The Memory System

Introduction

In practice, a *memory system* is a hierarchy of storage devices with different *capacities*, *costs*, and *access times*. CPU *registers* hold the most frequently used data. Small, fast *cache memories* nearby the CPU act as *staging areas* for a subset of the data and instructions stored in the relatively slow *main memory*. The main memory stages data stored on large, *slow disks*, which in turn often serve as staging areas for data stored on the *disks or tapes* of other machines connected by *networks*.

Memory hierarchies work because *well-written programs tend to access the storage at any particular level more frequently* than they access the storage at the next lower level. So the storage at the next level can be slower, and thus larger and cheaper per bit. The overall effect is a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy but that serves data to programs at the rate of the fast storage near the top of the hierarchy.

As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data your program needs are stored in a CPU register, then they can be accessed in 0 cycles during the execution of the instruction. If stored in a cache, 4 to 75 cycles. If stored in main memory, hundreds of cycles. And if stored in disk, tens of millions of cycles!

Here, then, is a fundamental and enduring idea in computer systems: if you understand how the system moves data up and down the memory hierarchy, then you can write your application programs so that their data items are stored higher in the hierarchy, where the CPU can access them more quickly.

This idea centers around a fundamental property of computer programs known as *locality*. Programs with good locality tend to access the same set of data items over and over again, or they tend to access sets of nearby data items. Programs with good locality tend to access more data items from the upper levels of the memory hierarchy than programs with poor locality, and thus run faster.

Storage Technologies

Much of the success of computer technology stems from the tremendous progress in storage technology. Early computers had a few kilobytes of random access memory. The earliest IBM PCs didn't even have a hard disk. That changed with the introduction of the IBM PC-XT in 1982, with its 10-megabyte disk. By the year 2015, typical machines had 300,000 times as much disk storage, and the amount of storage was increasing by a factor of 2 every couple of years.

Memory Terminology

- **Memory Cell.** A device or an electrical circuit used to store a single bit (0 or 1). Examples of memory cells include a *flip-flop*, a *charged capacitor*, and a *single spot* on magnetic tape or disk.
- **Memory Word.** A group of bits (cells) in a memory that represents instructions or data of some type. For example, a register consisting of eight FFs can be considered to be a memory that is storing an eight-bit word. Word sizes in computers typically range from 8 to 64 bits, depending on the size of the computer.
- **byte.** A special term used for a group of eight bits. A byte always consists of eight bits. Word sizes can be expressed in bytes as well as

in bits. For example, a word size of eight bits is also a word size of one byte, a word size of 16 bits is two bytes, and so on.

- **Capacity.** A way of specifying how many bits can be stored in a particular memory device or complete memory system. To illustrate, suppose that we have a memory that can store 4096 20-bit words. This represents a total capacity of 81,920 bits. We could also express this memory's capacity as 4096 X 20. When expressed this way, the first number (4096) is the number of words, and the second number (20) is the number of bits per word (word size). The number of words in a memory is often a multiple of 1024. It is common to use the designation "1K" to represent 1024 = 210 when referring to memory capacity. Thus, a memory that has a storage capacity of 4K X 20 is actually a 4096 X 20 memory. The development of larger memories has brought about the designation "1M" or "1 meg" to represent 220 = 1,048,576. Thus, a memory that has a capacity of 2M X 8 is actually one with a capacity of 2,097,152 X 8. The designation "giga" refers to 2^{30} = 1,073,741,824.

Example:

A certain semiconductor memory chip is specified as 2K X 8. How many words can be stored on this chip? What is the word size? How many total bits can this chip store?

Solution

$$2K = 2 \times 1024 = 2048 \text{ words}$$

Each word is eight bits (one byte). The total number of bits is therefore

$$2048 \times 8 = 16{,}384 \text{ bits}$$

Example:

Which memory stores the most bits: a 5M X 8 memory or a memory that stores 1M words at a word size of 16 bits?

Solution

5M X 8 = 5 X 1,048,576 X 8 = 41,943,040 bits

1M X 16 = 1,048,576 X 16 = 16,777,216 bits

The 5M X 8 memory stores more bits.

- **Density**. Another term for *capacity.* When we say that one memory device has a greater density than another, we mean that it can store more bits in the same amount of space. It is more dense.
- **Address.** A number that identifies the location of a word in memory. Each word stored in a memory device or system has a unique address. Addresses always exist in a digital system as a binary number, although octal, hexadecimal, and decimal numbers are often used to represent the address for convenience. Figure 12-2 illustrates a small memory consisting of eight words. Each of these eight words has a specific address represented as a three-bit number ranging from 000 to 111. Whenever we refer to a specific word location in memory, we use its address code to identify it.

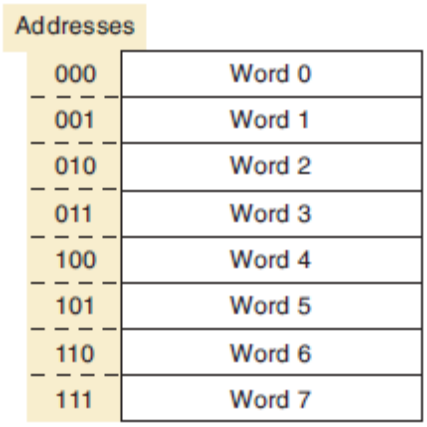


Figure 12-2 Each word location has a specific binary address.

Random Access Memory

Random access memory (RAM)comes in two varieties—*static* and *dynamic*. Static RAM (SRAM) is faster and significantly more expensive than dynamic RAM (DRAM). SRAM is used for *cache memories*, both on and off the CPU chip. DRAM is used for the *main memory* plus the *frame buffer* of a graphics system. Typically, a desktop system will have no more than a few tens of megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

Static RAM

SRAM stores each bit in a bistable memory cell. Each cell is implemented with a six-transistor circuit. This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or states. Any other state will be unstable—starting from there, the circuit will quickly move toward one of the stable states.

Due to its bistable nature, an SRAM memory cell will retain its value indefinitely, as long as it is kept powered. Even when a disturbance, such as electrical noise, perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.

Dynamic RAM

DRAM stores each bit as charge on a capacitor. This capacitor is very small—typically around 30 femtofarads—that is, 30×10^{-15} farads. Recall, however, that a farad is a very large unit of measure. DRAM storage can be made very dense — each cell consists of a capacitor and a single access transistor. Unlike SRAM, however, a DRAM memory cell is very sensitive to any disturbance. When the capacitor voltage is disturbed, it will never recover. Exposure to light rays will cause the capacitor voltages to change. In fact, the sensors in digital cameras and camcorders are essentially arrays of DRAM cells.

Various sources of leakage current cause a DRAM cell to lose its charge within a time period of around 10 to 100 milliseconds. Fortunately, for computers

operating with clock cycle times measured in nanoseconds, this retention time is quite long. The memory system must periodically refresh every bit of memory by reading it out and then rewriting it. Some systems also use error-correcting codes, where the computer words are encoded using a few more bits (e.g., a 64-bit word might be encoded using 72 bits), such that circuitry can detect and correct any single erroneous bit within a word.

Figure below summarizes the characteristics of SRAM and DRAM memory.

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative cost	Applications
SRAM	6	1×	Yes	No	1,000×	Cache memory
DRAM	1	10×	No	Yes	1×	Main memory, frame buffers

Figure 6.2 Characteristics of DRAM and SRAM memory.

SRAM is persistent as long as power is applied. Unlike DRAM, no refresh is necessary. SRAM can be accessed faster than DRAM. SRAM is not sensitive to disturbances such as light and electrical noise. *The trade-off is that SRAM cells use more transistors than DRAM cells and thus have lower densities, are more expensive, and consume more power.*

Conventional DRAMs

The cells (bits) in a DRAM chip are partitioned into ***d*** *supercells*, each consisting of ***w*** DRAM cells. A ***d*** × ***w*** DRAM stores a total of ***dw*** bits of information. The supercells are organized as a rectangular array with ***r*** rows and ***c*** columns, where ***rc = d***. Each supercell has an address of the form ***(i,j),*** where ***i*** denotes the row and ***j*** denotes the column.

For example, Figure 6.3 shows the organization of a 16 × 8 DRAM chip with d = 16 supercells, w = 8 bits per supercell, r = 4 rows, and c = 4 columns. The shaded box denotes the supercell at address (2, 1). Information flows in and out of the chip via external connectors called pins. Each pin carries a 1-bit signal. Figure 6.3 shows two of these sets of pins: eight data pins that can

transfer 1 byte in or out of the chip, and two addr pins that carry two-bit row and column supercell addresses. Other pins that carry control information are not shown.

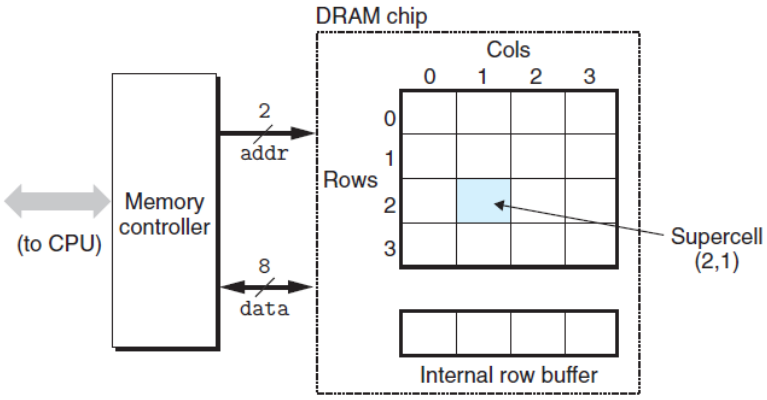


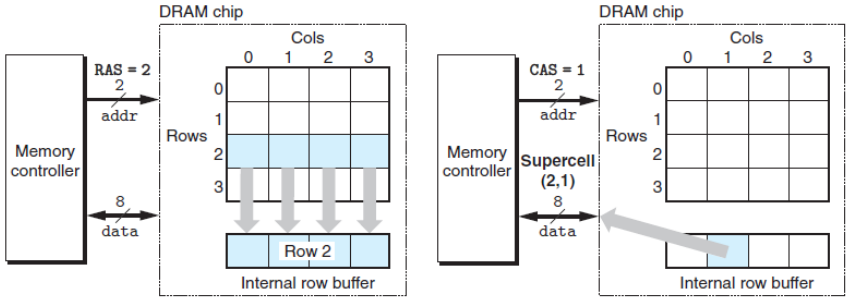
Figure 6.3: High-level view of a 128-bit 16 × 8 DRAM chip.

