-mail, and news servers Intranet servers Most versatile RAID level
6	Provides for an extremely high data fault tolerance and can sustain multiple simultaneous drive failures	More complex controller design Controller overhead to compute parity addresses is extremely high	Perfect solution for mission critical applications

state refers to electronic circuitry built with semiconductors. An SSD is a memory device made with solid state components that can be used as a replacement to a hard disk drive. The SSDs now on the market and coming on line use NAND flash memory, which is described in Chapter 5.

SSD Compared to HDD

As the cost of flash-based SSDs has dropped and the performance and bit density increased, SSDs have become increasingly competitive with HDDs. Table 6.5 shows typical measures of comparison at the time of this writing.

SSDs have the following advantages over HDDs:

- **High-performance input/output operations per second (IOPS):** Significantly increases performance I/O subsystems.
- **Durability:** Less susceptible to physical shock and vibration.
- **Longer lifespan:** SSDs are not susceptible to mechanical wear.
- **Lower power consumption:** SSDs use considerably less power than comparable-size HDDs.
- **Quieter and cooler running capabilities:** Less space required, lower energy costs, and a greener enterprise.
- **Lower access times and latency rates:** Over 10 times faster than the spinning disks in an HDD.

Currently, HDDs enjoy a cost per bit advantage and a capacity advantage, but these differences are shrinking.

SSD Organization

Figure 6.8 illustrates a general view of the common architectural system component associated with any SSD system. On the host system, the operating system invokes file system software to access data on the disk. The file system, in turn, invokes I/O driver software. The I/O driver software provides host access to the particular SSD product. The interface component in Figure 6.8 refers to the physical and electrical interface between the host processor and the SSD peripheral device. If the device is an internal hard drive, a common interface is PCIe. For external devices, one common interface is USB.

Table 6.5 Comparison of Solid State Drives and Disk Drives

	NAND Flash Drives	Seagate Laptop Internal HDD
File copy/write speed	200–550 Mbps	50–120 Mbps
Power draw/battery life	Less power draw, averages 2–3 watts, resulting in 30+ minute battery boost	More power draw, averages 6–7 watts and therefore uses more battery
Storage capacity	Typically not larger than 512 GB for notebook size drives; 1 TB max for desktops	Typically around 500 GB and 2 TB max for notebook size drives; 4 TB max for desktops
Cost	Approx. $0.50 per GB for a 1-TB drive	Approx. $0.15 per GB for a 4-TB drive

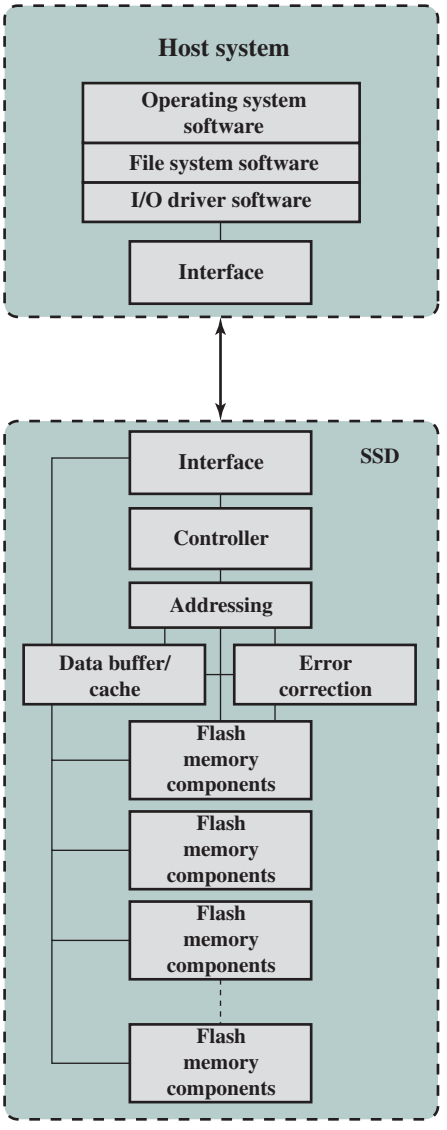


Figure 6.8 Solid State Drive Architecture

In addition to the interface to the host system, the SSD contains the following components:

- **Controller:** Provides SSD device level interfacing and firmware execution.
- **Addressing:** Logic that performs the selection function across the flash memory components.
- **Data buffer/cache:** High speed RAM memory components used for speed matching and to increased data throughput.

- **Error correction:** Logic for error detection and correction.
- **Flash memory components:** Individual NAND flash chips.

Practical Issues

There are two practical issues peculiar to SSDs that are not faced by HDDs. First, SSD performance has a tendency to slow down as the device is used. To understand the reason for this, you need to know that files are stored on disk as a set of pages, typically 4 KB in length. These pages are not necessarily, and indeed not typically, stored as a contiguous set of pages on the disk. The reason for this arrangement is explained in our discussion of virtual memory in Chapter 8. However, flash memory is accessed in blocks, with a typical block size of 512 KB, so that there are typically 128 pages per block. Now consider what must be done to write a page onto a flash memory.

1. The entire block must be read from the flash memory and placed in a RAM buffer. Then the appropriate page in the RAM buffer is updated.
2. Before the block can be written back to flash memory, the entire block of flash memory must be erased—it is not possible to erase just one page of the flash memory.
3. The entire block from the buffer is now written back to the flash memory.

Now, when a flash drive is relatively empty and a new file is created, the pages of that file are written on to the drive contiguously, so that one or only a few blocks are affected. However, over time, because of the way virtual memory works, files become fragmented, with pages scattered over multiple blocks. As the drive become more occupied, there is more fragmentation, so the writing of a new file can affect multiple blocks. Thus, the writing of multiple pages from one block becomes slower, the more fully occupied the disk is. Manufacturers have developed a variety of techniques to compensate for this property of flash memory, such as setting aside a substantial portion of the SSD as extra space for write operations (called overprovisioning), then to erase inactive pages during idle time used to defragment the disk. Another technique is the TRIM command, which allows an operating system to inform an SSD which blocks of data are no longer considered in use and can be wiped internally.[4]

A second practical issue with flash memory drives is that a flash memory becomes unusable after a certain number of writes. As flash cells are stressed, they lose their ability to record and retain values. A typical limit is 100,000 writes [GSOE08]. Techniques for prolonging the life of an SSD drive include front-ending the flash with a cache to delay and group write operations, using wear-leveling algorithms that evenly distribute writes across block of cells, and sophisticated bad-block management techniques. In addition, vendors are deploying SSDs in RAID configurations to further reduce the probability of data loss. Most flash devices are also capable of estimating their own remaining lifetimes so systems can anticipate failure and take preemptive action.

[4]While TRIM is frequently spelled in capital letters, it is not an acronym; it is merely a command name.

6.4 OPTICAL MEMORY

In 1983, one of the most successful consumer products of all time was introduced: the compact disk (CD) digital audio system. The CD is a nonerasable disk that can store more than 60 minutes of audio information on one side. The huge commercial success of the CD enabled the development of low-cost optical-disk storage technology that has revolutionized computer data storage. A variety of optical-disk systems have been introduced (Table 6.6). We briefly review each of these.

Compact Disk

CD–ROM Both the audio CD and the **CD-ROM** (compact disk read-only memory) share a similar technology. The main difference is that CD-ROM players are more rugged and have error correction devices to ensure that data are properly transferred from disk to computer. Both types of disk are made the same way. The disk is formed from a resin, such as polycarbonate. Digitally recorded information (either music or computer data) is imprinted as a series of microscopic pits on the surface of the polycarbonate. This is done, first of all, with a finely focused, high-intensity laser to create a master disk. The master is used, in turn, to make a die to stamp out copies onto polycarbonate. The pitted surface is then coated with a highly reflective surface, usually aluminum or gold. This shiny surface is protected against dust and scratches by a top coat of clear acrylic. Finally, a label can be silkscreened onto the acrylic.

Table 6.6 Optical Disk Products

CD
Compact Disk. A nonerasable disk that stores digitized audio information. The standard system uses 12-cm disks and can record more than 60 minutes of uninterrupted playing time.

CD-ROM
Compact Disk Read-Only Memory. A nonerasable disk used for storing computer data. The standard system uses 12-cm disks and can hold more than 650 Mbytes.

CD-R
CD Recordable. Similar to a CD-ROM. The user can write to the disk only once.

CD-RW
CD Rewritable. Similar to a CD-ROM. The user can erase and rewrite to the disk multiple times.

DVD
Digital Versatile Disk. A technology for producing digitized, compressed representation of video information, as well as large volumes of other digital data. Both 8 and 12 cm diameters are used, with a double-sided capacity of up to 17 Gbytes. The basic DVD is read-only (DVD-ROM).

DVD-R
DVD Recordable. Similar to a DVD-ROM. The user can write to the disk only once. Only one-sided disks can be used.

DVD-RW
DVD Rewritable. Similar to a DVD-ROM. The user can erase and rewrite to the disk multiple times. Only one-sided disks can be used.

Blu-ray DVD
High-definition video disk. Provides considerably greater data storage density than DVD, using a 405-nm (blue-violet) laser. A single layer on a single side can store 25 Gbytes.

Information is retrieved from a CD or CD-ROM by a low-powered laser housed in an optical-disk player, or drive unit. The laser shines through the clear polycarbonate while a motor spins the disk past it (Figure 6.9). The intensity of the reflected light of the laser changes as it encounters a **pit**. Specifically, if the laser beam falls on a pit, which has a somewhat rough surface, the light scatters and a low intensity is reflected back to the source. The areas between pits are called **lands**. A land is a smooth surface, which reflects back at higher intensity. The change between pits and lands is detected by a photosensor and converted into a digital signal. The sensor tests the surface at regular intervals. The beginning or end of a pit represents a 1; when no change in elevation occurs between intervals, a 0 is recorded.

Recall that on a magnetic disk, information is recorded in concentric tracks. With the simplest constant angular velocity (CAV) system, the number of bits per track is constant. An increase in density is achieved with **multiple zone recording**, in which the surface is divided into a number of zones, with zones farther from the center containing more bits than zones closer to the center. Although this technique increases capacity, it is still not optimal.

To achieve greater capacity, CDs and CD-ROMs do not organize information on concentric tracks. Instead, the disk contains a single spiral track, beginning near the center and spiraling out to the outer edge of the disk. Sectors near the outside of the disk are the same length as those near the inside. Thus, information is packed evenly across the disk in segments of the same size and these are scanned at the same rate by rotating the disk at a variable speed. The pits are then read by the laser at a **constant linear velocity (CLV)**. The disk rotates more slowly for accesses near the outer edge than for those near the center. Thus, the capacity of a track and the rotational delay both increase for positions nearer the outer edge of the disk. The data capacity for a CD-ROM is about 680 MB.

Data on the CD-ROM are organized as a sequence of blocks. A typical block format is shown in Figure 6.10. It consists of the following fields:

- **Sync:** The sync field identifies the beginning of a block. It consists of a byte of all 0s, 10 bytes of all 1s, and a byte of all 0s.
- **Header:** The header contains the block address and the mode byte. Mode 0 specifies a blank data field; mode 1 specifies the use of an error-correcting

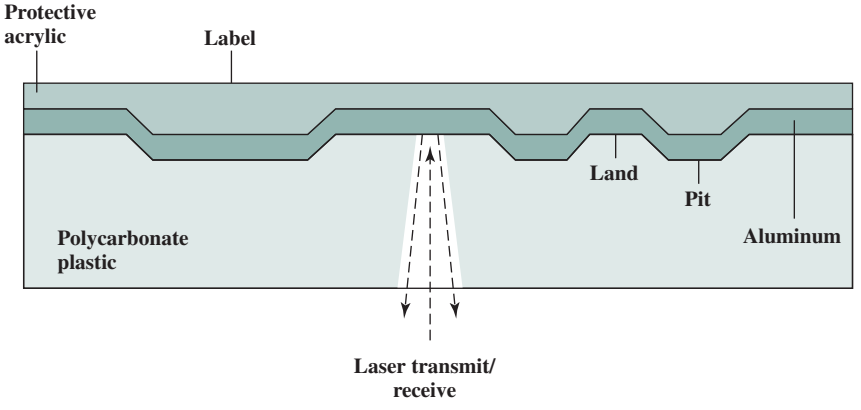


Figure 6.9 CD Operation

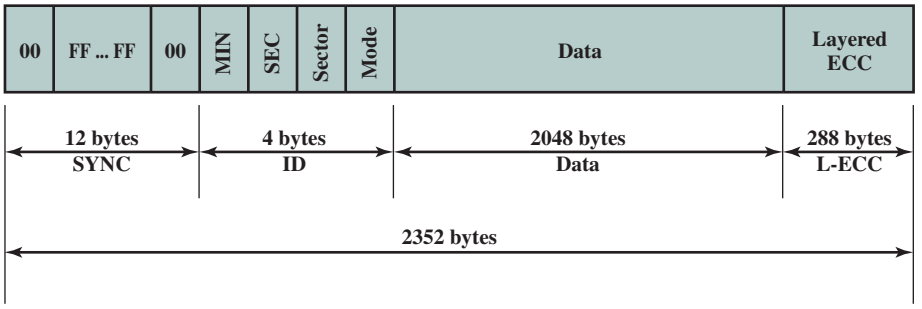


Figure 6.10 CD-ROM Block Format

code and 2048 bytes of data; mode 2 specifies 2336 bytes of user data with no error-correcting code.

- **Data:** User data.
- **Auxiliary:** Additional user data in mode 2. In mode 1, this is a 288-byte error-correcting code.

With the use of CLV, random access becomes more difficult. Locating a specific address involves moving the head to the general area, adjusting the rotation speed and reading the address, and then making minor adjustments to find and access the specific sector.

CD-ROM is appropriate for the distribution of large amounts of data to a large number of users. Because of the expense of the initial writing process, it is not appropriate for individualized applications. Compared with traditional magnetic disks, the CD-ROM has two advantages:

- The optical disk together with the information stored on it can be mass replicated inexpensively—unlike a magnetic disk. The database on a magnetic disk has to be reproduced by copying one disk at a time using two disk drives.
- The optical disk is removable, allowing the disk itself to be used for archival storage. Most magnetic disks are nonremovable. The information on nonremovable magnetic disks must first be copied to another storage medium before the disk drive/disk can be used to store new information.

The disadvantages of CD-ROM are as follows:

- It is read-only and cannot be updated.
- It has an access time much longer than that of a magnetic disk drive, as much as half a second.

CD RECORDABLE To accommodate applications in which only one or a small number of copies of a set of data is needed, the write-once read-many CD, known as the **CD recordable (CD-R),** has been developed. For CD-R, a disk is prepared in such a way that it can be subsequently written once with a laser beam of modest-intensity. Thus, with a somewhat more expensive disk controller than for CD-ROM, the customer can write once as well as read the disk.

The CD-R medium is similar to but not identical to that of a CD or CD-ROM. For CDs and CD-ROMs, information is recorded by the pitting of the surface

of the medium, which changes reflectivity. For a CD-R, the medium includes a dye layer. The dye is used to change reflectivity and is activated by a high-intensity laser. The resulting disk can be read on a CD-R drive or a CD-ROM drive.

The CD-R optical disk is attractive for archival storage of documents and files. It provides a permanent record of large volumes of user data.

CD REWRITABLE The **CD-RW** optical disk can be repeatedly written and overwritten, as with a magnetic disk. Although a number of approaches have been tried, the only pure optical approach that has proved attractive is called **phase change**. The phase change disk uses a material that has two significantly different reflectivities in two different phase states. There is an amorphous state, in which the molecules exhibit a random orientation that reflects light poorly; and a crystalline state, which has a smooth surface that reflects light well. A beam of laser light can change the material from one phase to the other. The primary disadvantage of phase change optical disks is that the material eventually and permanently loses its desirable properties. Current materials can be used for between 500,000 and 1,000,000 erase cycles.

The CD-RW has the obvious advantage over CD-ROM and CD-R that it can be rewritten and thus used as a true secondary storage. As such, it competes with magnetic disk. A key advantage of the optical disk is that the engineering tolerances for optical disks are much less severe than for high-capacity magnetic disks. Thus, they exhibit higher reliability and longer life.

Digital Versatile Disk

With the capacious **digital versatile disk (DVD),** the electronics industry has at last found an acceptable replacement for the analog VHS video tape. The DVD has replaced the videotape used in video cassette recorders (VCRs) and, more important for this discussion, replaced the CD-ROM in personal computers and servers. The DVD takes video into the digital age. It delivers movies with impressive picture quality, and it can be randomly accessed like audio CDs, which DVD machines can also play. Vast volumes of data can be crammed onto the disk, currently seven times as much as a CD-ROM. With DVD's huge storage capacity and vivid quality, PC games have become more realistic and educational software incorporates more video. Following in the wake of these developments has been a new crest of traffic over the Internet and corporate intranets, as this material is incorporated into Web sites.

The DVD's greater capacity is due to three differences from CDs (Figure 6.11):

1. Bits are packed more closely on a DVD. The spacing between loops of a spiral on a CD is 1.6 μm and the minimum distance between pits along the spiral is 0.834 μm.

