

# 5

*Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.*

**A. W. Burks, H. H. Goldstine, and  
J. von Neumann**

*Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, 1946

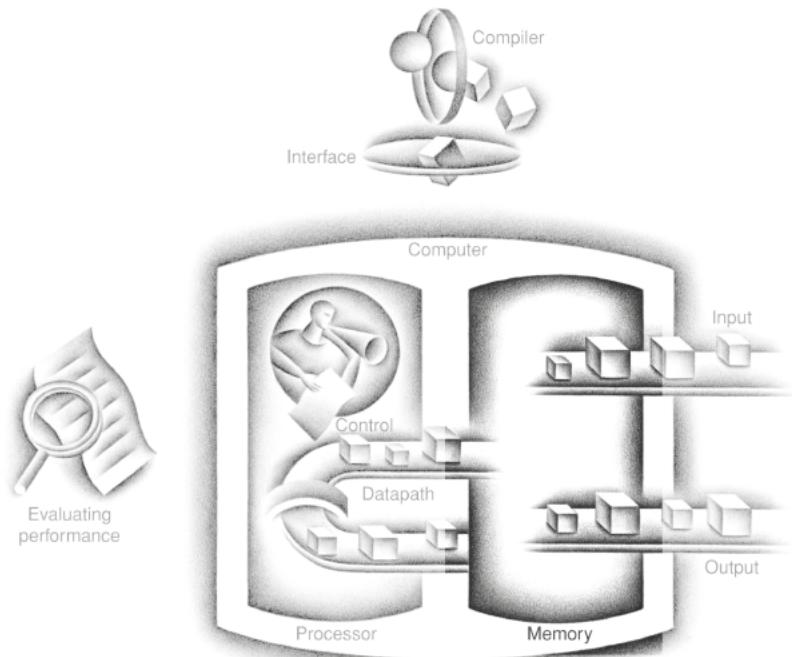
## **Large and Fast: Exploiting Memory Hierarchy**

- 5.1      Introduction**    374
- 5.2      Memory Technologies**    378
- 5.3      The Basics of Caches**    383
- 5.4      Measuring and Improving Cache Performance**    398
- 5.5      Dependable Memory Hierarchy**    418
- 5.6      Virtual Machines**    424
- 5.7      Virtual Memory**    427

- 5.8 A Common Framework for Memory Hierarchy** 454  
**5.9 Using a Finite-State Machine to Control a Simple Cache** 461  
**5.10 Parallelism and Memory Hierarchies: Cache Coherence** 466  
 **5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks** 470  
 **5.12 Advanced Material: Implementing Cache Controllers** 470  
**5.13 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Memory Hierarchies** 471  
**5.14 Going Faster: Cache Blocking and Matrix Multiply** 475  
**5.15 Fallacies and Pitfalls** 478  
**5.16 Concluding Remarks** 482  
 **5.17 Historical Perspective and Further Reading** 483  
**5.18 Exercises** 483

---

## 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 good 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 going back 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 find out about the EDSAC, you also noticed that there was another book shelved next to it about early mechanical computers, so you also 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 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.

Speed	Processor	Size	Cost (\$/bit)	Current technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic disk

**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.

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 high amounts of 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.

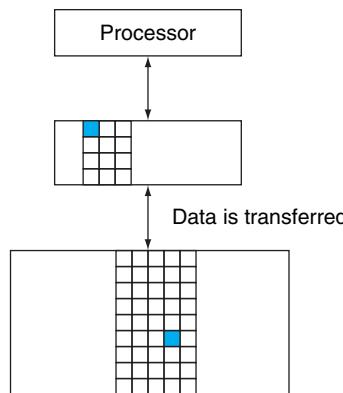
Figure 5.1 shows the faster memory is close to the processor and the slower, less expensive memory is below it. The goal is to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory.

The data is similarly hierarchical: a level closer to the processor is generally a subset of any level further away, and all the data is 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 is copied between only two adjacent levels at a time, so we can focus our attention on just two levels.

### 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 5.2 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.

**block (or line)** The minimum unit of information that can be either present or not present in a cache.

**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.

**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.

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.

If the data requested by the processor appears 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 is 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.

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.)

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 408, 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.

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.

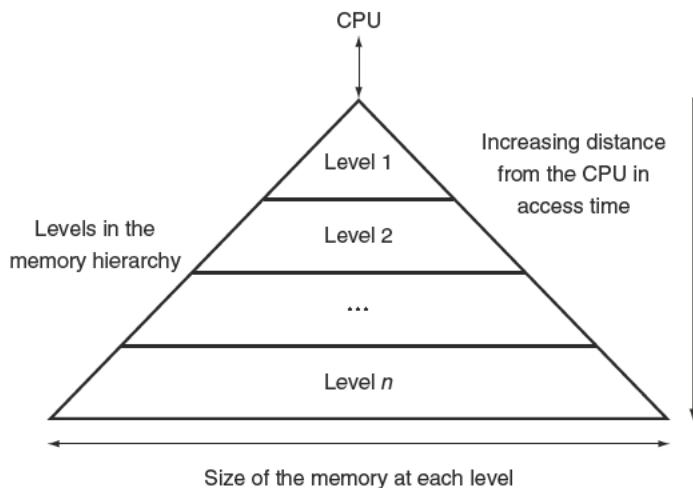
In most systems, the memory is a true hierarchy, meaning that data cannot be present in level  $i$  unless it is also present in level  $i + 1$ .

## The BIG Picture

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.

## Check Yourself



**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.

## 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 B.9](#) of [Appendix B](#). 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 separate SRAM chips has nearly evaporated.



## DRAM Technology

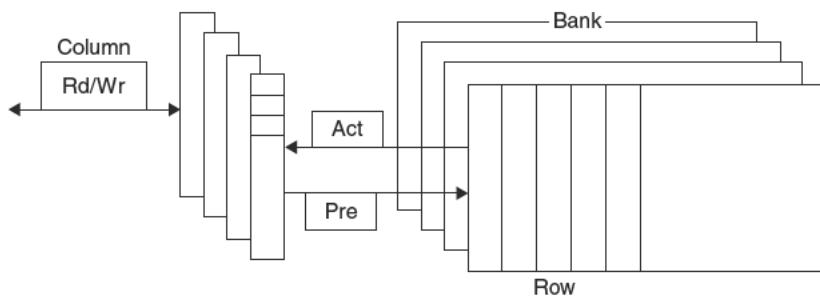
In a SRAM, as long as power is applied, the value can be kept indefinitely. In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single 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, as opposed 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.

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 further improve the interface to processors, 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.



**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 Kibit	\$1,500,000	250 ns	150 ns
1983	256 Kibit	\$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 three 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.

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 are 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  $8 \times 3200 = 25,600$  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.

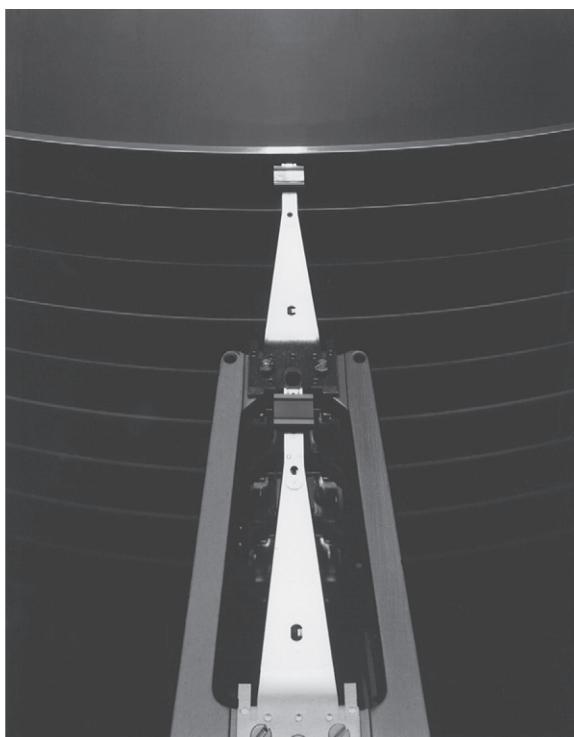
Each disk surface is divided into concentric circles, called *tracks*. There are typically tens of thousands of tracks per surface. Each track is in turn divided into

**track** One of thousands of concentric circles that makes 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.

**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.

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.



---

**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.

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

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*.

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 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

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

**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.

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—disk is 10 to 100 time 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

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

*Cache: a safe place for hiding or storing things.*

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

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 today, 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  $X_1, X_2, \dots, X_{n-1}$ , 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.

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:

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

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  $1_{\text{ten}}$  ( $00001_{\text{two}}$ ) and  $29_{\text{ten}}$  ( $11101_{\text{two}}$ ) map to locations  $1_{\text{ten}}$  ( $001_{\text{two}}$ ) and  $5_{\text{ten}}$  ( $101_{\text{two}}$ ) in a direct-mapped cache of eight words.

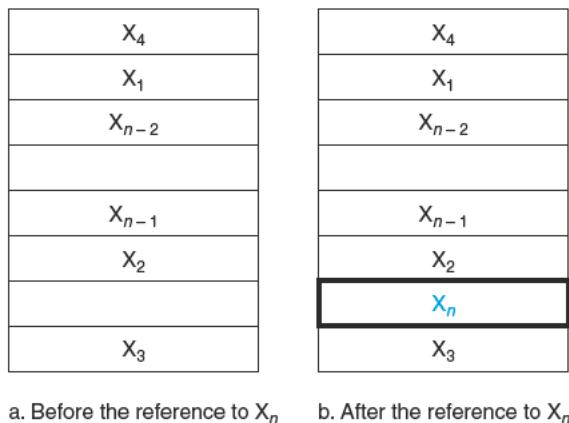
Because each cache location can contain the contents of a number of different memory locations, how do we know whether the data in the cache corresponds to a requested word? That is, how do we know whether a requested word is in the cache or not? We answer this question by adding a set of **tags** to 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 only 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 2 of the 5 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.

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,

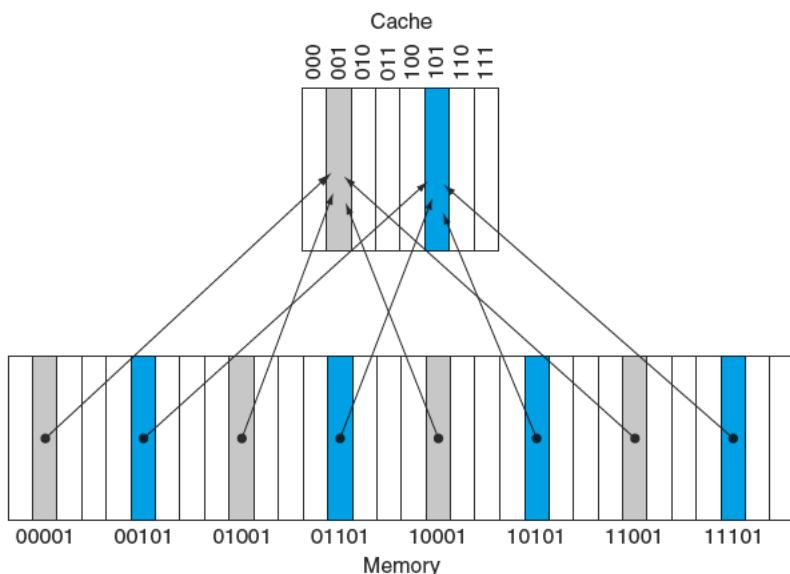
**direct-mapped cache**

A cache structure in which each memory location is mapped to exactly one location in the cache.

**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.



**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.



**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  $00001_{\text{two}}$ ,  $01001_{\text{two}}$ ,  $10001_{\text{two}}$ , and  $11001_{\text{two}}$  all map to entry  $001_{\text{two}}$  of the cache, while addresses  $00101_{\text{two}}$ ,  $01101_{\text{two}}$ ,  $10101_{\text{two}}$ , and  $11101_{\text{two}}$  all map to entry  $101_{\text{two}}$  of the cache.

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

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.

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.



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 higher than 95% (see [Figure 5.47](#)).

## 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 three 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 <sub>two</sub>	miss (5.6b)	(10110 <sub>two</sub> mod 8) = 110 <sub>two</sub>
26	11010 <sub>two</sub>	miss (5.6c)	(11010 <sub>two</sub> mod 8) = 010 <sub>two</sub>
22	10110 <sub>two</sub>	hit	(10110 <sub>two</sub> mod 8) = 110 <sub>two</sub>
26	11010 <sub>two</sub>	hit	(11010 <sub>two</sub> mod 8) = 010 <sub>two</sub>
16	10000 <sub>two</sub>	miss (5.6d)	(10000 <sub>two</sub> mod 8) = 000 <sub>two</sub>
3	00011 <sub>two</sub>	miss (5.6e)	(00011 <sub>two</sub> mod 8) = 011 <sub>two</sub>
16	10000 <sub>two</sub>	hit	(10000 <sub>two</sub> mod 8) = 000 <sub>two</sub>
18	10010 <sub>two</sub>	miss (5.6f)	(10010 <sub>two</sub> mod 8) = 010 <sub>two</sub>
16	10000 <sub>two</sub>	hit	(10000 <sub>two</sub> mod 8) = 000 <sub>two</sub>

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

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 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

b. After handling a miss of address (10110<sub>two</sub>)

Index	V	Tag	Data
000	N		
001	N		
010	Y	11 <sub>two</sub>	Memory (11010 <sub>two</sub> )
011	N		
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

c. After handling a miss of address (11010<sub>two</sub>)

Index	V	Tag	Data
000	Y	10 <sub>two</sub>	Memory (10000 <sub>two</sub> )
001	N		
010	Y	11 <sub>two</sub>	Memory (11010 <sub>two</sub> )
011	N		
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

d. After handling a miss of address (10000<sub>two</sub>)

Index	V	Tag	Data
000	Y	10 <sub>two</sub>	Memory (10000 <sub>two</sub> )
001	N		
010	Y	11 <sub>two</sub>	Memory (11010 <sub>two</sub> )
011	Y	00 <sub>two</sub>	Memory (00011 <sub>two</sub> )
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

e. After handling a miss of address (00011<sub>two</sub>)

Index	V	Tag	Data
000	Y	10 <sub>two</sub>	Memory (10000 <sub>two</sub> )
001	N		
010	Y	10 <sub>two</sub>	Memory (10010 <sub>two</sub> )
011	Y	00 <sub>two</sub>	Memory (00011 <sub>two</sub> )
100	N		
101	N		
110	Y	10 <sub>two</sub>	Memory (10110 <sub>two</sub> )
111	N		

f. After handling a miss of address (10010<sub>two</sub>)

**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 386.** The cache is initially empty, with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110<sub>two</sub> (miss), 11010<sub>two</sub> (miss), 10110<sub>two</sub> (hit), 11010<sub>two</sub> (hit), 10000<sub>two</sub> (miss), 00011<sub>two</sub> (miss), 10000<sub>two</sub> (hit), 10010<sub>two</sub> (miss), and 10000<sub>two</sub> (hit). The figures show the cache contents after each miss in the sequence has been handled. When address 10010<sub>two</sub> (18) is referenced, the entry for address 11010<sub>two</sub> (26) must be replaced, and a reference to 11010<sub>two</sub> 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<sub>two</sub> has tag 10<sub>two</sub> and corresponds to address 10010<sub>two</sub>.

we have conflicting demands for a block. The word at address 18 ( $10010_{\text{two}}$ ) should be brought into cache block 2 ( $010_{\text{two}}$ ). Hence, it must replace the word at address 26 ( $11010_{\text{two}}$ ), which is already in cache block 2 ( $010_{\text{two}}$ ). 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 only 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

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. In the MIPS architecture, since words are aligned to multiples of four bytes, the least significant two bits of every address specify a byte within a word. Hence, the least significant two bits are ignored when selecting a word in the block.

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, but normally it is several. For the following situation:

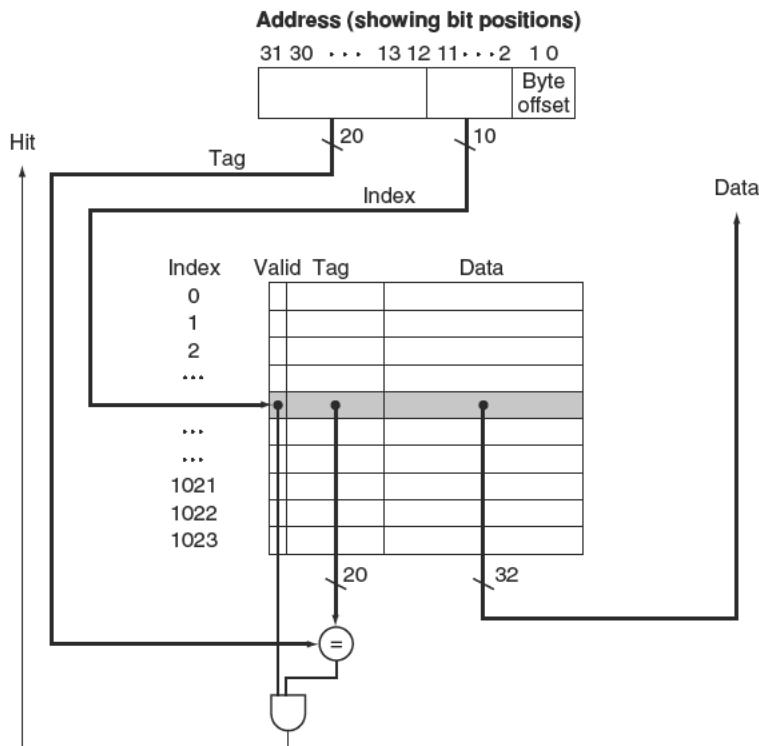
- 32-bit addresses
- A direct-mapped cache
- The cache size is  $2^n$  blocks, so  $n$  bits are used for the index
- The block size is  $2^m$  words ( $2^{m+2}$  bytes), so  $m$  bits are used for the word within the block, and two bits are used for the byte part of the address

the size of the tag field is

$$32 - (n + m + 2).$$

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

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



**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. We assume 32-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  $32 - 10 - 2 = 20$  bits to be compared against the tag. If the tag and upper 20 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.

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

$$2^n \times (2^m \times 32 + (32 - n - m - 2) + 1) = 2^n \times (2^m \times 32 + 31 - n - m).$$

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.

**EXAMPLE****ANSWER****Bits in a Cache**

How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address?

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

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kibibits}$$

or 18.4 KiB for a 16 KiB cache. For this cache, the total number of bits in the cache is about 1.15 times as many as needed just for the storage of the data.

**EXAMPLE****ANSWER****Mapping an Address to a Multiword Cache Block**

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

We saw the formula on page 384. The block is given by

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

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

$$\left\lceil \frac{\text{Byte address}}{\text{Bytes per block}} \right\rceil \times \text{Bytes per block}$$

and

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

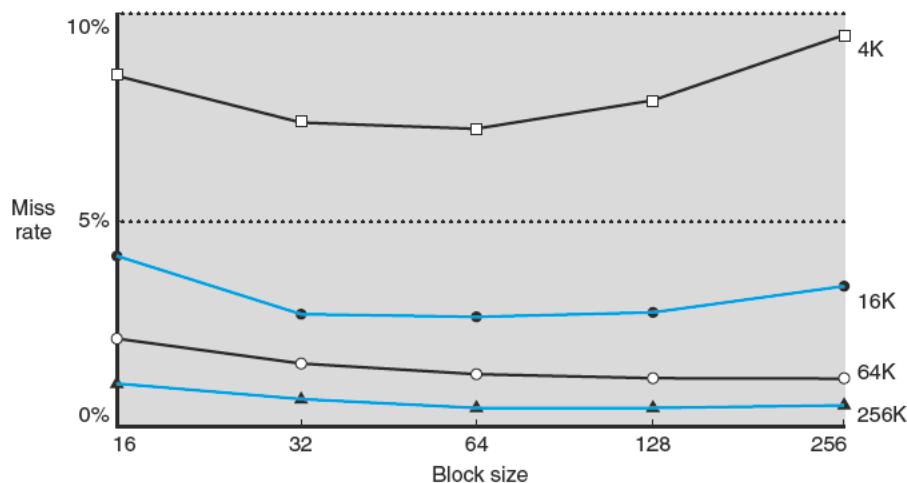
Thus, with 16 bytes per block, byte address 1200 is block address

$$\left\lceil \frac{1200}{6} \right\rceil = 75$$

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 in the miss rate become smaller.

A more serious problem associated with just increasing the block size is that the cost of a miss increases. The miss penalty is determined by the time required to fetch



**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 this data is based on SPEC92.

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 increases. 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 simplest 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 limit early restart.

## Handling Cache Misses

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

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.

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) as opposed to an 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 instruct 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 (current PC – 4) 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**.

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 very 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

### write-through

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

**write buffer** A queue that holds data while the data is waiting to be written to memory.

**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.

misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of  $1.0 + 100 \times 10\% = 11$ , 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 it is 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.

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.

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 is 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 is 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.

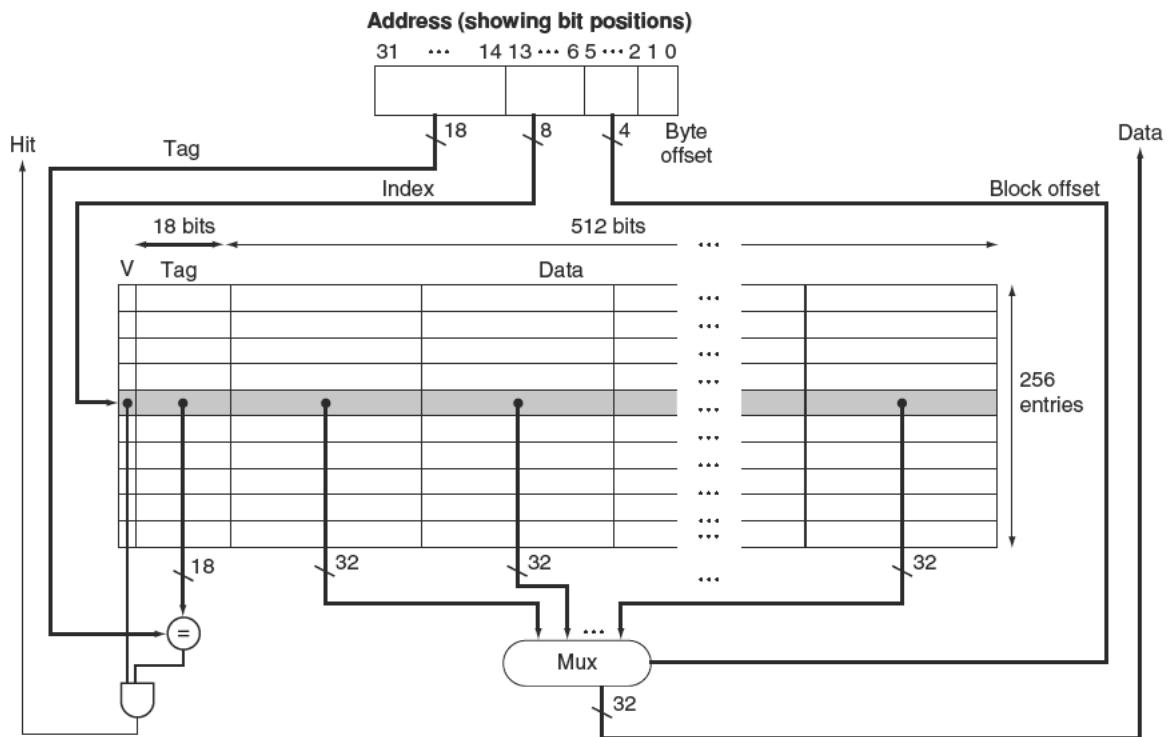
## An Example Cache: The Intrinsity FastMATH Processor

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.

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.



**FIGURE 5.12 The 16 KIB caches In the Intrinsity FastMATH each contain 256 blocks with 16 words per block.** 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.

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 larger 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 are 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 different 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 394 for more details on dealing with writes), and write buffer stalls, which occur when the write buffer is full when a write occurs. Thus, the cycles stalled for writes equals 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 not possible 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.

## EXAMPLE

### Calculating Cache Performance

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

$$\begin{aligned}\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} &= \frac{I \times \text{CPI}_{\text{stall}} \times \text{Clock cycle}}{I \times \text{CPI}_{\text{perfect}} \times \text{Clock cycle}} \\ &= \frac{\text{CPI}_{\text{stall}}}{\text{CPI}_{\text{perfect}}} = \frac{5.44}{2}\end{aligned}$$

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 increase

hit time shortly, one example is increasing the cache size. A larger cache could clearly have a longer 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:

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

## EXAMPLE

### Calculating Average Memory Access Time

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 5.15, Fallacies and Pitfalls.

### Reducing Cache Misses by More Flexible Placement of Blocks

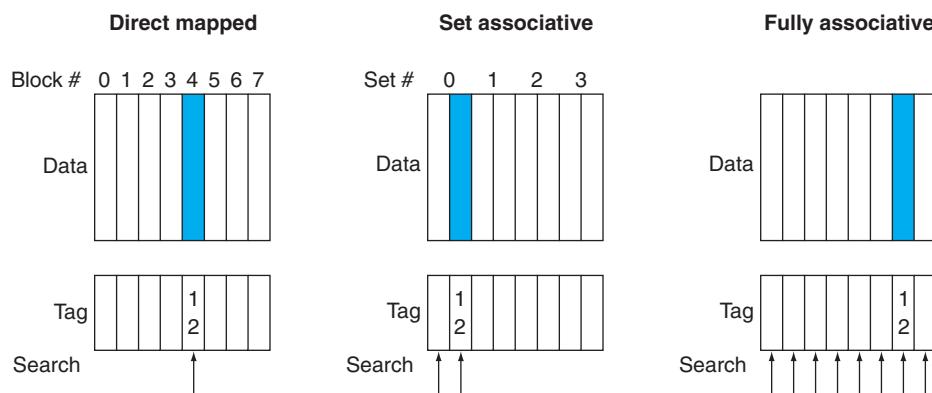
So far, when we place a block in the cache, we have used a simple placement scheme: A block can go in exactly one place in the cache. As mentioned earlier, it is called *direct mapped* because there is a direct mapping from any block address in memory to a single location in the upper level of the hierarchy. 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.

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 placed in a cache with eight blocks total, according to the three block placement policies.

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

$$(\text{Block number}) \bmod (\text{Number of blocks in the cache})$$



**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  $(12 \bmod 8) = 4$ . In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set  $(12 \bmod 4) = 0$ ; 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.

### fully associative cache

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

### set-associative cache

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

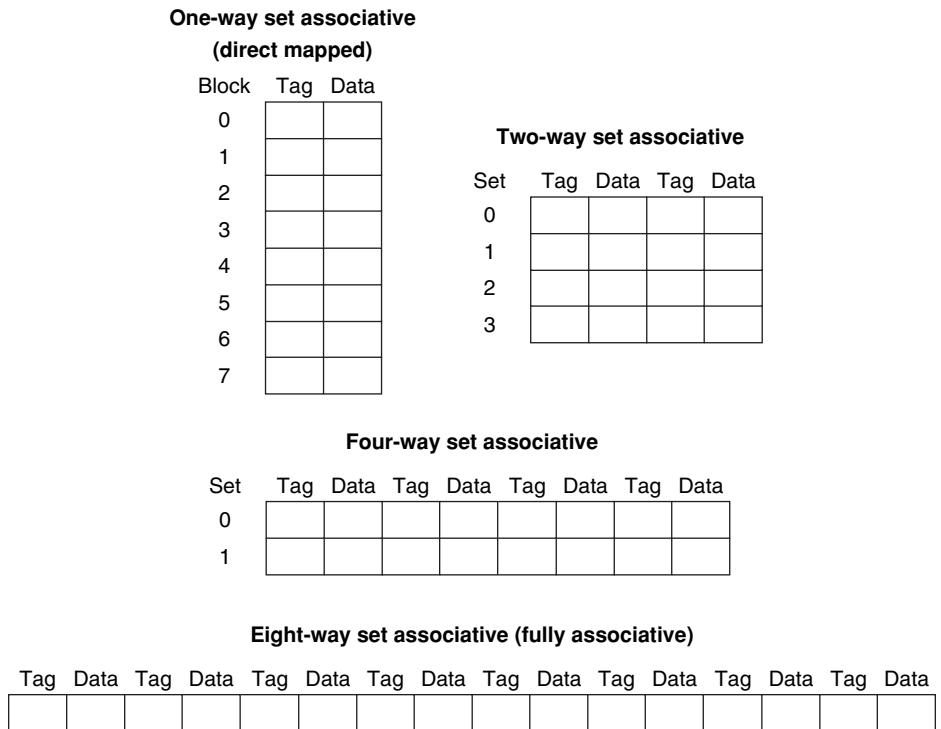
In a set-associative cache, the set containing a memory block is given by

$$(\text{Block number}) \bmod (\text{Number of sets in the cache})$$

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 simply 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.

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.



**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.

### Misses and Associativity in Caches

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.

### EXAMPLE

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

### ANSWER

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 looks 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 3 caches would have the same 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 Intrinsic 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).



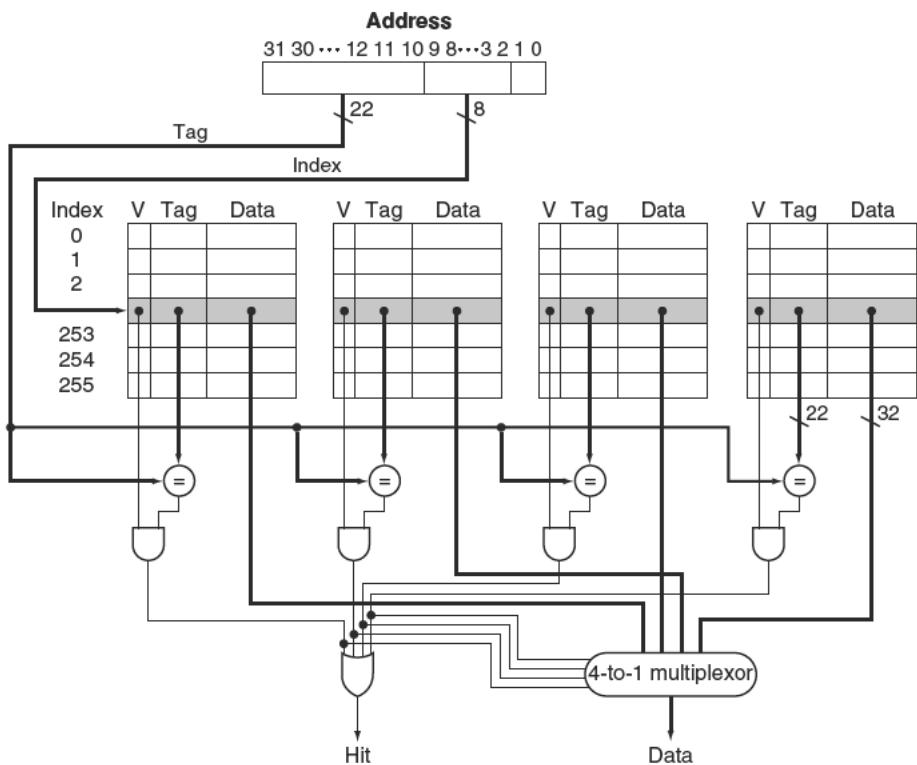
**FIGURE 5.17** The three portions of an address in a set-associative or direct-mapped cache. The index is used to select the set, then the tag is used to choose the block by comparison with the blocks in the selected set. The block offset is the address of the desired data within the block.

## 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.

If the total cache size is kept the same, increasing the associativity increases 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 increases 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 entire 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 supply 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 2-way and 4-way set associativity being built from standard SRAMs and comparators, with 8-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 405 uses LRU, which is why we replaced Memory(0) instead of Memory(6).

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.

**least recently used (LRU)** A replacement scheme in which the block replaced is the one that has been unused for the longest time.

### Size of Tags versus Set Associativity

Increasing associativity requires more comparators and more tag bits per cache block. Assuming a cache of 4096 blocks, a 4-word block size, and a 32-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.

### EXAMPLE

Since there are  $16 (= 2^4)$  bytes per block, a 32-bit address yields  $32 - 4 = 28$  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  $(28 - 12) \times 4096 = 16 \times 4096 = 66\text{ K}$  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  $(28 - 11) \times 2 \times 2048 = 34 \times 2048 = 70\text{ K}$  bits. For a four-way set-associative cache, the total number of sets is 1024, and the total number is  $(28 - 10) \times 4 \times 1024 = 72 \times 1024 = 74\text{ K}$  tag bits.

For a fully associative cache, there is only one set with 4096 blocks, and the tag is 28 bits, leading to  $28 \times 4096 \times 1 = 115\text{ K}$  tag bits.

### ANSWER

## 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.

### EXAMPLE

### ANSWER

#### Performance of Multilevel Caches

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%?

The miss penalty to main memory is

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

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

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

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 ( $(2\% - 0.5\%) \times 20 = 0.3$ ). 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  $(0.5\% \times (20 + 400) = 2.1)$ . 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.

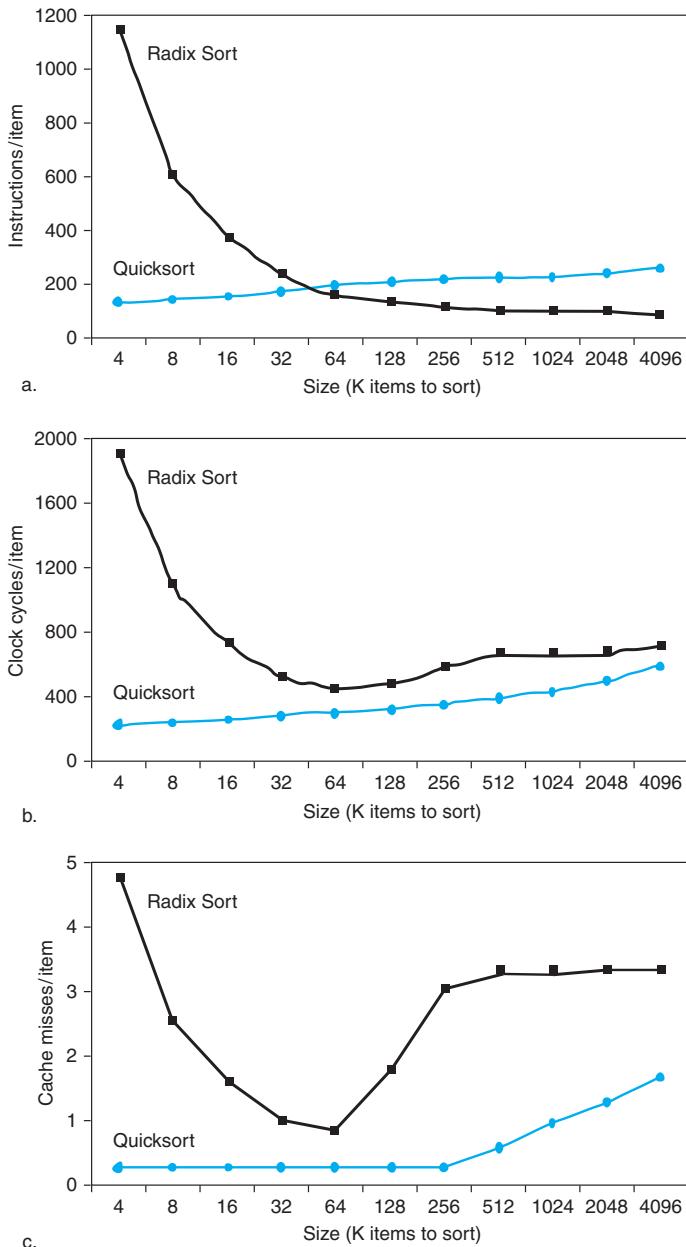
The effect of these changes on the two caches can be seen by comparing each cache to the optimal design for a single level of cache. In comparison to a single-level cache, the primary cache of a **multilevel cache** is often smaller. Furthermore, the primary cache may use a smaller block size, to go 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.

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

## Understanding Program Performance



**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.** This data is 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 it is replaced on a miss.

diverges as the data to sort increases. 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.

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 of the performance out of a stream of instructions, using the memory hierarchy well is critical 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.

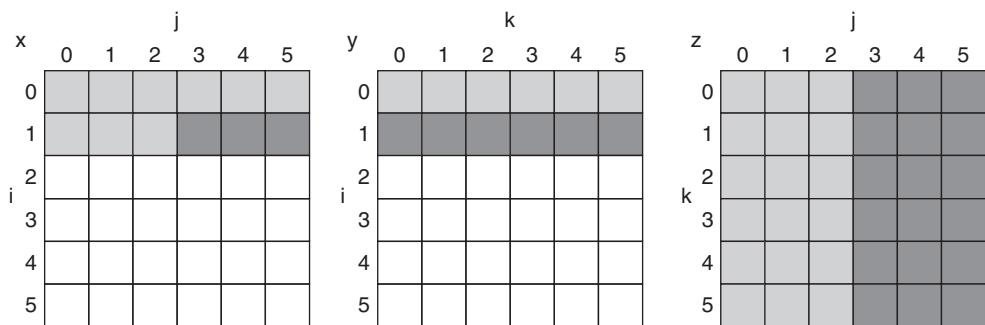
When dealing with arrays, we can get good performance from the memory system if we store the array in memory so that accesses to the array are sequential in memory. Suppose that we are dealing with multiple arrays, however, with some arrays accessed by rows and some by columns. Storing the arrays row-by-row (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 3.21 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 5.21, elements of A and B are read repeatedly to calculate new elements of x. 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  $32 \times 32 = 1024$  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  $i$ th 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  $2N^3 + N^2$  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 4.80 in Chapter 4 repeatedly on matrices of size BLOCKSIZE by BLOCKSIZE. BLOCKSIZE is called the *blocking factor*.

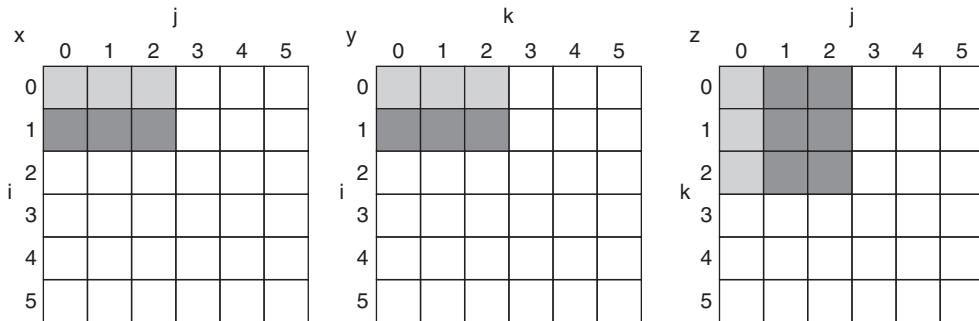
Figure 5.21 shows the blocked version of DGEMM. The function do\_block is DGEMM from Figure 3.21 with three new parameters si, sj, and sk to specify the starting position of each submatrix of A, B, and C. The two inner loops of the do\_block now compute in steps of size BLOCKSIZE rather than the full 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.

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  $2N^3/\text{BLOCKSIZE} + N^2$ . 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.

```

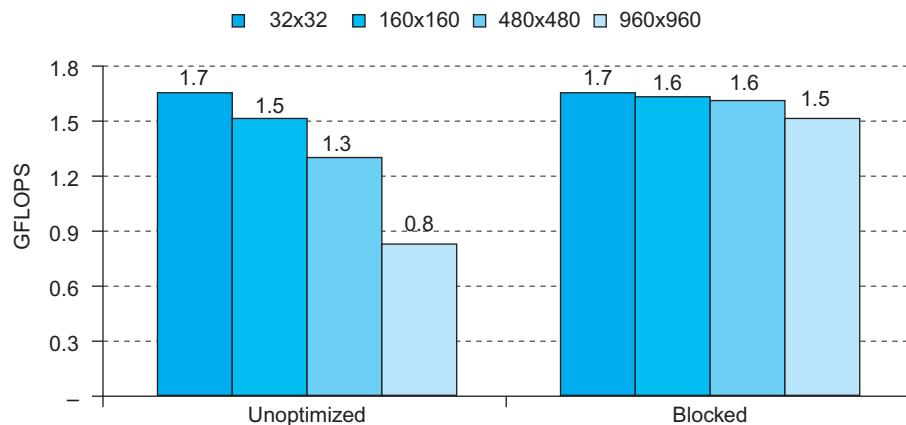
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5     for (int i = si; i < si+BLOCKSIZE; ++i)
6         for (int j = sj; j < sj+BLOCKSIZE; ++j)
7         {
8             double cij = C[i+j*n];/* cij = C[i][j] */
9             for( int k = 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 3.21.** 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 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 also improves performance.



**FIGURE 5.23 Performance of unoptimized DGEMM (Figure 3.21) versus cache blocked DGEMM (Figure 5.21) as the matrix dimension varies from 32x32 (where all three matrices fit in the cache) to 960x960.**

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 960x960, or 900 times larger than the 32 × 32 matrices in Chapters 3 and 4.

**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:** Multilevel caches create several complications. First, there are now several different types of misses and corresponding miss rates. In the example on pages 410–411, 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 410–411, 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.

**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:

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

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*.

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.

### Check Yourself

## 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 ten 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

Some disks today are quoted to have a 1,000,000-hour MTTF. As 1,000,000 hours is  $1,000,000 / (365 \times 24) = 114$  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 2 disks. Use AFR to calculate how many disks we would expect to fail per year.

## EXAMPLE

One year is  $365 \times 24 = 8760$  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 2 disk failures per day!

## ANSWER

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 example, a very good Internet service today offers 4 or 5 nines of availability. Given 365 days per year, which is  $365 \times 24 \times 60 = 526,000$  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 valid or not, we can detect single bit errors, and can say we have a single bit **error detection code**.

Hamming used a *parity code* for error detection. In a parity code, the number of 1s in a word is counted; the word has odd parity if the number of 1s 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.

### 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.

## EXAMPLE

Calculate the parity of a byte with the value  $31_{\text{ten}}$  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

$31_{\text{ten}}$  is  $00011111_{\text{two}}$ , which has five 1s. To make parity even, we need to write a 1 in the parity bit, or  $00011111_1_{\text{two}}$ . If the most significant bit is inverted when we read it back, we would see  $10011111_{\text{two}}$  which has seven 1s. 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  $11011111_{\text{two}}$  which has eight 1s 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 3 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, as opposed 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 ( $0001_{\text{two}}$ ) checks bits (1,3,5,7,9,11,...), which are bits where rightmost bit of address is 1 ( $0001_{\text{two}}, 0011_{\text{two}}, 0101_{\text{two}}, 0111_{\text{two}}, 1001_{\text{two}}, 1011_{\text{two}}, \dots$ ).
    - Bit 2 ( $0010_{\text{two}}$ ) 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 ( $0100_{\text{two}}$ ) 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 ( $1000_{\text{two}}$ ) 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
	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 then 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 ( $p_8, p_4, p_2, p_1$ ) was 0000, then there was no error. However, if the pattern was, say, 1010, which is  $10_{\text{ten}}$ , 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****ANSWER**

Assume one byte data value is  $10011010_{\text{two}}$ . 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  $\underline{\phantom{0}} \underline{\phantom{0}} 1 \underline{\phantom{0}} 0 1 \underline{\phantom{0}} 1 0 1 0$ . Position 1 checks bits 1,3,5,7,9, and 11, which we highlight:  $\underline{\phantom{0}} 1 \underline{\phantom{0}} 0 0 1 \underline{\phantom{0}} 1 0 1 0$ . 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  $0 \underline{\phantom{1}} 1 \underline{\phantom{0}} 0 0 1 \underline{\phantom{0}} 1 0 1 0$  or odd parity, so we set position 2 to a 1.

Position 4 checks bits 4,5,6,7,12, which is  $0 1 1 \underline{\phantom{0}} 0 0 1 \underline{\phantom{0}} 1 0 1$ , so we set it to a 1. Position 8 checks bits 8,9,10,11,12, which is  $0 1 1 1 0 0 1 \underline{\phantom{0}} 1 0 1 0$ , so we set it to a 0.

The final code word is 011100101010. Inverting bit 10 changes it to 011100101110.

Parity bit 1 is 0 (011100101110 is four 1s, so even parity; this group is OK). Parity bit 2 is 1 (011100101110 is five 1s, so odd parity; there is an error somewhere).

Parity bit 4 is 1 (011100101110 is two 1s, so even parity; this group is OK).

Parity bit 8 is 1 (011100101110 is three 1s, so odd parity; there is an error somewhere).

Parity bits 2 and 10 are incorrect. As  $2 + 8 = 10$ , bit 10 must be wrong. Hence, we can correct the error by inverting bit 10: 011100101010. 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 four-bit data word as an example, which would only need 7 bits for single bit error detection. Hamming parity bits H (p<sub>1</sub> p<sub>2</sub> p<sub>3</sub>) are computed (even parity as usual) plus the even parity over the entire word, p<sub>4</sub>:

1	2	3	4	5	6	7	<b>8</b>
p <sub>1</sub>	p <sub>2</sub>	d <sub>1</sub>	p <sub>3</sub>	d <sub>2</sub>	d <sub>3</sub>	d <sub>4</sub>	<b>p<sub>4</sub></b>

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<sub>4</sub>). There are four cases:

1. H is even and p<sub>4</sub> is even, so no error occurred.
2. H is odd and p<sub>4</sub> is odd, so a correctable single error occurred. (p<sub>4</sub> should calculate odd parity if one error occurred.)
3. H is even and p<sub>4</sub> is odd, a single error occurred in p<sub>4</sub> bit, not in the rest of the word, so correct the p<sub>4</sub> 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, eight 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 very large 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 three 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 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 is considered coefficients of a polynomials 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.

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.

## Hardware/ Software Interface



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 them. 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 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 currently 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 requirements of system virtual:

- 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 only 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 architecture proudly bears 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 effect 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 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 will be as efficient as the IBM mainframes and VMMs of the 1970s.

## 5.7

## Virtual Memory

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

*... 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

**virtual memory**

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

the secondary storage, usually 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 multiple 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.

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 only 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 efficiently share the processor 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 the virtual machines 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.

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 never 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

**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.

**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.

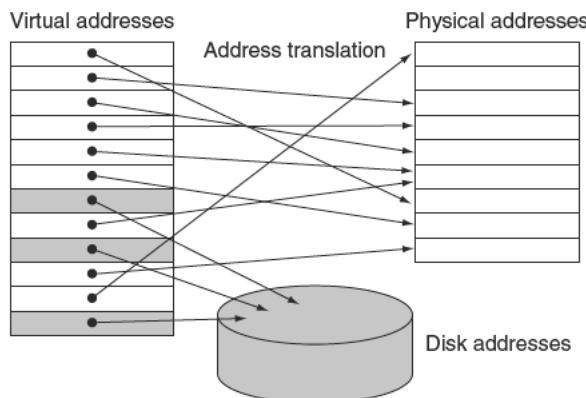
**address translation.** Today, the two memory hierarchy levels controlled by virtual memory are usually DRAMs and flash memory in personal mobile devices and DRAMs and magnetic disks in servers (see Section 5.2). If we return to our library analogy, we can think of a virtual address as the title of 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.

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 need only find a sufficient number of 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*. 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 need not match 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.

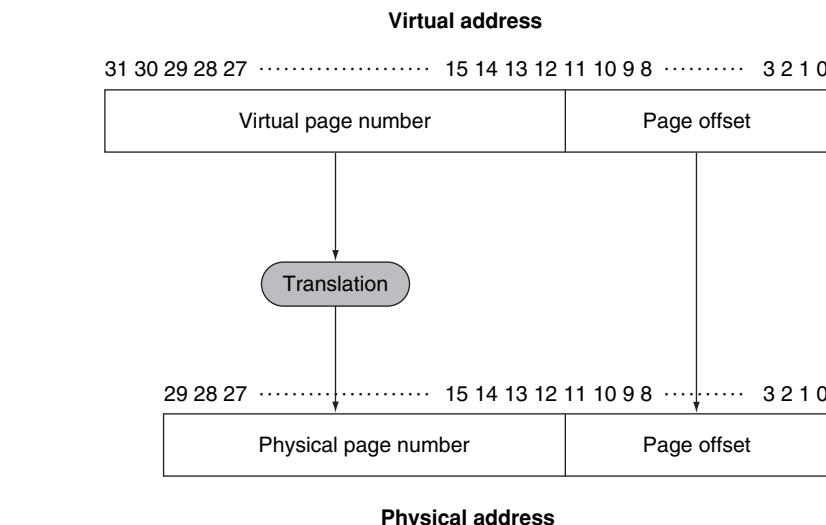
### address translation

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.



**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  $2^{18}$ , since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

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 378 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 16 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 small 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:** 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 to 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 logically share the address space. 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 memory; this structure is called a **page table**, and it resides in memory. A page table is indexed with 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.

---

## Hardware/ Software Interface

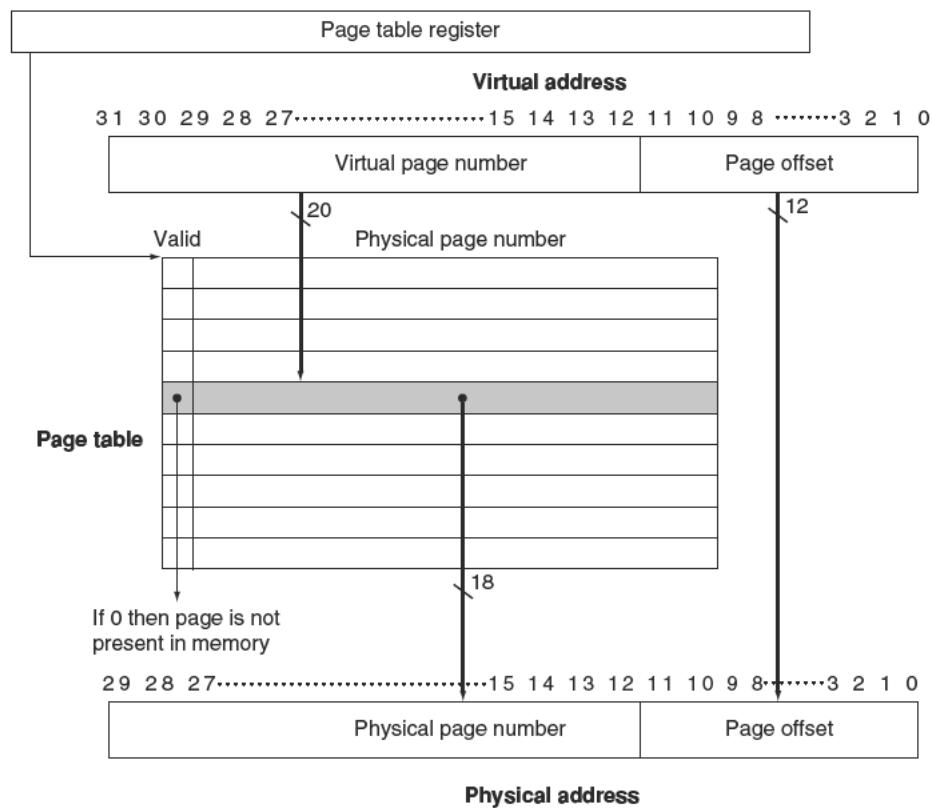
The page table, together with the program counter and the registers, specifies the *state* of a virtual machine. If we want to allow another virtual machine to use the processor, we must save this state. Later, after restoring this state, the virtual machine can continue execution. We often refer to 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 different 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.

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 is the virtual 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 32-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  $2^{32}$  bytes, or 4 GiB, and the physical address space is  $2^{30}$  bytes, which allows main memory of up to 1 GiB. The number of entries in the page table is  $2^{20}$ , or 1 million entries. 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 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

## Page Faults

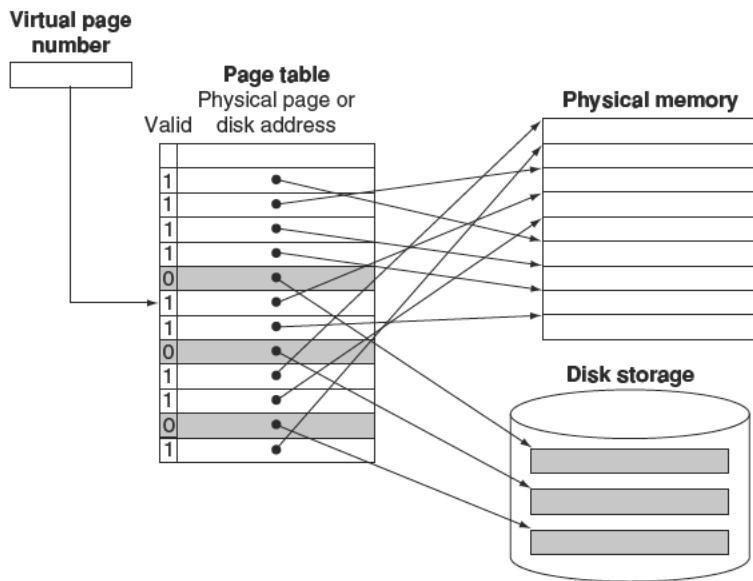
If the valid bit for a virtual page is off, a page fault occurs. The operating system must be given control. This transfer is done with the exception mechanism, which we saw in Chapter 4 and will discuss again later in this section. Once the operating system gets control, it must find the page in the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

The virtual address alone does not immediately tell us where the page is on disk. 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 on disk 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 disk address.

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 in the near future. 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 on the disk. 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.

**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.

Implementing a completely accurate LRU scheme is too expensive, since it requires updating a data structure on *every* memory reference. Instead, 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, some computers provide a **reference bit** or **use 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.

## Hardware/ Software Interface

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

**Elaboration:** With a 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry, we can compute the total page table size:

$$\text{Number of page table entries} = \frac{2^{32}}{2^{12}} = 2^{20}$$

$$\text{Size of page table} = 2^{20} \text{ page table entries} \times 2^2 \frac{\text{bytes}}{\text{page table entry}} = 4 \text{ MiB}$$

That is, we would need to use 4 MiB of memory for each program in execution at any time. This amount is not so bad for a single process. What if there are hundreds of processes running, each with their own page table? And how should we handle 64-bit addresses, which by this calculation would need  $2^{52}$  words?

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 only 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. This type of segmentation is used by many architectures, including MIPS. Unlike the type of segmentation discussed in the third elaboration on page 431, 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. Multiple levels of page tables can also be used to reduce the total amount of page table storage. The first level maps large fixed-size blocks of virtual address space, perhaps 64 to 256 pages in total. These large blocks are sometimes called *segments*, and this first-level mapping table is sometimes called a

segment table, though the segments are again invisible to the user. Each entry in the segment table indicates whether any pages in that segment are allocated and, if so, points to a page table for that segment. Address translation happens by first looking in the segment table, using the highest-order bits of the address. If the segment address is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry. 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 two-level mapping is the more complex process for address translation.

5. 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 on disk.

## 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 disk when it is replaced in the memory.

---

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 more efficient than transferring individual 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.

---

## Hardware/ Software Interface

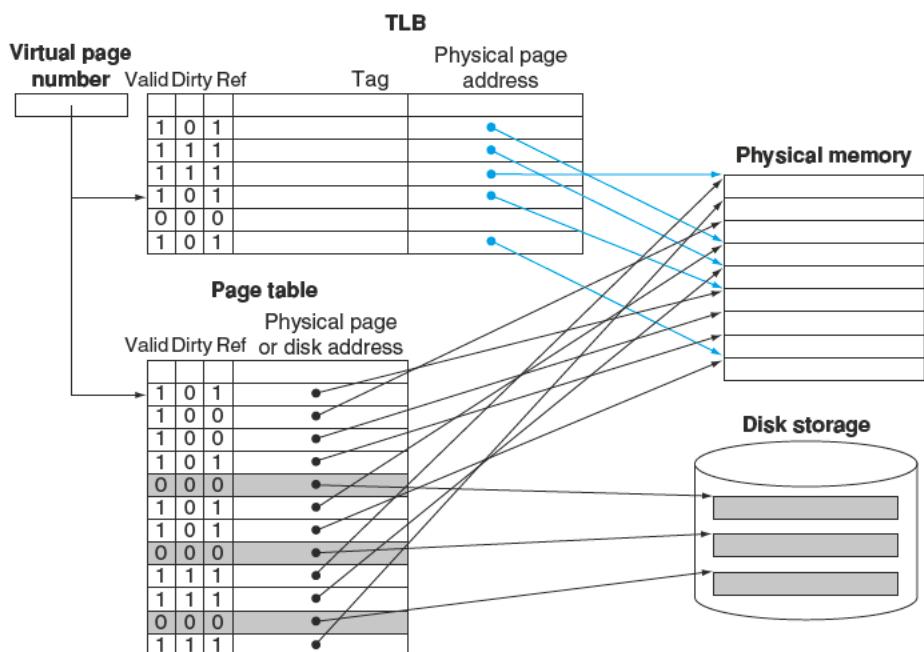
## Making Address Translation Fast: the TLB

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 in the near future, 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.

Figure 5.29 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.

**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 5.29 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.

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.

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 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 Intrinsicity FastMATH TLB

To see these ideas in a real processor, let's take a closer look at the TLB of the Intrinsicity FastMATH. The memory system uses 4 KiB pages and a 32-bit address space; thus, the virtual page number is 20 bits long, as in the top of [Figure 5.30](#). 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.

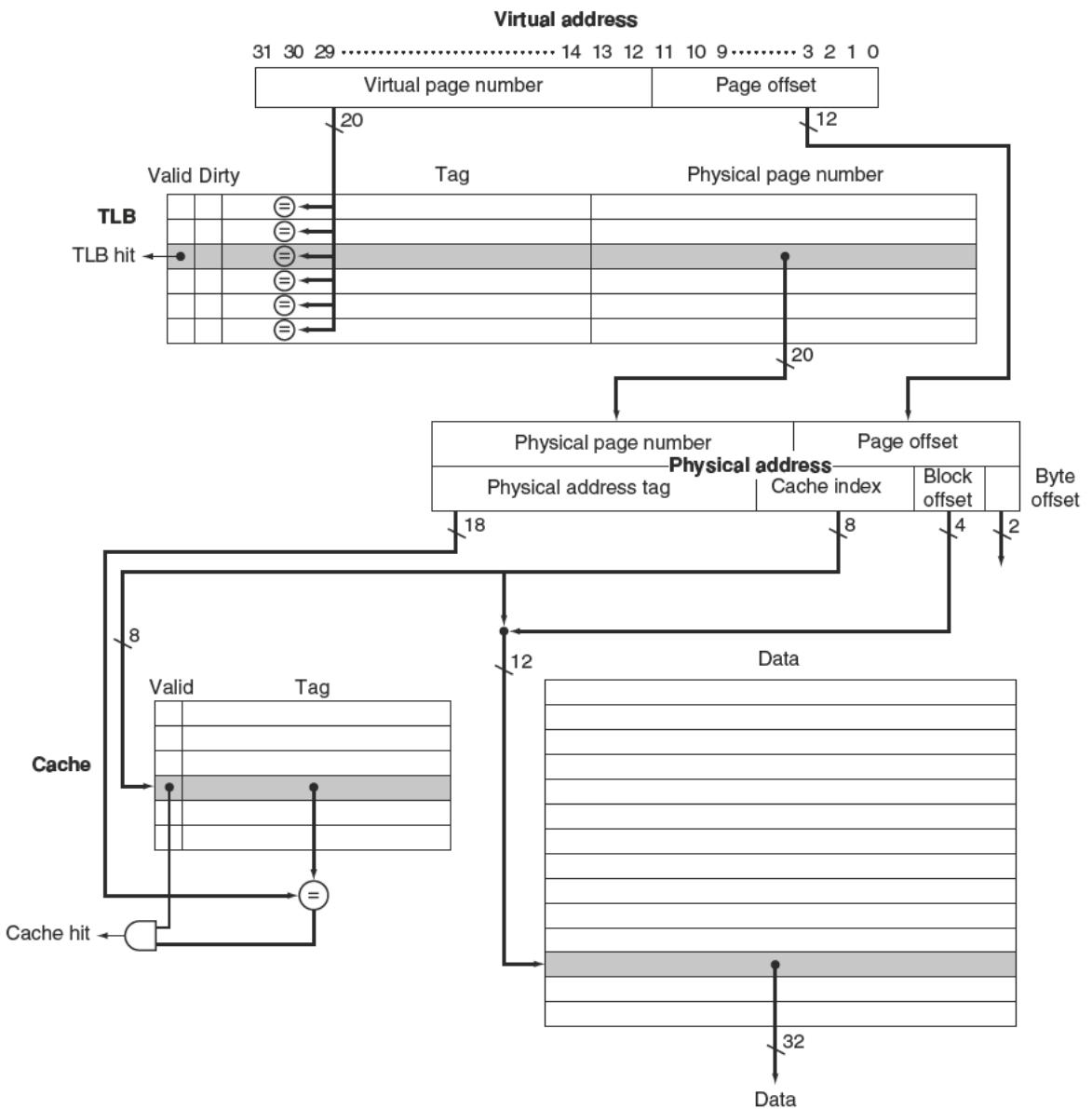
[Figure 5.30](#) shows the TLB and one of the caches, while [Figure 5.31](#) shows the steps in processing a read or write request. When a TLB miss occurs, the MIPS hardware saves the page number of the reference in a special register and generates an exception. The exception invokes the operating system, which handles the miss in software. To find the physical address for the missing page, the TLB miss routine 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. (We will see the MIPS TLB code on page 449.) 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; the recommended entry is chosen randomly.

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.

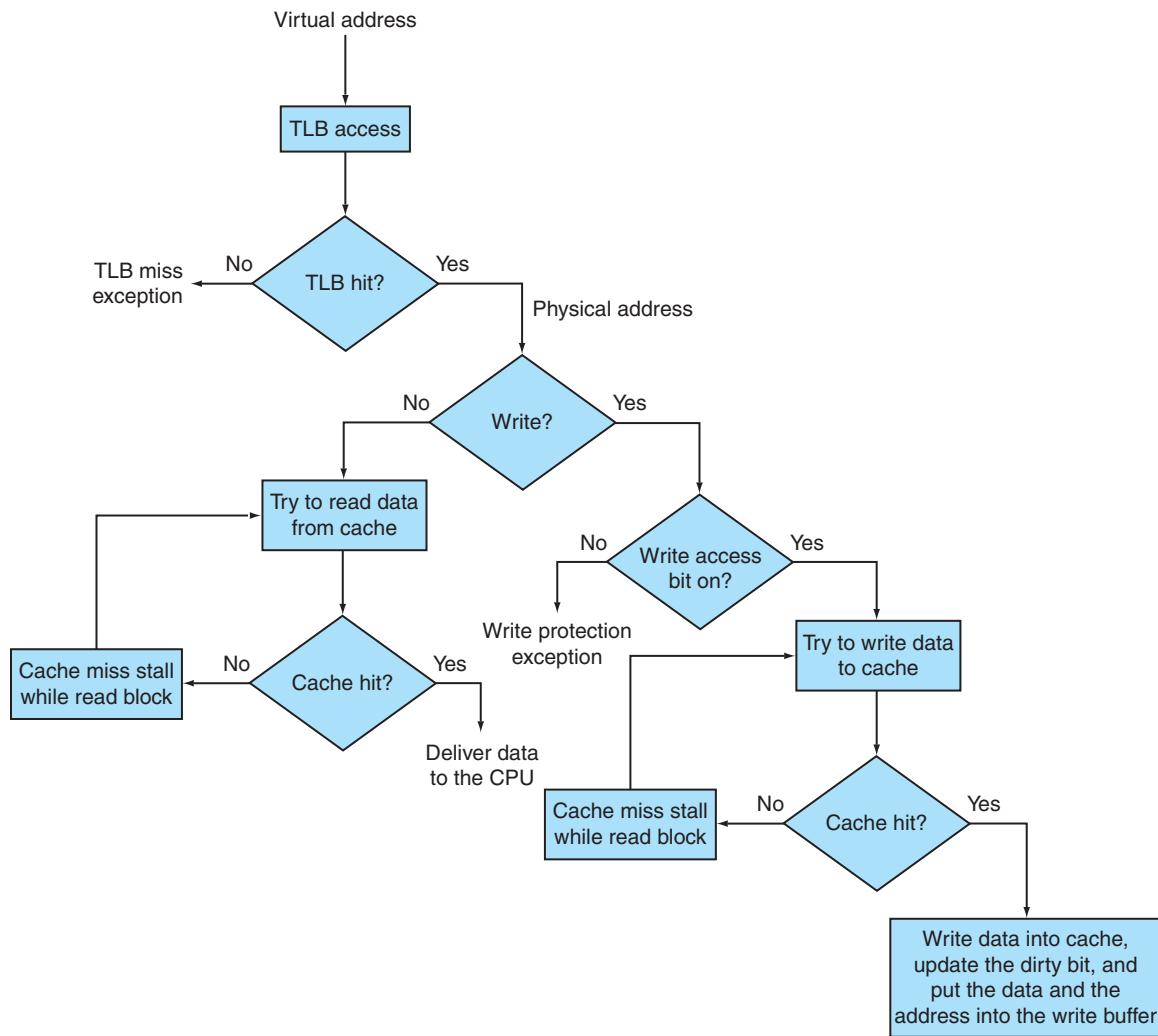
### Integrating Virtual Memory, TLBs, and Caches

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 disk. 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 is 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.



**FIGURE 5.30 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. This diagram focuses on a read; Figure 5.31 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 408.) 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 5.31 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 is brought from memory. If the operation is a write, a portion of the cache entry is overwritten for a hit and the data is 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.

### Overall Operation of a Memory Hierarchy

In a memory hierarchy like that of Figure 5.30, 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.

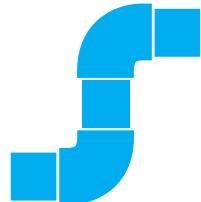
### EXAMPLE

Figure 5.32 shows all combinations and whether each is possible in practice.

### ANSWER

**Elaboration:** Figure 5.32 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.



PIPELINING

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

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 5.32 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, virtual memory hit, cache miss) but never detected.

**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.

**physically addressed cache** A cache that is addressed by 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 different 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.

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 5.30 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.

## 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

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

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 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 (*syscall* in the MIPS 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 exception PC (EPC), and the processor is placed in supervisor mode. To return to user mode from the exception, use the *return from exception* (ERET) instruction, which resets to user mode and jumps to the address in EPC.

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.

**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 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*.

**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.

**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. 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 different solutions, such as process identifiers, are used to ensure that a process gets its own data.

## 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.

MIPS traditionally handles a TLB miss in software. It brings in the page table entry from memory and then re-executes the instruction that caused the TLB miss. Upon re-executing, it will get a TLB hit. If the page table entry indicates the page is not in memory, this time it will get a page fault exception.

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. Just as in Chapter 4, the *exception program counter* (EPC) 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 `lw $1,0($1)`: 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 `$1` 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.

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 example, if another exception occurred when we were processing the first exception in the operating system, the control unit would overwrite the exception program counter, 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 *exception program counter* (EPC) and Cause registers. EPC and Cause are two of the special control registers that help with exceptions, TLB misses, and page faults; [Figure 5.33](#) shows the rest. 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.

## Hardware/ Software Interface

**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 on disk.
2. Choose a physical page to replace; if the chosen page is dirty, it must be written out to disk before we can bring a new virtual page into this physical page.
3. Start a read to bring the referenced page from disk into the chosen physical page.

Register	CP0 register number	Description
EPC	14	Where to restart after exception
Cause	13	Cause of exception
BadVAddr	8	Address that caused exception
Index	0	Location in TLB to be read or written
Random	1	Pseudorandom location in TLB
EntryLo	2	Physical page address and flags
EntryHi	10	Virtual page address
Context	4	Page table address and page number

**FIGURE 5.33 MIPS control registers.** These are considered to be in coprocessor 0, and hence are read using `mfc0` and written using `mtc0`.

Of course, this last step will take millions of processor clock cycles (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 disk 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.

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

Making instructions **restartable**, so that the exception can be handled and the instruction later continued, is relatively easy in an architecture like the MIPS. 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.

Let's look in more detail at MIPS. When a TLB miss occurs, the MIPS hardware saves the page number of the reference in a special register called `BadVAddr` and generates an exception.

The exception invokes the operating system, which handles the miss in software. Control is transferred to address  $8000\ 0000_{hex}$ , the location of the TLB miss **handler**. To find the physical address for the missing page, the TLB miss routine 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. To make this indexing fast, MIPS hardware places everything you need in the special Context register: the upper 12 bits have the address of the base of the page table, and the next 18 bits have the virtual address of the missing page. Each page table entry is one word, so the last 2 bits are 0. Thus, the first two instructions copy the Context register into the kernel temporary register  $\$k1$  and then load the page table entry from that address into  $\$k1$ . Recall that  $\$k0$  and  $\$k1$  are reserved for the operating system to use without saving; a major reason for this convention is to make the TLB miss handler fast. Below is the MIPS code for a typical TLB miss handler:

```
TLBmiss:
    mfc0 $k1,Context      # copy address of PTE into temp $k1
    lw   $k1,0($k1)        # put PTE into temp $k1
    mtc0 $k1,EntryLo       # put PTE into special register EntryLo
    tlbwr                   # put EntryLo into TLB entry at Random
    eret                    # return from TLB miss exception
```

As shown above, MIPS has a special set of system instructions to update the TLB. The instruction `tlbwr` copies from control register `EntryLo` into the TLB entry selected by the control register `Random`. `Random` implements random replacement, so it is basically a free-running counter. A TLB miss takes about a dozen clock cycles.

Note that the TLB miss handler does not check to see if the page table entry is valid. Because the exception for TLB entry missing is much more frequent than a page fault, the operating system loads the TLB from the page table without examining the entry and restarts the instruction. If the entry is invalid, another and different exception occurs, and the operating system recognizes the page fault. This method makes the frequent case of a TLB miss fast, at a slight performance penalty for the infrequent case of a page fault.

Once the process that generated the page fault has been interrupted, it transfers control to  $8000\ 0180_{hex}$ , a different address than the TLB miss handler. This is the general address for exception; TLB miss has a special entry point to lower the penalty for a TLB miss. The operating system uses the exception Cause register to diagnose the cause of the exception. Because the exception is a page fault, the operating system knows that extensive processing will be required. Thus, unlike a TLB miss, it saves the entire state of the active process. This state includes all the general-purpose and floating-point registers, the page table address register, the EPC, and the exception Cause register. Since exception handlers do not usually use the floating-point registers, the general entry point does not save them, leaving that to the few handlers that need them.

**handler** Name of a software routine invoked to “handle” an exception or interrupt.

[Figure 5.34](#) sketches the MIPS code of an exception handler. Note that we save and restore the state in MIPS code, taking care when we enable and disable exceptions, but we invoke C code to handle the particular exception.

The virtual address that caused the fault depends on whether the fault was an instruction or data fault. The address of the instruction that generated the fault is in the EPC. If it was an instruction page fault, the EPC contains the virtual address of the faulting page; otherwise, the faulting virtual address can be computed by examining the instruction (whose address is in the EPC) to find the base register and offset field.

**unmapped** A portion of the address space that cannot have page faults.

**Elaboration:** This simplified version assumes that the stack pointer (sp) is valid. To avoid the problem of a page fault during this low-level exception code, MIPS sets aside a portion of its address space that cannot have page faults, called **unmapped**. The operating system places the exception entry point code and the exception stack in unmapped memory. MIPS hardware translates virtual addresses  $8000\ 0000_{hex}$  to  $BFFF\ FFFF_{hex}$  to physical addresses simply by ignoring the upper bits of the virtual address, thereby placing these addresses in the low part of physical memory. Thus, the operating system places exception entry points and exception stacks in unmapped memory.

**Elaboration:** The code in [Figure 5.34](#) shows the MIPS-32 exception return sequence. The older MIPS-I architecture uses rfe and jr instead of eret.

**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 MIPS 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 VMM 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.

Save state		
Save GPR	addi \$k1,\$sp, -XCPSIZE sw \$sp, XCT_SP(\$k1) sw \$v0, XCT_V0(\$k1) ... sw \$ra, XCT_RA(\$k1)	# save space on stack for state # save \$sp on stack # save \$v0 on stack # save \$v1, \$ai, \$si, \$ti,... on stack # save \$ra on stack
Save hi, lo	mfhi \$v0 mflo \$v1 sw \$v0, XCT_HI(\$k1) sw \$v1, XCT_LO(\$k1)	# copy Hi # copy Lo # save Hi value on stack # save Lo value on stack
Save exception registers	mfc0 \$a0, \$cr sw \$a0, XCT_CR(\$k1) ... mfc0 \$a3, \$sr sw \$a3, XCT_SR(\$k1)	# copy cause register # save \$cr value on stack # save \$v1,... # copy status register # save \$sr on stack
Set sp	move \$sp, \$k1	# sp = sp - XCPSIZE
Enable nested exceptions		
	andi \$v0, \$a3, MASK1 mtc0 \$v0, \$sr	# \$v0 = \$sr & MASK1, enable exceptions # \$sr = value that enables exceptions
Call C exception handler		
Set \$gp	move \$gp, GPINIT	# set \$gp to point to heap area
Call C code	move \$a0, \$sp jal xcpt_deliver	# arg1 = pointer to exception stack # call C code to handle exception
Restoring state		
Restore most GPR, hi, lo	move \$at, \$sp lw \$ra, XCT_RA(\$at) ... lw \$a0, XCT_A0(\$k1)	# temporary value of \$sp # restore \$ra from stack # restore \$t0,..., \$a1 # restore \$a0 from stack
Restore status register	lw \$v0, XCT_SR(\$at) li \$v1, MASK2 and \$v0, \$v0, \$v1 mtc0 \$v0, \$sr	# load old \$sr from stack # mask to disable exceptions # \$v0 = \$sr & MASK2, disable exceptions # set status register
Exception return		
Restore \$sp and rest of GPR used as temporary registers	lw \$sp, XCT_SP(\$at) lw \$v0, XCT_V0(\$at) lw \$v1, XCT_V1(\$at) lw \$k1, XCT_EPC(\$at) lw \$at, XCT_AT(\$at)	# restore \$sp from stack # restore \$v0 from stack # restore \$v1 from stack # copy old \$epc from stack # restore \$at from stack
Restore ERC and return	mtc0 \$k1, \$epc eret \$ra	# restore \$epc # return to interrupted instruction

FIGURE 5.34 MIPS code to save and restore state on an exception.

**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 increasing 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 specified either via page tables, as in IBM VM/370 and the x86, or via the TLB structure, as in MIPS.

## 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 unchanged 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 changing the page tables. Controlled sharing of pages among 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.

---

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 memory and disk, 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:  $64 \times 4 \text{ KiB} = 0.25 \text{ MiB}$ . For example, TLB misses are often a challenge for Radix Sort. To try to alleviate this problem, most computer architectures now support variable page sizes. For example, in addition to the standard 4 KiB page, MIPS hardware supports 16 KiB, 64 KiB, 256 KiB, 1 MiB, 4 MiB, 16 MiB, 64 MiB, and 256 MiB 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

## Understanding Program Performance

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.

### Check Yourself

Match the definitions in the right column to the terms in the left column.

- |                |                                   |
|----------------|-----------------------------------|
| 1. L1 cache    | a. A cache for a cache            |
| 2. L2 cache    | b. A cache for disks              |
| 3. Main memory | c. A cache for a main memory      |
| 4. TLB         | d. A cache for page table entries |

## 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 5.35 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 as 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	2,500–25,000	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%

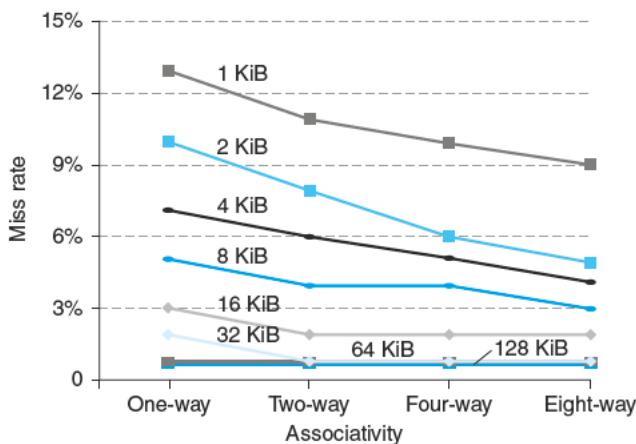
**FIGURE 5.35 The key quantitative design parameters that characterize the major elements of memory hierarchy in a computer.** These are typical values for these levels as of 2012. Although the range of values is wide, this is partially because many of the values that have shifted over time are related; 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	Number of blocks in the cache	Associativity (typically 2–16)
	Associativity	
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 5.36](#) 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



**FIGURE 5.36 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 this data was collected.

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.

## 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 required 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. For example, MIPS supports random replacement for TLB misses.
- *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 costly. 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 only 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 practical, 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**):

- **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**.

Figure 5.37 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 directly from contention for the same cache block, increasing associativity reduces conflict misses. Associativity, however, may slow access time, leading to lower overall performance.

Capacity misses can easily be reduced by enlarging the cache; indeed, second-level caches have been growing steadily larger 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.

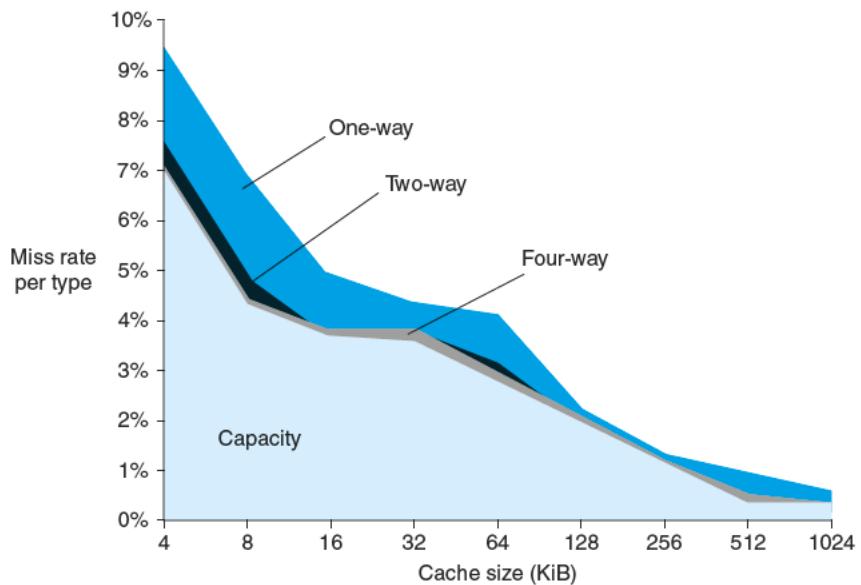
Because compulsory misses are generated by the first reference to a block, the primary way for the cache system to reduce the number of compulsory misses is to increase the block size. This will reduce the number of references required to touch each block of the program once, because the program will consist of fewer

**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 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 5.37 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. This data is for the SPEC CPU2000 integer and floating-point benchmarks and is from the same source as the data in Figure 5.36. 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.

## 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 5.38 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 5.38 Memory hierarchy design challenges.**

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 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.

Which of the following statements (if any) are 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.

### Check Yourself

## 5.9

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

We can now 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 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 simple cache. Here are the key characteristics of the cache:

- Direct-mapped cache

- Write-back using write allocate
- Block size is 4 words (16 bytes or 128 bits)
- Cache size is 16 KiB, so it holds 1024 blocks
- 32-byte 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 a set of 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.

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 thus may not require any gates. A simple example of a finite-state machine appears in Appendix B, and if you are unfamiliar with the concept of a finite-state machine, you may want to examine Appendix B 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 5.39 shows how such an implementation might look. Appendix D describes in detail how the finite-state machine is implemented using this structure. In Section B.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 B 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.

**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 only the next-state output.

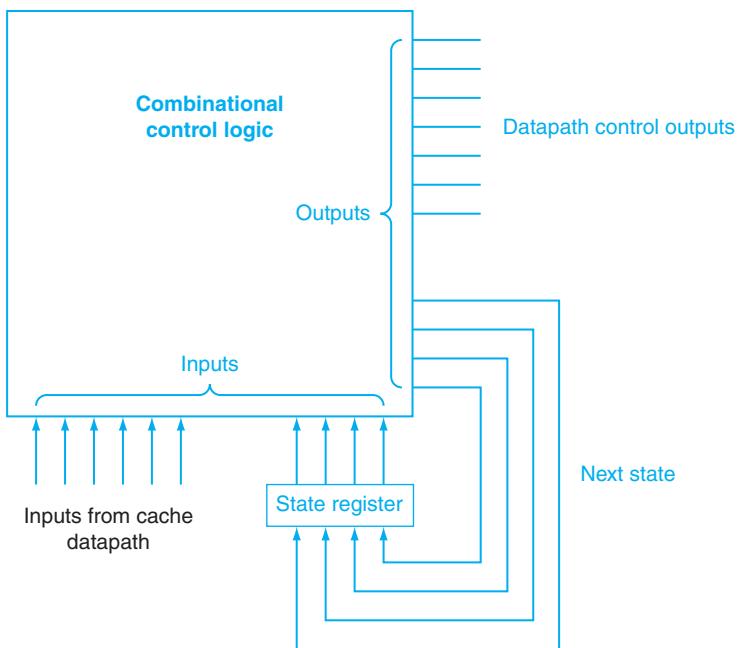
An alternative style of machine is a Mealy machine, named after George Mealy. The Mealy machine allows both the input and the current state to be used to determine the output. Moore machines have potential implementation advantages in speed and size of the control unit. The speed advantages arise because the control outputs, which are

### 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.



**FIGURE 5.39 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. The *Elaboration* explains this in more detail.

needed early in the clock cycle, do not depend on the inputs, but only on the current state. In Appendix B, 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 5.40 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 is valid, and the tag portion of the address matches the tag, then it is a hit. Either the data is 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

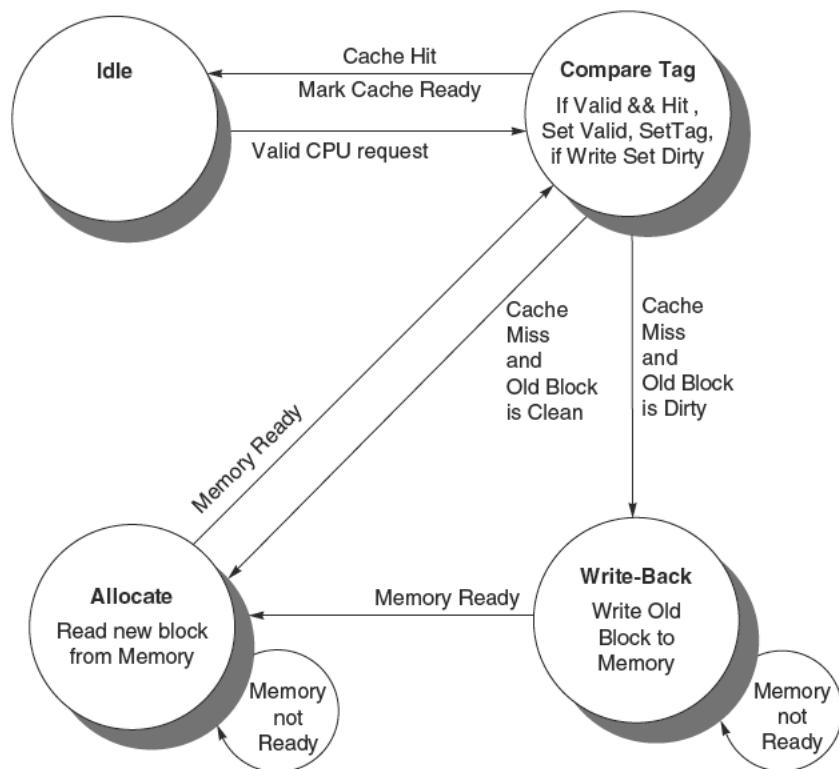


FIGURE 5.40 Four states of the simple controller.

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 also 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.

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 then 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 different values. [Figure 5.41](#) 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*.

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 5.41](#), 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 5.41](#), 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 example. 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 same 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.

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 5.41 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!

- *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 5.42 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 simultaneously, 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.

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 5.42 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.

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 full 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.

## Hardware/ Software Interface

**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 pages 466 and 467 are sufficient to ensure coherence, the question of when a written value will be seen is also important. To see why, observe that we cannot require that a read of X in Figure 5.41 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 forces 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.



## 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 much greater dependability offered by including a modest number of redundant disks. The section explains the differences in performance, cost, and **dependability** between the different RAID levels.



## 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.

## 5.13

### Real Stuff: The ARM Cortex-A8 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-A8 and Intel Core i7. This section is based on Section 2.6 of *Computer Architecture: A Quantitative Approach*, 5<sup>th</sup> edition.

Figure 5.43 summarizes the address sizes and TLBs of the two processors. Note that the A8 has two TLBs with a 32-bit virtual address space and a 32-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 the Core i7 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.

Figure 5.44 shows their caches. Keep in mind that the A8 has just one processor or core while the Core i7 has four. Both have identically organized 32 KiB, 4-way set associative, L1 instruction caches (per core) with 64 byte blocks. The A8 uses the same design for data cache, while the Core i7 keeps everything the same except the associativity, which it increases to 8-way. Both use an 8-way set associative unified L2 cache (per core) with 64 byte blocks, although the A8 varies in size from 128 KiB to 1 MiB while the Core i7 is fixed at 256 KiB. As the Core i7 is used for servers, it

Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data Both TLBs are fully associative, with 32 entries, round robin replacement 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, 7 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 5.43 Address translation and TLB hardware for the ARM Cortex-A8 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.

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 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 5.44 Caches In the ARM Cortex-A8 and Intel Core i7 920.



### nonblocking cache

A cache that allows the processor to make references to the cache while the cache is handling an earlier miss.

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.

A significant challenge facing cache designers is to support processors like the A8 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 Core i7 has 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. It also continues 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**, when building out-of-order processors. 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.

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 may only be able to take limited

advantage of this capability, but large servers and multiprocessors often have memory systems capable of handling more than one outstanding miss in parallel.

The Core i7 has a prefetch mechanism for data accesses. It looks at a pattern of data misses and use 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.

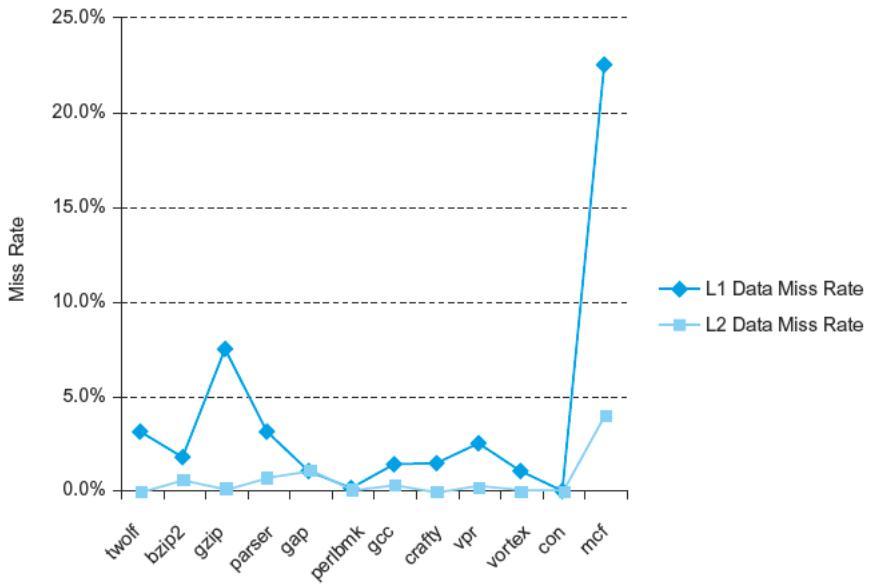
The sophisticated memory hierarchies of these chips and the large fraction of the dies dedicated to caches and TLBs show the significant design effort expended to try to close the gap between processor cycle times and memory latency.

## Performance of the A8 and Core i7 Memory Hierarchies

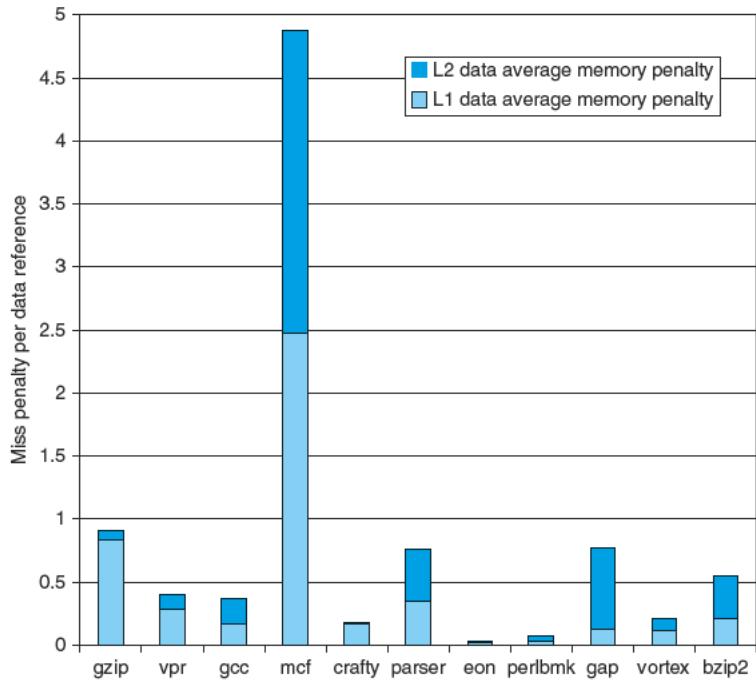
The memory hierarchy of the Cortex-A8 was simulated with a 1 MiB eight-way set associative L2 cache using the integer Minnespec benchmarks. As mentioned in Chapter 4, Minnespec is a set of benchmarks consisting of the SPEC2000 benchmarks but with different inputs that reduce the running times by several orders of magnitude. Although the use of smaller inputs does not change the instruction mix, it does affect the cache behavior. For example, on mcf, the most memory-intensive SPEC2000 integer benchmark, Minnespec has a miss rate for a 32 KiB cache that is only 65% of the miss rate for the full SPEC2000 version. For a 1 MiB cache the difference is a factor of six! For this reason, one cannot compare the Minnespec benchmarks against the SPEC2000 benchmarks, much less the even larger SPEC2006 benchmarks used for the Core i7 in [Figure 5.47](#). Instead, the data are useful for looking at the relative impact of L1 and L2 misses and on overall CPI, which we used in Chapter 4.

The A8 instruction cache miss rates for these benchmarks (and also for the full SPEC2000 versions on which Minnespec is based) are very small even for just the L1: close to zero for most and under 1% for all of them. This low rate probably results from the computationally intensive nature of the SPEC programs and the four-way set associative cache that eliminates most conflict misses. [Figure 5.45](#) shows the data cache results for the A8, which have significant L1 and L2 miss rates. The L1 miss penalty for a 1 GHz Cortex-A8 is 11 clock cycles, while the L2 miss penalty is assumed to be 60 clock cycles. Using these miss penalties, [Figure 5.46](#) shows the average miss penalty per data access.

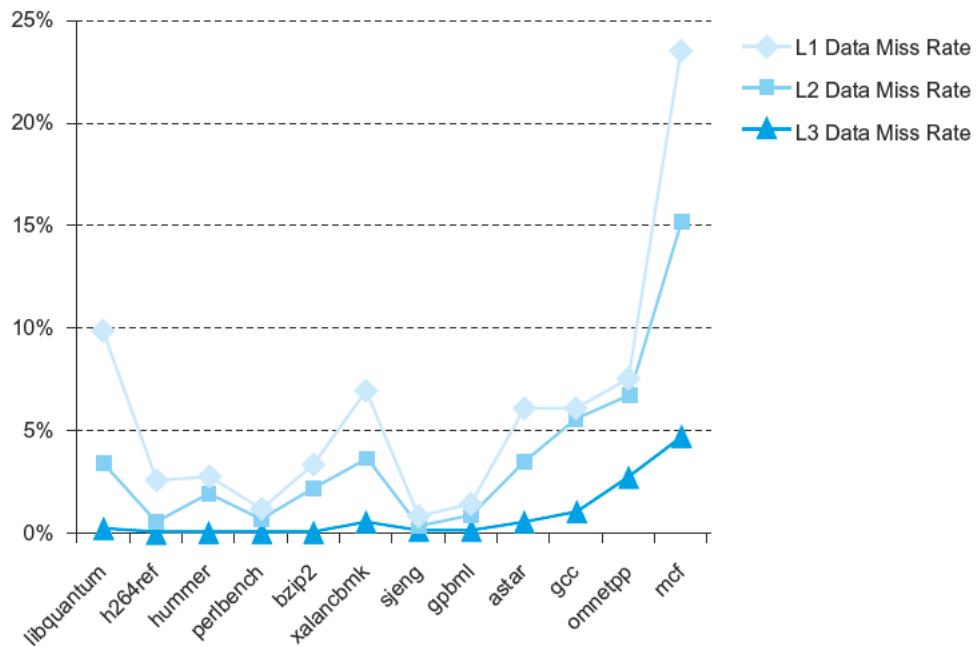
[Figure 5.47](#) shows the miss rates for the caches of the Core i7 using the SPEC2006 benchmarks. The L1 instruction cache miss rate varies from 0.1% to 1.8%, averaging just over 0.4%. This rate is in keeping with other studies of instruction cache behavior for the SPECCPU2006 benchmarks, which show 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 are 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.



**FIGURE 5.45 Data cache miss rates for ARM Cortex-A8 when running Minnespec, a small version of SPEC2000.** 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 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 4.76 in Chapter 4.



**FIGURE 5.46 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 Minnespec.** 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 5.47** The L1, L2, and L3 data cache miss rates for the Intel Core i7 920 running the full Integer SPECCPU2006 benchmarks.

**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 5.45 is 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)

## 5.14

### 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 5.48 shows the blocked version of DGEMM from Figure 4.80. The changes are the same as was made earlier in going from unoptimized DGEMM in Figure 3.21 to blocked DGEMM in Figure 5.21 above. This time we taking the unrolled version of DGEMM from Chapter 4 and invoke it many times on the submatrices

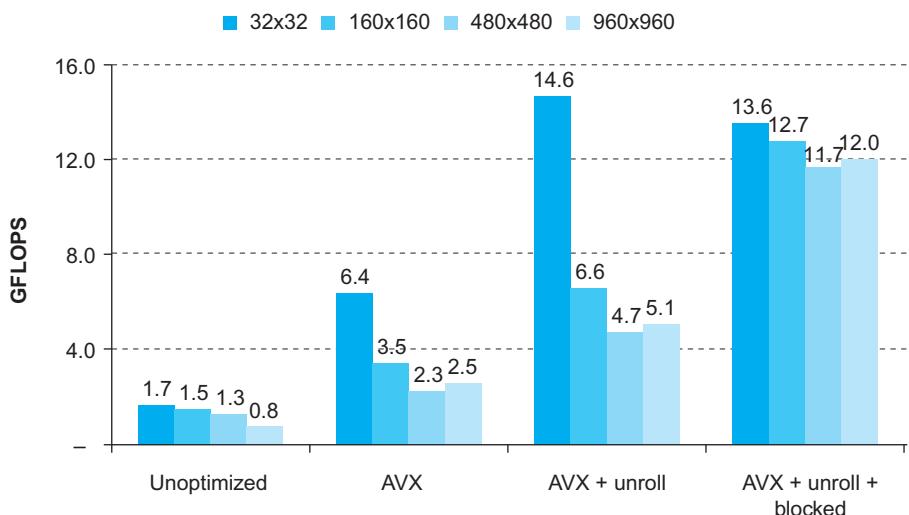
```

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]);
25 /* C[i][j] = c[x] */
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 5.48 Optimized C version of DGEMM from Figure 4.80 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 4.81. Once again, there is no overhead to call the `do_block` because the compiler inlines the function call.

of A, B, and C. Indeed, lines 28 – 34 and lines 7 – 8 in [Figure 5.48](#) are identical to lines 14 – 20 and lines 5 – 6 in [Figure 5.21](#), with the exception of incrementing the for loop in line 7 by the amount unrolled.