Each DRAM chip is connected to some circuitry, known as the *memory controller*, that can transfer ***w*** bits at a time to and from each DRAM chip. To read the contents of supercell ***(i,j)***, the memory controller sends the row address ***i*** to the DRAM, followed by the column address ***j***. The DRAM responds by sending the contents of supercell ***(i,j)*** back to the controller. The row address i is called a RAS (*row access strobe*) request. The column address j is called a CAS (*column access strobe*) request. Notice that the RAS and CAS requests share the same DRAM address pins.

For example, to read supercell ***(2,1)*** from the ***16 × 8*** DRAM in Figure 6.3, the memory controller sends row address 2, as shown in Figure 6.4(a). The DRAM responds by copying the entire contents of row 2 into an internal row buffer. Next, the memory controller sends column address 1, as shown in Figure 6.4(b). The DRAM responds by copying the 8 bits in supercell ***(2,1)*** from the row buffer and sending them to the memory controller.

One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays is to *reduce the number of address pins* on the chip. For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the chip would need four address

pins instead of two. The disadvantage of the two-dimensional array organization is that *addresses must be sent in two distinct steps, which increases the access time*.



(a) Select row 2 (RAS request). (b) Select column 1 (CAS request).

Figure 6.4: Reading the contents of a DRAM supercell.

Memory Modules

DRAM chips are packaged in memory modules that plug into expansion slots on the main system board (motherboard). Core i7 systems use the ***240-pin*** dual inline memory module (DIMM), which transfers data to and from the memory controller in ***64-bit*** chunks.

Figure 6.5 shows the basic idea of a memory module. The example module stores a total of 64 MB (megabytes) using eight 64-Mbit 8M × 8 DRAM chips, numbered 0 to 7. Each supercell stores 1 byte of main memory, and each 64-bit word at byte address A in main memory is represented by the eight supercells whose corresponding supercell address is ***(i,j)***. In the example in Figure 6.5, DRAM 0 stores the first (lower-order) byte, DRAM 1 stores the next byte, and so on.

To retrieve the word at memory address A, the memory controller converts A to a supercell address ***(i,j)*** and sends it to the memory module, which then broadcasts i and j to each DRAM. In response, each DRAM outputs the 8-bit contents of its ***(i,j)*** supercell. Circuitry in the module collects these outputs and forms them into a 64-bit word, which it returns to the memory controller.

Main memory can be aggregated by connecting multiple memory modules to the memory controller. In this case, when the controller receives an address ***A***, the controller selects the module ***k*** that contains ***A***, converts A to its ***(i,j)*** form, and sends ***(i,j)*** to module ***k***.

Practice Problem 6.1 (*solution ate the end of the notes*)

In the following, let ***r*** be the number of rows in a DRAM array, ***c*** the number of columns, ***br*** the number of bits needed to address the rows, and ***bc*** the number of bits needed to address the columns. For each of the following DRAMs, determine the power-of-2 array dimensions that minimize max(***br***, ***bc***), the maximum number of bits needed to address the rows or columns of the array.

Organization	r	c	b_r	b_c	$\max(b_r, b_c)$
16 × 1					
16 × 4					
128 × 8					
512 × 4					
1,024 × 4					

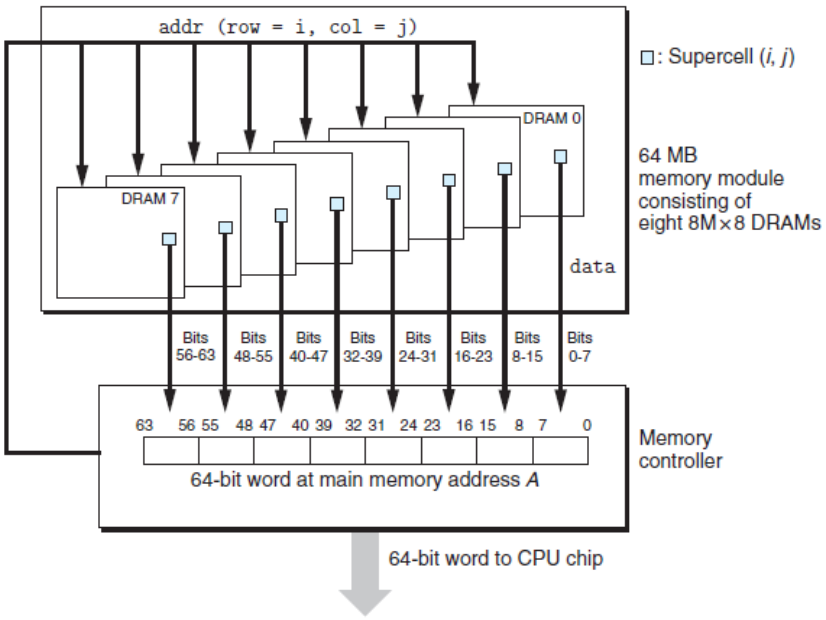


Figure 6.5: Reading the contents of a memory module.

Enhanced DRAMs

There are many kinds of DRAM memories, and new kinds appear on the market with regularity as manufacturers attempt to keep up with rapidly increasing processor speeds. Each is based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed.

(1) ***Fast page mode DRAM (FPM DRAM).*** A conventional DRAM copies an entire row of supercells into its internal row buffer, uses one, and then discards the rest. FPM DRAM improves on this by *allowing consecutive accesses* to the same row to be served directly from the row buffer. For example, to read four supercells from row ***i*** of a conventional DRAM, the memory controller must send four RAS/CAS requests, even though the row address ***i*** is identical in each case. To read supercells from the same row of an FPMDRAM, the memory controller sends an initial RAS/CAS request, followed by three CAS requests. The initial RAS/CAS request copies row ***i*** into the row buffer and returns the supercell addressed by the CAS. The next three supercells are served directly from the row buffer, and thus are returned more quickly than the initial supercell.

(2) ***Extended data out DRAM (EDO DRAM).*** An enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time.

(3) ***Synchronous DRAM (SDRAM)***. Conventional, FPM, and EDO DRAMs are asynchronous in the sense that they communicate with the memory controller using a set of explicit control signals. SDRAM replaces many of these control signals with the rising edges of the same external clock signal that drives the memory controller. Without going into detail, the net effect is that an SDRAM can output the contents of its supercells at a faster rate than its asynchronous counterparts.

(4) ***Double Data-Rate Synchronous DRAM (DDR SDRAM)***. DDR SDRAM is an enhancement of SDRAM that doubles the speed of the DRAM by

using both clock edges as control signals. Different types of DDR SDRAMs are characterized by the size of a small prefetch buffer that increases the effective bandwidth: DDR (2 bits), DDR2 (4 bits), and DDR3 (8 bits).

(5) ***Video RAM (VRAM).*** Used in the frame buffers of graphics systems. VRAM is similar in spirit to FPM DRAM. Two major differences are that.

(i) VRAM output is produced by shifting the entire contents of the internal buffer in sequence and

(ii) VRAM allows concurrent reads and writes to the memory. Thus, the system can be painting the screen with the pixels in the frame buffer (reads) while concurrently writing new values for the next update (writes).

Aside Note: Historical popularity of DRAM technologies

Until 1995, most PCs were built with FPM DRAMs. From 1996 to 1999, EDO DRAMs dominated the market, while FPM DRAMs all but disappeared. SDRAMs first appeared in 1995 in high-end systems, and by 2002 most PCs were built with SDRAMs and DDRSDRAMs. By 2010, most server and desktop systems were built with DDR3 SDRAMs. In fact, the Intel Core i7 supports only DDR3 SDRAM.

Nonvolatile Memory

DRAMs and SRAMs are volatile in the sense that they lose their information if the supply voltage is turned off. Nonvolatile memories, on the other hand, retain their information even when they are powered off. There are a variety of nonvolatile memories. For historical reasons, they are referred to collectively as read-only memories (ROMs), even though some types of ROMs can be written to as well as read. ROMs are distinguished by the number of times they can be reprogrammed (***written to***) and by the mechanism for reprogramming them.

(1) A *programmable ROM (PROM)* can be programmed exactly once. PROMs include a sort of fuse with each memory cell that can be blown once by zapping it with a high current.

(2) An *erasable programmable ROM(EPROM)* has a transparent quartz window that permits light to reach the storage cells. The EPROM cells are cleared to zeros by shining ultraviolet light through the window. Programming an EPROM is done by using a special device to write ones into the EPROM. An EPROM can be erased and reprogrammed on the order of 1,000 times.

(3) An *electrically erasable PROM (EEPROM)* is akin to an EPROM, but it does not require a physically separate programming device, and thus can be reprogrammed in-place on printed circuit cards. An EEPROM can be reprogrammed on the order of 105 times before it wears out.