   The DVD uses a laser with shorter wavelength and achieves a loop spacing of 0.74 μm and a minimum distance between pits of 0.4 μm. The result of these two improvements is about a seven-fold increase in capacity, to about 4.7 GB.

2. The DVD employs a second layer of pits and lands on top of the first layer. A dual-layer DVD has a semireflective layer on top of the reflective layer, and by adjusting focus, the lasers in DVD drives can read each layer separately. This technique almost doubles the capacity of the disk, to about 8.5 GB. The lower reflectivity of the second layer limits its storage capacity so that a full doubling is not achieved.

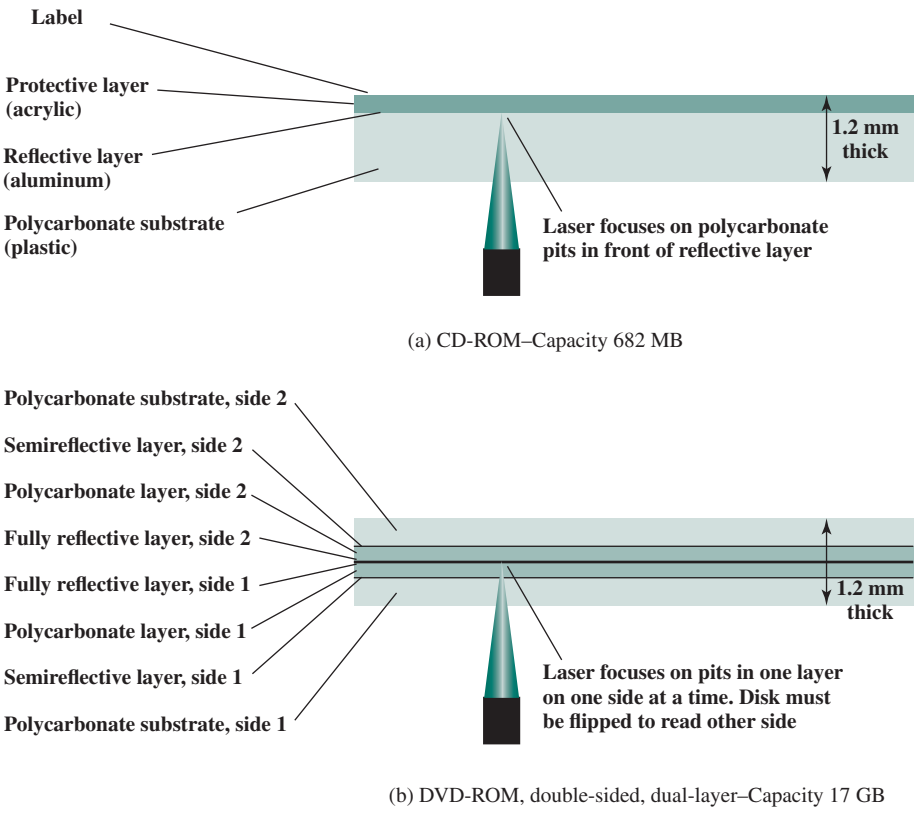


Figure 6.11 CD-ROM and DVD-ROM

3. The **DVD-ROM** can be two sided, whereas data are recorded on only one side of a CD. This brings total capacity up to 17 GB.

As with the CD, DVDs come in writeable as well as read-only versions (Table 6.6).

High-Definition Optical Disks

High-definition optical disks are designed to store high-definition videos and to provide significantly greater storage capacity compared to DVDs. The higher bit density is achieved by using a laser with a shorter wavelength, in the blue-violet range. The data pits, which constitute the digital 1s and 0s, are smaller on the high-definition optical disks compared to DVD because of the shorter laser wavelength.

Two competing disk formats and technologies initially competed for market acceptance: HD DVD and **Blu-ray** DVD. The Blu-ray scheme ultimately achieved market dominance. The HD DVD scheme can store 15 GB on a single layer on a single side. Blu-ray positions the data layer on the disk closer to the laser (shown on the right-hand side of each diagram in Figure 6.12). This enables a tighter focus and less distortion and thus smaller pits and tracks. Blu-ray can store 25 GB on a single layer. Three versions are available: read only (BD-ROM), recordable once (BD-R), and rerecordable (BD-RE).

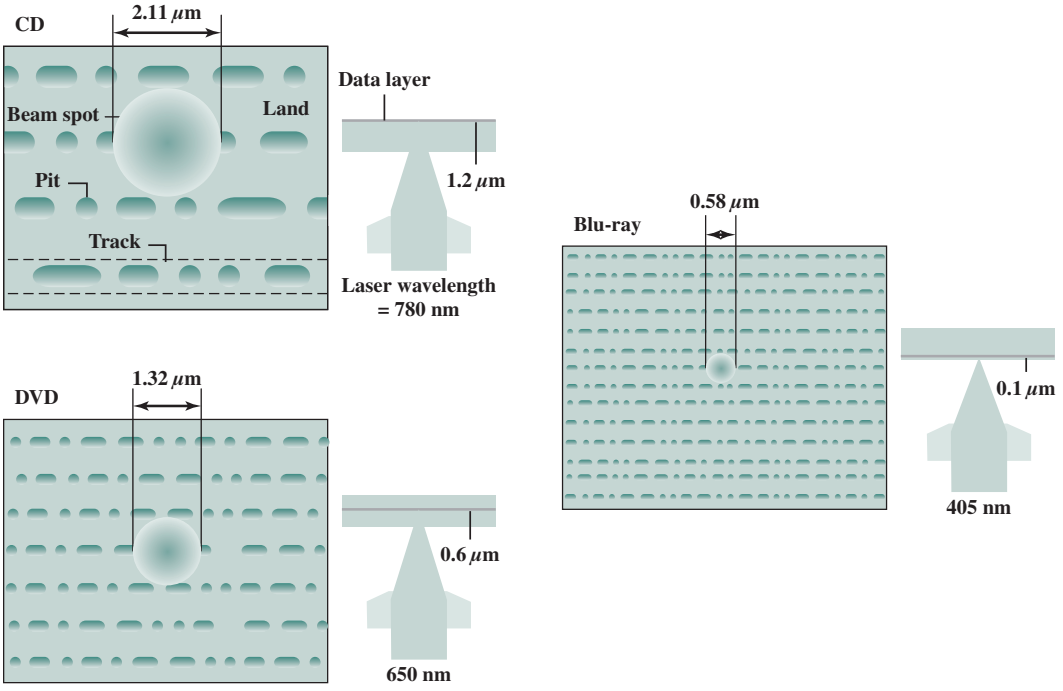


Figure 6.12 Optical Memory Characteristics

6.5 MAGNETIC TAPE

Tape systems use the same reading and recording techniques as disk systems. The medium is flexible polyester (similar to that used in some clothing) tape coated with magnetizable material. The coating may consist of particles of pure metal in special binders or vapor-plated metal films. The tape and the tape drive are analogous to a home tape recorder system. Tape widths vary from 0.38 cm (0.15 inch) to 1.27 cm (0.5 inch). Tapes used to be packaged as open reels that have to be threaded through a second spindle for use. Today, virtually all tapes are housed in cartridges.

Data on the tape are structured as a number of parallel tracks running lengthwise. Earlier tape systems typically used nine tracks. This made it possible to store data one byte at a time, with an additional parity bit as the ninth track. This was followed by tape systems using 18 or 36 tracks, corresponding to a digital word or double word. The recording of data in this form is referred to as **parallel recording**. Most modern systems instead use **serial recording**, in which data are laid out as a sequence of bits along each track, as is done with magnetic disks. As with the disk, data are read and written in contiguous blocks, called *physical records,* on a tape. Blocks on the tape are separated by gaps referred to as *interrecord* gaps. As with the disk, the tape is formatted to assist in locating physical records.

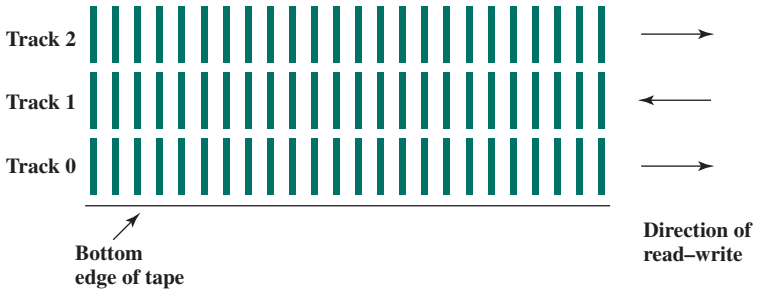
The typical recording technique used in serial tapes is referred to as **serpentine recording.** In this technique, when data are being recorded, the first set of bits is recorded along the whole length of the tape. When the end of the tape is reached,

the heads are repositioned to record a new track, and the tape is again recorded on its whole length, this time in the opposite direction. That process continues, back and forth, until the tape is full (Figure 6.13a). To increase speed, the read-write head is capable of reading and writing a number of adjacent tracks simultaneously (typically two to eight tracks). Data are still recorded serially along individual tracks, but blocks in sequence are stored on adjacent tracks, as suggested by Figure 6.13b.

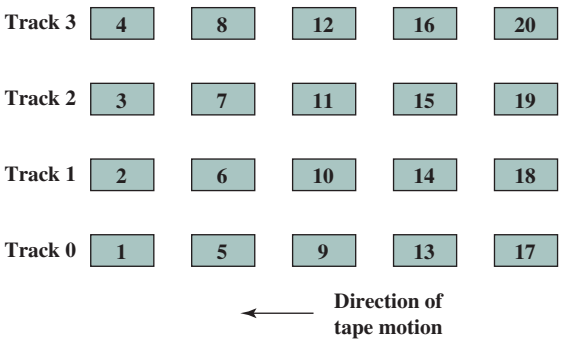
A tape drive is a *sequential-access* device. If the tape head is positioned at record 1, then to read record N, it is necessary to read physical records 1 through $N-1$, one at a time. If the head is currently positioned beyond the desired record, it is necessary to rewind the tape a certain distance and begin reading forward. Unlike the disk, the tape is in motion only during a read or write operation.

In contrast to the tape, the disk drive is referred to as a *direct-access* device. A disk drive need not read all the sectors on a disk sequentially to get to the desired one. It must only wait for the intervening sectors within one track and can make successive accesses to any track.

Magnetic tape was the first kind of secondary memory. It is still widely used as the lowest-cost, slowest-speed member of the memory hierarchy.



(a) Serpentine reading and writing



(b) Block layout for system that reads–writes four tracks simultaneously

Figure 6.13 Typical Magnetic Tape Features

Table 6.7 LTO Tape Drives

	LTO-1	LTO-2	LTO-3	LTO-4	LTO-5	LTO-6	LTO-7	LTO-8
Release date	2000	2003	2005	2007	2010	2012	TBA	TBA
Compressed capacity	200 GB	400 GB	800 GB	1600 GB	3.2 TB	8 TB	16 TB	32 TB
Compressed transfer rate	40 MB/s	80 MB/s	160 MB/s	240 MB/s	280 MB/s	400 MB/s	788 MB/s	1.18 GB/s
Linear density (bits/mm)	4880	7398	9638	13,250	15,142	15,143		
Tape tracks	384	512	704	896	1280	2176		
Tape length (m)	609	609	680	820	846	846		
Tape width (cm)	1.27	1.27	1.27	1.27	1.27	1.27		
Write elements	8	8	16	16	16	16		
WORM?	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Encryption Capable?	No	No	No	Yes	Yes	Yes	Yes	Yes
Partitioning?	No	No	No	No	Yes	Yes	Yes	Yes

The dominant tape technology today is a cartridge system known as linear tape-open (LTO). LTO was developed in the late 1990s as an open-source alternative to the various proprietary systems on the market. Table 6.7 shows parameters for the various LTO generations. See Appendix J for details.

6.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

access time	DVD-RW	optical memory
Blu-ray	fixed-head disk	pit
CD	flash memory	platter
CD-R	floppy disk	RAID
CD-ROM	gap	removable disk
CD-RW	hard disk drive (HDD)	rotational delay
constant angular velocity (CAV)	head	sector
constant linear velocity (CLV)	land	seek time
cylinder	magnetic disk	serpentine recording
DVD	magnetic tape	solid state drive (SSD)
DVD-R	magnetoresistive	striped data
DVD-ROM	movable-head disk	substrate
	multiple zone recording	track
	nonremovable disk	transfer time

Review Questions

6.1 What are the advantages of using a glass substrate for a magnetic disk?
6.2 How are data written onto a magnetic disk?
6.3 How are data read from a magnetic disk?
6.4 Explain the difference between a simple CAV system and a multiple zone recording system.
6.5 Define the terms *track, cylinder,* and *sector*.
6.6 What is the typical disk sector size?
6.7 Define the terms *seek time, rotational delay, access time,* and *transfer time*.
6.8 What common characteristics are shared by all RAID levels?
6.9 Briefly define the seven RAID levels.
6.10 Explain the term *striped data*.
6.11 How is redundancy achieved in a RAID system?
6.12 In the context of RAID, what is the distinction between parallel access and independent access?
6.13 What is the difference between CAV and CLV?
6.14 What differences between a CD and a DVD account for the larger capacity of the latter?
6.15 Explain serpentine recording.

Problems

6.1 Justify Equation 6.1. That is, explain how each of the three terms on the right-hand side of the equation contributes to the value on the left-hand side.

6.2 Consider a disk with N tracks numbered from 0 to $(N - 1)$ and assume that requested sectors are distributed randomly and evenly over the disk. We want to calculate the average number of tracks traversed by a seek.

a. First, calculate the probability of a seek of length j when the head is currently positioned over track t. *Hint*: This is a matter of determining the total number of combinations, recognizing that all track positions for the destination of the seek are equally likely.

b. Next, calculate the probability of a seek of length K. *Hint*: This involves the summing over all possible combinations of movements of K tracks.

c. Calculate the average number of tracks traversed by a seek, using the formula for expected value

$$\mathrm{E}[x] = \sum_{i=0}^{N-1} i \times \Pr[x = i]$$

Hint: Use the equalities: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$; $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

d. Show that for large values of N, the average number of tracks traversed by a seek approaches $N/3$.

6.3 Define the following for a disk system:

t_s = seek time; average time to position head over track
r = rotation speed of the disk, in revolutions per second
n = number of bits per sector
N = capacity of a track, in bits
t_{sector} = time to access a sector

Develop a formula for t_{sector} as a function of the other parameters.

6.4 Consider a magnetic disk drive with 8 surfaces, 512 tracks per surface, and 64 sectors per track. Sector size is 1 kB. The average seek time is 8 ms, the track-to-track access time is 1.5 ms, and the drive rotates at 3600 rpm. Successive tracks in a cylinder can be read without head movement.

a. What is the disk capacity?

b. What is the average access time? Assume this file is stored in successive sectors and tracks of successive cylinders, starting at sector 0, track 0, of cylinder *i*.

c. Estimate the time required to transfer a 5-MB file.

d. What is the burst transfer rate?

6.5 Consider a single-platter disk with the following parameters: rotation speed: 7200 rpm; number of tracks on one side of platter: 30,000; number of sectors per track: 600; seek time: one ms for every hundred tracks traversed. Let the disk receive a request to access a random sector on a random track and assume the disk head starts at track 0.

a. What is the average seek time?

b. What is the average rotational latency?

c. What is the transfer time for a sector?

d. What is the total average time to satisfy a request?

6.6 A distinction is made between physical records and logical records. A **logical record** is a collection of related data elements treated as a conceptual unit, independent of how or where the information is stored. A **physical record** is a contiguous area of storage space that is defined by the characteristics of the storage device and operating system. Assume a disk system in which each physical record contains thirty 120-byte logical records. Calculate how much disk space (in sectors, tracks, and surfaces) will be required to store 300,000 logical records if the disk is fixed-sector with 512 bytes/sector, with 96 sectors/track, 110 tracks per surface, and 8 usable surfaces. Ignore any file header record(s) and track indexes, and assume that records cannot span two sectors.

6.7 Consider a disk that rotates at 3600 rpm. The seek time to move the head between adjacent tracks is 2 ms. There are 32 sectors per track, which are stored in linear order from sector 0 through sector 31. The head sees the sectors in ascending order. Assume the read/write head is positioned at the start of sector 1 on track 8. There is a main memory buffer large enough to hold an entire track. Data is transferred between disk locations by reading from the source track into the main memory buffer and then writing the data from the buffer to the target track.

a. How long will it take to transfer sector 1 on track 8 to sector 1 on track 9?

b. How long will it take to transfer all the sectors of track 8 to the corresponding sectors of track 9?

6.8 It should be clear that disk striping can improve data transfer rate when the strip size is small compared to the I/O request size. It should also be clear that RAID 0 provides improved performance relative to a single large disk, because multiple I/O requests can be handled in parallel. However, in this latter case, is disk striping necessary? That is, does disk striping improve I/O request rate performance compared to a comparable disk array without striping?

6.9 Consider a 4-drive, 200 GB-per-drive RAID array. What is the available data storage capacity for each of the RAID levels 0, 1, 3, 4, 5, and 6?