Unlike the earlier chapters, we do not show the resulting x86 code because the inner loop code is nearly identical to Figure 4.81, 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 for loops. It expands from 14 instructions before the inner loop and 8 after the loop for Figure 4.80 to 40 and 28 instructions respectively for the bookkeeping code generated for [Figure 5.48](#). Nevertheless, the extra instructions executed pale in comparison to the performance improvement of reducing cache misses. [Figure 5.49](#) compares unoptimized to optimizations 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.



**FIGURE 5.49 Performance of four versions of DGEMM from matrix dimensions 32x32 to 960x960.** The fully optimized code for largest matrix is almost 15 times as fast the unoptimized version in Figure 3.21 in Chapter 3.

**Elaboration:** As mentioned in the Elaboration in Section 3.8, 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.

## 5.15 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.

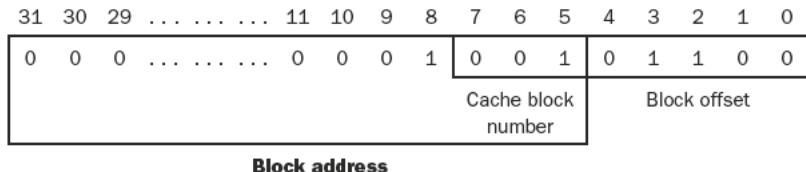
*Pitfall: Ignoring memory system behavior when writing programs or when generating code in a compiler.*

This could easily 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  $(9 \text{ modulo } 8) = 1$ . On the other hand, if address 36 is a word address, then it maps into block  $(36 \text{ mod } 8) = 4$ . 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:



Byte address 300 is block address

$$\left[ \frac{300}{32} \right] = 9$$

The number of blocks in the cache is

$$\left[ \frac{256}{32} \right] = 8$$

Block number 9 falls into cache block number  $(9 \text{ modulo } 8) = 1$ .

This mistake catches many people, including the authors (in earlier drafts) and instructors who forget whether they intended the addresses to be in 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.



*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 399).

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 specified 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.

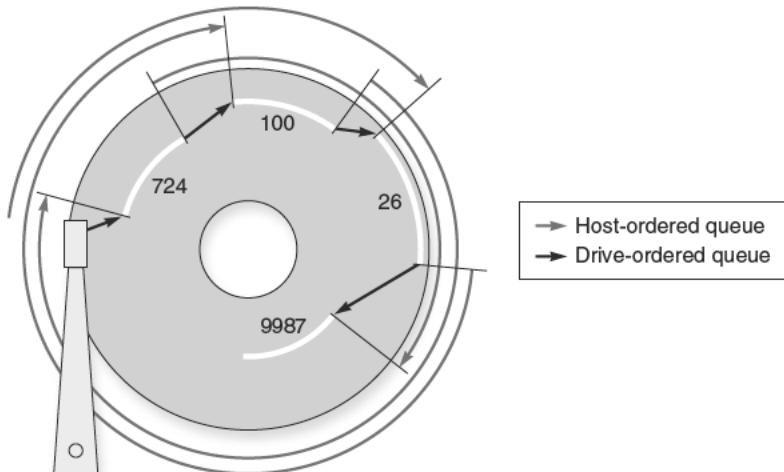
For example, suppose the workload is four reads [Anderson, 2003]:

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.



**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]).

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.

*Pitfall: Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.*

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

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 modifications 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.

## 5.16

### 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 larger 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 large 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.

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.

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.



#### **prefetching**

A technique in which data blocks needed in the future are brought into the cache early by the use of special instructions that specify the address of the block.

As we will see in Chapter 6, memory systems are a central design issue for parallel processors. The growing importance 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.



## 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 Exercises

**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 32-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 32-bit integers can be stored in a 16-byte cache block?

**5.1.2** [5] <§5.1> References to which variables exhibit temporal locality?

**5.1.3** [5] <§5.1> References to which variables 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 by storing 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** [10] <§5.1> How many 16-byte cache blocks are needed to store all 32-bit matrix elements being referenced?

**5.1.5** [5] <§5.1> References to which variables exhibit temporal locality?

**5.1.6** [5] <§5.1> References to which variables exhibit spatial locality?

**5.2** Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 32-bit memory address references, given as word addresses.

3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253

**5.2.1** [10] <§5.3> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list if 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 address, the tag, and the index given a direct-mapped cache with two-word blocks and a total size of 8 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 8 words of data: C1 has 1-word blocks, C2 has 2-word blocks, and C3 has 4-word blocks. In terms of miss rate, which cache design is the best? If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design?

There are many different design parameters that are important to a cache's overall performance. Below are listed parameters for different direct-mapped cache designs.

**Cache Data Size:** 32 KiB

**Cache Block Size:** 2 words

**Cache Access Time:** 1 cycle

**5.2.4** [15] <§5.3> Calculate the total number of bits required for the cache listed above, assuming a 32-bit address. Given that total size, find the total size of the closest direct-mapped cache with 16-word blocks of equal size or greater. Explain why the second cache, despite its larger data size, might provide slower performance than the first cache.

**5.2.5** [20] <§§5.3, 5.4> Generate a series of read requests that have a lower miss rate on a 2 KiB 2-way set associative cache than the cache listed above. Identify one possible solution that would make the cache listed have an equal or lower miss rate than the 2 KiB cache. Discuss the advantages and disadvantages of such a solution.

**5.2.6** [15] <§5.3> The formula shown in 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 32-bit address and 1024 blocks in the cache, consider a different

indexing function, specifically ( $\text{Block address}[31:27] \text{ XOR } \text{Block address}[26:22]$ ). 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.

**5.3** For a direct-mapped cache design with a 32-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
31–10	9–5	4–0

**5.3.1** [5] <§5.3> What is the cache block size (in words)?

**5.3.2** [5] <§5.3> How many entries does the cache have?

**5.3.3** [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?

Starting from power on, the following byte-addressed cache references are recorded.

Address											
0	4	16	132	232	160	1024	30	140	3100	180	2180

**5.3.4** [10] <§5.3> How many blocks are replaced?

**5.3.5** [10] <§5.3> What is the hit ratio?

**5.3.6** [20] <§5.3> List the final state of the cache, with each valid entry represented as a record of <index, tag, data>.

**5.4** Recall that we have two write policies and 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

**5.4.1** [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.

**5.4.2** [20] <§§5.3, 5.8> Describe the procedure of handling an L1 write-miss, considering the component involved and the possibility of replacing a dirty block.

**5.4.3** [20] <§§5.3, 5.8> For a multilevel exclusive cache (a block can only reside in one of the L1 and L2 caches), configuration, describe the procedure of handling an L1 write-miss, considering the component involved and the possibility of replacing a dirty block.

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 (byte)
250	100	0.30%	2%	64

**5.4.4** [5] <§§5.3, 5.8> For a write-through, write-allocate cache, what are the minimum read and write bandwidths (measured by byte per cycle) needed to achieve a CPI of 2?

**5.4.5** [5] <§§5.3, 5.8> For a write-back, write-allocate cache, assuming 30% of replaced data cache blocks are dirty, what are the minimal read and write bandwidths needed for a CPI of 2?

**5.4.6** [5] <§§5.3, 5.8> What are the minimal bandwidths needed to achieve the performance of CPI=1.5?

**5.5** 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 address stream:

0, 2, 4, 6, 8, 10, 12, 14, 16, ...

**5.5.1** [5] <§§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?

**5.5.2** [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?

**5.5.3** [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 is found in the prefetch buffer, it is considered as a hit and 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?

Cache block size ( $B$ ) can affect both miss rate and miss latency. Assuming a 1-CPI machine with an average of 1.35 references (both instruction and data) per instruction, help find the optimal block size given the following miss rates for various block sizes.

8: 4%	16: 3%	32: 2%	64: 1.5%	128: 1%
-------	--------	--------	----------	---------

**5.5.4** [10] <§5.3> What is the optimal block size for a miss latency of  $20 \times B$  cycles?

**5.5.5** [10] <§5.3> What is the optimal block size for a miss latency of  $24 + B$  cycles?

**5.5.6** [10] <§5.3> For constant miss latency, what is the optimal block size?

**5.6** 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 memory accesses are 36% of all instructions. 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

**5.6.1** [5] <§5.4> Assuming that the L1 hit time determines the cycle times for P1 and P2, what are their respective clock rates?

**5.6.2** [5] <§5.4> What is the Average Memory Access Time for P1 and P2?

**5.6.3** [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?

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

**5.6.4** [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?

**5.6.5** [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?

**5.6.6** [10] <§5.4> Which processor is faster, now that P1 has an L2 cache? If P1 is faster, what miss rate would P2 need in its L1 cache to match P1's performance? If P2 is faster, what miss rate would P1 need in its L1 cache to match P2's performance?

**5.7** This exercise examines the impact of different cache designs, specifically comparing associative caches to the direct-mapped caches from Section 5.4. For these exercises, refer to the address stream shown in Exercise 5.2.

**5.7.1** [10] <§5.4> Using the sequence of references from Exercise 5.2, show the final cache contents for a three-way set associative cache with two-word blocks and a total size of 24 words. Use LRU replacement. For each reference identify the index bits, the tag bits, the block offset bits, and if it is a hit or a miss.

**5.7.2** [10] <§5.4> Using the references from Exercise 5.2, show the final cache contents for a fully associative cache with one-word blocks and a total size of 8 words. Use LRU replacement. For each reference identify the index bits, the tag bits, and if it is a hit or a miss.

**5.7.3** [15] <§5.4> Using the references from Exercise 5.2, what is the miss rate for a fully associative cache with two-word blocks and a total size of 8 words, using LRU replacement? What is the miss rate using MRU (most recently used) replacement? Finally what is the best possible miss rate for this cache, given any replacement policy?

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 MissRate per Instruction	Second Level Cache, Direct-Mapped Speed	Global Miss Rate with Second Level Cache, Direct-Mapped	Second Level Cache, Eight-Way Set Associative Speed	Global 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%

**5.7.4** [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 is doubled? If it is cut in half?

**5.7.5** [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 reduce the global miss rate to 1.3%. Would this provide better performance? In general, what are the advantages and disadvantages of adding a third level cache?

**5.7.6** [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 global miss rate of 4%. If each additional 512 KiB of cache lowered global 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? Of the eight-way set associative cache?

**5.8** 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 devices with the following metrics.

MTTF	MTTR
3 Years	1 Day

**5.8.1** [5] <§5.5> Calculate the MTBF for each of the devices in the table.

**5.8.2** [5] <§5.5> Calculate the availability for each of the devices in the table.

**5.8.3** [5] <§5.5> What happens to availability as the MTTR approaches 0? Is this a realistic situation?

**5.8.4** [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?

**5.9** This Exercise examines the single error correcting, double error detecting (SEC/DED) Hamming code.

**5.9.1** [5] <§5.5> What is the minimum number of parity bits required to protect a 128-bit word using the SEC/DED code?

**5.9.2** [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 5.9.1. 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?

**5.9.3** 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.

**5.10** 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  $\log_2(\text{entries})$ . 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 16K.

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

**5.10.1** [10] <§5.7> What is the best page size if entries now become 128 bytes?

**5.10.2** [10] <§5.7> Based on 5.10.1, what is the best page size if pages are half full?

**5.10.3** [20] <§5.7> Based on 5.10.2, 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  $\$/\text{Disk}/\text{accesses\_per\_sec}$ , while the cost to keep a page in DRAM is  $\$/\text{DRAM\_MiB}/\text{page\_size}$ . 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

**5.10.4** [10] <§§5.1, 5.7> What are the reuse time thresholds for these three technology generations?

**5.10.5** [10] <§5.7> What are the reuse time thresholds if we keep using the same 4K page size? What’s the trend here?

**5.10.6** [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.

**5.11** 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 constitutes a stream of virtual addresses as seen on a system. Assume 4 KiB pages, a 4-entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the next largest page number.

4669, 2227, 13916, 34587, 48870, 12608, 49225

TLB

Valid	Tag	Physical Page Number
1	11	12
1	7	4
1	3	6
0	4	9

Page table

Valid	Physical Page or In Disk
1	5
0	Disk
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12

**5.11.1** [10] <§5.7> Given the address stream shown, and the initial TLB and page table states provided above, show the final state of the system. Also list for each reference if it is a hit in the TLB, a hit in the page table, or a page fault.

**5.11.2** [15] <§5.7> Repeat 5.11.1, 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?

**5.11.3** [15] <§§5.4, 5.7> Show the final contents of the TLB if it is 2-way set associative. Also show the contents of the TLB if it is direct mapped. Discuss the importance of having a TLB to high performance. How would virtual memory accesses be handled if there were no TLB?

There are several parameters that impact 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

**5.11.4** [5] <§5.7> Given the parameters shown above, calculate the total page table size for a system running 5 applications that utilize half of the memory available.

**5.11.5** [10] <§5.7> Given the parameters shown above, calculate the total page table size for a system running 5 applications that utilize half of the memory available, given a two level page table approach with 256 entries. Assume each entry of the main page table is 6 bytes. Calculate the minimum and maximum amount of memory required.

**5.11.6** [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 2 words per block? How would the designer increase the data size of the cache?

**5.12** 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

**5.12.1** [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?

**5.12.2** [10] <§5.7> Using a multilevel 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 in this case? And how many memory references are needed for address translation if missing in TLB?

**5.12.3** [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?

The following table shows the contents of a 4-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

**5.12.4** [5] <§5.7> Under what scenarios would entry 2's valid bit be set to zero?

**5.12.5** [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?

**5.12.6** [5] <§5.7> What happens when an instruction writes to VA page 200?

**5.13** In this exercise, we will examine how replacement policies impact miss rate. Assume a 2-way set associative cache with 4 blocks. To solve the problems in this exercise, you may find it helpful to draw a table like the one below, as demonstrated for the address sequence “0, 1, 2, 3, 4.”

Address of Memory Block Accessed	Hit or Miss	Evicted Block	Contents of Cache Blocks After Reference			
			Set 0	Set 0	Set 1	Set 1
0	Miss		Mem[0]			
1	Miss		Mem[0]		Mem[1]	
2	Miss		Mem[0]	Mem[2]	Mem[1]	
3	Miss		Mem[0]	Mem[2]	Mem[1]	Mem[3]
4	Miss	0	Mem[4]	Mem[2]	Mem[1]	Mem[3]
...						

Consider the following address sequence: 0, 2, 4, 8, 10, 12, 14, 16, 0

**5.13.1** [5] <§§5.4, 5.8> Assuming an LRU replacement policy, how many hits does this address sequence exhibit?

**5.13.2** [5] <§§5.4, 5.8> Assuming an MRU (most recently used) replacement policy, how many hits does this address sequence exhibit?

**5.13.3** [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?

**5.13.4** [10] <§§5.4, 5.8> Which address should be evicted at each replacement to maximize the number of hits? How many hits does this address sequence exhibit if you follow this “optimal” policy?

**5.13.5** [10] <§§5.4, 5.8> Describe why it is difficult to implement a cache replacement policy that is optimal for all address sequences.

**5.13.6** [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 impact could this have on miss rate?

**5.14** 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 ( $VA \Rightarrow PA$  and  $PA \Rightarrow MA$ ) 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;

**5.14.1** [10] <§§5.6, 5.7> What would happen for the given operation sequence for shadow page table and nested page table, respectively?

**5.14.2** [10] <§§5.6, 5.7> Assuming an x86-based 4-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?

**5.14.3** [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

**5.14.4** [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)?

**5.14.5** [10] <§5.6> What techniques can be used to reduce page table shadowing induced overhead?

**5.14.6** [10] <§5.6> What techniques can be used to reduce NPT induced overhead?

**5.15** 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	Performance Impact to Trap to the Guest O/S	Performance Impact 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

**5.15.1** [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 performance impact doubles? If it is cut in half? If a virtual machine software company wishes to obtain a 10% performance degradation, what is the longest possible penalty to trap to the VMM?

**5.15.2** [10] <§5.6> I/O accesses often have a large impact 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? Explain why I/O bound applications have a smaller impact from virtualization.

**5.15.3** [30] <§§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.

**5.15.4** [20] <§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?

**5.16** 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 5.40 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 465 (Figure 5.40 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.

**5.16.1** [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?

**5.16.2** [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?

**5.16.3** [30] <§§5.8, 5.9> Design a finite state machine to enable the use of a write buffer.

**5.17** Cache coherence concerns the views of multiple processors on a given cache block. The following data shows two processors and their read/write operations on two different words of a cache block X (initially  $X[0] = X[1] = 0$ ). Assume the size of integers is 32 bits.

P1	P2
$X[0] \text{ ++}; X[1] = 3;$	$X[0] = 5; X[1] \text{ +=} 2;$

**5.17.1** [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.

**5.17.2** [15] <§5.10> For a snooping protocol, list a valid operation sequence on each processor/cache to finish the above read/write operations.

**5.17.3** [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 shows two processors and their read/write operations on different cache blocks (A and B initially 0).

P1	P2
$A = 1; B = 2; A \text{ +=} 2; B \text{ ++};$	$C = B; D = A;$

**5.17.4** [15] <§5.10> List the possible values of C and D for an implementation that ensures both consistency assumptions on page 470.

**5.17.5** [15] <§5.10> List at least one more possible pair of values for C and D if such assumptions are not maintained.

**5.17.6** [15] <§§5.3, 5.10> For various combinations of write policies and write allocation policies, which combinations make the protocol implementation simpler?

**5.18** 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 L1 cache misses once every 32 instructions.

	Private	Shared
Benchmark A misses-per-instruction	0.30%	0.12%
Benchmark B misses-per-instruction	0.06%	0.03%

Assume the following hit latencies:

Private Cache	Shared Cache	Memory
5	20	180

**5.18.1** [15] <§5.13> Which cache design is better for each of these benchmarks? Use data to support your conclusion.

**5.18.2** [15] <§5.13> Shared cache latency increases with the CMP size. Choose the best design if the shared cache latency doubles. Off-chip bandwidth becomes the bottleneck as the number of CMP cores increases. Choose the best design if off-chip memory latency doubles.

**5.18.3** [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.

**5.18.4** [15] <§5.13> Assume both benchmarks have a base CPI of 1 (ideal L2 cache). If having non-blocking cache improves the average number of concurrent L2 misses from 1 to 2, how much performance improvement does this provide over a shared L2 cache? How much improvement can be achieved over private L2?

**5.18.5** [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?

**5.18.6** [15] <§5.13> Consider the entire memory hierarchy. What kinds of optimizations can improve the number of concurrent misses?

**5.19** 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);
```

**5.19.1** [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?

**5.19.2** [10] <§5.15> How can you reorganize the data structure to improve cache utilization and access locality? Show your structure definition code.

**5.19.3** [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.

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.

<b>a.</b>	Mesa / gcc
<b>b.</b>	mcf / swim

**5.19.4** [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?

**5.19.5** [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.

**5.19.6** [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.

**Answers to  
Check Yourself**

§5.1, page 377: 1 and 4. (3 is false because the cost of the memory hierarchy varies per computer, but in 2013 the highest cost is usually the DRAM.)

§5.3, page 398: 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 417: 1.

§5.7, page 454: 1-a, 2-c, 3-b, 4-d.

§5.8, page 461: 2. (Both large block sizes and prefetching may reduce compulsory misses, so 1 is false.)

This page intentionally left blank

# 6

*"I swing big, with  
everything I've got.  
I hit big or I miss big.  
I like to live as big as  
I can."*

**Babe Ruth**  
*American baseball player*

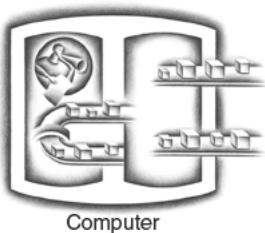
## Parallel Processors from Client to Cloud

- 6.1 Introduction** 502
- 6.2 The Difficulty of Creating Parallel Processing Programs** 504
- 6.3 SISD, MIMD, SIMD, SPMD, and Vector** 509
- 6.4 Hardware Multithreading** 516
- 6.5 Multicore and Other Shared Memory Multiprocessors** 519
- 6.6 Introduction to Graphics Processing Units** 524

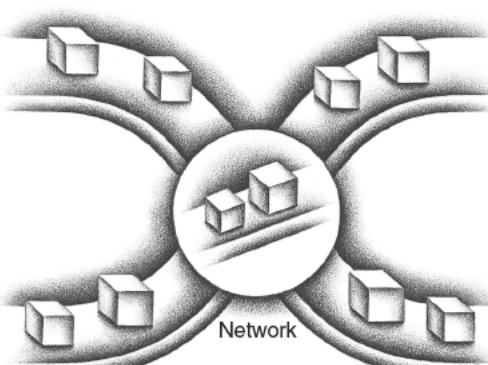
- 6.7 Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors** 531
- 6.8 Introduction to Multiprocessor Network Topologies** 536
-  **6.9 Communicating to the Outside World: Cluster Networking** 539
- 6.10 Multiprocessor Benchmarks and Performance Models** 540
- 6.11 Real Stuff: Benchmarking Intel Core i7 versus NVIDIA Tesla GPU** 550
- 6.12 Going Faster: Multiple Processors and Matrix Multiply** 555
- 6.13 Fallacies and Pitfalls** 558
- 6.14 Concluding Remarks** 560
-  **6.15 Historical Perspective and Further Reading** 563
- 6.16 Exercises** 563

---

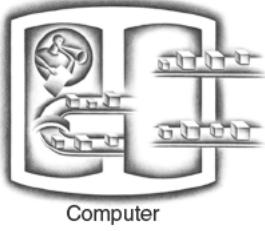
## Multiprocessor or Cluster Organization



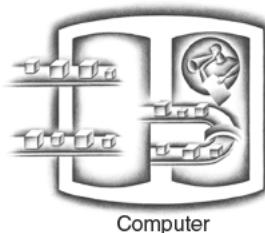
Computer



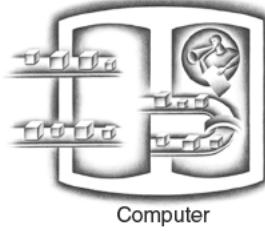
Network



Computer



Computer



Computer

## 6.1

# Introduction

*Over the Mountains Of  
the Moon, Down the  
Valley of the Shadow,  
Ride, boldly ride the  
shade replied— If you  
seek for El Dorado!*

Edgar Allan Poe,  
“El Dorado,”  
stanza 4, 1849

### **multiprocessor**

A computer system with at least two processors. This computer is in contrast to a uniprocessor, which has one, and is increasingly hard to find today.



**task-level parallelism or process-level parallelism** Utilizing multiple processors by running independent programs simultaneously.

**parallel processing program** A single program that runs on multiple processors simultaneously.

**cluster** A set of computers connected over a local area network that function as a single large multiprocessor.

Computer architects have long sought the “The City of Gold” (El Dorado) of computer design: to create powerful computers simply by connecting many existing smaller ones. This golden vision is the fountainhead of **multiprocessors**. Ideally, customers order as many processors as they can afford and receive a commensurate amount of performance. Thus, multiprocessor software must be designed to work with a variable number of processors. As mentioned in Chapter 1, energy has become the overriding issue for both microprocessors and datacenters. Replacing large inefficient processors with many smaller, efficient processors can deliver better performance per joule both in the large and in the small, if software can efficiently use them. Thus, improved energy efficiency joins scalable performance in the case for multiprocessors.

Since multiprocessor software should scale, some designs support operation in the presence of broken hardware; that is, if a single processor fails in a multiprocessor with  $n$  processors, these system would continue to provide service with  $n - 1$  processors. Hence, multiprocessors can also improve availability (see Chapter 5).

High performance can mean high throughput for independent tasks, called **task-level parallelism** or **process-level parallelism**. These tasks are independent single-threaded applications, and they are an important and popular use of multiple processors. This approach is in contrast to running a single job on multiple processors. We use the term **parallel processing program** to refer to a single program that runs on multiple processors simultaneously.

There have long been scientific problems that have needed much faster computers, and this class of problems has been used to justify many novel parallel computers over the decades. Some of these problems can be handled simply today, using a **cluster** composed of microprocessors housed in many independent servers (see Section 6.7). In addition, clusters can serve equally demanding applications outside the sciences, such as search engines, Web servers, email servers, and databases.

As described in Chapter 1, multiprocessors have been shoved into the spotlight because the energy problem means that future increases in performance will primarily come from explicit hardware parallelism rather than much higher clock rates or vastly improved CPI. As we said in Chapter 1, they are called

**multicore microprocessors** instead of multiprocessor microprocessors, presumably to avoid redundancy in naming. Hence, processors are often called *cores* in a multicore chip. The number of cores is expected to increase with **Moore's Law**. These multicores are almost always **Shared Memory Processors (SMPs)**, as they usually share a single physical address space. We'll see SMPs more in Section 6.5.

The state of technology today means that programmers who care about performance must become parallel programmers, for sequential code now means slow code.

The tall challenge facing the industry is to create hardware and software that will make it easy to write correct parallel processing programs that will execute efficiently in performance and energy as the number of cores per chip scales.

This abrupt shift in microprocessor design caught many off guard, so there is a great deal of confusion about the terminology and what it means. Figure 6.1 tries to clarify the terms serial, parallel, sequential, and concurrent. The columns of this figure represent the software, which is either inherently sequential or concurrent. The rows of the figure represent the hardware, which is either serial or parallel. For example, the programmers of compilers think of them as sequential programs: the steps include parsing, code generation, optimization, and so on. In contrast, the programmers of operating systems normally think of them as concurrent programs: cooperating processes handling I/O events due to independent jobs running on a computer.

The point of these two axes of Figure 6.1 is that concurrent software can run on serial hardware, such as operating systems for the Intel Pentium 4 uniprocessor, or on parallel hardware, such as an OS on the more recent Intel Core i7. The same is true for sequential software. For example, the MATLAB programmer writes a matrix multiply thinking about it sequentially, but it could run serially on the Pentium 4 or in parallel on the Intel Core i7.

You might guess that the only challenge of the parallel revolution is figuring out how to make naturally sequential software have high performance on parallel hardware, but it is also to make concurrent programs have high performance on multiprocessors as the number of processors increases. With this distinction made, in the rest of this chapter we will use *parallel processing program* or *parallel software* to mean either sequential or concurrent software running on parallel hardware. The next section of this chapter describes why it is hard to create efficient parallel processing programs.

### multicore microprocessor

A microprocessor containing multiple processors ("cores") in a single integrated circuit. Virtually all microprocessors today in desktops and servers are multicore.

### shared memory multiprocessor (SMP)

A parallel processor with a single physical address space.



MOORE'S LAW

		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

**FIGURE 6.1 Hardware/software categorization and examples of application perspective on concurrency versus hardware perspective on parallelism.**

Before proceeding further down the path to parallelism, don't forget our initial incursions from the earlier chapters:

- Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization
- Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Subword Parallelism
- Chapter 4, Section 4.10: Parallelism via Instructions
- Chapter 5, Section 5.10: Parallelism and Memory Hierarchy: Cache Coherence

**Check  
Yourself**

True or false: To benefit from a multiprocessor, an application must be concurrent.

## 6.2

### The Difficulty of Creating Parallel Processing Programs

The difficulty with parallelism is not the hardware; it is that too few important application programs have been rewritten to complete tasks sooner on multiprocessors. It is difficult to write software that uses multiple processors to complete one task faster, and the problem gets worse as the number of processors increases.

Why has this been so? Why have parallel processing programs been so much harder to develop than sequential programs?

The first reason is that you *must* get better performance or better energy efficiency from a parallel processing program on a multiprocessor; otherwise, you would just use a sequential program on a uniprocessor, as sequential programming is simpler. In fact, uniprocessor design techniques such as superscalar and out-of-order execution take advantage of instruction-level parallelism (see Chapter 4), normally without the involvement of the programmer. Such innovations reduced the demand for rewriting programs for multiprocessors, since programmers could do nothing and yet their sequential programs would run faster on new computers.

Why is it difficult to write parallel processing programs that are fast, especially as the number of processors increases? In Chapter 1, we used the analogy of eight reporters trying to write a single story in hopes of doing the work eight times faster. To succeed, the task must be broken into eight equal-sized pieces, because otherwise some reporters would be idle while waiting for the ones with larger pieces to finish. Another speed-up obstacle could be that the reporters would spend too much time communicating with each other instead of writing their pieces of the story. For both this analogy and parallel programming, the challenges include scheduling, partitioning the work into parallel pieces, balancing the load evenly between the workers, time to synchronize, and

overhead for communication between the parties. The challenge is stiffer with the more reporters for a newspaper story and with the more processors for parallel programming.

Our discussion in Chapter 1 reveals another obstacle, namely Amdahl's Law. It reminds us that even small parts of a program must be parallelized if the program is to make good use of many cores.

### Speed-up Challenge

Suppose you want to achieve a speed-up of 90 times faster with 100 processors. What percentage of the original computation can be sequential?

### EXAMPLE

Amdahl's Law (Chapter 1) says

Execution time after improvement =

$$\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

### ANSWER

We can reformulate Amdahl's Law in terms of speed-up versus the original execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \frac{\text{Execution time affected}}{\text{Amount of improvement}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of time, and the execution time affected by improvement is considered the fraction of the original execution time:

$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{\text{Amount of improvement}}}$$

Substituting 90 for speed-up and 100 for amount of improvement into the formula above:

$$90 = \frac{1}{(1 - \text{Fraction time affected}) + \frac{\text{Fraction time affected}}{100}}$$

Then simplifying the formula and solving for fraction time affected:

$$90 \times (1 - 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - (90 \times 0.99 \times \text{Fraction time affected}) = 1$$

$$90 - 1 = 90 \times 0.99 \times \text{Fraction time affected}$$

$$\text{Fraction time affected} = 89/89.1 = 0.999$$

Thus, to achieve a speed-up of 90 from 100 processors, the sequential percentage can only be 0.1%.

Yet, there are applications with plenty of parallelism, as we shall see next.

## EXAMPLE

### Speed-up Challenge: Bigger Problem

Suppose you want to perform two sums: one is a sum of 10 scalar variables, and one is a matrix sum of a pair of two-dimensional arrays, with dimensions 10 by 10. For now let's assume only the matrix sum is parallelizable; we'll see soon how to parallelize scalar sums. What speed-up do you get with 10 versus 40 processors? Next, calculate the speed-ups assuming the matrices grow to 20 by 20.

## ANSWER

If we assume performance is a function of the time for an addition,  $t$ , then there are 10 additions that do not benefit from parallel processors and 100 additions that do. If the time for a single processor is  $110t$ , the execution time for 10 processors is

Execution time after improvement =

$$\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$\text{Execution time after improvement} = \frac{100t}{10} + 10t = 20t$$

so the speed-up with 10 processors is  $110t/20t = 5.5$ . The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{100t}{40} + 10t = 12.5t$$

so the speed-up with 40 processors is  $110t/12.5t = 8.8$ . Thus, for this problem size, we get about 55% of the potential speed-up with 10 processors, but only 22% with 40.

Look what happens when we increase the matrix. The sequential program now takes  $10t + 400t = 410t$ . The execution time for 10 processors is

$$\text{Execution time after improvement} = \frac{400t}{10} + 10t = 50t$$

so the speed-up with 10 processors is  $410t/50t = 8.2$ . The execution time for 40 processors is

$$\text{Execution time after improvement} = \frac{400t}{40} + 10t = 20t$$

so the speed-up with 40 processors is  $410t/20t = 20.5$ . Thus, for this larger problem size, we get 82% of the potential speed-up with 10 processors and 51% with 40.

These examples show that getting good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the size of the problem. This insight allows us to introduce two terms that describe ways to scale up.

**Strong scaling** means measuring speed-up while keeping the problem size fixed.

**Weak scaling** means that the problem size grows proportionally to the increase in the number of processors. Let's assume that the size of the problem, M, is the working set in main memory, and we have P processors. Then the memory per processor for strong scaling is approximately  $M/P$ , and for weak scaling, it is approximately M.

Note that the **memory hierarchy** can interfere with the conventional wisdom about weak scaling being easier than strong scaling. For example, if the weakly scaled dataset no longer fits in the last level cache of a multicore microprocessor, the resulting performance could be much worse than by using strong scaling.

Depending on the application, you can argue for either scaling approach. For example, the TPC-C debit-credit database benchmark requires that you scale up the number of customer accounts in proportion to the higher transactions per minute. The argument is that it's nonsensical to think that a given customer base is suddenly going to start using ATMs 100 times a day just because the bank gets a faster computer. Instead, if you're going to demonstrate a system that can perform 100 times the numbers of transactions per minute, you should run the experiment with 100 times as many customers. Bigger problems often need more data, which is an argument for weak scaling.

This final example shows the importance of load balancing.

**strong scaling** Speed-up achieved on a multiprocessor without increasing the size of the problem.

**weak scaling** Speed-up achieved on a multiprocessor while increasing the size of the problem proportionally to the increase in the number of processors.



### Speed-up Challenge: Balancing Load

To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced. That is, each of the 40

**EXAMPLE**

**ANSWER**

processors had 2.5% of the work to do. Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at twice the load (5%) and five times the load (12.5%) for that hardest working processor. How well utilized are the rest of the processors?

If one processor has 5% of the parallel load, then it must do  $5\% \times 400$  or 20 additions, and the other 39 will share the remaining 380. Since they are operating simultaneously, we can just calculate the execution time as a maximum

$$\text{Execution time after improvement} = \text{Max}\left(\frac{380t}{39}, \frac{20t}{1}\right) + 10t = 30t$$

The speed-up drops from 20.5 to  $410t/30t = 14$ . The remaining 39 processors are utilized less than half the time: while waiting 20t for hardest working processor to finish, they only compute for  $380t/39 = 9.7t$ .

If one processor has 12.5% of the load, it must perform 50 additions. The formula is:

$$\text{Execution time after improvement} = \text{Max}\left(\frac{350t}{39}, \frac{50t}{1}\right) + 10t = 60t$$

The speed-up drops even further to  $410t/60t = 7$ . The rest of the processors are utilized less than 20% of the time ( $9t/50t$ ). This example demonstrates the importance of balancing load, for just a single processor with twice the load of the others cuts speed-up by a third, and five times the load on just one processor reduces speed-up by almost a factor of three.

Now that we better understand the goals and challenges of parallel processing, we give an overview of the rest of the chapter. The next Section (6.3) describes a much older classification scheme than in [Figure 6.1](#). In addition, it describes two styles of instruction set architectures that support running of sequential applications on parallel hardware, namely *SIMD* and *vector*. Section 6.4 then describes *multithreading*, a term often confused with multiprocessing, in part because it relies upon similar concurrency in programs. Section 6.5 describes the first the two alternatives of a fundamental parallel hardware characteristic, which is whether or not all the processors in the systems rely upon a single physical address space. As mentioned above, the two popular versions of these alternatives are called *shared memory multiprocessors* (SMPs) and *clusters*, and this section covers the former. Section 6.6 describes a relatively new style of computer from the graphics hardware community, called a *graphics-processing unit* (GPU) that also assumes a single physical address. ([Appendix C](#) describes GPUs in even more detail.) Section 6.7 describes clusters, a popular example of a computer with multiple physical address spaces. Section 6.8 shows typical topologies used to connect many processors together, either server nodes in a cluster or cores in a microprocessor. [Section 6.9](#) describes the hardware and software for communicating between

nodes in a cluster using Ethernet. It shows how to optimize its performance using custom software and hardware. We next discuss the difficulty of finding parallel benchmarks in Section 6.10. This section also includes a simple, yet insightful performance model that helps in the design of applications as well as architectures. We use this model as well as parallel benchmarks in Section 6.11 to compare a multicore computer to a GPU. Section 6.12 divulges the final and largest step in our journey of accelerating matrix multiply. For matrices that don't fit in the cache, parallel processing uses 16 cores to improve performance by a factor of 14. We close with fallacies and pitfalls and our conclusions for parallelism.

In the next section, we introduce acronyms that you probably have already seen to identify different types of parallel computers.

True or false: Strong scaling is not bound by Amdahl's Law.

### Check Yourself

## 6.3

## SISD, MIMD, SIMD, SPMD, and Vector

One categorization of parallel hardware proposed in the 1960s is still used today. It was based on the number of instruction streams and the number of data streams. Figure 6.2 shows the categories. Thus, a conventional uniprocessor has a single instruction stream and single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams. These two categories are abbreviated **SISD** and **MIMD**, respectively.

While it is possible to write separate programs that run on different processors on a MIMD computer and yet work together for a grander, coordinated goal, programmers normally write a single program that runs on all processors of an **MIMD** computer, relying on conditional statements when different processors should execute different sections of code. This style is called **Single Program Multiple Data (SPMD)**, but it is just the normal way to program a MIMD computer.

The closest we can come to multiple instruction streams and single data stream (**MISD**) processor might be a “stream processor” that would perform a series of computations on a single data stream in a pipelined fashion: parse the input from the network, decrypt the data, decompress it, search for match, and so on. The inverse of MISD is much more popular. **SIMD** computers operate on vectors of

**SISD** or Single Instruction stream, Single Data stream. A uniprocessor.

**MIMD** or Multiple Instruction streams, Multiple Data streams. A multiprocessor.

**SPMD** Single Program, Multiple Data streams. The conventional MIMD programming model, where a single program runs across all processors.

**SIMD** or Single Instruction stream, Multiple Data streams. The same instruction is applied to many data streams, as in a vector processor.

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Core i7

**FIGURE 6.2 Hardware categorization and examples based on number of instruction streams and data streams: SISD, SIMD, MISD, and MIMD.**

data. For example, a single SIMD instruction might add 64 numbers by sending 64 data streams to 64 ALUs to form 64 sums within a single clock cycle. The subword parallel instructions that we saw in Sections 3.6 and 3.7 are another example of SIMD; indeed, the middle letter of Intel's SSE acronym stands for SIMD.

The virtues of SIMD are that all the parallel execution units are synchronized and they all respond to a single instruction that emanates from a single *program counter* (PC). From a programmer's perspective, this is close to the already familiar SISD. Although every unit will be executing the same instruction, each execution unit has its own address registers, and so each unit can have different data addresses. Thus, in terms of Figure 6.1, a sequential application might be compiled to run on serial hardware organized as a SISD or in parallel hardware that was organized as a SIMD.

The original motivation behind SIMD was to amortize the cost of the control unit over dozens of execution units. Another advantage is the reduced instruction bandwidth and space—SIMD needs only one copy of the code that is being simultaneously executed, while message-passing MIMDs may need a copy in every processor, and shared memory MIMD will need multiple instruction caches.

SIMD works best when dealing with arrays in `for` loops. Hence, for parallelism to work in SIMD, there must be a great deal of identically structured data, which is called **data-level parallelism**. SIMD is at its weakest in `case` or `switch` statements, where each execution unit must perform a different operation on its data, depending on what data it has. Execution units with the wrong data must be disabled so that units with proper data may continue. If there are  $n$  cases, in these situations SIMD processors essentially run at  $1/n$ th of peak performance.

The so-called array processors that inspired the SIMD category have faded into history (see [Section 6.15](#) online), but two current interpretations of SIMD remain active today.

## SIMD in x86: Multimedia Extensions

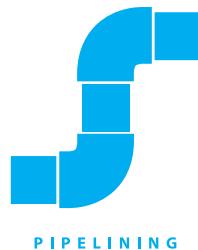
As described in Chapter 3, subword parallelism for narrow integer data was the original inspiration of the Multimedia Extension (MMX) instructions of the x86 in 1996. As **Moore's Law** continued, more instructions were added, leading first to *Streaming SIMD Extensions* (SSE) and now *Advanced Vector Extensions* (AVX). AVX supports the simultaneous execution of four 64-bit floating-point numbers. The width of the operation and the registers is encoded in the opcode of these multimedia instructions. As the data width of the registers and operations grew, the number of opcodes for multimedia instructions exploded, and now there are hundreds of SSE and AVX instructions (see Chapter 3).



## Vector

An older and, as we shall see, more elegant interpretation of SIMD is called a *vector architecture*, which has been closely identified with computers designed by Seymour Cray starting in the 1970s. It is also a great match to problems with lots of data-level parallelism. Rather than having 64 ALUs perform 64 additions simultaneously, like the old array processors, the vector architectures pipelined the ALU to get good performance at lower cost. The basic philosophy of vector architecture is to collect

data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers using **pipelined execution units**, and then write the results back to memory. A key feature of vector architectures is then a set of vector registers. Thus, a vector architecture might have 32 vector registers, each with 64 64-bit elements.



PIPELINING

### Comparing Vector to Conventional Code

Suppose we extend the MIPS instruction set architecture with vector instructions and vector registers. Vector operations use the same names as MIPS operations, but with the letter “V” appended. For example, `addv.d` adds two double-precision vectors. The vector instructions take as their input either a pair of vector registers (`addv.d`) or a vector register and a scalar register (`addvs.d`). In the latter case, the value in the scalar register is used as the input for all operations—the operation `addvs.d` will add the contents of a scalar register to each element in a vector register. The names `lv` and `sv` denote vector load and vector store, and they load or store an entire vector of double-precision data. One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory. Given this short description, show the conventional MIPS code versus the vector MIPS code for

$$Y = a \times X + Y$$

where  $X$  and  $Y$  are vectors of 64 double precision floating-point numbers, initially resident in memory, and  $a$  is a scalar double precision variable. (This example is the so-called DAXPY loop that forms the inner loop of the Linpack benchmark; DAXPY stands for double precision a  $\times$   $X$  plus  $Y$ .) Assume that the starting addresses of  $X$  and  $Y$  are in `$s0` and `$s1`, respectively.

Here is the conventional MIPS code for DAXPY:

```

l.d      $f0,a($sp)      :load scalar a
addiu   $t0,$s0,#512     :upper bound of what to load
loop: l.d      $f2,0($s0)    :load x(i)
      mul.d   $f2,$f2,$f0    :a x x(i)
      l.d      $f4,0($s1)    :load y(i)
      add.d   $f4,$f4,$f2    :a x x(i) + y(i)
      s.d      $f4,0($s1)    :store into y(i)
      addiu   $s0,$s0,#8      :increment index to x
      addiu   $s1,$s1,#8      :increment index to y
      subu   $t1,$t0,$s0      :compute bound
      bne    $t1,$zero,loop   :check if done

```

### EXAMPLE

### ANSWER

Here is the vector MIPS code for DAXPY:

l.d	\$f0,a(\$sp)	:load scalar a
lv	\$v1,0(\$s0)	:load vector x
mulvs.d	\$v2,\$v1,\$f0	:vector-scalar multiply
lv	\$v3,0(\$s1)	:load vector y
addv.d	\$v4,\$v2,\$v3	:add y to product
sv	\$v4,0(\$s1)	:store the result

There are some interesting comparisons between the two code segments in this example. The most dramatic is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 6 instructions versus almost 600 for the traditional MIPS architecture. This reduction occurs both because the vector operations work on 64 elements at a time and because the overhead instructions that constitute nearly half the loop on MIPS are not present in the vector code. As you might expect, this reduction in instructions fetched and executed saves energy.

Another important difference is the frequency of **pipeline** hazards (Chapter 4). In the straightforward MIPS code, every `add.d` must wait for a `mul.d`, every `s.d` must wait for the `add.d` and every `add.d` and `mul.d` must wait on `l.d`. On the vector processor, each vector instruction will only stall for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus, pipeline stalls are required only once per vector *operation*, rather than once per vector *element*. In this example, the pipeline stall frequency on MIPS will be about 64 times higher than it is on the vector version of MIPS. The pipeline stalls can be reduced on MIPS by using loop unrolling (see Chapter 4). However, the large difference in instruction bandwidth cannot be reduced.

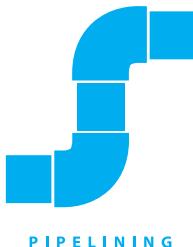
Since the vector elements are independent, they can be operated on in parallel, much like subword parallelism for AVX instructions. All modern vector computers have vector functional units with multiple parallel pipelines (called *vector lanes*; see Figures 6.2 and 6.3) that can produce two or more results per clock cycle.

**Elaboration:** The loop in the example above exactly matched the vector length. When loops are shorter, vector architectures use a register that reduces the length of vector operations. When loops are larger, we add bookkeeping code to iterate full-length vector operations and to handle the leftovers. This latter process is called *strip mining*.

## Vector versus Scalar

Vector instructions have several important properties compared to conventional instruction set architectures, which are called *scalar architectures* in this context:

- A single vector instruction specifies a great deal of work—it is equivalent to executing an entire loop. The instruction fetch and decode bandwidth needed is dramatically reduced.
- By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector, so hardware does not have to check for data hazards within a vector instruction.
- Vector architectures and compilers have a reputation of making it much easier than when using MIMD multiprocessors to write efficient applications when they contain data-level parallelism.



PIPELINING

- Hardware need only check for data hazards between two vector instructions once per vector operand, not once for every element within the vectors. Reduced checking can save energy as well as time.
- Vector instructions that access memory have a known access pattern. If the vector's elements are all adjacent, then fetching the vector from a set of heavily interleaved memory banks works very well. Thus, the cost of the latency to main memory is seen only once for the entire vector, rather than once for each word of the vector.
- Because an entire loop is replaced by a vector instruction whose behavior is predetermined, control hazards that would normally arise from the loop branch are nonexistent.
- The savings in instruction bandwidth and hazard checking plus the efficient use of memory bandwidth give vector architectures advantages in power and energy versus scalar architectures.

For these reasons, vector operations can be made faster than a sequence of scalar operations on the same number of data items, and designers are motivated to include vector units if the application domain can often use them.

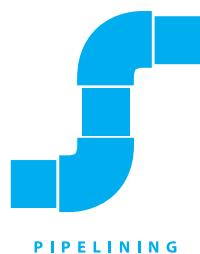
## Vector versus Multimedia Extensions

Like multimedia extensions found in the x86 AVX instructions, a vector instruction specifies multiple operations. However, multimedia extensions typically specify a few operations while vector specifies dozens of operations. Unlike multimedia extensions, the number of elements in a vector operation is not in the opcode but in a separate register. This distinction means different versions of the vector architecture can be implemented with a different number of elements just by changing the contents of that register and hence retain binary compatibility. In contrast, a new large set of opcodes is added each time the "vector" length changes in the multimedia extension architecture of the x86: MMX, SSE, SSE2, AVX, AVX2, ... .

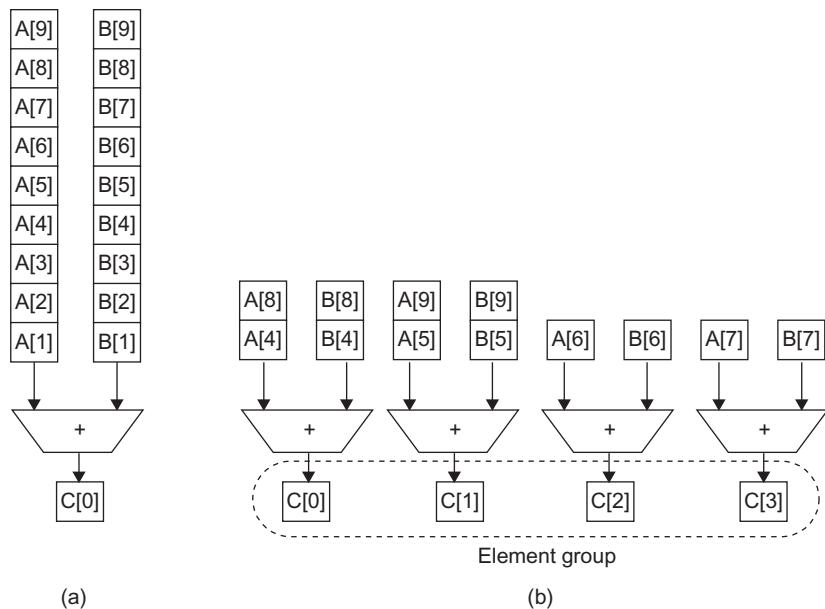
Also unlike multimedia extensions, the data transfers need not be contiguous. Vectors support both strided accesses, where the hardware loads every  $n$ th data element in memory, and indexed accesses, where hardware finds the addresses of the items to be loaded in a vector register. Indexed accesses are also called *gather-scatter*, in that indexed loads gather elements from main memory into contiguous vector elements and indexed stores scatter vector elements across main memory.

Like multimedia extensions, vector architectures easily capture the flexibility in data widths, so it is easy to make a vector operation work on 32 64-bit data elements or 64 32-bit data elements or 128 16-bit data elements or 256 8-bit data elements. The parallel semantics of a vector instruction allows an implementation to execute these operations using a deeply **pipelined** functional unit, an array of parallel functional units, or a combination of parallel and pipelined functional units. [Figure 6.3](#) illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.

Vector arithmetic instructions usually only allow element N of one vector register to take part in operations with element N from other vector registers. This



PIPELINING



**FIGURE 6.3 Using multiple functional units to improve the performance of a single vector add instruction,  $C = A + B$ .** The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines or lanes and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four lanes.

**vector lane** One or more vector functional units and a portion of the vector register file. Inspired by lanes on highways that increase traffic speed, multiple lanes execute vector operations simultaneously.

dramatically simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel **vector lanes**. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Figure 6.4 shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks per vector instruction by roughly a factor of four. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors. Otherwise, they will execute so quickly that you'll run out of instructions, requiring instruction level **parallel** techniques like those in Chapter 4 to supply enough vector instructions.

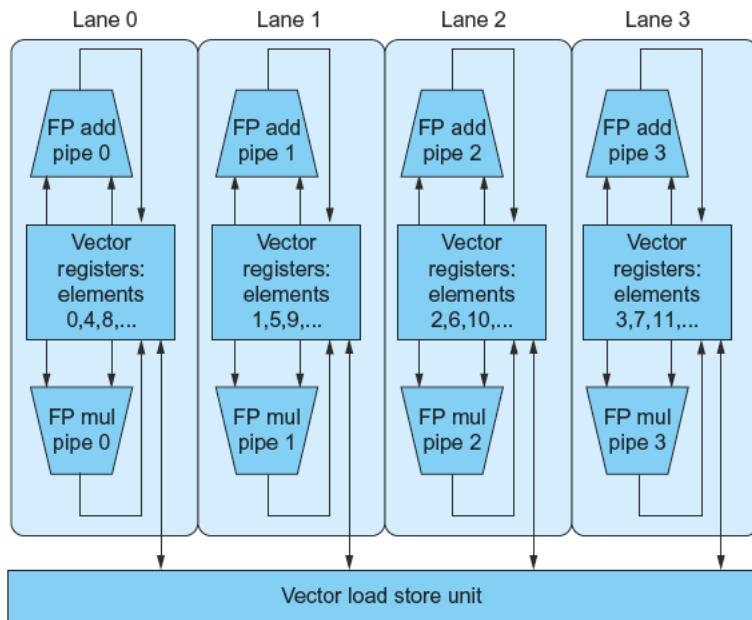
Generally, vector architectures are a very efficient way to execute data parallel processing programs; they are better matches to compiler technology than multimedia extensions; and they are easier to evolve over time than the multimedia extensions to the x86 architecture.

Given these classic categories, we next see how to exploit parallel streams of instructions to improve the performance of a *single* processor, which we will reuse with multiple processors.



## Check Yourself

True or false: As exemplified in the x86, multimedia extensions can be thought of as a vector architecture with short vectors that supports only contiguous vector data transfers.



**FIGURE 6.4 Structure of a vector unit containing four lanes.** The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which acts in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough read and write ports (see Chapter 4) for functional units local to its lane.

**Elaboration:** Given the advantages of vector, why aren't they more popular outside high-performance computing? There were concerns about the larger state for vector registers increasing context switch time and the difficulty of handling page faults in vector loads and stores, and SIMD instructions achieved some of the benefits of vector instructions. In addition, as long as advances in instruction level parallelism could deliver on the performance promise of Moore's Law, there was little reason to take the chance of changing architecture styles.

**Elaboration:** Another advantage of vector and multimedia extensions is that it is relatively easy to extend a scalar instruction set architecture with these instructions to improve performance of data parallel operations.

**Elaboration:** The Haswell-generation x86 processors from Intel support AVX2, which has a gather operation but not a scatter operation.

**hardware multithreading**

Increasing utilization of a processor by switching to another thread when one thread is stalled.

**thread** A thread includes the program counter, the register state, and the stack. It is a lightweight process; whereas threads commonly share a single address space, processes don't.

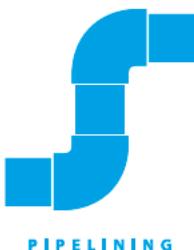
**process** A process includes one or more threads, the address space, and the operating system state. Hence, a process switch usually invokes the operating system, but not a thread switch.

**fine-grained multithreading**

A version of hardware multithreading that implies switching between threads after every instruction.

**coarse-grained multithreading**

A version of hardware multithreading that implies switching between threads only after significant events, such as a last-level cache miss.

**6.4****Hardware Multithreading**

A related concept to MIMD, especially from the programmer's perspective, is **hardware multithreading**. While MIMD relies on multiple **processes** or **threads** to try to keep multiple processors busy, hardware multithreading allows multiple threads to share the functional units of a *single* processor in an overlapping fashion to try to utilize the hardware resources efficiently. To permit this sharing, the processor must duplicate the independent state of each thread. For example, each thread would have a separate copy of the register file and the program counter. The memory itself can be shared through the virtual memory mechanisms, which already support multi-programming. In addition, the hardware must support the ability to change to a different thread relatively quickly. In particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.

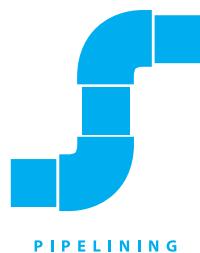
There are two main approaches to hardware multithreading. **Fine-grained multithreading** switches between threads on each instruction, resulting in interleaved execution of multiple threads. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle. To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle. One advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. The primary disadvantage of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

**Coarse-grained multithreading** was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as last-level cache misses. This change relieves the need to have thread switching be extremely fast and is much less likely to slow down the execution of an individual thread, since instructions from other threads will only be issued when a thread encounters a costly stall. Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the **pipeline** start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen. The new thread that begins executing after the stall must fill the pipeline before instructions will be able to complete. Due to this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high-cost stalls, where pipeline refill is negligible compared to the stall time.

**Simultaneous multithreading (SMT)** is a variation on hardware multithreading that uses the resources of a multiple-issue, dynamically scheduled **pipelined** processor to exploit thread-level parallelism at the same time it exploits instruction-level parallelism (see Chapter 4). The key insight that motivates SMT is that multiple-issue processors often have more functional unit parallelism available than most single threads can effectively use. Furthermore, with register renaming and dynamic scheduling (see Chapter 4), multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

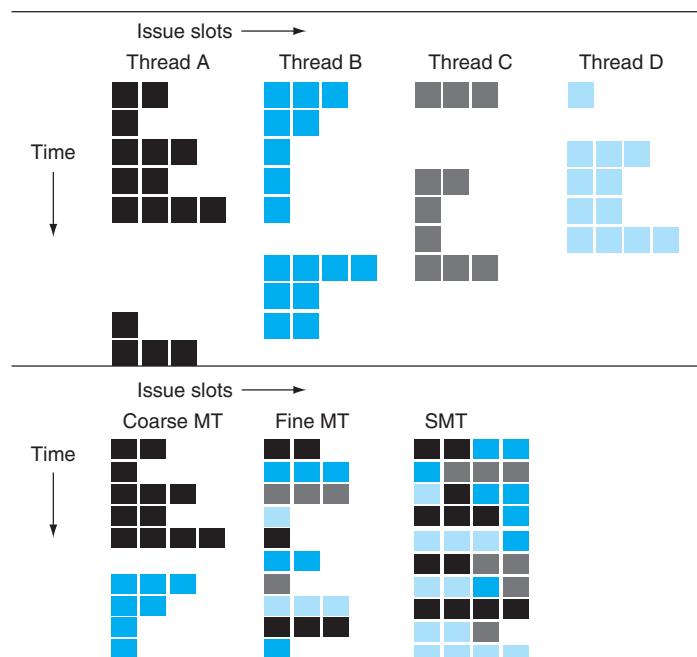
Since SMT relies on the existing dynamic mechanisms, it does not switch resources every cycle. Instead, SMT is *always* executing instructions from multiple threads, leaving it up to the hardware to associate instruction slots and renamed registers with their proper threads.

Figure 6.5 conceptually illustrates the differences in a processor's ability to exploit superscalar resources for the following processor configurations. The top portion shows



PIPELINING

**simultaneous multithreading (SMT)** A version of multithreading that lowers the cost of multithreading by utilizing the resources needed for multiple issue, dynamically scheduled microarchitecture.



**FIGURE 6.5 How four threads use the issue slots of a superscalar processor in different approaches.** The four threads at the top show how each would execute running alone on a standard superscalar processor without multithreading support. The three examples at the bottom show how they would execute running together in three multithreading options. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of gray and color correspond to four different threads in the multithreading processors. The additional pipeline start-up effects for coarse multithreading, which are not illustrated in this figure, would lead to further loss in throughput for coarse multithreading.

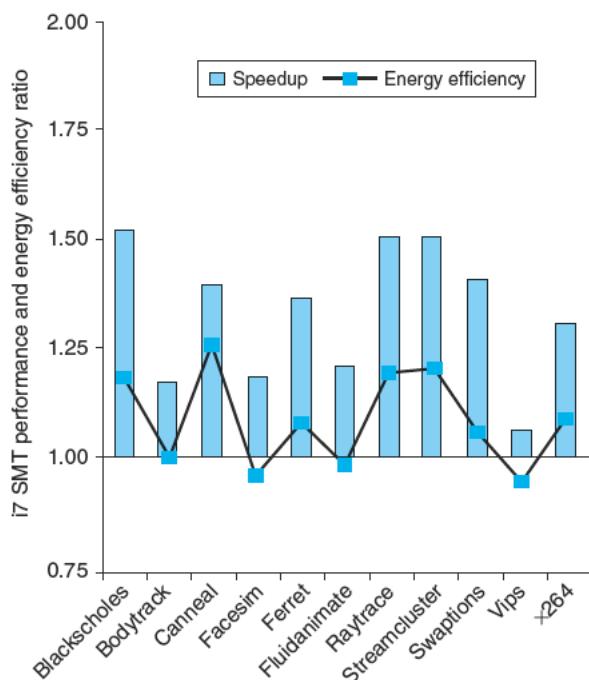
how four threads would execute independently on a superscalar with no multithreading support. The bottom portion shows how the four threads could be combined to execute on the processor more efficiently using three multithreading options:

- A superscalar with coarse-grained multithreading
- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading



In the superscalar without hardware multithreading support, the use of issue slots is limited by a lack of **instruction-level parallelism**. In addition, a major stall, such as an instruction cache miss, can leave the entire processor idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, the pipeline start-up overhead still leads to idle cycles, and limitations to ILP means all issue slots will not be used. In the fine-grained case, the interleaving of threads mostly eliminates idle clock cycles. Because only a single thread issues instructions in a given clock cycle, however, limitations in instruction-level parallelism still lead to idle slots within some clock cycles.



**FIGURE 6.6** The speed-up from using multithreading on one core on an I7 processor averages 1.31 for the PARSEC benchmarks (see [Section 6.9](#)) and the energy efficiency improvement is 1.07. This data was collected and analyzed by Esmaeilzadeh et. al. [2011].

In the SMT case, thread-level parallelism and instruction-level parallelism are both exploited, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors can restrict how many slots are used. Although Figure 6.5 greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

Figure 6.6 plots the performance and energy benefits of multithreading on a single processor of the Intel Core i7 960, which has hardware support for two threads. The average speed-up is 1.31, which is not bad given the modest extra resources for hardware multithreading. The average improvement in energy efficiency is 1.07, which is excellent. In general, you'd be happy with a performance speed-up being energy neutral.

Now that we have seen how multiple threads can utilize the resources of a single processor more effectively, we next show how to use them to exploit multiple processors.

1. True or false: Both multithreading and multicore rely on parallelism to get more efficiency from a chip.
2. True or false: *Simultaneous multithreading* (SMT) uses threads to improve resource utilization of a dynamically scheduled, out-of-order processor.

### Check Yourself

## 6.5

### Multicore and Other Shared Memory Multiprocessors

While hardware multithreading improved the efficiency of processors at modest cost, the big challenge of the last decade has been to deliver on the performance potential of Moore's Law by efficiently programming the increasing number of processors per chip.

Given the difficulty of rewriting old programs to run well on parallel hardware, a natural question is: what can computer designers do to simplify the task? One answer was to provide a single physical address space that all processors can share, so that programs need not concern themselves with where their data is, merely that programs may be executed in parallel. In this approach, all variables of a program can be made available at any time to any processor. The alternative is to have a separate address space per processor that requires that sharing must be explicit; we'll describe this option in the Section 6.7. When the physical address space is common then the hardware typically provides cache coherence to give a consistent view of the shared memory (see Section 5.8).

As mentioned above, a *shared memory multiprocessor* (SMP) is one that offers the programmer a *single physical address space* across all processors—which is

**uniform memory access (UMA)** A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

**nonuniform memory access (NUMA)** A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

**synchronization** The process of coordinating the behavior of two or more processes, which may be running on different processors.

**lock** A synchronization device that allows access to data to only one processor at a time.

nearly always the case for multicore chips—although a more accurate term would have been shared-*address* multiprocessor. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. Figure 6.7 shows the classic organization of an SMP. Note that such systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.

Single address space multiprocessors come in two styles. In the first style, the latency to a word in memory does not depend on which processor asks for it. Such machines are called **uniform memory access (UMA)** multiprocessors. In the second style, some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different microprocessors or to different memory controllers on the same chip. Such machines are called **nonuniform memory access (NUMA)** multiprocessors. As you might expect, the programming challenges are harder for a NUMA multiprocessor than for a UMA multiprocessor, but NUMAs can scale to larger sizes and NUMAs can have lower latency to nearby memory.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called **synchronization**, which we saw in Chapter 2. When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a **lock** for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable. Section 2.11 of Chapter 2 describes the instructions for locking in the MIPS instruction set.

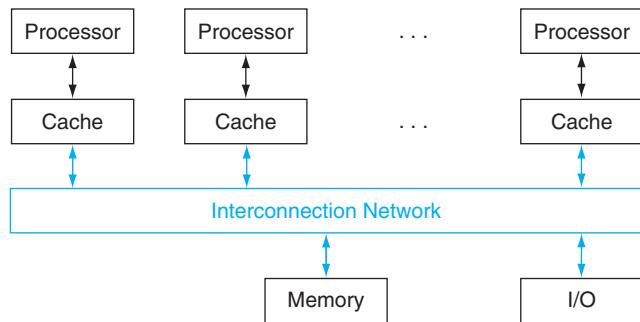


FIGURE 6.7 Classic organization of a shared memory multiprocessor.

### A Simple Parallel Processing Program for a Shared Address Space

Suppose we want to sum 64,000 numbers on a shared memory multiprocessor computer with uniform memory access time. Let's assume we have 64 processors.

### EXAMPLE

The first step is to ensure a balanced load per processor, so we split the set of numbers into subsets of the same size. We do not allocate the subsets to a different memory space, since there is a single memory space for this machine; we just give different starting addresses to each processor.  $P_n$  is the number that identifies the processor, between 0 and 63. All processors start the program by running a loop that sums their subset of numbers:

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i += 1)
    sum[Pn] += A[i]; /*sum the assigned areas*/
```

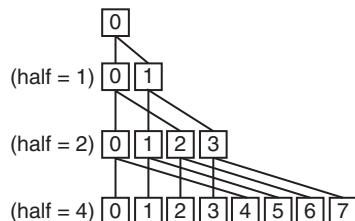
(Note the C code  $i \text{ } += \text{ } 1$  is just a shorter way to say  $i \text{ } = \text{ } i \text{ } + \text{ } 1$ .)

The next step is to add these 64 partial sums. This step is called a **reduction**, where we divide to conquer. Half of the processors add pairs of partial sums, and then a quarter add pairs of the new partial sums, and so on until we have the single, final sum. [Figure 6.8](#) illustrates the hierarchical nature of this reduction.

In this example, the two processors must synchronize before the “consumer” processor tries to read the result from the memory location written by the “producer” processor; otherwise, the consumer may read the old value of

### ANSWER

**reduction** A function that processes a data structure and returns a single value.



**FIGURE 6.8 The last four levels of a reduction that sums results from each processor, from bottom to top.** For all processors whose number  $i$  is less than half, add the sum produced by processor number  $(i + \text{half})$  to its sum.

the data. We want each processor to have its own version of the loop counter variable *i*, so we must indicate that it is a “private” variable. Here is the code (*half* is private also):

```
half = 64; /*64 processors in multiprocessor*/
do
    synch(); /*wait for partial sum completion*/
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1];
    /*Conditional sum needed when half is
    odd; Processor0 gets missing element */
    half = half/2; /*dividing line on who sums */
    if (Pn < half) sum[Pn] += sum[Pn+half];
while (half > 1); /*exit with final sum in Sum[0] */
```

## Hardware/ Software Interface

**OpenMP** An API for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

Given the long-term interest in parallel programming, there have been hundreds of attempts to build parallel programming systems. A limited but popular example is **OpenMP**. It is just an *Application Programmer Interface* (API) along with a set of compiler directives, environment variables, and runtime library routines that can extend standard programming languages. It offers a portable, scalable, and simple programming model for shared memory multiprocessors. Its primary goal is to parallelize loops and to perform reductions.

Most C compilers already have support for OpenMP. The command to uses the OpenMP API with the UNIX C compiler is just:

```
cc -fopenmp foo.c
```

OpenMP extends C using *pragmas*, which are just commands to the C macro preprocessor like `#define` and `#include`. To set the number of processors we want to use to be 64, as we wanted in the example above, we just use the command

```
#define P 64 /* define a constant that we'll use a few times */
#pragma omp parallel num_threads(P)
```

That is, the runtime libraries should use 64 parallel threads.

To turn the sequential for loop into a parallel for loop that divides the work equally between all the threads that we told it to use, we just write (assuming *sum* is initialized to 0)

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
    for (i = 0; 1000*Pn; i < 1000*(Pn+1); i += 1)
        sum[Pn] += A[i]; /*sum the assigned areas*/
```

To perform the reduction, we can use another command that tells OpenMP what the reduction operator is and what variable you need to use to place the result of the reduction.

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
    FinalSum += sum[i]; /* Reduce to a single number */
```

Note that it is now up to the OpenMP library to find efficient code to sum 64 numbers efficiently using 64 processors.

While OpenMP makes it easy to write simple parallel code, it is not very helpful with debugging, so many parallel programmers use more sophisticated parallel programming systems than OpenMP, just as many programmers today use more productive languages than C.

---

Given this tour of classic MIMD hardware and software, our next path is a more exotic tour of a type of MIMD architecture with a different heritage and thus a very different perspective on the parallel programming challenge.

True or false: Shared memory multiprocessors cannot take advantage of task-level parallelism.

**Check  
Yourself**

**Elaboration:** Some writers repurposed the acronym SMP to mean *symmetric multiprocessor*, to indicate that the latency from processor to memory was about the same for all processors. This shift was done to contrast them from large-scale NUMA multiprocessors, as both classes used a single address space. As clusters proved much more popular than large-scale NUMA multiprocessors, in this book we restore SMP to its original meaning, and use it to contrast against that use multiple address spaces, such as clusters.

**Elaboration:** An alternative to sharing the physical address space would be to have separate physical address spaces but share a common virtual address space, leaving it up to the operating system to handle communication. This approach has been tried, but it has too high an overhead to offer a practical shared memory abstraction to the performance-oriented programmer.



## 6.6

# Introduction to Graphics Processing Units

The original justification for adding SIMD instructions to existing architectures was that many microprocessors were connected to graphics displays in PCs and workstations, so an increasing fraction of processing time was used for graphics. As Moore’s Law increased the number of transistors available to microprocessors, it therefore made sense to improve graphics processing.

A major driving force for improving graphics processing was the computer game industry, both on PCs and in dedicated game consoles such as the Sony PlayStation. The rapidly growing game market encouraged many companies to make increasing investments in developing faster graphics hardware, and this positive feedback loop led graphics processing to improve at a faster rate than general-purpose processing in mainstream microprocessors.

Given that the graphics and game community had different goals than the microprocessor development community, it evolved its own style of processing and terminology. As the graphics processors increased in power, they earned the name *Graphics Processing Units* or *GPUs* to distinguish themselves from CPUs.

For a few hundred dollars, anyone can buy a GPU today with hundreds of parallel floating-point units, which makes high-performance computing more accessible. The interest in GPU computing blossomed when this potential was combined with a programming language that made GPUs easier to program. Hence, many programmers of scientific and multimedia applications today are pondering whether to use GPUs or CPUs.

(This section concentrates on using GPUs for computing. To see how GPU computing combines with the traditional role of graphics acceleration, see [Appendix C.](#))

Here are some of the key characteristics as to how GPUs vary from CPUs:

- GPUs are accelerators that supplement a CPU, so they do not need to be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics. It’s fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed.
- The GPU problem sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.

These differences led to different styles of architecture:

- Perhaps the biggest difference is that GPUs do not rely on multilevel caches to overcome the long latency to memory, as do CPUs. Instead, GPUs rely on hardware multithreading (Section 6.4) to hide the latency to memory. That is, between the time of a memory request and the time that data arrives, the GPU executes hundreds or thousands of threads that are independent of that request.

- The GPU memory is thus oriented toward bandwidth rather than latency. There are even special graphics DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs. In addition, GPU memories have traditionally had smaller main memories than conventional microprocessors. In 2013, GPUs typically have 4 to 6 GiB or less, while CPUs have 32 to 256 GiB. Finally, keep in mind that for general-purpose computation, you must include the time to transfer the data between CPU memory and GPU memory, since the GPU is a coprocessor.
- Given the reliance on many threads to deliver good memory bandwidth, GPUs can accommodate many parallel processors (MIMD) as well as many threads. Hence, each GPU processor is more highly multithreaded than a typical CPU, plus they have more processors.

Although GPUs were designed for a narrower set of applications, some programmers wondered if they could specify their applications in a form that would let them tap the high potential performance of GPUs. After tiring of trying to specify their problems using the graphics APIs and languages, they developed C-inspired programming languages to allow them to write programs directly for the GPUs. An example is NVIDIA's CUDA (Compute Unified Device Architecture), which enables the programmer to write C programs to execute on GPUs, albeit with some restrictions.  [Appendix C](#) gives examples of CUDA code. (OpenCL is a multi-company initiative to develop a portable programming language that provides many of the benefits of CUDA.)

NVIDIA decided that the unifying theme of all these forms of parallelism is the *CUDA Thread*. Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism. These threads are blocked together and executed in groups of 32 at a time. A multithreaded processor inside a GPU executes these blocks of threads, and a GPU consists of 8 to 32 of these multithreaded processors.

## Hardware/ Software Interface

### An Introduction to the NVIDIA GPU Architecture

We use NVIDIA systems as our example as they are representative of GPU architectures. Specifically, we follow the terminology of the CUDA parallel programming language and use the Fermi architecture as the example.

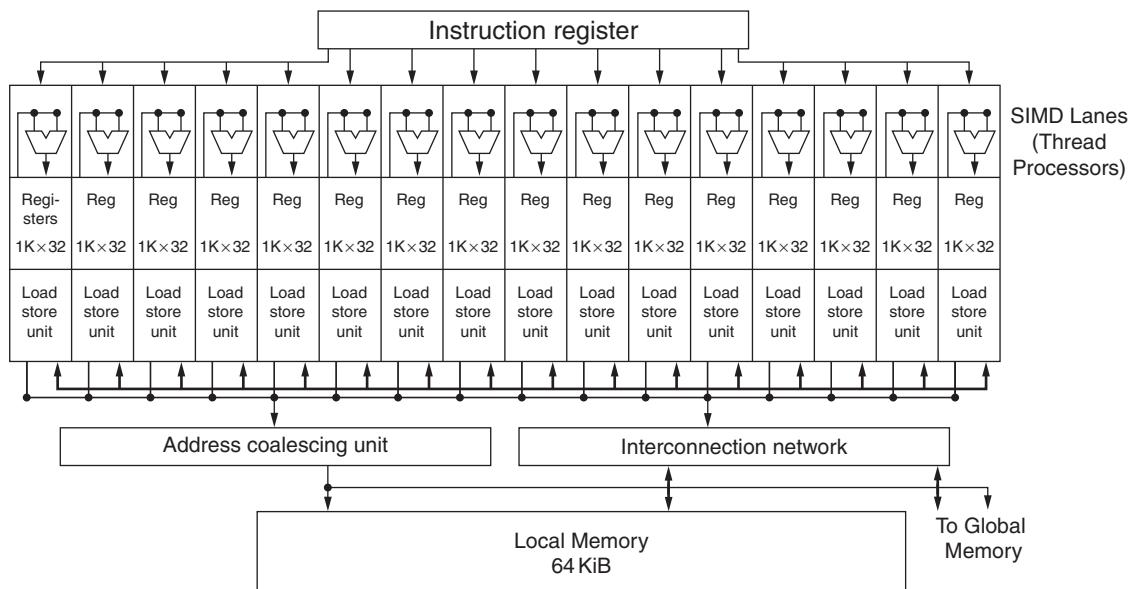
Like vector architectures, GPUs work well only with data-level parallel problems. Both styles have gather-scatter data transfers, and GPU processors have even more

registers than do vector processors. Unlike most vector architectures, GPUs also rely on hardware multithreading within a single multi-threaded SIMD processor to hide memory latency (see Section 6.4).

A multithreaded SIMD processor is similar to a Vector Processor, but the former has many parallel functional units instead of just a few that are deeply pipelined, as does the latter.

As mentioned above, a GPU contains a collection of multithreaded SIMD processors; that is, a GPU is a MIMD composed of multithreaded SIMD processors. For example, NVIDIA has four implementations of the Fermi architecture at different price points with 7, 11, 14, or 15 multithreaded SIMD processors. To provide transparent scalability across models of GPUs with differing number of multithreaded SIMD processors, the Thread Block Scheduler hardware assigns blocks of threads to multithreaded SIMD processors. [Figure 6.9](#) shows a simplified block diagram of a multithreaded SIMD processor.

Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*, which we will also call a *SIMD thread*. It is a traditional thread, but it contains exclusively SIMD instructions. These SIMD threads have their own program counters and they run on a multithreaded SIMD processor. The *SIMD Thread Scheduler* includes a controller that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded



**FIGURE 6.9 Simplified block diagram of the datapath of a multithreaded SIMD Processor.**  
It has 16 SIMD lanes. The SIMD Thread Scheduler has many independent SIMD threads that it chooses from to run on this processor.

SIMD processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor (see Section 6.4), except that it is scheduling threads of SIMD instructions. Thus, GPU hardware has two levels of hardware schedulers:

1. The *Thread Block Scheduler* that assigns blocks of threads to multithreaded SIMD processors, and
2. the SIMD Thread Scheduler *within* a SIMD processor, which schedules when SIMD threads should run.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions would compute 32 of the elements of the computation. Since the thread consists of SIMD instructions, the SIMD processor must have parallel functional units to perform the operation. We call them *SIMD Lanes*, and they are quite similar to the Vector Lanes in Section 6.3.

**Elaboration:** The number of lanes per SIMD processor varies across GPU generations. With Fermi, each 32-wide thread of SIMD instructions is mapped to 16 SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete. Each thread of SIMD instructions is executed in lock step. Staying with the analogy of a SIMD processor as a vector processor, you could say that it has 16 lanes, and the vector length would be 32. This wide but shallow nature is why we use the term SIMD processor instead of vector processor, as it is more intuitive.

Since by definition the threads of SIMD instructions are independent, the SIMD Thread Scheduler can pick whatever thread of SIMD instructions is ready, and need not stick with the next SIMD instruction in the sequence within a single thread. Thus, using the terminology of Section 6.4, it uses fine-grained multithreading.

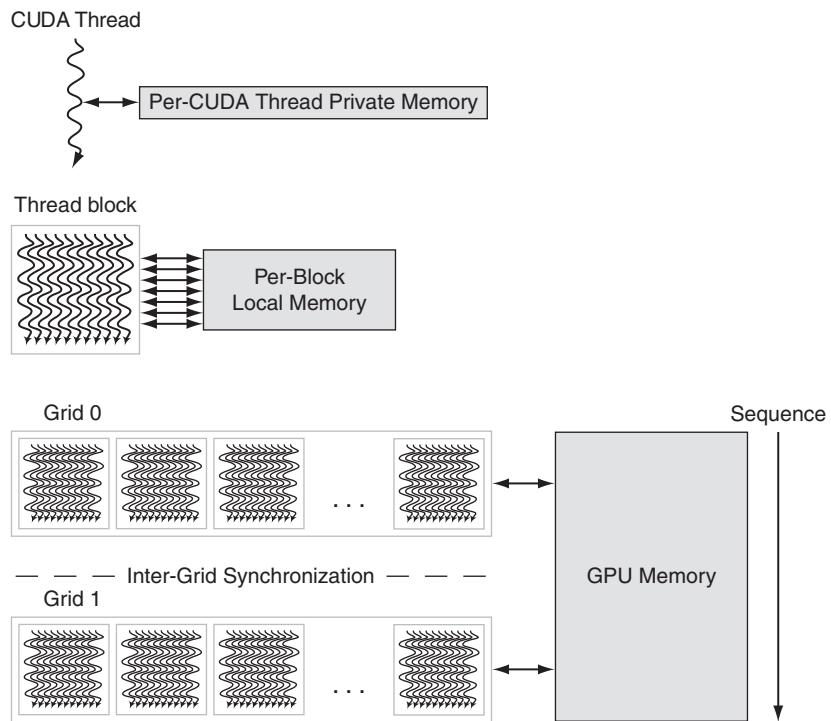
To hold these memory elements, a Fermi SIMD processor has an impressive 32,768 32-bit registers. Just like a vector processor, these registers are divided logically across the vector lanes or, in this case, SIMD Lanes. Each SIMD Thread is limited to no more than 64 registers, so you might think of a SIMD Thread as having up to 64 vector registers, with each vector register having 32 elements and each element being 32 bits wide.

Since Fermi has 16 SIMD Lanes, each contains 2048 registers. Each CUDA Thread gets one element of each of the vector registers. Note that a CUDA thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by one SIMD Lane. Beware that CUDA Threads are very different from POSIX threads; you can't make arbitrary system calls or synchronize arbitrarily in a CUDA Thread.

## NVIDIA GPU Memory Structures

Figure 6.10 shows the memory structures of an NVIDIA GPU. We call the on-chip memory that is local to each multithreaded SIMD processor *Local Memory*. It is shared by the SIMD Lanes within a multithreaded SIMD processor, but this memory is not shared between multithreaded SIMD processors. We call the off-chip DRAM shared by the whole GPU and all thread blocks *GPU Memory*.

Rather than rely on large caches to contain the whole working sets of an application, GPUs traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM,



**FIGURE 6.10 GPU Memory structures.** GPU Memory is shared by the vectorized loops. All threads of SIMD instructions within a thread block share Local Memory.

since their working sets can be hundreds of megabytes. Thus, they will not fit in the last level cache of a multicore microprocessor. Given the use of hardware multithreading to hide DRAM latency, the chip area used for caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of the many threads of SIMD instructions.