(4) *Flash memory* is a type of nonvolatile memory, based on EEPROMs, that has become an important storage technology. Flash memories are everywhere, providing fast and durable nonvolatile storage for a slew of electronic devices, including digital cameras, cell phones, and music players, as well as laptop, desktop, and server computer systems. In another Section of this topic, we will look in detail at a new form of flash-based disk drive, known as a solid state disk (SSD), that provides a *faster, sturdier, and less power-hungry* alternative to conventional rotating disks.

Programs stored in ROM devices are often referred to as *firmware*. When a computer system is powered up; it runs firmware stored in a ROM. Some systems provide a small set of primitive input and output functions in firmware—for example, a PC's BIOS (basic input/output system) routines. Complicated devices such as graphics cards and disk drive controllers also rely on firmware to translate I/O (input/output) requests from the CPU.

Accessing Main Memory

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A bus is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can *share the same set of wires* or can *use different sets*. Also, *more than two devices can share the same bus*. The *control wires* carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of an example computer system. The main components are the CPU chip, a chipset that we will call an I/O bridge (which includes the memory controller), and the DRAM memory modules that make up main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O bridge to the main memory. The I/O bridge:

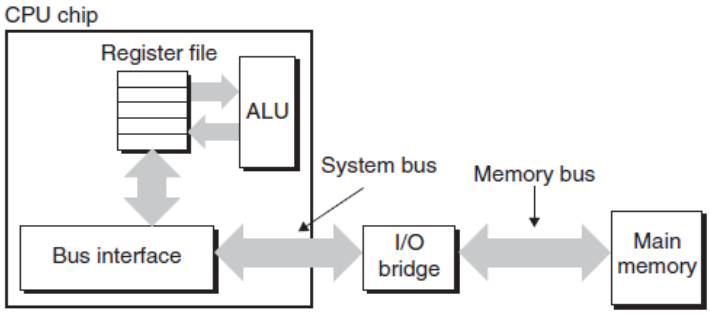


Figure 6.6: Example bus structure that connects the CPU and main memory.

(i) translates the electrical signals of the system bus into the electrical signals of the memory bus.

(ii) It also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards.

Consider what happens when the CPU performs a load operation such as

movq A,%rax

where the contents of address A are loaded into register %rax.

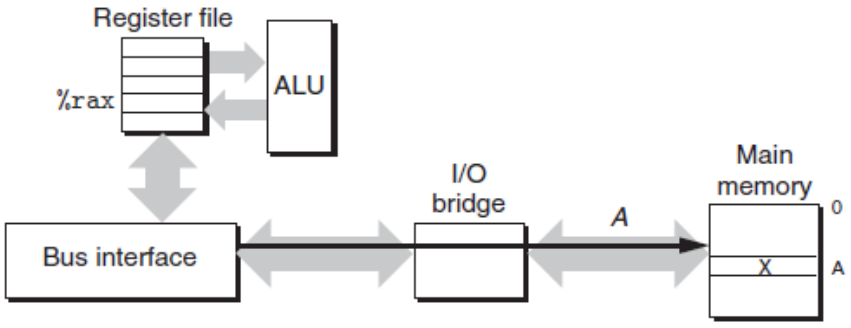
The Read transaction:

The *bus interface initiates* a read transaction on the bus.

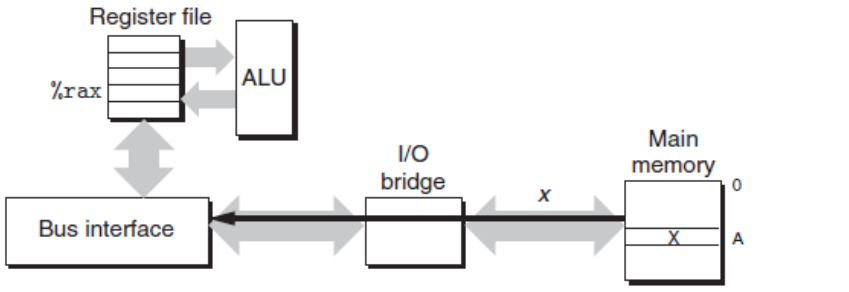
(i) the CPU places the address ***A*** on the system bus. The I/O bridge passes the signal along to the memory bus (Figure 6.7(a)).

(ii) Next, the main memory senses the address signal on the memory bus, reads the address from the memory bus, fetches the data from the DRAM, and writes the data to the memory bus. The I/O bridge translates the memory bus signal into a system bus signal and passes it along to the system bus (Figure 6.7(b)).

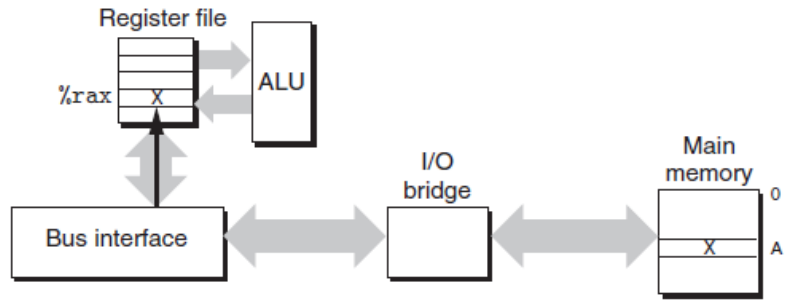
(iii) Finally, the CPU senses the data on the system bus, reads the data from the bus, and copies the data to register %rax (Figure 6.7(c)).



(a) CPU places address *A* on the memory bus.



(b) Main memory reads *A* from the bus, retrieves word *x*, and places it on the bus.



(c) CPU reads word *x* from the bus, and copies it into register %rax.

Figure 6.7: Memory read transaction for a load operation: movq A,%rax.

The Write transaction

Conversely, when the CPU performs a store operation such as

movq %rax,A,

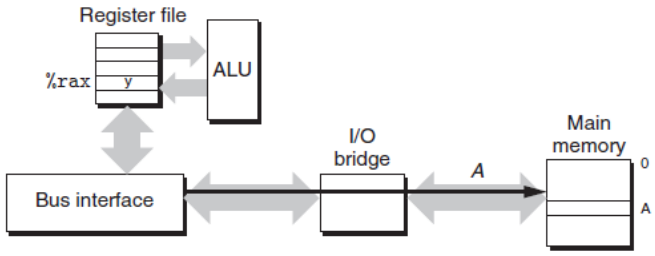
where the contents of register **%rax** are written to address **A**

The CPU initiates a write transaction.

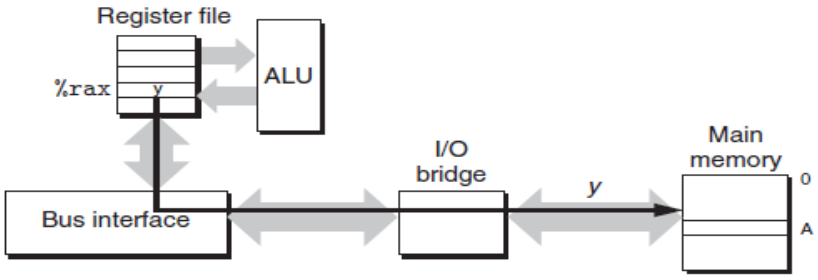
(i) The CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)).

(ii) Next, the CPU copies the data in %rax to the system bus (Figure 6.8(b)).

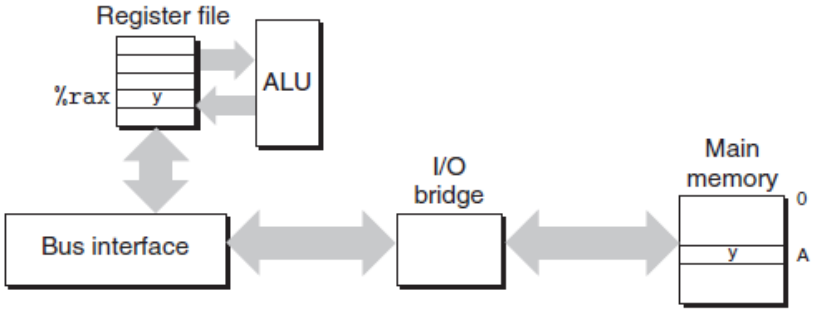
(iii) Finally, the main memory reads the data from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).



(a) CPU places address *A* on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word *y* on the bus.



(c) Main memory reads data word *y* from the bus and stores it at address *A*.

Figure 6.8: Memory write transaction for a store operation: movq %rax,A.

Example:

A certain memory has a capacity of 4K X 8.

(a) How many data input and data output lines does it have?

(b) (How many address lines does it have?

(c) What is its capacity in bytes?

Solution:

(a) Eight of each because the word size is eight.

(b) The memory stores 4K = 4 * 1024 = 4096 words. Thus, there are 4096 memory addresses. Because 4096 = 212, it requires a 12-bit address code to specify one of 4096 addresses.

(c) A byte is eight bits. This memory has a capacity of 4096 bytes.

Exercises

(1) A certain memory has a capacity of 256K X 32. How many words does it store? What is the number of bytes per word? How many memory cells does it contain?