6.10 For a compact disk, audio is converted to digital with 16-bit samples, and is treated a stream of 8-bit bytes for storage. One simple scheme for storing this data, called direct recording, would be to represent a 1 by a land and a 0 by a pit. Instead, each byte is expanded into a 14-bit binary number. It turns out that exactly 256 (2^8) of the total of 16,134 (2^{14}) 14-bit numbers have at least two 0s between every pair of 1s, and these are the numbers selected for the expansion from 8 to 14 bits. The optical system detects the presence of 1s by detecting a transition for pit to land or land to pit. It detects 0s by measuring the distances between intensity changes. This scheme requires that there are no 1s in succession; hence the use of the 8-to-14 code.

The advantage of this scheme is as follows. For a given laser beam diameter, there is a minimum-pit size, regardless of how the bits are represented. With this scheme, this minimum-pit size stores 3 bits, because at least two 0s follow every 1. With direct recording, the same pit would be able to store only one bit. Considering both the number of bits stored per pit and the 8-to-14 bit expansion, which scheme stores the most bits and by what factor?

6.11 Design a backup strategy for a computer system. One option is to use plug-in external disks, which cost $150 for each 500 GB drive. Another option is to buy a tape drive for $2500, and 400 GB tapes for $50 apiece. (These were realistic prices in 2008.) A typical backup strategy is to have two sets of backup media onsite, with backups alternately written on them so in case the system fails while making a backup, the previous version is still intact. There's also a third set kept offsite, with the offsite set periodically swapped with an on-site set.

a. Assume you have 1 TB (1000 GB) of data to back up. How much would a disk backup system cost?

b. How much would a tape backup system cost for 1 TB?

c. How large would each backup have to be in order for a tape strategy to be less expensive?

d. What kind of backup strategy favors tapes?

chapter

8

The Memory System

Chapter Objectives

In this chapter you will learn about:

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

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

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

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

8.1 BASIC CONCEPTS

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

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

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

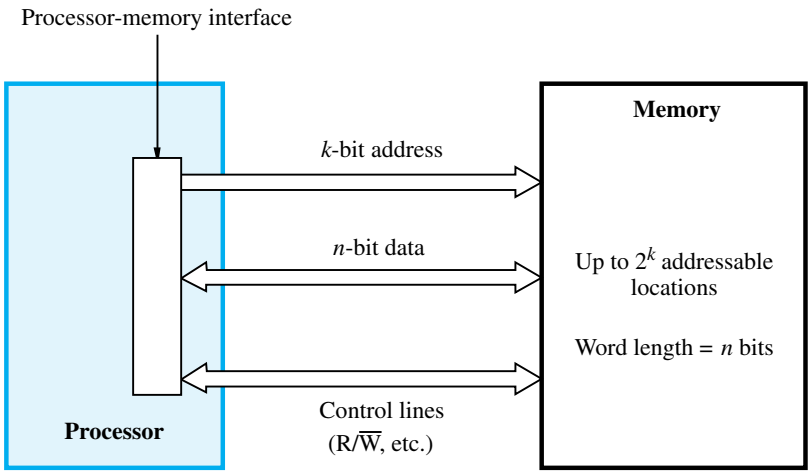


Figure 8.1 Connection of the memory to the processor.

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

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

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

Cache and Virtual Memory

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

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

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

Block Transfers

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

8.2 SEMICONDUCTOR RAM MEMORIES

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

8.2.1 INTERNAL ORGANIZATION OF MEMORY CHIPS

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

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

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

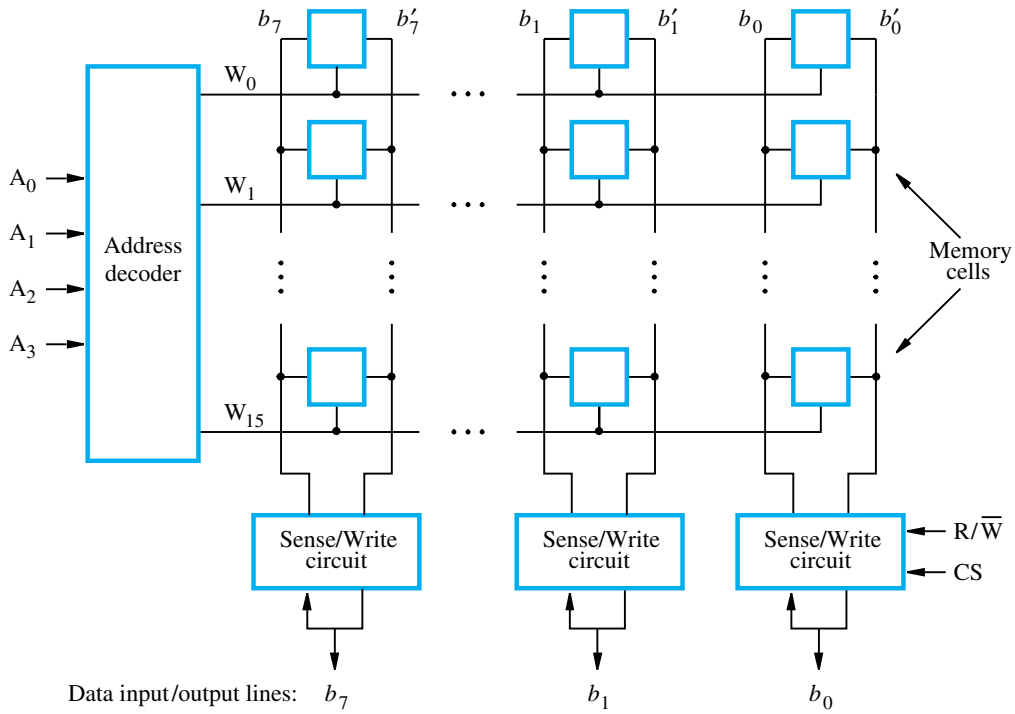


Figure 8.2 Organization of bit cells in a memory chip.

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

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

8.2.2 STATIC MEMORIES

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

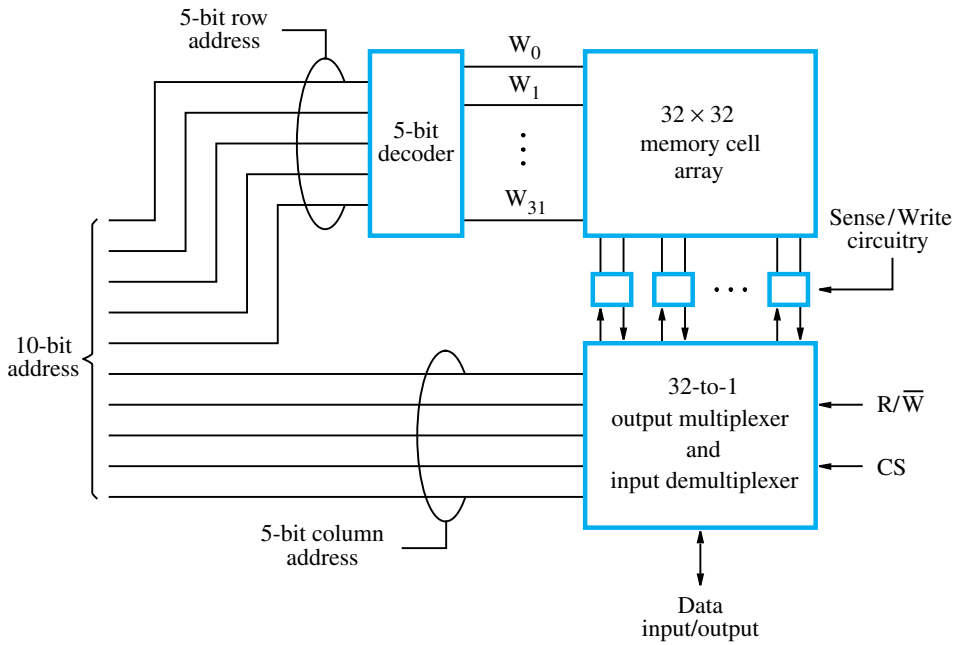


Figure 8.3 Organization of a 1K × 1 memory chip.

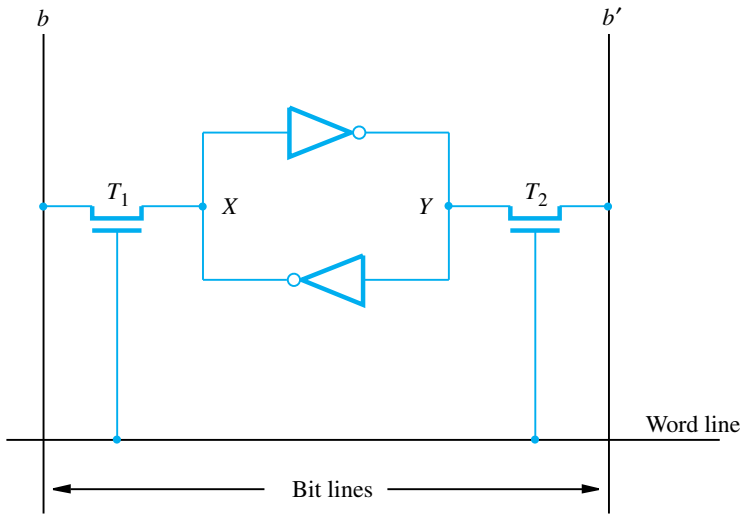


Figure 8.4 A static RAM cell.

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

Read Operation

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

Write Operation

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

CMOS Cell

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

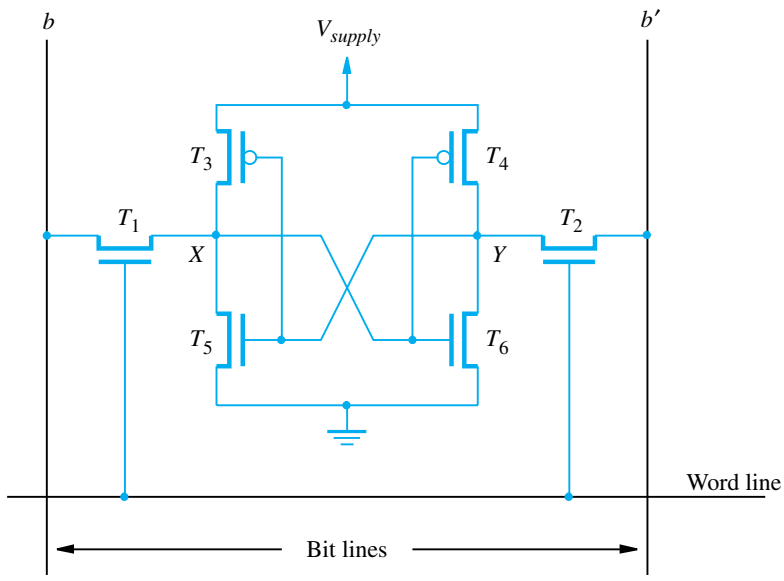


Figure 8.5 An example of a CMOS memory cell.

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

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

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

8.2.3 DYNAMIC RAMS

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

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

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

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

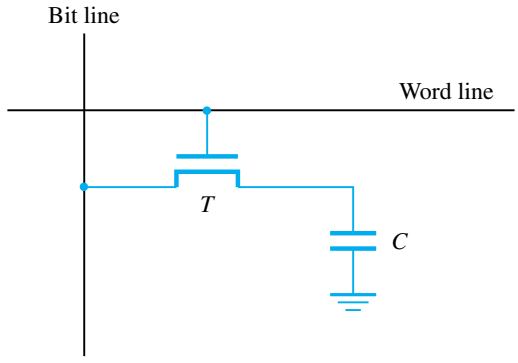


Figure 8.6 A single-transistor dynamic memory cell.

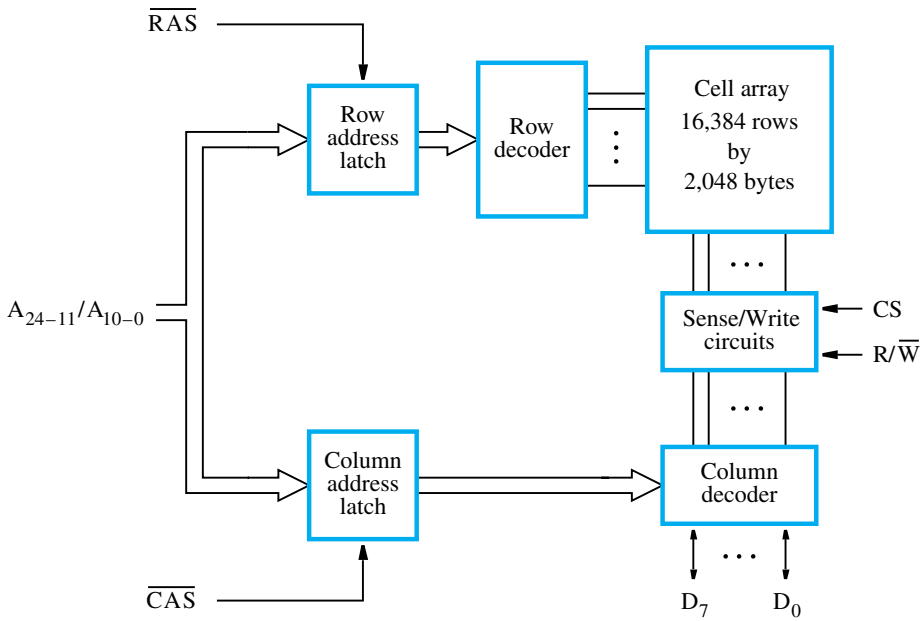


Figure 8.7 Internal organization of a 32M × 8 dynamic memory chip.

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

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

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

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

Fast Page Mode

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

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

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

8.2.4 SYNCHRONOUS DRAMS

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

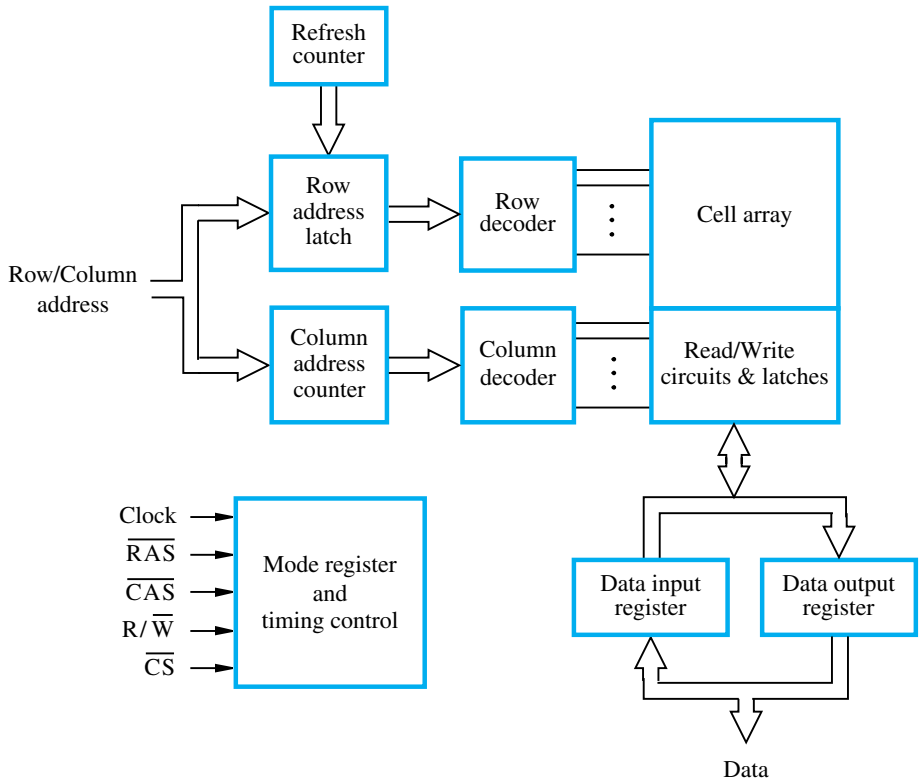


Figure 8.8 Synchronous DRAM.

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

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

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

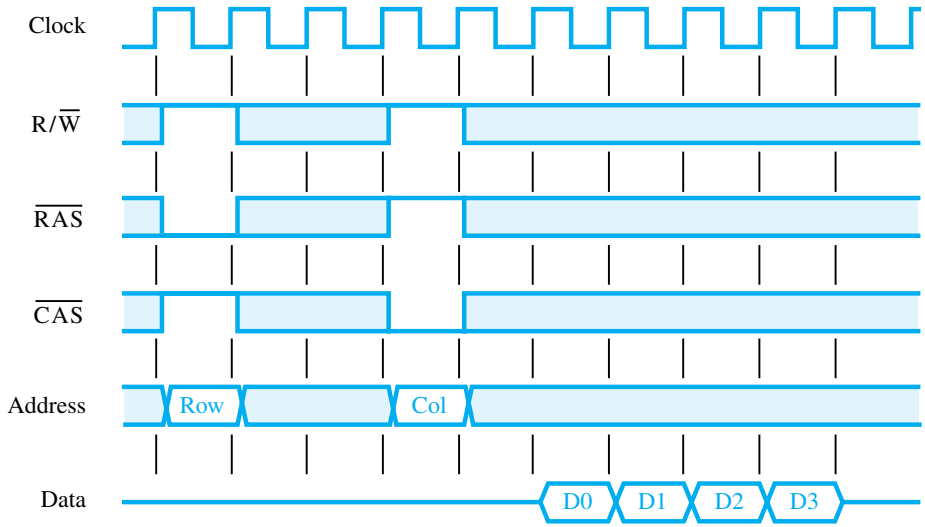


Figure 8.9 A burst read of length 4 in an SDRAM.