**Elaboration:** While hiding memory latency is the underlying philosophy, note that the latest GPUs and vector processors have added caches. For example, the recent Fermi architecture has added caches, but they are thought of as either bandwidth filters to reduce demands on GPU Memory or as accelerators for the few variables whose latency cannot be hidden by multithreading. Local memory for stack frames, function calls, and register spilling is a good match to caches, since latency matters when calling a function. Caches can also save energy, since on-chip cache accesses take much less energy than accesses to multiple, external DRAM chips.

## Putting GPUs into Perspective

At a high level, multicore computers with SIMD instruction extensions do share similarities with GPUs. Figure 6.11 summarizes the similarities and differences. Both are MIMDs whose processors use multiple SIMD lanes, although GPUs have more processors and many more lanes. Both use hardware multithreading to improve processor utilization, although GPUs have hardware support for many more threads. Both use caches, although GPUs use smaller streaming caches and multicore computers use large multilevel caches that try to contain whole working sets completely. Both use a 64-bit address space, although the physical main memory is much smaller in GPUs. While GPUs support memory protection at the page level, they do not yet support demand paging.

SIMD processors are also similar to vector processors. The multiple SIMD processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors. This view would consider the Fermi GTX 580 as a 16-core machine with hardware support for multithreading, where each core has 16 lanes. The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

GPUs and CPUs do not go back in computer architecture genealogy to a common ancestor; there is no Missing Link that explains both. As a result of this uncommon heritage, GPUs have not used the terms common in the computer architecture community, which has led to confusion about what GPUs are and how they work. To help resolve the confusion, Figure 6.12 (from left to right) lists the more descriptive term used in this section, the closest term from mainstream computing, the official NVIDIA GPU term in case you are interested, and then a short description of the term. This “GPU Rosetta Stone” may help relate this section and ideas to more conventional GPU descriptions, such as those found in [Appendix C](#).

While GPUs are moving toward mainstream computing, they can't abandon their responsibility to continue to excel at graphics. Thus, the design of GPUs may

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Largest cache size	8 MIB	0.75 MIB
Size of memory address	64-bit	64-bit
Size of main memory	8 GiB to 256 GiB	4 GiB to 6 GiB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Cache coherent	Yes	No

**FIGURE 6.11 Similarities and differences between multicore with Multimedia SIMD extensions and recent GPUs.**

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

**FIGURE 6.12 Quick guide to GPU terms.** We use the first column for hardware terms. Four groups cluster these 12 terms. From top to bottom: Program Abstractions, Machine Objects, Processing Hardware, and Memory Hardware.

make more sense when architects ask, given the hardware invested to do graphics well, how can we supplement it to improve the performance of a wider range of applications?

Having covered two different styles of MIMD that have a shared address space, we next introduce parallel processors where each processor has its own private address space, which makes it much easier to build much larger systems. The Internet services that you use every day depend on these large scale systems.

**Elaboration:** While the GPU was introduced as having a separate memory from the CPU, both AMD and Intel have announced “fused” products that combine GPUs and CPUs to share a single memory. The challenge will be to maintain the high bandwidth memory in a fused architecture that has been a foundation of GPUs.

True or false: GPUs rely on graphics DRAM chips to reduce memory latency and thereby increase performance on graphics applications.

### Check Yourself

## 6.7

# Clusters, Warehouse Scale Computers, and Other Message-Passing Multiprocessors

The alternative approach to sharing an address space is for the processors to each have their own private physical address space. Figure 6.13 shows the classic organization of a multiprocessor with multiple private address spaces. This alternative multiprocessor must communicate via explicit **message passing**, which traditionally is the name of such style of computers. Provided the system has routines to **send** and **receive messages**, coordination is built in with message passing, since one processor knows when a message is sent, and the receiving processor knows when a message arrives. If the sender needs confirmation that the message has arrived, the receiving processor can then send an acknowledgment message back to the sender.

There have been several attempts to build large-scale computers based on high-performance message-passing networks, and they do offer better absolute

#### message passing

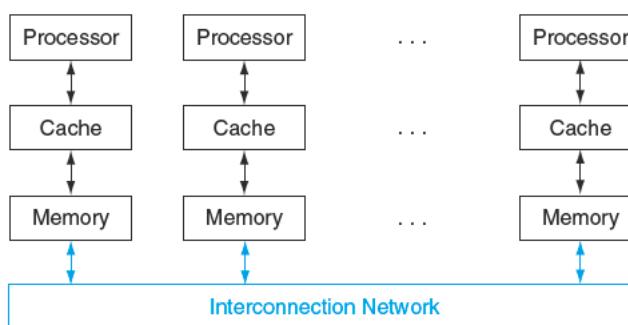
Communicating between multiple processors by explicitly sending and receiving information.

#### send message routine

A routine used by a processor in machines with private memories to pass a message to another processor.

#### receive message routine

A routine used by a processor in machines with private memories to accept a message from another processor.



**FIGURE 6.13 Classic organization of a multiprocessor with multiple private address spaces, traditionally called a message-passing multiprocessor.** Note that unlike the SMP in Figure 6.7, the interconnection network is not between the caches and memory but is instead between processor-memory nodes.

communication performance than clusters built using local area networks. Indeed, many supercomputers today use custom networks. The problem is that they are much more expensive than local area networks like Ethernet. Few applications today outside of high performance computing can justify the higher communication performance, given the much higher costs.

---

## Hardware/ Software Interface

Computers that rely on message passing for communication rather than cache coherent shared memory are much easier for hardware designers to build (see Section 5.8). There is an advantage for programmers as well, in that communication is explicit, which means there are fewer performance surprises than with the implicit communication in cache-coherent shared memory computers. The downside for programmers is that it's harder to port a sequential program to a message-passing computer, since every communication must be identified in advance or the program doesn't work. Cache-coherent shared memory allows the hardware to figure out what data needs to be communicated, which makes porting easier. There are differences of opinion as to which is the shortest path to high performance, given the pros and cons of implicit communication, but there is no confusion in the marketplace today. Multicore microprocessors use shared physical memory and nodes of a cluster communicate with each other using message passing.

---

**clusters** Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessor.

Some concurrent applications run well on parallel hardware, independent of whether it offers shared addresses or message passing. In particular, task-level parallelism and applications with little communication—like Web search, mail servers, and file servers—do not require shared addressing to run well. As a result, **clusters** have become the most widespread example today of the message-passing parallel computer. Given the separate memories, each node of a cluster runs a distinct copy of the operating system. In contrast, the cores inside a microprocessor are connected using a high-speed network inside the chip, and a multichip shared-memory system uses the memory interconnect for communication. The memory interconnect has higher bandwidth and lower latency, allowing much better communication performance for shared memory multiprocessors.

The weakness of separate memories for user memory from a parallel programming perspective turns into a strength in system dependability (see Section 5.5). Since a cluster consists of independent computers connected through a local area network, it is much easier to replace a computer without bringing down the system in a cluster than in a shared memory multiprocessor. Fundamentally, the shared address means that it is difficult to isolate a processor and replace it without heroic work by the operating system and in the physical design of the server. It is also easy for clusters to scale down gracefully when a server fails, thereby improving **dependability**. Since the cluster software is a layer that runs on top of the local operating systems running on each computer, it is much easier to disconnect and replace a broken computer.



Given that clusters are constructed from whole computers and independent, scalable networks, this isolation also makes it easier to expand the system without bringing down the application that runs on top of the cluster.

Their lower cost, higher availability, and rapid, incremental expandability make clusters attractive to service Internet providers, despite their poorer communication performance when compared to large-scale shared memory multiprocessors. The search engines that hundreds of millions of us use every day depend upon this technology. Amazon, Facebook, Google, Microsoft, and others all have multiple datacenters each with clusters of tens of thousands of servers. Clearly, the use of multiple processors in Internet service companies has been hugely successful.

## Warehouse-Scale Computers

Internet services, such as those described above, necessitated the construction of new buildings to house, power, and cool 100,000 servers. Although they may be classified as just large clusters, their architecture and operation are more sophisticated. They act as one giant computer and cost on the order of \$150M for the building, the electrical and cooling infrastructure, the servers, and the networking equipment that connects and houses 50,000 to 100,000 servers. We consider them a new class of computer, called *Warehouse-Scale Computers* (WSC).

*Anyone can build a fast CPU. The trick is to build a fast system.*

Seymour Cray, considered the father of the supercomputer.

The most popular framework for batch processing in a WSC is MapReduce [Dean, 2008] and its open-source twin Hadoop. Inspired by the Lisp functions of the same name, Map first applies a programmer-supplied function to each logical input record. Map runs on thousands of servers to produce an intermediate result of key-value pairs. Reduce collects the output of those distributed tasks and collapses them using another programmer-defined function. With appropriate software support, both are highly parallel yet easy to understand and to use. Within 30 minutes, a novice programmer can run a MapReduce task on thousands of servers.

For example, one MapReduce program calculates the number of occurrences of every English word in a large collection of documents. Below is a simplified version of that program, which shows just the inner loop and assumes just one occurrence of all English words found in a document:

## Hardware/ Software Interface

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1"); // Produce list of all words
    reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int result = 0;
        for each v in values:
            result += ParseInt(v); // get integer from key-value pair
        Emit(AsString(result));
```

The function `EmitIntermediate` used in the `Map` function emits each word in the document and the value one. Then the `Reduce` function sums all the values per word for each document using `ParseInt()` to get the number of occurrences per word in all documents. The MapReduce runtime environment schedules map tasks and reduce tasks to the servers of a WSC.

---

At this extreme scale, which requires innovation in power distribution, cooling, monitoring, and operations, the WSC is a modern descendant of the 1970s supercomputers—making Seymour Cray the godfather of today’s WSC architects. His extreme computers handled computations that could be done nowhere else, but were so expensive that only a few companies could afford them. This time the target is providing information technology for the world instead of high performance computing for scientists and engineers. Hence, WSCs surely play a more important societal role today than Cray’s supercomputers did in the past.

While they share some common goals with servers, WSCs have three major distinctions:

1. *Ample, easy parallelism:* A concern for a server architect is whether the applications in the targeted marketplace have enough parallelism to justify the amount of parallel hardware and whether the cost is too high for sufficient communication hardware to exploit this parallelism. A WSC architect has no such concern. First, batch applications like MapReduce benefit from the large number of independent data sets that need independent processing, such as billions of Web pages from a Web crawl. Second, interactive Internet service applications, also known as **Software as a Service (SaaS)**, can benefit from millions of independent users of interactive Internet services. Reads and writes are rarely dependent in SaaS, so SaaS rarely needs to synchronize. For example, search uses a read-only index and email is normally reading and writing independent information. We call this type of easy parallelism *Request-Level Parallelism*, as many independent efforts can proceed in parallel naturally with little need for communication or synchronization.
2. *Operational Costs Count:* Traditionally, server architects design their systems for peak performance within a cost budget and worry about energy only to make sure they don’t exceed the cooling capacity of their enclosure. They usually ignored operational costs of a server, assuming that they pale in comparison to purchase costs. WSC have longer lifetimes—the building and electrical and cooling infrastructure are often amortized over 10 or more years—so the operational costs add up: energy, power distribution, and cooling represent more than 30% of the costs of a WSC over 10 years.
3. *Scale and the Opportunities/Problems Associated with Scale:* To construct a single WSC, you must purchase 100,000 servers along with the supporting infrastructure, which means volume discounts. Hence, WSCs are so massive

**software as a service (SaaS)** Rather than selling software that is installed and run on customers’ own computers, software is run at a remote site and made available over the Internet typically via a Web interface to customers. SaaS customers are charged based on use versus on ownership.



internally that you get economy of scale even if there are not many WSCs. These economies of scale led to *cloud computing*, as the lower per unit costs of a WSC meant that cloud companies could rent servers at a profitable rate and still be below what it costs outsiders to do it themselves. The flip side of the economic opportunity of scale is the need to cope with the failure frequency of scale. Even if a server had a Mean Time To Failure of an amazing 25 years (200,000 hours), the WSC architect would need to design for 5 server failures every day. Section 5.15 mentioned annualized disk failure rate (AFR) was measured at Google at 2% to 4%. If there were 4 disks per server and their annual failure rate was 2%, the WSC architect should expect to see one disk fail every *hour*. Thus, fault tolerance is even more important for the WSC architect than the server architect.

The economies of scale uncovered by WSC have realized the long dreamed of goal of computing as a utility. Cloud computing means anyone anywhere with good ideas, a business model, and a credit card can tap thousands of servers to deliver their vision almost instantly around the world. Of course, there are important obstacles that could limit the growth of cloud computing—such as security, privacy, standards, and the rate of growth of Internet bandwidth—but we foresee them being addressed so that WSCs and cloud computing can flourish.

To put the growth rate of cloud computing into perspective, in 2012 Amazon Web Services announced that it adds enough new server capacity *every day* to support all of Amazon's global infrastructure as of 2003, when Amazon was a \$5.2Bn annual revenue enterprise with 6000 employees.

Now that we understand the importance of message-passing multiprocessors, especially for cloud computing, we next cover ways to connect the nodes of a WSC together. Thanks to **Moore's Law** and the increasing number of cores per chip, we now need networks inside a chip as well, so these topologies are important in the small as well as in the large.



MOORE'S LAW

**Elaboration:** The MapReduce framework shuffles and sorts the key-value pairs at the end of the Map phase to produce groups that all share the same key. These groups are then passed to the Reduce phase.

**Elaboration:** Another form of large scale computing is *grid computing*, where the computers are spread across large areas, and then the programs that run across them must communicate via long haul networks. The most popular and unique form of grid computing was pioneered by the SETI@home project. As millions of PCs are idle at any one time doing nothing useful, they could be harvested and put to good uses if someone developed software that could run on those computers and then gave each PC an independent piece of the problem to work on. The first example was the Search for ExtraTerrestrial Intelligence (SETI), which was launched at UC Berkeley in 1999. Over 5 million computer users in more than 200 countries have signed up for SETI@home, with more than 50% outside the US. By the end of 2011, the average performance of the SETI@home grid was 3.5 PetaFLOPS.

**Check  
Yourself**

1. True or false: Like SMPs, message-passing computers rely on locks for synchronization.
2. True or false: Clusters have separate memories and thus need many copies of the operating system.

**6.8****Introduction to Multiprocessor Network Topologies**

Multicore chips require on-chip networks to connect cores together, and clusters require local area networks to connect servers together. This section reviews the pros and cons of different interconnection network topologies.

Network costs include the number of switches, the number of links on a switch to connect to the network, the width (number of bits) per link, and length of the links when the network is mapped into silicon. For example, some cores or servers may be adjacent and others may be on the other side of the chip or the other side of the datacenter. Network performance is multifaceted as well. It includes the latency on an unloaded network to send and receive a message, the throughput in terms of the maximum number of messages that can be transmitted in a given time period, delays caused by contention for a portion of the network, and variable performance depending on the pattern of communication. Another obligation of the network may be fault tolerance, since systems may be required to operate in the presence of broken components. Finally, in this era of energy-limited systems, the energy efficiency of different organizations may trump other concerns.

Networks are normally drawn as graphs, with each edge of the graph representing a link of the communication network. In the figures in this section, the processor-memory node is shown as a black square and the switch is shown as a colored circle. We assume here that all links are *bidirectional*; that is, information can flow in either direction. All networks consist of *switches* whose links go to processor-memory nodes and to other switches. The first network connects a sequence of nodes together:



This topology is called a *ring*. Since some nodes are not directly connected, some messages will have to hop along intermediate nodes until they arrive at the final destination.

Unlike a bus—a shared set of wires that allows broadcasting to all connected devices—a ring is capable of many simultaneous transfers.

Because there are numerous topologies to choose from, performance metrics are needed to distinguish these designs. Two are popular. The first is **total network bandwidth**, which is the bandwidth of each link multiplied by the number of links. This represents the peak bandwidth. For the ring network above, with  $P$  processors, the total network bandwidth would be  $P$  times the bandwidth of one link; the total network bandwidth of a bus is just the bandwidth of that bus.

To balance this best bandwidth case, we include another metric that is closer to the worst case: the **bisection bandwidth**. This metric is calculated by dividing the machine into two halves. Then you sum the bandwidth of the links that cross that imaginary dividing line. The bisection bandwidth of a ring is two times the link bandwidth. It is one times the link bandwidth for the bus. If a single link is as fast as the bus, the ring is only twice as fast as a bus in the worst case, but it is  $P$  times faster in the best case.

Since some network topologies are not symmetric, the question arises of where to draw the imaginary line when bisecting the machine. Bisection bandwidth is a worst-case metric, so the answer is to choose the division that yields the most pessimistic network performance. Stated alternatively, calculate all possible bisection bandwidths and pick the smallest. We take this pessimistic view because parallel programs are often limited by the weakest link in the communication chain.

At the other extreme from a ring is a **fully connected network**, where every processor has a bidirectional link to every other processor. For fully connected networks, the total network bandwidth is  $P \times (P - 1)/2$ , and the bisection bandwidth is  $(P/2)^2$ .

The tremendous improvement in performance of fully connected networks is offset by the tremendous increase in cost. This consequence inspires engineers to invent new topologies that are between the cost of rings and the performance of fully connected networks. The evaluation of success depends in large part on the nature of the communication in the workload of parallel programs run on the computer.

The number of different topologies that have been discussed in publications would be difficult to count, but only a few have been used in commercial parallel processors. [Figure 6.14](#) illustrates two of the popular topologies.

An alternative to placing a processor at every node in a network is to leave only the switch at some of these nodes. The switches are smaller than processor-memory-switch nodes, and thus may be packed more densely, thereby lessening distance and increasing performance. Such networks are frequently called **multistage networks** to reflect the multiple steps that a message may travel. Types of multistage networks are as numerous as single-stage networks; [Figure 6.15](#) illustrates two of the popular multistage organizations. A **fully connected** or **crossbar network** allows any node to communicate with any other node in one pass through the network. An *Omega network* uses less hardware than the crossbar network ( $2n \log_2 n$  versus  $n^2$  switches), but contention can occur between messages, depending on the pattern

#### **network bandwidth**

Informally, the peak transfer rate of a network; can refer to the speed of a single link or the collective transfer rate of all links in the network.

#### **bisection bandwidth**

The bandwidth between two equal parts of a multiprocessor. This measure is for a worst case split of the multiprocessor.

#### **fully connected network**

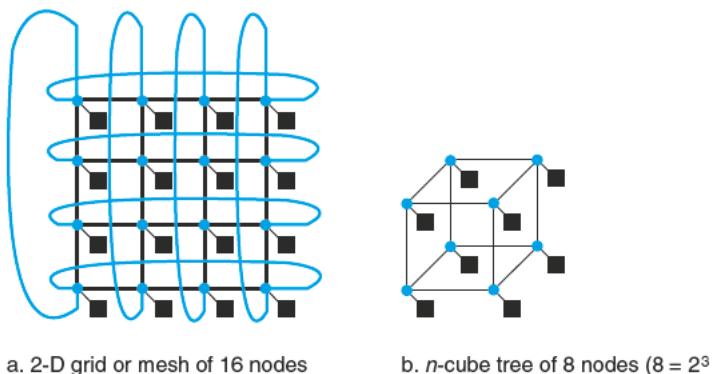
A network that connects processor-memory nodes by supplying a dedicated communication link between every node.

#### **multistage network**

A network that supplies a small switch at each node.

#### **crossbar network**

A network that allows any node to communicate with any other node in one pass through the network.



**FIGURE 6.14 Network topologies that have appeared in commercial parallel processors.**

The colored circles represent switches and the black squares represent processor-memory nodes. Even though a switch has many links, generally only one goes to the processor. The Boolean  $n$ -cube topology is an  $n$ -dimensional interconnect with  $2n$  nodes, requiring  $n$  links per switch (plus one for the processor) and thus  $n$  nearest-neighbor nodes. Frequently, these basic topologies have been supplemented with extra arcs to improve performance and reliability.

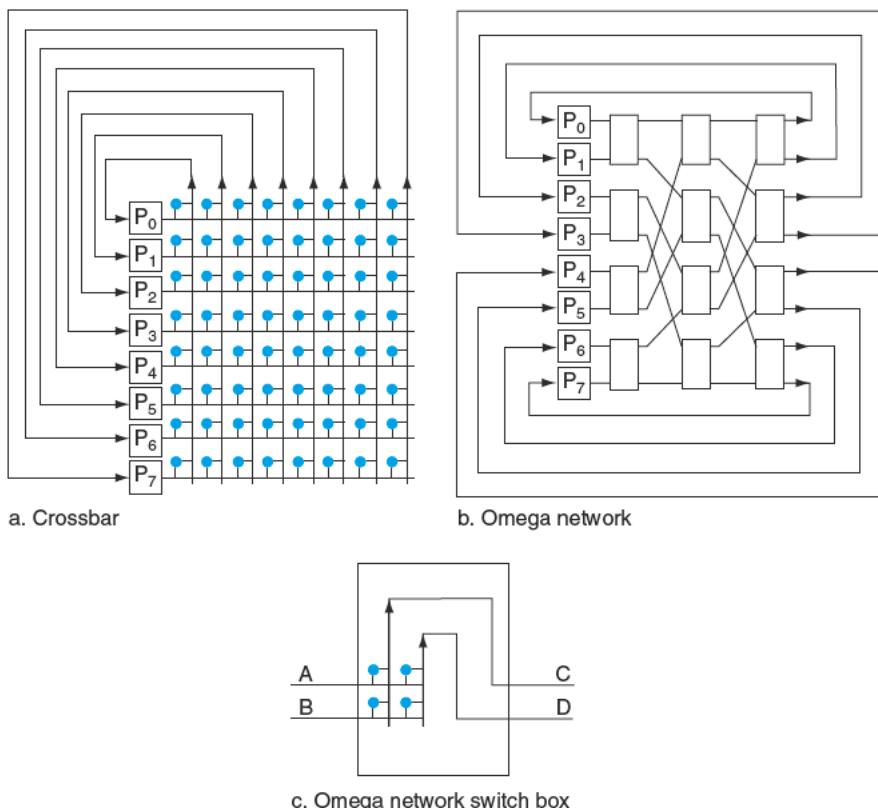
of communication. For example, the Omega network in Figure 6.15 cannot send a message from  $P_0$  to  $P_6$  at the same time that it sends a message from  $P_1$  to  $P_4$ .

### Implementing Network Topologies

This simple analysis of all the networks in this section ignores important practical considerations in the construction of a network. The distance of each link affects the cost of communicating at a high clock rate—generally, the longer the distance, the more expensive it is to run at a high clock rate. Shorter distances also make it easier to assign more wires to the link, as the power to drive many wires is less if the wires are short. Shorter wires are also cheaper than longer wires. Another practical limitation is that the three-dimensional drawings must be mapped onto chips that are essentially two-dimensional media. The final concern is energy. Energy concerns may force multicore chips to rely on simple grid topologies, for example. The bottom line is that topologies that appear elegant when sketched on the blackboard may be impractical when constructed in silicon or in a datacenter.

Now that we understand the importance of clusters and have seen topologies that we can follow to connect them together, we next look at the hardware and software of the interface of the network to the processor.

**Check Yourself** True or false: For a ring with  $P$  nodes, the ratio of the total network bandwidth to the bisection bandwidth is  $P/2$ .



**FIGURE 6.15 Popular multistage network topologies for eight nodes.** The switches in these drawings are simpler than in earlier drawings because the links are unidirectional; data comes in at the left and exits out the right link. The switch box in c can pass A to C and B to D or B to C and A to D. The crossbar uses  $n^2$  switches, where  $n$  is the number of processors, while the Omega network uses  $2n \log_2 n$  of the large switch boxes, each of which is logically composed of four of the smaller switches. In this case, the crossbar uses 64 switches versus 12 switch boxes, or 48 switches, in the Omega network. The crossbar, however, can support any combination of messages between processors, while the Omega network cannot.



## Communicating to the Outside World: Cluster Networking

This online section describes the networking hardware and software used to connect the nodes of a cluster together. The example is 10 gigabit/second Ethernet connected to the computer using *Peripheral Component Interconnect Express* (PCIe). It shows both software and hardware optimizations how to improve network performance, including zero copy messaging, user space communication, using polling instead of I/O interrupts, and hardware calculation of checksums. While the example is networking, the techniques in this section apply to storage controllers and other I/O devices as well.

After covering the performance of network at a low level of detail in this online section, the next section shows how to benchmark multiprocessors of all kinds with much higher-level programs.

## 6.10

### Multiprocessor Benchmarks and Performance Models

As we saw in Chapter 1, benchmarking systems is always a sensitive topic, because it is a highly visible way to try to determine which system is better. The results affect not only the sales of commercial systems, but also the reputation of the designers of those systems. Hence, all participants want to win the competition, but they also want to be sure that if someone else wins, they deserve to win because they have a genuinely better system. This desire leads to rules to ensure that the benchmark results are not simply engineering tricks for that benchmark, but are instead advances that improve performance of real applications.

To avoid possible tricks, a typical rule is that you can't change the benchmark. The source code and data sets are fixed, and there is a single proper answer. Any deviation from those rules makes the results invalid.

Many multiprocessor benchmarks follow these traditions. A common exception is to be able to increase the size of the problem so that you can run the benchmark on systems with a widely different number of processors. That is, many benchmarks allow weak scaling rather than require strong scaling, even though you must take care when comparing results for programs running different problem sizes.

Figure 6.16 gives a summary of several parallel benchmarks, also described below:

- *Linpack* is a collection of linear algebra routines, and the routines for performing Gaussian elimination constitute what is known as the Linpack benchmark. The DGEMM routine in the example on page 215 represents a small fraction of the source code of the Linpack benchmark, but it accounts for most of the execution time for the benchmark. It allows weak scaling, letting the user pick any size problem. Moreover, it allows the user to rewrite Linpack in almost any form and in any language, as long as it computes the proper result and performs the same number of floating point operations for a given problem size. Twice a year, the 500 computers with the fastest Linpack performance are published at [www.top500.org](http://www.top500.org). The first on this list is considered by the press to be the world's fastest computer.
- *SPECrate* is a throughput metric based on the SPEC CPU benchmarks, such as SPEC CPU 2006 (see Chapter 1). Rather than report performance of the individual programs, SPECrate runs many copies of the program simultaneously. Thus, it measures task-level parallelism, as there is no

Benchmark	Scaling?	Reprogram?	Description
Linpack	Weak	Yes	Dense matrix linear algebra [Dongarra, 1979]
SPECrate	Weak	No	Independent job parallelism [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Strong (although offers two problem sizes)	No	Complex 1D FFT Blocked LU Decomposition Blocked Sparse Cholesky Factorization Integer Radix Sort Barnes-Hut Adaptive Fast Multipole Ocean Simulation Hierarchical Radiosity Ray Tracer Volume Renderer Water Simulation with Spatial Data Structure Water Simulation without Spatial Data Structure
NAS Parallel Benchmarks [Bailey et al., 1991]	Weak	Yes (C or Fortran only)	EP: embarrassingly parallel MG: simplified multigrid CG: unstructured grid for a conjugate gradient method FT: 3-D partial differential equation solution using FFTs IS: large integer sort
PARSEC Benchmark Suite [Bienia et al., 2008]	Weak	No	Blackscholes—Option pricing with Black-Scholes PDE Bodytrack—Body tracking of a person Canneal—Simulated cache-aware annealing to optimize routing Dedup—Next-generation compression with data deduplication Facesim—Simulates the motions of a human face Ferret—Content similarity search server Fluidanimate—Fluid dynamics for animation with SPH method Freqmine—Frequent itemset mining Streamcluster—Online clustering of an input stream Swaptions—Pricing of a portfolio of swaptions Vips—Image processing x264—H.264 video encoding
Berkeley Design Patterns [Asanovic et al., 2006]	Strong or Weak	Yes	Finite-State Machine Combinational Logic Graph Traversal Structured Grid Dense Matrix Sparse Matrix Spectral Methods (FFT) Dynamic Programming N-Body MapReduce Backtrack/Branch and Bound Graphical Model Inference Unstructured Grid

**FIGURE 6.16 Examples of parallel benchmarks.**

communication between the tasks. You can run as many copies of the programs as you want, so this is again a form of weak scaling.

- *SPLASH* and *SPLASH 2* (Stanford Parallel Applications for Shared Memory) were efforts by researchers at Stanford University in the 1990s to put together a parallel benchmark suite similar in goals to the SPEC CPU benchmark suite. It includes both kernels and applications, including many from the high-performance computing community. This benchmark requires strong scaling, although it comes with two data sets.

**Pthreads** A UNIX API for creating and manipulating threads. It is structured as a library.

- The NAS (*NASA Advanced Supercomputing*) *parallel benchmarks* were another attempt from the 1990s to benchmark multiprocessors. Taken from computational fluid dynamics, they consist of five kernels. They allow weak scaling by defining a few data sets. Like Linpack, these benchmarks can be rewritten, but the rules require that the programming language can only be C or Fortran.
- The recent PARSEC (*Princeton Application Repository for Shared Memory Computers*) *benchmark suite* consists of multithreaded programs that use **Pthreads** (POSIX threads) and OpenMP (Open MultiProcessing; see Section 6.5). They focus on emerging computational domains and consist of nine applications and three kernels. Eight rely on data parallelism, three rely on pipelined parallelism, and one on unstructured parallelism.
- On the cloud front, the goal of the *Yahoo! Cloud Serving Benchmark* (YCSB) is to compare performance of cloud data services. It offers a framework that makes it easy for a client to benchmark new data services, using Cassandra and HBase as representative examples. [Cooper, 2010]

The downside of such traditional restrictions to benchmarks is that innovation is chiefly limited to the architecture and compiler. Better data structures, algorithms, programming languages, and so on often cannot be used, since that would give a misleading result. The system could win because of, say, the algorithm, and not because of the hardware or the compiler.

While these guidelines are understandable when the foundations of computing are relatively stable—as they were in the 1990s and the first half of this decade—they are undesirable during a programming revolution. For this revolution to succeed, we need to encourage innovation at all levels.

Researchers at the University of California at Berkeley have advocated one approach. They identified 13 design patterns that they claim will be part of applications of the future. Frameworks or kernels implement these design patterns. Examples are sparse matrices, structured grids, finite-state machines, map reduce, and graph traversal. By keeping the definitions at a high level, they hope to encourage innovations at any level of the system. Thus, the system with the fastest sparse matrix solver is welcome to use any data structure, algorithm, and programming language, in addition to novel architectures and compilers.

## Performance Models

A topic related to benchmarks is performance models. As we have seen with the increasing architectural diversity in this chapter—multithreading, SIMD, GPUs—it would be especially helpful if we had a simple model that offered insights into the performance of different architectures. It need not be perfect, just insightful.

The 3Cs for cache performance from Chapter 5 is an example performance model. It is not a perfect performance model, since it ignores potentially important

factors like block size, block allocation policy, and block replacement policy. Moreover, it has quirks. For example, a miss can be ascribed due to capacity in one design and to a conflict miss in another cache of the same size. Yet 3Cs model has been popular for 25 years, because it offers insight into the behavior of programs, helping both architects and programmers improve their creations based on insights from that model.

To find such a model for parallel computers, let's start with small kernels, like those from the 13 Berkeley design patterns in [Figure 6.16](#). While there are versions with different data types for these kernels, floating point is popular in several implementations. Hence, peak floating-point performance is a limit on the speed of such kernels on a given computer. For multicore chips, peak floating-point performance is the collective peak performance of all the cores on the chip. If there were multiple microprocessors in the system, you would multiply the peak per chip by the total number of chips.

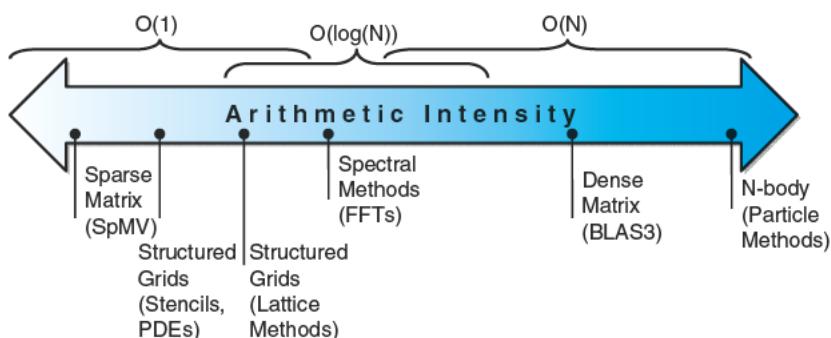
The demands on the memory system can be estimated by dividing this peak floating-point performance by the average number of floating-point operations per byte accessed:

$$\frac{\text{Floating-Point Operations/Sec}}{\text{Floating-Point Operations/Byte}} = \text{Bytes/Sec}$$

The ratio of floating-point operations per byte of memory accessed is called the **arithmetic intensity**. It can be calculated by taking the total number of floating-point operations for a program divided by the total number of data bytes transferred to main memory during program execution. [Figure 6.17](#) shows the arithmetic intensity of several of the Berkeley design patterns from [Figure 6.16](#).

#### arithmetic intensity

The ratio of floating-point operations in a program to the number of data bytes accessed by a program from main memory.



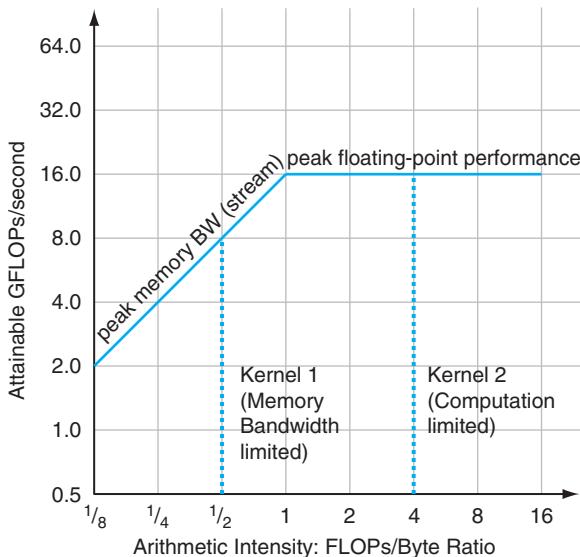
**FIGURE 6.17 Arithmetic Intensity, specified as the number of float-point operations to run the program divided by the number of bytes accessed in main memory [Williams, Waterman, and Patterson 2009].** Some kernels have an arithmetic intensity that scales with problem size, such as Dense Matrix, but there are many kernels with arithmetic intensities independent of problem size. For kernels in this former case, weak scaling can lead to different results, since it puts much less demand on the memory system.

## The Roofline Model

This simple model ties floating-point performance, arithmetic intensity, and memory performance together in a two-dimensional graph [Williams, Waterman, and Patterson 2009]. Peak floating-point performance can be found using the hardware specifications mentioned above. The working sets of the kernels we consider here do not fit in on-chip caches, so peak memory performance may be defined by the memory system behind the caches. One way to find the peak memory performance is the Stream benchmark. (See the *Elaboration* on page 381 in Chapter 5).

Figure 6.18 shows the model, which is done once for a computer, not for each kernel. The vertical Y-axis is achievable floating-point performance from 0.5 to 64.0 GFLOPs/second. The horizontal X-axis is arithmetic intensity, varying from 1/8 FLOPs/DRAM byte accessed to 16 FLOPs/DRAM byte accessed. Note that the graph is a log-log scale.

For a given kernel, we can find a point on the X-axis based on its arithmetic intensity. If we draw a vertical line through that point, the performance of the kernel on that computer must lie somewhere along that line. We can plot a horizontal line showing peak floating-point performance of the computer. Obviously, the actual floating-point performance can be no higher than the horizontal line, since that is a hardware limit.



**FIGURE 6.18 Roofline Model [Williams, Waterman, and Patterson 2009].** This example has a peak floating-point performance of 16 GFLOPS/sec and a peak memory bandwidth of 16 GB/sec from the Stream benchmark. (Since Stream is actually four measurements, this line is the average of the four.) The dotted vertical line in color on the left represents Kernel 1, which has an arithmetic intensity of 0.5 FLOPs/byte. It is limited by memory bandwidth to no more than 8 GFLOPS/sec on this Opteron X2. The dotted vertical line to the right represents Kernel 2, which has an arithmetic intensity of 4 FLOPs/byte. It is limited only computationally to 16 GFLOPS/s. (This data is based on the AMD Opteron X2 (Revision F) using dual cores running at 2 GHz in a dual socket system.)

How could we plot the peak memory performance, which is measured in bytes/second? Since the X-axis is FLOPs/byte and the Y-axis FLOPs/second, bytes/second is just a diagonal line at a 45-degree angle in this figure. Hence, we can plot a third line that gives the maximum floating-point performance that the memory system of that computer can support for a given arithmetic intensity. We can express the limits as a formula to plot the line in the graph in [Figure 6.18](#):

$$\text{Attainable GFLOPs/sec} = \text{Min}(\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{Peak Floating-Point Performance})$$

The horizontal and diagonal lines give this simple model its name and indicate its value. The “roofline” sets an upper bound on performance of a kernel depending on its arithmetic intensity. Given a roofline of a computer, you can apply it repeatedly, since it doesn’t vary by kernel.

If we think of arithmetic intensity as a pole that hits the roof, either it hits the slanted part of the roof, which means performance is ultimately limited by memory bandwidth, or it hits the flat part of the roof, which means performance is computationally limited. In [Figure 6.18](#), kernel 1 is an example of the former, and kernel 2 is an example of the latter.

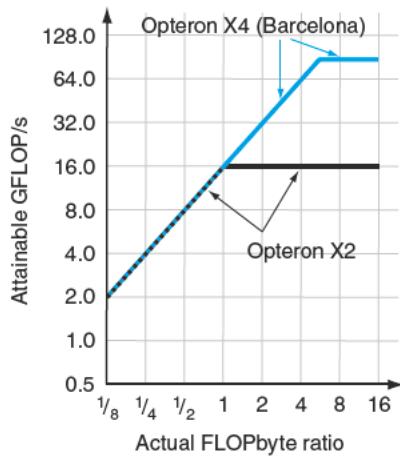
Note that the “ridge point,” where the diagonal and horizontal roofs meet, offers an interesting insight into the computer. If it is far to the right, then only kernels with very high arithmetic intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance.

## Comparing Two Generations of Opterons

The AMD Opteron X4 (Barcelona) with four cores is the successor to the Opteron X2 with two cores. To simplify board design, they use the same socket. Hence, they have the same DRAM channels and thus the same peak memory bandwidth. In addition to doubling the number of cores, the Opteron X4 also has twice the peak floating-point performance per core: Opteron X4 cores can issue two floating-point SSE2 instructions per clock cycle, while Opteron X2 cores issue at most one. As the two systems we’re comparing have similar clock rates—2.2 GHz for Opteron X2 versus 2.3 GHz for Opteron X4—the Opteron X4 has about four times the peak floating-point performance of the Opteron X2 with the same DRAM bandwidth. The Opteron X4 also has a 2MiB L3 cache, which is not found in the Opteron X2.

In [Figure 6.19](#) the roofline models for both systems are compared. As we would expect, the ridge point moves to the right, from 1 in the Opteron X2 to 5 in the Opteron X4. Hence, to see a performance gain in the next generation, kernels need an arithmetic intensity higher than 1, or their working sets must fit in the caches of the Opteron X4.

The roofline model gives an upper bound to performance. Suppose your program is far below that bound. What optimizations should you perform, and in what order?



**FIGURE 6.19 Roofline models of two generations of Opterons.** The Opteron X2 roofline, which is the same as in Figure 6.18, is in black, and the Opteron X4 roofline is in color. The bigger ridge point of Opteron X4 means that kernels that were computationally bound on the Opteron X2 could be memory-performance bound on the Opteron X4.

To reduce computational bottlenecks, the following two optimizations can help almost any kernel:

1. *Floating-point operation mix.* Peak floating-point performance for a computer typically requires an equal number of nearly simultaneous additions and multiplications. That balance is necessary either because the computer supports a fused multiply-add instruction (see the *Elaboration* on page 220 in Chapter 3) or because the floating-point unit has an equal number of floating-point adders and floating-point multipliers. The best performance also requires that a significant fraction of the instruction mix is floating-point operations and not integer instructions.
2. *Improve instruction-level parallelism and apply SIMD.* For modern architectures, the highest performance comes when fetching, executing, and committing three to four instructions per clock cycle (see Section 4.10). The goal for this step is to improve the code from the compiler to increase ILP. One way is by unrolling loops, as we saw in Section 4.12. For the x86 architectures, a single AVX instruction can operate on four double precision operands, so they should be used whenever possible (see Sections 3.7 and 3.8).

To reduce memory bottlenecks, the following two optimizations can help:

1. *Software prefetching.* Usually the highest performance requires keeping many memory operations in flight, which is easier to do by performing **predicting** accesses via software prefetch instructions rather than waiting until the data is required by the computation.



PARALLELISM



PREDICTION

2. *Memory affinity.* Microprocessors today include a memory controller on the same chip with the microprocessor, which improves performance of the **memory hierarchy**. If the system has multiple chips, this means that some addresses go to the DRAM that is local to one chip, and the rest require accesses over the chip interconnect to access the DRAM that is local to another chip. This split results in non-uniform memory accesses, which we described in Section 6.5. Accessing memory through another chip lowers performance. This second optimization tries to allocate data and the threads tasked to operate on that data to the same memory-processor pair, so that the processors rarely have to access the memory of the other chips.



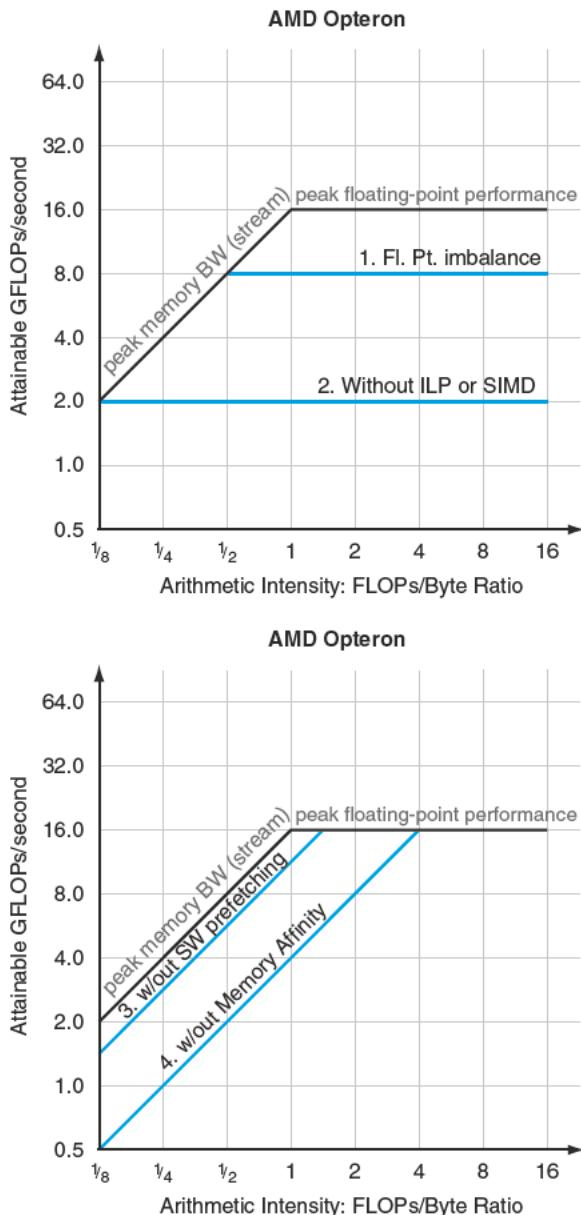
The roofline model can help decide which of these two optimizations to perform and the order in which to perform them. We can think of each of these optimizations as a “ceiling” below the appropriate roofline, meaning that you cannot break through a ceiling without performing the associated optimization.

The computational roofline can be found from the manuals, and the memory roofline can be found from running the Stream benchmark. The computational ceilings, such as floating-point balance, can also come from the manuals for that computer. A memory ceiling, such as memory affinity, requires running experiments on each computer to determine the gap between them. The good news is that this process only need be done once per computer, for once someone characterizes a computer’s ceilings, everyone can use the results to prioritize their optimizations for that computer.

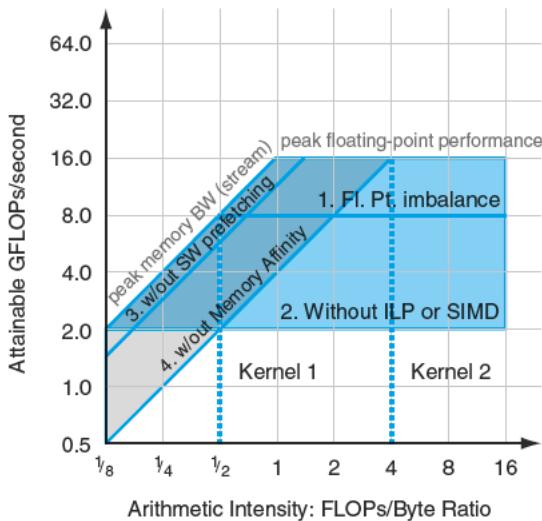
[Figure 6.20](#) adds ceilings to the roofline model in [Figure 6.18](#), showing the computational ceilings in the top graph and the memory bandwidth ceilings on the bottom graph. Although the higher ceilings are not labeled with both optimizations, they are implied in this figure; to break through the highest ceiling, you need to have already broken through all the ones below.

The width of the gap between the ceiling and the next higher limit is the reward for trying that optimization. Thus, [Figure 6.20](#) suggests that optimization 2, which improves ILP, has a large benefit for improving computation on that computer, and optimization 4, which improves memory affinity, has a large benefit for improving memory bandwidth on that computer.

[Figure 6.21](#) combines the ceilings of [Figure 6.20](#) into a single graph. The arithmetic intensity of a kernel determines the optimization region, which in turn suggests which optimizations to try. Note that the computational optimizations and the memory bandwidth optimizations overlap for much of the arithmetic intensity. Three regions are shaded differently in [Figure 6.21](#) to indicate the different optimization strategies. For example, Kernel 2 falls in the blue trapezoid on the right, which suggests working only on the computational optimizations. Kernel 1 falls in the blue-gray parallelogram in the middle, which suggests trying both types of optimizations. Moreover, it suggests starting with optimizations 2 and 4. Note that the Kernel 1 vertical lines fall below the floating-point imbalance optimization, so optimization 1 may be unnecessary. If a kernel fell in the gray triangle on the lower left, it would suggest trying just memory optimizations.



**FIGURE 6.20 Roofline model with ceilings.** The top graph shows the computational “ceilings” of 8 GFLOPs/sec if the floating-point operation mix is imbalanced and 2 GFLOPs/sec if the optimizations to increase ILP and SIMD are also missing. The bottom graph shows the memory bandwidth ceilings of 11 GB/sec without software prefetching and 4.8 GB/sec if memory affinity optimizations are also missing.



**FIGURE 6.21 Roofline model with ceilings, overlapping areas shaded, and the two kernels from Figure 6.18.** Kernels whose arithmetic intensity land in the blue trapezoid on the right should focus on computation optimizations, and kernels whose arithmetic intensity land in the gray triangle in the lower left should focus on memory bandwidth optimizations. Those that land in the blue-gray parallelogram in the middle need to worry about both. As Kernel 1 falls in the parallelogram in the middle, try optimizing ILP and SIMD, memory affinity, and software prefetching. Kernel 2 falls in the trapezoid on the right, so try optimizing ILP and SIMD and the balance of floating-point operations.

Thus far, we have been assuming that the arithmetic intensity is fixed, but that is not really the case. First, there are kernels where the arithmetic intensity increases with problem size, such as for Dense Matrix and N-body problems (see Figure 6.17). Indeed, this can be a reason that programmers have more success with weak scaling than with strong scaling. Second, the effectiveness of the **memory hierarchy** affects the number of accesses that go to memory, so optimizations that improve cache performance also improve arithmetic intensity. One example is improving temporal locality by unrolling loops and then grouping together statements with similar addresses. Many computers have special cache instructions that allocate data in a cache but do not first fill the data from memory at that address, since it will soon be over-written. Both these optimizations reduce memory traffic, thereby moving the arithmetic intensity pole to the right by a factor of, say, 1.5. This shift right could put the kernel in a different optimization region.

While the examples above show how to help programmers improve performance, architects can also use the model to decide where they should optimize hardware to improve performance of the kernels that they think will be important.

The next section uses the roofline model to demonstrate the performance difference between a multicore microprocessor and a GPU and to see whether these differences reflect performance of real programs.



**Elaboration:** The ceilings are ordered so that lower ceilings are easier to optimize. Clearly, a programmer can optimize in any order, but following this sequence reduces the chances of wasting effort on an optimization that has no benefit due to other constraints. Like the 3Cs model, as long as the roofline model delivers on insights, a model can have assumptions that may prove optimistic. For example, roofline assumes the load is balanced between all processors.

**Elaboration:** An alternative to the Stream benchmark is to use the raw DRAM bandwidth as the roofline. While the raw bandwidth definitely is a hard upper bound, actual memory performance is often so far from that boundary that it's not that useful. That is, no program can go close to that bound. The downside to using Stream is that very careful programming may exceed the Stream results, so the memory roofline may not be as hard a limit as the computational roofline. We stick with Stream because few programmers will be able to deliver more memory bandwidth than Stream discovers.

**Elaboration:** Although the roofline model shown is for multicore processors, it clearly would work for a uniprocessor as well.

### Check Yourself

True or false: The main drawback with conventional approaches to benchmarks for parallel computers is that the rules that ensure fairness also slow software innovation.

## 6.11

### Real Stuff: Benchmarking and Rooflines of the Intel Core i7 960 and the NVIDIA Tesla GPU

A group of Intel researchers published a paper [Lee et al., 2010] comparing a quad-core Intel Core i7 960 with multimedia SIMD extensions to the previous generation GPU, the NVIDIA Tesla GTX 280. [Figure 6.22](#) lists the characteristics of the two systems. Both products were purchased in Fall 2009. The Core i7 is in Intel's 45-nanometer semiconductor technology while the GPU is in TSMC's 65-nanometer technology. Although it might have been fairer to have a comparison by a neutral party or by both interested parties, the purpose of this section is *not* to determine how much faster one product is than another, but to try to understand the relative value of features of these two contrasting architecture styles.

The rooflines of the Core i7 960 and GTX 280 in [Figure 6.23](#) illustrate the differences in the computers. Not only does the GTX 280 have much higher memory bandwidth and double-precision floating-point performance, but also its double-precision ridge point is considerably to the left. The double-precision ridge point is 0.6 for the GTX 280 versus 3.1 for the Core i7. As mentioned above, it is much easier to hit peak computational performance the further the ridge point of

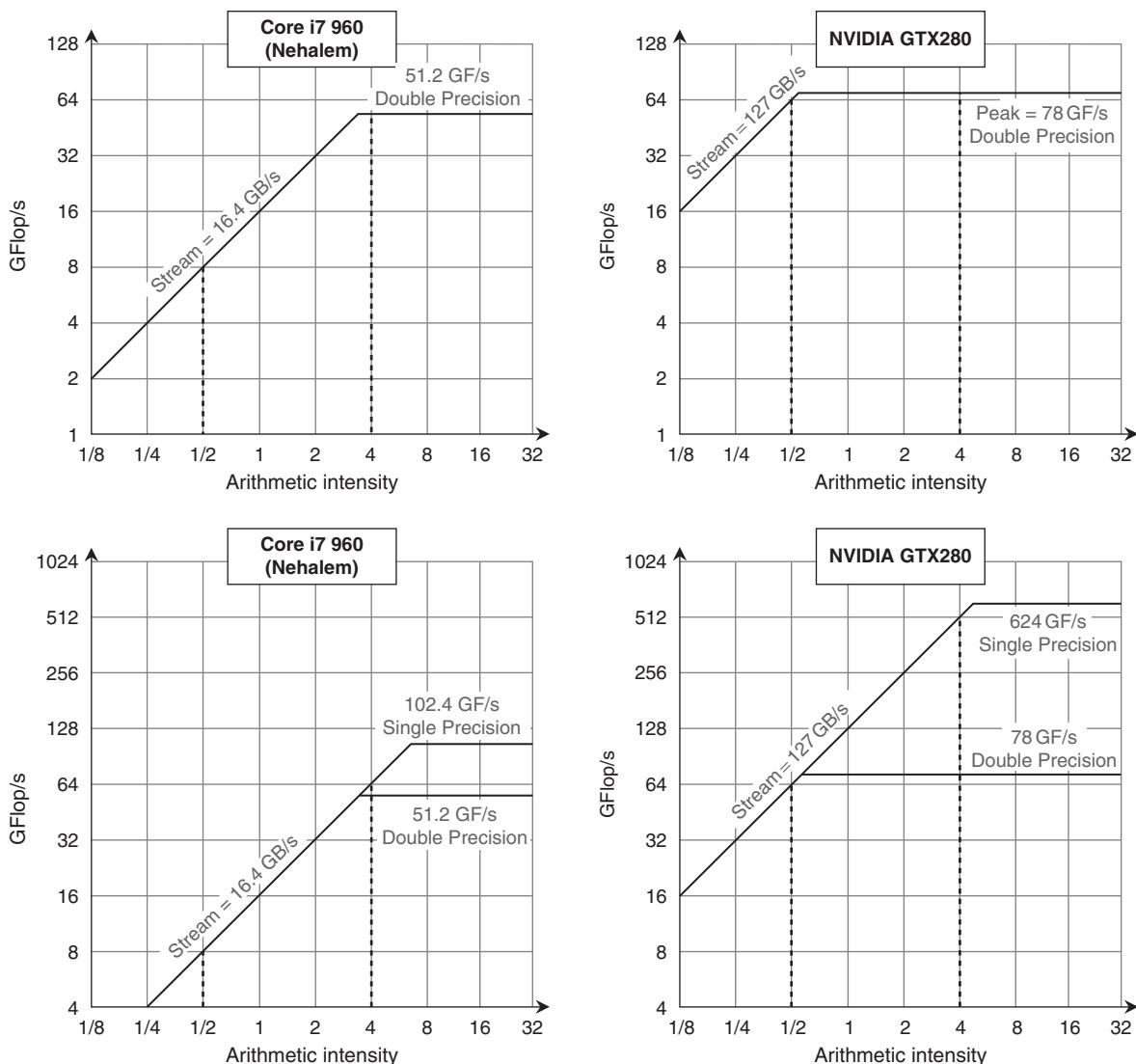
	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3030 M	2.0	4.4
Memory bandwidth (GBytes/sec)	32	141	177	4.4	5.5
Single-precision SIMD width	4	8	32	2.0	8.0
Double-precision SIMD width	2	1	16	0.5	8.0
Peak Single-precision scalar FLOPS (GFLOP/sec)	26	117	63	4.6	2.5
Peak Single-precision SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 or 1344	3.0–9.1	6.6–13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(1344)	(6.1)	(13.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	N.A.	(9.1)	–
Peak double-precision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

**FIGURE 6.22 Intel Core i7-960, NVIDIA GTX 280, and GTX 480 specifications.** The rightmost columns show the ratios of the Tesla GTX 280 and the Fermi GTX 480 to Core i7. Although the case study is between the Tesla 280 and i7, we include the Fermi 480 to show its relationship to the Tesla 280 since it is described in this chapter. Note that these memory bandwidths are higher than in Figure 6.23 because these are DRAM pin bandwidths and those in Figure 6.23 are at the processors as measured by a benchmark program. (From Table 2 in Lee et al. [2010].)

the roofline is to the left. For single-precision performance, the ridge point moves far to the right for both computers, so it's much harder to hit the roof of single-precision performance. Note that the arithmetic intensity of the kernel is based on the bytes that go to main memory, not the bytes that go to cache memory. Thus, as mentioned above, caching can change the arithmetic intensity of a kernel on a particular computer, if most references really go to the cache. Note also that this bandwidth is for unit-stride accesses in both architectures. Real gather-scatter addresses can be slower on the GTX 280 and on the Core i7, as we shall see.

The researchers selected the benchmark programs by analyzing the computational and memory characteristics of four recently proposed benchmark suites and then “formulated the set of *throughput computing kernels* that capture these characteristics.” Figure 6.24 shows the performance results, with larger numbers meaning faster. The Rooflines help explain the relative performance in this case study.

Given that the raw performance specifications of the GTX 280 vary from  $2.5 \times$  slower (clock rate) to  $7.5 \times$  faster (cores per chip) while the performance varies



**FIGURE 6.23** Roofline model [Williams, Waterman, and Patterson 2009]. These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 960 on the left has a peak DP FP performance of 51.2 GFLOP/sec, a SP FP peak of 102.4 GFLOP/sec, and a peak memory bandwidth of 16.4 GBytes/sec. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOP/sec, SP FP peak of 624 GFLOP/sec, and 127 GBytes/sec of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOP/sec or 8 SP GFLOP/sec on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 51.2 DP GFLOP/sec and 102.4 SP GFLOP/sec on the Core i7 and 78 DP GFLOP/sec and 512 DP GFLOP/sec on the GTX 280. To hit the highest computation rate on the Core i7 you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD processors.

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

**FIGURE 6.24 Raw and relative performance measured for the two platforms.** In this study, SAXPY is just used as a measure of memory bandwidth, so the right unit is GBytes/sec and not GFLOP/sec. (Based on Table 3 in [Lee et al., 2010].)

from  $2.0 \times$  slower (Solv) to  $15.2 \times$  faster (GJK), the Intel researchers decided to find the reasons for the differences:

- *Memory bandwidth.* The GPU has  $4.4 \times$  the memory bandwidth, which helps explain why LBM and SAXPY run  $5.0$  and  $5.3 \times$  faster; their working sets are hundreds of megabytes and hence don't fit into the Core i7 cache. (So as to access memory intensively, they purposely did not use cache blocking as in Chapter 5.) Hence, the slope of the rooflines explains their performance. SpMV also has a large working set, but it only runs  $1.9 \times$  faster because the double-precision floating point of the GTX 280 is only  $1.5 \times$  as fast as the Core i7.
- *Compute bandwidth.* Five of the remaining kernels are compute bound: SGEMM, Conv, FFT, MC, and Bilat. The GTX is faster by  $3.9$ ,  $2.8$ ,  $3.0$ ,  $1.8$ , and  $5.7 \times$ , respectively. The first three of these use single-precision floating-point arithmetic, and GTX 280 single precision is  $3$  to  $6 \times$  faster. MC uses double precision, which explains why it's only  $1.8 \times$  faster since DP performance is only  $1.5 \times$  faster. Bilat uses transcendental functions, which the GTX 280 supports directly. The Core i7 spends two-thirds of its time calculating transcendental functions for Bilat, so the GTX 280 is  $5.7 \times$  faster. This observation helps point out the value of hardware support for operations that occur in your workload: double-precision floating point and perhaps even transcendentals.

- *Cache benefits.* *Ray casting* (RC) is only  $1.6 \times$  faster on the GTX because cache blocking with the Core i7 caches prevents it from becoming memory bandwidth bound (see Sections 5.4 and 5.14), as it is on GPUs. Cache blocking can help Search, too. If the index trees are small so that they fit in the cache, the Core i7 is twice as fast. Larger index trees make them memory bandwidth bound. Overall, the GTX 280 runs search  $1.8 \times$  faster. Cache blocking also helps Sort. While most programmers wouldn't run Sort on a SIMD processor, it can be written with a 1-bit Sort primitive called *split*. However, the split algorithm executes many more instructions than a scalar sort does. As a result, the Core i7 runs  $1.25 \times$  as fast as the GTX 280. Note that caches also help other kernels on the Core i7, since cache blocking allows SGEMM, FFT, and SpMV to become compute bound. This observation re-emphasizes the importance of cache blocking optimizations in Chapter 5.
- *Gather-Scatter.* The multimedia SIMD extensions are of little help if the data are scattered throughout main memory; optimal performance comes only when accesses are to data aligned on 16-byte boundaries. Thus, GJK gets little benefit from SIMD on the Core i7. As mentioned above, GPUs offer gather-scatter addressing that is found in a vector architecture but omitted from most SIMD extensions. The memory controller even batches accesses to the same DRAM page together (see Section 5.2). This combination means the GTX 280 runs GJK a startling  $15.2 \times$  as fast as the Core i7, which is larger than any single physical parameter in [Figure 6.22](#). This observation reinforces the importance of gather-scatter to vector and GPU architectures that is missing from SIMD extensions.
- *Synchronization.* The performance of synchronization is limited by atomic updates, which are responsible for 28% of the total runtime on the Core i7 despite its having a hardware fetch-and-increment instruction. Thus, Hist is only  $1.7 \times$  faster on the GTX 280. Solv solves a batch of independent constraints in a small amount of computation followed by barrier synchronization. The Core i7 benefits from the atomic instructions and a memory consistency model that ensures the right results even if not all previous accesses to memory hierarchy have completed. Without the memory consistency model, the GTX 280 version launches some batches from the system processor, which leads to the GTX 280 running  $0.5 \times$  as fast as the Core i7. This observation points out how synchronization performance can be important for some data parallel problems.

It is striking how often weaknesses in the Tesla GTX 280 that were uncovered by kernels selected by Intel researchers were already being addressed in the successor architecture to Tesla: Fermi has faster double-precision floating-point performance, faster atomic operations, and caches. It was also interesting that the gather-scatter support of vector architectures that predate the SIMD instructions by decades was so important to the effective usefulness of these SIMD extensions, which some had predicted before the comparison. The Intel researchers noted that 6 of the 14 kernels would exploit SIMD better with more efficient gather-scatter support on the Core i7. This study certainly establishes the importance of cache blocking as well.

Now that we seen a wide range of results of benchmarking different multiprocessors, let's return to our DGEMM example to see in detail how much we have to change the C code to exploit multiple processors.

## 6.12

### Going Faster: Multiple Processors and Matrix Multiply

This section is the final and largest step in our incremental performance journey of adapting DGEMM to the underlying hardware of the Intel Core i7 (Sandy Bridge). Each Core i7 has 8 cores, and the computer we have been using has 2 Core i7s. Thus, we have 16 cores on which to run DGEMM.

Figure 6.25 shows the OpenMP version of DGEMM that utilizes those cores. Note that line 30 is the *single* line added to Figure 5.48 to make this code run on multiple processors: an OpenMP pragma that tells the compiler to use multiple threads in the outermost for loop. It tells the computer to spread the work of the outermost loop across all the threads.

Figure 6.26 plots a classic multiprocessor speedup graph, showing the performance improvement versus a single thread as the number of threads increase. This graph makes it easy to see the challenges of strong scaling versus weak scaling. When everything fits in the first level data cache, as is the case for  $32 \times 32$  matrices, adding threads actually hurts performance. The 16-threaded version of DGEMM is almost half as fast as the single-threaded version in this case. In contrast, the two largest matrices get a  $14 \times$  speedup from 16 threads, and hence the classic two “up and to the right” lines in Figure 6.26.

Figure 6.27 shows the absolute performance increase as we increase the number of threads from 1 to 16. DGEMM now operates at 174 GLOPS for  $960 \times 960$  matrices. As our unoptimized C version of DGEMM in Figure 3.21 ran this code at just 0.8 GFOPS, the optimizations in Chapters 3 to 6 that tailor the code to the underlying hardware result in a speedup of over 200 times!

Next up is our warnings of the fallacies and pitfalls of multiprocessing. The computer architecture graveyard is filled with parallel processing projects that have ignored them.

**Elaboration:** These results are with Turbo mode turned off. We are using a dual chip system in this system, so not surprisingly, we can get the full Turbo speedup ( $3.3/2.6 = 1.27$ ) with either 1 thread (only 1 core on one of the chips) or 2 threads (1 core per chip). As we increase the number of threads and hence the number of active cores, the benefit of Turbo mode decreases, as there is less of the power budget to spend on the active cores. For 4 threads the average Turbo speedup is 1.23, for 8 it is 1.13, and for 16 it is 1.11.

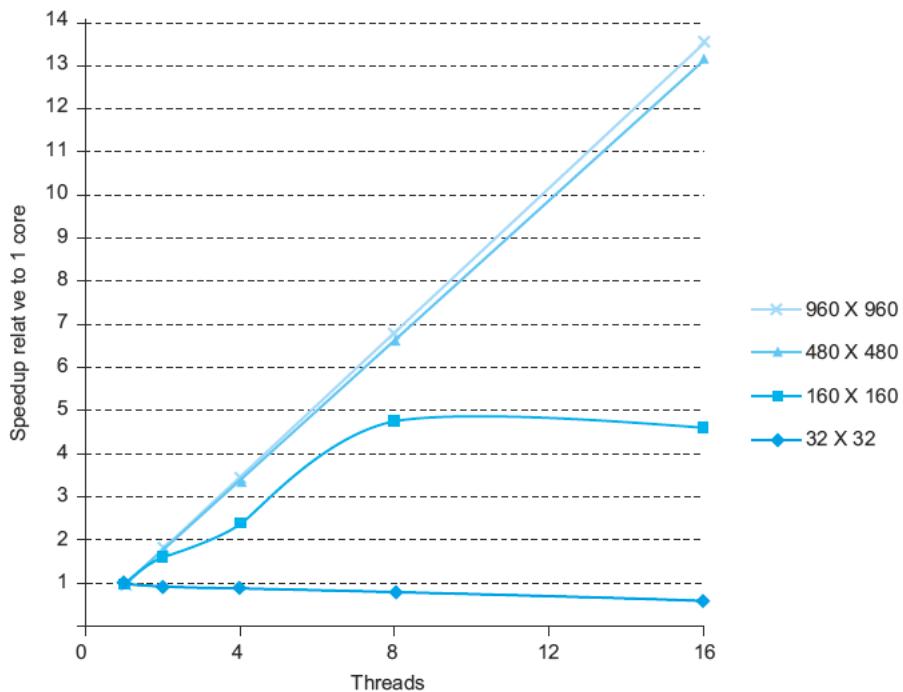
```

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            for ( int x = 0; x < UNROLL; x++ )
23                _mm256_store_pd(C+i+x*4+j*n, c[x]);
24            /* C[i][j] = c[x] */
25        }
26    }
27
28 void dgemm (int n, double* A, double* B, double* C)
29 {
30 #pragma omp parallel for
31     for ( int sj = 0; sj < n; sj += BLOCKSIZE )
32         for ( int si = 0; si < n; si += BLOCKSIZE )
33             for ( int sk = 0; sk < n; sk += BLOCKSIZE )
34                 do_block(n, si, sj, sk, A, B, C);
35 }
```

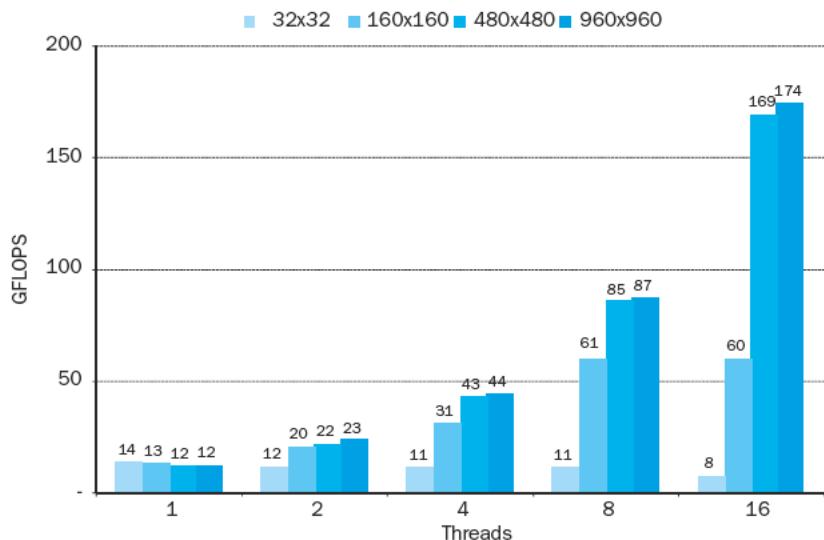
**FIGURE 6.25 OpenMP version of DGEMM from Figure 5.48.** Line 30 is the only OpenMP code, making the outermost for loop operate in parallel. This line is the only difference from Figure 5.48.

---

**Elaboration:** Although the Sandy Bridge supports two hardware threads per core, we do not get more performance from 32 threads. The reason is that a single AVX hardware is shared between the two threads multiplexed onto one core, so assigning two threads per core actually hurts performance due to the multiplexing overhead.



**FIGURE 6.26 Performance Improvements relative to a single thread as the number of threads Increase.** The most honest way to present such graphs is to make performance relative to the best version of a single processor program, which we did. This plot is relative to the performance of the code in Figure 5.48 *without* including OpenMP pragmas.



**FIGURE 6.27 DGEMM performance versus the number of threads for four matrix sizes.** The performance improvement compared unoptimized code in Figure 3.21 for the 960 × 960 matrix with 16 threads is an astounding 212 times faster!

## 6.13

### Fallacies and Pitfalls

*For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. ... Demonstration is made of the continued validity of the single processor approach ...*

Gene Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," Spring Joint Computer Conference, 1967

The many assaults on parallel processing have uncovered numerous fallacies and pitfalls. We cover four here.

*Fallacy: Amdahl's Law doesn't apply to parallel computers.*

In 1987, the head of a research organization claimed that a multiprocessor machine had broken Amdahl's Law. To try to understand the basis of the media reports, let's see the quote that gave us Amdahl's Law [1967, p. 483]:

*A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.*

This statement must still be true; the neglected portion of the program must limit performance. One interpretation of the law leads to the following lemma: portions of every program must be sequential, so there must be an economic upper bound to the number of processors—say, 100. By showing linear speed-up with 1000 processors, this lemma is disproved; hence the claim that Amdahl's Law was broken.

The approach of the researchers was just to use weak scaling: rather than going 1000 times faster on the same data set, they computed 1000 times more work in comparable time. For their algorithm, the sequential portion of the program was constant, independent of the size of the input, and the rest was fully parallel—hence, linear speed-up with 1000 processors.

Amdahl's Law obviously applies to parallel processors. What this research does point out is that one of the main uses of faster computers is to run larger problems. Just be sure that users really care about those problems versus being a justification to buying an expensive computer by finding a problem that just keeps lots of processors busy.

*Fallacy: Peak performance tracks observed performance.*

The supercomputer industry once used this metric in marketing, and the fallacy is exacerbated with parallel machines. Not only are marketers using the nearly unattainable peak performance of a uniprocessor node, but also they are then multiplying it by the total number of processors, assuming perfect speed-up! Amdahl's Law suggests how difficult it is to reach either peak; multiplying the two together multiplies the sins. The roofline model helps put peak performance in perspective.

*Pitfall: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.*

There is a long history of parallel software lagging behind on parallel hardware, possibly because the software problems are much harder. We give one example to show the subtlety of the issues, but there are many examples we could choose!

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the Silicon Graphics operating system originally protected the page table with a single lock, assuming that page allocation is infrequent. In a uniprocessor, this does not represent a performance problem. In a multiprocessor, it can become a major performance bottleneck for some programs. Consider a program that uses a large number of pages that are initialized at start-up, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time!).

This page table serialization eliminates parallelism in initialization and has significant impact on overall parallel performance. This performance bottleneck persists even for task-level parallelism. For example, suppose we split the parallel processing program apart into separate jobs and run them, one job per processor, so that there is no sharing between the jobs. (This is exactly what one user did, since he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the jobs—so even the independent job performance is poor.

This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminated the problem.

*Fallacy: You can get good vector performance without providing memory bandwidth.*

As we saw with the Roofline model, memory bandwidth is quite important to all architectures. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a Cray-1 could not increase the DAXPY performance of the vector sequence used, since it was memory limited. The Cray-1 performance on Linpack jumped when the compiler used blocking to change the computation so that values could be kept in the vector registers. This approach lowered the number of memory references per FLOP and improved the performance by nearly a factor of two! Thus, the memory bandwidth on the Cray-1 became sufficient for a loop that formerly required more bandwidth, which is just what the Roofline model would predict.

## 6.14

### Concluding Remarks

*We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry. ... This is not a race.*

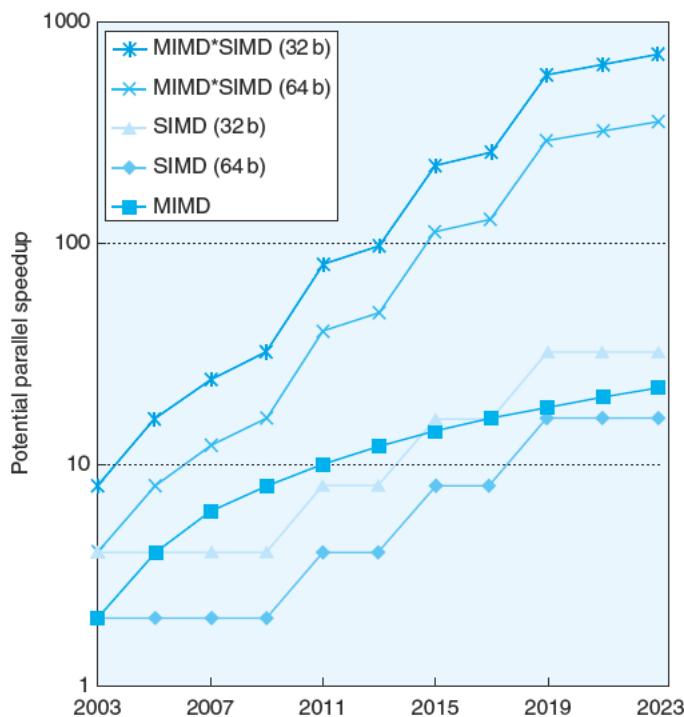
*This is a sea change in computing..."*

Paul Otellini, Intel  
President, Intel  
Developers Forum, 2004

The dream of building computers by simply aggregating processors has been around since the earliest days of computing. Progress in building and using effective and efficient parallel processors, however, has been slow. This rate of progress has been limited by difficult software problems as well as by a long process of evolving the architecture of multiprocessors to enhance usability and improve efficiency. We have discussed many of the software challenges in this chapter, including the difficulty of writing programs that obtain good speed-up due to Amdahl's Law. The wide variety of different architectural approaches and the limited success and short life of many of the parallel architectures of the past have compounded the software difficulties. We discuss the history of the development of these multiprocessors in online [Section 6.15](#). To go into even greater depth on topics in this chapter, see Chapter 4 of *Computer Architecture: A Quantitative Approach, Fifth Edition* for more on GPUs and comparisons between GPUs and CPUs and Chapter 6 for more on WSCs.

As we said in Chapter 1, despite this long and checkered past, the information technology industry has now tied its future to parallel computing. Although it is easy to make the case that this effort will fail like many in the past, there are reasons to be hopeful:

- Clearly, *software as a service* (SaaS) is growing in importance, and clusters have proven to be a very successful way to deliver such services. By providing redundancy at a higher-level, including geographically distributed datacenters, such services have delivered  $24 \times 7 \times 365$  availability for customers around the world.
- We believe that Warehouse-Scale Computers are changing the goals and principles of server design, just as the needs of mobile clients are changing the goals and principles of microprocessor design. Both are revolutionizing the software industry as well. Performance per dollar and performance per joule drive both mobile client hardware and the WSC hardware, and parallelism is the key to delivering on those sets of goals.
- SIMD and vector operations are a good match to multimedia applications, which are playing a larger role in the PostPC Era. They share the advantage of being easier for the programmer than classic parallel MIMD programming and being more energy efficient than MIMD. To put into perspective the importance of SIMD versus MIMD, [Figure 6.28](#) plots the number of cores for MIMD versus the number of 32-bit and 64-bit operations per clock cycle in SIMD mode for x86 computers over time. For x86 computers, we expect to see two additional cores per chip about every two years and the SIMD width to double about every four years. Given these assumptions, over the next decade the potential speed-up from SIMD parallelism is twice that of



**FIGURE 6.28 Potential speed-up via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers.** This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

MIMD parallelism. Given the effectiveness of SIMD for multimedia and its increasing importance in the PostPC Era, that emphasis may be appropriate. Hence, it's as least as important to understand SIMD parallelism as MIMD parallelism, even though the latter has received much more attention.

- The use of parallel processing in domains such as scientific and engineering computation is popular. This application domain has an almost limitless thirst for more computation. It also has many applications that have lots of natural concurrency. Once again, clusters dominate this application area. For example, using the 2012 Top 500 report, clusters are responsible for more than 80% of the 500 fastest Linpack results.
- All desktop and server microprocessor manufacturers are building multiprocessors to achieve higher performance, so, unlike in the past, there is no easy path to higher performance for sequential applications. As we said earlier, sequential programs are now slow programs. Hence, programmers who need higher performance *must* parallelize their codes or write new parallel processing programs.

- In the past, microprocessors and multiprocessors were subject to different definitions of success. When scaling uniprocessor performance, microprocessor architects were happy if single thread performance went up by the square root of the increased silicon area. Thus, they were happy with sublinear performance in terms of resources. Multiprocessor success used to be defined as *linear* speed-up as a function of the number of processors, assuming that the cost of purchase or cost of administration of  $n$  processors was  $n$  times as much as one processor. Now that parallelism is happening on-chip via multicore, we can use the traditional microprocessor metric of being successful with sublinear performance improvement.
- The success of just-in-time runtime compilation and autotuning makes it feasible to think of software adapting itself to take advantage of the increasing number of cores per chip, which provides flexibility that is not available when limited to static compilers.
- Unlike in the past, the open source movement has become a critical portion of the software industry. This movement is a meritocracy, where better engineering solutions can win the mind share of the developers over legacy concerns. It also embraces innovation, inviting change to old software and welcoming new languages and software products. Such an open culture could be extremely helpful in this time of rapid change.

To motivate readers to embrace this revolution, we demonstrated the potential of parallelism concretely for matrix multiply on the Intel Core i7 (Sandy Bridge) in the Going Faster sections of Chapters 3 to 6:

- Data-level parallelism in Chapter 3 improved performance by a factor of 3.85 by executing four 64-bit floating-point operations in parallel using the 256-bit operands of the AVX instructions, demonstrating the value of SIMD.
- Instruction-level parallelism in Chapter 4 pushed performance up by another factor of 2.3 by unrolling loops 4 times to give the out-of-order execution hardware more instructions to schedule.
- Cache optimizations in Chapter 5 improved performance of matrices that didn't fit into the L1 data cache by another factor of 2.0 to 2.5 by using cache blocking to reduce cache misses.
- Thread-level parallelism in this chapter improved performance of matrices that don't fit into a single L1 data cache by another factor of 4 to 14 by utilizing all 16 cores of our multicore chips, demonstrating the value of MIMD. We did this by adding a single line using an OpenMP pragma.

Using the ideas in this book and tailoring the software to this computer added 24 lines of code to DGEMM. For the matrix sizes of 32x32, 160x160, 480x480, and 960x960, the overall performance speedup from these ideas realized in those two-dozen lines of code is factors of 8, 39, 129, and 212!

This parallel revolution in the hardware/software interface is perhaps the greatest challenge facing the field in the last 60 years. You can also think of it as the greatest opportunity, as our Going Faster sections demonstrate. This revolution will provide many new research and business prospects inside and outside the IT field, and the companies that dominate the multicore era may not be the same ones that dominated the uniprocessor era. After understanding the underlying hardware trends and learning to adapt software to them, perhaps you will be one of the innovators who will seize the opportunities that are certain to appear in the uncertain times ahead. We look forward to benefiting from your inventions!



## Historical Perspective and Further Reading

This section online gives the rich and often disastrous history of multiprocessors over the last 50 years.

## References

- G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, 2004.
- B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with YCSB, In: Proceedings of the 1st ACM Symposium on Cloud computing, June 10–11, 2010, Indianapolis, Indiana, USA, doi:10.1145/1807128.1807152.

## 6.16

## Exercises

**6.1** First, write down a list of your daily activities that you typically do on a weekday. For instance, you might get out of bed, take a shower, get dressed, eat breakfast, dry your hair, brush your teeth. Make sure to break down your list so you have a minimum of 10 activities.

**6.1.1** [5] <§6.2> Now consider which of these activities is already exploiting some form of parallelism (e.g., brushing multiple teeth at the same time, versus one at a time, carrying one book at a time to school, versus loading them all into your

backpack and then carry them “in parallel”). For each of your activities, discuss if they are already working in parallel, but if not, why they are not.

**6.1.2** [5] <§6.2> Next, consider which of the activities could be carried out concurrently (e.g., eating breakfast and listening to the news). For each of your activities, describe which other activity could be paired with this activity.

**6.1.3** [5] <§6.2> For 6.1.2, what could we change about current systems (e.g., showers, clothes, TVs, cars) so that we could perform more tasks in parallel?

**6.1.4** [5] <§6.2> Estimate how much shorter time it would take to carry out these activities if you tried to carry out as many tasks in parallel as possible.

**6.2** You are trying to bake 3 blueberry pound cakes. Cake ingredients are as follows:

1 cup butter, softened  
1 cup sugar  
4 large eggs  
1 teaspoon vanilla extract  
1/2 teaspoon salt  
1/4 teaspoon nutmeg  
1 1/2 cups flour  
1 cup blueberries

The recipe for a single cake is as follows:

Step 1: Preheat oven to 325°F (160°C). Grease and flour your cake pan.

Step 2: In large bowl, beat together with a mixer butter and sugar at medium speed until light and fluffy. Add eggs, vanilla, salt and nutmeg. Beat until thoroughly blended. Reduce mixer speed to low and add flour, 1/2 cup at a time, beating just until blended.

Step 3: Gently fold in blueberries. Spread evenly in prepared baking pan. Bake for 60 minutes.

**6.2.1** [5] <§6.2> Your job is to cook 3 cakes as efficiently as possible. Assuming that you only have one oven large enough to hold one cake, one large bowl, one cake pan, and one mixer, come up with a schedule to make three cakes as quickly as possible. Identify the bottlenecks in completing this task.

**6.2.2** [5] <§6.2> Assume now that you have three bowls, 3 cake pans and 3 mixers. How much faster is the process now that you have additional resources?

**6.2.3** [5] <§6.2> Assume now that you have two friends that will help you cook, and that you have a large oven that can accommodate all three cakes. How will this change the schedule you arrived at in Exercise 6.2.1 above?

**6.2.4** [5] <§6.2> Compare the cake-making task to computing 3 iterations of a loop on a parallel computer. Identify data-level parallelism and task-level parallelism in the cake-making loop.

**6.3** Many computer applications involve searching through a set of data and sorting the data. A number of efficient searching and sorting algorithms have been devised in order to reduce the runtime of these tedious tasks. In this problem we will consider how best to parallelize these tasks.

**6.3.1** [10] <§6.2> Consider the following binary search algorithm (a classic divide and conquer algorithm) that searches for a value  $X$  in a sorted  $N$ -element array  $A$  and returns the index of matched entry:

```
BinarySearch(A[0..N-1], X) {
    low = 0
    high = N - 1
    while (low <= high) {
        mid = (low + high) / 2
        if (A[mid] > X)
            high = mid - 1
        else if (A[mid] < X)
            low = mid + 1
        else
            return mid // found
    }
    return -1 // not found
}
```

Assume that you have  $Y$  cores on a multi-core processor to run `BinarySearch`. Assuming that  $Y$  is much smaller than  $N$ , express the speedup factor you might expect to obtain for values of  $Y$  and  $N$ . Plot these on a graph.

**6.3.2** [5] <§6.2> Next, assume that  $Y$  is equal to  $N$ . How would this affect your conclusions in your previous answer? If you were tasked with obtaining the best speedup factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

**6.4** Consider the following piece of C code:

```
for (j=2;j<1000;j++)
    D[j] = D[j-1]+D[j-2];
```

The MIPS code corresponding to the above fragment is:

```

addiu    $s2,$zero,7992
addiu    $s1,$zero,16
loop:   l.d      $f0, -16($s1)
        l.d      $f2, -8($s1)
        add.d   $f4, $f0, $f2
        s.d     $f4, 0($s1)
        addiu   $s1, $s1, 8
        bne     $s1, $s2, loop

```

Instructions have the following associated latencies (in cycles):

<b>add.d</b>	<b>l.d</b>	<b>s.d</b>	<b>addiu</b>
4	6	1	2

**6.4.1** [10] <§6.2> How many cycles does it take for all instructions in a single iteration of the above loop to execute?

**6.4.2** [10] <§6.2> When an instruction in a later iteration of a loop depends upon a data value produced in an earlier iteration of the same loop, we say that there is a *loop carried dependence* between iterations of the loop. Identify the loop-carried dependences in the above code. Identify the dependent program variable and assembly-level registers. You can ignore the loop induction variable *j*.

**6.4.3** [10] <§6.2> Loop unrolling was described in Chapter 4. Apply loop unrolling to this loop and then consider running this code on a 2-node distributed memory message passing system. Assume that we are going to use message passing as described in Section 6.7, where we introduce a new operation send (*x*, *y*) that sends to node *x* the value *y*, and an operation receive( ) that waits for the value being sent to it. Assume that send operations take a cycle to issue (i.e., later instructions on the same node can proceed on the next cycle), but take 10 cycles be received on the receiving node. Receive instructions stall execution on the node where they are executed until they receive a message. Produce a schedule for the two nodes assuming an unroll factor of 4 for the loop body (i.e., the loop body will appear 4 times). Compute the number of cycles it will take for the loop to run on the message passing system.

**6.4.4** [10] <§6.2> The latency of the interconnect network plays a large role in the efficiency of message passing systems. How fast does the interconnect need to be in order to obtain any speedup from using the distributed system described in Exercise 6.4.3?

**6.5** Consider the following recursive mergesort algorithm (another classic divide and conquer algorithm). Mergesort was first described by John Von Neumann in 1945. The basic idea is to divide an unsorted list *x* of *m* elements into two sublists of about half the size of the original list. Repeat this operation on each sublist, and

continue until we have lists of size 1 in length. Then starting with sublists of length 1, “merge” the two sublists into a single sorted list.