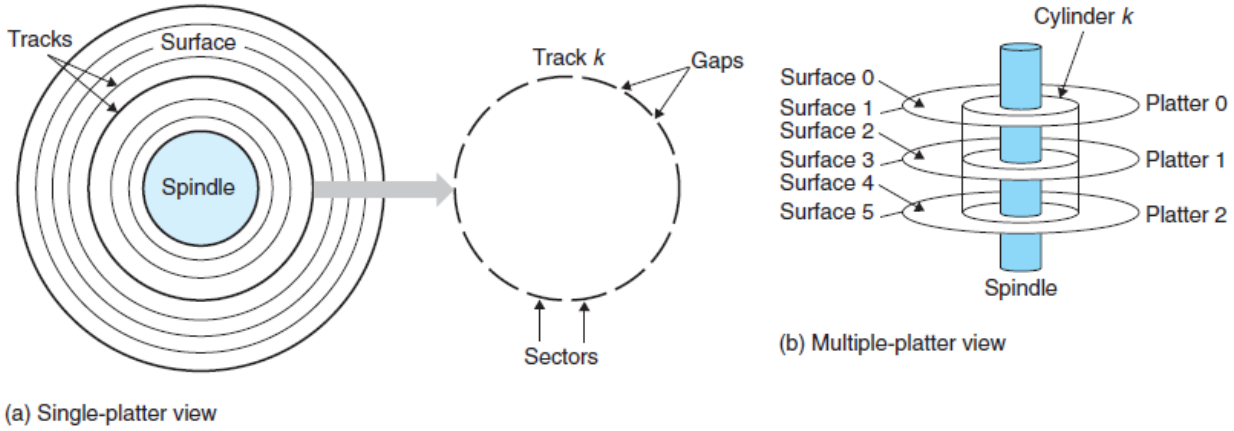
(2) If the memory of Problem (**1**) is to be doubled with the word size remaining the same, how many address lines will be required for it?

(3) A certain memory uses 28 bits to store addresses and 32 bits to store each data word. Find the capacity of the memory.

(4) A certain memory stores 128K eight-bit words. How many data input and data output lines does it have? How many address lines does it have? What is its capacity in bytes?

Disk Geometry

- Disks are constructed from ***platters***.
- Each platter consists of ***two sides***, or ***surfaces***, that are coated with magnetic recording material.
- A rotating ***spindle*** in the center of the platter spins the platter at a fixed rotational rate, typically between 5,400 and 15,000 revolutions per minute (RPM).
- A disk will typically ***contain one or more of these platters*** encased in a sealed container.
- Figure 6.9(a) shows the geometry of a typical disk surface. Each surface consists of a collection of concentric rings called ***tracks***. Each track is partitioned into a collection of ***sectors***.



- Each sector contains an equal number of data bits (typically **512** bytes) encoded in the magnetic material on the sector.
- Sectors are separated by ***gaps*** where no data bits are stored. Gaps *store formatting bits* that identify sectors.

A disk consists of one or more platters stacked on top of each other and encased in a sealed package, as shown in Figure 6.9(b). The entire assembly is often referred to as a ***disk drive***, although we will usually refer to it as simply a ***disk***. We will sometimes refer to disks as ***rotating disks*** to distinguish them from ***flash-based solid state disks (SSDs)***, which have no moving parts.

Disk manufacturers describe the geometry of multiple-platter drives in terms of ***cylinders***, where a cylinder is the *collection of tracks on all the surfaces that are equidistant from the center of the spindle*. For example, if a drive has three platters and six surfaces, and the tracks on each surface are numbered consistently, then ***cylinder k*** is the collection of the six instances of ***track k***.

Disk Capacity

The maximum number of bits that can be recorded by a disk is known as its ***maximum capacity***, or simply ***capacity***. Disk capacity is determined by the following technology factors:

(1) *Recording density (bits/in).* The number of bits that can be squeezed into a 1-inch segment of a track.

(2) *Track density (tracks/in)*. The number of tracks that can be squeezed into a 1-inch segment of the radius extending from the center of the platter.

(3) *Areal density (bits/in2)*. The product of the recording density and the track density.

Disk manufacturers work tirelessly to increase areal density (and thus capacity), and this is doubling every couple of years. The original disks, designed in an age of low areal density, partitioned every track into the same number of sectors, which was determined by the number of sectors that could be recorded on the innermost track. To maintain a fixed number of sectors per track, the sectors were spaced farther apart on the outer tracks.

This was a reasonable approach when areal densities were relatively low. However, as areal densities increased, the gaps between sectors (where no data bits were stored) became unacceptably large. Thus, modern high-capacity disks use a technique known as ***multiple zone recording***, *where the set of cylinders is partitioned into disjoint subsets known as recording zones*. Each zone consists of a contiguous collection of cylinders. Each track in each cylinder in a zone has the same number of sectors, which is determined by the number of sectors that can be packed into the innermost track of the zone.

The capacity of a disk is given by the following formula:

Capacity = # bytes/sector × average # sectors/track × # tracks/surface × # surfaces/platter × # platters disk.

Example:

Suppose we have a disk with five platters, 512 bytes per sector, 20,000 tracks per surface, and an average of 300 sectors per track.
Then the capacity of the disk is:

Capacity = 512 bytes/sector × 300 sectors/track × 20,000 tracks/surface × 2 surfaces/platter × 5 platters/disk

= 30,720,000,000 bytes

= 30.72 GB

Aside Note: How much is a gigabyte?

Unfortunately, the meanings of prefixes such as **kilo (K),** mega (M), giga (G), and tera (T) depend on the context. For measures that relate to the capacity of DRAMs and SRAMs, typically K = 2^{10}, M= 2^{20}, G = 2^{30}, and T = 2^{40}. For measures related to the capacity of I/O devices such as disks and networks, typically K = 10^3, M = 10^6, G = 10^9, and T = 10^{12}. Rates and throughputs usually use these prefix values as well. Fortunately, for the back-of-the-envelope estimates that we typically rely on, either assumption works fine in practice. For example, the relative difference between 2^{30} and 10^9 is not that large: $(2^{30} - 10^9)/10^9 \approx 7\%$. Similarly, $(2^{40} - 10^{12})/10^{12} \approx 10\%$.

Exercise:

What is the capacity of a disk with 3 platters, 15,000 cylinders, an average of 500 sectors per track, and 1,024 bytes per sector?

Disk Operation

Disks read and write bits stored on the magnetic surface using a ***read/write head*** connected to the end of an ***actuator arm***, as shown in Figure 6.10(a).

By moving the arm back and forth along its radial axis, the drive can position the head over any track on the surface. This mechanical motion is known as a seek. Once the head is positioned over the desired track, then, as each bit

on the track passes underneath, the head can either sense the value of the bit (read the bit) or alter the value of the bit (write the bit).

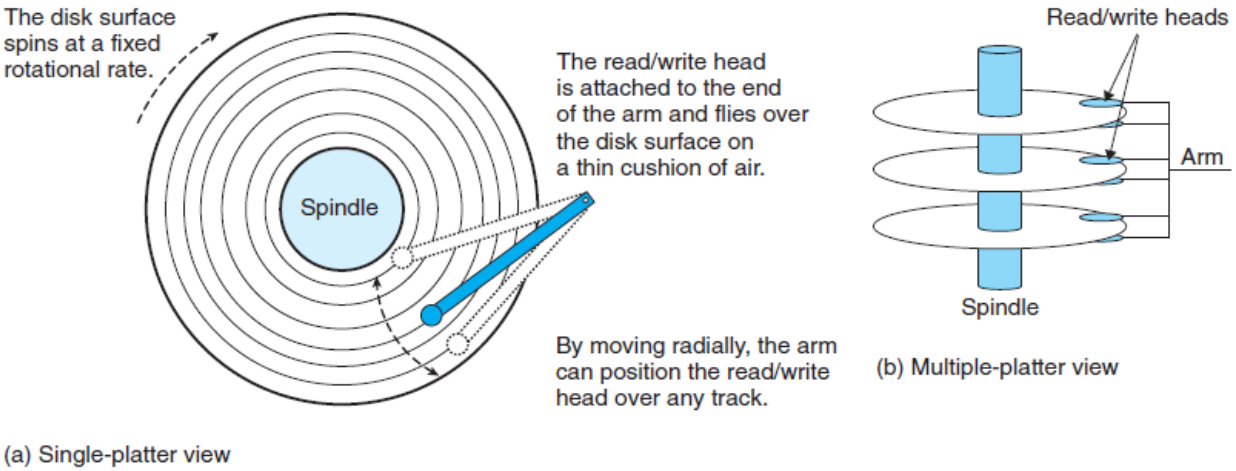


Figure 6.10: Disk dynamics.

Disks with multiple platters have a separate read/write head for each surface, as shown in Figure 6.10(b). The heads are lined up vertically and move in unison. At any point in time, all heads are positioned on the same cylinder.

The read/write head at the end of the arm flies (literally) on a thin cushion of air over the disk surface at a height of about 0.1 microns and a speed of about 80 km/h. This is analogous to placing a skyscraper on its side and flying it around the world at a height of 2.5 cm (1 inch) above the ground, with each orbit of the earth taking only 8 seconds!

At these tolerances, a tiny piece of dust on the surface is like a huge boulder. If the head were to strike one of these boulders, the head would cease flying and crash into the surface (a so-called head crash). For this reason, disks are always sealed in airtight packages.

Disks read and write data in sector-size blocks. The access time for a sector has three main components: *seek time*, *rotational latency*, and *transfer time*:

1) ***Seek time***. To read the contents of some target sector, the arm first positions the head over the track that contains the target sector. *The time*

required to move the arm is called the seek time. The seek time, T_{seek}, depends on:

(a) the previous position of the head and

(b) the speed that the arm moves across the surface.

The average seek time in modern drives, $T_{avg\ seek}$, measured by taking the mean of several thousand seeks to random sectors, is typically on the order of 3 to 9 ms. The maximum time for a single seek, $T_{max\ seek}$, can be as high as 20 ms.

2) ***Rotational latency***. Once the head is in position over the track, the drive waits for the first bit of the target sector to pass under the head. The performance of this step depends on:

(a) both the position of the surface when the head arrives at the target track and

(b) the rotational speed of the disk.