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

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

Latency and Bandwidth

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

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

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

Double-Data-Rate SDRAM

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

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

Rambus Memory

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

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

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

8.2.5 STRUCTURE OF LARGER MEMORIES

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

Static Memory Systems

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

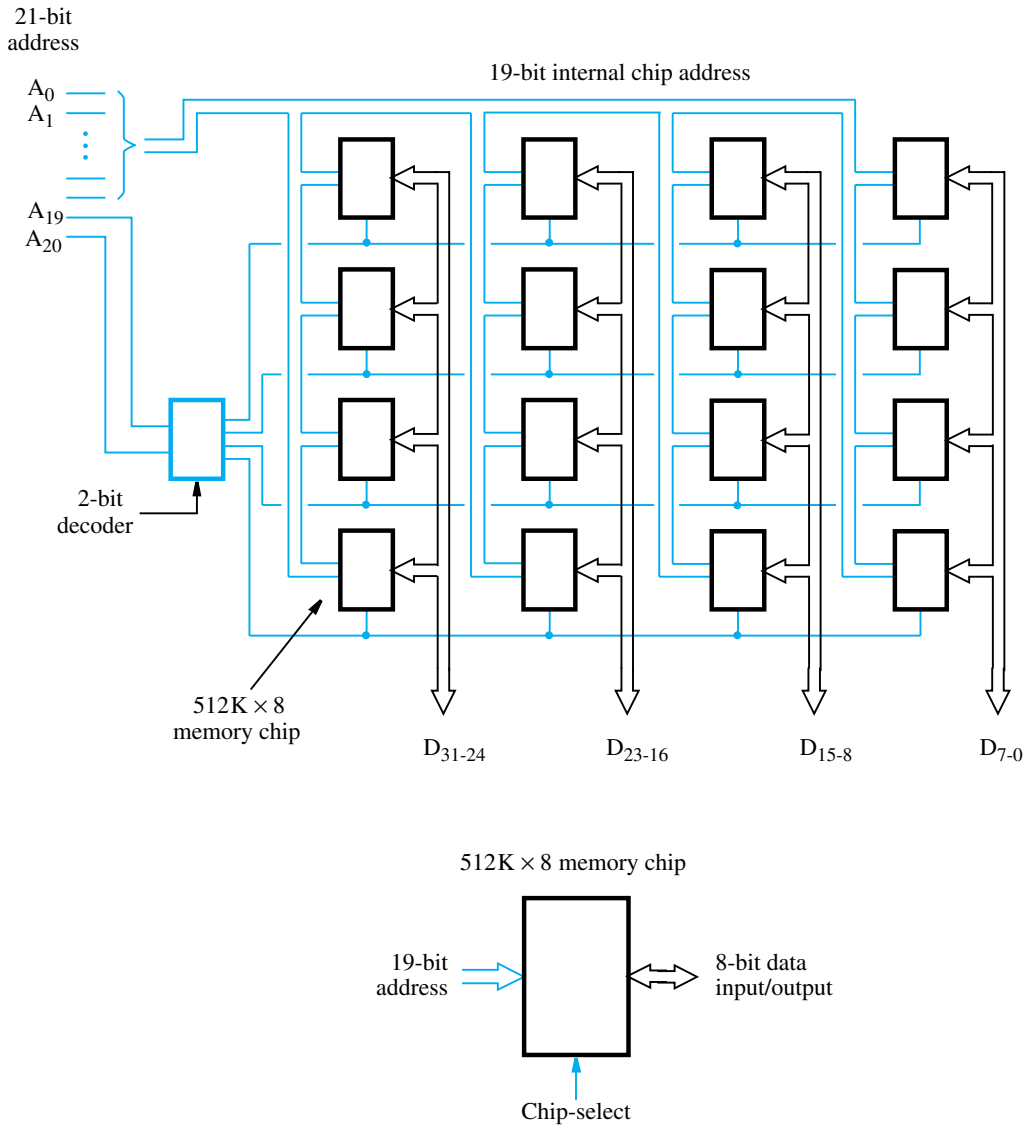


Figure 8.10 Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

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

Dynamic Memory Systems

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

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

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

Memory Controller

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

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

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

Refresh Overhead

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

Choice of Technology

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

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

8.3 READ-ONLY MEMORIES

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

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

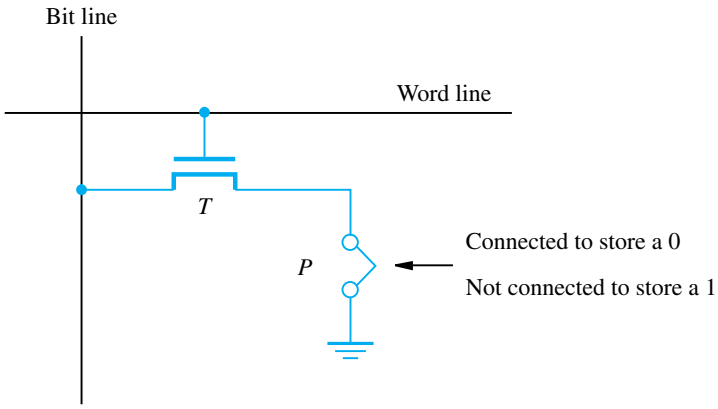


Figure 8.11 A ROM cell.

8.3.1 ROM

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

8.3.2 PROM

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

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

8.3.3 EPROM

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

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

8.3.4 EEPROM

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

8.3.5 FLASH MEMORY

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

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

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

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

Flash Cards

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

Flash Drives

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

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

8.4 DIRECT MEMORY ACCESS

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

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

```
Load    R2, DATAIN
```

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

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

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

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

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

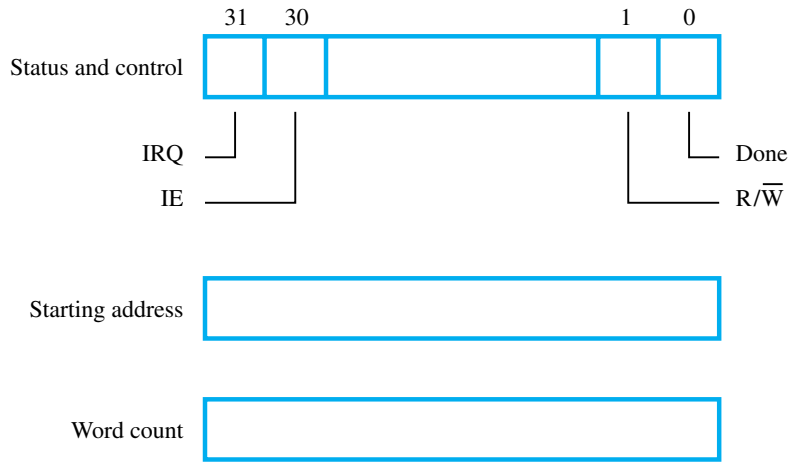


Figure 8.12 Typical registers in a DMA controller.

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

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

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

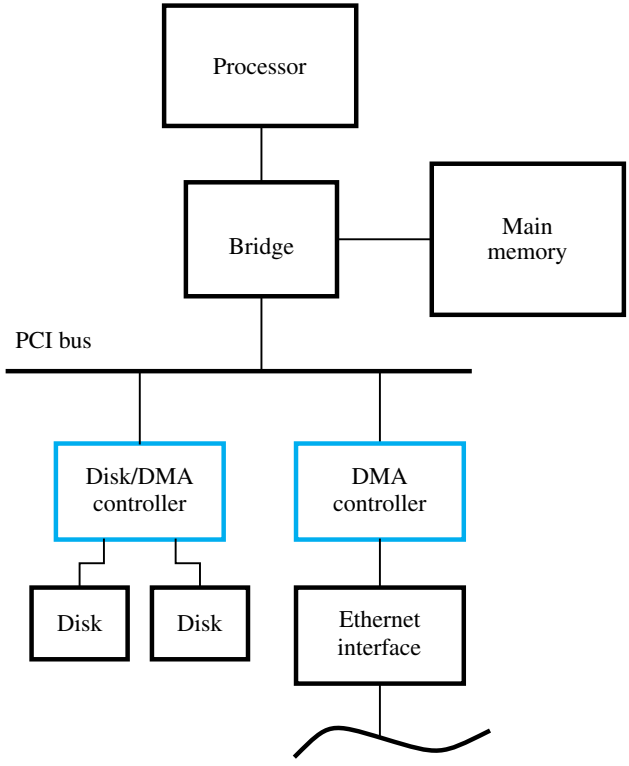


Figure 8.13 Use of DMA controllers in a computer system.

8.5 MEMORY HIERARCHY

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

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

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

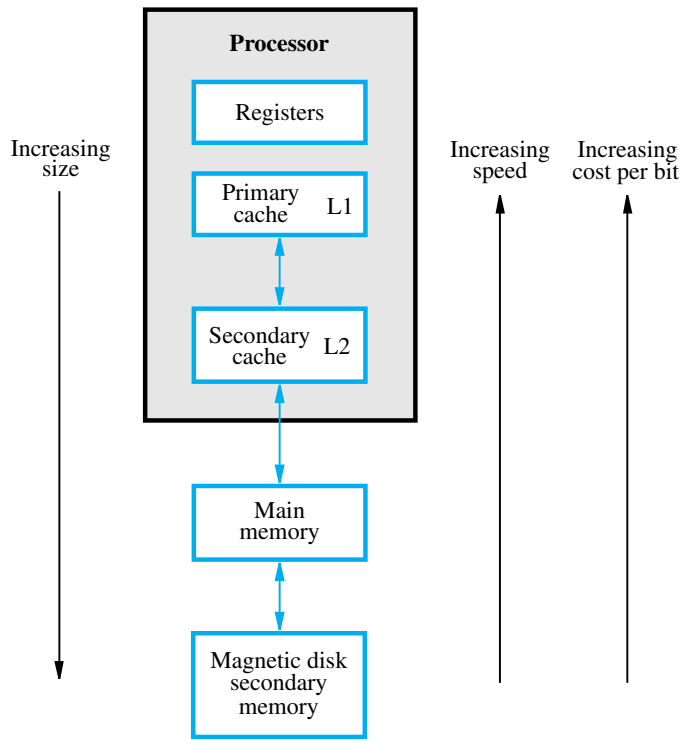


Figure 8.14 Memory hierarchy.

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

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

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

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

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

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

8.6 CACHE MEMORIES

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

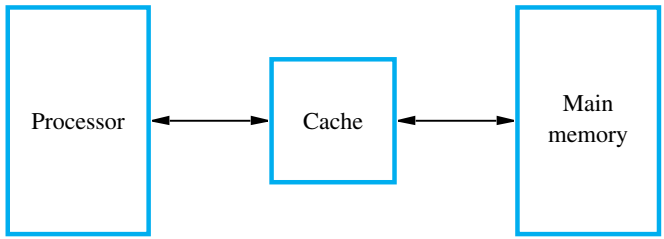


Figure 8.15 Use of a cache memory.

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

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

Cache Hits

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

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

Cache Misses

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

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

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

8.6.1 MAPPING FUNCTIONS

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

Direct Mapping

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

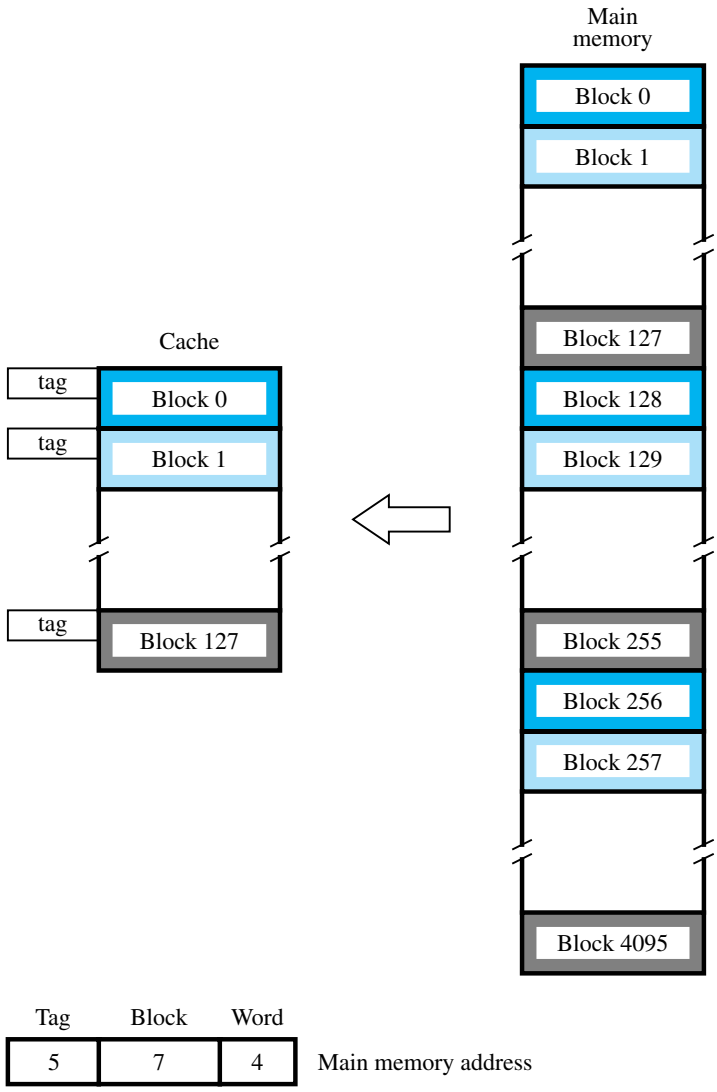


Figure 8.16 Direct-mapped cache.

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

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

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

Associative Mapping

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

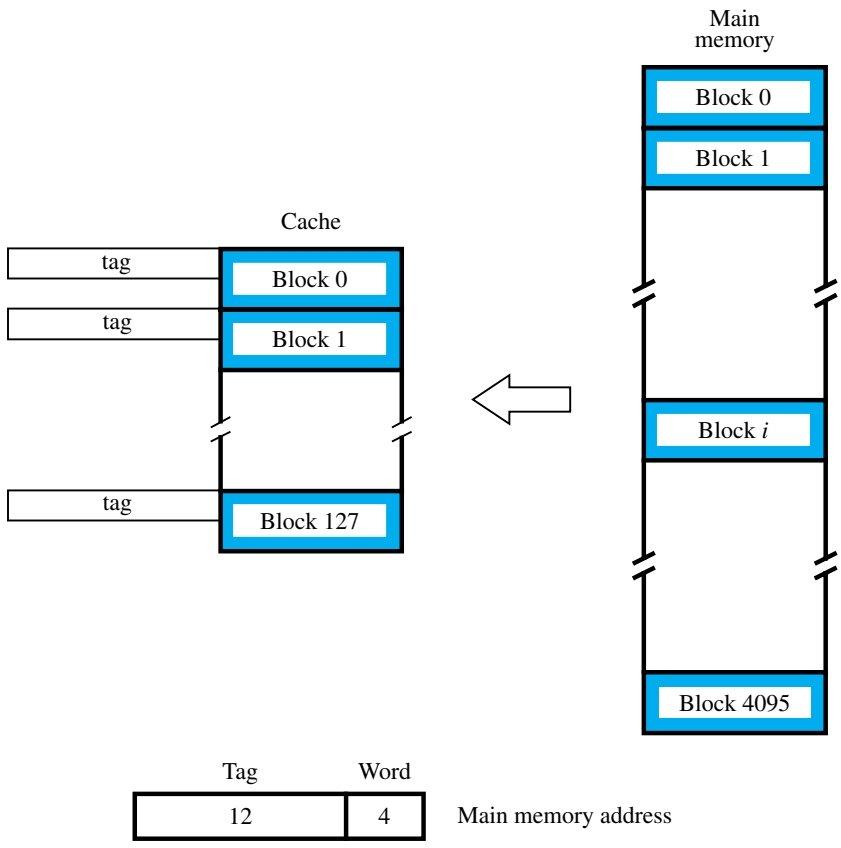


Figure 8.17 Associative-mapped cache.