```
Mergesort(m)
    var list left, right, result
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = Mergesort(left)
        right = Mergesort(right)
        result = Merge(left, right)
    return result
```

The merge step is carried out by the following code:

```
Merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    if length(left) > 0
        append rest(left) to result
    if length(right) > 0
        append rest(right) to result
    return result
```

**6.5.1** [10] <§6.2> Assume that you have  $Y$  cores on a multicore processor to run MergeSort. Assuming that  $Y$  is much smaller than  $\text{length}(m)$ , express the speedup factor you might expect to obtain for values of  $Y$  and  $\text{length}(m)$ . Plot these on a graph.

**6.5.2** [10] <§6.2> Next, assume that  $Y$  is equal to  $\text{length}(m)$ . How would this affect your conclusions from your previous answer? If you were tasked with obtaining the best speedup factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

**6.6** Matrix multiplication plays an important role in a number of applications. Two matrices can only be multiplied if the number of columns of the first matrix is equal to the number of rows in the second.

Let's assume we have an  $m \times n$  matrix  $A$  and we want to multiply it by an  $n \times p$  matrix  $B$ . We can express their product as an  $m \times p$  matrix denoted by  $AB$  (or  $A \cdot B$ ). If we assign  $C = AB$ , and  $c_{ij}$  denotes the entry in  $C$  at position  $(i, j)$ , then for each element  $i$  and  $j$  with  $1 \leq i \leq m$  and  $1 \leq j \leq p$ . Now we want to see if we can parallelize the computation of  $C$ . Assume that matrices are laid out in memory sequentially as follows:  $a_{1,1}, a_{2,1}, a_{3,1}, a_{4,1}, \dots$ , etc.

**6.6.1** [10] <§6.5> Assume that we are going to compute  $C$  on both a single core shared memory machine and a 4-core shared-memory machine. Compute the speedup we would expect to obtain on the 4-core machine, ignoring any memory issues.

**6.6.2** [10] <§6.5> Repeat Exercise 6.6.1, assuming that updates to  $C$  incur a cache miss due to false sharing when consecutive elements are in a row (i.e., index  $i$ ) are updated.

**6.6.3** [10] <§6.5> How would you fix the false sharing issue that can occur?

**6.7** Consider the following portions of two different programs running at the same time on four processors in a symmetric multicore processor (SMP). Assume that before this code is run, both  $x$  and  $y$  are 0.

Core 1:  $x = 2;$

Core 2:  $y = 2;$

Core 3:  $w = x + y + 1;$

Core 4:  $z = x + y;$

**6.7.1** [10] <§6.5> What are all the possible resulting values of  $w$ ,  $x$ ,  $y$ , and  $z$ ? For each possible outcome, explain how we might arrive at those values. You will need to examine all possible interleavings of instructions.

**6.7.2** [5] <§6.5> How could you make the execution more deterministic so that only one set of values is possible?

**6.8** The dining philosopher's problem is a classic problem of synchronization and concurrency. The general problem is stated as philosophers sitting at a round table doing one of two things: eating or thinking. When they are eating, they are not thinking, and when they are thinking, they are not eating. There is a bowl of pasta in the center. A fork is placed in between each philosopher. The result is that each philosopher has one fork to her left and one fork to her right. Given the nature of eating pasta, the philosopher needs two forks to eat, and can only use the forks on her immediate left and right. The philosophers do not speak to one another.

**6.8.1** [10] <§6.7> Describe the scenario where none of philosophers ever eats (i.e., starvation). What is the sequence of events that happen that lead up to this problem?

**6.8.2** [10] <§6.7> Describe how we can solve this problem by introducing the concept of a priority? But can we guarantee that we will treat all the philosophers fairly? Explain.

Now assume we hire a waiter who is in charge of assigning forks to philosophers. Nobody can pick up a fork until the waiter says they can. The waiter has global knowledge of all forks. Further, if we impose the policy that philosophers will always request to pick up their left fork before requesting to pick up their right fork, then we can guarantee to avoid deadlock.

**6.8.3** [10] <§6.7> We can implement requests to the waiter as either a queue of requests or as a periodic retry of a request. With a queue, requests are handled in the order they are received. The problem with using the queue is that we may not always be able to service the philosopher whose request is at the head of the queue (due to the unavailability of resources). Describe a scenario with 5 philosophers where a queue is provided, but service is not granted even though there are forks available for another philosopher (whose request is deeper in the queue) to eat.

**6.8.4** [10] <§6.7> If we implement requests to the waiter by periodically repeating our request until the resources become available, will this solve the problem described in Exercise 6.8.3? Explain.

**6.9** Consider the following three CPU organizations:

CPU SS: A 2-core superscalar microprocessor that provides out-of-order issue capabilities on 2 function units (FUs). Only a single thread can run on each core at a time.

CPU MT: A fine-grained multithreaded processor that allows instructions from 2 threads to be run concurrently (i.e., there are two functional units), though only instructions from a single thread can be issued on any cycle.

CPU SMT: An SMT processor that allows instructions from 2 threads to be run concurrently (i.e., there are two functional units), and instructions from either or both threads can be issued to run on any cycle.

Assume we have two threads X and Y to run on these CPUs that include the following operations:

Thread X	Thread Y
A1 – takes 3 cycles to execute	B1 – take 2 cycles to execute
A2 – no dependences	B2 – conflicts for a functional unit with B1
A3 – conflicts for a functional unit with A1	B3 – depends on the result of B2
A4 – depends on the result of A3	B4 – no dependences and takes 2 cycles to execute

Assume all instructions take a single cycle to execute unless noted otherwise or they encounter a hazard.

**6.9.1** [10] <§6.4> Assume that you have 1 SS CPU. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

**6.9.2** [10] <§6.4> Now assume you have 2 SS CPUs. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

**6.9.3** [10] <§6.4> Assume that you have 1 MT CPU. How many cycles will it take to execute these two threads? How many issue slots are wasted due to hazards?

**6.10** Virtualization software is being aggressively deployed to reduce the costs of managing today's high performance servers. Companies like VMWare, Microsoft and IBM have all developed a range of virtualization products. The general concept, described in Chapter 5, is that a hypervisor layer can be introduced between the hardware and the operating system to allow multiple operating systems to share the same physical hardware. The hypervisor layer is then responsible for allocating CPU and memory resources, as well as handling services typically handled by the operating system (e.g., I/O).

Virtualization provides an abstract view of the underlying hardware to the hosted operating system and application software. This will require us to rethink how multi-core and multiprocessor systems will be designed in the future to support the sharing of CPUs and memories by a number of operating systems concurrently.

**6.10.1** [30] <§6.4> Select two hypervisors on the market today, and compare and contrast how they virtualize and manage the underlying hardware (CPUs and memory).

**6.10.2** [15] <§6.4> Discuss what changes may be necessary in future multi-core CPU platforms in order to better match the resource demands placed on these systems. For instance, can multithreading play an effective role in alleviating the competition for computing resources?

**6.11** We would like to execute the loop below as efficiently as possible. We have two different machines, a MIMD machine and a SIMD machine.

```
for (i=0; i < 2000; i++)
    for (j=0; j<3000; j++)
        X_array[i][j] = Y_array[j][i] + 200;
```

**6.11.1** [10] <§6.3> For a 4 CPU MIMD machine, show the sequence of MIPS instructions that you would execute on each CPU. What is the speedup for this MIMD machine?

**6.11.2** [20] <§6.3> For an 8-wide SIMD machine (i.e., 8 parallel SIMD functional units), write an assembly program in using your own SIMD extensions to MIPS to execute the loop. Compare the number of instructions executed on the SIMD machine to the MIMD machine.

**6.12** A systolic array is an example of an MISD machine. A systolic array is a pipeline network or “wavefront” of data processing elements. Each of these elements does not need a program counter since execution is triggered by the arrival of data. Clocked systolic arrays compute in “lock-step” with each processor undertaking alternate compute and communication phases.

**6.12.1** [10] <§6.3> Consider proposed implementations of a systolic array (you can find these in on the Internet or in technical publications). Then attempt to program the loop provided in Exercise 6.11 using this MISD model. Discuss any difficulties you encounter.

**6.12.2** [10] <§6.3> Discuss the similarities and differences between an MISD and SIMD machine. Answer this question in terms of data-level parallelism.

**6.13** Assume we want to execute the DAXPY loop show on page 511 in MIPS assembly on the NVIDIA 8800 GTX GPU described in this chapter. In this problem, we will assume that all math operations are performed on single-precision floating-point numbers (we will rename the loop SAXPY). Assume that instructions take the following number of cycles to execute.

Loads	Stores	Add.S	Mult.S
5	2	3	4

**6.13.1** [20] <§6.6> Describe how you will constructs warps for the SAXPY loop to exploit the 8 cores provided in a single multiprocessor.

**6.14** Download the CUDA Toolkit and SDK from [http://www.nvidia.com/object/cuda\\_get.html](http://www.nvidia.com/object/cuda_get.html). Make sure to use the “emurelease” (Emulation Mode) version of the code (you will not need actual NVIDIA hardware for this assignment). Build the example programs provided in the SDK, and confirm that they run on the emulator.

**6.14.1** [90] <§6.6> Using the “template” SDK sample as a starting point, write a CUDA program to perform the following vector operations:

- 1)  $a - b$  (vector-vector subtraction)
- 2)  $a \cdot b$  (vector dot product)

The dot product of two vectors  $a = [a_1, a_2, \dots, a_n]$  and  $b = [b_1, b_2, \dots, b_n]$  is defined as:

$$a \cdot b = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n$$

Submit code for each program that demonstrates each operation and verifies the correctness of the results.

**6.14.2** [90] <§6.6> If you have GPU hardware available, complete a performance analysis your program, examining the computation time for the GPU and a CPU version of your program for a range of vector sizes. Explain any results you see.

**6.15** AMD has recently announced that they will be integrating a graphics processing unit with their x86 cores in a single package, though with different clocks for each of the cores. This is an example of a heterogeneous multiprocessor system which we expect to see produced commercially in the near future. One of the key design points will be to allow for fast data communication between the CPU and the GPU. Presently communications must be performed between discrete CPU and GPU chips. But this is changing in AMDs Fusion architecture. Presently the plan is to use multiple (at least 16) PCI express channels for facilitate intercommunication. Intel is also jumping into this arena with their Larrabee chip. Intel is considering to use their QuickPath interconnect technology.

**6.15.1** [25] <§6.6> Compare the bandwidth and latency associated with these two interconnect technologies.

**6.16** Refer to [Figure 6.14b](#), which shows an n-cube interconnect topology of order 3 that interconnects 8 nodes. One attractive feature of an n-cube interconnection network topology is its ability to sustain broken links and still provide connectivity.

**6.16.1** [10] <§6.8> Develop an equation that computes how many links in the n-cube (where n is the order of the cube) can fail and we can still guarantee an unbroken link will exist to connect any node in the n-cube.

**6.16.2** [10] <§6.8> Compare the resiliency to failure of n-cube to a fully-connected interconnection network. Plot a comparison of reliability as a function of the added number of links for the two topologies.

**6.17** Benchmarking is field of study that involves identifying representative workloads to run on specific computing platforms in order to be able to objectively compare performance of one system to another. In this exercise we will compare two classes of benchmarks: the Whetstone CPU benchmark and the PARSEC Benchmark suite. Select one program from PARSEC. All programs should be freely available on the Internet. Consider running multiple copies of Whetstone versus running the PARSEC Benchmark on any of systems described in Section 6.11.

**6.17.1** [60] <§6.10> What is inherently different between these two classes of workload when run on these multi-core systems?

**6.17.2** [60] <§6.10> In terms of the Roofline Model, how dependent will the results you obtain when running these benchmarks be on the amount of sharing and synchronization present in the workload used?

**6.18** When performing computations on sparse matrices, latency in the memory hierarchy becomes much more of a factor. Sparse matrices lack the spatial locality in the data stream typically found in matrix operations. As a result, new matrix representations have been proposed.

One the earliest sparse matrix representations is the Yale Sparse Matrix Format. It stores an initial sparse  $m \times n$  matrix,  $M$  in row form using three one-dimensional

arrays. Let  $R$  be the number of nonzero entries in  $M$ . We construct an array  $A$  of length  $R$  that contains all nonzero entries of  $M$  (in left-to-right top-to-bottom order). We also construct a second array  $IA$  of length  $m + 1$  (i.e., one entry per row, plus one).  $IA(i)$  contains the index in  $A$  of the first nonzero element of row  $i$ . Row  $i$  of the original matrix extends from  $A(IA(i))$  to  $A(IA(i+1)-1)$ . The third array,  $JA$ , contains the column index of each element of  $A$ , so it also is of length  $R$ .

**6.18.1** [15] <§6.10> Consider the sparse matrix  $X$  below and write C code that would store this code in Yale Sparse Matrix Format.

```
Row 1 [1, 2, 0, 0, 0, 0]
Row 2 [0, 0, 1, 1, 0, 0]
Row 3 [0, 0, 0, 0, 9, 0]
Row 4 [2, 0, 0, 0, 0, 2]
Row 5 [0, 0, 3, 3, 0, 7]
Row 6 [1, 3, 0, 0, 0, 1]
```

**6.18.2** [10] <§6.10> In terms of storage space, assuming that each element in matrix  $X$  is single precision floating point, compute the amount of storage used to store the Matrix above in Yale Sparse Matrix Format.

**6.18.3** [15] <§6.10> Perform matrix multiplication of Matrix X by Matrix Y shown below.

```
[2, 4, 1, 99, 7, 2]
```

Put this computation in a loop, and time its execution. Make sure to increase the number of times this loop is executed to get good resolution in your timing measurement. Compare the runtime of using a naïve representation of the matrix, and the Yale Sparse Matrix Format.

**6.18.4** [15] <§6.10> Can you find a more efficient sparse matrix representation (in terms of space and computational overhead)?

**6.19** In future systems, we expect to see heterogeneous computing platforms constructed out of heterogeneous CPUs. We have begun to see some appear in the embedded processing market in systems that contain both floating point DSPs and a microcontroller CPUs in a multichip module package.

Assume that you have three classes of CPU:

CPU A—A moderate speed multi-core CPU (with a floating point unit) that can execute multiple instructions per cycle.

CPU B—A fast single-core integer CPU (i.e., no floating point unit) that can execute a single instruction per cycle.

CPU C—A slow vector CPU (with floating point capability) that can execute multiple copies of the same instruction per cycle.

Assume that our processors run at the following frequencies:

CPU A	CPU B	CPU C
1 GHz	3 GHz	250 MHz

CPU A can execute 2 instructions per cycle, CPU B can execute 1 instruction per cycle, and CPU C can execute 8 instructions (though the same instruction) per cycle. Assume all operations can complete execution in a single cycle of latency without any hazards.

All three CPUs have the ability to perform integer arithmetic, though CPU B cannot perform floating point arithmetic. CPU A and B have an instruction set similar to a MIPS processor. CPU C can only perform floating point add and subtract operations, as well as memory loads and stores. Assume all CPUs have access to shared memory and that synchronization has zero cost.

The task at hand is to compare two matrices X and Y that each contain  $1024 \times 1024$  floating point elements. The output should be a count of the number indices where the value in X was larger or equal to the value in Y.

**6.19.1** [10] <§6.11> Describe how you would partition the problem on the 3 different CPUs to obtain the best performance.

**6.19.2** [10] <§6.11> What kind of instruction would you add to the vector CPU C to obtain better performance?

**6.20** Assume a quad-core computer system can process database queries at a steady state rate of requests per second. Also assume that each transaction takes, on average, a fixed amount of time to process. The following table shows pairs of transaction latency and processing rate.

Average Transaction Latency	Maximum transaction processing rate
1 ms	5000/sec
2 ms	5000/sec
1 ms	10,000/sec
2 ms	10,000/sec

For each of the pairs in the table, answer the following questions:

**6.20.1** [10] <§6.11> On average, how many requests are being processed at any given instant?

**6.20.2** [10] <§6.11> If move to an 8-core system, ideally, what will happen to the system throughput (i.e., how many queries/second will the computer process)?

**6.20.3** [10] <§6.11> Discuss why we rarely obtain this kind of speedup by simply increasing the number of cores.

§6.1, page 504: False. Task-level parallelism can help sequential applications and sequential applications can be made to run on parallel hardware, although it is more challenging.

§6.2, page 509: False. *Weak* scaling can compensate for a serial portion of the program that would otherwise limit scalability, but not so for strong scaling.

§6.3, page 514: True, but they are missing useful vector features like gather-scatter and vector length registers that improve the efficiency of vector architectures. (As an elaboration in this section mentions, the AVX2 SIMD extensions offers indexed loads via a gather operation but *not* scatter for indexed stores. The Haswell generation x86 microprocessor is the first to support AVX2.)

§6.4, page 519: 1. True. 2. True.

§6.5, page 523: False. Since the shared address is a *physical* address, multiple tasks each in their own *virtual* address spaces can run well on a shared memory multiprocessor.

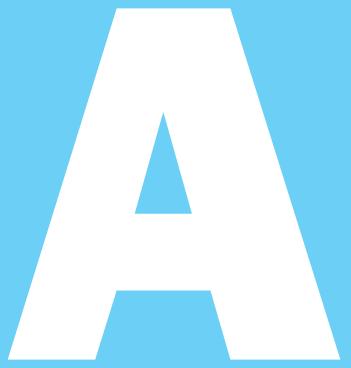
§6.6, page 531: False. Graphics DRAM chips are prized for their higher bandwidth.

§6.7, page 536: 1. False. Sending and receiving a message is an implicit synchronization, as well as a way to share data. 2. True.

§6.8, page 538: True.

§6.10, page 550: True. We likely need innovation at all levels of the hardware and software stack for parallel computing to succeed.

## Answers to Check Yourself



# A

## A P P E N D I X

*Fear of serious injury  
cannot alone justify  
suppression of free  
speech and assembly.*

**Louis Brandeis**  
*Whitney v. California*, 1927

## **Assemblers, Linkers, and the SPIM Simulator**

*James R. Larus*  
Microsoft Research  
Microsoft

<b>A.1</b>	<b>Introduction</b>	A-3
<b>A.2</b>	<b>Assemblers</b>	A-10
<b>A.3</b>	<b>Linkers</b>	A-18
<b>A.4</b>	<b>Loading</b>	A-19
<b>A.5</b>	<b>Memory Usage</b>	A-20
<b>A.6</b>	<b>Procedure Call Convention</b>	A-22
<b>A.7</b>	<b>Exceptions and Interrupts</b>	A-33
<b>A.8</b>	<b>Input and Output</b>	A-38
<b>A.9</b>	<b>SPIM</b>	A-40
<b>A.10</b>	<b>MIPS R2000 Assembly Language</b>	A-45
<b>A.11</b>	<b>Concluding Remarks</b>	A-81
<b>A.12</b>	<b>Exercises</b>	A-82

---

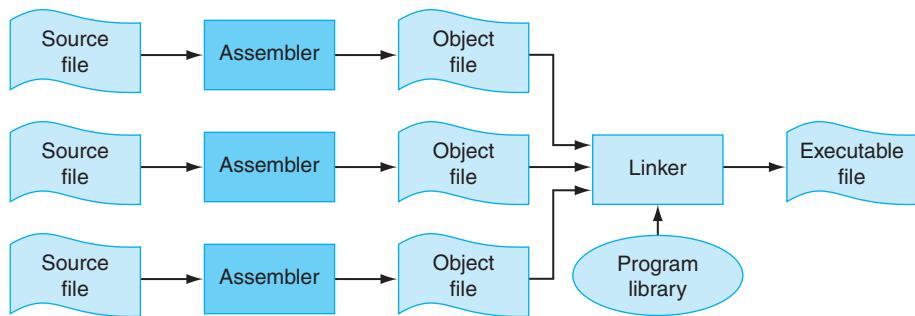
## A.1

### Introduction

Encoding instructions as binary numbers is natural and efficient for computers. Humans, however, have a great deal of difficulty understanding and manipulating these numbers. People read and write symbols (words) much better than long sequences of digits. Chapter 2 showed that we need not choose between numbers and words, because computer instructions can be represented in many ways. Humans can write and read symbols, and computers can execute the equivalent binary numbers. This appendix describes the process by which a human-readable program is translated into a form that a computer can execute, provides a few hints about writing assembly programs, and explains how to run these programs on SPIM, a simulator that executes MIPS programs. UNIX, Windows, and Mac OS X versions of the SPIM simulator are available on the CD.

*Assembly language* is the symbolic representation of a computer’s binary encoding—the **machine language**. Assembly language is more readable than machine language, because it uses symbols instead of bits. The symbols in assembly language name commonly occur in bit patterns, such as opcodes and register specifiers, so people can read and remember them. In addition, assembly language

**machine language**  
Binary representation used for communication within a computer system.



**FIGURE A.1.1 The process that produces an executable file.** An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

**assembler** A program that translates a symbolic version of instruction into the binary version.

**macro** A pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions.

**unresolved reference** A reference that requires more information from an outside source to be complete.

**linker** Also called **link editor**. A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

permits programmers to use *labels* to identify and name particular memory words that hold instructions or data.

A tool called an **assembler** translates assembly language into binary instructions. Assemblers provide a friendlier representation than a computer’s 0s and 1s, which simplifies writing and reading programs. Symbolic names for operations and locations are one facet of this representation. Another facet is programming facilities that increase a program’s clarity. For example, **macros**, discussed in Section A.2, enable a programmer to extend the assembly language by defining new operations.

An assembler reads a single assembly language *source file* and produces an *object file* containing machine instructions and bookkeeping information that helps combine several object files into a program. Figure A.1.1 illustrates how a program is built. Most programs consist of several files—also called *modules*—that are written, compiled, and assembled independently. A program may also use prewritten routines supplied in a *program library*. A module typically contains *references* to subroutines and data defined in other modules and in libraries. The code in a module cannot be executed when it contains **unresolved references** to labels in other object files or libraries. Another tool, called a **linker**, combines a collection of object and library files into an *executable file*, which a computer can run.

To see the advantage of assembly language, consider the following sequence of figures, all of which contain a short subroutine that computes and prints the sum of the squares of integers from 0 to 100. Figure A.1.2 shows the machine language that a MIPS computer executes. With considerable effort, you could use the opcode and instruction format tables in Chapter 2 to translate the instructions into a symbolic program similar to that shown in Figure A.1.3. This form of the routine is much easier to read, because operations and operands are written with symbols rather

```

0010011110111101111111111100000
10101111101111100000000000010100
10101111101001000000000000100000
101011111010010100000000000100100
1010111110100000000000000011000
1010111110100000000000000011100
10001111101011100000000000011100
1000111110111000000000000011000
000000011100111000000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
1010111110101000000000000011100
000000000000000011100000010010
0000001100011111100100000100001
000101000010000011111111110111
101011111011100100000000000011000
00111100000001000001000000000000
100011111010010100000000000011000
0000110000010000000000000011101100
0010010010000100000000000000110000
1000111110111110000000000000010100
001001111011110100000000000010000
000000111110000000000000000000001000
00000000000000000000000000000000100001

```

**FIGURE A.1.2 MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.**

than with bit patterns. However, this assembly language is still difficult to follow, because memory locations are named by their address rather than by a symbolic label.

Figure A.1.4 shows assembly language that labels memory addresses with mnemonic names. Most programmers prefer to read and write this form. Names that begin with a period, for example .data and .globl, are **assembler directives** that tell the assembler how to translate a program but do not produce machine instructions. Names followed by a colon, such as str: or main:, are labels that name the next memory location. This program is as readable as most assembly language programs (except for a glaring lack of comments), but it is still difficult to follow, because many simple operations are required to accomplish simple tasks and because assembly language's lack of control flow constructs provides few hints about the program's operation.

By contrast, the C routine in Figure A.1.5 is both shorter and clearer, since variables have mnemonic names and the loop is explicit rather than constructed with branches. In fact, the C routine is the only one that we wrote. The other forms of the program were produced by a C compiler and assembler.

In general, assembly language plays two roles (see Figure A.1.6). The first role is the output language of compilers. A *compiler* translates a program written in a *high-level language* (such as C or Pascal) into an equivalent program in machine or

#### assembler directive

An operation that tells the assembler how to translate a program but does not produce machine instructions; always begins with a period.

```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4, 32($29)
sw       $5, 36($29)
sw       $0, 24($29)
sw       $0, 28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu   $14, $14
addiu   $8, $14, 1
slti    $1, $8, 101
sw $8,   28($29)
mflo    $15
addu    $25, $24, $15
bne    $1, $0, -9
sw     $25, 24($29)
lui    $4, 4096
lw     $5, 24($29)
jal    1048812
addiu   $4, $4, 1072
lw     $31, 20($29)
addiu   $29, $29, 32
jr     $31
move    $2, $0
```

**FIGURE A.1.3 The same routine as in Figure A.1.2 written in assembly language.** However, the code for the routine does not label registers or memory locations or include comments.

**source language** The high-level language in which a program is originally written.

assembly language. The high-level language is called the **source language**, and the compiler's output is its *target language*.

Assembly language's other role is as a language in which to write programs. This role used to be the dominant one. Today, however, because of larger main memories and better compilers, most programmers write in a high-level language and rarely, if ever, see the instructions that a computer executes. Nevertheless, assembly language is still important to write programs in which speed or size is critical or to exploit hardware features that have no analogues in high-level languages.

Although this appendix focuses on MIPS assembly language, assembly programming on most other machines is very similar. The additional instructions and address modes in CISC machines, such as the VAX, can make assembly programs shorter but do not change the process of assembling a program or provide assembly language with the advantages of high-level languages, such as type-checking and structured control flow.

```

.text
.align 2
.globl main
main:
    subu    $sp, $sp, 32
    sw     $ra, 20($sp)
    sd     $a0, 32($sp)
    sw     $0, 24($sp)
    sw     $0, 28($sp)
loop:
    lw      $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw      $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw     $t9, 24($sp)
    addu  $t0, $t6, 1
    sw     $t0, 28($sp)
    ble   $t0, 100, loop
    la     $a0, str
    lw      $a1, 24($sp)
    jal   printf
    move  $v0, $0
    lw      $ra, 20($sp)
    addu  $sp, $sp, 32
    jr     $ra
.str:
    .asciiz "The sum from 0 .. 100 is %d\n"

```

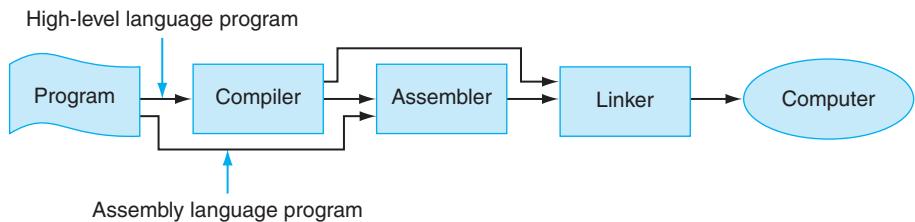
**FIGURE A.1.4 The same routine as in Figure A.1.2 written in assembly language with labels, but no comments.** The commands that start with periods are assembler directives (see pages A-47–49). `.text` indicates that succeeding lines contain instructions. `.data` indicates that they contain data. `.align n` indicates that the items on the succeeding lines should be aligned on a  $2^n$  byte boundary. Hence, `.align 2` means the next item should be on a word boundary. `.globl main` declares that `main` is a global symbol that should be visible to code stored in other files. Finally, `.asciiz` stores a null-terminated string in memory.

## When to Use Assembly Language

The primary reason to program in assembly language, as opposed to an available high-level language, is that the speed or size of a program is critically important. For example, consider a computer that controls a piece of machinery, such as a car's brakes. A computer that is incorporated in another device, such as a car, is called an *embedded computer*. This type of computer needs to respond rapidly and predictably to events in the outside world. Because a compiler introduces

```
#include <stdio.h>
int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;
    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

**FIGURE A.1.5** The routine in Figure A.1.2 written in the C programming language.



**FIGURE A.1.6** Assembly language either is written by a programmer or is the output of a compiler.

uncertainty about the time cost of operations, programmers may find it difficult to ensure that a high-level language program responds within a definite time interval—say, 1 millisecond after a sensor detects that a tire is skidding. An assembly language programmer, on the other hand, has tight control over which instructions execute. In addition, in embedded applications, reducing a program's size, so that it fits in fewer memory chips, reduces the cost of the embedded computer.

A hybrid approach, in which most of a program is written in a high-level language and time-critical sections are written in assembly language, builds on the strengths of both languages. Programs typically spend most of their time executing a small fraction of the program's source code. This observation is just the principle of locality that underlies caches (see Section 5.1 in Chapter 5).

Program profiling measures where a program spends its time and can find the time-critical parts of a program. In many cases, this portion of the program can be made faster with better data structures or algorithms. Sometimes, however, significant performance improvements only come from recoding a critical portion of a program in assembly language.

This improvement is not necessarily an indication that the high-level language's compiler has failed. Compilers typically are better than programmers at producing uniformly high-quality machine code across an entire program. Programmers, however, understand a program's algorithms and behavior at a deeper level than a compiler and can expend considerable effort and ingenuity improving small sections of the program. In particular, programmers often consider several procedures simultaneously while writing their code. Compilers typically compile each procedure in isolation and must follow strict conventions governing the use of registers at procedure boundaries. By retaining commonly used values in registers, even across procedure boundaries, programmers can make a program run faster.

Another major advantage of assembly language is the ability to exploit specialized instructions—for example, string copy or pattern-matching instructions. Compilers, in most cases, cannot determine that a program loop can be replaced by a single instruction. However, the programmer who wrote the loop can replace it easily with a single instruction.

Currently, a programmer's advantage over a compiler has become difficult to maintain as compilation techniques improve and machines' pipelines increase in complexity (Chapter 4).

The final reason to use assembly language is that no high-level language is available on a particular computer. Many older or specialized computers do not have a compiler, so a programmer's only alternative is assembly language.

## Drawbacks of Assembly Language

Assembly language has many disadvantages that strongly argue against its widespread use. Perhaps its major disadvantage is that programs written in assembly language are inherently machine-specific and must be totally rewritten to run on another computer architecture. The rapid evolution of computers discussed in Chapter 1 means that architectures become obsolete. An assembly language program remains tightly bound to its original architecture, even after the computer is eclipsed by new, faster, and more cost-effective machines.

Another disadvantage is that assembly language programs are longer than the equivalent programs written in a high-level language. For example, the C program in [Figure A.1.5](#) is 11 lines long, while the assembly program in [Figure A.1.4](#) is 31 lines long. In more complex programs, the ratio of assembly to high-level language (its *expansion factor*) can be much larger than the factor of three in this example. Unfortunately, empirical studies have shown that programmers write roughly the same number of lines of code per day in assembly as in high-level languages. This means that programmers are roughly  $x$  times more productive in a high-level language, where  $x$  is the assembly language expansion factor.

To compound the problem, longer programs are more difficult to read and understand, and they contain more bugs. Assembly language exacerbates the problem because of its complete lack of structure. Common programming idioms, such as *if-then* statements and loops, must be built from branches and jumps. The resulting programs are hard to read, because the reader must reconstruct every higher-level construct from its pieces and each instance of a statement may be slightly different. For example, look at [Figure A.1.4](#) and answer these questions: What type of loop is used? What are its lower and upper bounds?

**Elaboration:** Compilers can produce machine language directly instead of relying on an assembler. These compilers typically execute much faster than those that invoke an assembler as part of compilation. However, a compiler that generates machine language must perform many tasks that an assembler normally handles, such as resolving addresses and encoding instructions as binary numbers. The tradeoff is between compilation speed and compiler simplicity.

**Elaboration:** Despite these considerations, some embedded applications are written in a high-level language. Many of these applications are large and complex programs that must be extremely reliable. Assembly language programs are longer and more difficult to write and read than high-level language programs. This greatly increases the cost of writing an assembly language program and makes it extremely difficult to verify the correctness of this type of program. In fact, these considerations led the US Department of Defense, which pays for many complex embedded systems, to develop Ada, a new high-level language for writing embedded systems.

## A.2 Assemblers

An assembler translates a file of assembly language statements into a file of binary machine instructions and binary data. The translation process has two major parts. The first step is to find memory locations with labels so that the relationship between symbolic names and addresses is known when instructions are translated. The second step is to translate each assembly statement by combining the numeric equivalents of opcodes, register specifiers, and labels into a legal instruction. As shown in [Figure A.1.1](#), the assembler produces an output file, called an *object file*, which contains the machine instructions, data, and bookkeeping information.

An object file typically cannot be executed, because it references procedures or data in other files. A **label** is **external** (also called **global**) if the labeled object can

**external label** Also called **global label**. A label referring to an object that can be referenced from files other than the one in which it is defined.

be referenced from files other than the one in which it is defined. A label is *local* if the object can be used only within the file in which it is defined. In most assemblers, labels are local by default and must be explicitly declared global. Subroutines and global variables require external labels since they are referenced from many files in a program. **Local labels** hide names that should not be visible to other modules—for example, static functions in C, which can only be called by other functions in the same file. In addition, compiler-generated names—for example, a name for the instruction at the beginning of a loop—are local so that the compiler need not produce unique names in every file.

**local label** A label referring to an object that can be used only within the file in which it is defined.

### Local and Global Labels

Consider the program in [Figure A.1.4](#). The subroutine has an external (global) label `main`. It also contains two local labels—`loop` and `str`—that are only visible within this assembly language file. Finally, the routine also contains an unresolved reference to an external label `printf`, which is the library routine that prints values. Which labels in [Figure A.1.4](#) could be referenced from another file?

### EXAMPLE

Only global labels are visible outside a file, so the only label that could be referenced from another file is `main`.

### ANSWER

Since the assembler processes each file in a program individually and in isolation, it only knows the addresses of local labels. The assembler depends on another tool, the linker, to combine a collection of object files and libraries into an executable file by resolving external labels. The assembler assists the linker by providing lists of labels and unresolved references.

However, even local labels present an interesting challenge to an assembler. Unlike names in most high-level languages, assembly labels may be used before they are defined. In the example in [Figure A.1.4](#), the label `str` is used by the `la` instruction before it is defined. The possibility of a **forward reference**, like this one, forces an assembler to translate a program in two steps: first find all labels and then produce instructions. In the example, when the assembler sees the `la` instruction, it does not know where the word labeled `str` is located or even whether `str` labels an instruction or datum.

**forward reference**

A label that is used before it is defined.

An assembler's first pass reads each line of an assembly file and breaks it into its component pieces. These pieces, which are called *lexemes*, are individual words, numbers, and punctuation characters. For example, the line

```
ble    $t0, 100, loop
```

contains six lexemes: the opcode `ble`, the register specifier `$t0`, a comma, the number `100`, a comma, and the symbol `loop`.

**symbol table** A table that matches names of labels to the addresses of the memory words that instructions occupy.

If a line begins with a label, the assembler records in its **symbol table** the name of the label and the address of the memory word that the instruction occupies. The assembler then calculates how many words of memory the instruction on the current line will occupy. By keeping track of the instructions' sizes, the assembler can determine where the next instruction goes. To compute the size of a variable-length instruction, like those on the VAX, an assembler has to examine it in detail. However, fixed-length instructions, like those on MIPS, require only a cursory examination. The assembler performs a similar calculation to compute the space required for data statements. When the assembler reaches the end of an assembly file, the symbol table records the location of each label defined in the file.

The assembler uses the information in the symbol table during a second pass over the file, which actually produces machine code. The assembler again examines each line in the file. If the line contains an instruction, the assembler combines the binary representations of its opcode and operands (register specifiers or memory address) into a legal instruction. The process is similar to the one used in Section 2.5 in Chapter 2. Instructions and data words that reference an external symbol defined in another file cannot be completely assembled (they are unresolved), since the symbol's address is not in the symbol table. An assembler does not complain about unresolved references, since the corresponding label is likely to be defined in another file.

## The BIG Picture

Assembly language is a programming language. Its principal difference from high-level languages such as BASIC, Java, and C is that assembly language provides only a few, simple types of data and control flow. Assembly language programs do not specify the type of value held in a variable. Instead, a programmer must apply the appropriate operations (e.g., integer or floating-point addition) to a value. In addition, in assembly language, programs must implement all control flow with *go tos*. Both factors make assembly language programming for any machine—MIPS or x86—more difficult and error-prone than writing in a high-level language.

**Elaboration:** If an assembler's speed is important, this two-step process can be done in one pass over the assembly file with a technique known as **backpatching**. In its pass over the file, the assembler builds a (possibly incomplete) binary representation of every instruction. If the instruction references a label that has not yet been defined, the assembler records the label and instruction in a table. When a label is defined, the assembler consults this table to find all instructions that contain a forward reference to the label. The assembler goes back and corrects their binary representation to incorporate the address of the label. Backpatching speeds assembly because the assembler only reads its input once. However, it requires an assembler to hold the entire binary representation of a program in memory so instructions can be backpatched. This requirement can limit the size of programs that can be assembled. The process is complicated by machines with several types of branches that span different ranges of instructions. When the assembler first sees an unresolved label in a branch instruction, it must either use the largest possible branch or risk having to go back and readjust many instructions to make room for a larger branch.

### backpatching

A method for translating from assembly language to machine instructions in which the assembler builds a (possibly incomplete) binary representation of every instruction in one pass over a program and then returns to fill in previously undefined labels.

## Object File Format

Assemblers produce object files. An object file on UNIX contains six distinct sections (see [Figure A.2.1](#)):

- The *object file header* describes the size and position of the other pieces of the file.
- The **text segment** contains the machine language code for routines in the source file. These routines may be unexecutable because of unresolved references.
- The **data segment** contains a binary representation of the data in the source file. The data also may be incomplete because of unresolved references to labels in other files.
- The **relocation information** identifies instructions and data words that depend on **absolute addresses**. These references must change if portions of the program are moved in memory.
- The *symbol table* associates addresses with external labels in the source file and lists unresolved references.
- The *debugging information* contains a concise description of the way the program was compiled, so a debugger can find which instruction addresses correspond to lines in a source file and print the data structures in readable form.

The assembler produces an object file that contains a binary representation of the program and data and additional information to help link pieces of a program.

**text segment** The segment of a UNIX object file that contains the machine language code for routines in the source file.

**data segment** The segment of a UNIX object or executable file that contains a binary representation of the initialized data used by the program.

**relocation information** The segment of a UNIX object file that identifies instructions and data words that depend on absolute addresses.

**absolute address** A variable's or routine's actual address in memory.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

**FIGURE A.2.1 Object file.** A UNIX assembler produces an object file with six distinct sections.

This relocation information is necessary because the assembler does not know which memory locations a procedure or piece of data will occupy after it is linked with the rest of the program. Procedures and data from a file are stored in a contiguous piece of memory, but the assembler does not know where this memory will be located. The assembler also passes some symbol table entries to the linker. In particular, the assembler must record which external symbols are defined in a file and what unresolved references occur in a file.

**Elaboration:** For convenience, assemblers assume each file starts at the same address (for example, location 0) with the expectation that the linker will *relocate* the code and data when they are assigned locations in memory. The assembler produces *relocation information*, which contains an entry describing each instruction or data word in the file that references an absolute address. On MIPS, only the subroutine call, load, and store instructions reference absolute addresses. Instructions that use PC-relative addressing, such as branches, need not be relocated.

## Additional Facilities

Assemblers provide a variety of convenience features that help make assembler programs shorter and easier to write, but do not fundamentally change assembly language. For example, *data layout directives* allow a programmer to describe data in a more concise and natural manner than its binary representation.

In Figure A.1.4, the directive

```
.asciiz "The sum from 0 .. 100 is %d\n"
```

stores characters from the string in memory. Contrast this line with the alternative of writing each character as its ASCII value (Figure 2.15 in Chapter 2 describes the ASCII encoding for characters):

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

The `.asciiz` directive is easier to read because it represents characters as letters, not binary numbers. An assembler can translate characters to their binary representation much faster and more accurately than a human can. Data layout directives

specify data in a human-readable form that the assembler translates to binary. Other layout directives are described in Section A.10.

### String Directive

Define the sequence of bytes produced by this directive:

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

**EXAMPLE**

```
.byte 84, 104, 101, 32, 113, 117, 105, 99  
.byte 107, 32, 98, 114, 111, 119, 110, 32  
.byte 102, 111, 120, 32, 106, 117, 109, 112  
.byte 115, 32, 111, 118, 101, 114, 32, 116  
.byte 104, 101, 32, 108, 97, 122, 121, 32  
.byte 100, 111, 103, 0
```

**ANSWER**

*Macro* is a pattern-matching and replacement facility that provides a simple mechanism to name a frequently used sequence of instructions. Instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions. Macros, like subroutines, permit a programmer to create and name a new abstraction for a common operation. Unlike subroutines, however, macros do not cause a subroutine call and return when the program runs, since a macro call is replaced by the macro's body when the program is assembled. After this replacement, the resulting assembly is indistinguishable from the equivalent program written without macros.

### Macros

As an example, suppose that a programmer needs to print many numbers. The library routine `printf` accepts a format string and one or more values to print as its arguments. A programmer could print the integer in register \$7 with the following instructions:

```
.data  
int_str: .asciiz "%d"  
.text  
la    $a0, int_str # Load string address  
                  # into first arg
```

**EXAMPLE**

```
        mov    $a1, $7  # Load value into
                      # second arg
        jal    printf  # Call the printf routine
```

The `.data` directive tells the assembler to store the string in the program's data segment, and the `.text` directive tells the assembler to store the instructions in its text segment.

However, printing many numbers in this fashion is tedious and produces a verbose program that is difficult to understand. An alternative is to introduce a macro, `print_int`, to print an integer:

```
.data
int_str:.asciiz "%d"
.text
.macro print_int($arg)
la $a0, int_str # Load string address into
                  # first arg
mov $a1, $arg    # Load macro's parameter
                  # ($arg) into second arg
jal printf      # Call the printf routine
.end_macro
print_int($7)
```

### formal parameter

A variable that is the argument to a procedure or macro; it is replaced by that argument once the macro is expanded.

The macro has a **formal parameter**, `$arg`, that names the argument to the macro. When the macro is expanded, the argument from a call is substituted for the formal parameter throughout the macro's body. Then the assembler replaces the call with the macro's newly expanded body. In the first call on `print_int`, the argument is `$7`, so the macro expands to the code

```
la $a0, int_str
mov $a1, $7
jal printf
```

In a second call on `print_int`, say, `print_int($t0)`, the argument is `$t0`, so the macro expands to

```
la $a0, int_str
mov $a1, $t0
jal printf
```

What does the call `print_int($a0)` expand to?

```
la $a0, int_str  
mov $a1, $a0  
jal printf
```

**ANSWER**

This example illustrates a drawback of macros. A programmer who uses this macro must be aware that `print_int` uses register `$a0` and so cannot correctly print the value in that register.

Some assemblers also implement *pseudoinstructions*, which are instructions provided by an assembler but not implemented in hardware. Chapter 2 contains many examples of how the MIPS assembler synthesizes pseudoinstructions and addressing modes from the spartan MIPS hardware instruction set. For example, Section 2.7 in Chapter 2 describes how the assembler synthesizes the `blt` instruction from two other instructions: `slt` and `bne`. By extending the instruction set, the MIPS assembler makes assembly language programming easier without complicating the hardware. Many pseudoinstructions could also be simulated with macros, but the MIPS assembler can generate better code for these instructions because it can use a dedicated register (`$at`) and is able to optimize the generated code.

**Hardware/  
Software  
Interface**

**Elaboration:** Assemblers conditionally assemble pieces of code, which permits a programmer to include or exclude groups of instructions when a program is assembled. This feature is particularly useful when several versions of a program differ by a small amount. Rather than keep these programs in separate files—which greatly complicates fixing bugs in the common code—programmers typically merge the versions into a single file. Code particular to one version is conditionally assembled, so it can be excluded when other versions of the program are assembled.

If macros and conditional assembly are useful, why do assemblers for UNIX systems rarely, if ever, provide them? One reason is that most programmers on these systems write programs in higher-level languages like C. Most of the assembly code is produced by compilers, which find it more convenient to repeat code rather than define macros. Another reason is that other tools on UNIX—such as `cpp`, the C preprocessor, or `m4`, a general macro processor—can provide macros and conditional assembly for assembly language programs.

## A.3

### Linkers

#### separate compilation

Splitting a program across many files, each of which can be compiled without knowledge of what is in the other files.

**Separate compilation** permits a program to be split into pieces that are stored in different files. Each file contains a logically related collection of subroutines and data structures that form a *module* in a larger program. A file can be compiled and assembled independently of other files, so changes to one module do not require recompiling the entire program. As we discussed above, separate compilation necessitates the additional step of linking to combine object files from separate modules and fixing their unresolved references.

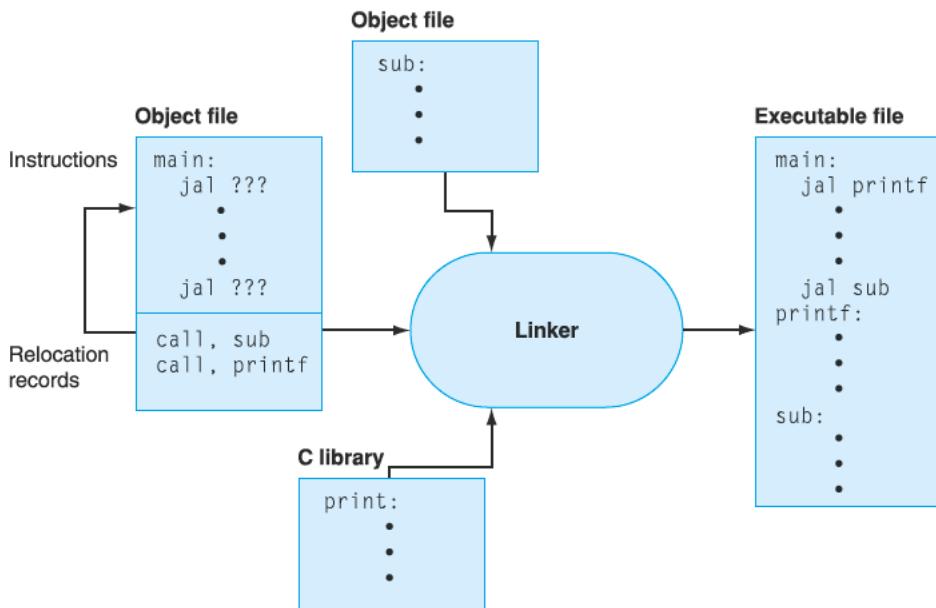
The tool that merges these files is the *linker* (see [Figure A.3.1](#)). It performs three tasks:

- Searches the program libraries to find library routines used by the program
- Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
- Resolves references among files

A linker's first task is to ensure that a program contains no undefined labels. The linker matches the external symbols and unresolved references from a program's files. An external symbol in one file resolves a reference from another file if both refer to a label with the same name. Unmatched references mean a symbol was used but not defined anywhere in the program.

Unresolved references at this stage in the linking process do not necessarily mean a programmer made a mistake. The program could have referenced a library routine whose code was not in the object files passed to the linker. After matching symbols in the program, the linker searches the system's program libraries to find predefined subroutines and data structures that the program references. The basic libraries contain routines that read and write data, allocate and deallocate memory, and perform numeric operations. Other libraries contain routines to access a database or manipulate terminal windows. A program that references an unresolved symbol that is not in any library is erroneous and cannot be linked. When the program uses a library routine, the linker extracts the routine's code from the library and incorporates it into the program text segment. This new routine, in turn, may depend on other library routines, so the linker continues to fetch other library routines until no external references are unresolved or a routine cannot be found.

If all external references are resolved, the linker next determines the memory locations that each module will occupy. Since the files were assembled in isolation,



**FIGURE A.3.1** The **Linker** searches a collection of **object files** and program **libraries** to find nonlocal routines used in a program, combines them into a single **executable file**, and resolves references between routines in different files.

the assembler could not know where a module's instructions or data would be placed relative to other modules. When the linker places a module in memory, all absolute references must be *relocated* to reflect its true location. Since the linker has relocation information that identifies all relocatable references, it can efficiently find and backpatch these references.

The linker produces an executable file that can run on a computer. Typically, this file has the same format as an object file, except that it contains no unresolved references or relocation information.

## A.4 Loading

A program that links without an error can be run. Before being run, the program resides in a file on secondary storage, such as a disk. On UNIX systems, the operating

system kernel brings a program into memory and starts it running. To start a program, the operating system performs the following steps:

1. It reads the executable file's header to determine the size of the text and data segments.
2. It creates a new address space for the program. This address space is large enough to hold the text and data segments, along with a stack segment (see Section A.5).
3. It copies instructions and data from the executable file into the new address space.
4. It copies arguments passed to the program onto the stack.
5. It initializes the machine registers. In general, most registers are cleared, but the stack pointer must be assigned the address of the first free stack location (see Section A.5).
6. It jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's `main` routine. If the `main` routine returns, the start-up routine terminates the program with the `exit` system call.

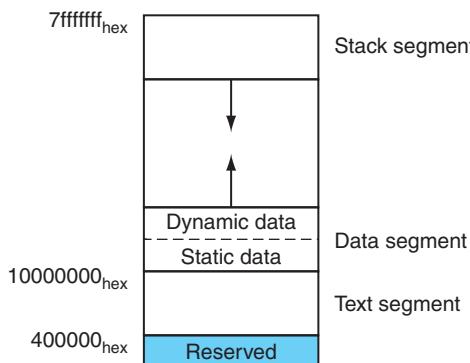
## A.5 Memory Usage

The next few sections elaborate the description of the MIPS architecture presented earlier in the book. Earlier chapters focused primarily on hardware and its relationship with low-level software. These sections focus primarily on how assembly language programmers use MIPS hardware. These sections describe a set of conventions followed on many MIPS systems. For the most part, the hardware does not impose these conventions. Instead, they represent an agreement among programmers to follow the same set of rules so that software written by different people can work together and make effective use of MIPS hardware.

Systems based on MIPS processors typically divide memory into three parts (see [Figure A.5.1](#)). The first part, near the bottom of the address space (starting at address  $400000_{\text{hex}}$ ), is the *text segment*, which holds the program's instructions.

The second part, above the text segment, is the *data segment*, which is further divided into two parts. **Static data** (starting at address  $10000000_{\text{hex}}$ ) contains objects whose size is known to the compiler and whose lifetime—the interval during which a program can access them—is the program's entire execution. For example, in C, global variables are statically allocated, since they can be referenced

**static data** The portion of memory that contains data whose size is known to the compiler and whose lifetime is the program's entire execution.



**FIGURE A.5.1 Layout of memory.**

anytime during a program's execution. The linker both assigns static objects to locations in the data segment and resolves references to these objects.

Immediately above static data is *dynamic data*. This data, as its name implies, is allocated by the program as it executes. In C programs, the `malloc` library routine

Because the data segment begins far above the program at address  $10000000_{hex}$ , load and store instructions cannot directly reference data objects with their 16-bit offset fields (see Section 2.5 in Chapter 2). For example, to load the word in the data segment at address  $10010020_{hex}$  into register  $\$v0$  requires two instructions:

```
lui $s0, 0x1001 # 0x1001 means 1001 base 16
lw $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

(The  $0x$  before a number means that it is a hexadecimal value. For example,  $0x8000$  is  $8000_{hex}$  or  $32,768_{ten}$ .)

To avoid repeating the `lui` instruction at every load and store, MIPS systems typically dedicate a register (`$gp`) as a *global pointer* to the static data segment. This register contains address  $10008000_{hex}$ , so load and store instructions can use their signed 16-bit offset fields to access the first 64 KB of the static data segment. With this global pointer, we can rewrite the example as a single instruction:

```
lw $v0, 0x8020($gp)
```

Of course, a global pointer register makes addressing locations  $10000000_{hex}$ – $10010000_{hex}$  faster than other heap locations. The MIPS compiler usually stores *global variables* in this area, because these variables have fixed locations and fit better than other global data, such as arrays.

## Hardware/ Software Interface

finds and returns a new block of memory. Since a compiler cannot predict how much memory a program will allocate, the operating system expands the dynamic data area to meet demand. As the upward arrow in the figure indicates, `malloc` expands the dynamic area with the `sbrk` system call, which causes the operating system to add more pages to the program's virtual address space (see Section 5.7 in Chapter 5) immediately above the dynamic data segment.

**stack segment** The portion of memory used by a program to hold procedure call frames.

The third part, the program **stack segment**, resides at the top of the virtual address space (starting at address  $7\text{fffffff}_{\text{hex}}$ ). Like dynamic data, the maximum size of a program's stack is not known in advance. As the program pushes values on to the stack, the operating system expands the stack segment down toward the data segment.

This three-part division of memory is not the only possible one. However, it has two important characteristics: the two dynamically expandable segments are as far apart as possible, and they can grow to use a program's entire address space.

## A.6

## Procedure Call Convention

**register use convention**  
Also called **procedure call convention**.  
A software protocol governing the use of registers by procedures.

Conventions governing the use of registers are necessary when procedures in a program are compiled separately. To compile a particular procedure, a compiler must know which registers it may use and which registers are reserved for other procedures. Rules for using registers are called **register use** or **procedure call conventions**. As the name implies, these rules are, for the most part, conventions followed by software rather than rules enforced by hardware. However, most compilers and programmers try very hard to follow these conventions because violating them causes insidious bugs.

The calling convention described in this section is the one used by the `gcc` compiler. The native MIPS compiler uses a more complex convention that is slightly faster.

The MIPS CPU contains 32 general-purpose registers that are numbered 0–31. Register `$0` always contains the hardwired value 0.

- Registers `$at` (1), `$k0` (26), and `$k1` (27) are reserved for the assembler and operating system and should not be used by user programs or compilers.
- Registers `$a0`–`$a3` (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers `$v0` and `$v1` (2, 3) are used to return values from functions.

- Registers \$t0-\$t9 (8–15, 24, 25) are **caller-saved registers** that are used to hold temporary quantities that need not be preserved across calls (see Section 2.8 in Chapter 2).
- Registers \$s0-\$s7 (16–23) are **callee-saved registers** that hold long-lived values that should be preserved across calls.
- Register \$gp (28) is a global pointer that points to the middle of a 64K block of memory in the static data segment.
- Register \$sp (29) is the stack pointer, which points to the last location on the stack. Register \$fp (30) is the frame pointer. The `jal` instruction writes register \$ra (31), the return address from a procedure call. These two registers are explained in the next section.

**caller-saved register**  
A register saved by the routine being called.

**callee-saved register**  
A register saved by the routine making a procedure call.

The two-letter abbreviations and names for these registers—for example \$sp for the stack pointer—reflect the registers’ intended uses in the procedure call convention. In describing this convention, we will use the names instead of register numbers. [Figure A.6.1](#) lists the registers and describes their intended uses.

## Procedure Calls

This section describes the steps that occur when one procedure (the *caller*) invokes another procedure (the *callee*). Programmers who write in a high-level language (like C or Pascal) never see the details of how one procedure calls another, because the compiler takes care of this low-level bookkeeping. However, assembly language programmers must explicitly implement every procedure call and return.

Most of the bookkeeping associated with a call is centered around a block of memory called a **procedure call frame**. This memory is used for a variety of purposes:

- To hold values passed to a procedure as arguments
- To save registers that a procedure may modify, but which the procedure’s caller does not want changed
- To provide space for variables local to a procedure

In most programming languages, procedure calls and returns follow a strict last-in, first-out (LIFO) order, so this memory can be allocated and deallocated on a stack, which is why these blocks of memory are sometimes called stack frames.

[Figure A.6.2](#) shows a typical stack frame. The frame consists of the memory between the frame pointer (\$fp), which points to the first word of the frame, and the stack pointer (\$sp), which points to the last word of the frame. The stack grows down from higher memory addresses, so the frame pointer points above the

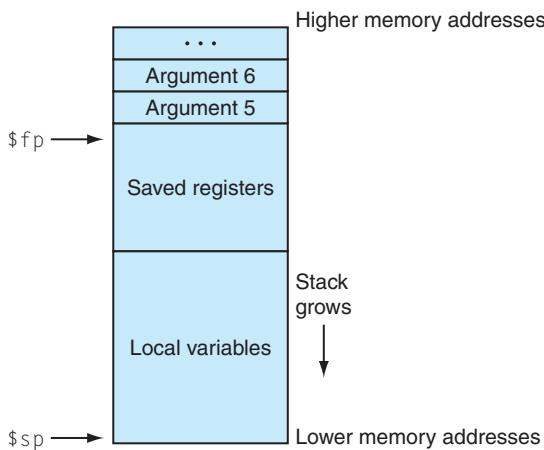
**procedure call frame**  
A block of memory that is used to hold values passed to a procedure as arguments, to save registers that a procedure may modify but that the procedure’s caller does not want changed, and to provide space for variables local to a procedure.

Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)
\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

**FIGURE A.6.1 MIPS registers and usage convention.**

stack pointer. The executing procedure uses the frame pointer to quickly access values in its stack frame. For example, an argument in the stack frame can be loaded into register \$v0 with the instruction

```
lw $v0, 0($fp)
```



**FIGURE A.6.2 Layout of a stack frame.** The frame pointer ( $\$fp$ ) points to the first word in the currently executing procedure's stack frame. The stack pointer ( $\$sp$ ) points to the last word of the frame. The first four arguments are passed in registers, so the fifth argument is the first one stored on the stack.

A stack frame may be built in many different ways; however, the caller and callee must agree on the sequence of steps. The steps below describe the calling convention used on most MIPS machines. This convention comes into play at three points during a procedure call: immediately before the caller invokes the callee, just as the callee starts executing, and immediately before the callee returns to the caller. In the first part, the caller puts the procedure call arguments in standard places and invokes the callee to do the following:

1. Pass arguments. By convention, the first four arguments are passed in registers  $\$a0-\$a3$ . Any remaining arguments are pushed on the stack and appear at the beginning of the called procedure's stack frame.
2. Save caller-saved registers. The called procedure can use these registers ( $\$a0-\$a3$  and  $\$t0-\$t9$ ) without first saving their value. If the caller expects to use one of these registers after a call, it must save its value before the call.
3. Execute a `jal` instruction (see Section 2.8 of Chapter 2), which jumps to the callee's first instruction and saves the return address in register  $\$ra$ .

Before a called routine starts running, it must take the following steps to set up its stack frame:

1. Allocate memory for the frame by subtracting the frame's size from the stack pointer.
2. Save callee-saved registers in the frame. A callee must save the values in these registers ( $\$s0-\$s7$ ,  $\$fp$ , and  $\$ra$ ) before altering them, since the caller expects to find these registers unchanged after the call. Register  $\$fp$  is saved by every procedure that allocates a new stack frame. However, register  $\$ra$  only needs to be saved if the callee itself makes a call. The other callee-saved registers that are used also must be saved.
3. Establish the frame pointer by adding the stack frame's size minus 4 to  $\$sp$  and storing the sum in register  $\$fp$ .

---

## Hardware/ Software Interface

The MIPS register use convention provides callee- and caller-saved registers, because both types of registers are advantageous in different circumstances. Callee-saved registers are better used to hold long-lived values, such as variables from a user's program. These registers are only saved during a procedure call if the callee expects to use the register. On the other hand, caller-saved registers are better used to hold short-lived quantities that do not persist across a call, such as immediate values in an address calculation. During a call, the callee can also use these registers for short-lived temporaries.

---

Finally, the callee returns to the caller by executing the following steps:

1. If the callee is a function that returns a value, place the returned value in register  $\$v0$ .
2. Restore all callee-saved registers that were saved upon procedure entry.
3. Pop the stack frame by adding the frame size to  $\$sp$ .
4. Return by jumping to the address in register  $\$ra$ .

**recursive procedures**  
Procedures that call themselves either directly or indirectly through a chain of calls.

**Elaboration:** A programming language that does not permit **recursive procedures**—procedures that call themselves either directly or indirectly through a chain of calls—need not allocate frames on a stack. In a nonrecursive language, each procedure's frame may be statically allocated, since only one invocation of a procedure can be active at a time. Older versions of Fortran prohibited recursion, because statically allocated frames produced faster code on some older machines. However, on load store architectures like MIPS, stack frames may be just as fast, because a frame pointer register points directly

to the active stack frame, which permits a single load or store instruction to access values in the frame. In addition, recursion is a valuable programming technique.

## Procedure Call Example

As an example, consider the C routine

```
main ()
{
    printf ("The factorial of 10 is %d\n", fact (10));
}

int fact (int n)
{
    if (n < 1)
        return (1);
    else
        return (n * fact (n - 1));
}
```

which computes and prints  $10!$  (the factorial of 10,  $10! = 10 \times 9 \times \dots \times 1$ ). `fact` is a recursive routine that computes  $n!$  by multiplying  $n$  times  $(n - 1)!$ . The assembly code for this routine illustrates how programs manipulate stack frames.

Upon entry, the routine `main` creates its stack frame and saves the two callee-saved registers it will modify: `$fp` and `$ra`. The frame is larger than required for these two register because the calling convention requires the minimum size of a stack frame to be 24 bytes. This minimum frame can hold four argument registers (`$a0-$a3`) and the return address `$ra`, padded to a double-word boundary (24 bytes). Since `main` also needs to save `$fp`, its stack frame must be two words larger (remember: the stack pointer is kept doubleword aligned).

```
.text
.globl main
main:
    subu $sp,$sp,32      # Stack frame is 32 bytes long
    sw    $ra,20($sp)    # Save return address
    sw    $fp,16($sp)    # Save old frame pointer
    addiu $fp,$sp,28     # Set up frame pointer
```

The routine `main` then calls the factorial routine and passes it the single argument 10. After `fact` returns, `main` calls the library routine `printf` and passes it both a format string and the result returned from `fact`:

```

    li      $a0,10      # Put argument (10) in $a0
    jal     fact         # Call factorial function

    la      $a0,$LC      # Put format string in $a0
    move   $a1,$v0        # Move fact result to $a1
    jal     printf       # Call the print function

```

Finally, after printing the factorial, `main` returns. But first, it must restore the registers it saved and pop its stack frame:

```

lw      $ra,20($sp)  # Restore return address
lw      $fp,16($sp)  # Restore frame pointer
addiu $sp,$sp,32      # Pop stack frame
jr      $ra            # Return to caller

.rdata
$L0:
.ascii  "The factorial of 10 is %d\n\000"

```

The factorial routine is similar in structure to `main`. First, it creates a stack frame and saves the callee-saved registers it will use. In addition to saving `$ra` and `$fp`, `fact` also saves its argument (`$a0`), which it will use for the recursive call:

```

.text
fact:
    subu $sp,$sp,32      # Stack frame is 32 bytes long
    sw    $ra,20($sp)    # Save return address
    sw    $fp,16($sp)    # Save frame pointer
    addiu $fp,$sp,28      # Set up frame pointer
    sw    $a0,0($fp)      # Save argument (n)

```

The heart of the `fact` routine performs the computation from the C program. It tests whether the argument is greater than 0. If not, the routine returns the value 1. If the argument is greater than 0, the routine recursively calls itself to compute `fact(n-1)` and multiplies that value times *n*:

```

lw      $v0,0($fp)    # Load n
bgtz $v0,$L2          # Branch if n > 0
li      $v0,1            # Return 1
jr      $L1              # Jump to code to return

$L2:
    lw      $v1,0($fp)    # Load n
    subu $v0,$v1,1        # Compute n - 1
    move  $a0,$v0          # Move value to $a0

```

```

jal      fact          # Call factorial function

lw       $v1,0($fp)    # Load n
mul      $v0,$v0,$v1   # Compute fact(n-1) * n

```

Finally, the factorial routine restores the callee-saved registers and returns the value in register \$v0:

```

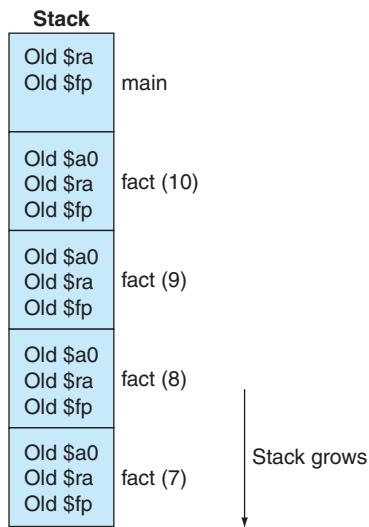
$L1:                                # Result is in $v0
lw      $ra, 20($sp)  # Restore $ra
lw      $fp, 16($sp)  # Restore $fp
addiu $sp, $sp, 32 # Pop stack
jr      $ra           # Return to caller

```

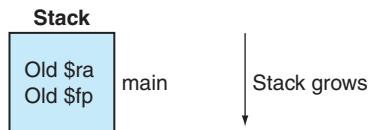
### Stack in Recursive Procedure

Figure A.6.3 shows the stack at the call `fact(7)`. `main` runs first, so its frame is deepest on the stack. `main` calls `fact(10)`, whose stack frame is next on the stack. Each invocation recursively invokes `fact` to compute the next-lowest factorial. The stack frames parallel the LIFO order of these calls. What does the stack look like when the call to `fact(10)` returns?

### EXAMPLE



**FIGURE A.6.3** Stack frames during the call of `fact(7)`.

**ANSWER**

**Elaboration:** The difference between the MIPS compiler and the gcc compiler is that the MIPS compiler usually does not use a frame pointer, so this register is available as another callee-saved register, `$s8`. This change saves a couple of instructions in the procedure call and return sequence. However, it complicates code generation, because a procedure must access its stack frame with `$sp`, whose value can change during a procedure's execution if values are pushed on the stack.

### Another Procedure Call Example

As another example, consider the following routine that computes the `tak` function, which is a widely used benchmark created by Ikuo Takeuchi. This function does not compute anything useful, but is a heavily recursive program that illustrates the MIPS calling convention.

```
int tak (int x, int y, int z)
{
    if (y < x)
        return 1+ tak (tak (x - 1, y, z),
                      tak (y - 1, z, x),
                      tak (z - 1, x, y));
    else
        return z;
}
int main ()
{
    tak(18, 12, 6);
}
```

The assembly code for this program is shown below. The `tak` function first saves its return address in its stack frame and its arguments in callee-saved registers, since the routine may make calls that need to use registers `$a0-$a2` and `$ra`. The function uses callee-saved registers, since they hold values that persist over the

lifetime of the function, which includes several calls that could potentially modify registers.

```
.text
.globl tak

tak:
    subu    $sp, $sp, 40
    sw      $ra, 32($sp)

    sw      $s0, 16($sp)    # x
    move   $s0, $a0
    sw      $s1, 20($sp)    # y
    move   $s1, $a1
    sw      $s2, 24($sp)    # z
    move   $s2, $a2
    sw      $s3, 28($sp)    # temporary
```

The routine then begins execution by testing if  $y < x$ . If not, it branches to label L1, which is shown below.

```
bge    $s1, $s0, L1    # if ( $y < x$ )
```

If  $y < x$ , then it executes the body of the routine, which contains four recursive calls. The first call uses almost the same arguments as its parent:

```
addiu   $a0, $s0, -1
move    $a1, $s1
move    $a2, $s2
jal     tak           # tak ( $x - 1, y, z$ )
move    $s3, $v0
```

Note that the result from the first recursive call is saved in register  $\$s3$ , so that it can be used later.

The function now prepares arguments for the second recursive call.

```
addiu   $a0, $s1, -1
move    $a1, $s2
move    $a2, $s0
jal     tak           # tak ( $y - 1, z, x$ )
```

In the instructions below, the result from this recursive call is saved in register  $\$s0$ . But first we need to read, for the last time, the saved value of the first argument from this register.

```

addiu    $a0, $s2, -1
move     $a1, $s0
move     $a2, $s1
move     $s0, $v0
jal      tak           # tak (z - 1, x, y)

```

After the three inner recursive calls, we are ready for the final recursive call. After the call, the function's result is in  $\$v0$  and control jumps to the function's epilogue.

```

move    $a0, $s3
move    $a1, $s0
move    $a2, $v0
jal     tak           # tak (tak(...), tak(...), tak(...))
addiu   $v0, $v0, 1
j      L2

```

This code at label  $L1$  is the consequent of the *if-then-else* statement. It just moves the value of argument  $z$  into the return register and falls into the function epilogue.

```

L1:
move    $v0, $s2

```

The code below is the function epilogue, which restores the saved registers and returns the function's result to its caller.

```

L2:
lw      $ra, 32($sp)
lw      $s0, 16($sp)
lw      $s1, 20($sp)
lw      $s2, 24($sp)
lw      $s3, 28($sp)
addiu  $sp, $sp, 40
jr      $ra

```

The `main` routine calls the `tak` function with its initial arguments, then takes the computed result (7) and prints it using SPIM's system call for printing integers.

```

.globl  main
main:
subu   $sp, $sp, 24
sw     $ra, 16($sp)

li     $a0, 18
li     $a1, 12

```

```

    li      $a2, 6
    jal     tak          # tak(18, 12, 6)

    move   $a0, $v0
    li      $v0, 1          # print_int syscall
    syscall

    lw      $ra, 16($sp)
    addiu $sp, $sp, 24
    jr      $ra

```

**A.7****Exceptions and Interrupts**

Section 4.9 of Chapter 4 describes the MIPS exception facility, which responds both to exceptions caused by errors during an instruction's execution and to external interrupts caused by I/O devices. This section describes exception and **interrupt handling** in more detail.<sup>1</sup> In MIPS processors, a part of the CPU called *coprocessor 0* records the information the software needs to handle exceptions and interrupts. The MIPS simulator SPIM does not implement all of coprocessor 0's registers, since many are not useful in a simulator or are part of the memory system, which SPIM does not implement. However, SPIM does provide the following coprocessor 0 registers:

**interrupt handler**

A piece of code that is run as a result of an exception or an interrupt.

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

1. This section discusses exceptions in the MIPS-32 architecture, which is what SPIM implements in Version 7.0 and later. Earlier versions of SPIM implemented the MIPS-1 architecture, which handled exceptions slightly differently. Converting programs from these versions to run on MIPS-32 should not be difficult, as the changes are limited to the Status and Cause register fields and the replacement of the `rfe` instruction by the `eret` instruction.

These seven registers are part of coprocessor 0's register set. They are accessed by the `mfc0` and `mtc0` instructions. After an exception, register EPC contains the address of the instruction that was executing when the exception occurred. If the exception was caused by an external interrupt, then the instruction will not have started executing. All other exceptions are caused by the execution of the instruction at EPC, except when the offending instruction is in the delay slot of a branch or jump. In that case, EPC points to the branch or jump instruction and the BD bit is set in the Cause register. When that bit is set, the exception handler must look at  $EPC + 4$  for the offending instruction. However, in either case, an exception handler properly resumes the program by returning to the instruction at EPC.

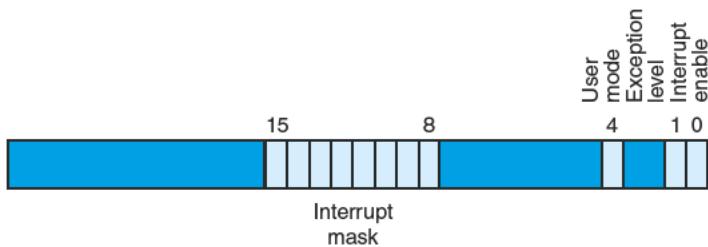
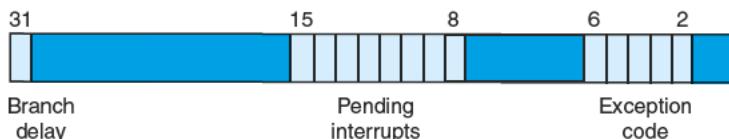
If the instruction that caused the exception made a memory access, register `BadVAddr` contains the referenced memory location's address.

The Count register is a timer that increments at a fixed rate (by default, every 10 milliseconds) while SPIM is running. When the value in the Count register equals the value in the Compare register, a hardware interrupt at priority level 5 occurs.

[Figure A.7.1](#) shows the subset of the Status register fields implemented by the MIPS simulator SPIM. The `interrupt mask` field contains a bit for each of the six hardware and two software interrupt levels. A mask bit that is 1 allows interrupts at that level to interrupt the processor. A mask bit that is 0 disables interrupts at that level. When an interrupt arrives, it sets its interrupt pending bit in the Cause register, even if the mask bit is disabled. When an interrupt is pending, it will interrupt the processor when its mask bit is subsequently enabled.

The user mode bit is 0 if the processor is running in kernel mode and 1 if it is running in user mode. On SPIM, this bit is fixed at 1, since the SPIM processor does not implement kernel mode. The exception level bit is normally 0, but is set to 1 after an exception occurs. When this bit is 1, interrupts are disabled and the EPC is not updated if another exception occurs. This bit prevents an exception handler from being disturbed by an interrupt or exception, but it should be reset when the handler finishes. If the `interrupt enable` bit is 1, interrupts are allowed. If it is 0, they are disabled.

[Figure A.7.2](#) shows the subset of Cause register fields that SPIM implements. The branch delay bit is 1 if the last exception occurred in an instruction executed in the delay slot of a branch. The interrupt pending bits become 1 when an interrupt

**FIGURE A.7.1** The Status register.**FIGURE A.7.2** The Cause register.

is raised at a given hardware or software level. The exception code register describes the cause of an exception through the following codes:

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load or instruction fetch)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction exception
11	Cpu	coprocessor unimplemented
12	Ov	arithmetic overflow exception
13	Tr	trap
15	FPE	floating point

Exceptions and interrupts cause a MIPS processor to jump to a piece of code, at address  $80000180_{\text{hex}}$  (in the kernel, not user address space), called an *exception handler*. This code examines the exception's cause and jumps to an appropriate point in the operating system. The operating system responds to an exception either by terminating the process that caused the exception or by performing some action. A process that causes an error, such as executing an unimplemented instruction, is killed by the operating system. On the other hand, other exceptions such as page

faults are requests from a process to the operating system to perform a service, such as bringing in a page from disk. The operating system processes these requests and resumes the process. The final type of exceptions are interrupts from external devices. These generally cause the operating system to move data to or from an I/O device and resume the interrupted process.

The code in the example below is a simple exception handler, which invokes a routine to print a message at each exception (but not interrupts). This code is similar to the exception handler (`exceptions.s`) used by the SPIM simulator.

## EXAMPLE

### Exception Handler

The exception handler first saves register `$at`, which is used in pseudo-instructions in the handler code, then saves `$a0` and `$a1`, which it later uses to pass arguments. The exception handler cannot store the old values from these registers on the stack, as would an ordinary routine, because the cause of the exception might have been a memory reference that used a bad value (such as 0) in the stack pointer. Instead, the exception handler stores these registers in an exception handler register (`$k1`, since it can't access memory without using `$at`) and two memory locations (`save0` and `save1`). If the exception routine itself could be interrupted, two locations would not be enough since the second exception would overwrite values saved during the first exception. However, this simple exception handler finishes running before it enables interrupts, so the problem does not arise.

```
.ktext 0x80000180
mov $k1, $at      # Save $at register
sw  $a0, save0   # Handler is not re-entrant and can't use
sw  $a1, save1   # stack to save $a0, $a1
                  # Don't need to save $k0/$k1
```

The exception handler then moves the Cause and EPC registers into CPU registers. The Cause and EPC registers are not part of the CPU register set. Instead, they are registers in coprocessor 0, which is the part of the CPU that handles exceptions. The instruction `mfc0 $k0, $13` moves coprocessor 0's register 13 (the Cause register) into CPU register `$k0`. Note that the exception handler need not save registers `$k0` and `$k1`, because user programs are not supposed to use these registers. The exception handler uses the value from the Cause register to test whether the exception was caused by an interrupt (see the preceding table). If so, the exception is ignored. If the exception was not an interrupt, the handler calls `print_excp` to print a message.

---

```

mfc0    $k0, $13          # Move Cause into $k0

srl     $a0, $k0, 2        # Extract ExcCode field
andi   $a0, $a0, 0xf

bgtz   $a0, done          # Branch if ExcCode is Int (0)

mov     $a0, $k0          # Move Cause into $a0
mfco   $a1, $14          # Move EPC into $a1
jal    print_excp       # Print exception error message

```

Before returning, the exception handler clears the Cause register; resets the Status register to enable interrupts and clear the EXL bit, which allows subsequent exceptions to change the EPC register; and restores registers \$a0, \$a1, and \$at. It then executes the `eret` (exception return) instruction, which returns to the instruction pointed to by EPC. This exception handler returns to the instruction following the one that caused the exception, so as to not re-execute the faulting instruction and cause the same exception again.

```

done:    mfc0    $k0, $14          # Bump EPC
         addiu   $k0, $k0, 4        # Do not re-execute
                               # faulting instruction
         mtc0    $k0, $14          # EPC

         mtc0    $0, $13          # Clear Cause register

         mfc0    $k0, $12          # Fix Status register
         andi   $k0, 0xffffd       # Clear EXL bit
         ori    $k0, 0x1           # Enable interrupts
         mtc0    $k0, $12

         lw      $a0, save0        # Restore registers
         lw      $a1, save1
         mov    $at, $k1

         eret                  # Return to EPC

         .kdata
save0:   .word 0
save1:   .word 0

```

**Elaboration:** On real MIPS processors, the return from an exception handler is more complex. The exception handler cannot always jump to the instruction following EPC. For example, if the instruction that caused the exception was in a branch instruction's delay slot (see Chapter 4), the next instruction to execute may not be the following instruction in memory.

## A.8

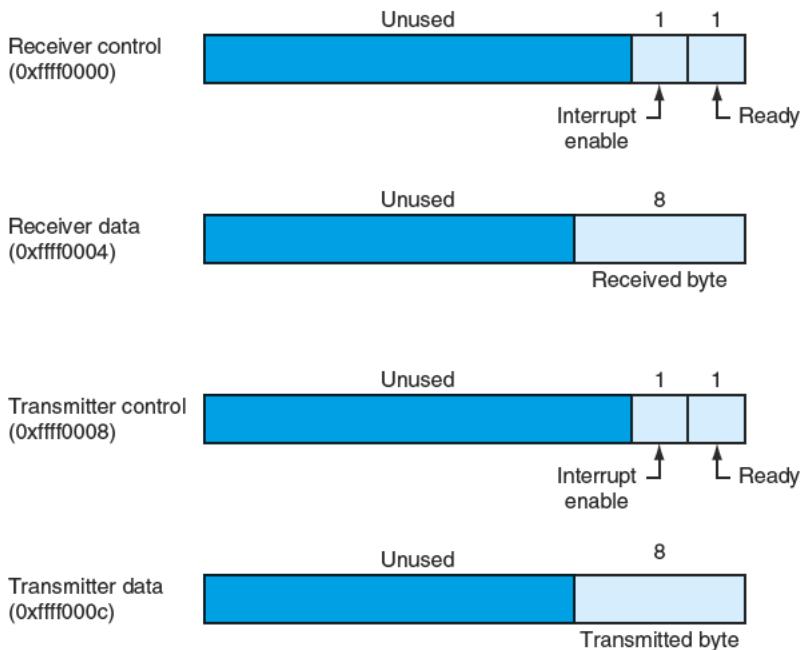
## Input and Output

SPIM simulates one I/O device: a memory-mapped console on which a program can read and write characters. When a program is running, SPIM connects its own terminal (or a separate console window in the X-window version `xspim` or the Windows version `PCSpim`) to the processor. A MIPS program running on SPIM can read the characters that you type. In addition, if the MIPS program writes characters to the terminal, they appear on SPIM's terminal or console window. One exception to this rule is control-C: this character is not passed to the program, but instead causes SPIM to stop and return to command mode. When the program stops running (for example, because you typed control-C or because the program hit a breakpoint), the terminal is reconnected to SPIM so you can type SPIM commands.

To use memory-mapped I/O (see below), `spim` or `xspim` must be started with the `-mapped_io` flag. `PCSpim` can enable memory-mapped I/O through a command line flag or the “Settings” dialog.

The terminal device consists of two independent units: a *receiver* and a *transmitter*. The receiver reads characters from the keyboard. The transmitter displays characters on the console. The two units are completely independent. This means, for example, that characters typed at the keyboard are not automatically echoed on the display. Instead, a program echoes a character by reading it from the receiver and writing it to the transmitter.

A program controls the terminal with four memory-mapped device registers, as shown in [Figure A.8.1](#). “Memory-mapped” means that each register appears as a special memory location. The *Receiver Control register* is at location  $ffff0000_{hex}$ . Only two of its bits are actually used. Bit 0 is called “ready”: if it is 1, it means that a character has arrived from the keyboard but has not yet been read from the Receiver Data register. The ready bit is read-only: writes to it are ignored. The ready bit changes from 0 to 1 when a character is typed at the keyboard, and it changes from 1 to 0 when the character is read from the Receiver Data register.



**FIGURE A.8.1 The terminal Is controlled by four device registers, each of which appears as a memory location at the given address.** Only a few bits of these registers are actually used. The others always read as 0s and are ignored on writes.

Bit 1 of the Receiver Control register is the keyboard “interrupt enable.” This bit may be both read and written by a program. The interrupt enable is initially 0. If it is set to 1 by a program, the terminal requests an interrupt at hardware level 1 whenever a character is typed, and the ready bit becomes 1. However, for the interrupt to affect the processor, interrupts must also be enabled in the Status register (see Section A.7). All other bits of the Receiver Control register are unused.

The second terminal device register is the *Receiver Data register* (at address  $\text{fffff0004}_{\text{hex}}$ ). The low-order eight bits of this register contain the last character typed at the keyboard. All other bits contain 0s. This register is read-only and changes only when a new character is typed at the keyboard. Reading the Receiver Data register resets the ready bit in the Receiver Control register to 0. The value in this register is undefined if the Receiver Control register is 0.

The third terminal device register is the *Transmitter Control register* (at address  $\text{fffff0008}_{\text{hex}}$ ). Only the low-order two bits of this register are used. They behave much like the corresponding bits of the Receiver Control register. Bit 0 is called “ready”

and is read-only. If this bit is 1, the transmitter is ready to accept a new character for output. If it is 0, the transmitter is still busy writing the previous character. Bit 1 is “interrupt enable” and is readable and writable. If this bit is set to 1, then the terminal requests an interrupt at hardware level 0 whenever the transmitter is ready for a new character, and the ready bit becomes 1.

The final device register is the *Transmitter Data register* (at address ffff000c<sub>hex</sub>). When a value is written into this location, its low-order eight bits (i.e., an ASCII character as in Figure 2.15 in Chapter 2) are sent to the console. When the Transmitter Data register is written, the ready bit in the Transmitter Control register is reset to 0. This bit stays 0 until enough time has elapsed to transmit the character to the terminal; then the ready bit becomes 1 again. The Transmitter Data register should only be written when the ready bit of the Transmitter Control register is 1. If the transmitter is not ready, writes to the Transmitter Data register are ignored (the write appears to succeed but the character is not output).

Real computers require time to send characters to a console or terminal. These time lags are simulated by SPIM. For example, after the transmitter starts to write a character, the transmitter’s ready bit becomes 0 for a while. SPIM measures time in instructions executed, not in real clock time. This means that the transmitter does not become ready again until the processor executes a fixed number of instructions. If you stop the machine and look at the ready bit, it will not change. However, if you let the machine run, the bit eventually changes back to 1.

## A.9 SPIM

SPIM is a software simulator that runs assembly language programs written for processors that implement the MIPS-32 architecture, specifically Release 1 of this architecture with a fixed memory mapping, no caches, and only coprocessors 0 and 1.<sup>2</sup> SPIM’s name is just MIPS spelled backwards. SPIM can read and immediately execute assembly language files. SPIM is a self-contained system for running

---

2. Earlier versions of SPIM (before 7.0) implemented the MIPS-1 architecture used in the original MIPS R2000 processors. This architecture is almost a proper subset of the MIPS-32 architecture, with the difference being the manner in which exceptions are handled. MIPS-32 also introduced approximately 60 new instructions, which are supported by SPIM. Programs that ran on the earlier versions of SPIM and did not use exceptions should run unmodified on newer versions of SPIM. Programs that used exceptions will require minor changes.

MIPS programs. It contains a debugger and provides a few operating system-like services. SPIM is much slower than a real computer (100 or more times). However, its low cost and wide availability cannot be matched by real hardware!

An obvious question is, “Why use a simulator when most people have PCs that contain processors that run significantly faster than SPIM?” One reason is that the processors in PCs are Intel 80×86s, whose architecture is far less regular and far more complex to understand and program than MIPS processors. The MIPS architecture may be the epitome of a simple, clean RISC machine.

In addition, simulators can provide a better environment for assembly programming than an actual machine because they can detect more errors and provide a better interface than can an actual computer.

Finally, simulators are useful tools in studying computers and the programs that run on them. Because they are implemented in software, not silicon, simulators can be examined and easily modified to add new instructions, build new systems such as multiprocessors, or simply collect data.

## Simulation of a Virtual Machine

The basic MIPS architecture is difficult to program directly because of delayed branches, delayed loads, and restricted address modes. This difficulty is tolerable since these computers were designed to be programmed in high-level languages and present an interface designed for compilers rather than assembly language programmers. A good part of the programming complexity results from delayed instructions. A *delayed branch* requires two cycles to execute (see the *Elaborations* on pages 284 and 322 of Chapter 4). In the second cycle, the instruction immediately following the branch executes. This instruction can perform useful work that normally would have been done before the branch. It can also be a *nop* (no operation) that does nothing. Similarly, *delayed loads* require two cycles to bring a value from memory, so the instruction immediately following a load cannot use the value (see Section 4.2 of Chapter 4).

MIPS wisely chose to hide this complexity by having its assembler implement a **virtual machine**. This virtual computer appears to have nondelayed branches and loads and a richer instruction set than the actual hardware. The assembler *reorganizes* (rearranges) instructions to fill the delay slots. The virtual computer also provides *pseudoinstructions*, which appear as real instructions in assembly language programs. The hardware, however, knows nothing about pseudoinstructions, so the assembler must translate them into equivalent sequences of actual machine instructions. For example, the MIPS hardware only provides instructions to branch when a register is equal to or not equal to 0. Other conditional branches, such as one that branches when one register is greater than another, are synthesized by comparing the two registers and branching when the result of the comparison is true (nonzero).

### virtual machine

A virtual computer that appears to have nondelayed branches and loads and a richer instruction set than the actual hardware.

By default, SPIM simulates the richer virtual machine, since this is the machine that most programmers will find useful. However, SPIM can also simulate the delayed branches and loads in the actual hardware. Below, we describe the virtual machine and only mention in passing features that do not belong to the actual hardware. In doing so, we follow the convention of MIPS assembly language programmers (and compilers), who routinely use the extended machine as if it was implemented in silicon.

## Getting Started with SPIM

The rest of this appendix introduces SPIM and the MIPS R2000 Assembly language. Many details should never concern you; however, the sheer volume of information can sometimes obscure the fact that SPIM is a simple, easy-to-use program. This section starts with a quick tutorial on using SPIM, which should enable you to load, debug, and run simple MIPS programs.

SPIM comes in different versions for different types of computer systems. The one constant is the simplest version, called `spim`, which is a command-line-driven program that runs in a console window. It operates like most programs of this type: you type a line of text, hit the `return` key, and `spim` executes your command. Despite its lack of a fancy interface, `spim` can do everything that its fancy cousins can do.

There are two fancy cousins to `spim`. The version that runs in the X-windows environment of a UNIX or Linux system is called `xspim`. `xspim` is an easier program to learn and use than `spim`, because its commands are always visible on the screen and because it continually displays the machine's registers and memory. The other fancy version is called `pcSpim` and runs on Microsoft Windows. The UNIX and Windows versions of [SPIM](#) are available online at the publisher's companion Web site for this book. Tutorials on `xspim`, `pcSpim`, `spim`, and [SPIM command-line options](#) are also online.

If you are going to run SPIM on a PC running Microsoft Windows, you should first look at the tutorial on [PCSpim](#) on the companion Web site. If you are going to run SPIM on a computer running UNIX or Linux, you should read the tutorial on [xspim](#).

## Surprising Features

Although SPIM faithfully simulates the MIPS computer, SPIM is a simulator, and certain things are not identical to an actual computer. The most obvious differences are that instruction timing and the memory systems are not identical. SPIM does not simulate caches or memory latency, nor does it accurately reflect floating-point operation or multiply and divide instruction delays. In addition, the floating-point instructions do not detect many error conditions, which would cause exceptions on a real machine.

Another surprise (which occurs on the real machine as well) is that a pseudo-instruction expands to several machine instructions. When you single-step or examine memory, the instructions that you see are different from the source program. The correspondence between the two sets of instructions is fairly simple, since SPIM does not reorganize instructions to fill slots.

## Byte Order

Processors can number bytes within a word so the byte with the lowest number is either the leftmost or rightmost one. The convention used by a machine is called its *byte order*. MIPS processors can operate with either *big-endian* or *little-endian* byte order. For example, in a big-endian machine, the directive `.byte 0, 1, 2, 3` would result in a memory word containing

Byte #			
0	1	2	3

while in a little-endian machine, the word would contain

Byte #			
3	2	1	0

SPIM operates with both byte orders. SPIM's byte order is the same as the byte order of the underlying machine that runs the simulator. For example, on an Intel 80x86, SPIM is little-endian, while on a Macintosh or Sun SPARC, SPIM is big-endian.

## System Calls

SPIM provides a small set of operating system-like services through the system call (`syscall`) instruction. To request a service, a program loads the system call code (see [Figure A.9.1](#)) into register `$v0` and arguments into registers `$a0-$a3` (or `$f12` for floating-point values). System calls that return values put their results in register `$v0` (or `$f0` for floating-point results). For example, the following code prints "the answer = 5":

```
.data
str:
.asciiiz "the answer = "
.text
```

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$v0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

**FIGURE A.9.1 System services.**

```

    li      $v0, 4    # system call code for print_str
    la      $a0, str  # address of string to print
    syscall          # print the string

    li      $v0, 1    # system call code for print_int
    li      $a0, 5    # integer to print
    syscall          # print it

```

The `print_int` system call is passed an integer and prints it on the console. `print_float` prints a single floating-point number; `print_double` prints a double precision number; and `print_string` is passed a pointer to a null-terminated string, which it writes to the console.

The system calls `read_int`, `read_float`, and `read_double` to read an entire line of input up to and including the newline. Characters following the number are ignored. `read_string` has the same semantics as the UNIX library routine `fgets`. It reads up to  $n - 1$  characters into a buffer and terminates the string with a null byte. If fewer than  $n - 1$  characters are on the current line, `read_string` reads up to and including the newline and again null-terminates the string.

*Warning:* Programs that use these syscalls to read from the terminal should not use memory-mapped I/O (see Section A.8).

`sbrk` returns a pointer to a block of memory containing  $n$  additional bytes. `exit` stops the program SPIM is running. `exit2` terminates the SPIM program, and the argument to `exit2` becomes the value returned when the SPIM simulator itself terminates.

`print_char` and `read_char` write and read a single character. `open`, `read`, `write`, and `close` are the standard UNIX library calls.

## A.10 MIPS R2000 Assembly Language

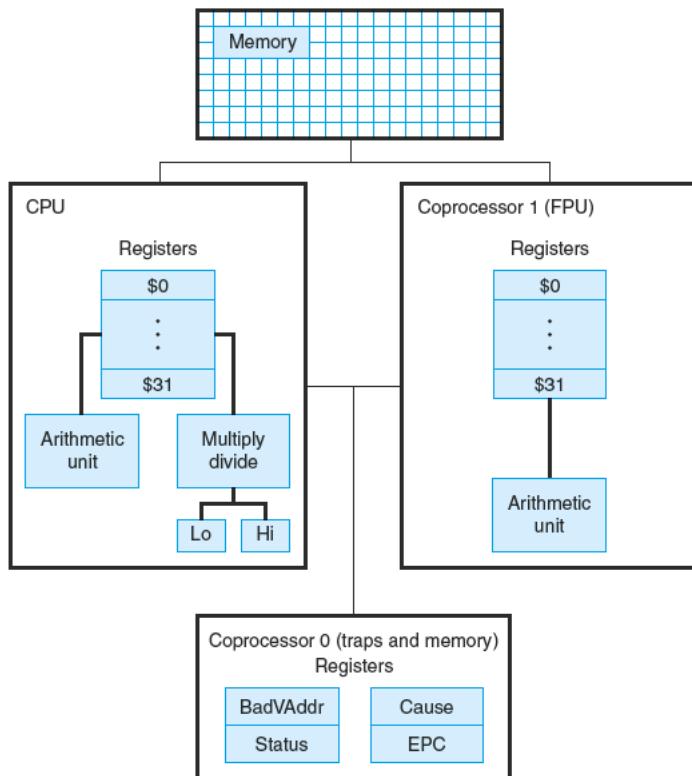
A MIPS processor consists of an integer processing unit (the CPU) and a collection of coprocessors that perform ancillary tasks or operate on other types of data, such as floating-point numbers (see [Figure A.10.1](#)). SPIM simulates two coprocessors. Coprocessor 0 handles exceptions and interrupts. Coprocessor 1 is the floating-point unit. SPIM simulates most aspects of this unit.

### Addressing Modes

MIPS is a load store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers. The bare machine provides only one memory-addressing mode:  $c(rx)$ , which uses the sum of the immediate  $c$  and register  $rx$  as the address. The virtual machine provides the following addressing modes for load and store instructions:

Format	Address computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
label	address of label
label $\pm$ imm	address of label + or - immediate
label $\pm$ imm (register)	address of label + or - (immediate + contents of register)

Most load and store instructions operate only on aligned data. A quantity is *aligned* if its memory address is a multiple of its size in bytes. Therefore, a halfword



**FIGURE A.10.1 MIPS R2000 CPU and FPU.**

object must be stored at even addresses, and a full word object must be stored at addresses that are a multiple of four. However, MIPS provides some instructions to manipulate unaligned data (`lw1`, `lwr`, `sw1`, and `swr`).

**Elaboration:** The MIPS assembler (and SPIM) synthesizes the more complex addressing modes by producing one or more instructions before the load or store to compute a complex address. For example, suppose that the label `table` referred to memory location `0x10000004` and a program contained the instruction

```
ld $a0, table + 4($a1)
```

The assembler would translate this instruction into the instructions

```

lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)

```

The first instruction loads the upper bits of the label's address into register `$at`, which is the register that the assembler reserves for its own use. The second instruction adds the contents of register `$a1` to the label's partial address. Finally, the load instruction uses the hardware address mode to add the sum of the lower bits of the label's address and the offset from the original instruction to the value in register `$at`.

## Assembler Syntax

Comments in assembler files begin with a sharp sign (`#`). Everything from the sharp sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (`_`), and dots (`.`) that do not begin with a number. Instruction opcodes are reserved words that *cannot* be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```

.data
item: .word 1
.text
.globl main      # Must be global
main: lw         $t0, item

```

Numbers are base 10 by default. If they are preceded by `0x`, they are interpreted as hexadecimal. Hence, 256 and `0x100` denote the same value.