In the worst case, the head just misses the target sector and waits for the disk to make a full rotation. Thus, the maximum rotational latency, in seconds, is given by

$$T_{\text{max rotation}} = \frac{1}{\text{RPM}} \times \frac{60\ \text{secs}}{1\ \text{min}}$$

The average rotational latency, $\mathbf{T_{avg\ rotation}}$, is simply half of $\mathbf{T_{max\ rotation}}$.

3) ***Transfer time***. When the first bit of the target sector is under the head, the drive can begin to read or write the contents of the sector. The transfer time for one sector depends on

(a) the rotational speed and

(b) the number of sectors per track.

Thus, we can roughly estimate the average transfer time for one sector in seconds as:

$$T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{average \# sectors/track})} \times \frac{60\ \text{secs}}{1\ \text{min}}$$

We can estimate the average time to access the contents of a disk sector as *the sum of the average seek time, the average rotational latency, and the average transfer time*.

For example, consider a disk with the following parameters:

Parameter	Value
Rotational rate	7,200 RPM
$T_{avg\ seek}$	9 ms
Average number of sectors/track	400

1) For this disk, the average rotational latency (in ms) is

$$T_{avg\ rotation} = 1/2 \times T_{max\ rotation}$$

$$= 1/2 \times (60\ secs/7{,}200\ RPM) \times 1{,}000\ ms/sec$$

$$\approx 4\ ms$$

2) The average transfer time is

$$T_{avg\ transfer} = 60/7{,}200\ RPM \times 1/400\ sectors/track \times 1{,}000\ ms/sec$$

$$\approx 0.02\ ms$$

Putting it all together, the total estimated access time is

3) $T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$

$$= 9ms + 4\ ms + 0.02\ ms$$

$$= 13.02\ ms$$

This example illustrates some important points:

- The time to access the 512 bytes in a disk sector is dominated by the *seek time* and the *rotational latency*. Accessing the first byte in the sector takes a long time, but the remaining bytes are essentially free.
- Since the seek time and rotational latency are roughly the same, twice the seek time is a simple and reasonable rule for estimating disk access time.

- The access time for a 64-bit word stored in SRAM is roughly 4 ns, and 60 ns for DRAM. Thus, the time to read a 512-byte sector-size block from memory is roughly 256 ns for SRAM and 4,000 ns for DRAM. The disk access time, roughly 10 ms, is about 40,000 times greater than SRAM, and about 2,500 times greater than DRAM.

Practice Problem 6.3

Estimate the average time (in ms) to access a sector on the following disk:
Rotational rate: = 12,000 RPM, $T_{avg\ seek}$ = 5 ms
Average number of sectors/track = 300.

Example

A disk drive uses two-sided disks and records data on 80 tracks per side. A track has 9 sectors and each holds 512 bytes of data. The disk rotates at 360 rpm, the seek time is 10 ms track-to-track, the head settling time is 10 ms, and the head-load time is 200 ms. From the above information calculate:

(i) The total capacity of the floppy disk in bytes

(ii) The average rotational latency

(iii) The average time to locate a given sector assuming that the head is initially parked at track 0, and is in an unloaded state. The head is loaded after the required track has been located.

(iv) The time taken to read a single sector once it has been located

(v) The average rate at which data is moved from the disk to the processor during the reading of a sector. This should be expressed in bits per second.

(vi) Estimate the packing density of the disk in terms of bits per inch around a track located at 3 inches from the center.

Solution

(i) Total capacity = sides x tracks x sectors x bytes/sector = 2 x 80 x 9 x 512 = 737,280 bytes

(ii) $T_{Avg.\ Rotational}$ Latency = 1/2 x $T_{max.\ Rotation.}$

½*1/RPM * 60 * 1000 – ½x1/360 x 60 x 1000 = 83.3ms.

(iii) Average time to locate sector = latency + head load time + head settling time + seek time

= 83.3 ms + 200 ms + 10 ms + 80/2 x 10 ms = 698.3 ms.

(iv) $T_{Avg.\ Transfer}$ = 1/RPM x 1/Avg. Sectors/Track x 60sec x 1000

= 1/360 x 1/9 x 60 x 1000 = 18.52 ms

(v) During the reading of a sector, 512 bytes are read in 18.52 ms. The average data rate is the number of bits read divided by the time taken

= (512 x 8)/0.01852 = 221,166 bits/second

(vi) Packing density = total number of bits divided by track length

= 9 x 512 x 8/(2 x 3.142 x 3) = 1955.4 bits/in

Practice Problem 6.3

1. A hard disk drive has 10 disks and 18 surfaces available for recording. Each surface is composed of 400 concentric tracks and the disks rotate at 10000 r.p.m. Each track is divided into 64 blocks of 512 64-bit words. There is one R/W head per surface and it is possible to read the 18 tracks of a given cylinder simultaneously. The time to step from track to track is 1ms (10^{-3}s). Between data transfers the head is parked at the outermost track of the disk. Calculate:

    (a) The total capacity in bits of the disk drive

    (b) The maximum data rate in bits/second

(c) The average access time in ms

(d) The average transfer rate when reading 512-64-bit word blocks located randomly on the disk

(e) The recording density (bits/in) of the innermost and the outermost track if the disk has a 6-in diameter and the outermost track comes to 1-in from the edge of the disk. The track density is 200 tracks/in.

Logical Disk Blocks

As we have seen, modern disks have complex geometries, with multiple surfaces and different recording zones on those surfaces. To hide this complexity from the operating system, modern disks present a simpler view of their geometry as a sequence of **B *sector-size logical blocks***, numbered **0, 1, . . . , B − 1**.

A small hardware/firmware device in the disk package, called the ***disk controller***, maintains the mapping between logical block numbers and actual (physical) disk sectors.

When the operating system wants to perform an I/O operation such as reading a disk sector into main memory, it sends a command to the disk controller asking it to read a particular logical block number. Firmware on the controller performs a fast table lookup that translates the logical block number into a (***surface, track, sector***) triple that uniquely identifies the corresponding physical sector. Hardware on the controller interprets this triple to move the heads to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed by the head into a small memory buffer on the controller, and copies them into main memory.

Example:

Suppose that a 1 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Rotational rate	= 13,000 RPM
$T_{avg\ seek}$	= 6 ms
Average number of sectors/track	= 5,000
Surfaces	= 4
Sector size	= 512 bytes

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{avg\ seek} + T_{avg\ rotation}$.

(a) *Best case:* Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential).

(b) *Random case:* Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

Solution:

This is a good check of your understanding of the factors that affect disk performance. First we need to determine a few basic properties of the file and the disk. The file consists of 10,000 512-byte logical blocks. For the disk, $T_{avg\ seek}$ = 6 ms, $T_{max\ rotation}$ = 4.61 ms, and $T_{avg\ rotation}$ = 2.30 ms.

(a) *Best case:* In the optimal case, the blocks are mapped to contiguous sectors, on the same cylinder, that can be read one after the other without moving the head. Once the head is positioned over the first sector it takes two full rotations (5,000 sectors per rotation) of the disk to read all 10,000 blocks.

So the total time to read the file is $T_{avg\ seek} + T_{avg\ rotation} + 2 \times T_{max\ rotation}$ = 6 + 2.30 + 9.22 = 17.52 ms.

(b) *Random case*: In this case, where blocks are mapped randomly to sectors, reading each of the 10,000 blocks requires $T_{avg\ seek} + T_{avg\ rotation}$ ms, so the total time to read the file is $(T_{avg\ seek} + T_{avg\ rotation}) \times 10{,}000$ = 83,000 ms (83 seconds!).

You can see now why it's often a good idea to defragment your disk drive!

Exercises

(1) Estimate the average time (in ms) to access a sector on the following disk: Rotational rate = 12,000 RPM, $T_{avg\ seek}$ = 3 ms, Average number of sectors/track = 500.

(2) Suppose that a 2 MB file consisting of 512-byte logical blocks is stored on a disk drive with the following characteristics:

Rotational rate = 18,000 RPM, $T_{avg\ seek}$ = 8 ms

Average number of sectors/track 2,000, Surfaces 4, Sector size 512 bytes.

For each case below, suppose that a program reads the logical blocks of the file sequentially, one after the other, and that the time to position the head over the first block is $T_{avg\ seek} + T_{avg\ rotation}$.

(a) *Best case*: Estimate the optimal time (in ms) required to read the file given the best possible mapping of logical blocks to disk sectors (i.e., sequential).

(b) *Random case:* Estimate the time (in ms) required to read the file if blocks are mapped randomly to disk sectors.

Connecting I/O Devices

Input/output (I/O) devices such as graphics cards, monitors, mice, keyboards, and disks are connected to the CPU and main memory using an I/O bus. Unlike the system bus and memory buses, which are CPU-specific, I/O buses are designed to be independent of the underlying CPU. Figure 6.11 shows a representative I/O bus structure that connects the CPU, main memory, and I/O devices.

Although the I/O bus is slower than the system and memory buses, it can accommodate a wide variety of third-party I/O devices. For example, the bus in Figure 6.11 has three different types of devices attached to it.