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

Set-Associative Mapping

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

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

Stale Data

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

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

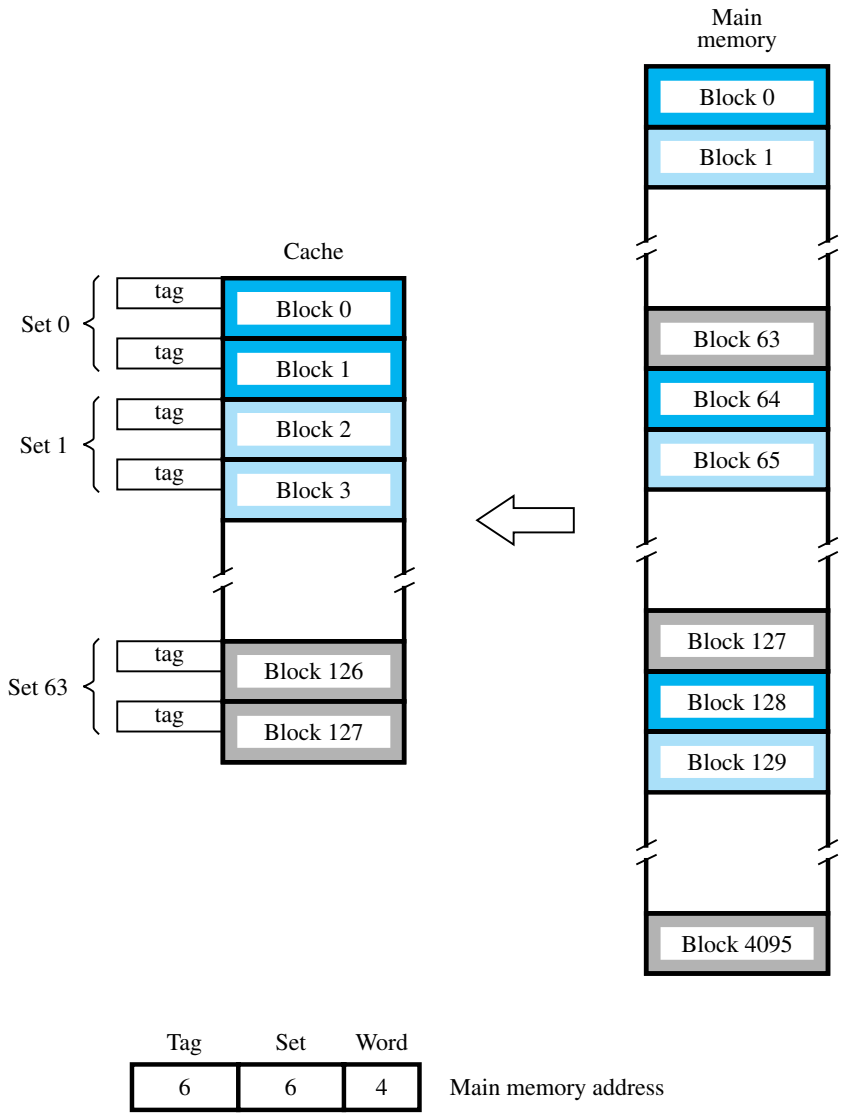


Figure 8.18 Set-associative-mapped cache with two blocks per set.

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

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

8.6.2 REPLACEMENT ALGORITHMS

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

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

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

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

8.6.3 EXAMPLES OF MAPPING TECHNIQUES

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

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

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

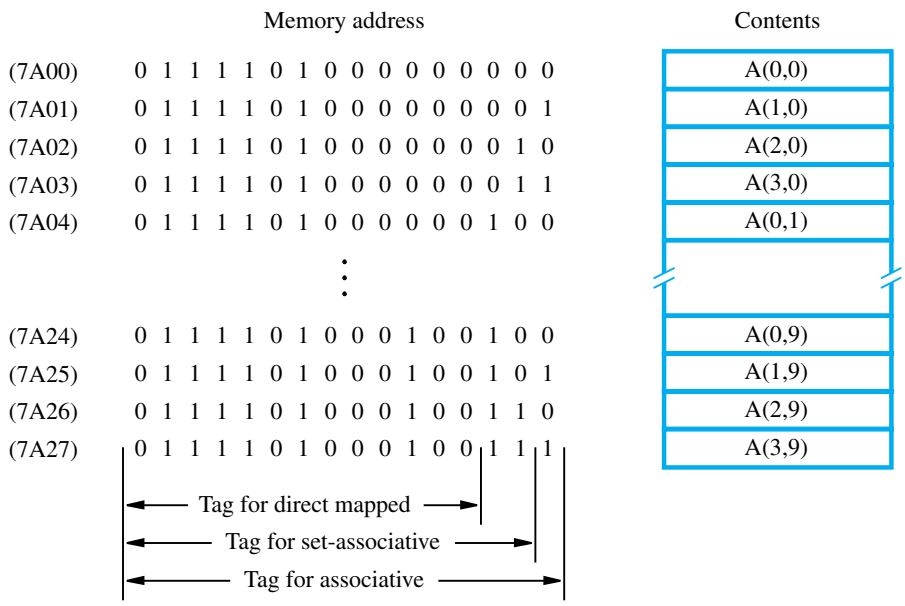


Figure 8.19 An array stored in the main memory.

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

Figure 8.20 Task for example in Section 8.6.3.

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

Direct-Mapped Cache

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

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

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

Figure 8.21 Contents of a direct-mapped data cache.

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

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

Associative-Mapped Cache

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

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

Set-Associative-Mapped Cache

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

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

Figure 8.22 Contents of an associative-mapped data cache.

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

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

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

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

8.7 PERFORMANCE CONSIDERATIONS

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

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

8.7.1 HIT RATE AND MISS PENALTY

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

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

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

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

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

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

Example 8.1

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

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

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

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

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

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

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

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

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

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

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

8.7.2 CACHES ON THE PROCESSOR CHIP

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

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

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

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

$$t_{avg} = h_1 C_1 + (1 - h_1)(h_2 C_2 + (1 - h_2)M)$$

where

h_1 is the hit rate in the L1 caches.

h_2 is the hit rate in the L2 cache.

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

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

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

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

8.7.3 OTHER ENHANCEMENTS

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

Write Buffer

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

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

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

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

Prefetching

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

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

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

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

Lockup-Free Cache

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

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

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

8.8 VIRTUAL MEMORY

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

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

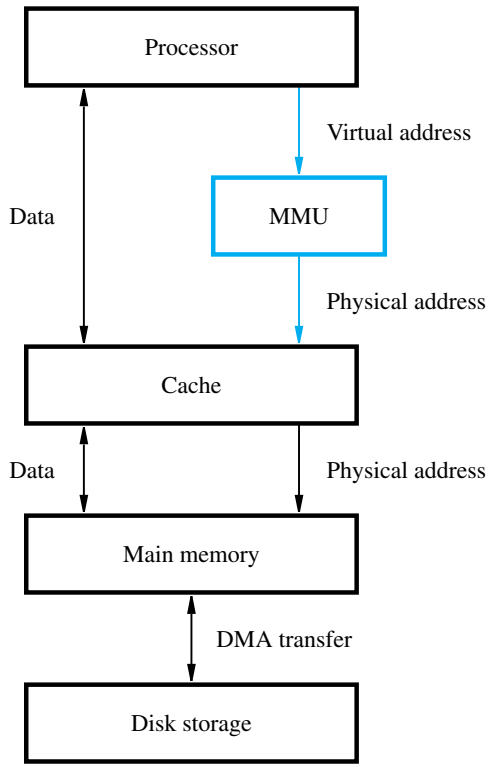


Figure 8.24 Virtual memory organization.

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

8.8.1 Address Translation

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

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

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

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

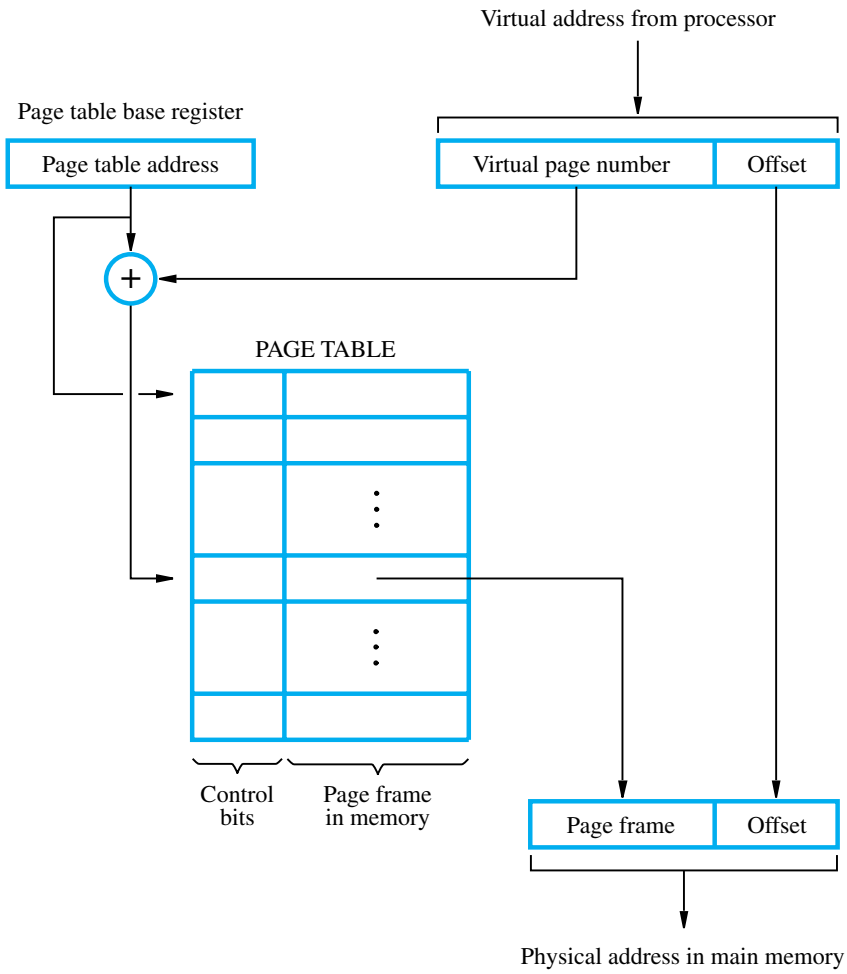


Figure 8.25 Virtual-memory address translation.

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

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

Translation Lookaside Buffer

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

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

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

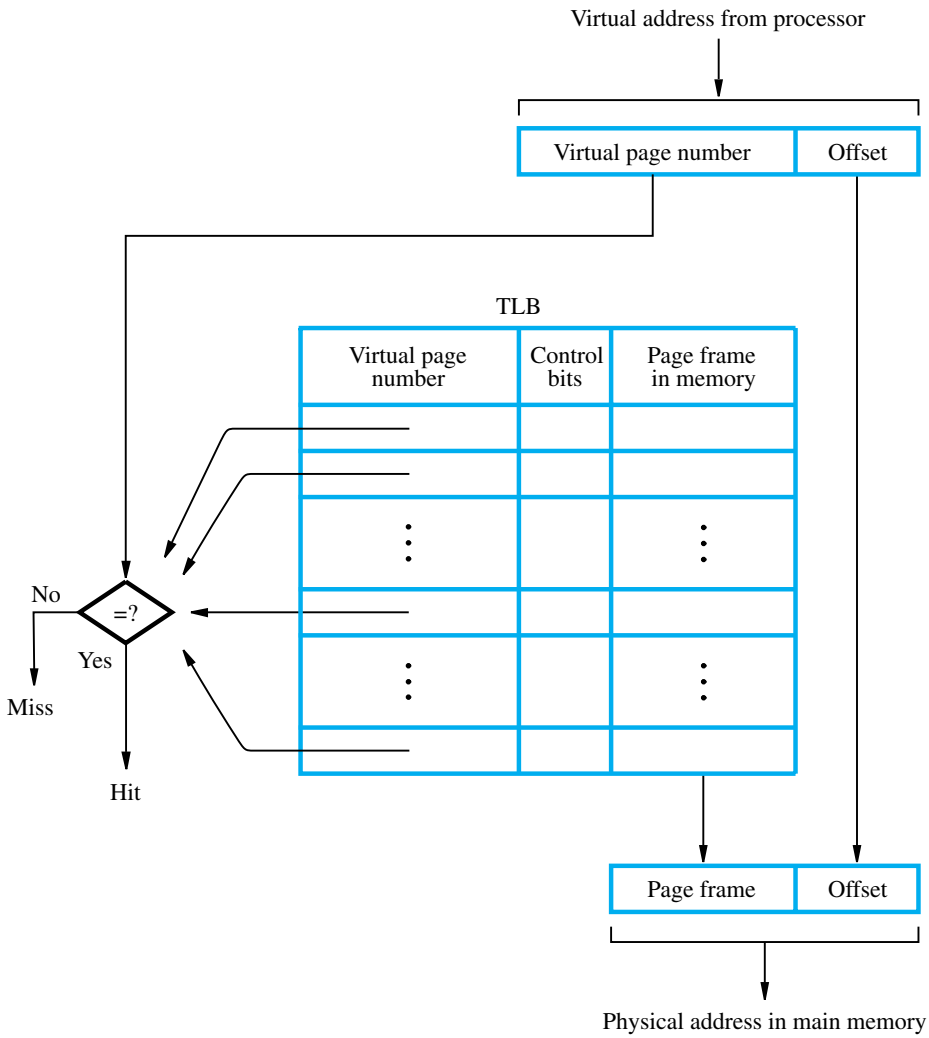


Figure 8.26 Use of an associative-mapped TLB.

Page Faults

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

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

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

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

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

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

8.9 MEMORY MANAGEMENT REQUIREMENTS

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

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

to be used in the translation process. Hence, by changing the contents of this register, the operating system can switch from one space to another. The physical main memory is thus shared by the active pages of the system space and several user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.

In any computer system in which independent user programs coexist in the main memory, the notion of *protection* must be addressed. No program should be allowed to destroy either the data or instructions of other programs in the memory. The needed protection can be provided in several ways. Let us first consider the most basic form of protection. Most processors can operate in one of two modes, the *supervisor mode* and the *user mode*. The processor is usually placed in the supervisor mode when operating system routines are being executed and in the user mode to execute user programs. In the user mode, some machine instructions cannot be executed. These are *privileged instructions*. They include instructions that modify the page table base register, which can only be executed while the processor is in the supervisor mode. Since a user program is executed in the user mode, it is prevented from accessing the page tables of other users or of the system space.

It is sometimes desirable for one application program to have access to certain pages belonging to another program. The operating system can arrange this by causing these pages to appear in both spaces. The shared pages will therefore have entries in two different page tables. The control bits in each table entry can be set to control the access privileges granted to each program. For example, one program may be allowed to read and write a given page, while the other program may be given only read access.

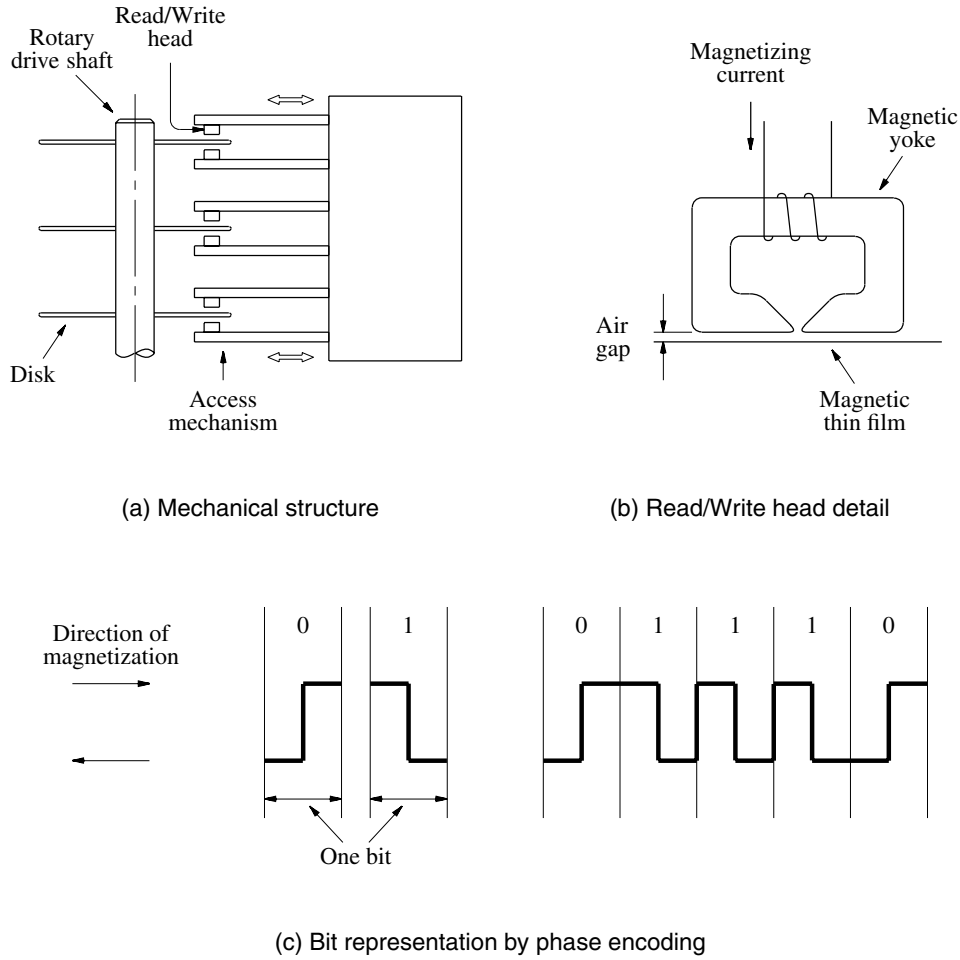
8.10 SECONDARY STORAGE

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

8.10.1 MAGNETIC HARD DISKS

The storage medium in a magnetic-disk system consists of one or more disk platters mounted on a common spindle. A thin magnetic film is deposited on each platter, usually on both sides. The assembly is placed in a drive that causes it to rotate at a constant speed. The magnetized surfaces move in close proximity to read/write heads, as shown in Figure 8.27*a*. Data are stored on concentric tracks, and the read/write heads move radially to access different tracks.