Strings are enclosed in double quotes (`"`). Special characters in strings follow the C convention:

- newline `\n`
- tab `\t`
- quote `\"`

SPIM supports a subset of the MIPS assembler directives:

<code>.align n</code>	Align the next datum on a $2^n$ byte boundary. For example, <code>.align 2</code> aligns the next value on a word boundary. <code>.align 0</code> turns off automatic alignment of <code>.half</code> , <code>.word</code> , <code>.float</code> , and <code>.double</code> directives until the next <code>.data</code> or <code>.kdata</code> directive.
<code>.ascii str</code>	Store the string <code>str</code> in memory, but do not null-terminate it.

.ascii str	Store the string <i>str</i> in memory and null-terminate it.
.byte b1, ..., bn	Store the <i>n</i> values in successive bytes of memory.
.data <addr>	Subsequent items are stored in the data segment. If the optional argument <i>addr</i> is present, subsequent items are stored starting at address <i>addr</i> .
.double d1, ..., dn	Store the <i>n</i> floating-point double precision numbers in successive memory locations.
.extern sym size	Declare that the datum stored at <i>sym</i> is <i>size</i> bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register \$gp.
.float f1, ..., fn	Store the <i>n</i> floating-point single precision numbers in successive memory locations.
.globl sym	Declare that label <i>sym</i> is global and can be referenced from other files.
.half h1, ..., hn	Store the <i>n</i> 16-bit quantities in successive memory halfwords.
.kdata <addr>	Subsequent data items are stored in the kernel data segment. If the optional argument <i>addr</i> is present, subsequent items are stored starting at address <i>addr</i> .
.ktext <addr>	Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument <i>addr</i> is present, subsequent items are stored starting at address <i>addr</i> .
.set noat and .set at	The first directive prevents SPIM from complaining about subsequent instructions that use register \$at. The second directive re-enables the warning. Since pseudoinstructions expand into code that uses register \$at, programmers must be very careful about leaving values in this register.
.space n	Allocates <i>n</i> bytes of space in the current segment (which must be the data segment in SPIM).

.text <addr>

Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.

.word w1, ..., wn

Store the *n* 32-bit quantities in successive memory words.

SPIM does not distinguish various parts of the data segment (.data, .rdata, and .sdata).

## Encoding MIPS Instructions

Figure A.10.2 explains how a MIPS instruction is encoded in a binary number. Each column contains instruction encodings for a field (a contiguous group of bits) from an instruction. The numbers at the left margin are values for a field. For example, the j opcode has a value of 2 in the opcode field. The text at the top of a column names a field and specifies which bits it occupies in an instruction. For example, the op field is contained in bits 26–31 of an instruction. This field encodes most instructions. However, some groups of instructions use additional fields to distinguish related instructions. For example, the different floating-point instructions are specified by bits 0–5. The arrows from the first column show which opcodes use these additional fields.

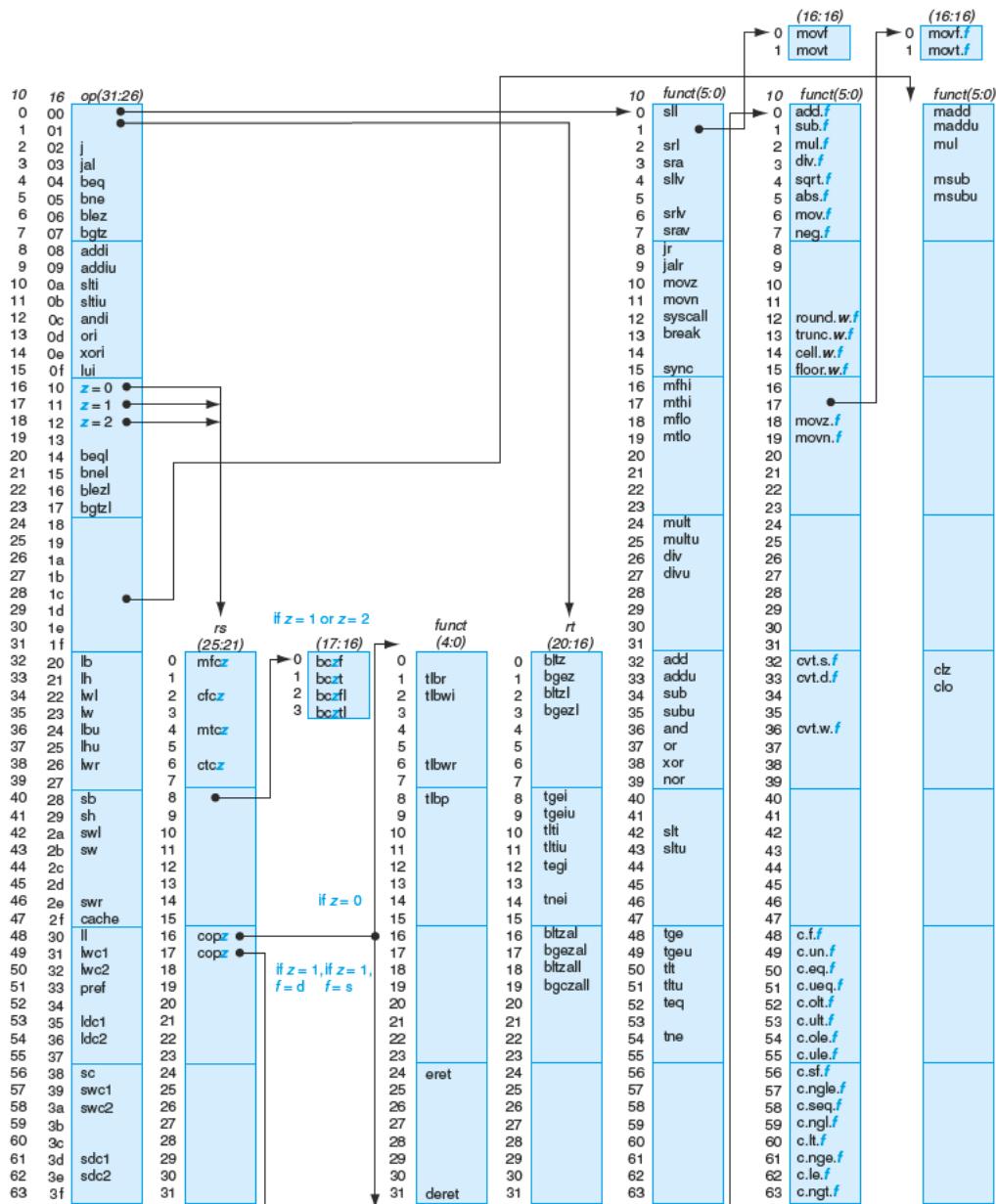
## Instruction Format

The rest of this appendix describes both the instructions implemented by actual MIPS hardware and the pseudoinstructions provided by the MIPS assembler. The two types of instructions are easily distinguished. Actual instructions depict the fields in their binary representation. For example, in

### Addition (with overflow)

	0	rs	rt	rd	0	0x20
add rd, rs, rt	6	5	5	5	5	6

the add instruction consists of six fields. Each field's size in bits is the small number below the field. This instruction begins with six bits of 0s. Register specifiers begin with an *r*, so the next field is a 5-bit register specifier called rs. This is the same register that is the second argument in the symbolic assembly at the left of this line. Another common field is imm<sub>16</sub>, which is a 16-bit immediate number.



**FIGURE A.10.2 MIPS opcode map.** The values of each field are shown to its left. The first column shows the values in base 10, and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for six op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by pointers. The last field (funct) uses “*f*” to mean “s” if rs = 16 and op = 17 or “d” if rs = 17 and op = 17. The second field (rs) uses “*z*” to mean “0”, “1”, “2”, or “3” if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if *z* = 0, the operations are specified in the fourth field (bits 4 to 0); if *z* = 1, then the operations are in the last field with *f* = s. If rs = 17 and *z* = 1, then the operations are in the last field with *f* = d.

Pseudoinstructions follow roughly the same conventions, but omit instruction encoding information. For example:

#### Multiply (without overflow)

`mul rdest, rsrcl, src2      pseudoinstruction`

In pseudoinstructions, `rdest` and `rsrcl` are registers and `src2` is either a register or an immediate value. In general, the assembler and SPIM translate a more general form of an instruction (e.g., `add $v1, $a0, 0x55`) to a specialized form (e.g., `addi $v1, $a0, 0x55`).

## Arithmetic and Logical Instructions

#### Absolute value

`abs rdest, rsrc      pseudoinstruction`

Put the absolute value of register `rsrc` in register `rdest`.

#### Addition (with overflow)

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

add rd, rs, rt

#### Addition (without overflow)

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

addu rd, rs, rt

Put the sum of registers `rs` and `rt` into register `rd`.

#### Addition immediate (with overflow)

8	rs	rt	imm
6	5	5	16

addi rt, rs, imm

#### Addition immediate (without overflow)

9	rs	rt	imm
6	5	5	16

addiu rt, rs, imm

Put the sum of register `rs` and the sign-extended immediate into register `rt`.

**AND**

and rd, rs, rt	<table border="1"> <tr> <td>0</td><td>rs</td><td>rt</td><td>rd</td><td>0</td><td>0x24</td></tr> </table>	0	rs	rt	rd	0	0x24
0	rs	rt	rd	0	0x24		
	6      5      5      5      5      6						

Put the logical AND of registers rs and rt into register rd.

**AND immediate**

andi rt, rs, imm	<table border="1"> <tr> <td>0xc</td><td>rs</td><td>rt</td><td>imm</td></tr> </table>	0xc	rs	rt	imm
0xc	rs	rt	imm		
	6      5      5      16				

Put the logical AND of register rs and the zero-extended immediate into register rt.

**Count leading ones**

clz rd, rs	<table border="1"> <tr> <td>0x1c</td><td>rs</td><td>0</td><td>rd</td><td>0</td><td>0x21</td></tr> </table>	0x1c	rs	0	rd	0	0x21
0x1c	rs	0	rd	0	0x21		
	6      5      5      5      5      6						

**Count leading zeros**

clz rd, rs	<table border="1"> <tr> <td>0x1c</td><td>rs</td><td>0</td><td>rd</td><td>0</td><td>0x20</td></tr> </table>	0x1c	rs	0	rd	0	0x20
0x1c	rs	0	rd	0	0x20		
	6      5      5      5      5      6						

Count the number of leading ones (zeros) in the word in register rs and put the result into register rd. If a word is all ones (zeros), the result is 32.

**Divide (with overflow)**

div rs, rt	<table border="1"> <tr> <td>0</td><td>rs</td><td>rt</td><td>0</td><td>0x1a</td></tr> </table>	0	rs	rt	0	0x1a
0	rs	rt	0	0x1a		
	6      5      5      10      6					

**Divide (without overflow)**

divu rs, rt	<table border="1"> <tr> <td>0</td><td>rs</td><td>rt</td><td>0</td><td>0x1b</td></tr> </table>	0	rs	rt	0	0x1b
0	rs	rt	0	0x1b		
	6      5      5      10      6					

Divide register rs by register rt. Leave the quotient in register lo and the remainder in register hi. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

**Divide (with overflow)**

`div rdest, rsrcl, src2`      *pseudoinstruction*

**Divide (without overflow)**

`divu rdest, rsrcl, src2`      *pseudoinstruction*

Put the quotient of register rsrcl and src2 into register rdest.

**Multiply**

<code>mult rs, rt</code>	0	rs	rt	0	0x18
	6	5	5	10	6

**Unsigned multiply**

<code>multu rs, rt</code>	0	rs	rt	0	0x19
	6	5	5	10	6

Multiply registers rs and rt. Leave the low-order word of the product in register lo and the high-order word in register hi.

**Multiply (without overflow)**

<code>mul rd, rs, rt</code>	0x1c	rs	rt	rd	0	2
	6	5	5	5	5	6

Put the low-order 32 bits of the product of rs and rt into register rd.

**Multiply (with overflow)**

`mulo rdest, rsrcl, src2`      *pseudoinstruction*

**Unsigned multiply (with overflow)**

`mulou rdest, rsrcl, src2`      *pseudoinstruction*

Put the low-order 32 bits of the product of register rsrcl and src2 into register rdest.

## Multiply add

madd rs, rt	0x1c	rs	rt	0	0
	6	5	5	10	6

## **Unsigned multiply add**

maddu rs, rt	0x1c	rs	rt	0	1
	6	5	5	10	6

Multiply registers  $rs$  and  $rt$  and add the resulting 64-bit product to the 64-bit value in the concatenated registers  $lo$  and  $hi$ .

## Multiply subtract

msub rs, rt	0x1c	rs	rt	0	4
	6	5	5	10	6

## **Unsigned multiply subtract**

msub	rs	rt	0	5
	6	5	10	6

Multiply registers `rs` and `rt` and subtract the resulting 64-bit product from the 64-bit value in the concatenated registers `lo` and `hi`.

### **Negate value (with overflow)**

`neg rdest, rsrc`      *pseudoinstruction*

### Negate value (without overflow)

Put the negative of register rs<sub>RC</sub> into register r<sub>DEST</sub>.

NOR

nor	rd,	rs,	rt	0	rs	rt	rd	0	0x27
	6	5	5	5	5	5	6	6	

Put the logical NOR of registers rs and rt into register rd.

**NOT**

`not rdest, rsrc`      *pseudoinstruction*

Put the bitwise logical negation of register `rsrc` into register `rdest`.

**OR**

<code>or rd, rs, rt</code>	0	rs	rt	rd	0	0x25
	6	5	5	5	5	6

Put the logical OR of registers `rs` and `rt` into register `rd`.

**OR immediate**

<code>ori rt, rs, imm</code>	Oxd	rs	rt	imm	
	6	5	5	16	

Put the logical OR of register `rs` and the zero-extended immediate into register `rt`.

**Remainder**

`rem rdest, rsrc1, rsrc2`      *pseudoinstruction*

**Unsigned remainder**

`remu rdest, rsrc1, rsrc2`      *pseudoinstruction*

Put the remainder of register `rsrc1` divided by register `rsrc2` into register `rdest`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

**Shift left logical**

<code>sll rd, rt, shamt</code>	0	rs	rt	rd	shamt	0
	6	5	5	5	5	6

**Shift left logical variable**

<code>sllv rd, rt, rs</code>	0	rs	rt	rd	0	4
	6	5	5	5	5	6

**Shift right arithmetic**

sra rd, rt, shamt

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

**Shift right arithmetic variable**

srav rd, rt, rs

0	rs	rt	rd	0	7
6	5	5	5	5	6

**Shift right logical**

srl rd, rt, shamt

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

**Shift right logical variable**

srlv rd, rt, rs

0	rs	rt	rd	0	6
6	5	5	5	5	6

Shift register rt left (right) by the distance indicated by immediate sham or the register rs and put the result in register rd. Note that argument rs is ignored for sll, sra, and srl.

**Rotate left**rol rdest, rsrcl, rsrc2      *pseudoinstruction***Rotate right**ror rdest, rsrcl, rsrc2      *pseudoinstruction*

Rotate register rsrcl left (right) by the distance indicated by rsrc2 and put the result in register rdest.

**Subtract (with overflow)**

sub rd, rs, rt

0	rs	rt	rd	0	0x22
6	5	5	5	5	6

**Subtract (without overflow)**

subu rd, rs, rt	0	rs	rt	rd	0	0x23
	6	5	5	5	5	6

Put the difference of registers rs and rt into register rd.

**Exclusive OR**

xor rd, rs, rt	0	rs	rt	rd	0	0x26
	6	5	5	5	5	6

Put the logical XOR of registers rs and rt into register rd.

**XOR immediate**

xori rt, rs, imm	Oxe	rs	rt	Imm		
	6	5	5	16		

Put the logical XOR of register rs and the zero-extended immediate into register rt.

**Constant-Manipulating Instructions****Load upper immediate**

lui rt, imm	0xf	0	rt	imm		
	6	5	5	16		

Load the lower halfword of the immediate imm into the upper halfword of register rt. The lower bits of the register are set to 0.

**Load immediate**

li rdest, imm *pseudoinstruction*

Move the immediate imm into register rdest.

**Comparison Instructions****Set less than**

slt rd, rs, rt	0	rs	rt	rd	0	0x2a
	6	5	5	5	5	6

**Set less than unsigned**

sltu rd, rs, rt	0	rs	rt	rd	0	0x2b
	6	5	5	5	5	6

Set register rd to 1 if register rs is less than rt, and to 0 otherwise.

**Set less than immediate**

slti rt, rs, imm	0xa	rs	rt	imm
	6	5	5	16

**Set less than unsigned immediate**

sltiu rt, rs, imm	0xb	rs	rt	imm
	6	5	5	16

Set register rt to 1 if register rs is less than the sign-extended immediate, and to 0 otherwise.

**Set equal**

seq rdest, rsrcl, rsrc2                   *pseudoinstruction*

Set register rdest to 1 if register rsrcl equals rsrc2, and to 0 otherwise.

**Set greater than equal**

sge rdest, rsrcl, rsrc2                   *pseudoinstruction*

**Set greater than equal unsigned**

sgeu rdest, rsrcl, rsrc2                   *pseudoinstruction*

Set register rdest to 1 if register rsrcl is greater than or equal to rsrc2, and to 0 otherwise.

**Set greater than**

sgt rdest, rsrcl, rsrc2                   *pseudoinstruction*

**Set greater than unsigned**

```
sgtu rdest, rsrc1, rsrc2      pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is greater than `rsrc2`, and to 0 otherwise.

**Set less than equal**

```
sle rdest, rsrc1, rsrc2      pseudoinstruction
```

**Set less than equal unsigned**

```
sieu rdest, rsrc1, rsrc2      pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is less than or equal to `rsrc2`, and to 0 otherwise.

**Set not equal**

```
sne rdest, rsrc1, rsrc2      pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is not equal to `rsrc2`, and to 0 otherwise.

## Branch Instructions

Branch instructions use a signed 16-bit instruction *offset* field; hence, they can jump  $2^{15} - 1$  *instructions* (not bytes) forward or  $2^{15}$  instructions backward. The *jump* instruction contains a 26-bit address field. In actual MIPS processors, branch instructions are delayed branches, which do not transfer control until the instruction following the branch (its “delay slot”) has executed (see Chapter 4). Delayed branches affect the offset calculation, since it must be computed relative to the address of the delay slot instruction ( $\text{PC} + 4$ ), which is when the branch occurs. SPIM does not simulate this delay slot, unless the `-bare` or `-delayed_branch` flags are specified.

In assembly code, offsets are not usually specified as numbers. Instead, an instruction branches to a label, and the assembler computes the distance between the branch and the target instructions.

In MIPS-32, all actual (not pseudo) conditional branch instructions have a “likely” variant (for example, `beq`’s likely variant is `beql`), which does *not* execute the instruction in the branch’s delay slot if the branch is not taken. Do not use

these instructions; they may be removed in subsequent versions of the architecture. SPIM implements these instructions, but they are not described further.

#### **Branch instruction**

b label

*pseudoinstruction*

Unconditionally branch to the instruction at the label.

#### **Branch coprocessor false**

bclf cc label

0x11	8	cc	0	Offset
6	5	3	2	16

#### **Branch coprocessor true**

bclt cc label

0x11	8	cc	1	Offset
6	5	3	2	16

Conditionally branch the number of instructions specified by the offset if the floating-point coprocessor's condition flag numbered *cc* is false (true). If *cc* is omitted from the instruction, condition code flag 0 is assumed.

#### **Branch on equal**

beq rs, rt, label

4	rs	rt	Offset
6	5	5	16

Conditionally branch the number of instructions specified by the offset if register *rs* equals *rt*.

#### **Branch on greater than equal zero**

bgez rs, label

1	rs	1	Offset
6	5	5	16

Conditionally branch the number of instructions specified by the offset if register *rs* is greater than or equal to 0.

**Branch on greater than equal zero and link**

bgezal rs, label	1 6	rs 5	0x11 5	Offset 16
------------------	--------	---------	-----------	--------------

Conditionally branch the number of instructions specified by the offset if register rs is greater than or equal to 0. Save the address of the next instruction in register 31.

**Branch on greater than zero**

bgtz rs, label	7 6	rs 5	0 5	Offset 16
----------------	--------	---------	--------	--------------

Conditionally branch the number of instructions specified by the offset if register rs is greater than 0.

**Branch on less than equal zero**

blez rs, label	6 6	rs 5	0 5	Offset 16
----------------	--------	---------	--------	--------------

Conditionally branch the number of instructions specified by the offset if register rs is less than or equal to 0.

**Branch on less than and link**

bltzal rs, label	1 6	rs 5	0x10 5	Offset 16
------------------	--------	---------	-----------	--------------

Conditionally branch the number of instructions specified by the offset if register rs is less than 0. Save the address of the next instruction in register 31.

**Branch on less than zero**

bltz rs, label	1 6	rs 5	0 5	Offset 16
----------------	--------	---------	--------	--------------

Conditionally branch the number of instructions specified by the offset if register rs is less than 0.

**Branch on not equal**

bne rs, rt, label	<table border="1"> <tr> <td>5</td><td>rs</td><td>rt</td><td>Offset</td></tr> </table>	5	rs	rt	Offset
5	rs	rt	Offset		
	6        5        5        16				

Conditionally branch the number of instructions specified by the offset if register rs is not equal to rt.

**Branch on equal zero**

beqz rsrc, label *pseudoinstruction*

Conditionally branch to the instruction at the label if rsrc equals 0.

**Branch on greater than equal**

bge rsrc1, rsrc2, label *pseudoinstruction*

**Branch on greater than equal unsigned**

bgeu rsrc1, rsrc2, label *pseudoinstruction*

Conditionally branch to the instruction at the label if register rsrc1 is greater than or equal to rsrc2.

**Branch on greater than**

bgt rsrc1, src2, label *pseudoinstruction*

**Branch on greater than unsigned**

bgtu rsrc1, src2, label *pseudoinstruction*

Conditionally branch to the instruction at the label if register rsrc1 is greater than src2.

**Branch on less than equal**

ble rsrc1, src2, label *pseudoinstruction*

**Branch on less than equal unsigned**

```
bleu rsrc1, src2, label      pseudoinstruction
```

Conditionally branch to the instruction at the label if register `rsrc1` is less than or equal to `src2`.

**Branch on less than**

```
blt rsrc1, rsrc2, label      pseudoinstruction
```

**Branch on less than unsigned**

```
bltu rsrc1, rsrc2, label     pseudoinstruction
```

Conditionally branch to the instruction at the label if register `rsrc1` is less than `rsrc2`.

**Branch on not equal zero**

```
bnez rsrc, label           pseudoinstruction
```

Conditionally branch to the instruction at the label if register `rsrc` is not equal to 0.

**Jump Instructions****Jump**

j target	2	target
	6	26

Unconditionally jump to the instruction at target.

**Jump and link**

jal target	3	target
	6	26

Unconditionally jump to the instruction at target. Save the address of the next instruction in register `$ra`.

**Jump and link register**

jalr rs, rd	0	rs	0	rd	0	9
	6	5	5	5	5	6

Unconditionally jump to the instruction whose address is in register rs. Save the address of the next instruction in register rd (which defaults to 31).

**Jump register**

jr rs	0	rs	0	8
	6	5	15	6

Unconditionally jump to the instruction whose address is in register rs.

**Trap Instructions****Trap if equal**

teq rs, rt	0	rs	rt	0	0x34
	6	5	5	10	6

If register rs is equal to register rt, raise a Trap exception.

**Trap if equal immediate**

teqi rs, imm	1	rs	Oxc	imm
	6	5	5	16

If register rs is equal to the sign-extended value imm, raise a Trap exception.

**Trap if not equal**

teq rs, rt	0	rs	rt	0	0x36
	6	5	5	10	6

If register rs is not equal to register rt, raise a Trap exception.

**Trap if not equal immediate**

teqi rs, imm	1	rs	Oxe	imm
	6	5	5	16

If register rs is not equal to the sign-extended value imm, raise a Trap exception.

**Trap if greater equal**

tge rs, rt	0	rs	rt	0	0x30
	6	5	5	10	6

**Unsigned trap if greater equal**

tgeu rs, rt	0	rs	rt	0	0x31
	6	5	5	10	6

If register rs is greater than or equal to register rt, raise a Trap exception.

**Trap if greater equal immediate**

tgei rs, imm	1	rs	8	imm	
	6	5	5	16	

**Unsigned trap if greater equal immediate**

tgeiu rs, imm	1	rs	9	imm	
	6	5	5	16	

If register rs is greater than or equal to the sign-extended value imm, raise a Trap exception.

**Trap if less than**

tlr rs, rt	0	rs	rt	0	0x32
	6	5	5	10	6

**Unsigned trap if less than**

tlru rs, rt	0	rs	rt	0	0x33
	6	5	5	10	6

If register rs is less than register rt, raise a Trap exception.

**Trap if less than immediate**

tlri rs, imm	1	rs	a	imm	
	6	5	5	16	

**Unsigned trap if less than immediate**

tltru rs, imm	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 40px; height: 15px;"></td></tr><tr><td style="text-align: center;">1</td><td style="text-align: center;">rs</td><td style="text-align: center;">b</td><td style="text-align: center;">imm</td></tr></table>					1	rs	b	imm
1	rs	b	imm						
	6            5            5            16								

If register *rs* is less than the sign-extended value *imm*, raise a Trap exception.

**Load Instructions****Load address**

la rdest, address                                  *pseudoinstruction*

Load computed *address*—not the contents of the location—into register *rdest*.

**Load byte**

lb rt, address	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 40px; height: 15px;"></td></tr><tr><td style="text-align: center;">0x20</td><td style="text-align: center;">rs</td><td style="text-align: center;">rt</td><td style="text-align: center;">Offset</td></tr></table>					0x20	rs	rt	Offset
0x20	rs	rt	Offset						
	6            5            5            16								

**Load unsigned byte**

lbu rt, address	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 40px; height: 15px;"></td></tr><tr><td style="text-align: center;">0x24</td><td style="text-align: center;">rs</td><td style="text-align: center;">rt</td><td style="text-align: center;">Offset</td></tr></table>					0x24	rs	rt	Offset
0x24	rs	rt	Offset						
	6            5            5            16								

Load the byte at *address* into register *rt*. The byte is sign-extended by *lb*, but not by *lbu*.

**Load halfword**

lh rt, address	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 40px; height: 15px;"></td></tr><tr><td style="text-align: center;">0x21</td><td style="text-align: center;">rs</td><td style="text-align: center;">rt</td><td style="text-align: center;">Offset</td></tr></table>					0x21	rs	rt	Offset
0x21	rs	rt	Offset						
	6            5            5            16								

**Load unsigned halfword**

lhu rt, address	<table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 20px; height: 15px;"></td><td style="width: 40px; height: 15px;"></td></tr><tr><td style="text-align: center;">0x25</td><td style="text-align: center;">rs</td><td style="text-align: center;">rt</td><td style="text-align: center;">Offset</td></tr></table>					0x25	rs	rt	Offset
0x25	rs	rt	Offset						
	6            5            5            16								

Load the 16-bit quantity (halfword) at *address* into register *rt*. The halfword is sign-extended by *lh*, but not by *lhu*.

**Load word**

lw rt, address	0x23	rs	rt	Offset
	6	5	5	16

Load the 32-bit quantity (word) at *address* into register *rt*.

**Load word coprocessor 1**

lwcl ft, address	0x31	rs	rt	Offset
	6	5	5	16

Load the word at *address* into register *ft* in the floating-point unit.

**Load word left**

lwl rt, address	0x22	rs	rt	Offset
	6	5	5	16

**Load word right**

lwr rt, address	0x26	rs	rt	Offset
	6	5	5	16

Load the left (right) bytes from the word at the possibly unaligned *address* into register *rt*.

**Load doubleword**

ld rdest, address *pseudoinstruction*

Load the 64-bit quantity at *address* into registers *rdest* and *rdest + 1*.

**Unaligned load halfword**

ulh rdest, address *pseudoinstruction*

**Unaligned load halfword unsigned**

`ulhu rdest, address`      *pseudoinstruction*

Load the 16-bit quantity (halfword) at the possibly unaligned *address* into register *rdest*. The halfword is sign-extended by `ulh`, but not `ulhu`.

**Unaligned load word**

`ulw rdest, address`      *pseudoinstruction*

Load the 32-bit quantity (word) at the possibly unaligned *address* into register *rdest*.

**Load linked**

11	rt, address	0x30	rs	rt	Offset
		6	5	5	16

Load the 32-bit quantity (word) at *address* into register *rt* and start an atomic read-modify-write operation. This operation is completed by a store conditional (sc) instruction, which will fail if another processor writes into the block containing the loaded word. Since SPIM does not simulate multiple processors, the store conditional operation always succeeds.

**Store Instructions****Store byte**

sb	rt, address	0x28	rs	rt	Offset
		6	5	5	16

Store the low byte from register *rt* at *address*.

**Store halfword**

sh	rt, address	0x29	rs	rt	Offset
		6	5	5	16

Store the low halfword from register *rt* at *address*.

**Store word**

sw rt, address	0x2b	rs	rt	Offset
	6	5	5	16

Store the word from register rt at *address*.

**Store word coprocessor 1**

swcl ft, address	0x31	rs	ft	Offset
	6	5	5	16

Store the floating-point value in register ft of floating-point coprocessor at *address*.

**Store double coprocessor 1**

sdc1 ft, address	0x3d	rs	ft	Offset
	6	5	5	16

Store the doubleword floating-point value in registers ft and ft + 1 of floating-point coprocessor at *address*. Register ft must be even numbered.

**Store word left**

swl rt, address	0x2a	rs	rt	Offset
	6	5	5	16

**Store word right**

swr rt, address	0x2e	rs	rt	Offset
	6	5	5	16

Store the left (right) bytes from register rt at the possibly unaligned *address*.

**Store doubleword**

sd rsrc, address *pseudoinstruction*

Store the 64-bit quantity in registers rsrc and rsrc + 1 at *address*.

**Unaligned store halfword**

ush rsrc, address	<i>pseudoinstruction</i>
-------------------	--------------------------

Store the low halfword from register rsrc at the possibly unaligned *address*.

**Unaligned store word**

usw rsrc, address	<i>pseudoinstruction</i>
-------------------	--------------------------

Store the word from register rsrc at the possibly unaligned *address*.

**Store conditional**

sc rt, address	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td>0x38</td> <td>rs</td> <td>rt</td> <td>Offset</td> </tr> <tr> <td style="padding: 2px;">6</td> <td style="padding: 2px;">5</td> <td style="padding: 2px;">5</td> <td style="padding: 2px;">16</td> </tr> </table>	0x38	rs	rt	Offset	6	5	5	16
0x38	rs	rt	Offset						
6	5	5	16						

Store the 32-bit quantity (word) in register rt into memory at *address* and complete an atomic read-modify-write operation. If this atomic operation is successful, the memory word is modified and register rt is set to 1. If the atomic operation fails because another processor wrote to a location in the block containing the addressed word, this instruction does not modify memory and writes 0 into register rt. Since SPIM does not simulate multiple processors, the instruction always succeeds.

**Data Movement Instructions****Move**

move rdest, rsrc	<i>pseudoinstruction</i>
------------------	--------------------------

Move register rsrc to rdest.

**Move from hi**

mfhi rd	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td>0</td> <td>0</td> <td>rd</td> <td>0</td> <td>0x10</td> </tr> <tr> <td style="padding: 2px;">6</td> <td style="padding: 2px;">10</td> <td style="padding: 2px;">5</td> <td style="padding: 2px;">5</td> <td style="padding: 2px;">6</td> </tr> </table>	0	0	rd	0	0x10	6	10	5	5	6
0	0	rd	0	0x10							
6	10	5	5	6							

**Move from lo**

mflo rd	0	0	rd	0	0x12
	6	10	5	5	6

The multiply and divide unit produces its result in two additional registers, hi and lo. These instructions move values to and from these registers. The multiply, divide, and remainder pseudoinstructions that make this unit appear to operate on the general registers move the result after the computation finishes.

Move the hi (lo) register to register rd.

**Move to hi**

mthi rs	0	rs	0	0x11
	6	5	15	6

**Move to lo**

mtlo rs	0	rs	0	0x13
	6	5	15	6

Move register rs to the hi (lo) register.

**Move from coprocessor 0**

mfc0 rt, rd	0x10	0	rt	rd	0
	6	5	5	5	11

**Move from coprocessor 1**

mfcl rt, fs	0x11	0	rt	fs	0
	6	5	5	5	11

Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

Move register rd in a coprocessor (register fs in the FPU) to CPU register rt. The floating-point unit is coprocessor 1.

**Move double from coprocessor 1**

`mfc1.d rdest, frsrc1`      *pseudoinstruction*

Move floating-point registers `frsrc1` and `frsrc1 + 1` to CPU registers `rdest` and `rdest + 1`.

**Move to coprocessor 0**

<code>mtc0 rd, rt</code>	0x10	4	rt	rd	0
	6	5	5	5	11

**Move to coprocessor 1**

<code>mtc1 rd, fs</code>	0x11	4	rt	fs	0
	6	5	5	5	11

Move CPU register `rt` to register `rd` in a coprocessor (register `fs` in the FPU).

**Move conditional not zero**

<code>movn rd, rs, rt</code>	0	rs	rt	rd	0xb
	6	5	5	5	11

Move register `rs` to register `rd` if register `rt` is not 0.

**Move conditional zero**

<code>movz rd, rs, rt</code>	0	rs	rt	rd	0xa
	6	5	5	5	11

Move register `rs` to register `rd` if register `rt` is 0.

**Move conditional on FP false**

<code>movf rd, rs, cc</code>	0	rs	cc	0	rd	0	1
	6	5	3	2	5	5	6

Move CPU register `rs` to register `rd` if FPU condition code flag number `cc` is 0. If `cc` is omitted from the instruction, condition code flag 0 is assumed.

**Move conditional on FP true**

movt rd, rs, cc	0	rs	cc	1	rd	0	1
	6	5	3	2	5	5	6

Move CPU register *rs* to register *rd* if FPU condition code flag number *cc* is 1. If *cc* is omitted from the instruction, condition code bit 0 is assumed.

**Floating-Point Instructions**

The MIPS has a floating-point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating-point numbers. This coprocessor has its own registers, which are numbered \$f0–\$f31. Because these registers are only 32 bits wide, two of them are required to hold doubles, so only floating-point registers with even numbers can hold double precision values. The floating-point coprocessor also has eight condition code (*cc*) flags, numbered 0–7, which are set by compare instructions and tested by branch (*bclf* or *bclt*) and conditional move instructions.

Values are moved in or out of these registers one word (32 bits) at a time by *lwcl*, *swcl*, *mtcl*, and *mfcl* instructions or one double (64 bits) at a time by *ldcl* and *sdcl*, described above, or by the *l.s*, *l.d*, *s.s*, and *s.d* pseudoinstructions described below.

In the actual instructions below, bits 21–26 are 0 for single precision and 1 for double precision. In the pseudoinstructions below, *fdest* is a floating-point register (e.g., \$f2).

**Floating-point absolute value double**

abs.d fd, fs	0x11	1	0	fs	fd	5
	6	5	5	5	5	6

**Floating-point absolute value single**

abs.s fd, fs	0x11	0	0	fs	fd	5
	6	5	5	5	5	6

Compute the absolute value of the floating-point double (single) in register *fs* and put it in register *fd*.

**Floating-point addition double**

add.d fd, fs, ft	0x11	0x11	ft	fs	fd	0
	6	5	5	5	5	6

**Floating-point addition single**

add.s fd, fs, ft	0x11	0x10	ft	fs	fd	0
	6	5	5	5	5	6

Compute the sum of the floating-point doubles (singles) in registers fs and ft and put it in register fd.

**Floating-point ceiling to word**

ceil.w.d fd, fs	0x11	0x11	0	fs	fd	0xe
	6	5	5	5	5	6
ceil.w.s fd, fs	0x11	0x10	0	fs	fd	0xe

Compute the ceiling of the floating-point double (single) in register fs, convert to a 32-bit fixed-point value, and put the resulting word in register fd.

**Compare equal double**

c.eq.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	2
	6	5	5	5	3	2	2	4

**Compare equal single**

c.eq.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	2
	6	5	5	5	3	2	2	4

Compare the floating-point double (single) in register fs against the one in ft and set the floating-point condition flag cc to 1 if they are equal. If cc is omitted, condition code flag 0 is assumed.

**Compare less than equal double**

c.le.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xe
	6	5	5	5	3	2	2	4

**Compare less than equal single**

c.le.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xe
	6	5	5	5	3	2	2	4

Compare the floating-point double (single) in register `fs` against the one in `ft` and set the floating-point condition flag `cc` to 1 if the first is less than or equal to the second. If `cc` is omitted, condition code flag 0 is assumed.

#### Compare less than double

c.lt.d cc fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

#### Compare less than single

c.lt.s cc fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xc
	6	5	5	5	3	2	2	4

Compare the floating-point double (single) in register `fs` against the one in `ft` and set the condition flag `cc` to 1 if the first is less than the second. If `cc` is omitted, condition code flag 0 is assumed.

#### Convert single to double

cvt.d.s fd, fs	0x11	0x10	0	fs	fd	0x21
	6	5	5	5	5	6

#### Convert integer to double

cvt.d.w fd, fs	0x11	0x14	0	fs	fd	0x21
	6	5	5	5	5	6

Convert the single precision floating-point number or integer in register `fs` to a double (single) precision number and put it in register `fd`.

#### Convert double to single

cvt.s.d fd, fs	0x11	0x11	0	fs	fd	0x20
	6	5	5	5	5	6

#### Convert integer to single

cvt.s.w fd, fs	0x11	0x14	0	fs	fd	0x20
	6	5	5	5	5	6

Convert the double precision floating-point number or integer in register `fs` to a single precision number and put it in register `fd`.

**Convert double to integer**

cvt.w.d fd, fs	0x11	0x11	0	fs	fd	0x24
	6	5	5	5	5	6

**Convert single to integer**

cvt.w.s fd, fs	0x11	0x10	0	fs	fd	0x24
	6	5	5	5	5	6

Convert the double or single precision floating-point number in register fs to an integer and put it in register fd.

**Floating-point divide double**

div.d fd, fs, ft	0x11	0x11	ft	fs	fd	3
	6	5	5	5	5	6

**Floating-point divide single**

div.s fd, fs, ft	0x11	0x10	ft	fs	fd	3
	6	5	5	5	5	6

Compute the quotient of the floating-point doubles (singles) in registers fs and ft and put it in register fd.

**Floating-point floor to word**

floor.w.d fd, fs	0x11	0x11	0	fs	fd	0xf
	6	5	5	5	5	6
floor.w.s fd, fs	0x11	0x10	0	fs	fd	0xf

Compute the floor of the floating-point double (single) in register fs and put the resulting word in register fd.

**Load floating-point double**

l.d fdest, address *pseudoinstruction*

**Load floating-point single**

`l.s fdest, address` *pseudoinstruction*

Load the floating-point double (single) at address into register fdest.

**Move floating-point double**

<code>mov.d fd, fs</code>	0x11	0x11	0	fs	fd	6
	6	5	5	5	5	6

**Move floating-point single**

<code>mov.s fd, fs</code>	0x11	0x10	0	fs	fd	6
	6	5	5	5	5	6

Move the floating-point double (single) from register fs to register fd.

**Move conditional floating-point double false**

<code>movf.d fd, fs, cc</code>	0x11	0x11	cc	0	fs	fd	0x11
	6	5	3	2	5	5	6

**Move conditional floating-point single false**

<code>movf.s fd, fs, cc</code>	0x11	0x10	cc	0	fs	fd	0x11
	6	5	3	2	5	5	6

Move the floating-point double (single) from register fs to register fd if condition code flag cc is 0. If cc is omitted, condition code flag 0 is assumed.

**Move conditional floating-point double true**

<code>movt.d fd, fs, cc</code>	0x11	0x11	cc	1	fs	fd	0x11
	6	5	3	2	5	5	6

**Move conditional floating-point single true**

<code>movt.s fd, fs, cc</code>	0x11	0x10	cc	1	fs	fd	0x11
	6	5	3	2	5	5	6

Move the floating-point double (single) from register `fs` to register `fd` if condition code flag `cc` is 1. If `cc` is omitted, condition code flag 0 is assumed.

#### **Move conditional floating-point double not zero**

<code>movn.d fd, fs, rt</code>	0x11	0x11	rt	fs	fd	0x13
	6	5	5	5	5	6

#### **Move conditional floating-point single not zero**

<code>movn.s fd, fs, rt</code>	0x11	0x10	rt	fs	fd	0x13
	6	5	5	5	5	6

Move the floating-point double (single) from register `fs` to register `fd` if processor register `rt` is not 0.

#### **Move conditional floating-point double zero**

<code>movz.d fd, fs, rt</code>	0x11	0x11	rt	fs	fd	0x12
	6	5	5	5	5	6

#### **Move conditional floating-point single zero**

<code>movz.s fd, fs, rt</code>	0x11	0x10	rt	fs	fd	0x12
	6	5	5	5	5	6

Move the floating-point double (single) from register `fs` to register `fd` if processor register `rt` is 0.

#### **Floating-point multiply double**

<code>mul.d fd, fs, ft</code>	0x11	0x11	ft	fs	fd	2
	6	5	5	5	5	6

#### **Floating-point multiply single**

<code>mul.s fd, fs, ft</code>	0x11	0x10	ft	fs	fd	2
	6	5	5	5	5	6

Compute the product of the floating-point doubles (singles) in registers `fs` and `ft` and put it in register `fd`.

#### **Negate double**

<code>neg.d fd, fs</code>	0x11	0x11	0	fs	fd	7
	6	5	5	5	5	6

**Negate single**

neg.s	fd,	fs	0x11	0x10	0	fs	fd	7
			6	5	5	5	5	6

Negate the floating-point double (single) in register fs and put it in register fd.

**Floating-point round to word**

round.w.d	fd,	fs	0x11	0x11	0	fs	fd	0xc
			6	5	5	5	5	6

round.w.s	fd,	fs	0x11	0x10	0	fs	fd	0xc
			6	5	5	5	5	6

Round the floating-point double (single) value in register fs, convert to a 32-bit fixed-point value, and put the resulting word in register fd.

**Square root double**

sqrt.d	fd,	fs	0x11	0x11	0	fs	fd	4
			6	5	5	5	5	6

**Square root single**

sqrt.s	fd,	fs	0x11	0x10	0	fs	fd	4
			6	5	5	5	5	6

Compute the square root of the floating-point double (single) in register fs and put it in register fd.

**Store floating-point double**

s.d fdest, address      *pseudoinstruction*

**Store floating-point single**

s.s fdest, address      *pseudoinstruction*

Store the floating-point double (single) in register fdest at *address*.

**Floating-point subtract double**

sub.d	fd,	fs,	ft	0x11	ft	fs	fd	1
			6	5	5	5	5	6

**Floating-point subtract single**

sub.s fd, fs, ft	0x11	0x10	ft	fs	fd	1
	6	5	5	5	5	6

Compute the difference of the floating-point doubles (singles) in registers fs and ft and put it in register fd.

**Floating-point truncate to word**

trunc.w.d fd, fs	0x11	0x11	0	fs	fd	0xd
	6	5	5	5	5	6
trunc.w.s fd, fs	0x11	0x10	0	fs	fd	0xd

Truncate the floating-point double (single) value in register fs, convert to a 32-bit fixed-point value, and put the resulting word in register fd.

**Exception and Interrupt Instructions****Exception return**

eret	0x10	1	0	0x18
	6	1	19	6

Set the EXL bit in coprocessor 0's Status register to 0 and return to the instruction pointed to by coprocessor 0's EPC register.

**System call**

syscall	0	0	0xc
	6	20	6

Register \$v0 contains the number of the system call (see Figure A.9.1) provided by SPIM.

**Break**

break code	0	code	0xd
	6	20	6

Cause exception code. Exception 1 is reserved for the debugger.

**No operation**

nop	0	0	0	0	0	0
	6	5	5	5	5	6

Do nothing.

## A.11 Concluding Remarks

Programming in assembly language requires a programmer to trade helpful features of high-level languages—such as data structures, type checking, and control constructs—for complete control over the instructions that a computer executes. External constraints on some applications, such as response time or program size, require a programmer to pay close attention to every instruction. However, the cost of this level of attention is assembly language programs that are longer, more time-consuming to write, and more difficult to maintain than high-level language programs.

Moreover, three trends are reducing the need to write programs in assembly language. The first trend is toward the improvement of compilers. Modern compilers produce code that is typically comparable to the best handwritten code—and is sometimes better. The second trend is the introduction of new processors that are not only faster, but in the case of processors that execute multiple instructions simultaneously, also more difficult to program by hand. In addition, the rapid evolution of the modern computer favors high-level language programs that are not tied to a single architecture. Finally, we witness a trend toward increasingly complex applications, characterized by complex graphic interfaces and many more features than their predecessors had. Large applications are written by teams of programmers and require the modularity and semantic checking features provided by high-level languages.

### Further Reading

Aho, A., R. Sethi, and J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

*Slightly dated and lacking in coverage of modern architectures, but still the standard reference on compilers.*

Sweetman, D. [1999]. See *MIPS Run*, San Francisco, CA: Morgan Kaufmann Publishers.

*A complete, detailed, and engaging introduction to the MIPS instruction set and assembly language programming on these machines.*

Detailed documentation on the MIPS-32 architecture is available on the Web:

MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture  
(<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00082-2B-MIPS32INT-AFP-02.00.pdf/getDownload>)

MIPS32™ Architecture for Programmers Volume II: The MIPS32™ Instruction Set  
(<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00086-2B-MIPS32BIS-AFP-02.00.pdf/getDownload>)

MIPS32™ Architecture for Programmers Volume III: The MIPS32™ Privileged Resource Architecture  
(<http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00090-2B-MIPS32PRA-AFP-02.00.pdf/getDownload>)

## A.12 Exercises

**A.1** [5] <§A.5> Section A.5 described how memory is partitioned on most MIPS systems. Propose another way of dividing memory that meets the same goals.

**A.2** [20] <§A.6> Rewrite the code for `fact` to use fewer instructions.

**A.3** [5] <§A.7> Is it ever safe for a user program to use registers `$k0` or `$k1`?

**A.4** [25] <§A.7> Section A.7 contains code for a very simple exception handler. One serious problem with this handler is that it disables interrupts for a long time. This means that interrupts from a fast I/O device may be lost. Write a better exception handler that is interruptable and enables interrupts as quickly as possible.

**A.5** [15] <§A.7> The simple exception handler always jumps back to the instruction following the exception. This works fine unless the instruction that causes the exception is in the delay slot of a branch. In that case, the next instruction is the target of the branch. Write a better handler that uses the EPC register to determine which instruction should be executed after the exception.

**A.6** [5] <§A.9> Using SPIM, write and test an adding machine program that repeatedly reads in integers and adds them into a running sum. The program should stop when it gets an input that is 0, printing out the sum at that point. Use the SPIM system calls described on pages A-43 and A-45.

**A.7** [5] <§A.9> Using SPIM, write and test a program that reads in three integers and prints out the sum of the largest two of the three. Use the SPIM system calls described on pages A-43 and A-45. You can break ties arbitrarily.

**A.8** [5] <§A.9> Using SPIM, write and test a program that reads in a positive integer using the SPIM system calls. If the integer is not positive, the program should terminate with the message “Invalid Entry”; otherwise the program should print out the names of the digits of the integers, delimited by exactly one space. For example, if the user entered “728,” the output would be “Seven Two Eight.”

**A.9** [25] <§A.9> Write and test a MIPS assembly language program to compute and print the first 100 prime numbers. A number  $n$  is prime if no numbers except 1 and  $n$  divide it evenly. You should implement two routines:

- `test_prime (n)`    Return 1 if  $n$  is prime and 0 if  $n$  is not prime.
- `main ()`    Iterate over the integers, testing if each is prime. Print the first 100 numbers that are prime.

Test your programs by running them on SPIM.

**A.10** [10] <§§A.6, A.9> Using SPIM, write and test a recursive program for solving the classic mathematical recreation, the Towers of Hanoi puzzle. (This will require the use of stack frames to support recursion.) The puzzle consists of three pegs (1, 2, and 3) and  $n$  disks (the number  $n$  can vary; typical values might be in the range from 1 to 8). Disk 1 is smaller than disk 2, which is in turn smaller than disk 3, and so forth, with disk  $n$  being the largest. Initially, all the disks are on peg 1, starting with disk  $n$  on the bottom, disk  $n - 1$  on top of that, and so forth, up to disk 1 on the top. The goal is to move all the disks to peg 2. You may only move one disk at a time, that is, the top disk from any of the three pegs onto the top of either of the other two pegs. Moreover, there is a constraint: You must not place a larger disk on top of a smaller disk.

The C program below can be used to help write your assembly language program.

```
/* move n smallest disks from start to finish using
extra */

void hanoi(int n, int start, int finish, int extra){
    if(n != 0){
        hanoi(n-1, start, extra, finish);
        print_string("Move disk");
        print_int(n);
        print_string("from peg");
        print_int(start);
        print_string("to peg");
        print_int(finish);
        print_string(".\n");
        hanoi(n-1, extra, finish, start);
    }
}

main(){
    int n;
    print_string("Enter number of disks>");
    n = read_int();
    hanoi(n, 1, 2, 3);
    return 0;
}
```

# B

## A P P E N D I X

*I always loved that word,* Boolean.

### Claude Shannon

*IEEE Spectrum*, April 1992  
(Shannon's master's thesis showed that the algebra invented by George Boole in the 1800s could represent the workings of electrical switches.)

## The Basics of Logic Design

- B.1      Introduction**    B-3
- B.2      Gates, Truth Tables, and Logic Equations**    B-4
- B.3      Combinational Logic**    B-9
- B.4      Using a Hardware Description Language**    B-20
- B.5      Constructing a Basic Arithmetic Logic Unit**    B-26
- B.6      Faster Addition: Carry Lookahead**    B-38
- B.7      Clocks**    B-48

<b>B.8</b>	<b>Memory Elements: Flip-Flops, Latches, and Registers</b>	B-50
<b>B.9</b>	<b>Memory Elements: SRAMs and DRAMs</b>	B-58
<b>B.10</b>	<b>Finite-State Machines</b>	B-67
<b>B.11</b>	<b>Timing Methodologies</b>	B-72
<b>B.12</b>	<b>Field Programmable Devices</b>	B-78
<b>B.13</b>	<b>Concluding Remarks</b>	B-79
<b>B.14</b>	<b>Exercises</b>	B-80

---

## B.1

### Introduction

This appendix provides a brief discussion of the basics of logic design. It does not replace a course in logic design, nor will it enable you to design significant working logic systems. If you have little or no exposure to logic design, however, this appendix will provide sufficient background to understand all the material in this book. In addition, if you are looking to understand some of the motivation behind how computers are implemented, this material will serve as a useful introduction. If your curiosity is aroused but not sated by this appendix, the references at the end provide several additional sources of information.

Section B.2 introduces the basic building blocks of logic, namely, *gates*. Section B.3 uses these building blocks to construct simple *combinational* logic systems, which contain no memory. If you have had some exposure to logic or digital systems, you will probably be familiar with the material in these first two sections. Section B.5 shows how to use the concepts of Sections B.2 and B.3 to design an ALU for the MIPS processor. Section B.6 shows how to make a fast adder, and

may be safely skipped if you are not interested in this topic. Section B.7 is a short introduction to the topic of clocking, which is necessary to discuss how memory elements work. Section B.8 introduces memory elements, and Section B.9 extends it to focus on random access memories; it describes both the characteristics that are important to understanding how they are used, as discussed in Chapter 4, and the background that motivates many of the aspects of memory hierarchy design discussed in Chapter 5. Section B.10 describes the design and use of finite-state machines, which are sequential logic blocks. If you intend to read [Appendix D](#), you should thoroughly understand the material in Sections B.2 through B.10. If you intend to read only the material on control in Chapter 4, you can skim the appendices; however, you should have some familiarity with all the material except Section B.11. Section B.11 is intended for those who want a deeper understanding of clocking methodologies and timing. It explains the basics of how edge-triggered clocking works, introduces another clocking scheme, and briefly describes the problem of synchronizing asynchronous inputs.

Throughout this appendix, where it is appropriate, we also include segments to demonstrate how logic can be represented in Verilog, which we introduce in Section B.4. A more extensive and complete Verilog tutorial appears elsewhere on the CD.

## B.2

## Gates, Truth Tables, and Logic Equations

The electronics inside a modern computer are *digital*. Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage. All other voltage values are temporary and occur while transitioning between the values. (As we discuss later in this section, a possible pitfall in digital design is sampling a signal when it is not clearly either high or low.) The fact that computers are digital is also a key reason they use binary numbers, since a binary system matches the underlying abstraction inherent in the electronics. In various logic families, the values and relationships between the two voltage values differ. Thus, rather than refer to the voltage levels, we talk about signals that are (logically) true, or 1, or are **asserted**; or signals that are (logically) false, or 0, or are **deasserted**. The values 0 and 1 are called *complements* or *inverses* of one another.

Logic blocks are categorized as one of two types, depending on whether they contain memory. Blocks without memory are called *combinational*; the output of a combinational block depends only on the current input. In blocks with memory, the outputs can depend on both the inputs and the value stored in memory, which is called the *state* of the logic block. In this section and the next, we will focus

**asserted signal** A signal that is (logically) true, or 1.

**deasserted signal** A signal that is (logically) false, or 0.

only on **combinational logic**. After introducing different memory elements in Section B.8, we will describe how **sequential logic**, which is logic including state, is designed.

## Truth Tables

Because a combinational logic block contains no memory, it can be completely specified by defining the values of the outputs for each possible set of input values. Such a description is normally given as a *truth table*. For a logic block with  $n$  inputs, there are  $2^n$  entries in the truth table, since there are that many possible combinations of input values. Each entry specifies the value of all the outputs for that particular input combination.

### combinational logic

A logic system whose blocks do not contain memory and hence compute the same output given the same input.

### sequential logic

A group of logic elements that contain memory and hence whose value depends on the inputs as well as the current contents of the memory.

### Truth Tables

Consider a logic function with three inputs,  $A$ ,  $B$ , and  $C$ , and three outputs,  $D$ ,  $E$ , and  $F$ . The function is defined as follows:  $D$  is true if at least one input is true,  $E$  is true if exactly two inputs are true, and  $F$  is true only if all three inputs are true. Show the truth table for this function.

The truth table will contain  $2^3 = 8$  entries. Here it is:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

### EXAMPLE

### ANSWER

Truth tables can completely describe any combinational logic function; however, they grow in size quickly and may not be easy to understand. Sometimes we want to construct a logic function that will be 0 for many input combinations, and we use a shorthand of specifying only the truth table entries for the nonzero outputs. This approach is used in Chapter 4 and [Appendix D](#).

## Boolean Algebra

Another approach is to express the logic function with logic equations. This is done with the use of *Boolean algebra* (named after Boole, a 19th-century mathematician). In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator is written as  $+$ , as in  $A + B$ . The result of an OR operator is 1 if either of the variables is 1. The OR operation is also called a *logical sum*, since its result is 1 if either operand is 1.
- The AND operator is written as  $\cdot$ , as in  $A \cdot B$ . The result of an AND operator is 1 only if both inputs are 1. The AND operator is also called *logical product*, since its result is 1 only if both operands are 1.
- The unary operator NOT is written as  $\bar{A}$ . The result of a NOT operator is 1 only if the input is 0. Applying the operator NOT to a logical value results in an inversion or negation of the value (i.e., if the input is 0 the output is 1, and vice versa).

There are several laws of Boolean algebra that are helpful in manipulating logic equations.

- Identity law:  $A + 0 = A$  and  $A \cdot 1 = A$
- Zero and One laws:  $A + 1 = 1$  and  $A \cdot 0 = 0$
- Inverse laws:  $A + \bar{A} = 1$  and  $A \cdot \bar{A} = 0$
- Commutative laws:  $A + B = B + A$  and  $A \cdot B = B \cdot A$
- Associative laws:  $A + (B + C) = (A + B) + C$  and  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive laws:  $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$  and  
 $A + (B \cdot C) = (A + B) \cdot (A + C)$

In addition, there are two other useful theorems, called DeMorgan's laws, that are discussed in more depth in the exercises.

Any set of logic functions can be written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables and the three operators above on the right-hand side.

## Logic Equations

Show the logic equations for the logic functions,  $D$ ,  $E$ , and  $F$ , described in the previous example.

**EXAMPLE**

Here's the equation for  $D$ :

$$D = A + B + C$$

**ANSWER**

$F$  is equally simple:

$$F = A \cdot B \cdot C$$

$E$  is a little tricky. Think of it in two parts: what must be true for  $E$  to be true (two of the three inputs must be true), and what cannot be true (all three cannot be true). Thus we can write  $E$  as

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

We can also derive  $E$  by realizing that  $E$  is true only if exactly two of the inputs are true. Then we can write  $E$  as an OR of the three possible terms that have two true inputs and one false input:

$$E = (A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$$

Proving that these two expressions are equivalent is explored in the exercises.

In Verilog, we describe combinational logic whenever possible using the assign statement, which is described beginning on page B-23. We can write a definition for  $E$  using the Verilog exclusive-OR operator as `assign E = (A ^ B ^ C) * (A + B + C) * (A * B * C)`, which is yet another way to describe this function.  $D$  and  $F$  have even simpler representations, which are just like the corresponding C code: `D = A | B | C` and `F = A & B & C`.

## Gates

**gate** A device that implements basic logic functions, such as AND or OR.

Logic blocks are built from **gates** that implement basic logic functions. For example, an AND gate implements the AND function, and an OR gate implements the OR function. Since both AND and OR are commutative and associative, an AND or an OR gate can have multiple inputs, with the output equal to the AND or OR of all the inputs. The logical function NOT is implemented with an inverter that always has a single input. The standard representation of these three logic building blocks is shown in [Figure B.2.1](#).

Rather than draw inverters explicitly, a common practice is to add “bubbles” to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted. For example, [Figure B.2.2](#) shows the logic diagram for the function  $\bar{A} + B$ , using explicit inverters on the left and bubbled inputs and outputs on the right.

Any logical function can be constructed using AND gates, OR gates, and inversion; several of the exercises give you the opportunity to try implementing some common logic functions with gates. In the next section, we’ll see how an implementation of any logic function can be constructed using this knowledge.

**NOR gate** An inverted OR gate.

**NAND gate** An inverted AND gate.

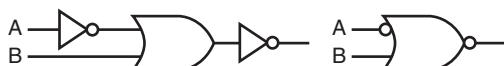
### Check Yourself

Are the following two logical expressions equivalent? If not, find a setting of the variables to show they are not:

- $(A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) + (B \cdot C \cdot \bar{A})$
- $B \cdot (A \cdot \bar{C} + C \cdot \bar{A})$



**FIGURE B.2.1** Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.



**FIGURE B.2.2** Logic gate implementation of  $\bar{A} + B$  using explicit inverts on the left and bubbled inputs and outputs on the right. This logic function can be simplified to  $A \cdot \bar{B}$  or in Verilog,  $A \& \sim B$ .

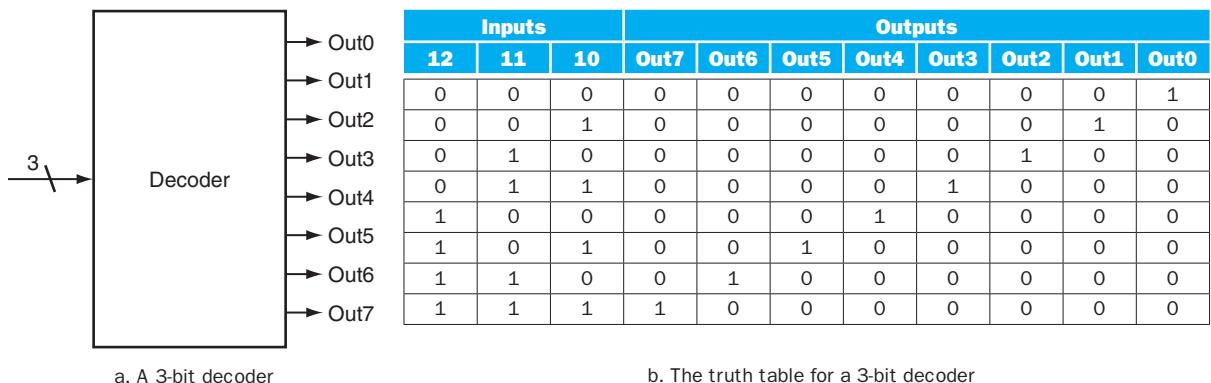
**B.3****Combinational Logic**

In this section, we look at a couple of larger logic building blocks that we use heavily, and we discuss the design of structured logic that can be automatically implemented from a logic equation or truth table by a translation program. Last, we discuss the notion of an array of logic blocks.

**Decoders**

One logic block that we will use in building larger components is a **decoder**. The most common type of decoder has an  $n$ -bit input and  $2^n$  outputs, where only one output is asserted for each input combination. This decoder translates the  $n$ -bit input into a signal that corresponds to the binary value of the  $n$ -bit input. The outputs are thus usually numbered, say, Out0, Out1, ..., Out $2^n - 1$ . If the value of the input is  $i$ , then Out $i$  will be true and all other outputs will be false. Figure B.3.1 shows a 3-bit decoder and the truth table. This decoder is called a *3-to-8 decoder* since there are 3 inputs and 8 ( $2^3$ ) outputs. There is also a logic element called an *encoder* that performs the inverse function of a decoder, taking  $2^n$  inputs and producing an  $n$ -bit output.

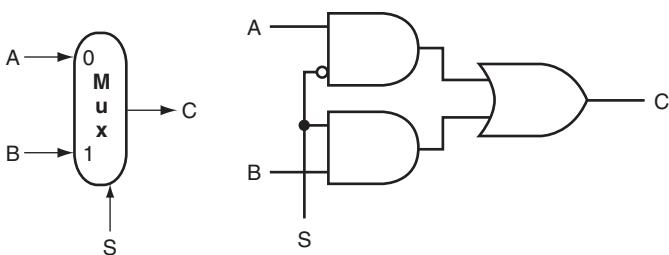
**decoder** A logic block that has an  $n$ -bit input and  $2n$  outputs, where only one output is asserted for each input combination.



a. A 3-bit decoder

b. The truth table for a 3-bit decoder

**FIGURE B.3.1 A 3-bit decoder has 3 inputs, called 12, 11, and 10, and  $2^3 = 8$  outputs, called Out0 to Out7.** Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.



**FIGURE B.3.2 A two-input multiplexor on the left and its implementation with gates on the right.** The multiplexor has two data inputs ( $A$  and  $B$ ), which are labeled 0 and 1, and one selector input ( $S$ ), as well as an output  $C$ . Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this beginning on page B-23.

## Multiplexors

**selector value** Also called **control value**. The control signal that is used to select one of the input values of a multiplexor as the output of the multiplexor.

One basic logic function that we use quite often in Chapter 4 is the *multiplexor*. A multiplexor might more properly be called a *selector*, since its output is one of the inputs that is selected by a control. Consider the two-input multiplexor. The left side of Figure B.3.2 shows this multiplexor has three inputs: two data values and a **selector** (or **control**) **value**. The selector value determines which of the inputs becomes the output. We can represent the logic function computed by a two-input multiplexor, shown in gate form on the right side of Figure B.3.2, as  $C = (A \cdot S) + (B \cdot \bar{S})$ .

Multiplexors can be created with an arbitrary number of data inputs. When there are only two inputs, the selector is a single signal that selects one of the inputs if it is true (1) and the other if it is false (0). If there are  $n$  data inputs, there will need to be  $\lceil \log_2 n \rceil$  selector inputs. In this case, the multiplexor basically consists of three parts:

1. A decoder that generates  $n$  signals, each indicating a different input value
2. An array of  $n$  AND gates, each combining one of the inputs with a signal from the decoder
3. A single large OR gate that incorporates the outputs of the AND gates

To associate the inputs with selector values, we often label the data inputs numerically (i.e., 0, 1, 2, 3, ...,  $n - 1$ ) and interpret the data selector inputs as a binary number. Sometimes, we make use of a multiplexor with undecoded selector signals.

Multiplexors are easily represented combinationally in Verilog by using *if* expressions. For larger multiplexors, *case* statements are more convenient, but care must be taken to synthesize combinational logic.

## Two-Level Logic and PLAs

As pointed out in the previous section, any logic function can be implemented with only AND, OR, and NOT functions. In fact, a much stronger result is true. Any logic function can be written in a canonical form, where every input is either a true or complemented variable and there are only two levels of gates—one being AND and the other OR—with a possible inversion on the final output. Such a representation is called a *two-level representation*, and there are two forms, called **sum of products** and **product of sums**. A sum-of-products representation is a logical sum (OR) of products (terms using the AND operator); a product of sums is just the opposite. In our earlier example, we had two equations for the output  $E$ :

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

and

$$E = (A \cdot B \cdot \bar{C}) + (A \cdot C \cdot \bar{B}) \cdot (B \cdot C \cdot \bar{A})$$

This second equation is in a sum-of-products form: it has two levels of logic and the only inversions are on individual variables. The first equation has three levels of logic.

**Elaboration:** We can also write  $E$  as a product of sums:

$$E = \overline{(\bar{A} + \bar{B} + C)} \cdot \overline{(\bar{A} + \bar{C} + B)} \cdot \overline{(\bar{B} + C + A)}$$

To derive this form, you need to use *DeMorgan's theorems*, which are discussed in the exercises.

In this text, we use the sum-of-products form. It is easy to see that any logic function can be represented as a sum of products by constructing such a representation from the truth table for the function. Each truth table entry for which the function is true corresponds to a product term. The product term consists of a logical product of all the inputs or the complements of the inputs, depending on whether the entry in the truth table has a 0 or 1 corresponding to this variable. The logic function is the logical sum of the product terms where the function is true. This is more easily seen with an example.

**sum of products** A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

**EXAMPLE****Sum of Products**

Show the sum-of-products representation for the following truth table for  $D$ .

Inputs		Outputs	
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

**ANSWER**

There are four product terms, since the function is true (1) for four different input combinations. These are:

$$\bar{A} \cdot \bar{B} \cdot C$$

$$\bar{A} \cdot B \cdot C$$

$$A \cdot \bar{B} \cdot \bar{C}$$

$$A \cdot B \cdot C$$

Thus, we can write the function for  $D$  as the sum of these terms:

$$D = (\bar{A} \cdot \bar{B} \cdot C)(\bar{A} \cdot B \cdot \bar{C})(A \cdot \bar{B} \cdot \bar{C})(A \cdot B \cdot C)$$

Note that only those truth table entries for which the function is true generate terms in the equation.

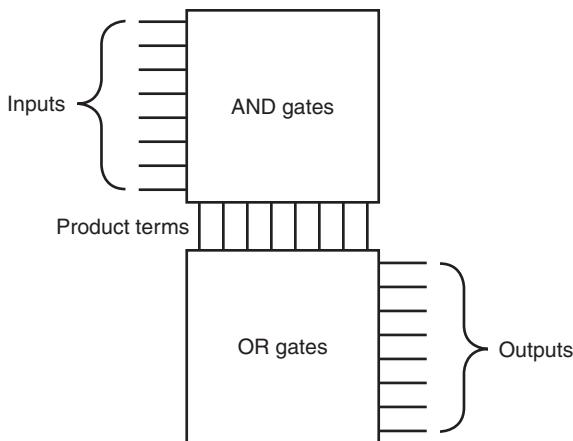
We can use this relationship between a truth table and a two-level representation to generate a gate-level implementation of any set of logic functions. A set of logic functions corresponds to a truth table with multiple output columns, as we saw in the example on page B-5. Each output column represents a different logic function, which may be directly constructed from the truth table.

The sum-of-products representation corresponds to a common structured-logic implementation called a **programmable logic array (PLA)**. A PLA has a set of inputs and corresponding input complements (which can be implemented with a set of inverters), and two stages of logic. The first stage is an array of AND gates that form a set of **product terms** (sometimes called **minterms**); each product term can consist of any of the inputs or their complements. The second stage is an array of OR gates, each of which forms a logical sum of any number of the product terms. Figure B.3.3 shows the basic form of a PLA.

**programmable logic array (PLA)**

A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic: the first generates product terms of the inputs and input complements, and the second generates sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

**minterms** Also called **product terms**. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the *programmable logic array (PLA)*.



**FIGURE B.3.3** The basic form of a PLA consists of an array of AND gates followed by an array of OR gates. Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.

A PLA can directly implement the truth table of a set of logic functions with multiple inputs and outputs. Since each entry where the output is true requires a product term, there will be a corresponding row in the PLA. Each output corresponds to a potential row of OR gates in the second stage. The number of OR gates corresponds to the number of truth table entries for which the output is true. The total size of a PLA, such as that shown in Figure B.3.3, is equal to the sum of the size of the AND gate array (called the *AND plane*) and the size of the OR gate array (called the *OR plane*). Looking at Figure B.3.3, we can see that the size of the AND gate array is equal to the number of inputs times the number of different product terms, and the size of the OR gate array is the number of outputs times the number of product terms.

A PLA has two characteristics that help make it an efficient way to implement a set of logic functions. First, only the truth table entries that produce a true value for at least one output have any logic gates associated with them. Second, each different product term will have only one entry in the PLA, even if the product term is used in multiple outputs. Let's look at an example.

### PLAs

Consider the set of logic functions defined in the example on page B-5. Show a PLA implementation of this example for  $D$ ,  $E$ , and  $F$ .

### EXAMPLE

## ANSWER

Here is the truth table we constructed earlier:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Since there are seven unique product terms with at least one true value in the output section, there will be seven columns in the AND plane. The number of rows in the AND plane is three (since there are three inputs), and there are also three rows in the OR plane (since there are three outputs). [Figure B.3.4](#) shows the resulting PLA, with the product terms corresponding to the truth table entries from top to bottom.

**read-only memory (ROM)** A memory whose contents are designated at creation time, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.

**programmable ROM (PROM)** A form of read-only memory that can be programmed when a designer knows its contents.

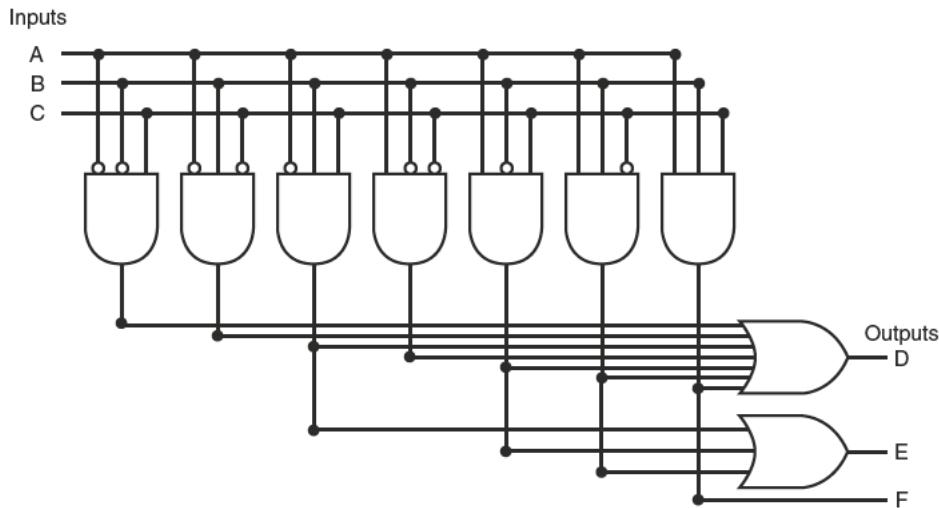
Rather than drawing all the gates, as we do in [Figure B.3.4](#), designers often show just the position of AND gates and OR gates. Dots are used on the intersection of a product term signal line and an input line or an output line when a corresponding AND gate or OR gate is required. [Figure B.3.5](#) shows how the PLA of [Figure B.3.4](#) would look when drawn in this way. The contents of a PLA are fixed when the PLA is created, although there are also forms of PLA-like structures, called *PALs*, that can be programmed electronically when a designer is ready to use them.

## ROMs

Another form of structured logic that can be used to implement a set of logic functions is a **read-only memory (ROM)**. A ROM is called a memory because it has a set of locations that can be read; however, the contents of these locations are fixed, usually at the time the ROM is manufactured. There are also **programmable ROMs (PROMs)** that can be programmed electronically, when a designer knows their contents. There are also erasable PROMs; these devices require a slow erasure process using ultraviolet light, and thus are used as read-only memories, except during the design and debugging process.

A ROM has a set of input address lines and a set of outputs. The number of addressable entries in the ROM determines the number of address lines: if the

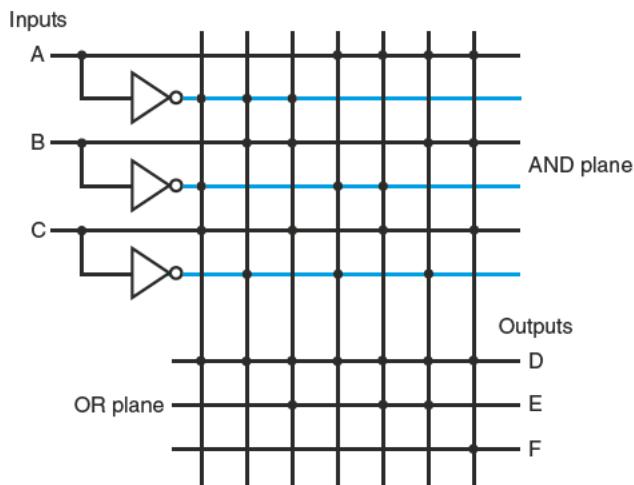
ROM contains  $2^m$  addressable entries, called the *height*, then there are  $m$  input lines. The number of bits in each addressable entry is equal to the number of output bits and is sometimes called the *width* of the ROM. The total number of bits in the ROM is equal to the height times the width. The height and width are sometimes collectively referred to as the *shape* of the ROM.



**FIGURE B.3.4** The PLA for implementing the logic function described in the example.

A ROM can encode a collection of logic functions directly from the truth table. For example, if there are  $n$  functions with  $m$  inputs, we need a ROM with  $m$  address lines (and  $2^m$  entries), with each entry being  $n$  bits wide. The entries in the input portion of the truth table represent the addresses of the entries in the ROM, while the contents of the output portion of the truth table constitute the contents of the ROM. If the truth table is organized so that the sequence of entries in the input portion constitutes a sequence of binary numbers (as have all the truth tables we have shown so far), then the output portion gives the ROM contents in order as well. In the example starting on page B-13, there were three inputs and three outputs. This leads to a ROM with  $2^3 = 8$  entries, each 3 bits wide. The contents of those entries in increasing order by address are directly given by the output portion of the truth table that appears on page B-14.

ROMs and PLAs are closely related. A ROM is fully decoded: it contains a full output word for every possible input combination. A PLA is only partially decoded. This means that a ROM will always contain more entries. For the earlier truth table on page B-14, the ROM contains entries for all eight possible inputs, whereas the PLA contains only the seven active product terms. As the number of inputs grows,



**FIGURE B.3.5 A PLA drawn using dots to indicate the components of the product terms and sum terms in the array.** Rather than use inverters on the gates, usually all the inputs are run the width of the AND plane in both true and complement forms. A dot in the AND plane indicates that the input, or its inverse, occurs in the product term. A dot in the OR plane indicates that the corresponding product term appears in the corresponding output.

the number of entries in the ROM grows exponentially. In contrast, for most real logic functions, the number of product terms grows much more slowly (see the examples in [Appendix D](#)). This difference makes PLAs generally more efficient for implementing combinational logic functions. ROMs have the advantage of being able to implement any logic function with the matching number of inputs and outputs. This advantage makes it easier to change the ROM contents if the logic function changes, since the size of the ROM need not change.

In addition to ROMs and PLAs, modern logic synthesis systems will also translate small blocks of combinational logic into a collection of gates that can be placed and wired automatically. Although some small collections of gates are usually not area efficient, for small logic functions they have less overhead than the rigid structure of a ROM and PLA and so are preferred.

For designing logic outside of a custom or semicustom integrated circuit, a common choice is a field programming device; we describe these devices in Section B.12.

## Don't Cares

Often in implementing some combinational logic, there are situations where we do not care what the value of some output is, either because another output is true or because a subset of the input combinations determines the values of the outputs. Such situations are referred to as *don't cares*. Don't cares are important because they make it easier to optimize the implementation of a logic function.

There are two types of don't cares: output don't cares and input don't cares, both of which can be represented in a truth table. *Output don't cares* arise when we don't care about the value of an output for some input combination. They appear as Xs in the output portion of a truth table. When an output is a don't care for some input combination, the designer or logic optimization program is free to make the output true or false for that input combination. *Input don't cares* arise when an output depends on only some of the inputs, and they are also shown as Xs, though in the input portion of the truth table.

### Don't Cares

Consider a logic function with inputs A, B, and C defined as follows:

- If A or C is true, then output D is true, whatever the value of B.
- If A or B is true, then output E is true, whatever the value of C.
- Output F is true if exactly one of the inputs is true, although we don't care about the value of F, whenever D and E are both true.

### EXAMPLE

Show the full truth table for this function and the truth table using don't cares. How many product terms are required in a PLA for each of these?

Here's the full truth table, without don't cares:

### ANSWER

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

This requires seven product terms without optimization. The truth table written with output don't cares looks like this:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

If we also use the input don't cares, this truth table can be further simplified to yield the following:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

This simplified truth table requires a PLA with four minterms, or it can be implemented in discrete gates with one two-input AND gate and three OR gates (two with three inputs and one with two inputs). This compares to the original truth table that had seven minterms and would have required four AND gates.

Logic minimization is critical to achieving efficient implementations. One tool useful for hand minimization of random logic is *Karnaugh maps*. Karnaugh maps represent the truth table graphically, so that product terms that may be combined are easily seen. Nevertheless, hand optimization of significant logic functions using Karnaugh maps is impractical, both because of the size of the maps and their complexity. Fortunately, the process of logic minimization is highly mechanical and can be performed by design tools. In the process of minimization, the tools take advantage of the don't cares, so specifying them is important. The text book references at the end of this appendix provide further discussion on logic minimization, Karnaugh maps, and the theory behind such minimization algorithms.

## Arrays of Logic Elements

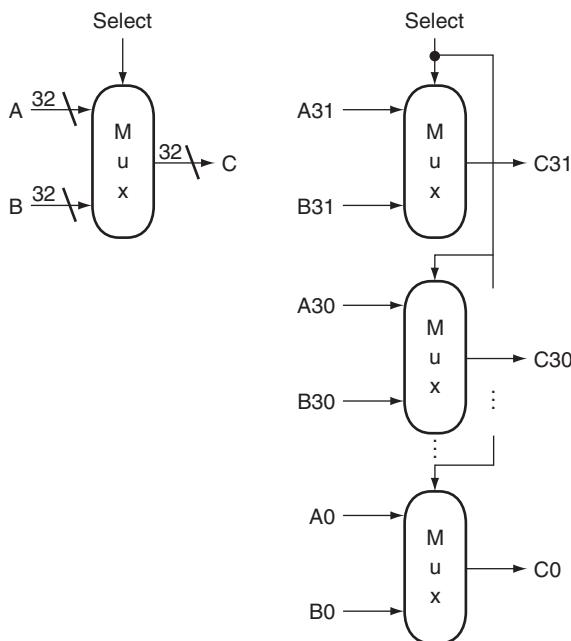
Many of the combinational operations to be performed on data have to be done to an entire word (32 bits) of data. Thus we often want to build an array of logic

elements, which we can represent simply by showing that a given operation will happen to an entire collection of inputs. Inside a machine, much of the time we want to select between a pair of *buses*. A **bus** is a collection of data lines that is treated together as a single logical signal. (The term *bus* is also used to indicate a shared collection of lines with multiple sources and uses.)

For example, in the MIPS instruction set, the result of an instruction that is written into a register can come from one of two sources. A multiplexor is used to choose which of the two buses (each 32 bits wide) will be written into the Result register. The 1-bit multiplexor, which we showed earlier, will need to be replicated 32 times.

We indicate that a signal is a bus rather than a single 1-bit line by showing it with a thicker line in a figure. Most buses are 32 bits wide; those that are not are explicitly labeled with their width. When we show a logic unit whose inputs and outputs are buses, this means that the unit must be replicated a sufficient number of times to accommodate the width of the input. Figure B.3.6 shows how we draw a multiplexor that selects between a pair of 32-bit buses and how this expands in terms of 1-bit-wide multiplexors. Sometimes we need to construct an array of logic elements where the inputs for some elements in the array are outputs from earlier elements. For example, this is how a multibit-wide ALU is constructed. In such cases, we must explicitly show how to create wider arrays, since the individual elements of the array are no longer independent, as they are in the case of a 32-bit-wide multiplexor.

**bus** In logic design, a collection of data lines that is treated together as a single logical signal; also, a shared collection of lines with multiple sources and uses.



a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

**FIGURE B.3.6 A multiplexor is arrayed 32 times to perform a selection between two 32-bit inputs.** Note that there is still only one data selection signal used for all 32 1-bit multiplexors.

## Check Yourself

Parity is a function in which the output depends on the number of 1s in the input. For an even parity function, the output is 1 if the input has an even number of ones. Suppose a ROM is used to implement an even parity function with a 4-bit input. Which of A, B, C, or D represents the contents of the ROM?

Address	A	B	C	D
0	0	1	0	1
1	0	1	1	0
2	0	1	0	1
3	0	1	1	0
4	0	1	0	1
5	0	1	1	0
6	0	1	0	1
7	0	1	1	0
8	1	0	0	1
9	1	0	1	0
10	1	0	0	1
11	1	0	1	0
12	1	0	0	1
13	1	0	1	0
14	1	0	0	1
15	1	0	1	0

## B.4

## Using a Hardware Description Language

### hardware description language

A programming language for describing hardware, used for generating simulations of a hardware design and also as input to synthesis tools that can generate actual hardware.

**Verilog** One of the two most common hardware description languages.

**VHDL** One of the two most common hardware description languages.

Today most digital design of processors and related hardware systems is done using a **hardware description language**. Such a language serves two purposes. First, it provides an abstract description of the hardware to simulate and debug the design. Second, with the use of logic synthesis and hardware compilation tools, this description can be compiled into the hardware implementation.

In this section, we introduce the hardware description language Verilog and show how it can be used for combinational design. In the rest of the appendix, we expand the use of Verilog to include design of sequential logic. In the optional sections of Chapter 4 that appear online, we use Verilog to describe processor implementations. In the optional section from Chapter 5 that appears online, we use system Verilog to describe cache controller implementations. System Verilog adds structures and some other useful features to Verilog.

**Verilog** is one of the two primary hardware description languages; the other is **VHDL**. Verilog is somewhat more heavily used in industry and is based on C, as opposed to VHDL, which is based on Ada. The reader generally familiar with C will find the basics of Verilog, which we use in this appendix, easy to follow.

Readers already familiar with VHDL should find the concepts simple, provided they have been exposed to the syntax of C.

Verilog can specify both a behavioral and a structural definition of a digital system. A **behavioral specification** describes how a digital system functionally operates. A **structural specification** describes the detailed organization of a digital system, usually using a hierarchical description. A structural specification can be used to describe a hardware system in terms of a hierarchy of basic elements such as gates and switches. Thus, we could use Verilog to describe the exact contents of the truth tables and datapath of the last section.

With the arrival of **hardware synthesis tools**, most designers now use Verilog or VHDL to structurally describe only the datapath, relying on logic synthesis to generate the control from a behavioral description. In addition, most CAD systems provide extensive libraries of standardized parts, such as ALUs, multiplexors, register files, memories, and programmable logic blocks, as well as basic gates.

Obtaining an acceptable result using libraries and logic synthesis requires that the specification be written with an eye toward the eventual synthesis and the desired outcome. For our simple designs, this primarily means making clear what we expect to be implemented in combinational logic and what we expect to require sequential logic. In most of the examples we use in this section and the remainder of this appendix, we have written the Verilog with the eventual synthesis in mind.

**behavioral specification** Describes how a digital system operates functionally.

**structural specification** Describes how a digital system is organized in terms of a hierarchical connection of elements.

**hardware synthesis tools** Computer-aided design software that can generate a gate-level design based on behavioral descriptions of a digital system.

## Datatypes and Operators in Verilog

There are two primary datatypes in Verilog:

1. A **wire** specifies a combinational signal.
2. A **reg** (register) holds a value, which can vary with time. A reg need not necessarily correspond to an actual register in an implementation, although it often will.

**wire** In Verilog, specifies a combinational signal.

**reg** In Verilog, a register.

A register or wire, named X, that is 32 bits wide is declared as an array: `reg [31:0] X` or `wire [31:0] X`, which also sets the index of 0 to designate the least significant bit of the register. Because we often want to access a subfield of a register or wire, we can refer to a contiguous set of bits of a register or wire with the notation `[starting bit: ending bit]`, where both indices must be constant values.

An array of registers is used for a structure like a register file or memory. Thus, the declaration

```
reg [31:0] registerfile[0:31]
```

specifies a variable registerfile that is equivalent to a MIPS registerfile, where register 0 is the first. When accessing an array, we can refer to a single element, as in C, using the notation `registerfile[regnum]`.

The possible values for a register or wire in Verilog are

- 0 or 1, representing logical false or true
- X, representing unknown, the initial value given to all registers and to any wire not connected to something
- Z, representing the high-impedance state for tristate gates, which we will not discuss in this appendix

Constant values can be specified as decimal numbers as well as binary, octal, or hexadecimal. We often want to say exactly how large a constant field is in bits. This is done by prefixing the value with a decimal number specifying its size in bits. For example:

- 4'b0100 specifies a 4-bit binary constant with the value 4, as does 4'd4.
- -8'h4 specifies an 8-bit constant with the value -4 (in two's complement representation)

Values can also be concatenated by placing them within {} separated by commas. The notation {x{bit field}} replicates bit field x times. For example:

- {16{2'b01}} creates a 32-bit value with the pattern 0101 ... 01.
- {A[31:16],B[15:0]} creates a value whose upper 16 bits come from A and whose lower 16 bits come from B.

Verilog provides the full set of unary and binary operators from C, including the arithmetic operators (+, -, \*, /), the logical operators (&, |, ~), the comparison operators (=, !=, >, <, <=, >=), the shift operators (<<, >>), and C's conditional operator (?), which is used in the form condition ? expr1 : expr2 and returns expr1 if the condition is true and expr2 if it is false). Verilog adds a set of unary logic reduction operators (&, |, ^) that yield a single bit by applying the logical operator to all the bits of an operand. For example, &A returns the value obtained by ANDing all the bits of A together, and ^A returns the reduction obtained by using exclusive OR on all the bits of A.

### Check Yourself

Which of the following define exactly the same value?

1. 8'bimoooo
2. 8'hF0
3. 8'd240
4. {{4{1'b1}}, {4{1'b0}}} {4'b1, 4'b0}
5. {4'b1, 4'b0}

## Structure of a Verilog Program

A Verilog program is structured as a set of modules, which may represent anything from a collection of logic gates to a complete system. Modules are similar to classes in C++, although not nearly as powerful. A module specifies its input and output ports, which describe the incoming and outgoing connections of a module. A module may also declare additional variables. The body of a module consists of:

- initial constructs, which can initialize reg variables
- Continuous assignments, which define only combinational logic
- always constructs, which can define either sequential or combinational logic
- Instances of other modules, which are used to implement the module being defined

## Representing Complex Combinational Logic in Verilog

A continuous assignment, which is indicated with the keyword assign, acts like a combinational logic function: the output is continuously assigned the value, and a change in the input values is reflected immediately in the output value. Wires may only be assigned values with continuous assignments. Using continuous assignments, we can define a module that implements a half-adder, as [Figure B.4.1](#) shows.

Assign statements are one sure way to write Verilog that generates combinational logic. For more complex structures, however, assign statements may be awkward or tedious to use. It is also possible to use the always block of a module to describe a combinational logic element, although care must be taken. Using an always block allows the inclusion of Verilog control constructs, such as *if-then-else*, *case* statements, *for* statements, and *repeat* statements, to be used. These statements are similar to those in C with small changes.

An always block specifies an optional list of signals on which the block is sensitive (in a list starting with @). The always block is re-evaluated if any of the

```
module half_adder (A,B,Sum,Carry);
    input A,B; //two 1-bit inputs
    output Sum, Carry; //two 1-bit outputs
    assign Sum = A ^ B; //sum is A xor B
    assign Carry = A & B; //Carry is A and B
endmodule
```

**FIGURE B.4.1** A Verilog module that defines a half-adder using continuous assignments.

**sensitivity list** The list of signals that specifies when an `always` block should be re-evaluated.

listed signals changes value; if the list is omitted, the `always` block is constantly re-evaluated. When an `always` block is specifying combinational logic, the **sensitivity list** should include all the input signals. If there are multiple Verilog statements to be executed in an `always` block, they are surrounded by the keywords `begin` and `end`, which take the place of the `{` and `}` in C. An `always` block thus looks like this:

```
always @(list of signals that cause reevaluation) begin
    Verilog statements including assignments and other
    control statements end
```

#### blocking assignment

In Verilog, an assignment that completes before the execution of the next statement.

#### nonblocking assignment

An assignment that continues after evaluating the right-hand side, assigning the left-hand side the value only after all right-hand sides are evaluated.

Reg variables may only be assigned inside an `always` block, using a procedural assignment statement (as distinguished from continuous assignment we saw earlier). There are, however, two different types of procedural assignments. The assignment operator `=` executes as it does in C; the right-hand side is evaluated, and the left-hand side is assigned the value. Furthermore, it executes like the normal C assignment statement: that is, it is completed before the next statement is executed. Hence, the assignment operator `=` has the name **blocking assignment**. This blocking can be useful in the generation of sequential logic, and we will return to it shortly. The other form of assignment (**nonblocking**) is indicated by `<=`. In nonblocking assignment, all right-hand sides of the assignments in an `always` group are evaluated and the assignments are done simultaneously. As a first example of combinational logic implemented using an `always` block, Figure B.4.2 shows the implementation of a 4-to-1 multiplexor, which uses a `case` construct to make it easy to write. The `case` construct looks like a C `switch` statement. Figure B.4.3 shows a definition of a MIPS ALU, which also uses a `case` statement.

Since only reg variables may be assigned inside `always` blocks, when we want to describe combinational logic using an `always` block, care must be taken to ensure that the reg does not synthesize into a register. A variety of pitfalls are described in the elaboration below.

**Elaboration:** Continuous assignment statements always yield combinational logic, but other Verilog structures, even when in `always` blocks, can yield unexpected results during logic synthesis. The most common problem is creating sequential logic by implying the existence of a latch or register, which results in an implementation that is both slower and more costly than perhaps intended. To ensure that the logic that you intend to be combinational is synthesized that way, make sure you do the following:

1. Place all combinational logic in a continuous assignment or an `always` block.
2. Make sure that all the signals used as inputs appear in the sensitivity list of an `always` block.
3. Ensure that every path through an `always` block assigns a value to the exact same set of bits.

The last of these is the easiest to overlook; read through the example in Figure B.5.15 to convince yourself that this property is adhered to.

---

```

module Mult4to1 (In1,In2,In3,In4,Sel,Out);
    input [31:0] In1, In2, In3, In4; /four 32-bit inputs
    input [1:0] Sel; //selector signal
    output reg [31:0] Out;// 32-bit output
    always @(In1, In2, In3, In4, Sel)
        case (Sel) //a 4->1 multiplexor
            0: Out <= In1;
            1: Out <= In2;
            2: Out <= In3;
            default: Out <= In4;
        endcase
    endmodule

```

---

**FIGURE B.4.2 A Verilog definition of a 4-to-1 multiplexor with 32-bit inputs, using a case statement.** The case statement acts like a C switch statement, except that in Verilog only the code associated with the selected case is executed (as if each case state had a break at the end) and there is no fall-through to the next statement.

---

```

module MIPSALU (ALUctl, A, B, ALUOut, Zero);
    input [3:0] ALUctl;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
    always @(ALUctl, A, B) //reevaluate if these change
        case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1:0;
            12: ALUOut <= ~(A | B); // result is nor
            default: ALUOut <= 0; //default to 0, should not happen;
        endcase
    endmodule

```

---

**FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU.** This could be synthesized using a module library containing basic arithmetic and logical operations.

**Check Yourself**

Assuming all values are initially zero, what are the values of A and B after executing this Verilog code inside an `always` block?