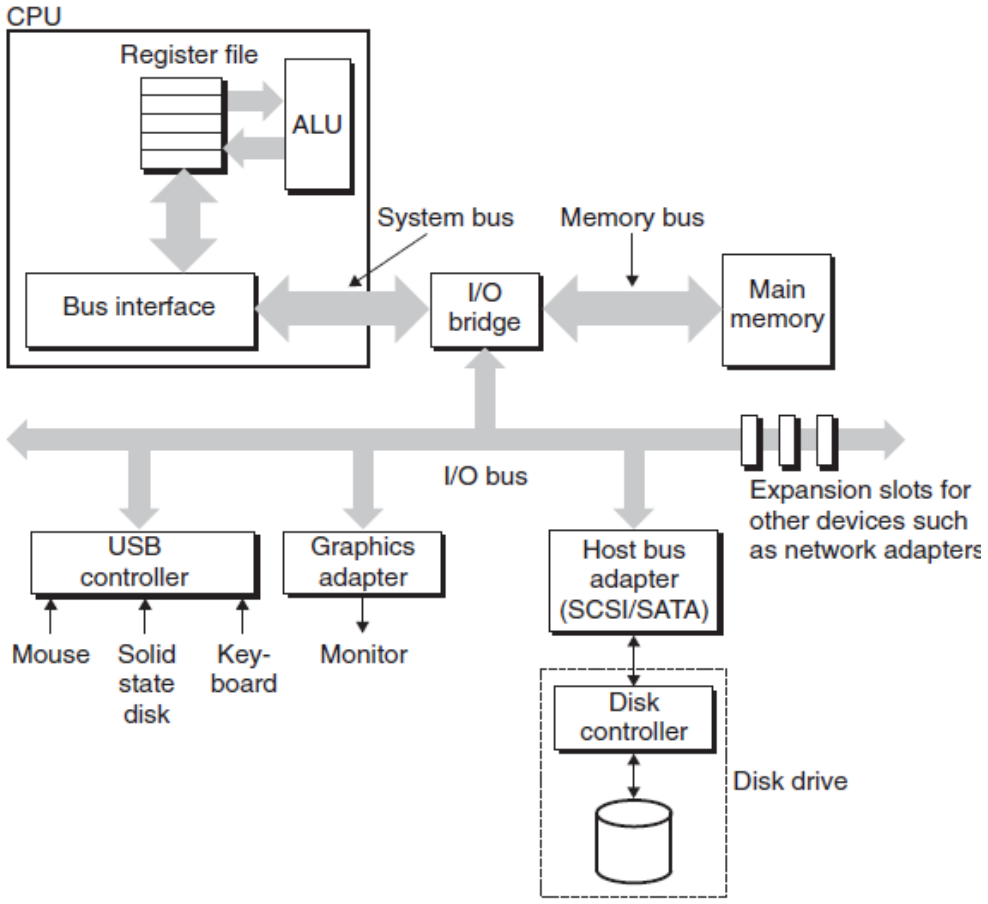


Figure 6.11: Example bus structure that connects the CPU, main memory, and I/O devices.

- A *Universal Serial Bus (USB)* controller is a conduit for devices attached to a USB bus, which is a wildly popular standard for connecting a variety of peripheral I/O devices, including keyboards, mice, modems, digital cameras, game controllers, printers, external disk drives, and solid state disks. USB 3.0 buses have a maximum bandwidth of 625 MB/s. USB 3.1 buses have a maximum bandwidth of 1,250 MB/s.

- A *graphics card (or adapter)* contains hardware and software logic that is responsible for painting the pixels on the display monitor on behalf of the CPU.

- A **host bus adapter** that connects one or more disks to the I/O bus using a communication protocol defined by a particular *host bus interface*. The two most popular such interfaces for disks are SCSI (pronounced "scuzzy") and SATA (pronounced "sat-uh"). SCSI disks are typically faster and more expensive than SATA drives. A SCSI host bus adapter (often called a SCSI *controller*) can support multiple disk drives, as opposed to SATA adapters, which can only support one drive.

Additional devices such as *network adapters* can be attached to the I/O bus by plugging the adapter into empty *expansion slots* on the motherboard that provide a direct electrical connection to the bus.

Accessing Disks

While a detailed description of how I/O devices work and how they are programmed is outside our scope here, we can give you a general idea. For example, Figure 6.12 summarizes the steps that take place when a CPU reads data from a disk.

The CPU issues commands to I/O devices using a technique called *memory mapped I/O* (Figure 6.12(a)). In a system with memory-mapped I/O, a block of addresses in the address space is reserved for communicating with I/O

devices. Each of these addresses is known as an *I/O port*. Each device is associated with (or mapped to) one or more ports when it is attached to the bus.

As a simple example, suppose that the disk controller is mapped to port **0xa0**. Then the CPU might initiate a disk read by executing three store instructions to address **0xa0**:

(i) The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished. (We will discuss interrupts later)

(ii) The second instruction indicates the logical block number that should be read.

(iii) The third instruction indicates the main memory address where the contents of the disk sector should be stored.

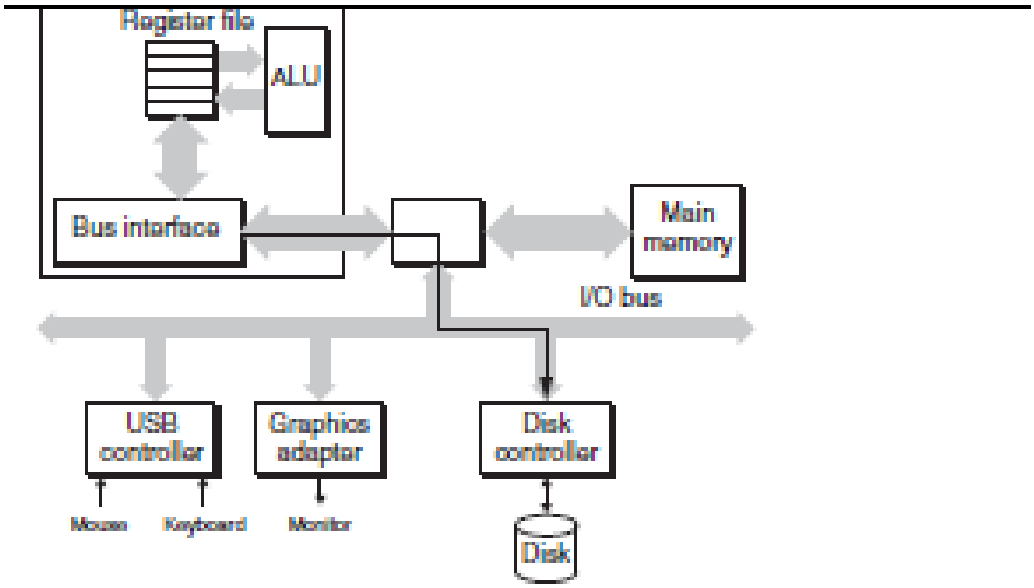
After it issues the request, the CPU will typically do other work while the disk is performing the read. Recall that a 1 GHz processor with a 1 ns clock cycle can potentially execute 16 million instructions in the 16 ms it takes to read the disk. Simply waiting and doing nothing while the transfer is taking place would be enormously wasteful.

After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU (Figure 6.12(b)). This process, whereby a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as direct *memory access (DMA).* The transfer of data is known as a ***DMA transfer***.

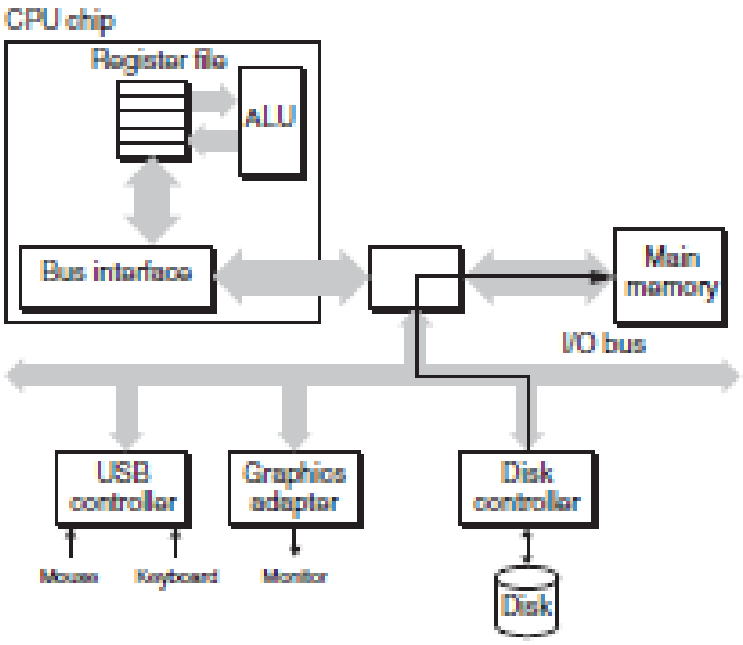
After the DMA transfer is complete and the contents of the disk sector are safely stored in main memory, the disk controller notifies the CPU by sending an interrupt signal to the CPU (Figure 6.12(c)). The basic idea is that an

interrupt signals an external pin on the CPU chip. This causes the CPU to stop what it is currently working on and jump to an operating system routine. The routine records the fact that the I/O has finished and then returns control to the point where the CPU was interrupted.