Each read/write head consists of a magnetic yoke and a magnetizing coil, as indicated in Figure 8.27*b*. Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately underneath the head to switch to a direction parallel to the applied



(a) Mechanical structure

(b) Read/Write head detail

(c) Bit representation by phase encoding

Figure 8.27 Magnetic disk principles.

field. The same head can be used for reading the stored information. In this case, changes in the magnetic field in the vicinity of the head caused by the movement of the film relative to the yoke induce a voltage in the coil, which now serves as a sense coil. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film. Only changes in the magnetic field under the head can be sensed during the Read operation. Therefore, if the binary states 0 and 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-to-1 and at 1-to-0 transitions in the bit stream. A long string of 0s or 1s causes an induced voltage only at the beginning and end of the string. Therefore, to determine the number of consecutive 0s or 1s stored, a clock must provide information for synchronization.

In some early designs, a clock was stored on a separate track, on which a change in magnetization is forced for each bit period. Using the clock signal as a reference, the data stored on other tracks can be read correctly. The modern approach is to combine the clocking information with the data. Several different techniques have been developed for such encoding. One simple scheme, depicted in Figure 8.27*c*, is known as *phase encoding* or *Manchester encoding*. In this scheme, changes in magnetization occur for each data bit, as shown in the figure. Clocking information is provided by the change in magnetization at the midpoint of each bit period. The drawback of Manchester encoding is its poor bit-storage density. The space required to represent each bit must be large enough to accommodate two changes in magnetization. We use the Manchester encoding example to illustrate how a *self-clocking* scheme may be implemented, because it is easy to understand. Other, more compact codes have been developed. They are much more efficient and provide better storage density. They also require more complex control circuitry. The discussion of such codes is beyond the scope of this book.

Read/write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities and reliable Read and Write operations. When the disks are moving at their steady rate, air pressure develops between the disk surface and the head and forces the head away from the surface. This force is counterbalanced by a spring-loaded mounting arrangement that presses the head toward the surface. The flexible spring connection between the head and its arm mounting permits the head to fly at the desired distance away from the surface in spite of any small variations in the flatness of the surface.

In most modern disk units, the disks and the read/write heads are placed in a sealed, air-filtered enclosure. This approach is known as *Winchester technology*. In such units, the read/write heads can operate closer to the magnetized track surfaces, because dust particles, which are a problem in unsealed assemblies, are absent. The closer the heads are to a track surface, the more densely the data can be packed along the track, and the closer the tracks can be to each other. Thus, Winchester disks have a larger capacity for a given physical size compared to unsealed units. Another advantage of Winchester technology is that data integrity tends to be greater in sealed units, where the storage medium is not exposed to contaminating elements.

The read/write heads of a disk system are movable. There is one head per surface. All heads are mounted on a comb-like arm that can move radially across the stack of disks to provide access to individual tracks, as shown in Figure 8.27*a*. To read or write data on a given track, the read/write heads must first be positioned over that track.

The disk system consists of three key parts. One part is the assembly of disk platters, which is usually referred to as the *disk*. The second part comprises the electromechanical mechanism that spins the disk and moves the read/write heads; it is called the *disk drive*. The third part is the *disk controller*, which is the electronic circuitry that controls the operation of the system. The disk controller may be implemented as a separate module, or it may be incorporated into the enclosure that contains the entire disk system. We should note that the term disk is often used to refer to the combined package of the disk drive and the disk it contains. We will do so in the sections that follow, when there is no ambiguity in the meaning of the term.

Organization and Accessing of Data on a Disk

The organization of data on a disk is illustrated in Figure 8.28. Each surface is divided into concentric *tracks*, and each track is divided into *sectors*. The set of corresponding tracks on all surfaces of a stack of disks forms a logical *cylinder*. All tracks of a cylinder can be accessed without moving the read/write heads. Data are accessed by specifying the surface number, the track number, and the sector number. Read and Write operations always start at sector boundaries.

Data bits are stored serially on each track. Each sector may contain 512 or more bytes. The data are preceded by a *sector header* that contains identification (addressing) information used to find the desired sector on the selected track. Following the data, there are additional bits that constitute an *error-correcting code* (ECC). The ECC bits are used to detect and correct errors that may have occurred in writing or reading the data bytes. There is a small *inter-sector gap* that enables the disk control circuitry to distinguish easily between two consecutive sectors.

An unformatted disk has no information on its tracks. The formatting process writes markers that divide the disk into tracks and sectors. During this process, the disk controller may discover some sectors or even whole tracks that are defective. The disk controller keeps a record of such defects and excludes them from use. The formatting information comprises sector headers, ECC bits, and inter-sector gaps. The capacity of a formatted disk, after accounting for the formating information overhead, is the proper indicator of the disk's storage capability. After formatting, the disk is divided into logical partitions.

Figure 8.28 indicates that each track has the same number of sectors, which means that all tracks have the same storage capacity. In this case, the stored information is packed more densely on inner tracks than on outer tracks. It is also possible to increase the storage density by placing more sectors on the outer tracks, which have longer circumference. This would be at the expense of more complicated access circuitry.

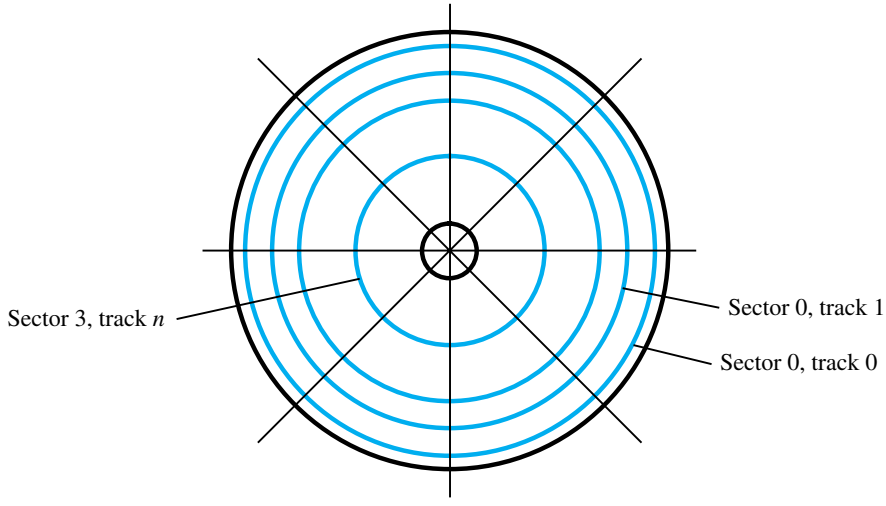


Figure 8.28 Organization of one surface of a disk.

Access Time

There are two components involved in the time delay between the disk receiving an address and the beginning of the actual data transfer. The first, called the *seek time*, is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. Average values are in the 5- to 8-ms range. The second component is the *rotational delay*, also called *latency time*, which is the time taken to reach the addressed sector after the read/write head is positioned over the correct track. On average, this is the time for half a rotation of the disk. The sum of these two delays is called the disk *access time*. If only a few sectors of data are accessed in a single operation, the access time is at least an order of magnitude longer than the time it takes to transfer the data.

Data Buffer/Cache

A disk drive is connected to the rest of a computer system using some standard interconnection scheme, such as SCSI or SATA. The interconnection hardware is usually capable of transferring data at much higher rates than the rate at which data can be read from disk tracks. An efficient way to deal with the possible differences in transfer rates is to include a *data buffer* in the disk unit. The buffer is a semiconductor memory, capable of storing a few megabytes of data. The requested data are transferred between the disk tracks and the buffer at a rate dependent on the rotational speed of the disk. Transfers between the data buffer and the main memory can then take place at the maximum rate allowed by the interconnect between them.

The data buffer in the disk controller can also be used to provide a caching mechanism for the disk. When a Read request arrives at the disk, the controller can first check to see if the desired data are already available in the buffer. If so, the data are transferred to the memory in microseconds instead of milliseconds. Otherwise, the data are read from a disk track in the usual way, stored in the buffer, then transferred to the memory. Because of locality of reference, a subsequent request is likely to refer to data that sequentially follow the data specified in the current request. In anticipation of future requests, the disk controller may read more data than needed and place them into the buffer. When used as a cache, the buffer is typically large enough to store entire tracks of data. So, a possible strategy is to begin transferring the contents of the track into the data buffer as soon as the read/write head is positioned over the desired track.

Disk Controller

Operation of a disk drive is controlled by a *disk controller* circuit, which also provides an interface between the disk drive and the rest of the computer system. One disk controller may be used to control more than one drive.

A disk controller that communicates directly with the processor contains a number of registers that can be read and written by the operating system. Thus, communication between the OS and the disk controller is achieved in the same manner as with any I/O interface, as discussed in Chapter 7. The disk controller uses the DMA scheme to transfer data between the disk and the main memory. Actually, these transfers are from/to the data buffer, which is implemented as a part of the disk controller module. The OS initiates the transfers by issuing Read and Write requests, which entail loading the controller's

registers with the necessary addressing and control information. Typically, this information includes:

Main memory address—The address of the first main memory location of the block of words involved in the transfer.

Disk address—The location of the sector containing the beginning of the desired block of words.

Word count—The number of words in the block to be transferred.

The disk address issued by the OS is a logical address. The corresponding physical address on the disk may be different. For example, bad sectors may be detected when the disk is formatted. The disk controller keeps track of such sectors and maintains the mapping between logical and physical addresses. Normally, a few spare sectors are kept on each track, or on another track in the same cylinder, to be used as substitutes for the bad sectors.

On the disk drive side, the controller's major functions are:

Seek—Causes the disk drive to move the read/write head from its current position to the desired track.

Read—Initiates a Read operation, starting at the address specified in the disk address register. Data read serially from the disk are assembled into words and placed into the data buffer for transfer to the main memory. The number of words is determined by the word count register.

Write—Transfers data to the disk, using a control method similar to that for Read operations.

Error checking—Computes the error correcting code (ECC) value for the data read from a given sector and compares it with the corresponding ECC value read from the disk. In the case of a mismatch, it corrects the error if possible; otherwise, it raises an interrupt to inform the OS that an error has occurred. During a Write operation, the controller computes the ECC value for the data to be written and stores this value on the disk.

Floppy Disks

The disks discussed above are known as hard or rigid disk units. *Floppy disks* are smaller, simpler, and cheaper disk units that consist of a flexible, removable, plastic *diskette* coated with magnetic material. The diskette is enclosed in a plastic jacket, which has an opening where the read/write head can be positioned. A hole in the center of the diskette allows a spindle mechanism in the disk drive to position and rotate the diskette.

The main feature of floppy disks is their low cost and shipping convenience. However, they have much smaller storage capacities, longer access times, and higher failure rates than hard disks. In recent years, they have largely been replaced by CDs, DVDs, and flash cards as portable storage media.

RAID Disk Arrays

Processor speeds have increased dramatically. At the same time, access times to disk drives are still on the order of milliseconds, because of the limitations of the mechanical motion involved. One way to reduce access time is to use multiple disks operating in parallel. In 1988, researchers at the University of California-Berkeley proposed such a storage system [5]. They called it RAID, for Redundant Array of Inexpensive Disks. (Since all disks are now inexpensive, the acronym was later reinterpreted as Redundant Array of Independent Disks.) Using multiple disks also makes it possible to improve the reliability of the overall system. Different configurations were proposed, and many more have been developed since.

The basic configuration, known as RAID 0, is simple. A single large file is stored in several separate disk units by dividing the file into a number of smaller pieces and storing these pieces on different disks. This is called *data striping*. When the file is accessed for a Read operation, all disks access their portions of the data in parallel. As a result, the rate at which the data can be transferred is equal to the data rate of individual disks times the number of disks. However, access time, that is, the seek and rotational delay needed to locate the beginning of the data on each disk, is not reduced. Since each disk operates independently, access times vary. Individual pieces of the data are buffered, so that the complete file can be reassembled and transferred to the memory as a single entity.

Various RAID configurations form a hierarchy, with each level in the hierarchy providing additional features. For example, RAID 1 is intended to provide better reliability by storing identical copies of the data on two disks rather than just one. The two disks are said to be mirrors of each other. If one disk drive fails, all Read and Write operations are directed to its mirror drive. Other levels of the hierarchy achieve increased reliability through various parity-checking schemes, without requiring a full duplication of disks. Some also have error-recovery capability.

The RAID concept has gained commercial acceptance. RAID systems are available from many manufacturers for use with a variety of operating systems.

8.10.2 OPTICAL DISKS

Storage devices can also be implemented using optical means. The familiar compact disk (CD), used in audio systems, was the first practical application of this technology. Soon after, the optical technology was adapted to the computer environment to provide a high-capacity read-only storage medium known as a CD-ROM.

The first generation of CDs was developed in the mid-1980s by the Sony and Philips companies. The technology exploited the possibility of using a digital representation for analog sound signals. To provide high-quality sound recording and reproduction, 16-bit samples of the analog signal are taken at a rate of 44,100 samples per second. Initially, CDs were designed to hold up to 75 minutes, requiring a total of about 3×10^9 bits (3 gigabits) of storage. Since then, higher-capacity devices have been developed.

CD Technology

The optical technology that is used for CD systems makes use of the fact that laser light can be focused on a very small spot. A laser beam is directed onto a spinning disk,

with tiny indentations arranged to form a long spiral track on its surface. The indentations reflect the focused beam toward a photodetector, which detects the stored binary patterns.

The laser emits a coherent light beam that is sharply focused on the surface of the disk. Coherent light consists of synchronized waves that have the same wavelength. If a coherent light beam is combined with another beam of the same kind, and the two beams are in phase, the result is a brighter beam. But, if the waves of the two beams are 180 degrees out of phase, they cancel each other. Thus, a photodetector can be used to detect the beams. It will see a bright spot in the first case and a dark spot in the second case.

A cross-section of a small portion of a CD is shown in Figure 8.29*a*. The bottom layer is made of transparent polycarbonate plastic, which serves as a clear glass base. The surface of this plastic is programmed to store data by indenting it with *pits*. The unindented parts are called *lands*. A thin layer of reflecting aluminum material is placed on top of a programmed disk. The aluminum is then covered by a protective acrylic. Finally, the topmost layer is deposited and stamped with a label. The total thickness of the disk is 1.2 mm, almost all of it contributed by the polycarbonate plastic. The other layers are very thin.

The laser source and the photodetector are positioned below the polycarbonate plastic. The emitted beam travels through the plastic layer, reflects off the aluminum layer, and travels back toward the photodetector. Note that from the laser side, the pits actually appear as bumps rising above the lands.

Figure 8.29*b* shows what happens as the laser beam scans across the disk and encounters a transition from a pit to a land. Three different positions of the laser source and the detector are shown, as would occur when the disk is rotating. When the light reflects solely from a pit, or from a land, the detector sees the reflected beam as a bright spot. But, a different situation arises when the beam moves over the edge between a pit and the adjacent land. The pit is one quarter of a wavelength closer to the laser source. Thus, the reflected beams from the pit and the adjacent land will be 180 degrees out of phase, cancelling each other. Hence, the detector will not see a reflected beam at pit-land and land-pit transitions, and will detect a dark spot.

Figure 8.29*c* depicts several transitions between lands and pits. If each transition, detected as a dark spot, is taken to denote the binary value 1, and the flat portions represent 0s, then the detected binary pattern will be as shown in the figure. This pattern is not a direct representation of the stored data. CDs use a complex encoding scheme to represent data. Each byte of data is represented by a 14-bit code, which provides considerable error detection capability. We will not delve into details of this code.

The pits are arranged on a long track on the surface of the disk, spiraling from the middle of the disk toward the outer edge. But, it is customary to refer to each circular path spanning 360 degrees as a separate track, which is analogous to the terminology used for magnetic disks. The CD is 120 mm in diameter, with a 15-mm hole in the center. The tracks cover the area from a 25-mm radius to a 58-mm radius. The space between the tracks is 1.6 microns. Pits are 0.5 microns wide and 0.8 to 3 microns long. There are more than 15,000 tracks on a disk. If the entire track spiral were unraveled, it would be over 5 km long!

CD-ROM

Since CDs store information in a binary form, they are suitable for use as a storage medium in computer systems. The main challenge is to ensure the integrity of stored data.