```
C=1;
A <= C;
B = C;
```

**B.5**

## Constructing a Basic Arithmetic Logic Unit

*ALU n. [Arithmetic Logic Unit or (rare) Arithmetic Logic Unit] A random-number generator supplied as standard with all computer systems.*

Stan Kelly-Bootle, *The Devil's DP Dictionary*, 1981

The **arithmetic logic unit (ALU)** is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. This section constructs an ALU from four hardware building blocks (AND and OR gates, inverters, and multiplexors) and illustrates how combinational logic works. In the next section, we will see how addition can be sped up through more clever designs.

Because the MIPS word is 32 bits wide, we need a 32-bit-wide ALU. Let's assume that we will connect 32 1-bit ALUs to create the desired ALU. We'll therefore start by constructing a 1-bit ALU.

### A 1-Bit ALU

The logical operations are easiest, because they map directly onto the hardware components in [Figure B.2.1](#).

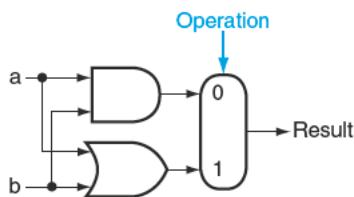
The 1-bit logical unit for AND and OR looks like [Figure B.5.1](#). The multiplexor on the right then selects  $a$  AND  $b$  or  $a$  OR  $b$ , depending on whether the value of *Operation* is 0 or 1. The line that controls the multiplexor is shown in color to distinguish it from the lines containing data. Notice that we have renamed the control and output lines of the multiplexor to give them names that reflect the function of the ALU.

The next function to include is addition. An adder must have two inputs for the operands and a single-bit output for the sum. There must be a second output to pass on the carry, called *CarryOut*. Since the *CarryOut* from the neighbor adder must be included as an input, we need a third input. This input is called *CarryIn*. [Figure B.5.2](#) shows the inputs and the outputs of a 1-bit adder. Since we know what addition is supposed to do, we can specify the outputs of this “black box” based on its inputs, as [Figure B.5.3](#) demonstrates.

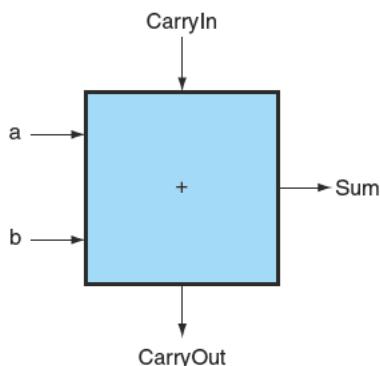
We can express the output functions *CarryOut* and *Sum* as logical equations, and these equations can in turn be implemented with logic gates. Let's do *CarryOut*. [Figure B.5.4](#) shows the values of the inputs when *CarryOut* is a 1.

We can turn this truth table into a logical equation:

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$



**FIGURE B.5.1** The 1-bit logical unit for AND and OR.



**FIGURE B.5.2** A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{two}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{two}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{two}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{two}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{two}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{two}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{two}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{two}}$

**FIGURE B.5.3** Input and output specification for a 1-bit adder.

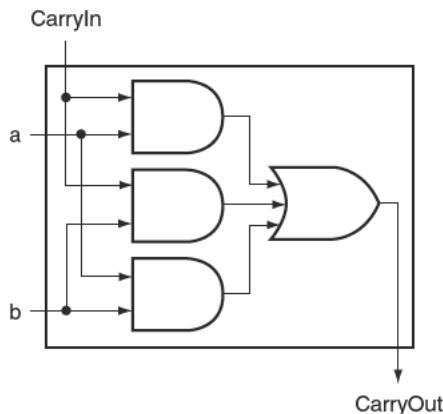
If  $a \cdot b \cdot \text{CarryIn}$  is true, then all of the other three terms must also be true, so we can leave out this last term corresponding to the fourth line of the table. We can thus simplify the equation to

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

Figure B.5.5 shows that the hardware within the adder black box for CarryOut consists of three AND gates and one OR gate. The three AND gates correspond exactly to the three parenthesized terms of the formula above for CarryOut, and the OR gate sums the three terms.

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

**FIGURE B.5.4** Values of the Inputs when CarryOut Is a 1.



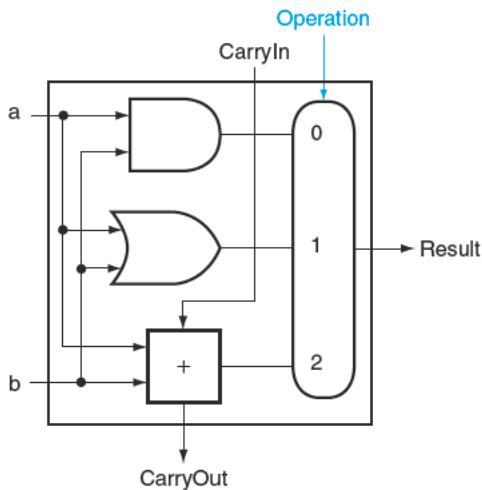
**FIGURE B.5.5** Adder hardware for the CarryOut signal. The rest of the adder hardware is the logic for the Sum output given in the equation on this page.

The Sum bit is set when exactly one input is 1 or when all three inputs are 1. The Sum results in a complex Boolean equation (recall that  $\bar{a}$  means NOT a):

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

The drawing of the logic for the Sum bit in the adder black box is left as an exercise for the reader.

Figure B.5.6 shows a 1-bit ALU derived by combining the adder with the earlier components. Sometimes designers also want the ALU to perform a few more simple operations, such as generating 0. The easiest way to add an operation is to expand the multiplexor controlled by the Operation line and, for this example, to connect 0 directly to the new input of that expanded multiplexor.



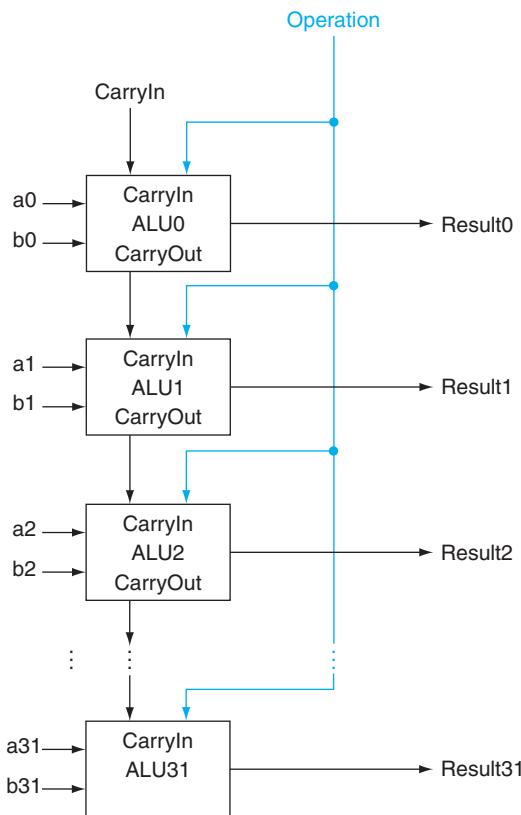
**FIGURE B.5.6** A 1-bit ALU that performs AND, OR, and addition (see Figure B.5.5).

## A 32-Bit ALU

Now that we have completed the 1-bit ALU, the full 32-bit ALU is created by connecting adjacent “black boxes.” Using  $x_i$  to mean the  $i$ th bit of  $x$ , Figure B.5.7 shows a 32-bit ALU. Just as a single stone can cause ripples to radiate to the shores of a quiet lake, a single carry out of the least significant bit (Result<sub>0</sub>) can ripple all the way through the adder, causing a carry out of the most significant bit (Result<sub>31</sub>). Hence, the adder created by directly linking the carries of 1-bit adders is called a *ripple carry* adder. We’ll see a faster way to connect the 1-bit adders starting on page B-38.

Subtraction is the same as adding the negative version of an operand, and this is how adders perform subtraction. Recall that the shortcut for negating a two’s complement number is to invert each bit (sometimes called the *one’s complement*) and then add 1. To invert each bit, we simply add a 2:1 multiplexor that chooses between  $b$  and  $\bar{b}$ , as Figure B.5.8 shows.

Suppose we connect 32 of these 1-bit ALUs, as we did in Figure B.5.7. The added multiplexor gives the option of  $b$  or its inverted value, depending on Binvert, but

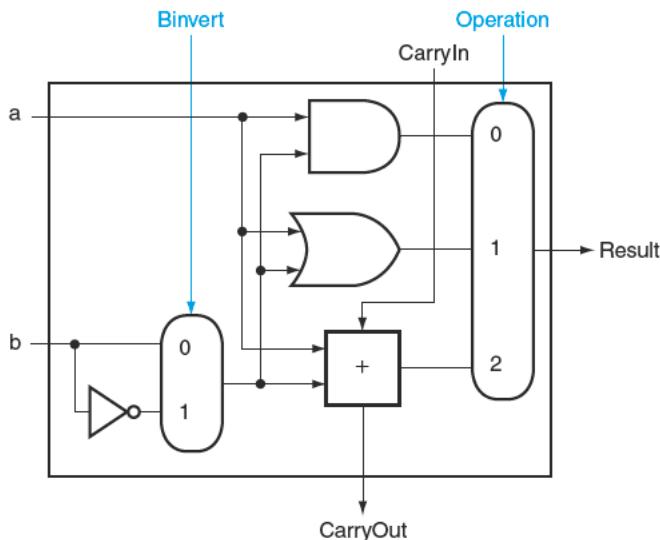


**FIGURE B.5.7 A 32-bit ALU constructed from 32 1-bit ALUs.** CarryOut of the less significant bit is connected to the CarryIn of the more significant bit. This organization is called ripple carry.

this is only one step in negating a two's complement number. Notice that the least significant bit still has a CarryIn signal, even though it's unnecessary for addition. What happens if we set this CarryIn to 1 instead of 0? The adder will then calculate  $a + b + 1$ . By selecting the inverted version of  $b$ , we get exactly what we want:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

The simplicity of the hardware design of a two's complement adder helps explain why two's complement representation has become the universal standard for integer computer arithmetic.



**FIGURE B.5.8 A 1-bit ALU that performs AND, OR, and addition on *a* and *b* or *a* and  $\bar{b}$ .** By selecting  $\bar{b}$  (*Binvert* = 1) and setting *CarryIn* to 1 in the least significant bit of the ALU, we get two's complement subtraction of *b* from *a* instead of addition of *b* to *a*.

A MIPS ALU also needs a NOR function. Instead of adding a separate gate for NOR, we can reuse much of the hardware already in the ALU, like we did for subtract. The insight comes from the following truth about NOR:

$$(\overline{a + b}) = \overline{a} \cdot \overline{b}$$

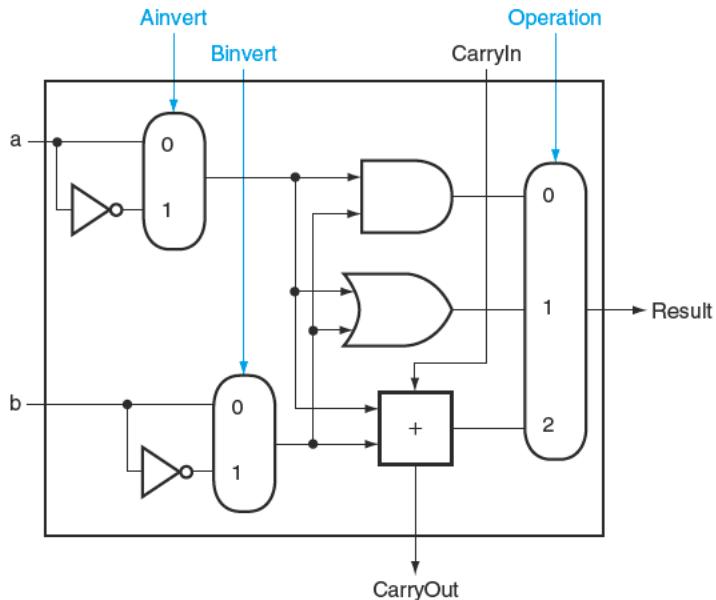
That is, NOT (*a* OR *b*) is equivalent to NOT *a* AND NOT *b*. This fact is called DeMorgan's theorem and is explored in the exercises in more depth.

Since we have AND and NOT *b*, we only need to add NOT *a* to the ALU. Figure B.5.9 shows that change.

### Tailoring the 32-Bit ALU to MIPS

These four operations—add, subtract, AND, OR—are found in the ALU of almost every computer, and the operations of most MIPS instructions can be performed by this ALU. But the design of the ALU is incomplete.

One instruction that still needs support is the set on less than instruction (*slt*). Recall that the operation produces 1 if *rs* < *rt*, and 0 otherwise. Consequently, *slt* will set all but the least significant bit to 0, with the least significant bit set according to the comparison. For the ALU to perform *slt*, we first need to expand the three-input



**FIGURE B.5.9 A 1-bit ALU that performs AND, OR, and addition on *a* and *b* or  $\bar{a}$  and  $\bar{b}$ .** By selecting  $\bar{a}$  (*Ainvert* = 1) and  $\bar{b}$  (*Binvert* = 1), we get a NOR *b* instead of a AND *b*.

multiplexor in Figure B.5.8 to add an input for the *s lt* result. We call that new input *Less* and use it only for *s lt*.

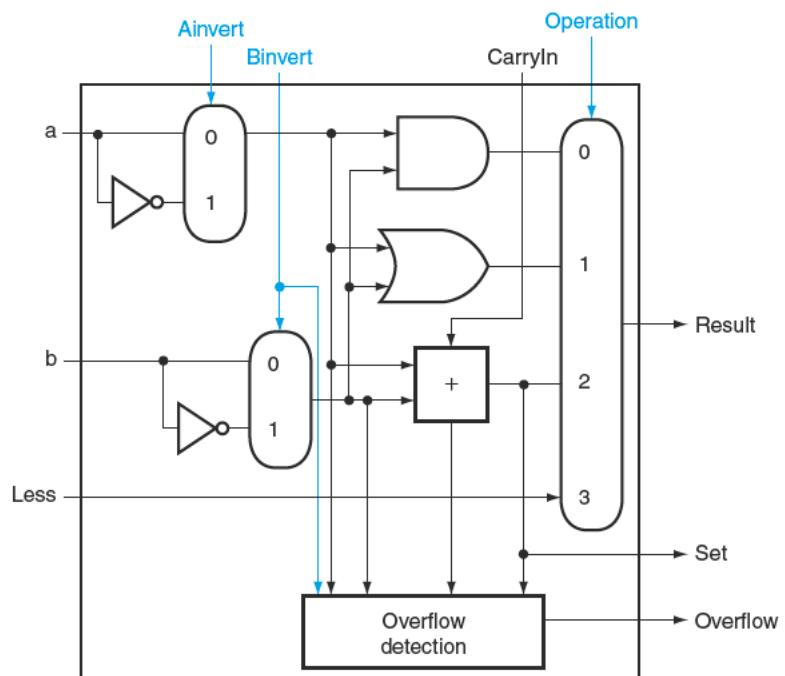
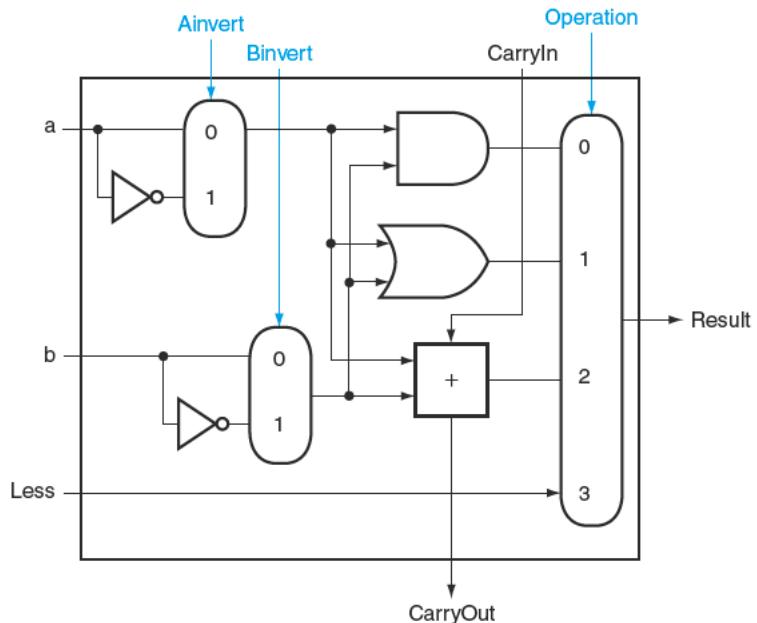
The top drawing of Figure B.5.10 shows the new 1-bit ALU with the expanded multiplexor. From the description of *s lt* above, we must connect 0 to the *Less* input for the upper 31 bits of the ALU, since those bits are always set to 0. What remains to consider is how to compare and set the *least significant bit* for set on less than instructions.

What happens if we subtract *b* from *a*? If the difference is negative, then  $a < b$  since

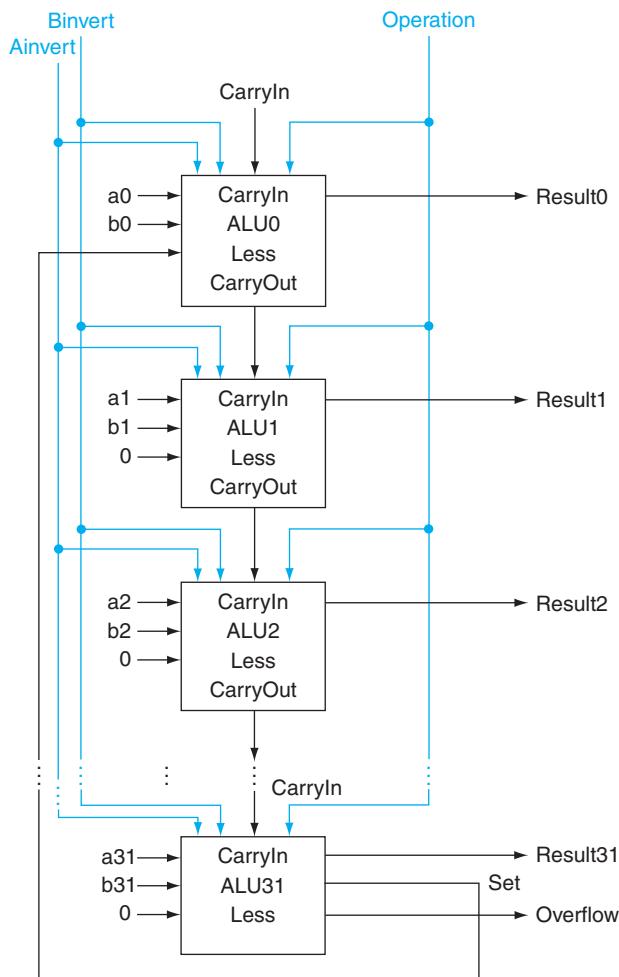
$$\begin{aligned}(a - b) < 0 &\Rightarrow ((a - b) + b) < (0 + b) \\ &\Rightarrow a < b\end{aligned}$$

We want the least significant bit of *a* set on less than operation to be a 1 if  $a < b$ ; that is, a 1 if  $a - b$  is negative and a 0 if it's positive. This desired result corresponds exactly to the sign bit values: 1 means negative and 0 means positive. Following this line of argument, we need only connect the sign bit from the adder output to the least significant bit to get set on less than.

Unfortunately, the *Result* output from the most significant ALU bit in the top of Figure B.5.10 for the *s lt* operation is *not* the output of the adder; the ALU output for the *s lt* operation is obviously the input value *Less*.



**FIGURE B.5.10 (Top) A 1-bit ALU that performs AND, OR, and addition on  $a$  and  $b$  or  $\bar{b}$ , and (bottom) a 1-bit ALU for the most significant bit.** The top drawing includes a direct input that is connected to perform the set on less than operation (see Figure B.5.11); the bottom has a direct output from the adder for the less than comparison called Set. (See Exercise B.24 at the end of this appendix to see how to calculate overflow with fewer inputs.)



**FIGURE B.5.11 A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top of Figure B.5.10 and one 1-bit ALU in the bottom of that figure.** The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit. If the ALU performs  $a - b$  and we select the input 3 in the multiplexer in Figure B.5.10, then Result = 0 ... 001 if  $a < b$ , and Result = 0 ... 000 otherwise.

Thus, we need a new 1-bit ALU for the most significant bit that has an extra output bit: the adder output. The bottom drawing of Figure B.5.10 shows the design, with this new adder output line called *Set*, and used only for *s lt*. As long as we need a special ALU for the most significant bit, we added the overflow detection logic since it is also associated with that bit.

Alas, the test of less than is a little more complicated than just described because of overflow, as we explore in the exercises. [Figure B.5.11](#) shows the 32-bit ALU.

Notice that every time we want the ALU to subtract, we set both CarryIn and Binvert to 1. For adds or logical operations, we want both control lines to be 0. We can therefore simplify control of the ALU by combining the CarryIn and Binvert to a single control line called *Bnegate*.

To further tailor the ALU to the MIPS instruction set, we must support conditional branch instructions. These instructions branch either if two registers are equal or if they are unequal. The easiest way to test equality with the ALU is to subtract b from a and then test to see if the result is 0, since

$$(a - b = 0) \Rightarrow a = b$$

Thus, if we add hardware to test if the result is 0, we can test for equality. The simplest way is to OR all the outputs together and then send that signal through an inverter:

$$\text{Zero} = \overline{(\text{Result31} + \text{Result30} + \dots + \text{Result2} + \text{Result1} + \text{Result0})}$$

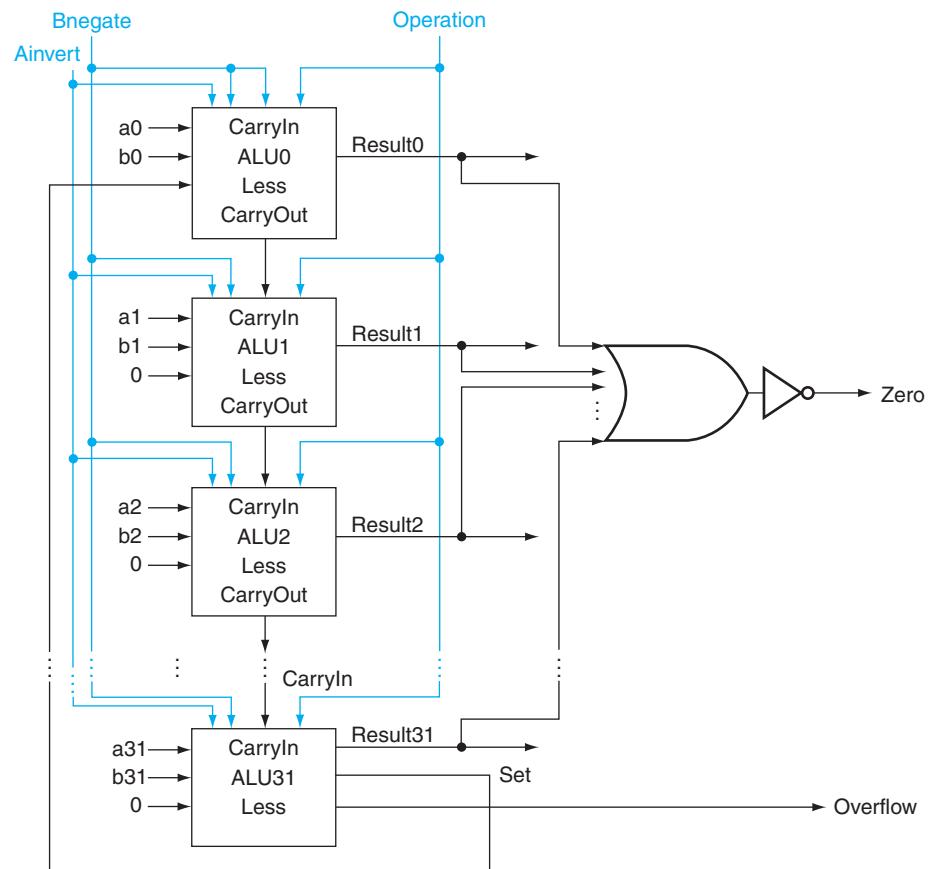
[Figure B.5.12](#) shows the revised 32-bit ALU. We can think of the combination of the 1-bit Ainvert line, the 1-bit Binvert line, and the 2-bit Operation lines as 4-bit control lines for the ALU, telling it to perform add, subtract, AND, OR, or set on less than. [Figure B.5.13](#) shows the ALU control lines and the corresponding ALU operation.

Finally, now that we have seen what is inside a 32-bit ALU, we will use the universal symbol for a complete ALU, as shown in [Figure B.5.14](#).

## Defining the MIPS ALU in Verilog

[Figure B.5.15](#) shows how a combinational MIPS ALU might be specified in Verilog; such a specification would probably be compiled using a standard parts library that provided an adder, which could be instantiated. For completeness, we show the ALU control for MIPS in [Figure B.5.16](#), which is used in Chapter 4, where we build a Verilog version of the MIPS datapath.

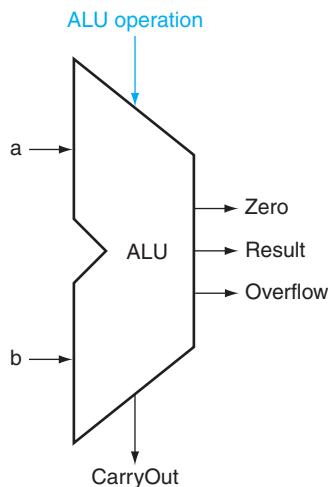
The next question is, “How quickly can this ALU add two 32-bit operands?” We can determine the a and b inputs, but the CarryIn input depends on the operation in the adjacent 1-bit adder. If we trace all the way through the chain of dependencies, we connect the most significant bit to the least significant bit, so the most significant bit of the sum must wait for the *sequential* evaluation of all 32 1-bit adders. This sequential chain reaction is too slow to be used in time-critical hardware. The next section explores how to speed-up addition. This topic is not crucial to understanding the rest of the appendix and may be skipped.



**FIGURE B.5.12** The final 32-bit ALU. This adds a Zero detector to Figure B.5.11.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

**FIGURE B.5.13** The values of the three ALU control lines, Bnegate, and Operation, and the corresponding ALU operations.



**FIGURE B.5.14** The symbol commonly used to represent an ALU, as shown in Figure B.5.12. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

```

module MIPSALU (ALUctl, A, B, ALUOut, Zero);
  input [3:0] ALUctl;
  input [31:0] A,B;
  output reg [31:0] ALUOut;
  output Zero;

  assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
  always @ (ALUctl, A, B) begin //reevaluate if these change
    case (ALUctl)
      0: ALUOut <= A & B;
      1: ALUOut <= A | B;
      2: ALUOut <= A + B;
      6: ALUOut <= A - B;
      7: ALUOut <= A < B ? 1 : 0;
      12: ALUOut <= ~(A | B); // result is nor
    default: ALUOut <= 0;
    endcase
  end
endmodule

```

**FIGURE B.5.15** A Verilog behavioral definition of a MIPS ALU.

```

module ALUControl (ALUOp, FuncCode, ALUCl);
    input [1:0] ALUOp;
    input [5:0] FuncCode;
    output [3:0] reg ALUCl;
    always case (FuncCode)
        32: ALUOp<=2; // add
        34: ALUOp<=6; //subtract
        36: ALUOP<=0; // and
        37: ALUOp<=1; // or
        39: ALUOp<=12; // nor
        42: ALUOp<=7; // slt
        default: ALUOp<=15; // should not happen
    endcase
endmodule

```

**FIGURE B.5.16** The MIPS ALU control: a simple piece of combinational control logic.

### Check Yourself

Suppose you wanted to add the operation NOT (a AND b), called NAND. How could the ALU change to support it?

1. No change. You can calculate NAND quickly using the current ALU since  $(a \cdot b) = \bar{a} + \bar{b}$  and we already have NOT a, NOT b, and OR.
2. You must expand the big multiplexor to add another input, and then add new logic to calculate NAND.

## B.6

### Faster Addition: Carry Lookahead

The key to speeding up addition is determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the  $\log_2$  of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry.

A key to understanding fast-carry schemes is to remember that, unlike software, hardware executes in parallel whenever inputs change.

#### Fast Carry Using “Infinite” Hardware

As we mentioned earlier, any equation can be represented in two levels of logic. Since the only external inputs are the two operands and the CarryIn to the least

significant bit of the adder, in theory we could calculate the CarryIn values to all the remaining bits of the adder in just two levels of logic.

For example, the CarryIn for bit 2 of the adder is exactly the CarryOut of bit 1, so the formula is

$$\text{CarryIn2} = (b_1 \cdot \text{CarryIn1}) + (a_1 \cdot \text{CarryIn1}) + (a_1 \cdot b_1)$$

Similarly, CarryIn1 is defined as

$$\text{CarryIn1} = (b_0 \cdot \text{CarryIn0}) + (a_0 \cdot \text{CarryIn0}) + (a_0 \cdot b_0)$$

Using the shorter and more traditional abbreviation of  $ci$  for  $\text{CarryIn}_i$ , we can rewrite the formulas as

$$\begin{aligned} c_2 &= (b_1 \cdot c_1) + (a_1 \cdot c_1) + (a_1 \cdot b_1) \\ c_1 &= (b_0 \cdot c_0) + (a_0 \cdot c_0) + (a_0 \cdot b_0) \end{aligned}$$

Substituting the definition of  $c_1$  for the first equation results in this formula:

$$\begin{aligned} c_2 &= (a_1 \cdot a_0 \cdot b_0) + (a_1 \cdot a_0 \cdot c_0) \cdot (a_1 \cdot b_0 \cdot c_0) \\ &\quad + (b_1 \cdot a_0 \cdot b_0) + (b_1 \cdot a_0 \cdot c_0) + (b_1 \cdot b_0 \cdot c_0) + (a_1 \cdot b_1) \end{aligned}$$

You can imagine how the equation expands as we get to higher bits in the adder; it grows rapidly with the number of bits. This complexity is reflected in the cost of the hardware for fast carry, making this simple scheme prohibitively expensive for wide adders.

## Fast Carry Using the First Level of Abstraction: Propagate and Generate

Most fast-carry schemes limit the complexity of the equations to simplify the hardware, while still making substantial speed improvements over ripple carry. One such scheme is a *carry-lookahead adder*. In Chapter 1, we said computer systems cope with complexity by using levels of abstraction. A carry-lookahead adder relies on levels of abstraction in its implementation.

Let's factor our original equation as a first step:

$$\begin{aligned} ci + 1 &= (bi \cdot ci) + (ai \cdot ci) + (ai \cdot bi) \\ &= (ai \cdot bi) + (ai + bi) \cdot ci \end{aligned}$$

If we were to rewrite the equation for  $c_2$  using this formula, we would see some repeated patterns:

$$c_2 = (a_1 \cdot b_1) + (a_1 \cdot b_1) \cdot ((a_0 \cdot b_0) + (a_0 + b_0) \cdot c_0)$$

Note the repeated appearance of  $(ai \cdot bi)$  and  $(ai + bi)$  in the formula above. These two important factors are traditionally called *generate* ( $gi$ ) and *propagate* ( $pi$ ):

$$\begin{aligned}g_i &= a_i \cdot b_i \\p_i &= a_i + b_i\end{aligned}$$

Using them to define  $c_i + 1$ , we get

$$c_i + 1 = g_i + p_i \cdot c_i$$

To see where the signals get their names, suppose  $g_i$  is 1. Then

$$c_i + 1 = g_i + p_i \cdot c_i = 1 + p_i \cdot c_i = 1$$

That is, the adder *generates* a CarryOut ( $c_i + 1$ ) independent of the value of CarryIn ( $c_i$ ). Now suppose that  $g_i$  is 0 and  $p_i$  is 1. Then

$$c_i + 1 = g_i + p_i \cdot c_i = 0 + 1 \cdot c_i = c_i$$

That is, the adder *propagates* CarryIn to a CarryOut. Putting the two together, CarryIn*i* + 1 is a 1 if either  $g_i$  is 1 or both  $p_i$  is 1 and CarryIn*i* is 1.

As an analogy, imagine a row of dominoes set on edge. The end domino can be tipped over by pushing one far away, provided there are no gaps between the two. Similarly, a carry out can be made true by a generate far away, provided all the propagates between them are true.

Relying on the definitions of propagate and generate as our first level of abstraction, we can express the CarryIn signals more economically. Let's show it for 4 bits:

$$\begin{aligned}c_1 &= g_0 + (p_0 \cdot c_0) \\c_2 &= g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot c_0) \\c_3 &= g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot c_0) \\c_4 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\&\quad + (p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0)\end{aligned}$$

These equations just represent common sense: CarryIn*i* is a 1 if some earlier adder generates a carry and all intermediary adders propagate a carry. [Figure B.6.1](#) uses plumbing to try to explain carry lookahead.

Even this simplified form leads to large equations and, hence, considerable logic even for a 16-bit adder. Let's try moving to two levels of abstraction.

## Fast Carry Using the Second Level of Abstraction

First, we consider this 4-bit adder with its carry-lookahead logic as a single building block. If we connect them in ripple carry fashion to form a 16-bit adder, the add will be faster than the original with a little more hardware.

To go faster, we'll need carry lookahead at a higher level. To perform carry lookahead for 4-bit adders, we need to propagate and generate signals at this higher level. Here they are for the four 4-bit adder blocks:

$$\begin{aligned} P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\ P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\ P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\ P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12} \end{aligned}$$

That is, the “super” propagate signal for the 4-bit abstraction ( $P_i$ ) is true only if each of the bits in the group will propagate a carry.

For the “super” generate signal ( $G_i$ ), we care only if there is a carry out of the most significant bit of the 4-bit group. This obviously occurs if generate is true for that most significant bit; it also occurs if an earlier generate is true *and* all the intermediate propagates, including that of the most significant bit, are also true:

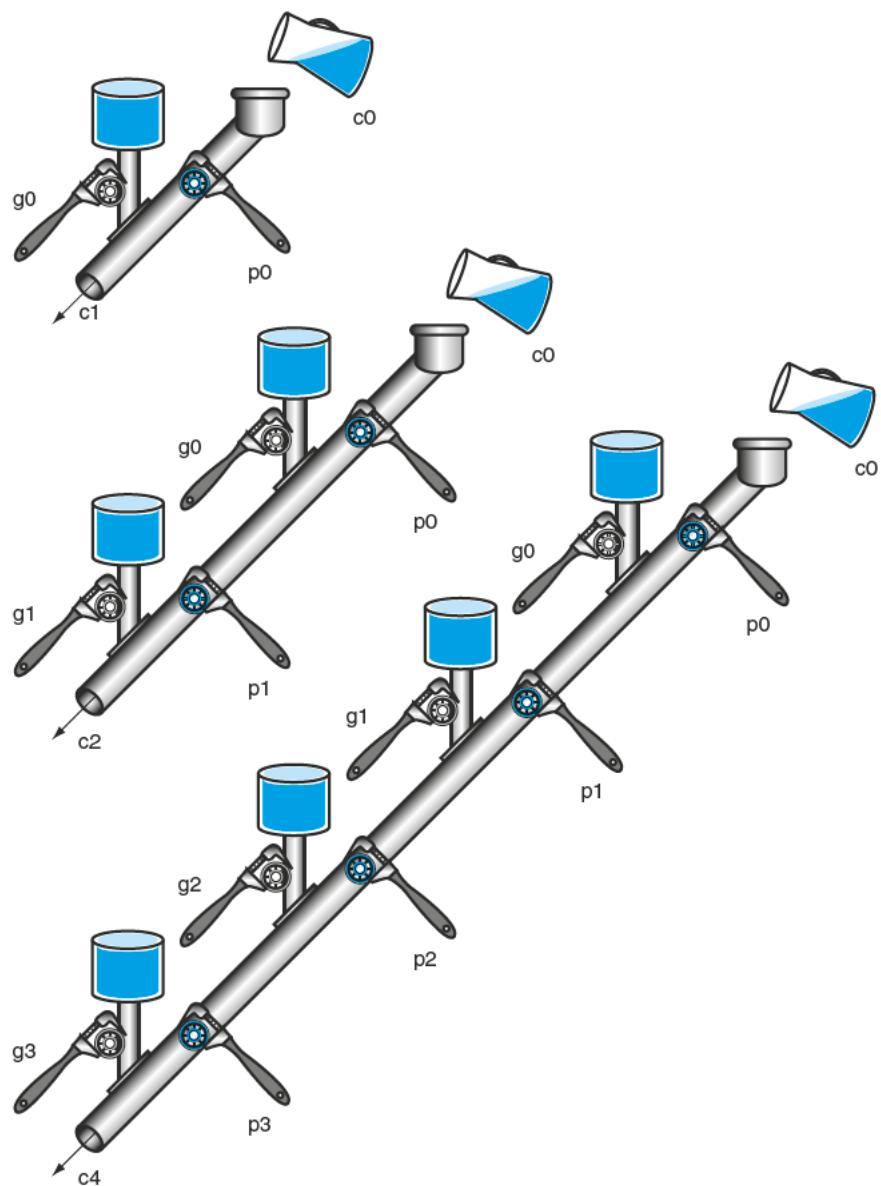
$$\begin{aligned} G_0 &= g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0) \\ G_1 &= g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4) \\ G_2 &= g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8) \\ G_3 &= g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}) \end{aligned}$$

[Figure B.6.2](#) updates our plumbing analogy to show  $P_0$  and  $G_0$ .

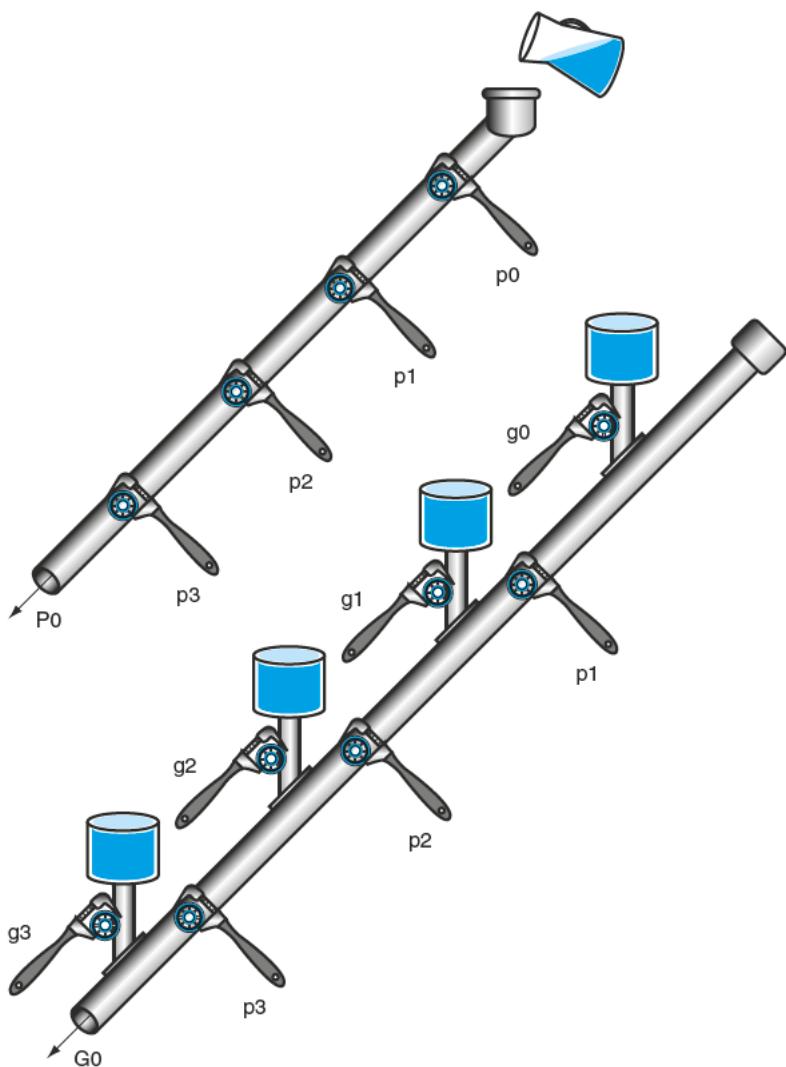
Then the equations at this higher level of abstraction for the carry in for each 4-bit group of the 16-bit adder ( $C_1, C_2, C_3, C_4$  in [Figure B.6.3](#)) are very similar to the carry out equations for each bit of the 4-bit adder ( $c_1, c_2, c_3, c_4$ ) on page B-40:

$$\begin{aligned} C_1 &= G_0 + (P_0 \cdot c_0) \\ C_2 &= G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0) \\ C_3 &= G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot c_0) \\ C_4 &= G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) \\ &\quad + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0) \end{aligned}$$

[Figure B.6.3](#) shows 4-bit adders connected with such a carry-lookahead unit. The exercises explore the speed differences between these carry schemes, different notations for multibit propagate and generate signals, and the design of a 64-bit adder.



**FIGURE B.6.1 A plumbing analogy for carry lookahead for 1 bit, 2 bits, and 4 bits using water pipes and valves.** The wrenches are turned to open and close valves. Water is shown in color. The output of the pipe ( $c_i + 1$ ) will be full if either the nearest generate value ( $g_i$ ) is turned on or if the  $i$  propagate value ( $p_i$ ) is on and there is water further upstream, either from an earlier generate or a propagate with water behind it. CarryIn ( $c_0$ ) can result in a carry out without the help of any generates, but with the help of *all* propagates.



**FIGURE B.6.2 A plumbing analogy for the next-level carry-lookahead signals  $P_0$  and  $G_0$ .**  
P0 is open only if all four propagates ( $p_i$ ) are open, while water flows in G0 only if at least one generate ( $g_i$ ) is open and all the propagates downstream from that generate are open.

**EXAMPLE****ANSWER****Both Levels of the Propagate and Generate**

Determine the  $gi$ ,  $pi$ ,  $Pi$ , and  $Gi$  values of these two 16-bit numbers:

$$\begin{array}{ll} a: & 0001 \quad 1010 \quad 0011 \quad 0011_{\text{two}} \\ b: & 1110 \quad 0101 \quad 1110 \quad 1011_{\text{two}} \end{array}$$

Also, what is CarryOut15 (C4)?

Aligning the bits makes it easy to see the values of generate  $gi$  ( $ai \cdot bi$ ) and propagate  $pi$  ( $ai + bi$ ):

$$\begin{array}{ll} a: & 0001 \quad 1010 \quad 0011 \quad 0011 \\ b: & 1110 \quad 0101 \quad 1110 \quad 1011 \\ gi: & 0000 \quad 0000 \quad 0010 \quad 0011 \\ pi: & 1111 \quad 1111 \quad 1111 \quad 1011 \end{array}$$

where the bits are numbered 15 to 0 from left to right. Next, the “super” propagates ( $P3$ ,  $P2$ ,  $P1$ ,  $P0$ ) are simply the AND of the lower-level propagates:

$$\begin{aligned} P3 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P2 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P1 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P0 &= 1 \cdot 0 \cdot 1 \cdot 1 = 0 \end{aligned}$$

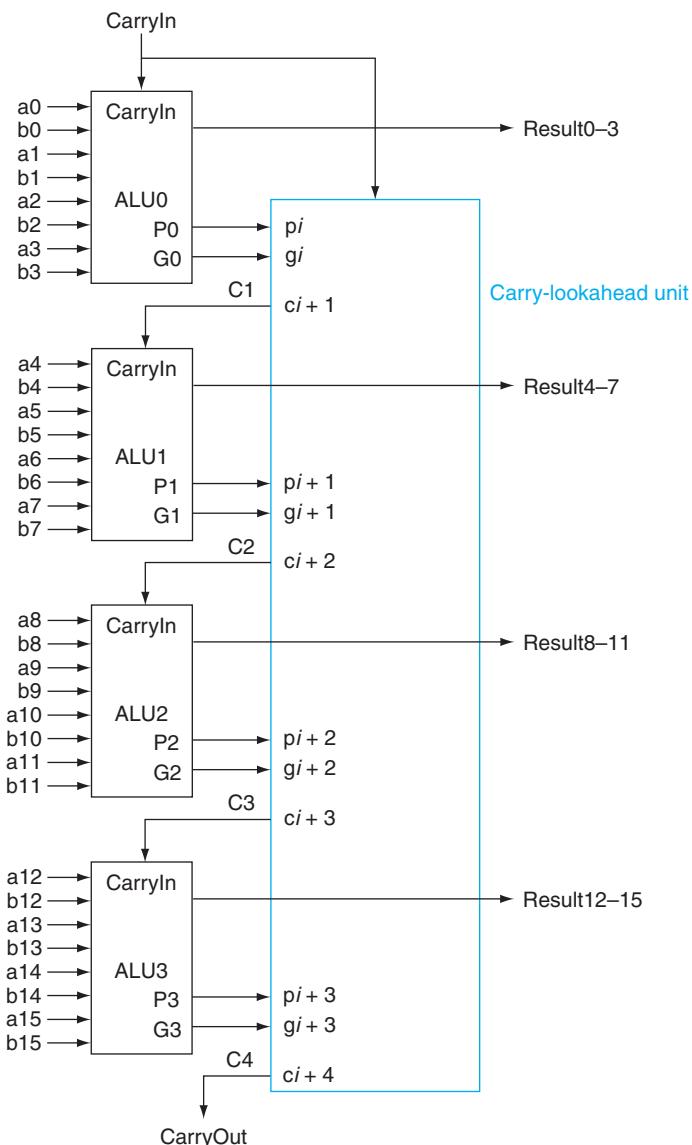
The “super” generates are more complex, so use the following equations:

$$\begin{aligned} G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\ G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\ G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\ G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \end{aligned}$$

Finally, CarryOut15 is

$$\begin{aligned} C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\ &\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\ &= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\ &= 0 + 0 + 1 + 0 + 0 = 1 \end{aligned}$$

Hence, there *is* a carry out when adding these two 16-bit numbers.



**FIGURE B.6.3 Four 4-bit ALUs using carry lookahead to form a 16-bit adder.** Note that the carries come from the carry-lookahead unit, not from the 4-bit ALUs.

The reason carry lookahead can make carries faster is that all logic begins evaluating the moment the clock cycle begins, and the result will not change once the output of each gate stops changing. By taking the shortcut of going through fewer gates to send the carry in signal, the output of the gates will stop changing sooner, and hence the time for the adder can be less.

To appreciate the importance of carry lookahead, we need to calculate the relative performance between it and ripple carry adders.

## EXAMPLE

## ANSWER

### Speed of Ripple Carry versus Carry Lookahead

One simple way to model time for logic is to assume each AND or OR gate takes the same time for a signal to pass through it. Time is estimated by simply counting the number of gates along the path through a piece of logic. Compare the number of *gate delays* for paths of two 16-bit adders, one using ripple carry and one using two-level carry lookahead.

Figure B.5.5 on page B-28 shows that the carry out signal takes two gate delays per bit. Then the number of gate delays between a carry in to the least significant bit and the carry out of the most significant is  $16 \times 2 = 32$ .

For carry lookahead, the carry out of the most significant bit is just  $C_4$ , defined in the example. It takes two levels of logic to specify  $C_4$  in terms of  $P_i$  and  $G_i$  (the OR of several AND terms).  $P_i$  is specified in one level of logic (AND) using  $p_i$ , and  $G_i$  is specified in two levels using  $p_i$  and  $g_i$ , so the worst case for this next level of abstraction is two levels of logic.  $p_i$  and  $g_i$  are each one level of logic, defined in terms of  $a_i$  and  $b_i$ . If we assume one gate delay for each level of logic in these equations, the worst case is  $2 + 2 + 1 = 5$  gate delays.

Hence, for the path from carry in to carry out, the 16-bit addition by a carry-lookahead adder is six times faster, using this very simple estimate of hardware speed.

### Summary

Carry lookahead offers a faster path than waiting for the carries to ripple through all 32 1-bit adders. This faster path is paved by two signals, generate and propagate.

The former creates a carry regardless of the carry input, and the latter passes a carry along. Carry lookahead also gives another example of how abstraction is important in computer design to cope with complexity.

Using the simple estimate of hardware speed above with gate delays, what is the relative performance of a ripple carry 8-bit add versus a 64-bit add using carry-lookahead logic?

1. A 64-bit carry-lookahead adder is three times faster: 8-bit adds are 16 gate delays and 64-bit adds are 7 gate delays.
2. They are about the same speed, since 64-bit adds need more levels of logic in the 16-bit adder.
3. 8-bit adds are faster than 64 bits, even with carry lookahead.

### Check Yourself

**Elaboration:** We have now accounted for all but one of the arithmetic and logical operations for the core MIPS instruction set: the ALU in [Figure B.5.14](#) omits support of shift instructions. It would be possible to widen the ALU multiplexor to include a left shift by 1 bit or a right shift by 1 bit. But hardware designers have created a circuit called a *barrel shifter*, which can shift from 1 to 31 bits in no more time than it takes to add two 32-bit numbers, so shifting is normally done outside the ALU.

**Elaboration:** The logic equation for the Sum output of the full adder on page B-28 can be expressed more simply by using a more powerful gate than AND and OR. An exclusive OR gate is true if the two operands disagree; that is,

$$x \neq y \Rightarrow 1 \text{ and } x == y \Rightarrow 0$$

In some technologies, exclusive OR is more efficient than two levels of AND and OR gates. Using the symbol  $\oplus$  to represent exclusive OR, here is the new equation:

$$\text{Sum} = a \oplus b \oplus \text{CarryIn}$$

Also, we have drawn the ALU the traditional way, using gates. Computers are designed today in CMOS transistors, which are basically switches. CMOS ALU and barrel shifters take advantage of these switches and have many fewer multiplexors than shown in our designs, but the design principles are similar.

**Elaboration:** Using lowercase and uppercase to distinguish the hierarchy of generate and propagate symbols breaks down when you have more than two levels. An alternate notation that scales is  $g_{i..j}$  and  $p_{i..j}$  for the generate and propagate signals for bits  $i$  to  $j$ . Thus,  $g_{1..1}$  is generated for bit 1,  $g_{4..1}$  is for bits 4 to 1, and  $g_{16..1}$  is for bits 16 to 1.

**B.7****Clocks**

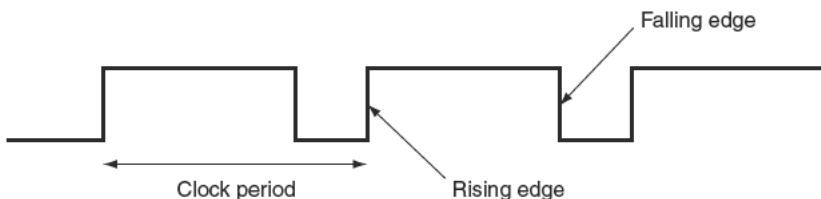
Before we discuss memory elements and sequential logic, it is useful to discuss briefly the topic of clocks. This short section introduces the topic and is similar to the discussion found in Section 4.2. More details on clocking and timing methodologies are presented in Section B.11.

**Clocks** are needed in sequential logic to decide when an element that contains state should be updated. A clock is simply a free-running signal with a fixed *cycle time*; the *clock frequency* is simply the inverse of the cycle time. As shown in Figure B.7.1, the *clock cycle time* or *clock period* is divided into two portions: when the clock is high and when the clock is low. In this text, we use only **edge-triggered clocking**. This means that all state changes occur on a clock edge. We use an edge-triggered methodology because it is simpler to explain. Depending on the technology, it may or may not be the best choice for a **clocking methodology**.

**edge-triggered clocking** A clocking scheme in which all state changes occur on a clock edge.

**clocking methodology**

The approach used to determine when data is valid and stable relative to the clock.



**FIGURE B.7.1 A clock signal oscillates between high and low values.** The clock period is the time for one full cycle. In an edge-triggered design, either the rising or falling edge of the clock is active and causes state to be changed.

**state element**  
A memory element.

**synchronous system**  
A memory system that employs clocks and where data signals are read only when the clock indicates that the signal values are stable.

In an edge-triggered methodology, either the rising edge or the falling edge of the clock is *active* and causes state changes to occur. As we will see in the next section, the **state elements** in an edge-triggered design are implemented so that the contents of the state elements only change on the active clock edge. The choice of which edge is active is influenced by the implementation technology and does not affect the concepts involved in designing the logic.

The clock edge acts as a sampling signal, causing the value of the data input to a state element to be sampled and stored in the state element. Using an edge trigger means that the sampling process is essentially instantaneous, eliminating problems that could occur if signals were sampled at slightly different times.

The major constraint in a clocked system, also called a **synchronous system**, is that the signals that are written into state elements must be *valid* when the active

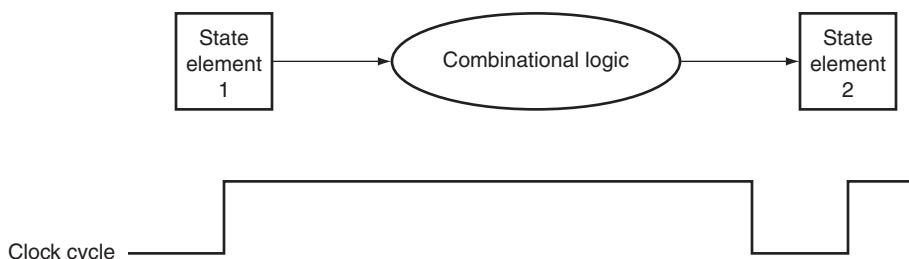
clock edge occurs. A signal is valid if it is stable (i.e., not changing), and the value will not change again until the inputs change. Since combinational circuits cannot have feedback, if the inputs to a combinational logic unit are not changed, the outputs will eventually become valid.

[Figure B.7.2](#) shows the relationship among the state elements and the combinational logic blocks in a synchronous, sequential logic design. The state elements, whose outputs change only after the clock edge, provide valid inputs to the combinational logic block. To ensure that the values written into the state elements on the active clock edge are valid, the clock must have a long enough period so that all the signals in the combinational logic block stabilize, and then the clock edge samples those values for storage in the state elements. This constraint sets a lower bound on the length of the clock period, which must be long enough for all state element inputs to be valid.

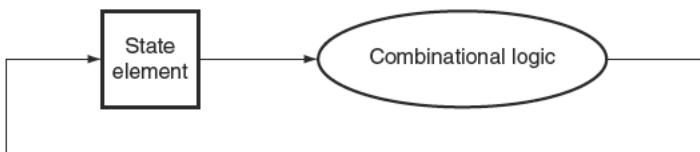
In the rest of this appendix, as well as in Chapter 4, we usually omit the clock signal, since we are assuming that all state elements are updated on the same clock edge. Some state elements will be written on every clock edge, while others will be written only under certain conditions (such as a register being updated). In such cases, we will have an explicit write signal for that state element. The write signal must still be gated with the clock so that the update occurs only on the clock edge if the write signal is active. We will see how this is done and used in the next section.

One other advantage of an edge-triggered methodology is that it is possible to have a state element that is used as both an input and output to the same combinational logic block, as shown in [Figure B.7.3](#). In practice, care must be taken to prevent races in such situations and to ensure that the clock period is long enough; this topic is discussed further in Section B.11.

Now that we have discussed how clocking is used to update state elements, we can discuss how to construct the state elements.



**FIGURE B.7.2 The inputs to a combinational logic block come from a state element, and the outputs are written into a state element.** The clock edge determines when the contents of the state elements are updated.



**FIGURE B.7.3** An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to undetermined data values. Of course, the clock cycle must still be long enough so that the input values are stable when the active clock edge occurs.

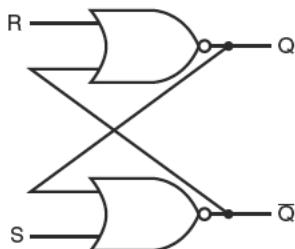
**register file** A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.

**Elaboration:** Occasionally, designers find it useful to have a small number of state elements that change on the opposite clock edge from the majority of the state elements. Doing so requires extreme care, because such an approach has effects on both the inputs and the outputs of the state element. Why then would designers ever do this? Consider the case where the amount of combinational logic before and after a state element is small enough so that each could operate in one-half clock cycle, rather than the more usual full clock cycle. Then the state element can be written on the clock edge corresponding to a half clock cycle, since the inputs and outputs will both be usable after one-half clock cycle. One common place where this technique is used is in **register files**, where simply reading or writing the register file can often be done in half the normal clock cycle. Chapter 4 makes use of this idea to reduce the pipelining overhead.

## B.8

# Memory Elements: Flip-Flops, Latches, and Registers

In this section and the next, we discuss the basic principles behind memory elements, starting with flip-flops and latches, moving on to register files, and finishing with memories. All memory elements store state: the output from any memory element depends both on the inputs and on the value that has been stored inside the memory element. Thus all logic blocks containing a memory element contain state and are sequential.



**FIGURE B.8.1** A pair of cross-coupled NOR gates can store an internal value. The value stored on the output  $Q$  is recycled by inverting it to obtain  $\bar{Q}$  and then inverting  $\bar{Q}$  to obtain  $Q$ . If either  $R$  or  $\bar{Q}$  is asserted,  $Q$  will be deasserted and vice versa.

The simplest type of memory elements are *unclocked*; that is, they do not have any clock input. Although we only use clocked memory elements in this text, an unclocked latch is the simplest memory element, so let's look at this circuit first. [Figure B.8.1](#) shows an *S-R latch* (set-reset latch), built from a pair of NOR gates (OR gates with inverted outputs). The outputs  $Q$  and  $\bar{Q}$  represent the value of the stored state and its complement. When neither  $S$  nor  $R$  are asserted, the cross-coupled NOR gates act as inverters and store the previous values of  $Q$  and  $\bar{Q}$ .

For example, if the output,  $Q$ , is true, then the bottom inverter produces a false output (which is  $\bar{Q}$ ), which becomes the input to the top inverter, which produces a true output, which is  $Q$ , and so on. If  $S$  is asserted, then the output  $Q$  will be asserted and  $\bar{Q}$  will be deasserted, while if  $R$  is asserted, then the output  $\bar{Q}$  will be asserted and  $Q$  will be deasserted. When  $S$  and  $R$  are both deasserted, the last values of  $Q$  and  $\bar{Q}$  will continue to be stored in the cross-coupled structure. Asserting  $S$  and  $R$  simultaneously can lead to incorrect operation: depending on how  $S$  and  $R$  are deasserted, the latch may oscillate or become metastable (this is described in more detail in [Section B.11](#)).

This cross-coupled structure is the basis for more complex memory elements that allow us to store data signals. These elements contain additional gates used to store signal values and to cause the state to be updated only in conjunction with a clock. The next section shows how these elements are built.

## Flip-Flops and Latches

**Flip-flops** and **latches** are the simplest memory elements. In both flip-flops and latches, the output is equal to the value of the stored state inside the element. Furthermore, unlike the S-R latch described above, all the latches and flip-flops we will use from this point on are clocked, which means that they have a clock input and the change of state is triggered by that clock. The difference between a flip-flop and a latch is the point at which the clock causes the state to actually change. In a clocked latch, the state is changed whenever the appropriate inputs change and the clock is asserted, whereas in a flip-flop, the state is changed only on a clock edge. Since throughout this text we use an edge-triggered timing methodology where state is only updated on clock edges, we need only use flip-flops. Flip-flops are often built from latches, so we start by describing the operation of a simple clocked latch and then discuss the operation of a flip-flop constructed from that latch.

For computer applications, the function of both flip-flops and latches is to store a signal. A *D latch* or **D flip-flop** stores the value of its data input signal in the internal memory. Although there are many other types of latch and flip-flop, the D type is the only basic building block that we will need. A D latch has two inputs and two outputs. The inputs are the data value to be stored (called  $D$ ) and a clock signal (called  $C$ ) that indicates when the latch should read the value on the  $D$  input and store it. The outputs are simply the value of the internal state ( $Q$ )

**flip-flop** A memory element for which the output is equal to the value of the stored state inside the element and for which the internal state is changed only on a clock edge.

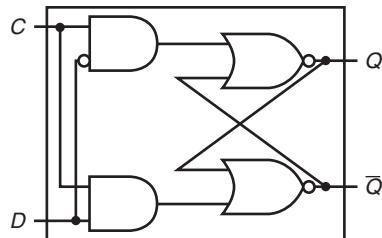
**latch** A memory element in which the output is equal to the value of the stored state inside the element and the state is changed whenever the appropriate inputs change and the clock is asserted.

**D flip-flop** A flip-flop with one data input that stores the value of that input signal in the internal memory when the clock edge occurs.

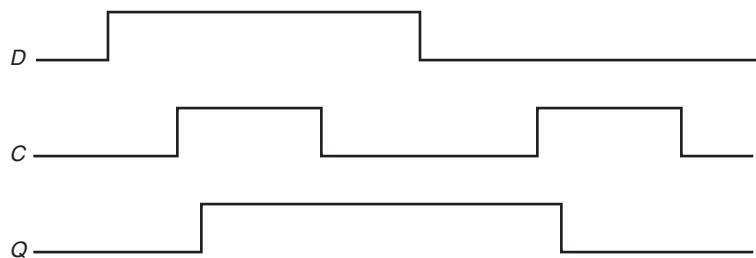
and its complement ( $\bar{Q}$ ). When the clock input  $C$  is asserted, the latch is said to be *open*, and the value of the output ( $Q$ ) becomes the value of the input  $D$ . When the clock input  $C$  is deasserted, the latch is said to be *closed*, and the value of the output ( $Q$ ) is whatever value was stored the last time the latch was open.

[Figure B.8.2](#) shows how a D latch can be implemented with two additional gates added to the cross-coupled NOR gates. Since when the latch is open the value of  $Q$  changes as  $D$  changes, this structure is sometimes called a *transparent latch*. [Figure B.8.3](#) shows how this D latch works, assuming that the output  $Q$  is initially false and that  $D$  changes first.

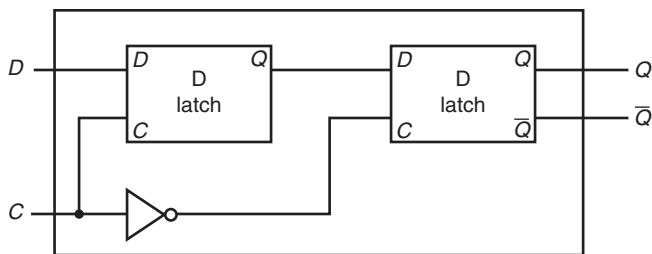
As mentioned earlier, we use flip-flops as the basic building block, rather than latches. Flip-flops are not transparent: their outputs change *only* on the clock edge. A flip-flop can be built so that it triggers on either the rising (positive) or falling (negative) clock edge; for our designs we can use either type. [Figure B.8.4](#) shows how a falling-edge D flip-flop is constructed from a pair of D latches. In a D flip-flop, the output is stored when the clock edge occurs. [Figure B.8.5](#) shows how this flip-flop operates.



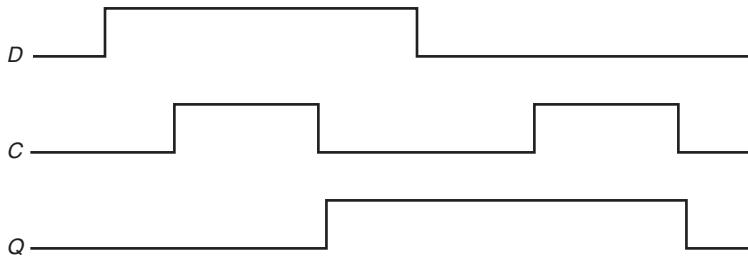
**FIGURE B.8.2 A D latch implemented with NOR gates.** A NOR gate acts as an inverter if the other input is 0. Thus, the cross-coupled pair of NOR gates acts to store the state value unless the clock input,  $C$ , is asserted, in which case the value of input  $D$  replaces the value of  $Q$  and is stored. The value of input  $D$  must be stable when the clock signal  $C$  changes from asserted to deasserted.



**FIGURE B.8.3 Operation of a D latch, assuming the output is initially deasserted.** When the clock,  $C$ , is asserted, the latch is open and the  $Q$  output immediately assumes the value of the  $D$  input.



**FIGURE B.8.4 A D flip-flop with a falling-edge trigger.** The first latch, called the master, is open and follows the input  $D$  when the clock input,  $C$ , is asserted. When the clock input,  $C$ , falls, the first latch is closed, but the second latch, called the slave, is open and gets its input from the output of the master latch.



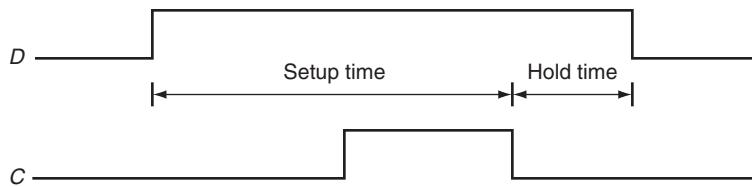
**FIGURE B.8.5 Operation of a D flip-flop with a falling-edge trigger, assuming the output is initially deasserted.** When the clock input ( $C$ ) changes from asserted to deasserted, the  $Q$  output stores the value of the  $D$  input. Compare this behavior to that of the clocked D latch shown in Figure B.8.3. In a clocked latch, the stored value and the output,  $Q$ , both change whenever  $C$  is high, as opposed to only when  $C$  transitions.

Here is a Verilog description of a module for a rising-edge D flip-flop, assuming that  $C$  is the clock input and  $D$  is the data input:

```
module DFF(clock,D,Q,Qbar);
    input clock, D;
    output reg Q; // Q is a reg since it is assigned in an
    always block
    output Qbar;
    assign Qbar = ~ Q; // Qbar is always just the inverse
    of Q
    always @(posedge clock) // perform actions whenever the
    clock rises
        Q = D;
endmodule
```

Because the  $D$  input is sampled on the clock edge, it must be valid for a period of time immediately before and immediately after the clock edge. The minimum time that the input must be valid before the clock edge is called the **setup time**; the

**setup time** The minimum time that the input to a memory device must be valid before the clock edge.



**FIGURE B.8.6 Setup and hold time requirements for a D flip-flop with a falling-edge trigger.**

The input must be stable for a period of time before the clock edge, as well as after the clock edge. The minimum time the signal must be stable before the clock edge is called the setup time, while the minimum time the signal must be stable after the clock edge is called the hold time. Failure to meet these minimum requirements can result in a situation where the output of the flip-flop may not be predictable, as described in Section B.11. Hold times are usually either 0 or very small and thus not a cause of worry.

**hold time** The minimum time during which the input must be valid after the clock edge.

minimum time during which it must be valid after the clock edge is called the **hold time**. Thus the inputs to any flip-flop (or anything built using flip-flops) must be valid during a window that begins at time  $t_{\text{setup}}$  before the clock edge and ends at  $t_{\text{hold}}$  after the clock edge, as shown in Figure B.8.6. Section B.11 talks about clocking and timing constraints, including the propagation delay through a flip-flop, in more detail.

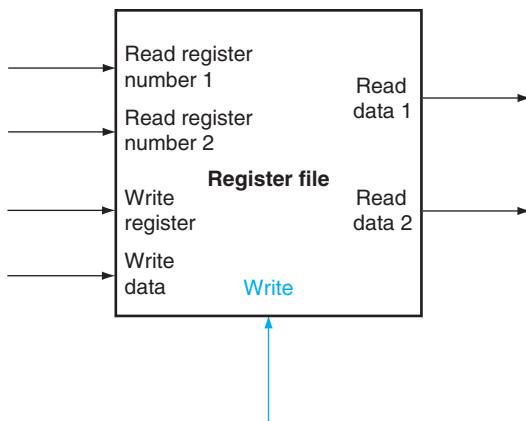
We can use an array of D flip-flops to build a register that can hold a multibit datum, such as a byte or word. We used registers throughout our datapaths in Chapter 4.

## Register Files

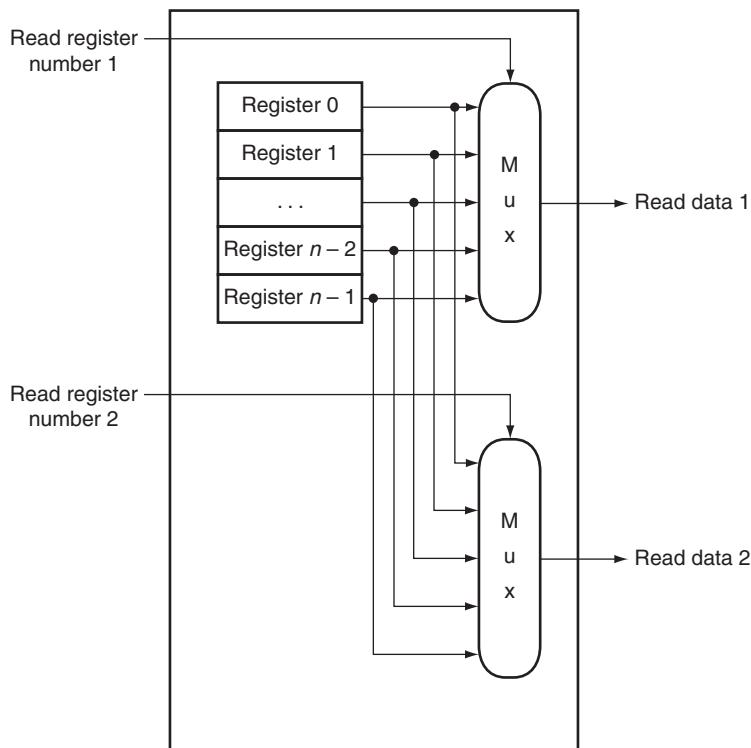
One structure that is central to our datapath is a *register file*. A register file consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file can be implemented with a decoder for each read or write port and an array of registers built from D flip-flops. Because reading a register does not change any state, we need only supply a register number as an input, and the only output will be the data contained in that register. For writing a register we will need three inputs: a register number, the data to write, and a clock that controls the writing into the register. In Chapter 4, we used a register file that has two read ports and one write port. This register file is drawn as shown in Figure B.8.7. The read ports can be implemented with a pair of multiplexors, each of which is as wide as the number of bits in each register of the register file. Figure B.8.8 shows the implementation of two register read ports for a 32-bit-wide register file.

Implementing the write port is slightly more complex, since we can only change the contents of the designated register. We can do this by using a decoder to generate a signal that can be used to determine which register to write. Figure B.8.9 shows how to implement the write port for a register file. It is important to remember that the flip-flop changes state only on the clock edge. In Chapter 4, we hooked up write signals for the register file explicitly and assumed the clock shown in Figure B.8.9 is attached implicitly.

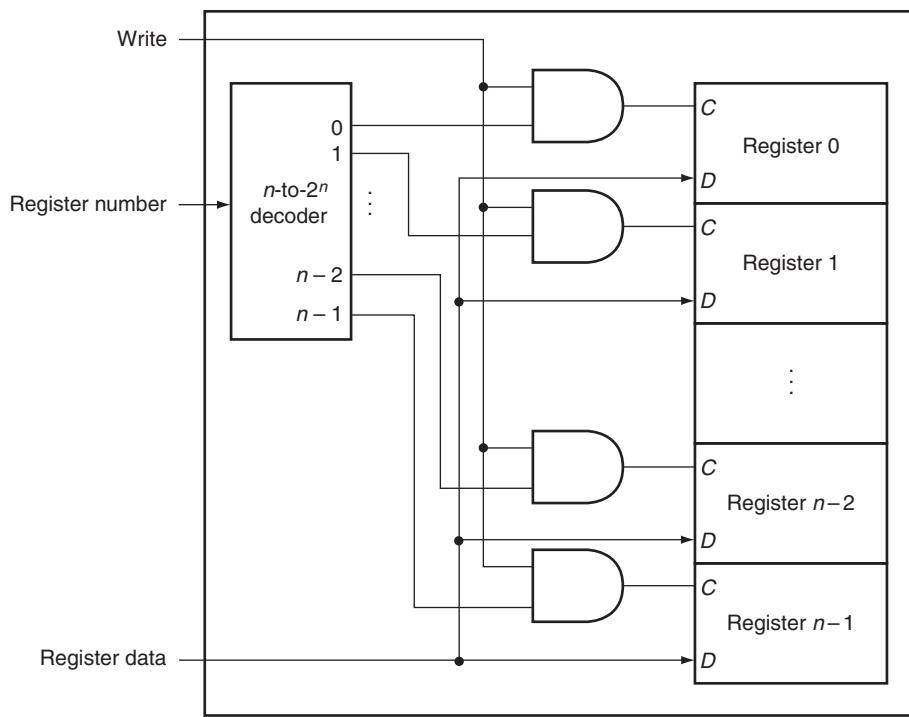
What happens if the same register is read and written during a clock cycle? Because the write of the register file occurs on the clock edge, the register will be



**FIGURE B.8.7** A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.



**FIGURE B.8.8** The implementation of two read ports for a register file with  $n$  registers can be done with a pair of  $n$ -to-1 multiplexors, each 32 bits wide. The register read number signal is used as the multiplexor selector signal. Figure B.8.9 shows how the write port is implemented.



**FIGURE B.8.9** The write port for a register file is implemented with a decoder that is used with the write signal to generate the C input to the registers. All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data is written into the register file.

valid during the time it is read, as we saw earlier in [Figure B.7.2](#). The value returned will be the value written in an earlier clock cycle. If we want a read to return the value currently being written, additional logic in the register file or outside of it is needed. Chapter 4 makes extensive use of such logic.

### Specifying Sequential Logic in Verilog

To specify sequential logic in Verilog, we must understand how to generate a clock, how to describe when a value is written into a register, and how to specify sequential control. Let us start by specifying a clock. A clock is not a predefined object in Verilog; instead, we generate a clock by using the Verilog notation `#n` before a statement; this causes a delay of `n` simulation time steps before the execution of the statement. In most Verilog simulators, it is also possible to generate a clock as an external input, allowing the user to specify at simulation time the number of clock cycles during which to run a simulation.

The code in [Figure B.8.10](#) implements a simple clock that is high or low for one simulation unit and then switches state. We use the delay capability and blocking assignment to implement the clock.

---

```
reg clock; // clock is a register
always
#1 clock = 1; #1 clock = 0;
```

**FIGURE B.8.10 A specification of a clock.**

Next, we must be able to specify the operation of an edge-triggered register. In Verilog, this is done by using the sensitivity list on an `always` block and specifying as a trigger either the positive or negative edge of a binary variable with the notation `posedge` or `negedge`, respectively. Hence, the following Verilog code causes register A to be written with the value b at the positive edge clock:

```
reg [31:0] A;
wire [31:0] b;

always @(posedge clock) A <= b;

module registerfile (Read1,Read2,WriteReg,WriteData,RegWrite,
Data1,Data2,clock);
    input [5:0] Read1,Read2,WriteReg; // the register numbers
    to read or write
    input [31:0] WriteData; // data to write
    input RegWrite, // the write control
    clock; // the clock to trigger write
    output [31:0] Data1, Data2; // the register values read
    reg [31:0] RF [31:0]; // 32 registers each 32 bits long

    assign Data1 = RF[Read1];
    assign Data2 = RF[Read2];

    always begin
        // write the register with new value if Regwrite is
        high
        @(posedge clock) if (RegWrite) RF[WriteReg] <=
        WriteData;
    end
endmodule
```

**FIGURE B.8.11 A MIPS register file written in behavioral Verilog.** This register file writes on the rising clock edge.

Throughout this chapter and the Verilog sections of Chapter 4, we will assume a positive edge-triggered design. Figure B.8.11 shows a Verilog specification of a MIPS register file that assumes two reads and one write, with only the write being clocked.

### Check Yourself

In the Verilog for the register file in [Figure B.8.11](#), the output ports corresponding to the registers being read are assigned using a continuous assignment, but the register being written is assigned in an `always` block. Which of the following is the reason?

- There is no special reason. It was simply convenient.
- Because Data1 and Data2 are output ports and WriteData is an input port.
- Because reading is a combinational event, while writing is a sequential event.

## B.9

## Memory Elements: SRAMs and DRAMs

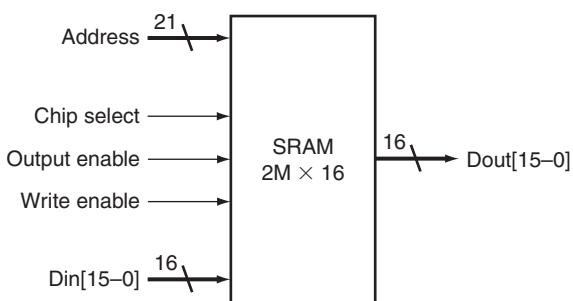
### static random access memory (SRAM)

A memory where data is stored statically (as in flip-flops) rather than dynamically (as in DRAM). SRAMs are faster than DRAMs, but less dense and more expensive per bit.

Registers and register files provide the basic building blocks for small memories, but larger amounts of memory are built using either **SRAMs (static random access memories)** or **DRAMs** (dynamic random access memories). We first discuss SRAMs, which are somewhat simpler, and then turn to DRAMs.

### SRAMs

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 characteristics often differ. An SRAM chip has a specific configuration in terms of the number of addressable locations, as well as the width of each addressable location. For example, a  $4M \times 8$  SRAM provides 4M entries, each of which is 8 bits wide. Thus it will have 22 address lines (since  $4M = 2^{22}$ ), an 8-bit data output line, and an 8-bit single data input line. As with ROMs, the number of addressable locations is often called the *height*, with the number of bits per unit called the *width*. For a variety of technical reasons, the newest and fastest SRAMs are typically available in narrow configurations:  $\times 1$  and  $\times 4$ . [Figure B.9.1](#) shows the input and output signals for a  $2M \times 16$  SRAM.



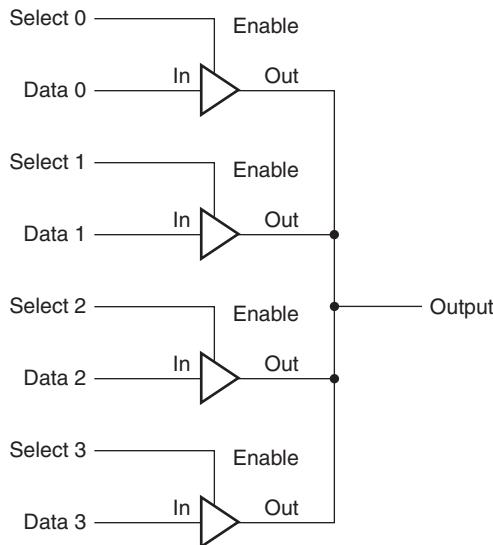
**FIGURE B.9.1 A  $32K \times 8$  SRAM showing the 21 address lines ( $32K = 2^{15}$ ) and 16 data inputs, the 3 control lines, and the 16 data outputs.**

To initiate a read or write access, the Chip select signal must be made active. For reads, we must also activate the Output enable signal that controls whether or not the datum selected by the address is actually driven on the pins. The Output enable is useful for connecting multiple memories to a single-output bus and using Output enable to determine which memory drives the bus. The SRAM read access time is usually specified as the delay from the time that Output enable is true and the address lines are valid until the time that the data is on the output lines. Typical read access times for SRAMs in 2004 varied from about 2–4 ns for the fastest CMOS parts, which tend to be somewhat smaller and narrower, to 8–20 ns for the typical largest parts, which in 2004 had more than 32 million bits of data. The demand for low-power SRAMs for consumer products and digital appliances has grown greatly in the past five years; these SRAMs have much lower stand-by and access power, but usually are 5–10 times slower. Most recently, synchronous SRAMs—similar to the synchronous DRAMs, which we discuss in the next section—have also been developed.

For writes, we must supply the data to be written and the address, as well as signals to cause the write to occur. When both the Write enable and Chip select are true, the data on the data input lines is written into the cell specified by the address. There are setup-time and hold-time requirements for the address and data lines, just as there were for D flip-flops and latches. In addition, the Write enable signal is not a clock edge but a pulse with a minimum width requirement. The time to complete a write is specified by the combination of the setup times, the hold times, and the Write enable pulse width.

Large SRAMs cannot be built in the same way we build a register file because, unlike a register file where a 32-to-1 multiplexor might be practical, the 64K-to-1 multiplexor that would be needed for a  $64K \times 1$  SRAM is totally impractical. Rather than use a giant multiplexor, large memories are implemented with a shared output line, called a *bit line*, which multiple memory cells in the memory array can assert. To allow multiple sources to drive a single line, a *three-state buffer* (or *tristate buffer*) is used. A three-state buffer has two inputs—a data signal and an Output enable—and a single output, which is in one of three states: asserted, deasserted, or high impedance. The output of a tristate buffer is equal to the data input signal, either asserted or deasserted, if the Output enable is asserted, and is otherwise in a *high-impedance state* that allows another three-state buffer whose Output enable is asserted to determine the value of a shared output.

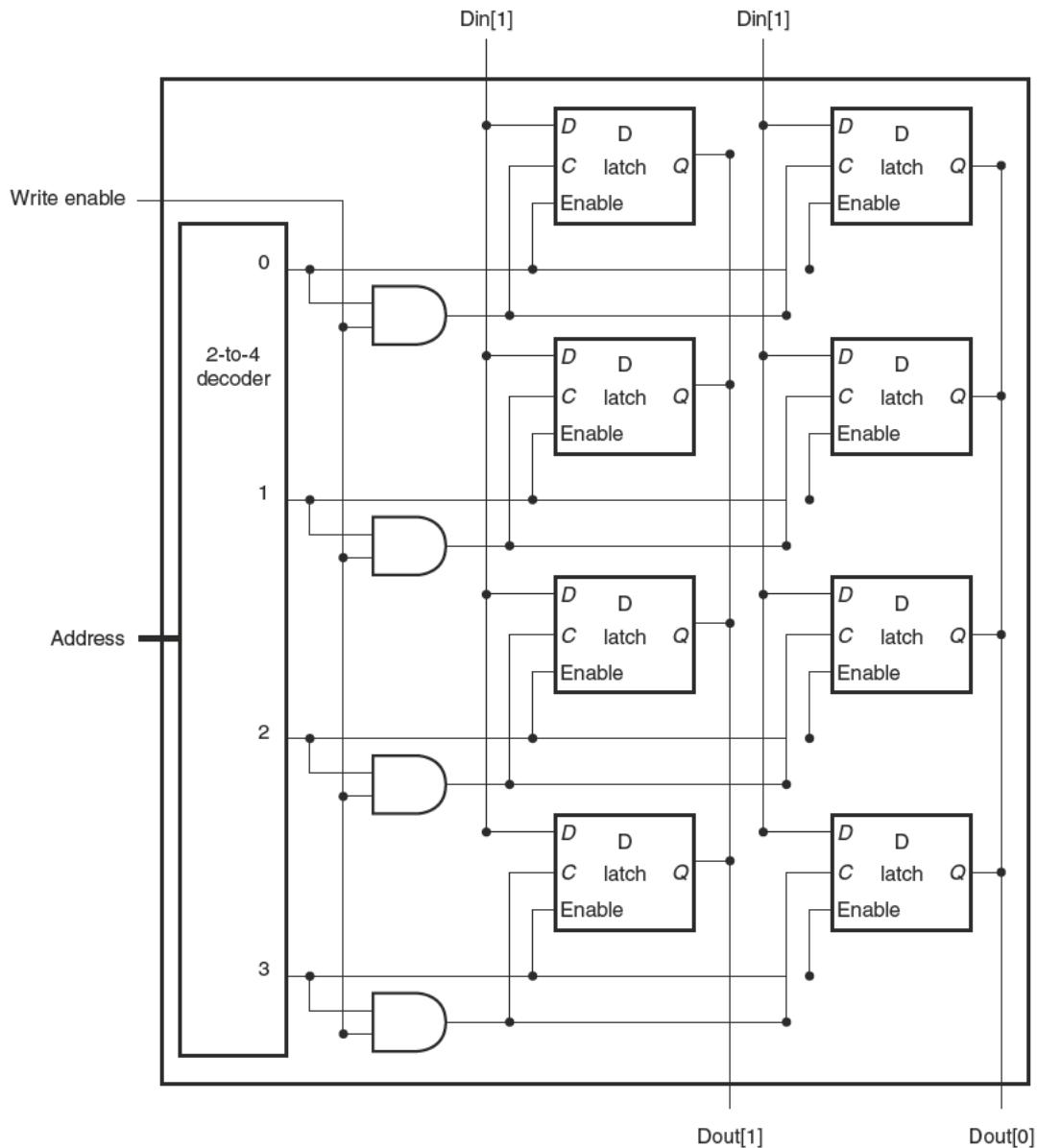
Figure B.9.2 shows a set of three-state buffers wired to form a multiplexor with a decoded input. It is critical that the Output enable of at most one of the three-state buffers be asserted; otherwise, the three-state buffers may try to set the output line differently. By using three-state buffers in the individual cells of the SRAM, each cell that corresponds to a particular output can share the same output line. The use of a set of distributed three-state buffers is a more efficient implementation than a large centralized multiplexor. The three-state buffers are incorporated into the flip-flops that form the basic cells of the SRAM. Figure B.9.3 shows how a small  $4 \times 2$  SRAM might be built, using D latches with an input called Enable that controls the three-state output.



**FIGURE B.9.2 Four three-state buffers are used to form a multiplexor.** Only one of the four Select inputs can be asserted. A three-state buffer with a deasserted Output enable has a high-impedance output that allows a three-state buffer whose Output enable is asserted to drive the shared output line.

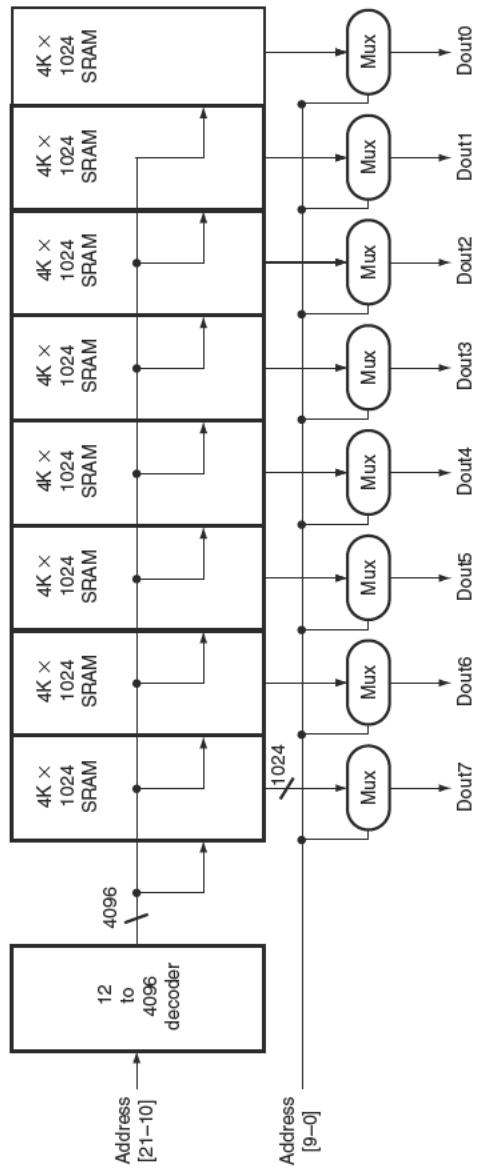
The design in [Figure B.9.3](#) eliminates the need for an enormous multiplexor; however, it still requires a very large decoder and a correspondingly large number of word lines. For example, in a  $4M \times 8$  SRAM, we would need a 22-to-4M decoder and 4M word lines (which are the lines used to enable the individual flip-flops)! To circumvent this problem, large memories are organized as rectangular arrays and use a two-step decoding process. [Figure B.9.4](#) shows how a  $4M \times 8$  SRAM might be organized internally using a two-step decode. As we will see, the two-level decoding process is quite important in understanding how DRAMs operate.

Recently we have seen the development of both synchronous SRAMs (SSRAMs) and synchronous DRAMs (SDRAMs). The key capability provided by synchronous RAMs is the ability to transfer a *burst* of data from a series of sequential addresses within an array or row. The burst is defined by a starting address, supplied in the usual fashion, and a burst length. The speed advantage of synchronous RAMs comes from the ability to transfer the bits in the burst without having to specify additional address bits. Instead, a clock is used to transfer the successive bits in the burst. The elimination of the need to specify the address for the transfers within the burst significantly improves the rate for transferring the block of data. Because of this capability, synchronous SRAMs and DRAMs are rapidly becoming the RAMs of choice for building memory systems in computers. We discuss the use of synchronous DRAMs in a memory system in more detail in the next section and in Chapter 5.



**FIGURE B.9.3** The basic structure of a  $4 \times 2$  SRAM consists of a decoder that selects which pair of cells to activate.

The activated cells use a three-state output connected to the vertical bit lines that supply the requested data. The address that selects the cell is sent on one of a set of horizontal address lines, called word lines. For simplicity, the Output enable and Chip select signals have been omitted, but they could easily be added with a few AND gates.



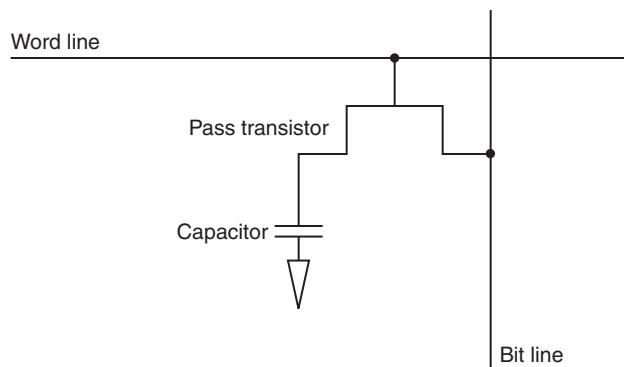
**FIGURE B-9.4 Typical organization of a  $4M \times 8$  SRAM as an array of  $4K \times 1024$  arrays.** The first decoder generates the addresses for eight  $4K \times 1024$  arrays; then a set of multiplexors is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decode that would need either an enormous decoder or a gigantic multiplexor. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.

## DRAMs

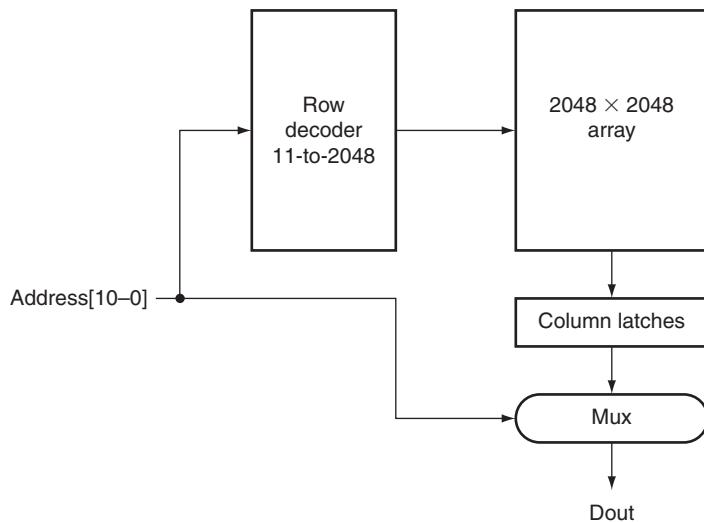
In a static RAM (SRAM), the value stored in a cell is kept on a pair of inverting gates, and as long as power is applied, the value can be kept indefinitely. In a dynamic RAM (DRAM), the value kept in a cell is stored as a charge in a capacitor. A single transistor is then used to access this stored charge, either to read the value or to overwrite the charge stored there. Because DRAMs use only a single transistor per bit of storage, they are much denser and cheaper per bit. By comparison, SRAMs require four to six transistors per bit. Because 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*, as opposed to the static storage in a SRAM cell.

To refresh the cell, we merely read its contents and write it back. The charge can be kept for several milliseconds, which might correspond to close to a million clock cycles. Today, single-chip memory controllers often handle the refresh function independently of the processor. If every bit had to be read out of the DRAM and then written back individually, with large DRAMs containing multiple megabytes, we would constantly be refreshing the DRAM, leaving no time for accessing it. Fortunately, DRAMs also 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. Typically, refresh operations consume 1% to 2% of the active cycles of the DRAM, leaving the remaining 98% to 99% of the cycles available for reading and writing data.

**Elaboration:** How does a DRAM read and write the signal stored in a cell? The transistor inside the cell is a switch, called a *pass transistor*, that allows the value stored on the capacitor to be accessed for either reading or writing. [Figure B.9.5](#) shows how the single-transistor cell looks. The pass transistor acts like a switch: when the signal on the word line is asserted, the switch is closed, connecting the capacitor to the bit line. If the operation is a write, then the value to be written is placed on the bit line. If the value is a 1, the capacitor will be charged. If the value is a 0, then the capacitor will be discharged. Reading is slightly more complex, since the DRAM must detect a very small charge stored in the capacitor. Before activating the word line for a read, the bit line is charged to the voltage that is halfway between the low and high voltage. Then, by activating the word line, the charge on the capacitor is read out onto the bit line. This causes the bit line to move slightly toward the high or low direction, and this change is detected with a sense amplifier, which can detect small changes in voltage.



**FIGURE B.9.5** A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.



**FIGURE B.9.6** A 4M × 1 DRAM is built with a 2048 × 2048 array. The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.

DRAMs use a two-level decoder consisting of a *row access* followed by a *column access*, as shown in [Figure B.9.6](#). The row access chooses one of a number of rows and activates the corresponding word line. The contents of all the columns in the active row are then stored in a set of latches. The column access then selects the data from the column latches. To save pins and reduce the package cost, the same address lines are used for both the row and column address; a pair of signals called RAS (*Row Access Strobe*) and CAS (*Column Access Strobe*) are used to signal the DRAM that either a row or column address is being supplied. Refresh is performed by simply reading the columns into the column latches and then writing the same values back. Thus, an entire row is refreshed in one cycle. The two-level addressing scheme, combined with the internal circuitry, makes DRAM access times much longer (by a factor of 5–10) than SRAM access times. In 2004, typical DRAM access times ranged from 45 to 65 ns; 256 Mbit DRAMs are in full production, and the first customer samples of 1 GB DRAMs became available in the first quarter of 2004. The much lower cost per bit makes DRAM the choice for main memory, while the faster access time makes SRAM the choice for caches.

You might observe that a  $64M \times 4$  DRAM actually accesses 8K bits on every row access and then throws away all but 4 of those during a column access. DRAM designers have used the internal structure of the DRAM as a way to provide higher bandwidth out of a DRAM. This is done by allowing the column address to change without changing the row address, resulting in an access to other bits in the column latches. To make this process faster and more precise, the address inputs were clocked, leading to the dominant form of DRAM in use today: synchronous DRAM or SDRAM.

Since about 1999, SDRAMs have been the memory chip of choice for most cache-based main memory systems. SDRAMs provide fast access to a series of bits within a row by sequentially transferring all the bits in a burst under the control of a clock signal. In 2004, DDRRAMs (Double Data Rate RAMs), which are called double data rate because they transfer data on both the rising and falling edge of an externally supplied clock, were the most heavily used form of SDRAMs. As we discuss in Chapter 5, these high-speed transfers can be used to boost the bandwidth available out of main memory to match the needs of the processor and caches.

## Error Correction

Because of the potential for data corruption in large memories, most computer systems use some sort of error-checking code to detect possible corruption of data. One simple code that is heavily used is a *parity code*. In a parity code the number of 1s in a word is counted; the word has odd parity if the number of 1s is odd and

even otherwise. When a word is written into memory, the parity bit is also written (1 for odd, 0 for 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.

A 1-bit parity scheme can detect at most 1 bit of error in a data item; 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 tell which bit in a data item is in error.

#### **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.

A 1-bit parity scheme is an **error detection code**; there are also *error correction codes* (ECC) that will detect and allow correction of an error. For large main memories, many systems use a code that allows the detection of up to 2 bits of error and the correction of a single bit of error. These codes work by using more bits to encode the data; for example, the typical codes used for main memories require 7 or 8 bits for every 128 bits of data.

**Elaboration:** A 1-bit parity code is a *distance-2 code*, which means that if we look at the data plus the parity bit, no 1-bit change is sufficient to generate another legal combination of the data plus parity. For example, if we change a bit in the data, the parity will be wrong, and vice versa. Of course, if we change 2 bits (any 2 data bits or 1 data bit and the parity bit), the parity will match the data and the error cannot be detected. Hence, there is a distance of two between legal combinations of parity and data.

To detect more than one error or correct an error, we need a *distance-3 code*, which has the property that any legal combination of the bits in the error correction code and the data has at least 3 bits differing from any other combination. Suppose we have such a code and we have one error in the data. In that case, the code plus data will be one bit away from a legal combination, and we can correct the data to that legal combination. If we have two errors, we can recognize that there is an error, but we cannot correct the errors. Let's look at an example. Here are the data words and a distance-3 error correction code for a 4-bit data item.

Data Word	Code bits	Data	Code bits
0000	000	1000	111
0001	011	1001	100
0010	101	1010	010
0011	110	1011	001
0100	110	1100	001
0101	101	1101	010
0110	011	1110	100
0111	000	1111	111

To see how this works, let's choose a data word, say 0110, whose error correction code is 011. Here are the four 1-bit error possibilities for this data: 1110, 0010, 0100, and 0111. Now look at the data item with the same code (011), which is the entry with the value 0001. If the error correction decoder received one of the four possible data words with an error, it would have to choose between correcting to 0110 or 0001. While these four words with error have only one bit changed from the correct pattern of 0110, they each have two bits that are different from the alternate correction of 0001. Hence, the error correction mechanism can easily choose to correct to 0110, since a single error is a much higher probability. To see that two errors can be detected, simply notice that all the combinations with two bits changed have a different code. The one reuse of the same code is with three bits different, but if we correct a 2-bit error, we will correct to the wrong value, since the decoder will assume that only a single error has occurred. If we want to correct 1-bit errors and detect, but not erroneously correct, 2-bit errors, we need a distance-4 code.

Although we distinguished between the code and data in our explanation, in truth, an error correction code treats the combination of code and data as a single word in a larger code (7 bits in this example). Thus, it deals with errors in the code bits in the same fashion as errors in the data bits.

While the above example requires  $n - 1$  bits for  $n$  bits of data, the number of bits required grows slowly, so that for a distance-3 code, a 64-bit word needs 7 bits and a 128-bit word needs 8. This type of code is called a *Hamming code*, after R. Hamming, who described a method for creating such codes.

## B.10 Finite-State Machines

As we saw earlier, digital logic systems can be classified as combinational or sequential. Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system. Thus, a sequential system cannot be described with a truth table. Instead, a sequential system is described as a **finite-state machine** (or often just *state machine*). A finite-state machine has a set of states and two functions, called the **next-state function** and the **output function**. The set of states corresponds to all the possible values of the internal storage. Thus, if there are  $n$  bits of storage, there are  $2^n$  states. The next-state function is a combinational function that, given the inputs and the current state, determines the next state of the system. The output function produces a set of outputs from the current state and the inputs. Figure B.10.1 shows this diagrammatically.

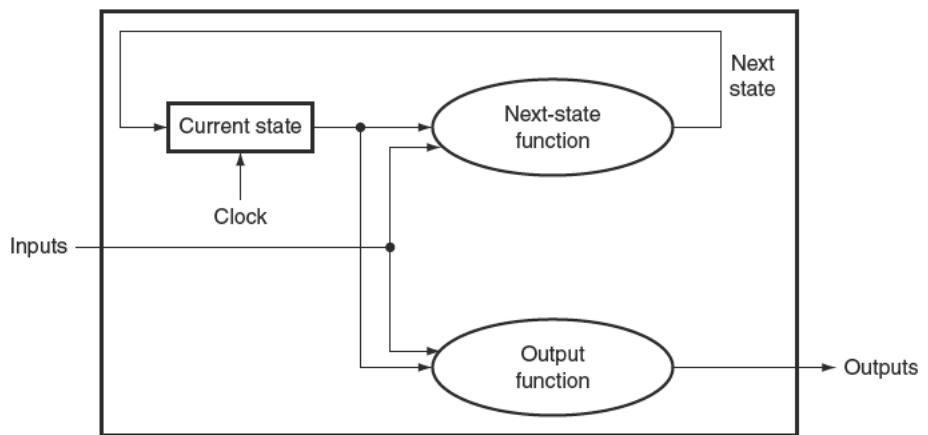
The state machines we discuss here and in Chapter 4 are *synchronous*. This means that the state changes together with the clock cycle, and a new state is computed once every clock. Thus, the state elements are updated only on the clock edge. We use this methodology in this section and throughout Chapter 4, and we do not

### 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.



**FIGURE B.10.1 A state machine consists of internal storage that contains the state and two combinational functions: the next-state function and the output function.** Often, the output function is restricted to take only the current state as its input; this does not change the capability of a sequential machine, but does affect its internals.

usually show the clock explicitly. We use state machines throughout Chapter 4 to control the execution of the processor and the actions of the datapath.

To illustrate how a finite-state machine operates and is designed, let's look at a simple and classic example: controlling a traffic light. (Chapters 4 and 5 contain more detailed examples of using finite-state machines to control processor execution.) When a finite-state machine is used as a controller, the output function is often restricted to depend on just the current state. Such a finite-state machine is called a *Moore machine*. This is the type of finite-state machine we use throughout this book. If the output function can depend on both the current state and the current input, the machine is called a *Mealy machine*. These two machines are equivalent in their capabilities, and one can be turned into the other mechanically. The basic advantage of a Moore machine is that it can be faster, while a Mealy machine may be smaller, since it may need fewer states than a Moore machine. In Chapter 5, we discuss the differences in more detail and show a Verilog version of finite-state control using a Mealy machine.

Our example concerns the control of a traffic light at an intersection of a north-south route and an east-west route. For simplicity, we will consider only the green and red lights; adding the yellow light is left for an exercise. We want the lights to cycle no faster than 30 seconds in each direction, so we will use a 0.033 Hz clock so that the machine cycles between states at no faster than once every 30 seconds. There are two output signals:

- *NSlite*: When this signal is asserted, the light on the north-south road is green; when this signal is deasserted, the light on the north-south road is red.
- *EWlite*: When this signal is asserted, the light on the east-west road is green; when this signal is deasserted, the light on the east-west road is red.

In addition, there are two inputs:

- *NScar*: Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south).
- *EWcar*: Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west).

The traffic light should change from one direction to the other only if a car is waiting to go in the other direction; otherwise, the light should continue to show green in the same direction as the last car that crossed the intersection.

To implement this simple traffic light we need two states:

- *NSgreen*: The traffic light is green in the north-south direction.
- *EWgreen*: The traffic light is green in the east-west direction.

We also need to create the next-state function, which can be specified with a table:

	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

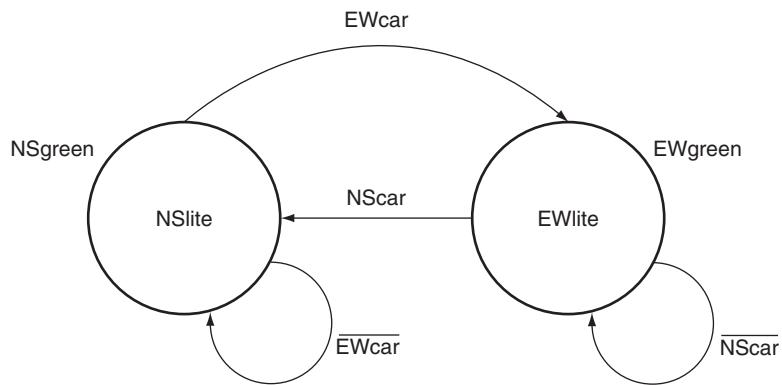
Notice that we didn't specify in the algorithm what happens when a car approaches from both directions. In this case, the next-state function given above changes the state to ensure that a steady stream of cars from one direction cannot lock out a car in the other direction.

The finite-state machine is completed by specifying the output function.

Before we examine how to implement this finite-state machine, let's look at a graphical representation, which is often used for finite-state machines. In this representation, nodes are used to indicate states. Inside the node we place a list of the outputs that are active for that state. Directed arcs are used to show the next-state

	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

function, with labels on the arcs specifying the input condition as logic functions. Figure B.10.2 shows the graphical representation for this finite-state machine.



**FIGURE B.10.2 The graphical representation of the two-state traffic light controller.** We simplified the logic functions on the state transitions. For example, the transition from NSgreen to EWgreen in the next-state table is  $(NScar \cdot EWcar) + (NScar \cdot \overline{EWcar})$ , which is equivalent to  $EWcar$ .

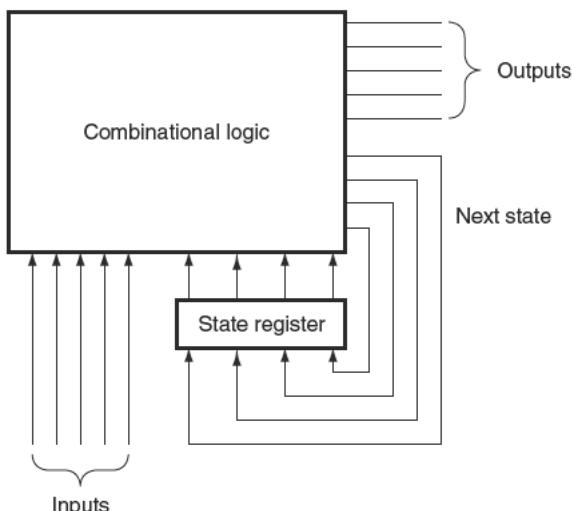
A finite-state machine can be implemented with a register to hold the current state and a block of combinational logic that computes the next-state function and the output function. Figure B.10.3 shows how a finite-state machine with 4 bits of state, and thus up to 16 states, might look. To implement the finite-state machine in this way, we must first assign state numbers to the states. This process is called *state assignment*. For example, we could assign NSgreen to state 0 and EWgreen to state 1. The state register would contain a single bit. The next-state function would be given as

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

where CurrentState is the contents of the state register (0 or 1) and NextState is the output of the next-state function that will be written into the state register at the end of the clock cycle. The output function is also simple:

$$\begin{aligned} \text{NSlite} &= \overline{\text{CurrentState}} \\ \text{EWlite} &= \text{CurrentState} \end{aligned}$$

The combinational logic block is often implemented using structured logic, such as a PLA. A PLA can be constructed automatically from the next-state and output function tables. In fact, there are *computer-aided design* (CAD) programs



**FIGURE B.10.3** A finite-state machine is implemented with a state register that holds the current state and a combinational logic block to compute the next state and output functions. The latter two functions are often split apart and implemented with two separate blocks of logic, which may require fewer gates.

that take either a graphical or textual representation of a finite-state machine and produce an optimized implementation automatically. In Chapters 4 and 5, finite-state machines were used to control processor execution. [Appendix D](#) discusses the detailed implementation of these controllers with both PLAs and ROMs.

To show how we might write the control in Verilog, [Figure B.10.4](#) shows a Verilog version designed for synthesis. Note that for this simple control function, a Mealy machine is not useful, but this style of specification is used in Chapter 5 to implement a control function that is a Mealy machine and has fewer states than the Moore machine controller.

```
module TrafficLite (EWCar,NSCar,EWLite,NSLite,clock);
    input EWCar, NSCar,clock;
    output EWLite,NSLite;
    reg state;
    initial state=0; //set initial state
    //following two assignments set the output, which is based
    only on the state variable
    assign NSLite = ~ state; //NSLite on if state = 0;
    assign EWLite = state; //EWLite on if state = 1
    always @(posedge clock) // all state updates on a positive
    clock edge
        case (state)
            0: state = EWCar; //change state only if EWCar
            1: state = NSCar; //change state only if NSCar
        endcase
    endmodule
```

---

**FIGURE B.10.4 A Verilog version of the traffic light controller.**

### Check Yourself

What is the smallest number of states in a Moore machine for which a Mealy machine could have fewer states?

- Two, since there could be a one-state Mealy machine that might do the same thing.
- Three, since there could be a simple Moore machine that went to one of two different states and always returned to the original state after that. For such a simple machine, a two-state Mealy machine is possible.
- You need at least four states to exploit the advantages of a Mealy machine over a Moore machine.

## B.11

### Timing Methodologies

Throughout this appendix and in the rest of the text, we use an edge-triggered timing methodology. This timing methodology has an advantage in that it is simpler to explain and understand than a level-triggered methodology. In this section, we explain this timing methodology in a little more detail and also introduce level-sensitive clocking. We conclude this section by briefly discussing

the issue of asynchronous signals and synchronizers, an important problem for digital designers.

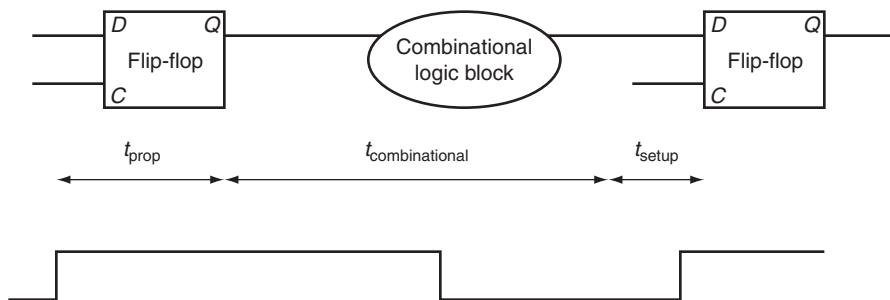
The purpose of this section is to introduce the major concepts in clocking methodology. The section makes some important simplifying assumptions; if you are interested in understanding timing methodology in more detail, consult one of the references listed at the end of this appendix.

We use an edge-triggered timing methodology because it is simpler to explain and has fewer rules required for correctness. In particular, if we assume that all clocks arrive at the same time, we are guaranteed that a system with edge-triggered registers between blocks of combinational logic can operate correctly without races if we simply make the clock long enough. A *race* occurs when the contents of a state element depend on the relative speed of different logic elements. In an edge-triggered design, the clock cycle must be long enough to accommodate the path from one flip-flop through the combinational logic to another flip-flop where it must satisfy the setup-time requirement. [Figure B.11.1](#) shows this requirement for a system using rising edge-triggered flip-flops. In such a system the clock period (or cycle time) must be at least as large as

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}}$$

for the worst-case values of these three delays, which are defined as follows:

- $t_{\text{prop}}$  is the time for a signal to propagate through a flip-flop; it is also sometimes called clock-to-Q.
- $t_{\text{combinational}}$  is the longest delay for any combinational logic (which by definition is surrounded by two flip-flops).
- $t_{\text{setup}}$  is the time before the rising clock edge that the input to a flip-flop must be valid.



**FIGURE B.11.1** In an edge-triggered design, the clock must be long enough to allow signals to be valid for the required setup time before the next clock edge. The time for a flip-flop input to propagate to the flip-flop outputs is  $t_{\text{prop}}$ ; the signal then takes  $t_{\text{combinational}}$  to travel through the combinational logic and must be valid  $t_{\text{setup}}$  before the next clock edge.

**clock skew** The difference in absolute time between the times when two state elements see a clock edge.

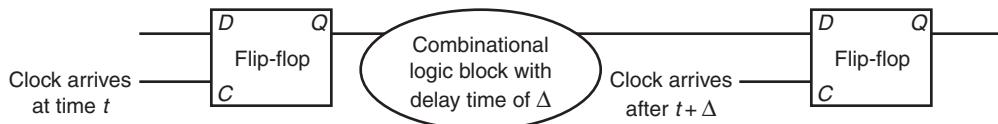
We make one simplifying assumption: the hold-time requirements are satisfied, which is almost never an issue with modern logic.

One additional complication that must be considered in edge-triggered designs is **clock skew**. Clock skew is the difference in absolute time between when two state elements see a clock edge. Clock skew arises because the clock signal will often use two different paths, with slightly different delays, to reach two different state elements. If the clock skew is large enough, it may be possible for a state element to change and cause the input to another flip-flop to change before the clock edge is seen by the second flip-flop.

Figure B.11.2 illustrates this problem, ignoring setup time and flip-flop propagation delay. To avoid incorrect operation, the clock period is increased to allow for the maximum clock skew. Thus, the clock period must be longer than

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}} + t_{\text{skew}}$$

With this constraint on the clock period, the two clocks can also arrive in the opposite order, with the second clock arriving  $t_{\text{skew}}$  earlier, and the circuit will work



**FIGURE B.11.2 Illustration of how clock skew can cause a race, leading to incorrect operation.** Because of the difference in when the two flip-flops see the clock, the signal that is stored into the first flip-flop can race forward and change the input to the second flip-flop before the clock arrives at the second flip-flop.

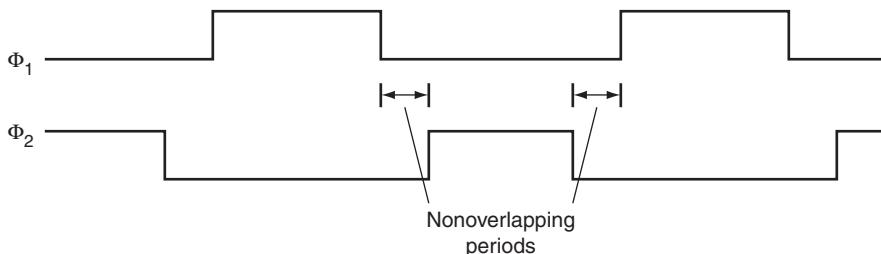
**level-sensitive clocking** A timing methodology in which state changes occur at either high or low clock levels but are not instantaneous as such changes are in edge-triggered designs.

correctly. Designers reduce clock-skew problems by carefully routing the clock signal to minimize the difference in arrival times. In addition, smart designers also provide some margin by making the clock a little longer than the minimum; this allows for variation in components as well as in the power supply. Since clock skew can also affect the hold-time requirements, minimizing the size of the clock skew is important.

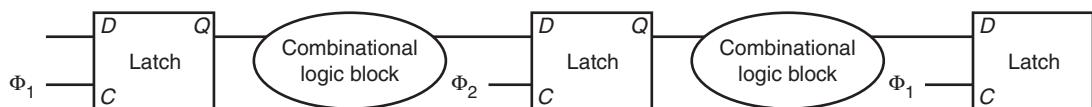
Edge-triggered designs have two drawbacks: they require extra logic and they may sometimes be slower. Just looking at the D flip-flop versus the level-sensitive latch that we used to construct the flip-flop shows that edge-triggered design requires more logic. An alternative is to use **level-sensitive clocking**. Because state changes in a level-sensitive methodology are not instantaneous, a level-sensitive scheme is slightly more complex and requires additional care to make it operate correctly.

## Level-Sensitive Timing

In level-sensitive timing, the state changes occur at either high or low levels, but they are not instantaneous as they are in an edge-triggered methodology. Because of the noninstantaneous change in state, races can easily occur. To ensure that a level-sensitive design will also work correctly if the clock is slow enough, designers use *two-phase clocking*. Two-phase clocking is a scheme that makes use of two nonoverlapping clock signals. Since the two clocks, typically called  $\Phi_1$  and  $\Phi_2$ , are nonoverlapping, at most one of the clock signals is high at any given time, as Figure B.11.3 shows. We can use these two clocks to build a system that contains level-sensitive latches but is free from any race conditions, just as the edge-triggered designs were.



**FIGURE B.11.3** A two-phase clocking scheme showing the cycle of each clock and the nonoverlapping periods.



**FIGURE B.11.4** A two-phase timing scheme with alternating latches showing how the system operates on both clock phases. The output of a latch is stable on the opposite phase from its C input. Thus, the first block of combinational inputs has a stable input during  $\Phi_2$ , and its output is latched by  $\Phi_1$ . The second (rightmost) combinational block operates in just the opposite fashion, with stable inputs during  $\Phi_1$ . Thus, the delays through the combinational blocks determine the minimum time that the respective clocks must be asserted. The size of the nonoverlapping period is determined by the maximum clock skew and the minimum delay of any logic block.

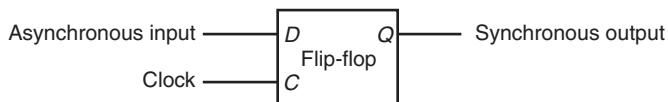
One simple way to design such a system is to alternate the use of latches that are open on  $\Phi_1$  with latches that are open on  $\Phi_2$ . Because both clocks are not asserted at the same time, a race cannot occur. If the input to a combinational block is a  $\Phi_1$  clock, then its output is latched by a  $\Phi_2$  clock, which is open only during  $\Phi_2$  when the input latch is closed and hence has a valid output. Figure B.11.4 shows how a system with two-phase timing and alternating latches operates. As in an edge-triggered design, we must pay attention to clock skew, particularly between the two

clock phases. By increasing the amount of nonoverlap between the two phases, we can reduce the potential margin of error. Thus, the system is guaranteed to operate correctly if each phase is long enough and if there is large enough nonoverlap between the phases.

## Asynchronous Inputs and Synchronizers

By using a single clock or a two-phase clock, we can eliminate race conditions if clock-skew problems are avoided. Unfortunately, it is impractical to make an entire system function with a single clock and still keep the clock skew small. While the CPU may use a single clock, I/O devices will probably have their own clock. An asynchronous device may communicate with the CPU through a series of handshaking steps. To translate the asynchronous input to a synchronous signal that can be used to change the state of a system, we need to use a *synchronizer*, whose inputs are the asynchronous signal and a clock and whose output is a signal synchronous with the input clock.

Our first attempt to build a synchronizer uses an edge-triggered D flip-flop, whose *D* input is the asynchronous signal, as Figure B.11.5 shows. Because we communicate with a handshaking protocol, it does not matter whether we detect the asserted state of the asynchronous signal on one clock or the next, since the signal will be held asserted until it is acknowledged. Thus, you might think that this simple structure is enough to sample the signal accurately, which would be the case except for one small problem.



**FIGURE B.11.5** A synchronizer built from a D flip-flop is used to sample an asynchronous signal to produce an output that is synchronous with the clock. This “synchronizer” will not work properly!

### metastability

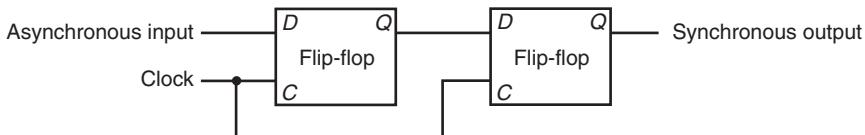
A situation that occurs if a signal is sampled when it is not stable for the required setup and hold times, possibly causing the sampled value to fall in the indeterminate region between a high and low value.

The problem is a situation called **metastability**. Suppose the asynchronous signal is transitioning between high and low when the clock edge arrives. Clearly, it is not possible to know whether the signal will be latched as high or low. That problem we could live with. Unfortunately, the situation is worse: when the signal that is sampled is not stable for the required setup and hold times, the flip-flop may go into a *metastable* state. In such a state, the output will not have a legitimate high or low value, but will be in the indeterminate region between them. Furthermore,

the flip-flop is not guaranteed to exit this state in any bounded amount of time. Some logic blocks that look at the output of the flip-flop may see its output as 0, while others may see it as 1. This situation is called a **synchronizer failure**.

In a purely synchronous system, synchronizer failure can be avoided by ensuring that the setup and hold times for a flip-flop or latch are always met, but this is impossible when the input is asynchronous. Instead, the only solution possible is to wait long enough before looking at the output of the flip-flop to ensure that its output is stable, and that it has exited the metastable state, if it ever entered it. How long is long enough? Well, the probability that the flip-flop will stay in the metastable state decreases exponentially, so after a very short time the probability that the flip-flop is in the metastable state is very low; however, the probability never reaches 0! So designers wait long enough such that the probability of a synchronizer failure is very low, and the time between such failures will be years or even thousands of years.

For most flip-flop designs, waiting for a period that is several times longer than the setup time makes the probability of synchronization failure very low. If the clock rate is longer than the potential metastability period (which is likely), then a safe synchronizer can be built with two D flip-flops, as Figure B.11.6 shows. If you are interested in reading more about these problems, look into the references.



**FIGURE B.11.6** This synchronizer will work correctly if the period of metastability that we wish to guard against is less than the clock period. Although the output of the first flip-flop may be metastable, it will not be seen by any other logic element until the second clock, when the second D flip-flop samples the signal, which by that time should no longer be in a metastable state.

Suppose we have a design with very large clock skew—longer than the register **propagation time**. Is it always possible for such a design to slow the clock down enough to guarantee that the logic operates properly?

- Yes, if the clock is slow enough the signals can always propagate and the design will work, even if the skew is very large.
- No, since it is possible that two registers see the same clock edge far enough apart that a register is triggered, and its outputs propagated and seen by a second register with the same clock edge.

### synchronizer failure

A situation in which a flip-flop enters a metastable state and where some logic blocks reading the output of the flip-flop see a 0 while others see a 1.

### Check Yourself

**propagation time** The time required for an input to a flip-flop to propagate to the outputs of the flip-flop.

## B.12

## Field Programmable Devices

### field programmable devices (FPD)

An integrated circuit containing combinational logic, and possibly memory devices, that are configurable by the end user.

### programmable logic device (PLD)

An integrated circuit containing combinational logic whose function is configured by the end user.

### field programmable gate array (FPGA)

A configurable integrated circuit containing both combinational logic blocks and flip-flops.

### simple programmable logic device (SPLD)

Programmable logic device, usually containing either a single PAL or PLA.

### programmable array logic (PAL)

Contains a programmable and-plane followed by a fixed or-plane.

### antifuse

A structure in an integrated circuit that when programmed makes a permanent connection between two wires.

Within a custom or semicustom chip, designers can make use of the flexibility of the underlying structure to easily implement combinational or sequential logic. How can a designer who does not want to use a custom or semicustom IC implement a complex piece of logic taking advantage of the very high levels of integration available? The most popular component used for sequential and combinational logic design outside of a custom or semicustom IC is a **field programmable device (FPD)**. An FPD is an integrated circuit containing combinational logic, and possibly memory devices, that are configurable by the end user.

FPDs generally fall into two camps: **programmable logic devices (PLDs)**, which are purely combinational, and **field programmable gate arrays (FPGAs)**, which provide both combinational logic and flip-flops. PLDs consist of two forms: **simple PLDs (SPLDs)**, which are usually either a PLA or a **programmable array logic (PAL)**, and complex PLDs, which allow more than one logic block as well as configurable interconnections among blocks. When we speak of a PLA in a PLD, we mean a PLA with user programmable and-plane and or-plane. A PAL is like a PLA, except that the or-plane is fixed.

Before we discuss FPGAs, it is useful to talk about how FPDs are configured. Configuration is essentially a question of where to make or break connections. Gate and register structures are static, but the connections can be configured. Notice that by configuring the connections, a user determines what logic functions are implemented. Consider a configurable PLA: by determining where the connections are in the and-plane and the or-plane, the user dictates what logical functions are computed in the PLA. Connections in FPDs are either permanent or reconfigurable. Permanent connections involve the creation or destruction of a connection between two wires. Current FPLDs all use an **antifuse** technology, which allows a connection to be built at programming time that is then permanent. The other way to configure CMOS FPLDs is through a SRAM. The SRAM is downloaded at power-on, and the contents control the setting of switches, which in turn determines which metal lines are connected. The use of SRAM control has the advantage in that the FPD can be reconfigured by changing the contents of the SRAM. The disadvantages of the SRAM-based control are two fold: the configuration is volatile and must be reloaded on power-on, and the use of active transistors for switches slightly increases the resistance of such connections.

FPGAs include both logic and memory devices, usually structured in a two-dimensional array with the corridors dividing the rows and columns used for

global interconnect between the cells of the array. Each cell is a combination of gates and flip-flops that can be programmed to perform some specific function. Because they are basically small, programmable RAMs, they are also called **lookup tables (LUTs)**. Newer FPGAs contain more sophisticated building blocks such as pieces of adders and RAM blocks that can be used to build register files. A few large FPGAs even contain 32-bit RISC cores!

In addition to programming each cell to perform a specific function, the interconnections between cells are also programmable, allowing modern FPGAs with hundreds of blocks and hundreds of thousands of gates to be used for complex logic functions. Interconnect is a major challenge in custom chips, and this is even more true for FPGAs, because cells do not represent natural units of decomposition for structured design. In many FPGAs, 90% of the area is reserved for interconnect and only 10% is for logic and memory blocks.

Just as you cannot design a custom or semicustom chip without CAD tools, you also need them for FPDs. Logic synthesis tools have been developed that target FPGAs, allowing the generation of a system using FPGAs from structural and behavioral Verilog.

### lookup tables (LUTs)

In a field programmable device, the name given to the cells because they consist of a small amount of logic and RAM.

## B.13 Concluding Remarks

This appendix introduces the basics of logic design. If you have digested the material in this appendix, you are ready to tackle the material in Chapters 4 and 5, both of which use the concepts discussed in this appendix extensively.

## Further Reading

There are a number of good texts on logic design. Here are some you might like to look into.

Ciletti, M. D. [2002]. *Advanced Digital Design with the Verilog HDL*, Englewood Cliffs, NJ: Prentice Hall.

A thorough book on logic design using Verilog.

Katz, R. H. [2004]. *Modern Logic Design*, 2nd ed., Reading, MA: Addison-Wesley.  
A general text on logic design.

Wakerly, J. F. [2000]. *Digital Design: Principles and Practices*, 3rd ed., Englewood Cliffs, NJ: Prentice Hall.

A general text on logic design.

## B.14 Exercises

**B.1** [10] <§B.2> In addition to the basic laws we discussed in this section, there are two important theorems, called DeMorgan's theorems:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \text{ and } \overline{A \cdot B} = \overline{A} + \overline{B}$$

Prove DeMorgan's theorems with a truth table of the form

A	B	$\bar{A}$	$\bar{B}$	$\bar{A} + \bar{B}$	$\bar{A} \cdot \bar{B}$	$\bar{A} \cdot B$	$\bar{A} + \bar{B}$
0	0	1	1	1	1	1	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	0	0	0	0

**B.2** [15] <§B.2> Prove that the two equations for E in the example starting on page B-7 are equivalent by using DeMorgan's theorems and the axioms shown on page B-7.

**B.3** [10] <§B.2> Show that there are  $2n$  entries in a truth table for a function with  $n$  inputs.

**B.4** [10] <§B.2> One logic function that is used for a variety of purposes (including within adders and to compute parity) is *exclusive OR*. The output of a two-input exclusive OR function is true only if exactly one of the inputs is true. Show the truth table for a two-input exclusive OR function and implement this function using AND gates, OR gates, and inverters.

**B.5** [15] <§B.2> Prove that the NOR gate is universal by showing how to build the AND, OR, and NOT functions using a two-input NOR gate.

**B.6** [15] <§B.2> Prove that the NAND gate is universal by showing how to build the AND, OR, and NOT functions using a two-input NAND gate.

**B.7** [10] <§§B.2, B.3> Construct the truth table for a four-input odd-parity function (see page B-65 for a description of parity).

**B.8** [10] <§§B.2, B.3> Implement the four-input odd-parity function with AND and OR gates using bubbled inputs and outputs.

**B.9** [10] <§§B.2, B.3> Implement the four-input odd-parity function with a PLA.

**B.10** [15] <§§B.2, B.3> Prove that a two-input multiplexor is also universal by showing how to build the NAND (or NOR) gate using a multiplexor.

**B.11** [5] <§§4.2, B.2, B.3> Assume that X consists of 3 bits,  $x_2 \ x_1 \ x_0$ . Write four logic functions that are true if and only if

- X contains only one 0
- X contains an even number of 0s
- X when interpreted as an unsigned binary number is less than 4
- X when interpreted as a signed (two's complement) number is negative

**B.12** [5] <§§4.2, B.2, B.3> Implement the four functions described in Exercise B.11 using a PLA.

**B.13** [5] <§§4.2, B.2, B.3> Assume that X consists of 3 bits,  $x_2 \ x_1 \ x_0$ , and Y consists of 3 bits,  $y_2 \ y_1 \ y_0$ . Write logic functions that are true if and only if

- $X < Y$ , where X and Y are thought of as unsigned binary numbers
- $X < Y$ , where X and Y are thought of as signed (two's complement) numbers
- $X = Y$

Use a hierarchical approach that can be extended to larger numbers of bits. Show how can you extend it to 6-bit comparison.

**B.14** [5] <§§B.2, B.3> Implement a switching network that has two data inputs ( $A$  and  $B$ ), two data outputs ( $C$  and  $D$ ), and a control input ( $S$ ). If  $S$  equals 1, the network is in pass-through mode, and  $C$  should equal  $A$ , and  $D$  should equal  $B$ . If  $S$  equals 0, the network is in crossing mode, and  $C$  should equal  $B$ , and  $D$  should equal  $A$ .

**B.15** [15] <§§B.2, B.3> Derive the product-of-sums representation for  $E$  shown on page B-11 starting with the sum-of-products representation. You will need to use DeMorgan's theorems.

**B.16** [30] <§§B.2, B.3> Give an algorithm for constructing the sum-of-products representation for an arbitrary logic equation consisting of AND, OR, and NOT. The algorithm should be recursive and should not construct the truth table in the process.

**B.17** [5] <§§B.2, B.3> Show a truth table for a multiplexor (inputs  $A$ ,  $B$ , and  $S$ ; output  $C$ ), using don't cares to simplify the table where possible.

**B.18** [5] <§B.3> What is the function implemented by the following Verilog modules:

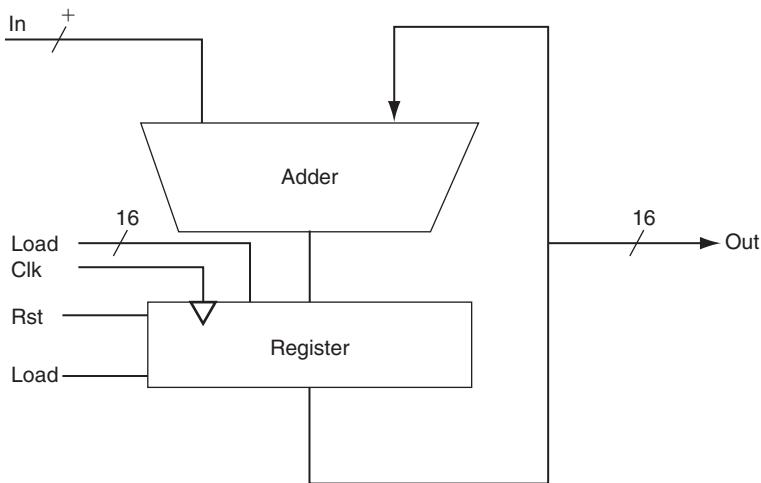
```
module FUNC1 (I0, I1, S, out);
    input I0, I1;
    input S;
    output out;
    out = S? I1: I0;
endmodule