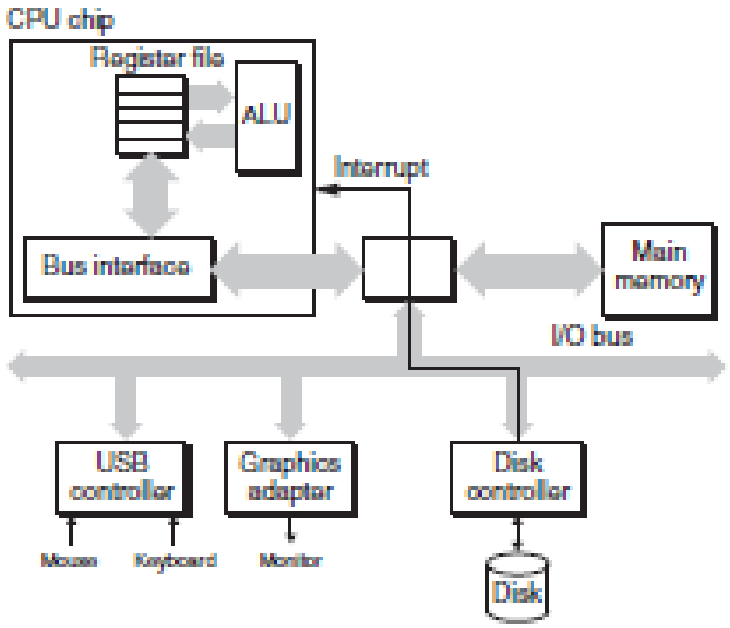
Figure 6.12: reading a disk secto



(a) The CPU initiates a disk read by writing a command, logical block number, and destination memory address to the memory-mapped address associated with the disk.



(b) The disk controller reads the sector and performs a DMA transfer into main memory.



(c) When the DMA transfer is complete, the disk controller notifies the CPU with an interrupt.

Solid State Disks

A solid state disk (SSD) is a storage technology, based on flash memory, that in some situations is an attractive alternative to the conventional rotating disk. Figure 6.13 shows the basic idea. An SSD package plugs into a standard disk slot on the I/O bus (typically USB or SATA) and behaves like any other disk, processing requests from the CPU to read and write logical disk blocks.

An SSD package consists of one or more flash memory chips, which replace the mechanical drive in a conventional rotating disk, and a *flash translation layer*, which is a hardware/firmware device that plays the same role as a disk controller, translating requests for logical blocks into accesses of the underlying physical device.

Reading from SSDs is faster than writing. The difference between random reading and writing performance is caused by a fundamental property of the underlying flash memory. As shown in Figure 6.13, a flash memory consists of a sequence of **B *blocks***, where each block consists of **P** pages. Typically, pages are 512 bytes to 4 KB in size, and a block consists of 32–128 pages,

with total block sizes ranging from 16 KB to 512 KB. Data are read and written in units of ***pages***. A page can be written only after the entire block to which it belongs has been erased (typically, this means that all bits in the block are set to 1). However, once a block is erased, each page in the block can be written once with no further erasing. A block wears out after roughly 100,000 repeated writes. Once a block wears out, it can no longer be used.

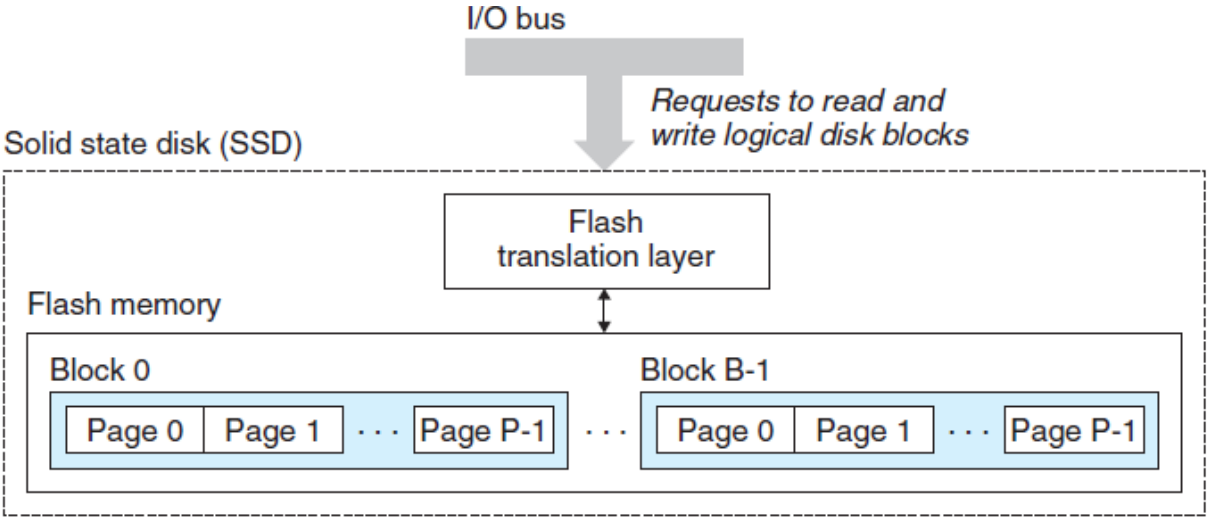


Figure 6.13: Solid state disk (SSD).

Random writes are slower for two reasons. First, erasing a block takes a relatively long time, on the order of 1 ms, which is more than an order of magnitude longer than it takes to access a page. Second, if a write operation attempts to modify a page **p** that contains existing data (i.e., not all ones), then any pages in the same block with useful data must be copied to a new (erased) block before the write to page **p** can occur. Manufacturers have developed sophisticated logic in the flash translation layer that attempts to amortize the high cost of erasing blocks and to minimize the number of internal copies on writes, but it is unlikely that random writing will ever perform as well as reading.

SSDs have a number of advantages over rotating disks.

(i) They are built of semiconductor memory, with no moving parts, and thus have much faster random access times than rotating disks,

(ii) use less power,

(iii) and are more rugged.

However, there are some disadvantages.

(i) First, because flash blocks wear out after repeated writes, SSDs have the potential to wear out as well. ***Wear-leveling*** logic in the flash translation layer attempts to maximize the lifetime of each block by spreading erasures evenly across all blocks. In practice, the wear-leveling logic is so good that it takes many years for SSDs to wear out.

(ii) SSDs are about 30 times more expensive per byte than rotating disks, and thus the typical storage capacities are significantly less than rotating disks. However, SSD prices are decreasing rapidly as they become more popular, and the gap between the two is decreasing.

SSDs have completely replaced rotating disks in portable music devices, are popular as disk replacements in laptops, and have even begun to appear in desktops and servers. While rotating disks are here to stay, it is clear that SSDs are an important alternative.

Locality

Well-written computer programs tend to exhibit good locality. That is, they tend to reference data items that are near other recently referenced data items or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

Locality is typically described as having two distinct forms: ***temporal locality*** and ***spatial locality***.

(i) In a program with good *temporal locality*, a memory location that is referenced once is likely to be referenced again multiple times in the near future.

(ii) In a program with good *spatial locality*, if a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future.

Programmers should understand the principle of locality because, in general, programs with good locality run faster than programs with poor locality. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality.

- At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as cache memories that hold blocks of the most recently referenced instructions and data items.
- At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system.

- The principle of locality also plays a crucial role in the design of application programs. For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk. High-volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

Aside Note: *When cycle time stood still: The advent of multi-core processors*
The history of computers is marked by some singular events that caused profound changes in the industry and the world. Interestingly, these inflection points tend to occur about once per decade: the development of Fortran in the 1950s, the introduction of the IBM 360 in the early 1960s, the dawn of the Internet (then called ARPANET) in the early 1970s, the introduction of the IBM PC in the early 1980s, and the creation of the World Wide Web in the early 1990s.

The most recent such event occurred early in the 21st century, when computer manufacturers ran headlong into the so-called power wall, discovering that they could no longer increase CPU clock frequencies as quickly because the chips would then consume too much power. The solution was to improve performance by replacing a single large processor with multiple smaller processor ***cores***, each a complete processor capable of executing programs independently and in parallel with the other cores.
This ***multi-core*** approach works in part because the power consumed by a processor is proportional to $\boldsymbol{P = fCV^2}$, where ***f*** is the clock frequency, ***C*** is the capacitance, and ***V*** is the voltage. The capacitance ***C*** is roughly proportional to the area, so the power drawn by multiple cores can be held constant as long as the total area of the cores is constant. As long as feature sizes continue to shrink at the exponential Moore's Law rate, the number of cores in each processor, and thus its effective performance, will continue to increase.

From this point forward, computers will get faster not because the clock frequency increases but because the number of cores in each processor increases, and because architectural innovations increase the efficiency of programs running on those cores. We can see this trend clearly in Figure 6.16. CPU cycle time reached its lowest point in 2003 and then actually started to rise before leveling off and starting to decline again at a slower rate than before. However, because of the advent of multi-core processors (dual-core in 2004 and quad-core in 2007), the effective cycle time continues to decrease at close to its previous rate.

The Memory Hierarchy

In our previous sections, we described some fundamental and enduring properties of storage technology and computer software:

- *Storage technology*. Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.
- *Computer software*. Well-written programs tend to exhibit good locality.