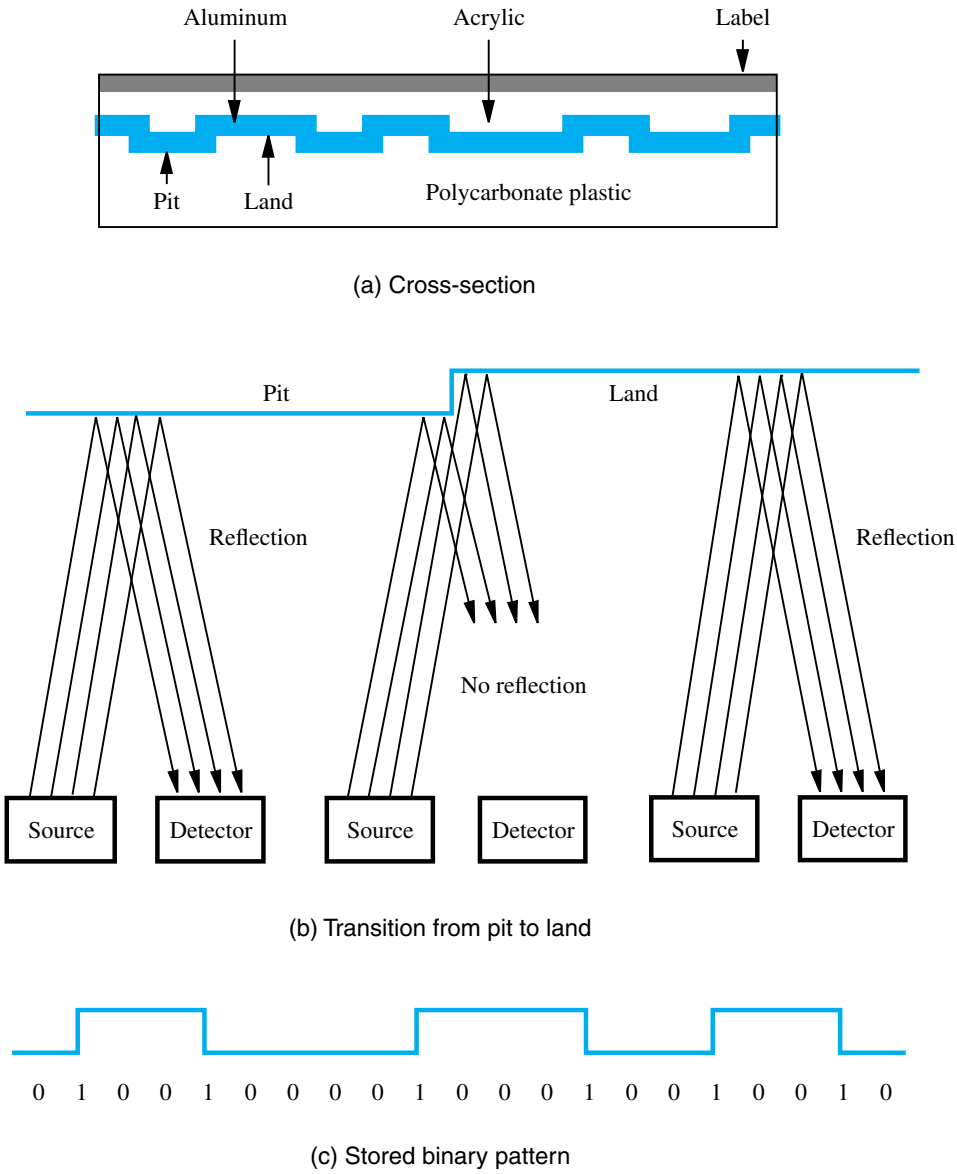


Figure 8.29 Optical disk.

Because the pits are very small, it is difficult to implement all of the pits perfectly. In audio and video applications, some errors in the data can be tolerated, because they are unlikely to affect the reproduced sound or image in a perceptible way. However, such errors are not acceptable in computer applications. Since physical imperfections cannot be avoided, it is

necessary to use additional bits to provide error detection and correction capability. The CDs used to store computer data are called *CD-ROMs*, because, like semiconductor ROM chips, their contents can only be read.

Stored data are organized on CD-ROM tracks in the form of blocks called *sectors*. There are several different formats for a sector. One format, known as Mode 1, uses 2352-byte sectors. There is a 16-byte header that contains a synchronization field used to detect the beginning of the sector and addressing information used to identify the sector. This is followed by 2048 bytes of stored data. At the end of the sector, there are 288 bytes used to implement the error-correcting scheme. The number of sectors per track is variable; there are more sectors on the longer outer tracks. With the Mode 1 format, a CD-ROM has a storage capacity of about 650 Mbytes.

Error detection and correction is done at more than one level. As mentioned earlier, each byte of information stored on a CD is encoded using a 14-bit code that has some error-correcting capability. This code can correct single-bit errors. Errors that occur in short bursts, affecting several bits, are detected and corrected using the error-checking bits at the end of the sector.

CD-ROM drives operate at a number of different rotational speeds. The basic speed, known as 1X, is 75 sectors per second. This provides a data rate of 153,600 bytes/s (150 Kbytes/s), using the Mode 1 format. Higher speed CD-ROM drives are identified in relation to the basic speed. Thus, a 56X CD-ROM has a data transfer rate that is 56 times that of the 1X CD-ROM, or about 6 Mbytes/s. This transfer rate is considerably lower than the transfer rates of magnetic hard disks, which are in the range of tens of megabytes per second. Another significant difference in performance is the seek time, which in CD-ROMs may be several hundred milliseconds. So, in terms of performance, CD-ROMs are clearly inferior to magnetic disks. Their attraction lies in their small physical size, low cost, and ease of handling as a removable and transportable mass-storage medium. As a result, they are widely used for the distribution of software, textbooks, application programs, video games, and so on.

CD-Recordable

The CDs described above are read-only devices, in which the information is stored at the time of manufacture. First, a master disk is produced using a high-power laser to burn holes that correspond to the required pits. A mold is then made from the master disk, which has bumps in the place of holes. Copies are made by injecting molten polycarbonate plastic into the mold to make CDs that have the same pattern of holes (pits) as the master disk. This process is clearly suitable only for volume production of CDs containing the same information.

A new type of CD was developed in the late 1990s on which data can be easily recorded by a computer user. It is known as CD-Recordable (CD-R). A shiny spiral track covered by an organic dye is implemented on a disk during the manufacturing process. Then, a laser in a CD-R drive burns pits into the organic dye. The burned spots become opaque. They reflect less light than the shiny areas when the CD is being read. This process is irreversible, which means that the written data are stored permanently. Unused portions of a disk can be used to store additional data at a later time.

CD-Rewritable

The most flexible CDs are those that can be written multiple times by the user. They are known as CD-RWs (CD-ReWritables).

The basic structure of CD-RWs is similar to the structure of CD-Rs. Instead of using an organic dye in the recording layer, an alloy of silver, indium, antimony, and tellurium is used. This alloy has interesting and useful behavior when it is heated and cooled. If it is heated above its melting point (500 degrees C) and then cooled down, it goes into an amorphous state in which it absorbs light. But, if it is heated only to about 200 degrees C and this temperature is maintained for an extended period, a process known as *annealing* takes place, which leaves the alloy in a crystalline state that allows light to pass through. If the crystalline state represents land area, pits can be created by heating selected spots past the melting point. The stored data can be erased using the annealing process, which returns the alloy to a uniform crystalline state. A reflective material is placed above the recording layer to reflect the light when the disk is read.

A CD-RW drive uses three different laser powers. The highest power is used to record the pits. The middle power is used to put the alloy into its crystalline state; it is referred to as the "erase power." The lowest power is used to read the stored information.

CD drives designed to read and write CD-RW disks can usually be used with other compact disk media. They can read CD-ROMs and can read and write CD-Rs. They are designed to meet the requirements of standard interconnection interfaces, such as SATA and USB.

CD-RW disks provide low-cost storage media. They are suitable for archival storage of information that may range from databases to photographic images. They can be used for low-volume distribution of information, just like CD-Rs, and for backup purposes. The CD-RW technology has made CD-Rs less relevant because it offers superior capability at only slightly higher cost.

DVD Technology

The success of CD technology and the continuing quest for greater storage capability has led to the development of DVD (Digital Versatile Disk) technology. The first DVD standard was defined in 1996 by a consortium of companies, with the objective of being able to store a full-length movie on one side of a DVD disk.

The physical size of a DVD disk is the same as that of CDs. The disk is 1.2 mm thick, and it is 120 mm in diameter. Its storage capacity is made much larger than that of CDs by several design changes:

- A red-light laser with a wavelength of 635 nm is used instead of the infrared light laser used in CDs, which has a wavelength of 780 nm. The shorter wavelength makes it possible to focus the light to a smaller spot.
- Pits are smaller, having a minimum length of 0.4 micron.
- Tracks are placed closer together; the distance between tracks is 0.74 micron.

Using these improvements leads to a DVD capacity of 4.7 Gbytes.

Further increases in capacity have been achieved by going to two-layered and two-sided disks. The single-layered single-sided disk, defined in the standard as DVD-5, has a structure

that is almost the same as the CD in Figure 8.29*a*. A double-layered disk makes use of two layers on which tracks are implemented on top of each other. The first layer is the clear base, as in CD disks. But, instead of using reflecting aluminum, the lands and pits of this layer are covered by a translucent material that acts as a semi-reflector. The surface of this material is then also programmed with indented pits to store data. A reflective material is placed on top of the second layer of pits and lands. The disk is read by focusing the laser beam on the desired layer. When the beam is focused on the first layer, sufficient light is reflected by the translucent material to detect the stored binary patterns. When the beam is focused on the second layer, the light reflected by the reflective material corresponds to the information stored on this layer. In both cases, the layer on which the beam is not focused reflects a much smaller amount of light, which is eliminated by the detector circuit as noise. The total storage capacity of both layers is 8.5 Gbytes. This disk is called DVD-9 in the standard.

Two single-sided disks can be put together to form a sandwich-like structure where the top disk is turned upside down. This can be done with single-layered disks, as specified in DVD-10, giving a composite disk with a capacity of 9.4 Gbytes. It can also be done with the double-layered disks, as specified in DVD-18, yielding a capacity of 17 Gbytes.

Access times for DVD drives are similar to CD drives. However, when the DVD disks rotate at the same speed, the data transfer rates are much higher because of the higher density of pits. Rewritable versions of DVD devices have also been developed, providing large storage capacities.

8.10.3 MAGNETIC TAPE SYSTEMS

Magnetic tapes are suited for off-line storage of large amounts of data. They are typically used for backup purposes and for archival storage. Magnetic-tape recording uses the same principle as magnetic disks. The main difference is that the magnetic film is deposited on a very thin 0.5- or 0.25-inch wide plastic tape. Seven or nine bits (corresponding to one character) are recorded in parallel across the width of the tape, perpendicular to the direction of motion. A separate read/write head is provided for each bit position on the tape, so that all bits of a character can be read or written in parallel. One of the character bits is used as a parity bit.

Data on the tape are organized in the form of *records* separated by gaps, as shown in Figure 8.30. Tape motion is stopped only when a record gap is underneath the read/write heads. The record gaps are long enough to allow the tape to attain its normal speed before the beginning of the next record is reached. If a coding scheme such as that in Figure 8.27*c* is used for recording data on the tape, record gaps are identified as areas where there is no change in magnetization. This allows record gaps to be detected independently of the recorded data. To help users organize large amounts of data, a group of related records is called a *file*. The beginning of a file is identified by a *file mark*, as shown in Figure 8.30. The file mark is a special single- or multiple-character record, usually preceded by a gap longer than the inter-record gap. The first record following a file mark can be used as a *header* or *identifier* for the file. This allows the user to search a tape containing a large number of files for a particular file.

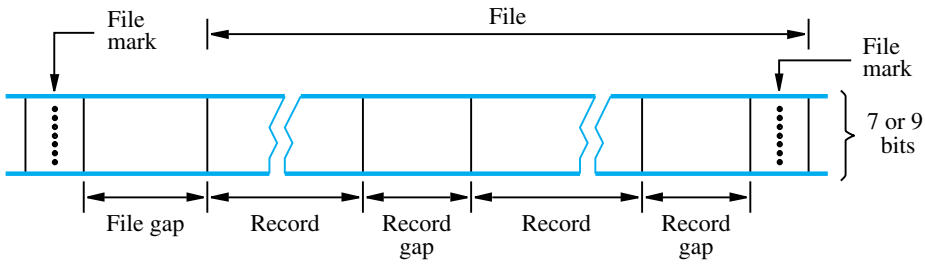


Figure 8.30 Organization of data on magnetic tape.

Cartridge Tape System

Tape systems have been developed for backup of on-line disk storage. One such system uses an 8-mm video-format tape housed in a cassette. These units are called cartridge tapes. They have capacities in the range of 2 to 5 gigabytes and handle data transfers at the rate of a few hundred kilobytes per second. Reading and writing is done by a helical scan system operating across the tape, similar to that used in video cassette tape drives. Bit densities of tens of millions of bits per square inch are achievable. Multiple-cartridge systems are available that automate the loading and unloading of cassettes so that tens of gigabytes of on-line storage can be backed up unattended.

8.11 CONCLUDING REMARKS

The design of the memory hierarchy is critical to the performance of a computer system. Modern operating systems and application programs place heavy demands on both the capacity and speed of the memory. In this chapter, we presented the most important technological and organizational details of memory systems and how they have evolved to meet these demands.

Developments in semiconductor technology have led to significant improvements in the speed and capacity of memory chips, accompanied by a large decrease in the cost per bit. The performance of computer memories is enhanced further by the use of a memory hierarchy. Today, a large yet affordable main memory is implemented with dynamic memory chips. One or more levels of cache memory are always provided. The introduction of the cache memory reduces significantly the effective memory access time seen by the processor. Virtual memory makes the main memory appear larger than the physical memory.

Magnetic disks continue to be the primary technology for secondary storage. They provide enormous storage capacity, reaching and exceeding a trillion bytes on a single drive, with a very low cost per bit. But, flash semiconductor technology is beginning to compete effectively in some applications.

8.12 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 8.2 **Problem:** Describe a structure similar to the one in Figure 8.10 for an 8M $\times$ 32 memory using 512K $\times$ 8 memory chips.

Solution: The required structure is essentially the same as in Figure 8.10, except that 16 rows are needed, each with four 512 $\times$ 8 chips. Address lines A_{18-0} should be connected to all chips. Address lines A_{22-19} should be connected to a 4-bit decoder to select one of the 16 rows.

Example 8.3 **Problem:** A computer system uses 32-bit memory addresses and it has a main memory consisting of 1G bytes. It has a 4K-byte cache organized in the block-set-associative manner, with 4 blocks per set and 64 bytes per block.

(*a*) Calculate the number of bits in each of the Tag, Set, and Word fields of the memory address.

(*b*) Assume that the cache is initially empty. Suppose that the processor fetches 1088 words of four bytes each from successive word locations starting at location 0. It then repeats this fetch sequence nine more times. If the cache is 10 times faster than the memory, estimate the improvement factor resulting from the use of the cache. Assume that the LRU algorithm is used for block replacement.

Solution: Consecutive addresses refer to bytes.

(*a*) A block has 64 bytes; hence the Word field is 6 bits long. With $4 \times 64 = 256$ bytes in a set, there are $4K/256 = 16$ sets, requiring a Set field of 4 bits. This leaves $32 - 4 - 6 = 22$ bits for the Tag field.

(*b*) The 1088 words constitute 68 blocks, occuping blocks 0 to 67 in the memory. The cache has space for 64 blocks. Hence, after blocks 0, 1, 2, . . . , 63 have been read from the memory into the cache on the first pass, the cache is full. The next four blocks, numbered 64 to 67, map to sets 0, 1, 2, and 3. Each of them will replace the least recently used cache block in its set, which is block 0. During the second pass, memory block 0 has to be reloaded into set 0 of the cache, since it has been overwritten by block 64. It will be placed in the least recently used block of set 0 at that point, which is block 1. Next, memory blocks 1, 2, and 3 will replace block 1 of sets 1, 2 and 3 in the cache, respectively. Memory blocks 4 to 15 will be found in the cache. Memory blocks 16 to 19, which were in block location 1 of sets 0 to 3, have now been overwritten, and will be reloaded in block location 2 of these sets.

As execution proceeds, all memory blocks that occupy the first four of the 16 cache sets are always overwritten before they can be used on a succeeding pass. Memory blocks 0, 16, 32, 48, and 64 continually displace each other as they compete for the 4 block positions in cache set 0. The same thing occurs in cache set 1 (memory blocks 1, 17, 33, 49, 65), cache set 2 (memory blocks 2, 18, 34, 50, 66), and cache set 3 (memory blocks 3, 19, 35, 51, 67). Memory blocks that occupy the last 12 sets (sets 4 through 15) are fetched once on the first pass and remain in the cache for the next 9 passes.

In summary, on the first pass, all 68 blocks of the loop are fetched from the memory. On each of the 9 successive passes, 48 blocks are found in sets 4 through 15 of the cache, and the remaining 20 blocks must be fetched from the memory. Let τ be the access time of the cache. Therefore,

$$\begin{aligned}\text{Improvement factor} &= \frac{\text{Time without cache}}{\text{Time with cache}} \\ &= \frac{10 \times 68 \times 10\tau}{1 \times 68 \times 11\tau + 9(20 \times 11\tau + 48\tau)} \\ &= 2.15\end{aligned}$$

This example illustrates a weakness of the LRU algorithm during the execution of program loops. See Problem 8.9 for the performance of an alternative algorithm in this case.

Example 8.4

Problem: Suppose that a computer has a processor with two L1 caches, one for instructions and one for data, and an L2 cache. Let τ be the access time for the two L1 caches. The miss penalties are approximately 15τ for transferring a block from L2 to L1, and 100τ for transferring a block from the main memory to L2. For the purpose of this problem, assume that the hit rates are the same for instructions and data and that the hit rates in the L1 and L2 caches are 0.96 and 0.80, respectively.

(*a*) What fraction of accesses miss in both the L1 and L2 caches, thus requiring access to the main memory?

(*b*) What is the average access time as seen by the processor?

(*c*) Suppose that the L2 cache has an ideal hit rate of 1. By what factor would this reduce the average memory access time as seen by the processor?

(*d*) Consider the following change to the memory hierarchy. The L2 cache is removed and the size of the L1 caches is increased so that their miss rate is cut in half. What is the average memory access time as seen by the processor in this case?

Solution: The average memory access time with one cache level is given in Section 8.7.1 as

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

With L1 and L2 caches, the average memory access time is given in Section 8.7.2 as

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

(*a*) The fraction of memory accesses that miss in both the L1 and L2 caches is

$$(1 - h_1)(1 - h_2) = (1 - 0.96)(1 - 0.80) = 0.008$$

(*b*) The average memory access time using two cache levels is

$$\begin{aligned} t_{avg} &= 0.96\tau + 0.04(0.80 \times 15\tau + 0.20 \times 100\tau) \\ &= 2.24\tau \end{aligned}$$

(*c*) With no misses in the L2 cache, we get:

$$t_{avg}(\text{ideal}) = 0.96\tau + 0.04 \times 15\tau = 1.56\tau$$

Therefore,

$$\frac{t_{avg}(\text{actual})}{t_{avg}(\text{ideal})} = \frac{2.24\tau}{1.56\tau} = 1.44$$

(*d*) With larger L1 caches and the L2 cache removed, the access time is

$$t_{avg} = 0.98\tau + 0.02 \times 100\tau = 2.98\tau$$

Example 8.5 **Problem:** A 1024 × 1024 array of 32-bit numbers is to be normalized as follows. For each column, the largest element is found and all elements of the column are divided by the value of this element. Assume that each page in the virtual memory consists of 4K bytes, and that 1M bytes of the main memory are allocated for storing array data during this computation. Assume that it takes 10 ms to load a page from the disk into the main memory when a page fault occurs.

(*a*) Assume that the array is processed one column at a time. How many page faults would occur and how long does it take to complete the normalization process if the elements of the array are stored in column order in the virtual memory?

(*b*) Repeat part (*a*) assuming the elements are stored in row order?

(*c*) Propose an alternative way for processing the array to reduce the number of page faults when the array is stored in the memory in row order. Estimate the number of page faults and the time needed for your solution.

Solution: Each 32-bit number comprises 4 bytes. Hence, each page holds 1024 numbers. There is space for 256 pages in the 1M-byte portion of the main memory that is allocated for storing data during the computation.

(*a*) Each column is stored in one page; there is a page fault to bring each column to the main memory, for a total of 1024 page faults.

Processing time $= 1024 \times 10$ ms $= 10.24$ s.

(*b*) Processing of each column requires two passes, the first to find the largest element and the second to perform the normalization. When processing the first column, each element access results in a page fault that brings all elements of the corresponding row into the main memory. After 256 elements have been examined, the main memory is full. Accessing the next 256 elements results in page faults that replace all the data in the memory, and the process repeats. Thus, a page fault occurs for every access to every element in the array.

Processing time $= 2 \times 1024 \times 1024 \times 10$ ms $= 20{,}972$ s $= 5.8$ hours.

(*c*) A more efficient alternative for this arrangement of the data is to complete the first pass for only one quarter of each column for all columns, then process the second quarter, and so on. The second pass is handled in the same way. In this case, each pass through the array results in 1024 page faults, for a total of 2048.

Processing time $= 2048 \times 10$ ms $= 20.48$ s.

This example illustrates how the number of page faults can increase dramatically in some cases when the size of the main memory is insufficient for the application. This behavior is called *thrashing*.

Example 8.6

Problem: Consider a long sequence of accesses to a disk with an average seek time of 6 ms and an average rotational delay of 3 ms. The average size of a block being accessed is 8K bytes. The data transfer rate from the disk is 34 Mbytes/sec.

(*a*) Assuming that the data blocks are randomly located on the disk, estimate the average percentage of the total time occupied by seek operations and rotational delays.

(*b*) Repeat part (*a*) for the situation in which disk accesses are arranged so that in 90 percent of the cases, the next access will be to a data block on the same cylinder.

Solution: It takes $8\text{K}/34\text{M} = 0.23$ ms to transfer a block of data.

(*a*) The total time needed to access each block is $6 + 3 + 0.23 = 9.23$ ms. The portion of time occupied by seek and rotational delay is $9/9.23 = 0.97 = 97\%$.

(*b*) In 90% of the cases, only rotational delays are involved. Therefore, the average time to access a block is $0.9 \times 3 + 0.1 \times 9 + 0.23 = 3.89$ ms. The portion of time occupied by seek and rotational delay is $3.6/3.89 = 0.92 = 92\%$.

PROBLEMS

8.1 **[M]** Consider the dynamic memory cell of Figure 8.6. Assume that $C = 30$ femtofarads (10^{-15} F) and that leakage current through the transistor is about 0.25 picoamperes (10^{-12} A). The voltage across the capacitor when it is fully charged is 1.5 V. The cell must be refreshed before this voltage drops below 0.9 V. Estimate the minimum refresh rate.

8.2 **[M]** Consider a main memory built with SDRAM chips. Data are transferred in bursts as shown in Figure 8.9, except that the burst length is 8. Assume that 32 bits of data are transferred in parallel. If a 400-MHz clock is used, how much time does it take to transfer:

(*a*) 32 bytes of data

(*b*) 64 bytes of data

What is the latency in each case?

8.3 **[E]** Describe a structure similar to that in Figure 8.10 for a 16M $\times$ 32 memory using 1M $\times$ 4 memory chips.

8.4 **[E]** Give a critique of the following statement: "Using a faster processor chip results in a corresponding increase in performance of a computer even if the main memory speed remains the same."

8.5 **[M]** The memory of a computer is byte-addressable, and the word length is 32 bits. A program consists of two nested loops—a small inner loop and a much larger outer loop. The general structure of the program is given in Figure P8.1. The decimal memory addresses shown delineate the location of the two loops and the beginning and end of the total program. All memory locations in the various sections of the program, 8-52, 56-136, 140-240, and so on, contain instructions to be executed in straight-line sequencing. The program is to be run on a computer that has an instruction cache organized in the direct-mapped manner (see Figure 8.16) with the following parameters:

Cache size	1K bytes
Block size	128 bytes

The miss penalty in the instruction cache is 80τ, where τ is the access time of the cache. Compute the total time needed for instruction fetching during execution of the program in Figure P8.1.

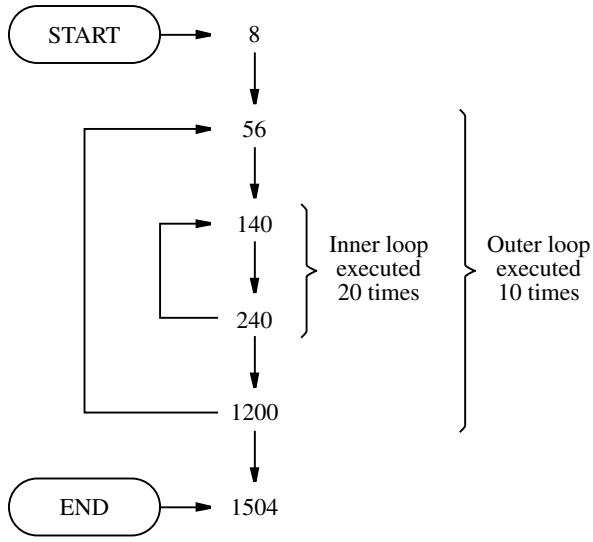


Figure P8.1 A program structure for Problem 8.5.

8.6 **[M]** A computer with a 16-bit word length has a direct-mapped cache, used for both instructions and data. Memory addresses are 16 bits long, and the memory is byte-addressable. The cache is small for illustrative purposes. It contains only four 16-bit words. Each word constitutes a cache block and has an associated 13-bit tag, as shown in Figure P8.2*a*. Words are accessed in the cache using the low-order 3 bits of an address. When a miss occurs during a Read operation for either an instruction or a data operand, the requested word is read from the main memory and sent to the processor. At the same time, it is copied into the cache, and its block number is stored in the associated tag. Consider the following short loop, in which all instructions are 16 bits long:

```
LOOP:   Add        R0, (R1)+
        Decrement  R2
        BNE        LOOP
```

Assume that, before this loop is entered, registers R0, R1, and R2 contain 0, 054E, and 3, respectively. Also assume that the main memory contains the data shown in Figure P8.2*b*, where all entries are given in hexadecimal notation. The loop starts at location LOOP = 02EC. The Autoincrement address mode in the Add instruction is used to access successive numbers in a 3-number list and add them into register R0. The counter register, R2, is decremented until it reaches 0, at which point an exit is made from the loop.

(*a*) Starting with an empty cache, show the contents of the cache, including the tags, at the end of each pass through the loop.

(*b*) Assume that the access times of the cache and the main memory are τ and 10τ, respectively. Calculate the execution time for each pass, counting only memory access times.

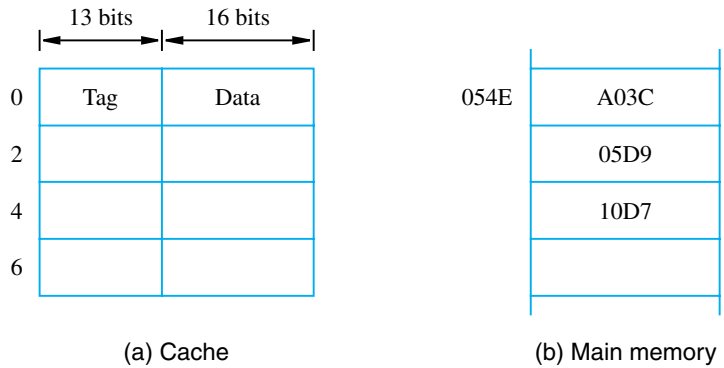


Figure P8.2 Cache and main memory contents in Problem 8.6.

8.7 **[M]** Repeat Problem 8.6 assuming that only instructions are stored in the cache. Data operands are fetched directly from the main memory and not copied into the cache. Why does this choice lead to faster execution than when both instructions and data are loaded into the cache?

8.8 **[E]** A block-set-associative cache consists of a total of 64 blocks, divided into 4-block sets. The main memory contains 4096 blocks, each consisting of 32 words. Assuming a 32-bit byte-addressable address space, how many bits are there in each of the Tag, Set, and Word fields?

8.9 **[M]** Consider the cache in Example 8.3. Assume that whenever a block is to be brought from the main memory and the corresponding set in the cache is full, the new block replaces the most recently used block of this set. Derive the solution for part (*b*) in this case.

8.10 **[D]** Section 8.6.3 illustrates the effect of different cache-mapping techniques, using the program in Figure 8.20. Suppose that this program is changed so that in the second loop the elements are handled in the same order as in the first loop; that is, the control for the second loop is specified as

for $i := 0$ **to** 9 **do**

Derive the equivalents of Figures 8.21 through 8.23 for this program. What conclusions can be drawn from this exercise?

8.11 **[M]** A byte-addressable computer has a small data cache capable of holding eight 32-bit words. Each cache block consists of one 32-bit word. When a given program is executed, the processor reads data sequentially from the following hex addresses:

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

This pattern is repeated four times.

(*a*) Assume that the cache is initially empty. Show the contents of the cache at the end of each pass through the loop if a direct-mapped cache is used, and compute the hit rate.

(*b*) Repeat part (*a*) for an associative-mapped cache that uses the LRU replacement algorithm.

(*c*) Repeat part (*a*) for a four-way set-associative cache.

8.12 **[M]** Repeat Problem 8.11, assuming that each cache block consists of two 32-bit words. For part (*c*), use a two-way set-associative cache that uses the LRU replacement algorithm.

8.13 **[E]** The cache block size in many computers is in the range of 32 to 128 bytes. What would be the main advantages and disadvantages of making the size of cache blocks larger or smaller?

8.14 **[M]** A computer has two cache levels L1 and L2. Plot two graphs for the average memory access time (y-axis) versus hit rate h_1 (x-axis) for the two values $h_2 = 0.75$ and $h_2 = 0.85$. Use the values 0.90, 0.92, 0.94, and 0.96, for h_1. Assume that the miss penalties are 15τ and 100τ for the L1 and L2 caches, respectively, where τ is the access time of the L1 caches.

8.15 **[E]** Consider the two-level cache described in Example 8.4. The average access time is given in the solution to part (*b*) of the example as 2.24τ. What value for h_1 would be needed to reduce t_{avg} to 1.5τ, if all other parameters are the same as in the example? Can the same result be achieved by improving the hit rate of L2?

8.16 **[E]** Consider the following analogy for the concept of caching. A serviceman comes to a house to repair the heating system. He carries a toolbox that contains a number of tools that he has used recently in similar jobs. He uses these tools repeatedly, until he reaches a point where other tools are needed. It is likely that he has the required tools in his truck outside the house. But, if the needed tools are not in the truck, he must go to his shop to get them.

Suppose we argue that the toolbox, the truck, and the shop correspond to the L1 cache, the L2 cache, and the main memory of a computer. How good is this analogy? Discuss its correct and incorrect features.

8.17 **[E]** The purpose of using an L2 cache is to reduce the miss penalty of the L1 cache, and in turn to reduce the memory access time as seen by the processor. An alternative is to increase the size of the L1 cache to increase its hit rate. What limits the utility of this approach?

8.18 **[M]** Give a critique of the assumption made in Example 8.1, in Section 8.7.1, that the miss penalty is the same for both read and write accesses. Consider both the write-through and write-back cases, as described in Section 8.6, in formulating your answer.

8.19 **[M]** Consider a computer system in which the available pages in the physical memory are divided among several application programs. The operating system monitors the page transfer activity and dynamically adjusts the number of pages allocated to various programs. Suggest a suitable strategy that the operating system can use to minimize the overall rate of page transfers.

8.20 **[M]** In a computer with a virtual-memory system, the execution of an instruction may be interrupted by a page fault. What state information has to be saved so that this instruction can be resumed later? Note that bringing a new page into the main memory involves a DMA transfer, which requires execution of other instructions. Is it simpler to abandon the interrupted instruction and completely re-execute it later? Can this be done?

8.21 **[E]** When a program generates a reference to a page that does not reside in the physical main memory, execution of the program is suspended until the requested page is loaded into the main memory from a disk. What difficulties might arise when an instruction in one page has an operand in a different page? What capabilities must the processor have to handle this situation?

8.22 **[M]** A disk unit has 24 recording surfaces. It has a total of 14,000 cylinders. There is an average of 400 sectors per track. Each sector contains 512 bytes of data.

(*a*) What is the maximum number of bytes that can be stored in this unit?

(*b*) What is the data transfer rate in bytes per second at a rotational speed of 7200 rpm?

(*c*) Using a 32-bit word, suggest a suitable scheme for specifying the disk address.

8.23 **[M]** Consider a long sequence of accesses to a disk with 8 ms average seek time, 3 ms average rotational delay, and a data transfer rate of 60 Mbytes/sec. The average size of a block being accessed is 64 Kbytes. Assume that each data block is stored in contiguous sectors.

(*a*) Assuming that the blocks are randomly located on the disk, estimate the average percentage of the total time occupied by seek operations and rotational delays.

(*b*) Suppose that 20 blocks are transferred in sequence from adjacent cylinders, reducing seek time to 1 ms. The blocks are randomly located on these cylinders. What is the total transfer time?

8.24 **[M]** The average seek time and rotational delay in a disk system are 6 ms and 3 ms, respectively. The rate of data transfer to or from the disk is 30 Mbytes/sec, and all disk accesses are for 8 Kbytes of data, stored in contiguous sectors. Data blocks are stored at random locations on the disk. The disk controller has an 8-Kbyte buffer. The disk controller, the processor, and the main memory are all attached to a single bus. The bus data width is 32 bits, and a single bus transfer to or from the main memory takes 10 nanoseconds.

(*a*) What is the maximum number of disk units that can be simultaneously transferring data to or from the main memory?

(*b*) What percentage of main memory accesses are used by one disk unit, on average, over a long period of time during which a sequence of independent 8-Kbyte transfers takes place?

8.25 **[M]** Magnetic disks are used as the secondary storage for program and data files in most virtual-memory systems. Which disk parameter(s) should influence the choice of page size?

REFERENCES

1. T.C. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching," *Tech. Report CSL-TR-94-628,* Stanford University, Calif., 1994.
2. J.L. Baer and T.F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputing '91*, 1991, pp. 176–186.

3. J.W.C. Fu and J.H. Patel, "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 24th International Symposium on Microarchitecture,* 1992, pp. 102–110.

4. D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th Annual International Symposium on Computer Architecture,* 1981, pp. 81–85.

5. D.A. Patterson, G.A. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988, pp. 109-166.