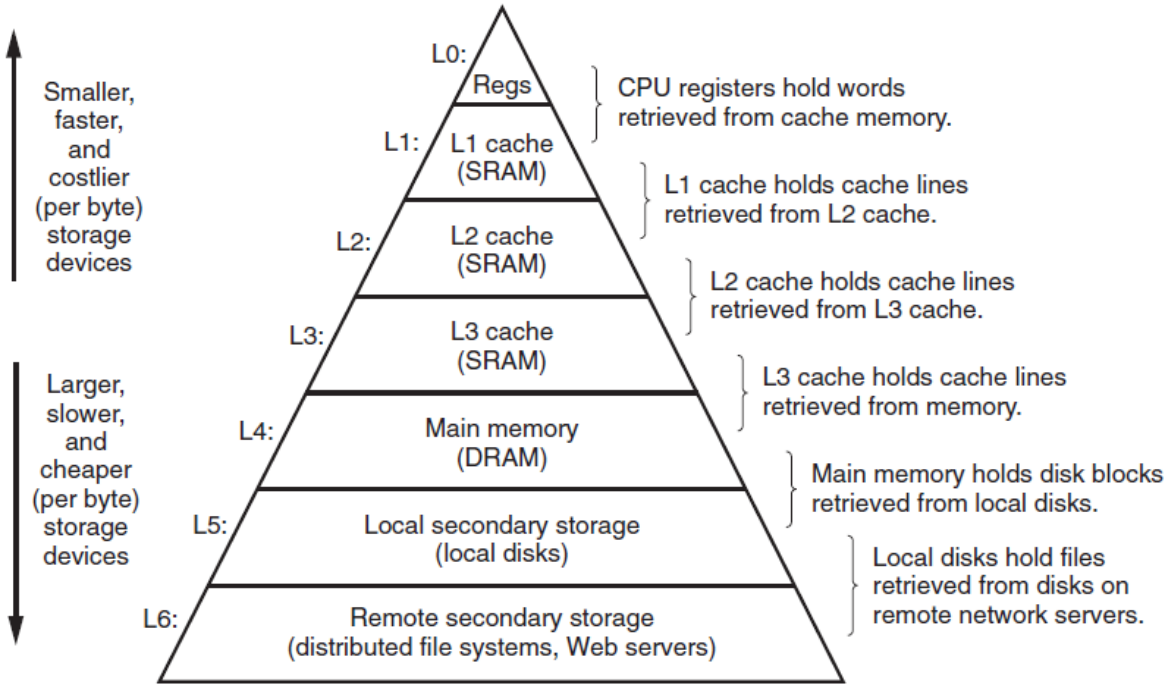


Figure 6.21: The memory hierarchy.

These fundamental properties of hardware and software complement each other beautifully. Their complementary nature suggests an approach for organizing memory systems, known as the memory hierarchy, that is used in all modern computer systems. Figure 6.21 shows a typical *memory hierarchy*. In general, the storage devices get *slower*, *cheaper*, and *larger* as we move from higher to lower levels.

- At the highest level (***L0***) are a small number of fast *CPU registers* that the CPU can access in a single clock cycle.

- Next are one or more small to moderate-size SRAM-based cache memories (*L1*, *L2*, and *L3*) that can be accessed in a few CPU clock cycles.
- These are followed by a large DRAM-based main memory that can be accessed in tens to hundreds of clock cycles.
- Next are slow but enormous local disks.
- Finally, some systems even include an additional level of disks on remote servers that can be accessed over a network. For example, distributed file systems such as the Andrew File System (AFS) or the Network File System (NFS) allow a program to access files that are stored on remote network-connected servers. Similarly, the World Wide Web allows programs to access remote files stored on Web servers anywhere in the world.

6.3.1 Caching in the Memory Hierarchy

In general, a cache (pronounced "cash") is a small, fast storage device that *acts as a staging area for the data objects stored in a larger, slower device*. The process of using a cache is known as caching (pronounced "cashing").

The central idea of a memory hierarchy is that for each ***k***, the faster and smaller storage device at level ***k*** serves as a cache for the larger and slower storage device at level ***k + 1***. In other words, each level in the hierarchy caches data objects from the next lower level. For example, the local disk serves as a cache for files (such as Web pages) retrieved from remote disks over the network, the main memory serves as a cache for data on the local disks, and so on, until we get to the smallest cache of all, the set of CPU registers.

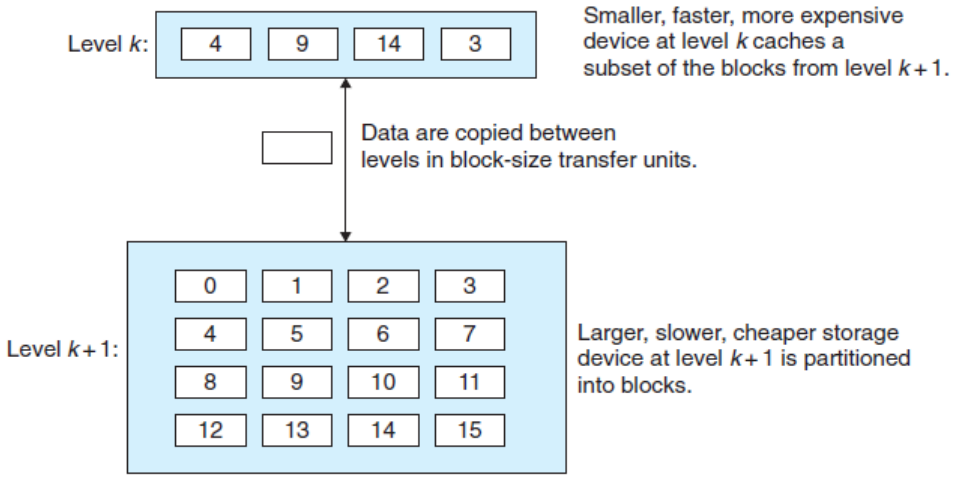


Figure 6.22: The basic principle of caching in a memory hierarchy.

Figure 6.22 shows the general concept of caching in a memory hierarchy. The storage at level ***k + 1*** is partitioned into contiguous chunks of data objects called *blocks*. Each block has a unique address or name that distinguishes it from other blocks. Blocks can be either fixed size (the usual case) or variable size (e.g., the remote HTML files stored on Web servers). For example, the level ***k + 1*** storage in Figure 6.22 is partitioned into 16 fixed-size blocks, numbered 0 to 15.

Similarly, the storage at level ***k*** is partitioned into a smaller set of blocks that are the same size as the blocks at level ***k + 1***. At any point in time, the cache at level ***k*** contains copies of a subset of the blocks from level ***k + 1***. For example, in Figure 6.22, the cache at level ***k*** has room for four blocks and currently contains copies of blocks 4, 9, 14, and 3.

Data are always copied back and forth between level ***k*** and level ***k + 1*** in block-size transfer units. It is important to realize that while the block size is fixed between any particular pair of adjacent levels in the hierarchy, other pairs of levels can have different block sizes. For example, in Figure 6.21, transfers between ***L1*** and ***L0*** typically use word-size blocks. Transfers between ***L2*** and ***L1*** (and ***L3*** and ***L2***, and ***L4*** and ***L3***) typically use blocks of tens of bytes. And transfers between ***L5*** and ***L4*** use blocks with hundreds or

thousands of bytes. In general, devices lower in the hierarchy (further from the CPU) have longer access times, and thus tend to use larger block sizes in order to amortize these longer access times.

Cache Hits

When a program needs a particular data object d from level ***k + 1***, it first looks for ***d*** in one of the blocks currently stored at level ***k***. If ***d*** happens to be cached at level ***k***, then we have what is called a *cache hit*. The program reads ***d*** directly from level ***k***, which by the nature of the memory hierarchy is faster than reading ***d*** from level ***k + 1***. For example, a program with good temporal locality might read a data object from block 14, resulting in a cache hit from level ***k***.

Cache Misses

If, on the other hand, the data object d is not cached at level ***k***, then we have what is called a *cache miss*. When there is a miss, the cache at level ***k*** fetches the block containing ***d*** from the cache at level ***k + 1***, possibly overwriting an existing block if the level ***k*** cache is already full.

This process of overwriting an existing block is known as *replacing or evicting* the block. The block that is evicted is sometimes referred to as a *victim block*. The decision about which block to replace is governed by the cache's *replacement policy*. For example, a cache with a random replacement policy would choose a random victim block. A cache with a *least recently used* (LRU)replacement policy would choose the block that was last accessed the furthest in the past.

After the cache at level ***k*** has fetched the block from level ***k + 1***, the program can read d from level ***k*** as before. For example, in Figure 6.22, reading a data object from block 12 in the level ***k*** cache would result in a cache miss because block 12 is not currently stored in the level k cache. Once it has been copied

from level ***k + 1*** to level ***k***, block 12 will remain there in expectation of later accesses.

Kinds of Cache Misses

It is sometimes helpful to distinguish between different kinds of cache misses. If the cache at level ***k*** is empty, then any access of any data object will miss. An empty cache is sometimes referred to as a *cold cache*, and misses of this kind are called *compulsory misses* or *cold misses*. Cold misses are important because they are often transient events that might not occur in steady state, after the cache has been ***warmed up*** by repeated memory accesses.

Whenever there is a miss, the cache at level ***k*** must implement some placement policy that determines where to place the block it has retrieved from level ***k + 1***. The most flexible placement policy is to allow any block from level ***k + 1*** to be stored in any block at level k. For caches high in the memory hierarchy (close to the CPU) that are implemented in hardware and where speed is at a premium, this policy is usually too expensive to implement because randomly placed blocks are expensive to locate.

Thus, hardware caches typically implement a simpler placement policy that restricts a particular block at level ***k + 1*** to a small subset (sometimes a singleton) of the blocks at level ***k***. For example, in Figure 6.22, we might decide that a block ***i*** at level ***k + 1*** must be placed in block (***i mod 4***) at level ***k***. For example, blocks **0, 4, 8**, and **12** at level ***k + 1*** would map to block ***0*** at level ***k***; blocks ***1, 5, 9***, and ***13*** would map to block ***1***; and so on. Notice that our example cache in Figure 6.22 uses this policy.

Restrictive placement policies of this kind lead to a type of miss known as a *conflict miss*, in which the cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing. For example, in Figure 6.22, if the program requests block 0, then

block 8, then block 0, then block 8, and so on, each of the references to these two blocks would miss in the cache at level k, even though this cache can hold a total of four blocks.

Programs often run as a sequence of phases (e.g., loops) where each phase accesses some reasonably constant set of cache blocks. For example, a nested loop might access the elements of the same array over and over again. This set of blocks is called the *working set* of the phase. When the size of the working set exceeds the size of the cache, the cache will experience what are known as *capacity misses*. In other words, the cache is just too small to handle this particular working set.