module FUNC2 (out,ctl,clk,reset);
    output [7:0] out;
    input ctl, clk, reset;
    reg [7:0] out;
    always @(posedge clk)
        if (reset) begin
            out <= 8'b0 ;
        end
        else if (ctl) begin
            out <= out + 1;
        end
        else begin
            out <= out - 1;
        end
    endmodule
```

**B.19** [5] <§B.4> The Verilog code on page B-53 is for a D flip-flop. Show the Verilog code for a D latch.

**B.20** [10] <§§B.3, B.4> Write down a Verilog module implementation of a 2-to-4 decoder (and/or encoder).

**B.21** [10] <§§B.3, B.4> Given the following logic diagram for an accumulator, write down the Verilog module implementation of it. Assume a positive edge-triggered register and asynchronous Rst.



**B.22** [20] <§§B3, B.4, B.5> Section 3.3 presents basic operation and possible implementations of multipliers. A basic unit of such implementations is a shift-and-add unit. Show a Verilog implementation for this unit. Show how can you use this unit to build a 32-bit multiplier.

**B.23** [20] <§§B3, B.4, B.5> Repeat Exercise B.22, but for an unsigned divider rather than a multiplier.

**B.24** [15] <§B.5> The ALU supported set on less than (`slt`) using just the sign bit of the adder. Let's try a set on less than operation using the values  $-7_{\text{ten}}$  and  $6_{\text{ten}}$ . To make it simpler to follow the example, let's limit the binary representations to 4 bits:  $1001_{\text{two}}$  and  $0110_{\text{two}}$ .

$$1001_{\text{two}} - 0110_{\text{two}} = 1001_{\text{two}} + 1010_{\text{two}} = 0011_{\text{two}}$$

This result would suggest that  $-7 > 6$ , which is clearly wrong. Hence, we must factor in overflow in the decision. Modify the 1-bit ALU in Figure B.5.10 on page B-33 to handle `slt` correctly. Make your changes on a photocopy of this figure to save time.

**B.25** [20] <§B.6> A simple check for overflow during addition is to see if the CarryIn to the most significant bit is not the same as the CarryOut of the most significant bit. Prove that this check is the same as in Figure 3.2.

**B.26** [5] <§B.6> Rewrite the equations on page B-44 for a carry-lookahead logic for a 16-bit adder using a new notation. First, use the names for the CarryIn signals of the individual bits of the adder. That is, use  $c_4, c_8, c_{12}, \dots$  instead of  $C_1, C_2, C_3, \dots$ . In addition, let  $P_{i,j}$  mean a propagate signal for bits  $i$  to  $j$ , and  $G_{i,j}$  mean a generate signal for bits  $i$  to  $j$ . For example, the equation

$$C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot c_0)$$

can be rewritten as

$$c_8 = G_{7,4} + (P_{7,4} \cdot G_{3,0}) + (P_{7,4} \cdot P_{3,0} \cdot c_0)$$

This more general notation is useful in creating wider adders.

**B.27** [15] <\$B.6> Write the equations for the carry-lookahead logic for a 64-bit adder using the new notation from Exercise B.26 and using 16-bit adders as building blocks. Include a drawing similar to [Figure B.6.3](#) in your solution.

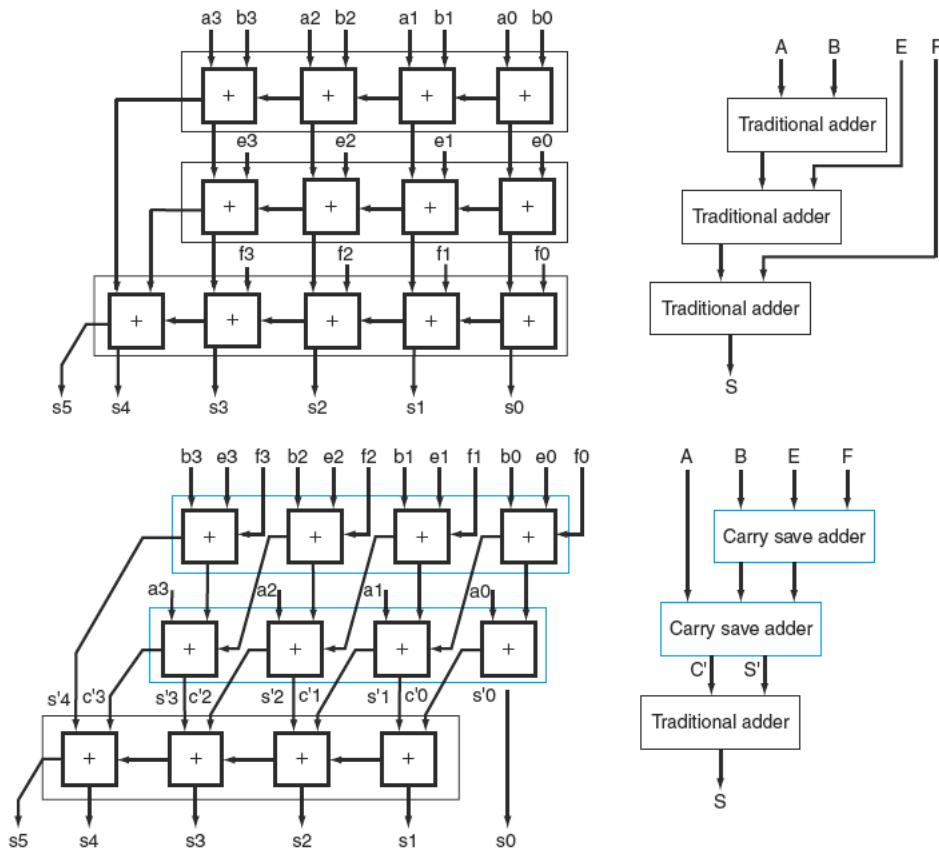
**B.28** [10] <\$B.6> Now calculate the relative performance of adders. Assume that hardware corresponding to any equation containing only OR or AND terms, such as the equations for  $p_i$  and  $g_i$  on page B-40, takes one time unit T. Equations that consist of the OR of several AND terms, such as the equations for  $c_1$ ,  $c_2$ ,  $c_3$ , and  $c_4$  on page B-40, would thus take two time units, 2T. The reason is it would take T to produce the AND terms and then an additional T to produce the result of the OR. Calculate the numbers and performance ratio for 4-bit adders for both ripple carry and carry lookahead. If the terms in equations are further defined by other equations, then add the appropriate delays for those intermediate equations, and continue recursively until the actual input bits of the adder are used in an equation. Include a drawing of each adder labeled with the calculated delays and the path of the worst-case delay highlighted.

**B.29** [15] <\$B.6> This exercise is similar to Exercise B.28, but this time calculate the relative speeds of a 16-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, and the carry-lookahead scheme on page B-39.

**B.30** [15] <\$B.6> This exercise is similar to Exercises B.28 and B.29, but this time calculate the relative speeds of a 64-bit adder using ripple carry only, ripple carry of 4-bit groups that use carry lookahead, ripple carry of 16-bit groups that use carry lookahead, and the carry-lookahead scheme from Exercise B.27.

**B.31** [10] <\$B.6> Instead of thinking of an adder as a device that adds two numbers and then links the carries together, we can think of the adder as a hardware device that can add three inputs together ( $a_i$ ,  $b_i$ ,  $c_i$ ) and produce two outputs ( $s$ ,  $c_i + 1$ ). When adding two numbers together, there is little we can do with this observation. When we are adding more than two operands, it is possible to reduce the cost of the carry. The idea is to form two independent sums, called  $S'$  (sum bits) and  $C'$  (carry bits). At the end of the process, we need to add  $C'$  and  $S'$  together using a normal adder. This technique of delaying carry propagation until the end of a sum of numbers is called *carry save addition*. The block drawing on the lower right of [Figure B.14.1](#) (see below) shows the organization, with two levels of carry save adders connected by a single normal adder.

Calculate the delays to add four 16-bit numbers using full carry-lookahead adders versus carry save with a carry-lookahead adder forming the final sum. (The time unit T in Exercise B.28 is the same.)



**FIGURE B.14.1 Traditional ripple carry and carry save addition of four 4-bit numbers.** The details are shown on the left, with the individual signals in lowercase, and the corresponding higher-level blocks are on the right, with collective signals in uppercase. Note that the sum of four  $n$ -bit numbers can take  $n + 2$  bits.

**B.32** [20] <§B.6> Perhaps the most likely case of adding many numbers at once in a computer would be when trying to multiply more quickly by using many adders to add many numbers in a single clock cycle. Compared to the multiply algorithm in Chapter 3, a carry save scheme with many adders could multiply more than 10 times faster. This exercise estimates the cost and speed of a combinational multiplier to multiply two positive 16-bit numbers. Assume that you have 16 intermediate terms  $M_{15}, M_{14}, \dots, M_0$ , called *partial products*, that contain the multiplicand ANDed with multiplier bits  $m_{15}, m_{14}, \dots, m_0$ . The idea is to use carry save adders to reduce the  $n$  operands into  $2n/3$  in parallel groups of three, and do this repeatedly until you get two large numbers to add together with a traditional adder.

First, show the block organization of the 16-bit carry save adders to add these 16 terms, as shown on the right in [Figure B.14.1](#). Then calculate the delays to add these 16 numbers. Compare this time to the iterative multiplication scheme in Chapter 3 but only assume 16 iterations using a 16-bit adder that has full carry lookahead whose speed was calculated in Exercise B.29.

**B.33** [10] <\$B.6> There are times when we want to add a collection of numbers together. Suppose you wanted to add four 4-bit numbers (A, B, E, F) using 1-bit full adders. Let's ignore carry lookahead for now. You would likely connect the 1-bit adders in the organization at the top of [Figure B.14.1](#). Below the traditional organization is a novel organization of full adders. Try adding four numbers using both organizations to convince yourself that you get the same answer.

**B.34** [5] <\$B.6> First, show the block organization of the 16-bit carry save adders to add these 16 terms, as shown in [Figure B.14.1](#). Assume that the time delay through each 1-bit adder is 2T. Calculate the time of adding four 4-bit numbers to the organization at the top versus the organization at the bottom of [Figure B.14.1](#).

**B.35** [5] <\$B.8> Quite often, you would expect that given a timing diagram containing a description of changes that take place on a data input D and a clock input C (as in [Figures B.8.3 and B.8.6](#) on pages B-52 and B-54, respectively), there would be differences between the output waveforms (Q) for a D latch and a D flip-flop. In a sentence or two, describe the circumstances (e.g., the nature of the inputs) for which there would not be any difference between the two output waveforms.

**B.36** [5] <\$B.8> [Figure B.8.8](#) on page B-55 illustrates the implementation of the register file for the MIPS datapath. Pretend that a new register file is to be built, but that there are only two registers and only one read port, and that each register has only 2 bits of data. Redraw [Figure B.8.8](#) so that every wire in your diagram corresponds to only 1 bit of data (unlike the diagram in [Figure B.8.8](#), in which some wires are 5 bits and some wires are 32 bits). Redraw the registers using D flip-flops. You do not need to show how to implement a D flip-flop or a multiplexor.

**B.37** [10] <\$B.10> A friend would like you to build an “electronic eye” for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light “moves” from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite-state machine used to specify the electronic eye. Note that the rate of the eye’s movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.

**B.38** [10] <\$B.10> Assign state numbers to the states of the finite-state machine you constructed for Exercise B.37 and write a set of logic equations for each of the outputs, including the next-state bits.

**B.39** [15] <§§B.2, B.8, B.10> Construct a 3-bit counter using three D flip-flops and a selection of gates. The inputs should consist of a signal that resets the counter to 0, called *reset*, and a signal to increment the counter, called *inc*. The outputs should be the value of the counter. When the counter has value 7 and is incremented, it should wrap around and become 0.

**B.40** [20] <\$B.10> A *Gray code* is a sequence of binary numbers with the property that no more than 1 bit changes in going from one element of the sequence to another. For example, here is a 3-bit binary Gray code: 000, 001, 011, 010, 110, 111, 101, and 100. Using three D flip-flops and a PLA, construct a 3-bit Gray code counter that has two inputs: *reset*, which sets the counter to 000, and *inc*, which makes the counter go to the next value in the sequence. Note that the code is cyclic, so that the value after 100 in the sequence is 000.

**B.41** [25] <\$B.10> We wish to add a yellow light to our traffic light example on page B-68. We will do this by changing the clock to run at 0.25 Hz (a 4-second clock cycle time), which is the duration of a yellow light. To prevent the green and red lights from cycling too fast, we add a 30-second timer. The timer has a single input, called *TimerReset*, which restarts the timer, and a single output, called *TimerSignal*, which indicates that the 30-second period has expired. Also, we must redefine the traffic signals to include yellow. We do this by defining two output signals for each light: green and yellow. If the output NSgreen is asserted, the green light is on; if the output NSyellow is asserted, the yellow light is on. If both signals are off, the red light is on. Do *not* assert both the green and yellow signals at the same time, since American drivers will certainly be confused, even if European drivers understand what this means! Draw the graphical representation for the finite-state machine for this improved controller. Choose names for the states that are *different* from the names of the outputs.

**B.42** [15] <\$B.10> Write down the next-state and output-function tables for the traffic light controller described in Exercise B.41.

**B.43** [15] <§§B.2, B.10> Assign state numbers to the states in the traf-fic light example of Exercise B.41 and use the tables of Exercise B.42 to write a set of logic equations for each of the outputs, including the next-state outputs.

**B.44** [15] <§§B.3, B.10> Implement the logic equations of Exercise B.43 as a PLA.

§B.2, page B-8: No. If  $A = 1, C = 1, B = 0$ , the first is true, but the second is false.

§B.3, page B-20: C.

§B.4, page B-22: They are all exactly the same.

§B.4, page B-26:  $A = 0, B = 1$ .

§B.5, page B-38: 2.

§B.6, page B-47: 1.

§B.8, page B-58: c.

§B.10, page B-72: b.

§B.11, page B-77: b.

**Answers to  
Check Yourself**

This page intentionally left blank

# Index

*Note:* Online information is listed by chapter and section number followed by page numbers (OL3.11-7). Page references preceded by a single letter with hyphen refer to appendices.

1-bit ALU, B-26–29. *See also* Arithmetic logic unit (ALU)  
adder, B-27  
CarryOut, B-28  
for most significant bit, B-33  
illustrated, B-29  
logical unit for AND/OR, B-27  
performing AND, OR, and addition, B-31, B-33  
32-bit ALU, B-29–38. *See also* Arithmetic logic unit (ALU)  
defining in Verilog, B-35–38  
from 31 copies of 1-bit ALU, B-34  
illustrated, B-36  
ripple carry adder, B-29  
tailoring to MIPS, B-31–35  
with 32 1-bit ALUs, B-30  
32-bit immediate operands, 112–113  
7090/7094 hardware, OL3.11-7

## A

Absolute references, 126  
Abstractions  
hardware/software interface, 22  
principle, 22  
to simplify design, 11  
Accumulator architectures, OL2.21-2  
Acronyms, 9  
Active matrix, 18  
add (Add), 64  
add.d (FP Add Double), A-73  
add.s (FP Add Single), A-74  
Add unsigned instruction, 180  
addi (Add Immediate), 64  
Addition, 178–182. *See also* Arithmetic binary, 178–179  
floating-point, 203–206, 211, A-73–74  
instructions, A-51  
operands, 179  
significands, 203  
speed, 182

addiu (Add Imm.Unsigned), 119  
Address interleaving, 381  
Address select logic, D-24, D-25  
Address space, 428, 431  
extending, 479  
flat, 479  
ID (ASID), 446  
inadequate, OL5.17–6  
shared, 519–520  
single physical, 517  
unmapped, 450  
virtual, 446  
Address translation  
for ARM cortex-A8, 471  
defined, 429  
fast, 438–439  
for Intel core i7, 471  
TLB for, 438–439  
Address-control lines, D-26  
Addresses  
32-bit immediates, 113–116  
base, 69  
byte, 69  
defined, 68  
memory, 77  
virtual, 428–431, 450  
Addressing  
32-bit immediates, 113–116  
base, 116  
displacement, 116  
immediate, 116  
in jumps and branches, 113–116  
MIPS modes, 116–118  
PC-relative, 114, 116  
pseudodirect, 116  
register, 116  
x86 modes, 152  
Addressing modes, A-45–47  
desktop architectures, E-6  
addu (Add Unsigned), 64  
Advanced Vector Extensions (AVX), 225,  
227

AGP, C-9  
Algol-60, OL2.21-7  
Aliasing, 444  
Alignment restriction, 69–70  
All-pairs N-body algorithm, C-65  
Alpha architecture  
bit count instructions, E-29  
floating-point instructions, E-28  
instructions, E-27–29  
no divide, E-28  
PAL code, E-28  
unaligned load-store, E-28  
VAX floating-point formats, E-29  
ALU control, 259–261. *See also* Arithmetic logic unit (ALU)  
bits, 260  
logic, D-6  
mapping to gates, D-4–7  
truth tables, D-5  
ALU control block, 263  
defined, D-4  
generating ALU control bits, D-6  
ALUOp, 260, D-6  
bits, 260, 261  
control signal, 263  
Amazon Web Services (AWS), 425  
AMD Opteron X4 (Barcelona), 543, 544  
AMD64, 151, 224, OL2.21-6  
Amdahl's law, 401, 503  
corollary, 49  
defined, 49  
fallacy, 556  
and (AND), 64  
AND gates, B-12, D-7  
AND operation, 88  
AND operation, A-52, B-6  
andi (And Immediate), 64  
Annual failure rate (AFR), 418  
versus.MTTF of disks, 419–420  
Antidependence, 338  
Antifuse, B-78  
Apple computer, OL1.12-7

- Apple iPad 2 A1395, 20  
 logic board of, 20  
 processor integrated circuit of, 21
- Application binary interface (ABI), 22
- Application programming interfaces (APIs)  
 defined, C-4  
 graphics, C-14
- Architectural registers, 347
- Arithmetic, 176–236  
 addition, 178–182  
 addition and subtraction, 178–182  
 division, 189–195  
 fallacies and pitfalls, 229–232  
 floating-point, 196–222  
 historical perspective, 236  
 multiplication, 183–188  
 parallelism and, 222–223  
 Streaming SIMD Extensions and  
   advanced vector extensions in x86, 224–225  
 subtraction, 178–182  
 subword parallelism, 222–223  
 subword parallelism and matrix  
   multiply, 225–228
- Arithmetic instructions. *See also*  
 Instructions  
 desktop RISC, E-11  
 embedded RISC, E-14  
 logical, 251  
 MIPS, A-51–57  
 operands, 66–, 73
- Arithmetic intensity, 541
- Arithmetic logic unit (ALU). *See also*  
 ALU control; Control units  
 1-bit, B-26–29  
 32-bit, B-29–38  
 before forwarding, 309  
 branch datapath, 254  
 hardware, 180  
 memory-reference instruction use, 245  
 for register values, 252  
 R-format operations, 253  
 signed-immediate input, 312
- ARM Cortex-A8, 244, 345–346  
 address translation for, 471  
 caches in, 472  
 data cache miss rates for, 474  
 memory hierarchies of, 471–475  
 performance of, 473–475  
 specification, 345
- TLB hardware for, 471
- ARM instructions, 145–147  
 12-bit immediate field, 148  
 addressing modes, 145  
 block loads and stores, 149  
 brief history, OL2.21–5  
 calculations, 145–146  
 compare and conditional branch, 147–148  
 condition field, 324  
 data transfer, 146  
 features, 148–149  
 formats, 148  
 logical, 149  
 MIPS similarities, 146  
 register-register, 146  
 unique, E-36–37
- ARMv7, 62
- ARMv8, 158–159
- ARPANET, OL1.12–10
- Arrays, 415  
 logic elements, B-18–19  
 multiple dimension, 218  
 pointers *versus*, 141–145  
 procedures for setting to zero, 142
- ASCII  
 binary numbers *versus*, 107  
 character representation, 106  
 defined, 106  
 symbols, 109
- Assembler directives, A-5
- Assemblers, 124–126, A-10–17  
 conditional code assembly, A-17  
 defined, 14, A-4  
 function, 125, A-10  
 macros, A-4, A-15–17  
 microcode, D-30  
 number acceptance, 125  
 object file, 125  
 pseudoinstructions, A-17  
 relocation information, A-13, A-14  
 speed, A-13  
 symbol table, A-12
- Assembly language, 15  
 defined, 14, 123  
 drawbacks, A-9–10  
 floating-point, 212  
 high-level languages *versus*, A-12  
 illustrated, 15  
 MIPS, 64, 84, A-45–80  
 production of, A-8–9
- programs, 123  
 translating into machine language, 84  
 when to use, A-7–9
- Asserted signals, 250, B-4
- Associativity  
 in caches, 405  
 degree, increasing, 404, 455  
 increasing, 409  
 set, tag size *versus*, 409
- Atomic compare and swap, 123
- Atomic exchange, 121
- Atomic fetch-and-increment, 123
- Atomic memory operation, C-21
- Attribute interpolation, C-43–44
- Automobiles, computer application in, 4
- Average memory access time (AMAT), 402  
 calculating, 403
- B**
- Backpatching, A-13
- Bandwidth, 30–31  
 bisection, 532  
 external to DRAM, 398  
 memory, 380–381, 398  
 network, 535
- Barrier synchronization, C-18  
 defined, C-20  
 for thread communication, C-34
- Base addressing, 69, 116
- Base registers, 69
- Basic block, 93
- Benchmarks, 538–540  
 defined, 46  
 Linpack, 538, OL3.11–4  
 multicores, 522–529  
 multiprocessor, 538–540  
 NAS parallel, 540  
 parallel, 539  
 PARSEC suite, 540  
 SPEC CPU, 46–48  
 SPEC power, 48–49  
 SPECrate, 538–539  
 Stream, 548
- beq (Branch On Equal), 64
- bge (Branch Greater Than or Equal), 125
- bgt (Branch Greater Than), 125
- Biased notation, 79, 200
- Big-endian byte order, 70, A-43
- Binary numbers, 81–82

ASCII *versus*, 107  
 conversion to decimal numbers, 76  
 defined, 73  
 Bisection bandwidth, 532  
 Bit maps  
   defined, 18, 73  
   goal, 18  
   storing, 18  
 Bit-Interleaved Parity (RAID 3), OL5.11-5  
 Bits  
   ALUOp, 260, 261  
   defined, 14  
   dirty, 437  
   guard, 220  
   patterns, 220–221  
   reference, 435  
   rounding, 220  
   sign, 75  
   state, D-8  
   sticky, 220  
   valid, 383  
 ble (Branch Less Than or Equal), 125  
 Blocking assignment, B-24  
 Blocking factor, 414  
 Block-Interleaved Parity (RAID 4),  
   OL5.11-5–5.11-6  
 Blocks  
   combinational, B-4  
   defined, 376  
   finding, 456  
   flexible placement, 402–404  
   least recently used (LRU), 409  
   loads/stores, 149  
   locating in cache, 407–408  
   miss rate and, 391  
   multiword, mapping addresses to, 390  
   placement locations, 455–456  
   placement strategies, 404  
   replacement selection, 409  
   replacement strategies, 457  
   spatial locality exploitation, 391  
   state, B-4  
   valid data, 386  
 blt (Branch Less Than), 125  
 bne (Branch On Not Equal), 64  
 Bonding, 28  
 Boolean algebra, B-6  
 Bounds check shortcut, 95  
 Branch datapath  
   ALU, 254

operations, 254  
 Branch delay slots  
   defined, 322  
   scheduling, 323  
 Branch equal, 318  
 Branch instructions, A-59–63  
   jump instruction *versus*, 270  
   list of, A-60–63  
   pipeline impact, 317  
 Branch not taken  
   assumption, 318  
   defined, 254  
 Branch prediction  
   as control hazard solution, 284  
   buffers, 321, 322  
   defined, 283  
   dynamic, 284, 321–323  
   static, 335  
 Branch predictors  
   accuracy, 322  
   correlation, 324  
   information from, 324  
   tournament, 324  
 Branch taken  
   cost reduction, 318  
   defined, 254  
 Branch target  
   addresses, 254  
   buffers, 324  
 Branches. *See also* Conditional branches  
   addressing in, 113–116  
   compiler creation, 91  
   condition, 255  
   decision, moving up, 318  
   delayed, 96, 255, 284, 318–319, 322,  
     324  
   ending, 93  
   execution in ID stage, 319  
   pipelined, 318  
   target address, 318  
   unconditional, 91  
 Branch-on-equal instruction, 268  
 Bubble Sort, 140  
 Bubbles, 314  
 Bus-based coherent multiprocessors,  
   OL6.15-7  
 Buses, B-19  
 Bytes  
   addressing, 70  
   order, 70, A-43

**C**

C.mmp, OL6.15-4  
 C language  
   assignment, compiling into MIPS,  
     65–66  
   compiling, 145, OL2.15-2–2.15-3  
   compiling assignment with registers,  
     67–68  
   compiling while loops in, 92  
   sort algorithms, 141  
   translation hierarchy, 124  
   translation to MIPS assembly language,  
     65  
   variables, 102  
 C++ language, OL2.15-27, OL2.21-8  
 Cache blocking and matrix multiply,  
   475–476  
 Cache coherence, 466–470  
   coherence, 466  
   consistency, 466  
   enforcement schemes, 467–468  
   implementation techniques,  
     OL5.12-11–5.12-12  
   migration, 467  
   problem, 466, 467, 470  
   protocol example, OL5.12-12–5.12-16  
   protocols, 468  
   replication, 468  
   snooping protocol, 468–469  
   snoopy, OL5.12-17  
   state diagram, OL5.12-16  
 Cache coherency protocol, OL5.12-12–16  
   finite-state transition diagram, OL5.12-15  
   functioning, OL5.12-14  
   mechanism, OL5.12-14  
   state diagram, OL5.12-16  
   states, OL5.12-13  
   write-back cache, OL5.12-15  
 Cache controllers, 470  
   coherent cache implementation  
     techniques, OL5.12-11–5.12-12  
   implementing, OL5.12-2  
   snoopy cache coherence, OL5.12-17  
   SystemVerilog, OL5.12-2  
 Cache hits, 443  
 Cache misses  
   block replacement on, 457  
   capacity, 459

Cache misses (*Continued*)

compulsory, 459  
 conflict, 459  
 defined, 392  
 direct-mapped cache, 404  
 fully associative cache, 406  
 handling, 392–393  
 memory-stall clock cycles, 399  
 reducing with flexible block placement, 402–404  
 set-associative cache, 405  
 steps, 393  
 in write-through cache, 393  
 Cache performance, 398–417  
 calculating, 400  
 hit time and, 401–402  
 impact on processor performance, 400  
 Cache-aware instructions, 482  
 Caches, 383–398. *See also* Blocks  
     accessing, 386–389  
     in ARM cortex-A8, 472  
     associativity in, 405–406  
     bits in, 390  
     bits needed for, 390  
     contents illustration, 387  
     defined, 21, 383–384  
     direct-mapped, 384, 385, 390, 402  
     empty, 386–387  
     FSM for controlling, 461–462  
     fully associative, 403  
     GPU, C-38  
     inconsistent, 393  
     index, 388  
     in Intel Core i7, 472  
     Intrinsity FastMATH example, 395–398  
     locating blocks in, 407–408  
     locations, 385  
     multilevel, 398, 410  
     nonblocking, 472  
     physically addressed, 443  
     physically indexed, 443  
     physically tagged, 443  
     primary, 410, 417  
     secondary, 410, 417  
     set-associative, 403  
     simulating, 478  
     size, 389  
     split, 397  
     summary, 397–398  
     tag field, 388  
     tags, OL5.12-3, OL5.12-11

virtual memory and TLB integration, 440–441  
 virtually addressed, 443  
 virtually indexed, 443  
 virtually tagged, 443  
 write-back, 394, 395, 458  
 write-through, 393, 395, 457  
 writes, 393–395  
 Callee, 98, 99  
 Callee-saved register, A-23  
 Caller, 98  
 Caller-saved register, A-23  
 Capabilities, OL5.17-8  
 Capacity misses, 459  
 Carry lookahead, B-38–47  
     4-bit ALUs using, B-45  
     adder, B-39  
     fast, with first level of abstraction, B-39–40  
     fast, with “infinite” hardware, B-38–39  
     fast, with second level of abstraction, B-40–46  
     plumbing analogy, B-42, B-43  
     ripple carry speed *versus*, B-46  
     summary, B-46–47  
 Carry save adders, 188  
 Cause register  
     defined, 327  
     fields, A-34, A-35  
 OLC 6600, OL1.12-7, OL4.16-3  
 Cell phones, 7  
 Central processor unit (CPU). *See also* Processors  
     classic performance equation, 36–40  
     coprocessor 0, A-33–34  
     defined, 19  
     execution time, 32, 33–34  
     performance, 33–35  
     system, time, 32  
     time, 399  
     time measurements, 33–34  
     user, time, 32  
 Cg pixel shader program, C-15–17  
 Characters  
     ASCII representation, 106  
     in Java, 109–111  
 Chips, 19, 25, 26  
     manufacturing process, 26  
 Classes  
     defined, OL2.15–15  
     packages, OL2.15–21  
 Clock cycles

defined, 33  
 memory-stall, 399  
 number of registers and, 67  
 worst-case delay and, 272  
 Clock cycles per instruction (CPI), 35, 282  
 one level of caching, 410  
 two levels of caching, 410  
 Clock rate  
     defined, 33  
     frequency switched as function of, 41  
     power and, 40  
 Clocking methodology, 249–251, B-48  
     edge-triggered, 249, B-48, B-73  
     level-sensitive, B-74, B-75–76  
     for predictability, 249  
 Clocks, B-48–50  
     edge, B-48, B-50  
     in edge-triggered design, B-73  
     skew, B-74  
     specification, B-57  
     synchronous system, B-48–49  
 Cloud computing, 533  
     defined, 7  
 Cluster networking, 537–538, OL6.9–12  
 Clusters, OL6.15–8–6.15–9  
     defined, 30, 500, OL6.15–8  
     isolation, 530  
     organization, 499  
     scientific computing on, OL6.15–8  
 Cm\*, OL6.15–4  
 CMOS (complementary metal oxide semiconductor), 41  
 Coarse-grained multithreading, 514  
 Cobol, OL2.21–7  
 Code generation, OL2.15–13  
 Code motion, OL2.15–7  
 Cold-start miss, 459  
 Collision misses, 459  
 Column major order, 413  
 Combinational blocks, B-4  
 Combinational control units, D-4–8  
 Combinational elements, 248  
 Combinational logic, 249, B-3, B-9–20  
     arrays, B-18–19  
     decoders, B-9  
     defined, B-5  
     don’t cares, B-17–18  
     multiplexors, B-10  
     ROMs, B-14–16  
     two-level, B-11–14  
 Verilog, B-23–26

- Commercial computer development, OL1.12-4–1.12-10  
Commit units  
  buffer, 339–340  
  defined, 339–340  
  in update control, 343  
Common case fast, 11  
Common subexpression elimination, OL2.15-6  
Communication, 23–24  
  overhead, reducing, 44–45  
  thread, C-34  
Compact code, OL2.21-4  
Comparison instructions, A-57–59  
  floating-point, A-74–75  
  list of, A-57–59  
Comparisons, 93  
  constant operands in, 93  
  signed *versus* unsigned, 94–95  
Compilers, 123–124  
  branch creation, 92  
  brief history, OL2.21-9  
  conservative, OL2.15-6  
  defined, 14  
  front end, OL2.15-3  
  function, 14, 123–124, A-5–6  
  high-level optimizations, OL2.15-4  
  ILP exploitation, OL4.16-5  
  Just In Time (JIT), 132  
  machine language production, A-8–9, A-10  
  optimization, 141, OL2.21-9  
  speculation, 333–334  
  structure, OL2.15-2  
Compiling  
  C assignment statements, 65–66  
  C language, 92–93, 145, OL2.15-2–2.15-3  
  floating-point programs, 214–217  
  if-then-else, 91  
  in Java, OL2.15-19  
  procedures, 98, 101–102  
  recursive procedures, 101–102  
  while loops, 92–93  
Compressed sparse row (CSR) matrix, C-55, C-56  
Compulsory misses, 459  
Computer architects, 11–12  
  abstraction to simplify design, 11  
  common case fast, 11  
  dependability via redundancy, 12  
  hierarchy of memories, 12  
Moore's law, 11  
parallelism, 12  
pipelining, 12  
prediction, 12  
Computers  
  application classes, 5–6  
  applications, 4  
  arithmetic for, 176–236  
  characteristics, OL1.12–12  
  commercial development, OL1.12-4–1.12-10  
  component organization, 17  
  components, 17, 177  
  design measure, 53  
  desktop, 5  
  embedded, 5, A-7  
  first, OL1.12-2–1.12-4  
  in information revolution, 4  
  instruction representation, 80–87  
  performance measurement, OL1.12-10  
PostPC Era, 6–7  
principles, 86  
servers, 5  
Condition field, 324  
Conditional branches  
  ARM, 147–148  
  changing program counter with, 324  
  compiling if-then-else into, 91  
  defined, 90  
  desktop RISC, E-16  
  embedded RISC, E-16  
  implementation, 96  
  in loops, 115  
  PA-RISC, E-34, E-35  
  PC-relative addressing, 114  
  RISC, E-10–16  
  SPARC, E-10–12  
Conditional move instructions, 324  
Conflict misses, 459  
Constant memory, C-40  
Constant operands, 72–73  
  in comparisons, 93  
  frequent occurrence, 72  
Constant-manipulating instructions, A-57  
Content Addressable Memory (CAM), 408  
Context switch, 446  
Control  
  ALU, 259–261  
  challenge, 325–326  
  finishing, 269–270  
forwarding, 307  
FSM, D-8–21  
implementation, optimizing, D-27–28  
for jump instruction, 270  
mapping to hardware, D-2–32  
memory, D-26  
organizing, to reduce logic, D-31–32  
pipelined, 300–303  
Control flow graphs, OL2.15-9–2.15-10  
  illustrated examples, OL2.15-9, OL2.15-10  
Control functions  
  ALU, mapping to gates, D-4–7  
  defining, 264  
  PLA, implementation, D-7, D-20–21  
ROM, encoding, D-18–19  
  for single-cycle implementation, 269  
Control hazards, 281–282, 316–325  
  branch delay reduction, 318–319  
  branch not taken assumption, 318  
  branch prediction as solution, 284  
  delayed decision approach, 284  
  dynamic branch prediction, 321–323  
  logic implementation in Verilog, OL4.13-8  
  pipeline stalls as solution, 282  
  pipeline summary, 324  
  simplicity, 317  
  solutions, 282  
  static multiple-issue processors and, 335–336  
Control lines  
  asserted, 264  
  in datapath, 263  
  execution/address calculation, 300  
  final three stages, 303  
  instruction decode/register file read, 300  
  instruction fetch, 300  
  memory access, 302  
  setting of, 264  
  values, 300  
  write-back, 302  
Control signals  
  ALUOp, 263  
  defined, 250  
  effect of, 264  
  multi-bit, 264  
  pipelined datapaths with, 300–303  
  truth tables, D-14

Control units, 247. *See also* Arithmetic logic unit (ALU)  
 address select logic, D-24, D-25  
 combinational, implementing, D-4–8  
 with explicit counter, D-23  
 illustrated, 265  
 logic equations, D-11  
 main, designing, 261–264  
 as microcode, D-28  
 MIPS, D-10  
 next-state outputs, D-10, D-12–13  
 output, 259–261, D-10  
 Conversion instructions, A-75–76  
 Cooperative thread arrays (CTAs), C-30  
 Coprocessors, A-33–34  
 defined, 218  
 move instructions, A-71–72  
 Core MIPS instruction set, 236. *See also*  
 MIPS  
 abstract view, 246  
 desktop RISC, E-9–11  
 implementation, 244–248  
 implementation illustration, 247  
 overview, 245  
 subset, 244  
 Cores  
 defined, 43  
 number per chip, 43  
 Correlation predictor, 324  
 Cosmic Cube, OL6.15–7  
 Count register, A-34  
 CPU, 9  
 Cray computers, OL3.11–5–3.11–6  
 Critical word first, 392  
 Crossbar networks, 535  
 CTSS (Compatible Time-Sharing System), OL5.18–9  
 CUDA programming environment, 523, C-5  
 barrier synchronization, C-18, C-34  
 development, C-17, C-18  
 hierarchy of thread groups, C-18  
 kernels, C-19, C-24  
 key abstractions, C-18  
 paradigm, C-19–23  
 parallel plus-scan template, C-61  
 per-block shared memory, C-58  
 plus-reduction implementation, C-63  
 programs, C-6, C-24  
 scalable parallel programming with, C-17–23

shared memories, C-18  
 threads, C-36  
 Cyclic redundancy check, 423  
 Cylinder, 381

**D**

D flip-flops, B-51, B-53  
 D latches, B-51, B-52  
 Data bits, 421  
 Data flow analysis, OL2.15–11  
 Data hazards, 278, 303–316. *See also*  
 Hazards  
 forwarding, 278, 303–316  
 load-use, 280, 318  
 stalls and, 313–316  
 Data layout directives, A-14  
 Data movement instructions, A-70–73  
 Data parallel problem decomposition, C-17, C-18  
 Data race, 121  
 Data segment, A-13  
 Data selectors, 246  
 Data transfer instructions. *See also*  
 Instructions  
 defined, 68  
 load, 68  
 offset, 69  
 store, 71  
 Datacenters, 7  
 Data-level parallelism, 508  
 Datapath elements  
 defined, 251  
 sharing, 256  
 Datapaths  
 branch, 254  
 building, 251–259  
 control signal truth tables, D-14  
 control unit, 265  
 defined, 19  
 design, 251  
 exception handling, 329  
 for fetching instructions, 253  
 for hazard resolution via forwarding, 311  
 for jump instruction, 270  
 for memory instructions, 256  
 for MIPS architecture, 257  
 in operation for branch-on-equal instruction, 268  
 in operation for load instruction, 267

in operation for R-type instruction, 266  
 operation of, 264–269  
 pipelined, 286–303  
 for R-type instructions, 256, 264–265  
 single, creating, 256  
 single-cycle, 283  
 static two-issue, 336  
 Deasserted signals, 250, B-4  
 Debugging information, A-13  
 DEC PDP-8, OL2.21–3  
 Decimal numbers  
 binary number conversion to, 76  
 defined, 73  
 Decision-making instructions, 90–96  
 Decoders, B-9  
 two-level, B-65  
 Decoding machine language, 118–120  
 Defect, 26  
 Delayed branches, 96. *See also* Branches  
 as control hazard solution, 284  
 defined, 255  
 embedded RISCs and, E-23  
 for five-stage pipelines, 26, 323–324  
 reducing, 318–319  
 scheduling limitations, 323  
 Delayed decision, 284  
 DeMorgan's theorems, B-11  
 Denormalized numbers, 222  
 Dependability via redundancy, 12  
 Dependable memory hierarchy, 418–423  
 failure, defining, 418  
 Dependences  
 between pipeline registers, 308  
 between pipeline registers and ALU inputs, 308  
 bubble insertion and, 314  
 detection, 306–308  
 name, 338  
 sequence, 304  
 Design  
 compromises and, 161  
 datapath, 251  
 digital, 354  
 logic, 248–251, B-1–79  
 main control unit, 261–264  
 memory hierarchy, challenges, 460  
 pipelining instruction sets, 277  
 Desktop and server RISCs. *See also*  
 Reduced instruction set computer (RISC) architectures

- addressing modes, E-6  
 architecture summary, E-4  
 arithmetic/logical instructions, E-11  
 conditional branches, E-16  
 constant extension summary, E-9  
 control instructions, E-11  
 conventions equivalent to MIPS core, E-12  
 data transfer instructions, E-10  
 features added to, E-45  
 floating-point instructions, E-12  
 instruction formats, E-7  
 multimedia extensions, E-16–18  
 multimedia support, E-18  
 types of, E-3
- Desktop computers, defined, 5  
 Device driver, OL6.9–5  
**DGEMM** (Double precision General Matrix Multiply), 225, 352, 413, 553  
 cache blocked version of, 415  
 optimized C version of, 226, 227, 476  
 performance, 354, 416
- Dicing**, 27  
**Dies**, 26, 26–27  
 Digital design pipeline, 354  
 Digital signal-processing (DSP) extensions, E-19  
 DIMMs (dual inline memory modules), OL5.17–5  
 Direct Data IO (DDIO), OL6.9–6  
 Direct memory access (DMA), OL6.9–4  
 Direct3D, C-13  
 Direct-mapped caches. *See also* Caches  
     address portions, 407  
     choice of, 456  
     defined, 384, 402  
     illustrated, 385  
     memory block location, 403  
     misses, 405  
     single comparator, 407  
     total number of bits, 390
- Dirty bit, 437  
 Dirty pages, 437  
 Disk memory, 381–383  
 Displacement addressing, 116  
 Distributed Block-Interleaved Parity (RAID 5), OL5.11–6  
**div** (Divide), A-52  
**div.d** (FP Divide Double), A-76  
**div.s** (FP Divide Single), A-76  
 Divide algorithm, 190
- Dividend, 189  
**Division**, 189–195  
     algorithm, 191  
     dividend, 189  
     divisor, 189  
**Divisor**, 189  
**divu** (Divide Unsigned), A-52. *See also* Arithmetic  
     faster, 194  
     floating-point, 211, A-76  
     hardware, 189–192  
     hardware, improved version, 192  
     instructions, A-52–53  
     in MIPS, 194  
     operands, 189  
     quotient, 189  
     remainder, 189  
     signed, 192–194  
     SRT, 194  
**Don't cares**, B-17–18  
     example, B-17–18  
     term, 261  
**Double data rate (DDR)**, 379  
**Double Data Rate RAMs (DDRRAMs)**, 379–380, B-65  
**Double precision.** *See also* Single precision  
     defined, 198  
     FMA, C-45–46  
     GPU, C-45–46, C-74  
     representation, 201  
**Double words**, 152  
**Dual inline memory modules (DIMMs)**, 381  
**Dynamic branch prediction**, 321–323. *See also* Control hazards  
     branch prediction buffer, 321  
     loops and, 321–323  
**Dynamic hardware predictors**, 284  
**Dynamic multiple-issue processors**, 333, 339–341. *See also* Multiple issue  
     pipeline scheduling, 339–341  
     superscalar, 339  
**Dynamic pipeline scheduling**, 339–341  
     commit unit, 339–340  
     concept, 339–340  
     hardware-based speculation, 341  
     primary units, 340  
     reorder buffer, 343  
     reservation station, 339–340  
**Dynamic random access memory (DRAM)**, 378, 379–381, B-63–65
- bandwidth external to, 398  
 cost, 23  
 defined, 19, B-63  
 DIMM, OL5.17–5  
 Double Date Rate (DDR), 379–380  
 early board, OL5.17–4  
 GPU, C-37–38  
 growth of capacity, 25  
 history, OL5.17–2  
 internal organization of, 380  
 pass transistor, B-63  
 SIMM, OL5.17–5, OL5.17–6  
 single-transistor, B-64  
 size, 398  
 speed, 23  
 synchronous (SDRAM), 379–380, B-60, B-65  
 two-level decoder, B-65
- Dynamically linked libraries (DLLs)**, 129–131  
 defined, 129  
 lazy procedure linkage version, 130
- E**
- Early restart, 392  
**Edge-triggered clocking methodology**, 249, 250, B-48, B-73  
 advantage, B-49  
 clocks, B-73  
 drawbacks, B-74  
 illustrated, B-50  
 rising edge/falling edge, B-48
- EDSAC** (Electronic Delay Storage Automatic Calculator), OL1.12–3, OL5.17–2  
**Eispack**, OL3.11–4  
**Electrically erasable programmable read-only memory (EEPROM)**, 381
- Elements**  
     combinational, 248  
     datapath, 251, 256  
     memory, B-50–58  
     state, 248, 250, 252, B-48, B-50
- Embedded computers**, 5  
     application requirements, 6  
     defined, A-7  
     design, 5  
     growth, OL1.12–12–1.12–13
- Embedded Microprocessor Benchmark Consortium (EEMBC)**, OL1.12–12

Embedded RISCs. *See also* Reduced instruction set computer (RISC) architectures  
 addressing modes, E-6  
 architecture summary, E-4  
 arithmetic/logical instructions, E-14  
 conditional branches, E-16  
 constant extension summary, E-9  
 control instructions, E-15  
 data transfer instructions, E-13  
 delayed branch and, E-23  
 DSP extensions, E-19  
 general purpose registers, E-5  
 instruction conventions, E-15  
 instruction formats, E-8  
 multiply-accumulate approaches, E-19  
 types of, E-4

**Encoding**  
 defined, D-31  
 floating-point instruction, 213  
 MIPS instruction, 83, 119, A-49  
 ROM control function, D-18–19  
 ROM logic function, B-15  
 x86 instruction, 155–156

**ENIAC** (Electronic Numerical Integrator and Calculator), OL1.12-2, OL1.12-3, OL5.17-2

**EPIC**, OL4.16–5

**Error correction**, B-65–67

**Error Detecting and Correcting Code (RAID 2)**, OL5.11–5

**Error detection**, B-66

**Error detection code**, 420

**Ethernet**, 23

**EX stage**  
 load instructions, 292  
 overflow exception detection, 328  
 store instructions, 294

**Exabyte**, 6

**Exception enable**, 447

**Exception handlers**, A-36–38  
 defined, A-35  
 return from, A-38

**Exception program counters (EPCs)**, 326  
 address capture, 331  
 copying, 181  
 defined, 181, 327  
 in restart determination, 326–327  
 transferring, 182

**Exceptions**, 325–332, A-33–38  
 association, 331–332

datapath with controls for handling, 329  
 defined, 180, 326  
 detecting, 326  
 event types and, 326  
 imprecise, 331–332  
 instructions, A-80  
 interrupts *versus*, 325–326  
 in MIPS architecture, 326–327  
 overflow, 329  
 PC, 445, 446–447  
 pipelined computer example, 328  
 in pipelined implementation, 327–332  
 precise, 332  
 reasons for, 326–327  
 result due to overflow in add instruction, 330  
 saving/restoring stage on, 450

**Exclusive OR (XOR) instructions**, A-57

**Executable files**, A-4  
 defined, 126  
 linker production, A-19

**Execute or address calculation stage**, 292

**Execute/address calculation**  
 control line, 300  
 load instruction, 292  
 store instruction, 292

**Execution time**  
 as valid performance measure, 51  
 CPU, 32, 33–34  
 pipelining and, 286

**Explicit counters**, D-23, D-26

**Exponents**, 197–198

**External labels**, A-10

**F**

**Facilities**, A-14–17

**Failures, synchronizer**, B-77

**Fallacies**. *See also* Pitfalls  
 add immediate unsigned, 227  
 Amdahl's law, 556  
 arithmetic, 229–232  
 assembly language for performance, 159–160  
 commercial binary compatibility importance, 160  
 defined, 49  
 GPUs, C-72–74, C-75  
 low utilization uses little power, 50  
 peak performance, 556

**pipelining**, 355–356  
**powerful instructions mean higher performance**, 159  
 right shift, 229

**False sharing**, 469

**Fast carry**  
 with “infinite” hardware, B-38–39  
 with first level of abstraction, B-39–40  
 with second level of abstraction, B-40–46

**Fast Fourier Transforms (FFT)**, C-53

**Fault avoidance**, 419

**Fault forecasting**, 419

**Fault tolerance**, 419

**Fermi architecture**, 523, 552

**Field programmable devices (FPDs)**, B-78

**Field programmable gate arrays (FPGAs)**, B-78

**Fields**  
 Cause register, A-34, A-35  
 defined, 82  
 format, D-31  
 MIPS, 82–83  
 names, 82  
 Status register, A-34, A-35

**Files, register**, 252, 257, B-50, B-54–56

**Fine-grained multithreading**, 514

**Finite-state machines (FSMs)**, 451–466, B-67–72  
 control, D-8–22  
 controllers, 464  
 for multicycle control, D-9  
 for simple cache controller, 464–466  
 implementation, 463, B-70  
 Mealy, 463  
 Moore, 463  
 next-state function, 463, B-67  
 output function, B-67, B-69  
 state assignment, B-70  
 state register implementation, B-71  
 style of, 463  
 synchronous, B-67  
 SystemVerilog, OL5.12–7  
 traffic light example, B-68–70

**Flash memory**, 381  
 characteristics, 23  
 defined, 23

**Flat address space**, 479

**Flip-flops**  
 D flip-flops, B-51, B-53  
 defined, B-51

Floating point, 196–222, 224  
assembly language, 212  
backward step, OL3.11-4–3.11-5  
binary to decimal conversion, 202  
branch, 211  
challenges, 232–233  
diversity *versus* portability, OL3.11-3–3.11-4  
division, 211  
first dispute, OL3.11-2–3.11-3  
form, 197  
fused multiply add, 220  
guard digits, 218–219  
history, OL3.11-3  
IEEE 754 standard, 198, 199  
instruction encoding, 213  
intermediate calculations, 218  
machine language, 212  
MIPS instruction frequency for, 236  
MIPS instructions, 211–213  
operands, 212  
overflow, 198  
packed format, 224  
precision, 230  
procedure with two-dimensional  
matrices, 215–217  
programs, compiling, 214–217  
registers, 217  
representation, 197–202  
rounding, 218–219  
sign and magnitude, 197  
SSE2 architecture, 224–225  
subtraction, 211  
underflow, 198  
units, 219  
in x86, 224

Floating vectors, OL3.11-3

Floating-point addition, 203–206  
arithmetic unit block diagram, 207  
binary, 204  
illustrated, 205  
instructions, 211, A-73–74  
steps, 203–204

Floating-point arithmetic (GPUs), C-41–46  
basic, C-42  
double precision, C-45–46, C-74  
performance, C-44  
specialized, C-42–44  
supported formats, C-42  
texture operations, C-44

Floating-point instructions, A-73–80  
absolute value, A-73  
addition, A-73–74  
comparison, A-74–75  
conversion, A-75–76  
desktop RISC, E-12  
division, A-76  
load, A-76–77  
move, A-77–78  
multiplication, A-78  
negation, A-78–79  
SPARC, E-31  
square root, A-79  
store, A-79  
subtraction, A-79–80  
truncation, A-80

Floating-point multiplication, 206–210  
binary, 210–211  
illustrated, 209  
instructions, 211  
significands, 206  
steps, 206–210

Flow-sensitive information, OL2.15-15

Flushing instructions, 318, 319  
defined, 319  
exceptions and, 331

For loops, 141, OL2.15-26  
inner, OL2.15-24  
SIMD and, OL6.15-2

Formal parameters, A-16

Format fields, D-31

Fortran, OL2.21-7

Forward references, A-11

Forwarding, 303–316  
ALU before, 309  
control, 307  
datapath for hazard resolution, 311  
defined, 278  
functioning, 306  
graphical representation, 279  
illustrations, OL4.13-26–4.13-26  
multiple results and, 281  
multiplexors, 310  
pipeline registers before, 309  
with two instructions, 278

Verilog implementation, OL4.13-2–4.13-4

Fractions, 197, 198

Frame buffer, 18

Frame pointers, 103

Front end, OL2.15-3

Fully associative caches. *See also* Caches  
block replacement strategies, 457  
choice of, 456  
defined, 403  
memory block location, 403  
misses, 406

Fully connected networks, 535

Function code, 82

Fused-multiply-add (FMA) operation, 220, C-45–46

**G**

Game consoles, C-9

Gates, B-3, B-8  
AND, B-12, D-7  
delays, B-46  
mapping ALU control function to,  
D-4–7  
NAND, B-8  
NOR, B-8, B-50

Gather-scatter, 511, 552

General Purpose GPUs (GPGPUs), C-5

General-purpose registers, 150  
architectures, OL2.21-3  
embedded RISCs, E-5

Generate  
defined, B-40  
example, B-44  
super, B-41

Gigabyte, 6

Global common subexpression elimination, OL2.15-6

Global memory, C-21, C-39

Global miss rates, 416

Global optimization, OL2.15-5  
code, OL2.15-7  
implementing, OL2.15-8–2.15-11

Global pointers, 102

GPU computing. *See also* Graphics  
processing units (GPUs)  
defined, C-5  
visual applications, C-6–7

GPU system architectures, C-7–12  
graphics logical pipeline, C-10  
heterogeneous, C-7–9  
implications for, C-24  
interfaces and drivers, C-9  
unified, C-10–12

Graph coloring, OL2.15-12

Graphics displays  
 computer hardware support, 18  
 LCD, 18

Graphics logical pipeline, C-10

Graphics processing units (GPUs), 522–529. *See also* GPU computing  
 as accelerators, 522  
 attribute interpolation, C-43–44  
 defined, 46, 506, C-3  
 evolution, C-5  
 fallacies and pitfalls, C-72–75  
 floating-point arithmetic, C-17, C-41–46, C-74  
 GeForce 8-series generation, C-5  
 general computation, C-73–74  
 General Purpose (GPGPUs), C-5  
 graphics mode, C-6  
 graphics trends, C-4  
 history, C-3–4  
 logical graphics pipeline, C-13–14  
 mapping applications to, C-55–72  
 memory, 523  
 multilevel caches and, 522  
 N-body applications, C-65–72  
 NVIDIA architecture, 523–526  
 parallel memory system, C-36–41  
 parallelism, 523, C-76  
 performance doubling, C-4  
 perspective, 527–529  
 programming, C-12–24  
 programming interfaces to, C-17  
 real-time graphics, C-13  
 summary, C-76

Graphics shader programs, C-14–15

Gresham's Law, 236, OL3.11-2

Grid computing, 533

Grids, C-19

GTX 280, 548–553

Guard digits  
 defined, 218  
 rounding with, 219

## H

Half precision, C-42

Halfwords, 110

Hamming, Richard, 420

Hamming distance, 420

Hamming Error Correction Code (ECC), 420–421  
 calculating, 420–421

Handlers  
 defined, 449  
 TLB miss, 448

Hard disks  
 access times, 23  
 defined, 23

Hardware  
 as hierarchical layer, 13  
 language of, 14–16  
 operations, 63–66  
 supporting procedures in, 96–106  
 synthesis, B-21  
 translating microprograms to, D-28–32  
 virtualizable, 426

Hardware description languages. *See also*  
 Verilog  
 defined, B-20  
 using, B-20–26  
 VHDL, B-20–21

Hardware multithreading, 514–517  
 coarse-grained, 514  
 options, 516  
 simultaneous, 515–517

Hardware-based speculation, 341

Harvard architecture, OL1.12-4

Hazard detection units, 313–314  
 functions, 314  
 pipeline connections for, 314

Hazards, 277–278. *See also* Pipelining  
 control, 281–282, 316–325  
 data, 278, 303–316  
 forwarding and, 312  
 structural, 277, 294

Heap  
 allocating space on, 104–106  
 defined, 104

Heterogeneous systems, C-4–5  
 architecture, C-7–9  
 defined, C-3

Hexadecimal numbers, 81–82  
 binary number conversion to, 81–82

Hierarchy of memories, 12

High-level languages, 14–16, A-6  
 benefits, 16  
 computer architectures, OL2.21–5  
 importance, 16

High-level optimizations, OL2.15-4–2.15–5

Hit rate, 376

Hit time  
 cache performance and, 401–402

defined, 376

Hit under miss, 472

Hold time, B-54

Horizontal microcode, D-32

Hot-swapping, OL5.11-7

Human genome project, 4

I

I/O, A-38–40, OL6.9-2, OL6.9-3  
 memory-mapped, A-38  
 on system performance, OL5.11–2

I/O benchmarks. *See* Benchmarks

IBM 360/85, OL5.17-7

IBM 701, OL1.12-5

IBM 7030, OL4.16-2

IBM ALOG, OL3.11-7

IBM Blue Gene, OL6.15-9–6.15-10

IBM Personal Computer, OL1.12-7, OL2.21-6

IBM System/360 computers, OL1.12-6, OL3.11-6, OL4.16-2

IBM z/VM, OL5.17-8

ID stage  
 branch execution in, 319  
 load instructions, 292  
 store instruction in, 291

IEEE 754 floating-point standard, 198, 199, OL3.11-8–3.11-10. *See also*  
 Floating point

first chips, OL3.11-8–3.11-9  
 in GPU arithmetic, C-42–43  
 implementation, OL3.11-10  
 rounding modes, 219  
 today, OL3.11-10

If statements, 114

I-format, 83

If-then-else, 91

Immediate addressing, 116

Immediate instructions, 72

Imprecise interrupts, 331, OL4.16–4

Index-out-of-bounds check, 94–95

Induction variable elimination, OL2.15-7

Inheritance, OL2.15-15

In-order commit, 341

Input devices, 16

Inputs, 261

Instances, OL2.15-15

Instruction count, 36, 38

Instruction decode/register file read stage

control line, 300  
load instruction, 289  
store instruction, 294  
Instruction execution illustrations, OL4.13–16–4.13–17  
clock cycle 9, OL4.13–24  
clock cycles 1 and 2, OL4.13–21  
clock cycles 3 and 4, OL4.13–22  
clock cycles 5 and 6, OL4.13–23, OL4.13–23  
clock cycles 7 and 8, OL4.13–24  
examples, OL4.13–20–4.13–25  
forwarding, OL4.13–26–4.13–31  
no hazard, OL4.13–17  
pipelines with stalls and forwarding, OL4.13–26, OL4.13–20  
Instruction fetch stage  
control line, 300  
load instruction, 289  
store instruction, 294  
Instruction formats, 157  
ARM, 148  
defined, 81  
desktop/server RISC architectures, E-7  
embedded RISC architectures, E-8  
I-type, 83  
J-type, 113  
jump instruction, 270  
MIPS, 148  
R-type, 83, 261  
x86, 157  
Instruction latency, 356  
Instruction mix, 39, OL1.12–10  
Instruction set architecture  
ARM, 145–147  
branch address calculation, 254  
defined, 22, 52  
history, 163  
maintaining, 52  
protection and, 427  
thread, C-31–34  
virtual machine support, 426–427  
Instruction sets, 235, C-49  
ARM, 324  
design for pipelining, 277  
MIPS, 62, 161, 234  
MIPS-32, 235  
Pseudo MIPS, 233  
x86 growth, 161  
Instruction-level parallelism (ILP), 354.  
*See also* Parallelism

compiler exploitation, OL4.16–5–4.16–6  
defined, 43, 333  
exploitation, increasing, 343  
and matrix multiply, 351–354  
Instructions, 60–164, E-25–27, E-40–42.  
*See also* Arithmetic instructions; MIPS; Operands  
add immediate, 72  
addition, 180, A-51  
Alpha, E-27–29  
arithmetic-logical, 251, A-51–57  
ARM, 145–147, E-36–37  
assembly, 66  
basic block, 93  
branch, A-59–63  
cache-aware, 482  
comparison, A-57–59  
conditional branch, 90  
conditional move, 324  
constant-manipulating, A-57  
conversion, A-75–76  
core, 233  
data movement, A-70–73  
data transfer, 68  
decision-making, 90–96  
defined, 14, 62  
desktop RISC conventions, E-12  
division, A-52–53  
as electronic signals, 80  
embedded RISC conventions, E-15  
encoding, 83  
exception and interrupt, A-80  
exclusive OR, A-57  
fetching, 253  
fields, 80  
floating-point (x86), 224  
floating-point, 211–213, A-73–80  
flushing, 318, 319, 331  
immediate, 72  
introduction to, 62–63  
jump, 95, 97, A-63–64  
left-to-right flow, 287–288  
load, 68, A-66–68  
load linked, 122  
logical operations, 87–89  
M32R, E-40  
memory access, C-33–34  
memory-reference, 245  
multiplication, 188, A-53–54  
negation, A-54  
nop, 314  
PA-RISC, E-34–36  
performance, 35–36  
pipeline sequence, 313  
PowerPC, E-12–13, E-32–34  
PTX, C-31, C-32  
remainder, A-55  
representation in computer, 80–87  
restartable, 450  
resuming, 450  
R-type, 252  
shift, A-55–56  
SPARC, E-29–32  
store, 71, A-68–70  
store conditional, 122  
subtraction, 180, A-56–57  
SuperH, E-39–40  
thread, C-30–31  
Thumb, E-38  
trap, A-64–66  
vector, 510  
as words, 62  
x86, 149–155  
Instructions per clock cycle (IPC), 333  
Integrated circuits (ICs), 19. *See also* specific chips  
cost, 27  
defined, 25  
manufacturing process, 26  
very large-scale (VLSIs), 25  
Intel Core i7, 46–49, 244, 501, 548–553  
address translation for, 471  
architectural registers, 347  
caches in, 472  
memory hierarchies of, 471–475  
microarchitecture, 338  
performance of, 473  
SPEC CPU benchmark, 46–48  
SPEC power benchmark, 48–49  
TLB hardware for, 471  
Intel Core i7 920, 346–349  
microarchitecture, 347  
Intel Core i7 960  
benchmarking and rooflines of, 548–553  
Intel Core i7 Pipelines, 344, 346–349  
memory components, 348  
performance, 349–351  
program performance, 351  
specification, 345  
Intel IA-64 architecture, OL2.21–3  
Intel Paragon, OL6.15–8

Intel Threading Building Blocks, C-60  
 Intel x86 microprocessors  
     clock rate and power for, 40  
 Interference graphs, OL2.15-12  
 Interleaving, 398  
 Interprocedural analysis, OL2.15-14  
 Interrupt enable, 447  
 Interrupt handlers, A-33  
 Interrupt-driven I/O, OL6.9-4  
 Interrupts  
     defined, 180, 326  
     event types and, 326  
     exceptions *versus*, 325–326  
     imprecise, 331, OL4.16-4  
     instructions, A-80  
     precise, 332  
     vectored, 327  
 Intrinsic FastMATH processor, 395–398  
     caches, 396  
     data miss rates, 397, 407  
     read processing, 442  
     TLB, 440  
     write-through processing, 442  
 Inverted page tables, 436  
 Issue packets, 334

**J**

j (Jump), 64  
 jal (Jump And Link), 64  
 Java  
     bytecode, 131  
     bytecode architecture, OL2.15-17  
     characters in, 109–111  
     compiling in, OL2.15-19–2.15-20  
     goals, 131  
     interpreting, 131, 145, OL2.15-15–2.15-16  
     keywords, OL2.15-21  
     method invocation in, OL2.15-21  
     pointers, OL2.15-26  
     primitive types, OL2.15-26  
     programs, starting, 131–132  
     reference types, OL2.15-26  
     sort algorithms, 141  
     strings in, 109–111  
     translation hierarchy, 131  
     while loop compilation in, OL2.15-18–2.15-19  
 Java Virtual Machine (JVM), 145, OL2.15-16

jr (Jump Register), 64  
 J-type instruction format, 113  
 Jump instructions, 254, E-26  
     branch instruction *versus*, 270  
     control and datapath for, 271  
     implementing, 270  
     instruction format, 270  
     list of, A-63–64  
 Just In Time (JIT) compilers, 132, 560

**K**

Karnaugh maps, B-18  
 Kernel mode, 444  
 Kernels  
     CUDA, C-19, C-24  
     defined, C-19  
 Kilobyte, 6

**L**

Labels  
     global, A-10, A-11  
     local, A-11  
 LAPACK, 230  
 Large-scale multiprocessors, OL6.15-7, OL6.15-9–6.15-10  
 Latches  
     D latch, B-51, B-52  
     defined, B-51  
 Latency  
     instruction, 356  
     memory, C-74–75  
     pipeline, 286  
     use, 336–337  
 lbu (Load Byte Unsigned), 64  
 Leaf procedures. *See also* Procedures  
     defined, 100  
     example, 109  
 Least recently used (LRU)  
     as block replacement strategy, 457  
     defined, 409  
     pages, 434  
 Least significant bits, B-32  
     defined, 74  
     SPARC, E-31  
 Left-to-right instruction flow, 287–288  
 Level-sensitive clocking, B-74, B-75–76  
     defined, B-74  
     two-phase, B-75  
 lhu (Load Halfword Unsigned), 64  
 li (Load Immediate), 162  
 Link, OL6.9-2  
 Linkers, 126–129, A-18–19  
     defined, 126, A-4  
     executable files, 126, A-19  
     function illustration, A-19  
     steps, 126  
     using, 126–129  
 Linking object files, 126–129  
 Linpack, 538, OL3.11-4  
 Liquid crystal displays (LCDs), 18  
 LISP, SPARC support, E-30  
 Little-endian byte order, A-43  
 Live range, OL2.15-11  
 Livermore Loops, OL1.12-11  
 ll (Load Linked), 64  
 Load balancing, 505–506  
 Load instructions. *See also* Store  
     instructions  
     access, C-41  
     base register, 262  
     block, 149  
     compiling with, 71  
     datapath in operation for, 267  
     defined, 68  
     details, A-66–68  
     EX stage, 292  
     floating-point, A-76–77  
     halfword unsigned, 110  
     ID stage, 291  
     IF stage, 291  
     linked, 122, 123  
     list of, A-66–68  
     load byte unsigned, 76  
     load half, 110  
     load upper immediate, 112, 113  
     MEM stage, 293  
     pipelined datapath in, 296  
     signed, 76  
     unit for implementing, 255  
     unsigned, 76  
     WB stage, 293  
 Load word, 68, 71  
 Loaders, 129  
 Loading, A-19–20  
 Load-store architectures, OL2.21-3  
 Load-use data hazard, 280, 318  
 Load-use stalls, 318  
 Local area networks (LANs), 24. *See also* Networks

- Local labels, A-11  
 Local memory, C-21, C-40  
 Local miss rates, 416  
 Local optimization, OL2.15-5.  
*See also* Optimization  
 implementing, OL2.15-8  
**Locality**  
 principle, 374  
 spatial, 374, 377  
 temporal, 374, 377  
 Lock synchronization, 121  
 Locks, 518  
**Logic**  
 address select, D-24, D-25  
 ALU control, D-6  
 combinational, 250, B-5, B-9–20  
 components, 249  
 control unit equations, D-11  
 design, 248–251, B-1–79  
 equations, B-7  
 minimization, B-18  
 programmable array (PAL),  
     B-78  
 sequential, B-5, B-56–58  
 two-level, B-11–14  
**Logical operations**, 87–89  
 AND, 88, A-52  
 ARM, 149  
 desktop RISC, E-11  
 embedded RISC, E-14  
 MIPS, A-51–57  
 NOR, 89, A-54  
 NOT, 89, A-55  
 OR, 89, A-55  
 shifts, 87  
**Long instruction word (LIW)**,  
     OL4.16-5  
**Lookup tables (LUTs)**, B-79  
**Loop unrolling**  
 defined, 338, OL2.15-4  
 for multiple-issue pipelines, 338  
 register renaming and, 338  
**Loops**, 92–93  
 conditional branches in, 114  
 for, 141  
 prediction and, 321–323  
 test, 142, 143  
 while, compiling, 92–93  
**lui (Load Upper Imm.)**, 64  
**lw (Load Word)**, 64  
**lwc1 (Load FP Single)**, A-73
- M**
- M32R, E-15, E-40  
 Machine code, 81  
 Machine instructions, 81  
 Machine language, 15  
 branch offset in, 115  
 decoding, 118–120  
 defined, 14, 81, A-3  
 floating-point, 212  
 illustrated, 15  
 MIPS, 85  
 SRAM, 21  
 translating MIPS assembly language  
     into, 84  
**Macros**  
 defined, A-4  
 example, A-15–17  
 use of, A-15  
**Main memory**, 428. *See also* Memory  
 defined, 23  
 page tables, 437  
 physical addresses, 428  
**Mapping applications**, C-55–72  
**Mark computers**, OL1.12–14  
**Matrix multiply**, 225–228, 553–555  
**Mealy machine**, 463–464, B-68, B-71,  
     B-72  
**Mean time to failure (MTTF)**, 418  
 improving, 419  
*versus* AFR of disks, 419–420  
**Media Access Control (MAC) address**,  
     OL6.9-7  
**Megabyte**, 6  
**Memory**  
 addresses, 77  
 affinity, 545  
 atomic, C-21  
 bandwidth, 380–381, 397  
 cache, 21, 383–398, 398–417  
 CAM, 408  
 constant, C-40  
 control, D-26  
 defined, 19  
 DRAM, 19, 379–380, B-63–65  
 flash, 23  
 global, C-21, C-39  
 GPU, 523  
 instructions, datapath for, 256  
 layout, A-21  
 local, C-21, C-40  
 main, 23  
 nonvolatile, 22  
 operands, 68–69  
 parallel system, C-36–41  
 read-only (ROM), B-14–16  
 SDRAM, 379–380  
 secondary, 23  
 shared, C-21, C-39–40  
 spaces, C-39  
 SRAM, B-58–62  
 stalls, 400  
 technologies for building, 24–28  
 texture, C-40  
 usage, A-20–22  
 virtual, 427–454  
 volatile, 22  
**Memory access instructions**, C-33–34  
**Memory access stage**  
 control line, 302  
 load instruction, 292  
 store instruction, 292  
**Memory bandwidth**, 551, 557  
**Memory consistency model**, 469  
**Memory elements**, B-50–58  
 clocked, B-51  
 D flip-flop, B-51, B-53  
 D latch, B-52  
 DRAMs, B-63–67  
 flip-flop, B-51  
 hold time, B-54  
 latch, B-51  
 setup time, B-53, B-54  
 SRAMs, B-58–62  
 unclocked, B-51  
**Memory hierarchies**, 545  
 of ARM cortex-A8, 471–475  
 block (or line), 376  
 cache performance, 398–417  
 caches, 383–417  
 common framework, 454–461  
 defined, 375  
 design challenges, 461  
 development, OL5.17-6–5.17-8  
 exploiting, 372–498  
 of Intel core i7, 471–475  
 level pairs, 376  
 multiple levels, 375  
 overall operation of, 443–444  
 parallelism and, 466–470, OL5.11-2  
 pitfalls, 478–482  
 program execution time and, 417

- Memory hierarchies (*Continued*)  
 quantitative design parameters, 454  
 redundant arrays and inexpensive disks, 470  
 reliance on, 376  
 structure, 375  
 structure diagram, 378  
 variance, 417  
 virtual memory, 427–454
- Memory rank, 381
- Memory technologies, 378–383  
 disk memory, 381–383  
 DRAM technology, 378, 379–381  
 flash memory, 381  
 SRAM technology, 378, 379
- Memory-mapped I/O, OL6.9-3  
 use of, A-38
- Memory-stall clock cycles, 399
- Message passing  
 defined, 529  
 multiprocessors, 529–534
- Metastability, B-76
- Methods  
 defined, OL2.15-5  
 invoking in Java, OL2.15-20–2.15-21  
 static, A-20
- mfc0 (Move From Control), A-71
- mfhi (Move From Hi), A-71
- mflo (Move From Lo), A-71
- Microarchitectures, 347  
 Intel Core i7 920, 347
- Microcode  
 assembler, D-30  
 control unit as, D-28  
 defined, D-27  
 dispatch ROMs, D-30–31  
 horizontal, D-32  
 vertical, D-32
- Microinstructions, D-31
- Microprocessors  
 design shift, 501  
 multicore, 8, 43, 500–501
- Microprograms  
 as abstract control representation, D-30  
 field translation, D-29  
 translating to hardware, D-28–32
- Migration, 467
- Million instructions per second (MIPS), 51
- Minterms
- defined, B-12, D-20  
 in PLA implementation, D-20
- MIP-map, C-44
- MIPS, 64, 84, A-45–80  
 addressing for 32-bit immediates, 116–118  
 addressing modes, A-45–47  
 arithmetic core, 233  
 arithmetic instructions, 63, A-51–57  
 ARM similarities, 146  
 assembler directive support, A-47–49  
 assembler syntax, A-47–49  
 assembly instruction, mapping, 80–81  
 branch instructions, A-59–63  
 comparison instructions, A-57–59  
 compiling C assignment statements into, 65  
 compiling complex C assignment into, 65–66  
 constant-manipulating instructions, A-57  
 control registers, 448  
 control unit, D-10  
 CPU, A-46  
 divide in, 194  
 exceptions in, 326–327  
 fields, 82–83  
 floating-point instructions, 211–213  
 FPU, A-46  
 instruction classes, 163  
 instruction encoding, 83, 119, A-49  
 instruction formats, 120, 148, A-49–51  
 instruction set, 62, 162, 234  
 jump instructions, A-63–66  
 logical instructions, A-51–57  
 machine language, 85  
 memory addresses, 70  
 memory allocation for program and data, 104  
 multiply in, 188  
 opcode map, A-50  
 operands, 64  
 Pseudo, 233, 235  
 register conventions, 105  
 static multiple issue with, 335–338
- MIPS core  
 architecture, 195  
 arithmetic/logical instructions not in, E-21, E-23  
 common extensions to, E-20–25  
 control instructions not in, E-21
- data transfer instructions not in, E-20, E-22  
 floating-point instructions not in, E-22  
 instruction set, 233, 244–248, E-9–10
- MIPS-16  
 16-bit instruction set, E-41–42  
 immediate fields, E-41  
 instructions, E-40–42  
 MIPS core instruction changes, E-42  
 PC-relative addressing, E-41  
 MIPS-32 instruction set, 235  
 MIPS-64 instructions, E-25–27  
 conditional procedure call instructions, E-27  
 constant shift amount, E-25  
 jump/call not PC-relative, E-26  
 move to/from control registers, E-26  
 nonaligned data transfers, E-25  
 NOR, E-25  
 parallel single precision floating-point operations, E-27  
 reciprocal and reciprocal square root, E-27  
 SYSCALL, E-25  
 TLB instructions, E-26–27
- Mirroring, OL5.11-5
- Miss penalty  
 defined, 376  
 determination, 391–392  
 multilevel caches, reducing, 410
- Miss rates  
 block size *versus*, 392  
 data cache, 455  
 defined, 376  
 global, 416  
 improvement, 391–392  
 Intrinsic FastMATH processor, 397  
 local, 416  
 miss sources, 460  
 split cache, 397
- Miss under miss, 472
- MMX (MultiMedia eXtension), 224
- Modules, A-4
- Moore machines, 463–464, B-68, B-71, B-72
- Moore's law, 11, 379, 522, OL6.9-2, C-72–73
- Most significant bit  
 1-bit ALU for, B-33  
 defined, 74
- move (Move), 139

- Move instructions, A-70–73  
  coprocessor, A-71–72  
  details, A-70–73  
  floating-point, A-77–78  
MS-DOS, OL5.17–11  
mul.d (FP Multiply Double), A-78  
mul.s (FP Multiply Single), A-78  
mult (Multiply), A-53  
Multicore, 517–521  
Multicore multiprocessors, 8, 43  
  defined, 8, 500–501  
MULTICS (Multiplexed Information and Computing Service), OL5.17–9–5.17–10  
Multilevel caches. *See also* Caches  
  complications, 416  
  defined, 398, 416  
  miss penalty, reducing, 410  
  performance of, 410  
  summary, 417–418  
Multimedia extensions  
  desktop/server RISCs, E-16–18  
  as SIMD extensions to instruction sets, OL6.15–4  
  vector *versus*, 511–512  
Multiple dimension arrays, 218  
Multiple instruction multiple data (MIMD), 558  
  defined, 507, 508  
  first multiprocessor, OL6.15–14  
Multiple instruction single data (MISD), 507  
Multiple issue, 332–339  
  code scheduling, 337–338  
  dynamic, 333, 339–341  
  issue packets, 334  
  loop unrolling and, 338  
  processors, 332, 333  
  static, 333, 334–339  
  throughput and, 342  
Multiple processors, 553–555  
Multiple-clock-cycle pipeline diagrams, 296–297  
  five instructions, 298  
  illustrated, 298  
Multiplexors, B-10  
  controls, 463  
  in datapath, 263  
  defined, 246  
  forwarding, control values, 310  
  selector control, 256–257  
  two-input, B-10  
Multiplicand, 183  
Multiplication, 183–188. *See also* Arithmetic  
  fast, hardware, 188  
  faster, 187–188  
  first algorithm, 185  
  floating-point, 206–208, A-78  
  hardware, 184–186  
  instructions, 188, A-53–54  
  in MIPS, 188  
  multiplicand, 183  
  multiplier, 183  
  operands, 183  
  product, 183  
  sequential version, 184–186  
  signed, 187  
Multiplier, 183  
Multiply algorithm, 186  
Multiply-add (MAD), C-42  
Multiprocessors  
  benchmarks, 538–540  
  bus-based coherent, OL6.15–7  
  defined, 500  
  historical perspective, 561  
  large-scale, OL6.15–7–6.15–8, OL6.15–9–6.15–10  
  message-passing, 529–534  
  multithreaded architecture, C-26–27, C-35–36  
  organization, 499, 529  
  for performance, 559  
  shared memory, 501, 517–521  
  software, 500  
  TFLOPS, OL6.15–6  
  UMA, 518  
Multistage networks, 535  
Multithreaded multiprocessor  
  architecture, C-25–36  
  conclusion, C-36  
  ISA, C-31–34  
  massive multithreading, C-25–26  
  multiprocessor, C-26–27  
  multiprocessor comparison, C-35–36  
  SIMT, C-27–30  
  special function units (SFUs), C-35  
  streaming processor (SP), C-34  
  thread instructions, C-30–31  
  threads/thread blocks management, C-30  
Multithreading, C-25–26  
  coarse-grained, 514  
  defined, 506  
  fine-grained, 514  
  hardware, 514–517  
  simultaneous (SMT), 515–517  
multu (Multiply Unsigned), A-54  
Must-information, OL2.15–5  
Mutual exclusion, 121
- N**
- Name dependence, 338  
NAND gates, B-8  
NAS (NASA Advanced Supercomputing), 540  
N-body  
  all-pairs algorithm, C-65  
  GPU simulation, C-71  
  mathematics, C-65–67  
  multiple threads per body, C-68–69  
  optimization, C-67  
  performance comparison, C-69–70  
  results, C-70–72  
  shared memory use, C-67–68  
Negation instructions, A-54, A-78–79  
Negation shortcut, 76  
Nested procedures, 100–102  
  compiling recursive procedure  
    showing, 101–102  
NetFPGA 10-Gigabit Ethernet card, OL6.9–2, OL6.9–3  
Network of Workstations, OL6.15–8–6.15–9  
Network topologies, 534–537  
  implementing, 536  
  multistage, 537  
Networking, OL6.9–4  
  operating system in, OL6.9–4–6.9–5  
  performance improvement, OL6.9–7–6.9–10  
Networks, 23–24  
  advantages, 23  
  bandwidth, 535  
  crossbar, 535  
  fully connected, 535  
  local area (LANs), 24  
  multistage, 535  
  wide area (WANs), 24  
Newton's iteration, 218  
Next state  
  nonsequential, D-24  
  sequential, D-23

Next-state function, 463, B-67  
 defined, 463  
 implementing, with sequencer, D-22–28  
 Next-state outputs, D-10, D-12–13  
 example, D-12–13  
 implementation, D-12  
 logic equations, D-12–13  
 truth tables, D-15  
 No Redundancy (RAID 0), OL5.11–4  
 No write allocation, 394  
 Nonblocking assignment, B-24  
 Nonblocking caches, 344, 472  
 Nonuniform memory access (NUMA), 518  
 Nonvolatile memory, 22  
 Nops, 314  
 nor (NOR), 64  
 NOR gates, B-8  
 cross-coupled, B-50  
 D latch implemented with, B-52  
 NOR operation, 89, A-54, E-25  
 NOT operation, 89, A-55, B-6  
 Numbers  
 binary, 73  
 computer *versus* real-world, 221  
 decimal, 73, 76  
 denormalized, 222  
 hexadecimal, 81–82  
 signed, 73–78  
 unsigned, 73–78  
 NVIDIA GeForce 8800, C-46–55  
 all-pairs N-body algorithm, C-71  
 dense linear algebra computations, C-51–53  
 FFT performance, C-53  
 instruction set, C-49  
 performance, C-51  
 rasterization, C-50  
 ROP, C-50–51  
 scalability, C-51  
 sorting performance, C-54–55  
 special function approximation statistics, C-43  
 special function unit (SFU), C-50  
 streaming multiprocessor (SM), C-48–49  
 streaming processor, C-49–50  
 streaming processor array (SPA), C-46  
 texture/processor cluster (TPC), C-47–48

NVIDIA GPU architecture, 523–526  
 NVIDIA GTX 280, 548–553  
 NVIDIA Tesla GPU, 548–553

## O

Object files, 125, A-4  
 debugging information, 124  
 defined, A-10  
 format, A-13–14  
 header, 125, A-13  
 linking, 126–129  
 relocation information, 125  
 static data segment, 125  
 symbol table, 125, 126  
 text segment, 125  
 Object-oriented languages. *See also* Java  
 brief history, OL2.21–8  
 defined, 145, OL2.15–5  
 One's complement, 79, B-29  
 Opcodes  
 control line setting and, 264  
 defined, 82, 262  
 OpenGL, C-13  
 OpenMP (Open MultiProcessing), 520, 540

Operands, 66–73. *See also* Instructions

32-bit immediate, 112–113  
 adding, 179  
 arithmetic instructions, 66  
 compiling assignment when in memory, 69  
 constant, 72–73  
 division, 189  
 floating-point, 212  
 memory, 68–69  
 MIPS, 64  
 multiplication, 183  
 shifting, 148

Operating systems  
 brief history, OL5.17–9–5.17–12  
 defined, 13  
 encapsulation, 22  
 in networking, OL6.9–4–6.9–5

Operations  
 atomic, implementing, 121  
 hardware, 63–66  
 logical, 87–89  
 x86 integer, 152, 154–155  
 Optimization  
 class explanation, OL2.15–14

compiler, 141  
 control implementation, D-27–28  
 global, OL2.15–5  
 high-level, OL2.15–4–2.15–5  
 local, OL2.15–5, OL2.15–8  
 manual, 144

or (OR), 64  
 OR operation, 89, A-55, B-6

ori (Or Immediate), 64

Out-of-order execution

defined, 341  
 performance complexity, 416  
 processors, 344

Output devices, 16

Overflow

defined, 74, 198  
 detection, 180  
 exceptions, 329  
 floating-point, 198  
 occurrence, 75  
 saturation and, 181  
 subtraction, 179

## P

P+Q redundancy (RAID 6), OL5.11–7

Packed floating-point format, 224

Page faults, 434. *See also* Virtual memory

for data access, 450  
 defined, 428  
 handling, 429, 446–453  
 virtual address causing, 449, 450

Page tables, 456

defined, 432  
 illustrated, 435  
 indexing, 432  
 inverted, 436  
 levels, 436–437  
 main memory, 437  
 register, 432  
 storage reduction techniques, 436–437  
 updating, 432  
 VMM, 452

Pages. *See also* Virtual memory

defined, 428  
 dirty, 437  
 finding, 432–434  
 LRU, 434  
 offset, 429  
 physical number, 429  
 placing, 432–434

- size, 430  
virtual number, 429
- Parallel bus, OL6.9-3
- Parallel execution, 121
- Parallel memory system, C-36–41. *See also* Graphics processing units (GPUs)  
caches, C-38  
constant memory, C-40  
DRAM considerations, C-37–38  
global memory, C-39  
load/store access, C-41  
local memory, C-40  
memory spaces, C-39  
MMU, C-38–39  
ROP, C-41  
shared memory, C-39–40  
surfaces, C-41  
texture memory, C-40
- Parallel processing programs, 502–507  
creation difficulty, 502–507  
defined, 501  
for message passing, 519–520  
great debates in, OL6.15–5  
for shared address space, 519–520  
use of, 559
- Parallel reduction, C-62
- Parallel scan, C-60–63  
CUDA template, C-61  
inclusive, C-60  
tree-based, C-62
- Parallel software, 501
- Parallelism, 12, 43, 332–344  
and computers arithmetic, 222–223  
data-level, 233, 508  
debates, OL6.15–5–6.15–7  
GPUs and, 523, C-76  
instruction-level, 43, 332, 343  
memory hierarchies and, 466–470, OL5.11–2  
multicore and, 517  
multiple issue, 332–339  
multithreading and, 517  
performance benefits, 44–45  
process-level, 500  
redundant arrays and inexpensive disks, 470  
subword, E-17  
task, C-24  
task-level, 500  
thread, C-22
- Paravirtualization, 482
- PA-RISC, E-14, E-17  
branch vectored, E-35  
conditional branches, E-34, E-35  
debug instructions, E-36  
decimal operations, E-35  
extract and deposit, E-35  
instructions, E-34–36  
load and clear instructions, E-36  
multiply/add and multiply/subtract, E-36  
nullification, E-34  
nullifying branch option, E-25  
store bytes short, E-36  
synthesized multiply and divide, E-34–35
- Parity, OL5.11–5  
bits, 421  
code, 420, B-65
- PARSEC (Princeton Application Repository for Shared Memory Computers), 540
- Pass transistor, B-63
- PCI-Express (PCIe), 537, C-8, OL6.9–2
- PC-relative addressing, 114, 116
- Peak floating-point performance, 542
- Pentium bug morality play, 231–232
- Performance, 28–36  
assessing, 28  
classic CPU equation, 36–40  
components, 38  
CPU, 33–35  
defining, 29–32  
equation, using, 36  
improving, 34–35  
instruction, 35–36  
measuring, 33–35, OL1.12–10  
program, 39–40  
ratio, 31  
relative, 31–32  
response time, 30–31  
sorting, C-54–55  
throughput, 30–31  
time measurement, 32
- Personal computers (PCs), 7  
defined, 5
- Personal mobile device (PMD)  
defined, 7
- Petabyte, 6
- Physical addresses, 428  
mapping to, 428–429
- space, 517, 521
- Physically addressed caches, 443
- Pipeline registers  
before forwarding, 309  
dependences, 308  
forwarding unit selection, 312
- Pipeline stalls, 280  
avoiding with code reordering, 280  
data hazards and, 313–316  
insertion, 315  
load-use, 318  
as solution to control hazards, 282
- Pipelined branches, 319
- Pipelined control, 300–303. *See also* Control  
control lines, 300, 303  
overview illustration, 316  
specifying, 300
- Pipelined datapaths, 286–303  
with connected control signals, 304  
with control signals, 300–303  
corrected, 296  
illustrated, 289  
in load instruction stages, 296
- Pipelined dependencies, 305
- Pipelines  
branch instruction impact, 317  
effectiveness, improving, OL4.16–4–4.16–5  
execute and address calculation stage, 290, 292  
five-stage, 274, 290, 299  
graphic representation, 279, 296–300  
instruction decode and register file  
read stage, 289, 292  
instruction fetch stage, 290, 292  
instructions sequence, 313  
latency, 286  
memory access stage, 290, 292  
multiple-clock-cycle diagrams, 296–297  
performance bottlenecks, 343  
single-clock-cycle diagrams, 296–297  
stages, 274  
static two-issue, 335  
write-back stage, 290, 294
- Pipelining, 12, 272–286  
advanced, 343–344  
benefits, 272  
control hazards, 281–282  
data hazards, 278

- Pipelining (*Continued*)  
 exceptions and, 327–332  
 execution time and, 286  
 fallacies, 355–356  
 hazards, 277–278  
 instruction set design for, 277  
 laundry analogy, 273  
 overview, 272–286  
 paradox, 273  
 performance improvement, 277  
 pitfall, 355–356  
 simultaneous executing instructions, 286  
 speed-up formula, 273  
 structural hazards, 277, 294  
 summary, 285  
 throughput and, 286
- Pitfalls. *See also* Fallacies  
 address space extension, 479  
 arithmetic, 229–232  
 associativity, 479  
 defined, 49  
 GPUs, C-74–75  
 ignoring memory system behavior, 478  
 memory hierarchies, 478–482  
 out-of-order processor evaluation, 479  
 performance equation subset, 50–51  
 pipelining, 355–356  
 pointer to automatic variables, 160  
 sequential word addresses, 160  
 simulating cache, 478  
 software development with multiprocessors, 556  
 VMM implementation, 481, 481–482
- Pixel shader example, C-15–17
- Pixels, 18
- Pointers  
 arrays *versus*, 141–145  
 frame, 103  
 global, 102  
 incrementing, 143  
 Java, OL2.15–26  
 stack, 98, 102
- Polling, OL6.9–8
- Pop, 98
- Power  
 clock rate and, 40  
 critical nature of, 53  
 efficiency, 343–344  
 relative, 41
- PowerPC  
 algebraic right shift, E-33
- branch registers, E-32–33  
 condition codes, E-12  
 instructions, E-12–13  
 instructions unique to, E-31–33  
 load multiple/store multiple, E-33  
 logical shifted immediate, E-33  
 rotate with mask, E-33
- Precise interrupts, 332
- Prediction, 12  
 2-bit scheme, 322  
 accuracy, 321, 324  
 dynamic branch, 321–323  
 loops and, 321–323  
 steady-state, 321
- Prefetching, 482, 544
- Primitive types, OL2.15–26
- Procedure calls  
 convention, A-22–33  
 examples, A-27–33  
 frame, A-23  
 preservation across, 102
- Procedures, 96–106  
 compiling, 98  
 compiling, showing nested procedure linking, 101–102  
 execution steps, 96  
 frames, 103  
 leaf, 100  
 nested, 100–102  
 recursive, 105, A-26–27  
 for setting arrays to zero, 142  
 sort, 135–139  
 strcpy, 108–109  
 string copy, 108–109  
 swap, 133
- Process identifiers, 446
- Process-level parallelism, 500
- Processors, 242–356  
 as cores, 43  
 control, 19  
 datapath, 19  
 defined, 17, 19  
 dynamic multiple-issue, 333  
 multiple-issue, 333  
 out-of-order execution, 344, 416  
 performance growth, 44  
 ROP, C-12, C-41  
 speculation, 333–334  
 static multiple-issue, 333, 334–339  
 streaming, C-34  
 superscalar, 339, 515–516, OL4.16–5  
 technologies for building, 24–28
- two-issue, 336–337  
 vector, 508–510  
 VLIW, 335
- Product, 183
- Product of sums, B-11
- Program counters (PCs), 251  
 changing with conditional branch, 324  
 defined, 98, 251  
 exception, 445, 447  
 incrementing, 251, 253  
 instruction updates, 289
- Program libraries, A-4
- Program performance  
 elements affecting, 39  
 understanding, 9
- Programmable array logic (PAL), B-78
- Programmable logic arrays (PLAs)  
 component dots illustration, B-16  
 control function implementation, D-7, D-20–21  
 defined, B-12  
 example, B-13–14  
 illustrated, B-13  
 ROMs and, B-15–16  
 size, D-20  
 truth table implementation, B-13
- Programmable logic devices (PLDs), B-78
- Programmable ROMs (PROMs), B-14
- Programming languages. *See also* specific languages  
 brief history of, OL2.21–7–2.21–8  
 object-oriented, 145  
 variables, 67
- Programs  
 assembly language, 123  
 Java, starting, 131–132  
 parallel processing, 502–507  
 starting, 123–132  
 translating, 123–132
- Propagate  
 defined, B-40  
 example, B-44  
 super, B-41
- Protected keywords, OL2.15–21
- Protection  
 defined, 428  
 implementing, 444–446  
 mechanisms, OL5.17–9  
 VMs for, 424
- Protection group, OL5.11–5
- Pseudo MIPS  
 defined, 233

instruction set, 235  
**Pseudodirect addressing**, 116  
**Pseudoinstructions**  
  defined, 124  
  summary, 125  
**Pthreads** (POSIX threads), 540  
**PTX instructions**, C-31, C-32  
**Public keywords**, OL2.15-21  
**Push**  
  defined, 98  
  using, 100

## Q

**Quad words**, 154  
**Quicksort**, 411, 412  
**Quotient**, 189

## R

**Race**, B-73  
**Radix sort**, 411, 412, C-63-65  
  CUDA code, C-64  
  implementation, C-63-65  
**RAID**, *See* **Redundant arrays of inexpensive disks (RAID)**  
**RAM**, 9  
**Raster operation (ROP) processors**, C-12, C-41, C-50-51  
  fixed function, C-41  
**Raster refresh buffer**, 18  
**Rasterization**, C-50  
**Ray casting (RC)**, 552  
**Read-only memories (ROMs)**, B-14-16  
  control entries, D-16-17  
  control function encoding, D-18-19  
  dispatch, D-25  
  implementation, D-15-19  
  logic function encoding, B-15  
  overhead, D-18  
  PLAs and, B-15-16  
  programmable (PROM), B-14  
  total size, D-16  
**Read-stall cycles**, 399  
**Read-write head**, 381  
**Receive message routine**, 529  
**Receiver Control register**, A-39  
**Receiver Data register**, A-38, A-39  
**Recursive procedures**, 105, A-26-27. *See also* **Procedures**  
  clone invocation, 100  
  stack in, A-29-30

**Reduced instruction set computer (RISC)**  
  architectures, E-2-45, OL2.21-5,  
  OL4.16-4. *See also* **Desktop and server RISCs; Embedded RISCs**  
  group types, E-3-4  
  instruction set lineage, E-44  
**Reduction**, 519  
**Redundant arrays of inexpensive disks (RAID)**, OL5.11-2-5.11-8  
  history, OL5.11-8  
  RAID 0, OL5.11-4  
  RAID 1, OL5.11-5  
  RAID 2, OL5.11-5  
  RAID 3, OL5.11-5  
  RAID 4, OL5.11-5-5.11-6  
  RAID 5, OL5.11-6-5.11-7  
  RAID 6, OL5.11-7  
  spread of, OL5.11-6  
  summary, OL5.11-7-5.11-8  
  use statistics, OL5.11-7  
**Reference bit**, 435  
**References**  
  absolute, 126  
  forward, A-11  
  types, OL2.15-26  
  unresolved, A-4, A-18  
**Register addressing**, 116  
**Register allocation**, OL2.15-11-2.15-13  
**Register files**, B-50, B-54-56  
  defined, 252, B-50, B-54  
  in behavioral Verilog, B-57  
  single, 257  
  two read ports implementation, B-55  
  with two read ports/one write port,  
    B-55  
  write port implementation, B-56  
**Register-memory architecture**, OL2.21-3  
**Registers**, 152, 153-154  
  architectural, 325-332  
  base, 69  
  callee-saved, A-23  
  caller-saved, A-23  
**Cause**, A-35  
  clock cycle time and, 67  
  compiling C assignment with, 67-68  
**Count**, A-34  
  defined, 66  
  destination, 83, 262  
  floating-point, 217  
  left half, 290  
  mapping, 80  
  MIPS conventions, 105  
  number specification, 252  
  page table, 432  
  pipeline, 308, 309, 312  
  primitives, 66  
  Receiver Control, A-39  
  Receiver Data, A-38, A-39  
  renaming, 338  
  right half, 290  
  spilling, 71  
  Status, 327, A-35  
  temporary, 67, 99  
  Transmitter Control, A-39-40  
  Transmitter Data, A-40  
  usage convention, A-24  
  use convention, A-22  
  variables, 67  
**Relative performance**, 31-32  
**Relative power**, 41  
**Reliability**, 418  
**Relocation information**, A-13, A-14  
**Remainder**  
  defined, 189  
  instructions, A-55  
**Reorder buffers**, 343  
**Replication**, 468  
**Requested word first**, 392  
**Request-level parallelism**, 532  
**Reservation stations**  
  buffering operands in, 340-341  
  defined, 339-340  
**Response time**, 30-31  
**Restartable instructions**, 448  
**Return address**, 97  
**Return from exception (ERET)**, 445  
**R-format**, 262  
  ALU operations, 253  
  defined, 83  
**Ripple carry**  
  adder, B-29  
  carry lookahead speed *versus*, B-46  
**Roofline model**, 542-543, 544, 545  
  with ceilings, 546, 547  
  computational roofline, 545  
  illustrated, 542  
  Opteron generations, 543, 544  
  with overlapping areas shaded, 547  
  peak floating-point performance, 542  
  peak memory performance, 543  
  with two kernels, 547  
**Rotational delay**. *See* **Rotational latency**  
**Rotational latency**, 383

Rounding, 218  
 accurate, 218  
 bits, 220  
 with guard digits, 219  
 IEEE 754 modes, 219  
 Row-major order, 217, 413  
 R-type instructions, 252  
 datapath for, 264–265  
 datapath in operation for, 266

**S**

Saturation, 181  
 sb (Store Byte), 64  
 sc (Store Conditional), 64  
 SCALAPAK, 230  
 Scaling  
   strong, 505, 507  
   weak, 505  
 Scientific notation  
   adding numbers in, 203  
   defined, 196  
   for reals, 197  
 Search engines, 4  
 Secondary memory, 23  
 Sectors, 381  
 Seek, 382  
 Segmentation, 431  
 Selector values, B-10  
 Semiconductors, 25  
 Send message routine, 529  
 Sensitivity list, B-24  
 Sequencers  
   explicit, D-32  
   implementing next-state function with,  
     D-22–28  
 Sequential logic, B-5  
 Servers, OL5. *See also* Desktop and server  
   RISCs  
     cost and capability, 5  
 Service accomplishment, 418  
 Service interruption, 418  
 Set instructions, 93  
 Set-associative caches, 403. *See also*  
   Caches  
     address portions, 407  
     block replacement strategies, 457  
     choice of, 456  
     four-way, 404, 407  
     memory-block location, 403  
     misses, 405–406

*n*-way, 403  
   two-way, 404  
 Setup time, B-53, B-54  
 sh (Store Halfword), 64  
 Shaders  
   defined, C-14  
   floating-point arithmetic, C-14  
   graphics, C-14–15  
   pixel example, C-15–17  
 Shading languages, C-14  
 Shadowing, OL5.11–5  
 Shared memory. *See also* Memory  
   as low-latency memory, C-21  
   caching in, C-58–60  
   CUDA, C-58  
   N-body and, C-67–68  
   per-CTA, C-39  
   SRAM banks, C-40  
 Shared memory multiprocessors (SMP),  
   517–521  
   defined, 501, 517  
   single physical address space, 517  
   synchronization, 518  
 Shift amount, 82  
 Shift instructions, 87, A-55–56  
 Sign and magnitude, 197  
 Sign bit, 76  
 Sign extension, 254  
   defined, 76  
   shortcut, 78  
 Signals  
   asserted, 250, B-4  
   control, 250, 263–264  
   deasserted, 250, B-4  
 Signed division, 192–194  
 Signed multiplication, 187  
 Signed numbers, 73–78  
   sign and magnitude, 75  
   treating as unsigned, 94–95  
 Significands, 198  
   addition, 203  
   multiplication, 206  
 Silicon, 25  
   as key hardware technology, 53  
   crystal ingot, 26  
   defined, 26  
   wafers, 26  
 Silicon crystal ingot, 26  
 SIMD (Single Instruction Multiple Data),  
   507–508, 558  
   computers, OL6.15–2–6.15–4  
 data vector, C-35  
 extensions, OL6.15–4  
 for loops and, OL6.15–3  
 massively parallel multiprocessors,  
   OL6.15–2  
 small-scale, OL6.15–4  
 vector architecture, 508–510  
 in x86, 508  
 SIMMs (single inline memory modules),  
   OL5.17–5, OL5.17–6  
 Simple programmable logic devices  
   (SPLDs), B-78  
 Simplicity, 161  
 Simultaneous multithreading (SMT),  
   515–517  
   support, 515  
   thread-level parallelism, 517  
   unused issue slots, 515  
 Single error correcting/Double error  
   correcting (SEC/DEC), 420–422  
 Single instruction single data (SISD), 507  
 Single precision. *See also* Double  
   precision  
   binary representation, 201  
   defined, 198  
 Single-clock-cycle pipeline diagrams,  
   296–297  
   illustrated, 299  
 Single-cycle datapaths. *See also* Datapaths  
   illustrated, 287  
   instruction execution, 288  
 Single-cycle implementation  
   control function for, 269  
   defined, 270  
   nonpipelined execution *versus*  
     pipelined execution, 276  
   non-use of, 271–272  
   penalty, 271–272  
   pipelined performance *versus*, 274  
 Single-instruction multiple-thread  
   (SIMT), C-27–30  
   overhead, C-35  
   multithreaded warp scheduling, C-28  
   processor architecture, C-28  
   warp execution and divergence,  
     C-29–30  
 Single-program multiple data (SPMD),  
   C-22  
 sll (Shift Left Logical), 64  
 slt (Set Less Than), 64  
 slti (Set Less Than Imm.), 64

- sltiu (Set Less Than Imm.Unsigned), 64  
sltu (Set Less Than Unsg.), 64  
Smalltalk-80, OL2.21-8  
Smart phones, 7  
Snooping protocol, 468–470  
Snoopy cache coherence, OL5.12-7  
Software optimization  
  via blocking, 413–418  
Sort algorithms, 141  
Software  
  layers, 13  
  multiprocessor, 500  
  parallel, 501  
  as service, 7, 532, 558  
  systems, 13  
Sort procedure, 135–139. *See also*  
  Procedures  
    code for body, 135–137  
  full procedure, 138–139  
  passing parameters in, 138  
  preserving registers in, 138  
  procedure call, 137  
    register allocation for, 135  
Sorting performance, C-54–55  
Source files, A-4  
Source language, A-6  
Space allocation  
  on heap, 104–106  
  on stack, 103  
SPARC  
  annulling branch, E-23  
  CASA, E-31  
  conditional branches, E-10–12  
  fast traps, E-30  
  floating-point operations, E-31  
  instructions, E-29–32  
  least significant bits, E-31  
  multiple precision floating-point  
    results, E-32  
  nonfaulting loads, E-32  
  overlapping integer operations, E-31  
  quadruple precision floating-point  
    arithmetic, E-32  
  register windows, E-29–30  
  support for LISP and Smalltalk, E-30  
Sparse matrices, C-55–58  
Sparse Matrix-Vector multiply (SpMV),  
  C-55, C-57, C-58  
  CUDA version, C-57  
  serial code, C-57  
  shared memory version, C-59  
Spatial locality, 374  
  large block exploitation of, 391  
  tendency, 378  
SPEC, OL1.12-11–1.12-12  
  CPU benchmark, 46–48  
  power benchmark, 48–49  
SPEC2000, OL1.12-12  
SPEC2006, 233, OL1.12-12  
SPEC89, OL1.12-11  
SPEC92, OL1.12-12  
SPEC95, OL1.12-12  
SPECrate, 538–539  
SPECratio, 47  
Special function units (SFUs), C-35, C-50  
  defined, C-43  
Speculation, 333–334  
  hardware-based, 341  
  implementation, 334  
  performance and, 334  
  problems, 334  
  recovery mechanism, 334  
Speed-up challenge, 503–505  
  balancing load, 505–506  
  bigger problem, 504–505  
Spilling registers, 71, 98  
SPIM, A-40–45  
  byte order, A-43  
  features, A-42–43  
  getting started with, A-42  
  MIPS assembler directives support,  
    A-47–49  
  speed, A-41  
  system calls, A-43–45  
  versions, A-42  
  virtual machine simulation, A-41–42  
Split algorithm, 552  
Split caches, 397  
Square root instructions, A-79  
sra (Shift Right Arith.), A-56  
srl (Shift Right Logical), 64  
Stack architectures, OL2.21-4  
Stack pointers  
  adjustment, 100  
  defined, 98  
  values, 100  
Stack segment, A-22  
Stacks  
  allocating space on, 103  
  for arguments, 140  
  defined, 98  
  pop, 98  
push, 98, 100  
recursive procedures, A-29–30  
Stalls, 280  
  as solution to control hazard, 282  
  avoiding with code reordering, 280  
behavioral Verilog with detection,  
  OL4.13-6–4.13-8  
data hazards and, 313–316  
illustrations, OL4.13-23, OL4.13-30  
insertion into pipeline, 315  
load-use, 318  
memory, 400  
write-back scheme, 399  
  write buffer, 399  
Standby spares, OL5.11-8  
State  
  in 2-bit prediction scheme, 322  
  assignment, B-70, D-27  
  bits, D-8  
  exception, saving/restoring, 450  
  logic components, 249  
  specification of, 432  
State elements  
  clock and, 250  
  combinational logic and, 250  
  defined, 248, B-48  
  inputs, 249  
  in storing/accessing instructions,  
    252  
  register file, B-50  
Static branch prediction, 335  
Static data  
  as dynamic data, A-21  
  defined, A-20  
  segment, 104  
Static multiple-issue processors, 333,  
  334–339. *See also* Multiple issue  
control hazards and, 335–336  
instruction sets, 335  
  with MIPS ISA, 335–338  
Static random access memories (SRAMs),  
  378, 379, B-58–62  
array organization, B-62  
basic structure, B-61  
defined, 21, B-58  
fixed access time, B-58  
large, B-59  
read/write initiation, B-59  
synchronous (SSRAMs), B-60  
three-state buffers, B-59, B-60  
Static variables, 102

Status register  
 fields, A-34, A-35  
 Steady-state prediction, 321  
 Sticky bits, 220  
 Store buffers, 343  
 Store instructions. *See also* Load instructions  
 access, C-41  
 base register, 262  
 block, 149  
 compiling with, 71  
 conditional, 122  
 defined, 71  
 details, A-68–70  
 EX stage, 294  
 floating-point, A-79  
 ID stage, 291  
 IF stage, 291  
 instruction dependency, 312  
 list of, A-68–70  
 MEM stage, 295  
 unit for implementing, 255  
 WB stage, 295  
 Store word, 71  
 Stored program concept, 63  
 as computer principle, 86  
 illustrated, 86  
 principles, 161  
 Strcpy procedure, 108–109. *See also*  
   Procedures  
     as leaf procedure, 109  
     pointers, 109  
 Stream benchmark, 548  
 Streaming multiprocessor (SM), C-48–49  
 Streaming processors, C-34, C-49–50  
   array (SPA), C-41, C-46  
 Streaming SIMD Extension 2 (SSE2)  
   floating-point architecture, 224  
 Streaming SIMD Extensions (SSE) and advanced vector extensions in x86, 224–225  
 Stretch computer, OL4.16-2  
 Strings  
   defined, 107  
   in Java, 109–111  
   representation, 107  
 Strip mining, 510  
 Striping, OL5.11–4  
 Strong scaling, 505, 517  
 Structural hazards, 277, 294  
 sub (Subtract), 64

sub.d (FP Subtract Double), A-79  
 sub.s (FP Subtract Single), A-80  
 Subnormals, 222  
 Subtraction, 178–182. *See also* Arithmetic  
   binary, 178–179  
   floating-point, 211, A-79–80  
   instructions, A-56–57  
   negative number, 179  
   overflow, 179  
 subu (Subtract Unsigned), 119  
 Subword parallelism, 222–223, 352, E-17  
   and matrix multiply, 225–228  
 Sum of products, B-11, B-12  
 Supercomputers, OL4.16-3  
   defined, 5  
 SuperH, E-15, E-39–40  
 Superscalars  
   defined, 339, OL4.16-5  
   dynamic pipeline scheduling, 339  
   multithreading options, 516  
 Surfaces, C-41  
 sw (Store Word), 64  
 Swap procedure, 133. *See also* Procedures  
   body code, 135  
   full, 135, 138–139  
   register allocation, 133  
 Swap space, 434  
 swc1 (Store FP Single), A-73  
 Symbol tables, 125, A-12, A-13  
 Synchronization, 121–123, 552  
   barrier, C-18, C-20, C-34  
   defined, 518  
   lock, 121  
   overhead, reducing, 44–45  
   unlock, 121  
 Synchronizers  
   defined, B-76  
   failure, B-77  
   from D flip-flop, B-76  
 Synchronous DRAM (SRAM), 379–380, B-60, B-65  
 Synchronous SRAM (SSRAM), B-60  
 Synchronous system, B-48  
 Syntax tree, OL2.15-3  
 System calls, A-43–45  
   code, A-43–44  
   defined, 445  
   loading, A-43  
 Systems software, 13  
 SystemVerilog  
   cache controller, OL5.12-2

cache data and tag modules, OL5.12–6  
 FSM, OL5.12–7  
 simple cache block diagram, OL5.12–4  
 type declarations, OL5.12–2

## T

Tablets, 7  
 Tags  
   defined, 384  
   in locating block, 407  
   page tables and, 434  
   size of, 409  
 Tail call, 105–106  
 Task identifiers, 446  
 Task parallelism, C-24  
 Task-level parallelism, 500  
 Tebibyte (TiB), 5  
 Telsa PTX ISA, C-31–34  
   arithmetic instructions, C-33  
   barrier synchronization, C-34  
   GPU thread instructions, C-32  
   memory access instructions, C-33–34  
 Temporal locality, 374  
   tendency, 378  
 Temporary registers, 67, 99  
 Terabyte (TB), 6  
   defined, 5  
 Text segment, A-13  
 Texture memory, C-40  
 Texture/processor cluster (TPC), C-47–48  
 TFLOPS multiprocessor, OL6.15–6  
 Thrashing, 453  
 Thread blocks, 528  
   creation, C-23  
   defined, C-19  
   managing, C-30  
   memory sharing, C-20  
   synchronization, C-20  
 Thread parallelism, C-22  
 Threads  
   creation, C-23  
   CUDA, C-36  
   ISA, C-31–34  
   managing, C-30  
   memory latencies and, C-74–75  
   multiple, per body, C-68–69  
   warps, C-27  
 Three Cs model, 459–461  
 Three-state buffers, B-59, B-60

- Throughput  
defined, 30–31  
multiple issue and, 342  
pipelining and, 286, 342
- Thumb, E-15, E-38
- Timing  
asynchronous inputs, B-76–77  
level-sensitive, B-75–76  
methodologies, B-72–77  
two-phase, B-75
- TLB misses, 439. *See also* Translation-lookaside buffer (TLB)  
entry point, 449  
handler, 449  
handling, 446–453  
occurrence, 446  
problem, 453
- Tomasulo's algorithm, OL4.16–3
- Touchscreen, 19
- Tournament branch predictors, 324
- Tracks, 381–382
- Transfer time, 383
- Transistors, 25
- Translation-lookaside buffer (TLB), 438–439, E-26–27, OL5.17–6. *See also* TLB misses  
associativities, 439  
illustrated, 438  
integration, 440–441  
Intrinsic FastMATH, 440  
typical values, 439
- Transmit driver and NIC hardware time *versus* receive driver and NIC hardware time, OL6.9–8
- Transmitter Control register, A-39–40
- Transmitter Data register, A-40
- Trap instructions, A-64–66
- Tree-based parallel scan, C-62
- Truth tables, B-5  
ALU control lines, D-5  
for control bits, 260–261  
datapath control outputs, D-17  
datapath control signals, D-14  
defined, 260  
example, B-5  
next-state output bits, D-15  
PLA implementation, B-13
- Two's complement representation, 75–76  
advantage, 75–76  
negation shortcut, 76  
rule, 79
- sign extension shortcut, 78
- Two-level logic, B-11–14
- Two-phase clocking, B-75
- TX-2 computer, OL6.15–4
- U**
- Unconditional branches, 91
- Underflow, 198
- Unicode  
alphabets, 109  
defined, 110  
example alphabets, 110
- Unified GPU architecture, C-10–12  
illustrated, C-11  
processor array, C-11–12
- Uniform memory access (UMA), 518, C-9  
multiprocessors, 519
- Units  
commit, 339–340, 343  
control, 247–248, 259–261, D-4–8, D-10, D-12–13  
defined, 219  
floating point, 219  
hazard detection, 313, 314–315  
for load/store implementation, 255  
special function (SFUs), C-35, C-43, C-50
- UNIVAC I, OL1.12–5
- UNIX, OL2.21–8, OL5.17–9–5.17–12  
AT&T, OL5.17–10  
Berkeley version (BSD), OL5.17–10  
genius, OL5.17–12  
history, OL5.17–9–5.17–12
- Unlock synchronization, 121
- Unresolved references  
defined, A-4  
linkers and, A-18
- Unsigned numbers, 73–78
- Use latency  
defined, 336–337  
one-instruction, 336–337
- V**
- Vacuum tubes, 25
- Valid bit, 386
- Variables  
C language, 102  
programming language, 67
- register, 67
- static, 102
- storage class, 102
- type, 102
- VAX architecture, OL2.21–4, OL5.17–7
- Vector lanes, 512
- Vector processors, 508–510. *See also* Processors  
conventional code comparison, 509–510  
instructions, 510  
multimedia extensions and, 511–512  
scalar *versus*, 510–511
- Vectored interrupts, 327
- Verilog  
behavioral definition of MIPS ALU, B-25  
behavioral definition with bypassing, OL4.13–4–4.13–6  
behavioral definition with stalls for loads, OL4.13–6–4.13–8  
behavioral specification, B-21, OL4.13–2–4.13–4  
behavioral specification of multicycle MIPS design, OL4.13–12–4.13–13  
behavioral specification with simulation, OL4.13–2  
behavioral specification with stall detection, OL4.13–6–4.13–8  
behavioral specification with synthesis, OL4.13–11–4.13–16  
blocking assignment, B-24  
branch hazard logic implementation, OL4.13–8–4.13–10  
combinational logic, B-23–26  
datatypes, B-21–22  
defined, B-20  
forwarding implementation, OL4.13–4  
MIPS ALU definition in, B-35–38  
modules, B-23  
multicycle MIPS datapath, OL4.13–14  
nonblocking assignment, B-24  
operators, B-22  
program structure, B-23  
reg, B-21–22  
sensitivity list, B-24  
sequential logic specification, B-56–58  
structural specification, B-21  
wire, B-21–22
- Vertical microcode, D-32

Very large-scale integrated (VLSI)  
circuits, 25

Very Long Instruction Word (VLIW)  
defined, 334–335  
first generation computers, OL4.16–5  
processors, 335

VHDL, B-20–21

Video graphics array (VGA) controllers,  
C-3–4

Virtual addresses

causing page faults, 449  
defined, 428  
mapping from, 428–429  
size, 430

Virtual machine monitors (VMMs)  
defined, 424

implementing, 481, 481–482

*laissez-faire* attitude, 481

page tables, 452

in performance improvement, 427  
requirements, 426

Virtual machines (VMs), 424–427

benefits, 424

defined, A-41

illusion, 452

instruction set architecture support,  
426–427

performance improvement, 427

for protection improvement, 424

simulation of, A-41–42

Virtual memory, 427–454. *See also* Pages

address translation, 429, 438–439

integration, 440–441

mechanism, 452–453

motivations, 427–428

page faults, 428, 434

protection implementation,

444–446

segmentation, 431

summary, 452–453

virtualization of, 452

writes, 437

Virtualizable hardware, 426

Virtually addressed caches, 443

Visual computing, C-3

Volatile memory, 22

## W

Wafers, 26  
defects, 26  
dies, 26–27  
yield, 27

Warehouse Scale Computers (WSCs), 7,  
531–533, 558

Warps, 528, C-27

Weak scaling, 505

Wear levelling, 381

While loops, 92–93

Whirlwind, OL5.17–2

Wide area networks (WANs), 24. *See also*

Networks

Words

accessing, 68  
defined, 66  
double, 152  
load, 68, 71  
quad, 154  
store, 71

Working set, 453

World Wide Web, 4

Worst-case delay, 272

Write buffers

defined, 394  
stalls, 399  
write-back cache, 395

Write invalidate protocols, 468, 469

Write serialization, 467

Write-back caches. *See also* Caches

advantages, 458  
cache coherency protocol, OL5.12–5  
complexity, 395  
defined, 394, 458  
stalls, 399  
write buffers, 395

Write-back stage

control line, 302  
load instruction, 292  
store instruction, 294

Writes

complications, 394  
expense, 453  
handling, 393–395

memory hierarchy handling of,  
457–458

schemes, 394

virtual memory, 437

write-back cache, 394, 395

write-through cache, 394, 395

Write-stall cycles, 400

Write-through caches. *See also* Caches

advantages, 458

defined, 393, 457

tag mismatch, 394

## X

x86, 149–158

Advanced Vector Extensions in, 225

brief history, OL2.21–6

conclusion, 156–158

data addressing modes, 152, 153–154

evolution, 149–152

first address specifier encoding, 158

historical timeline, 149–152

instruction encoding, 155–156

instruction formats, 157

instruction set growth, 161

instruction types, 153

integer operations, 152–155

registers, 152, 153–154

SIMD in, 507–508, 508

Streaming SIMD Extensions in,

224–225

typical instructions/functions, 155

typical operations, 157

Xerox Alto computer, OL1.12–8

XMM, 224

## Y

Yahoo! Cloud Serving Benchmark  
(YCSB), 540

Yield, 27

YMM, 225

## Z

Zettabyte, 6

# MIPS Reference Data

①



## CORE INSTRUCTION SET

NAME, MNEMONIC	MAT	FOR- / FUNCT	OPERATION (in Verilog)	OPCODE
Add	add	R	R[rd] = R[rs] + R[rt]	(1) 0 / 20 <sub>hex</sub>
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2) 8 <sub>hex</sub>
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2) 9 <sub>hex</sub>
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]	0 / 21 <sub>hex</sub>
And	and	R	R[rd] = R[rs] & R[rt]	0 / 24 <sub>hex</sub>
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3) c <sub>hex</sub>
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4) 4 <sub>hex</sub>
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4) 5 <sub>hex</sub>
Jump	j	J	PC=JumpAddr	(5) 2 <sub>hex</sub>
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5) 3 <sub>hex</sub>
Jump Register	jr	R	PC=R[rs]	0 / 08 <sub>hex</sub>
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]]+SignExtImm}(7:0)}	(2) 24 <sub>hex</sub>
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]]+SignExtImm}(15:0)}	(2) 25 <sub>hex</sub>
Load Linked	ll	I	R[rt]=M[R[rs]+SignExtImm]	(2,7) 30 <sub>hex</sub>
Load Upper Imm.	lui	I	R[rt]={imm, 16'b0}	f <sub>hex</sub>
Load Word	lw	I	R[rt]=M[R[rs]+SignExtImm]	(2) 23 <sub>hex</sub>
Nor	nor	R	R[rd] = ~ (R[rs]   R[rt])	0 / 27 <sub>hex</sub>
Or	or	R	R[rd] = R[rs]   R[rt]	0 / 25 <sub>hex</sub>
Or Immediate	ori	I	R[rt] = R[rs]   ZeroExtImm	(3) d <sub>hex</sub>
Set Less Than	slt	R	R[rd]=(R[rs] < R[rt]) ? 1 : 0	0 / 2a <sub>hex</sub>
Set Less Than Imm.	slti	I	R[rt]=(R[rs] < SignExtImm) ? 1 : 0 (2)	a <sub>hex</sub>
Set Less Than Imm. Unsigned	sltiu	I	R[rt]=(R[rs] < SignExtImm) ? 1 : 0 (2,6)	b <sub>hex</sub>
Set Less Than Unsigned	sltu	R	R[rd]=(R[rs] < R[rt]) ? 1 : 0 (6)	0 / 2b <sub>hex</sub>
Shift Left Logical	sll	R	R[rd]=R[rt] << shampt	0 / 00 <sub>hex</sub>
Shift Right Logical	srl	R	R[rd]=R[rt] >> shampt	0 / 02 <sub>hex</sub>
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2) 28 <sub>hex</sub>
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (atomic) ? 1 : 0	(2,7) 38 <sub>hex</sub>
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2) 29 <sub>hex</sub>
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2) 2b <sub>hex</sub>
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1) 0 / 22 <sub>hex</sub>
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]	0 / 23 <sub>hex</sub>

(1) May cause overflow exception

(2) SignExtImm = { 16 {immediate[15]}, immediate }

(3) ZeroExtImm = { 16 {1b'0}, immediate }

(4) BranchAddr = { 14 {immediate[15]}, immediate, 2'b0 }

(5) JumpAddr = { PC+4[31:28], address, 2'b0 }

(6) Operands considered unsigned numbers (vs. 2's comp.)

(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

## BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26 25	21 20	16 15	11 10	6 5	0
I	opcode	rs	rt			immediate
	31 26 25	21 20	16 15			0
J	opcode			address		
	31 26 25					0

© 2014 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, Computer Organization and Design, 5th ed.

## ARITHMETIC CORE INSTRUCTION SET

② OPCODE / FMT / FT / FUNCT (Hex)

NAME, MNEMONIC	MAT	FOR-	OPERATION
Branch On FP True	bcjt	FI	if(FPcond)PC=PC+4+BranchAddr (4)
Branch On FP False	bcjf	FI	if(!FPcond)PC=PC+4+BranchAddr(4)
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs] % R[rt]
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs] % R[rt] (6)
FP Add Single	add.s	FR	F[fd] = F[fs] + F[ft]
FP Add		FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}
Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}
FP Compare Single	c.x.s*	FR	FPcond = {F[fs] op F[ft]} ? 1 : 0
FP Compare	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} op {F[ft],F[ft+1]}) ? 1 : 0
Double			(* (x is eq, lt, or le) (op is ==, <, or <=) (y is 32, 3c, or 3e))
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]
FP Divide	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]
FP Multiply	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]
FP Subtract	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}
Double			
Load FP Single	lwcl	I	F[rt]=M[R[rs]+SignExtImm] (2)
Load FP	ldcl	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4] (2)
Double			35/-/-/-
Move From Hi	mfhi	R	R[rd] = Hi
Move From Lo	mflo	R	R[rd] = Lo
Move From Control	mfc0	R	R[rd] = CR[rs]
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt] (6)
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shampt
Store FP Single	swcl	I	M[R[rs]+SignExtImm] = F[rt] (2)
Store FP	sdcl	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1] (2)
Double			3d/-/-/-

## FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
31 26 25	21 20	16 15	11 10	6 5	0	
FI	opcode	fmt	ft		immediate	
31 26 25	21 20	16 15				0

## PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

## REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

### OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS	(1) MIPS opcode	(2) MIPS funct	Binary	Deci- mal	Hexa- decimal	ASCII Char- acter	Deci- mal	Hexa- decimal	ASCII Char- acter
(31:26)	(5:0)	(5:0)							
(1)	sll	addf	00 0000	0	0	NUL	64	40	@
		subf	00 0001	1	1	SOH	65	41	A
j	srl	multf	00 0010	2	2	STX	66	42	B
jal	sra	divf	00 0011	3	3	ETX	67	43	C
beq	slrv	sqrtf	00 0100	4	4	EOT	68	44	D
bne		absf	00 0101	5	5	ENQ	69	45	E
blez	srsv	movf	00 0110	6	6	ACK	70	46	F
bgtz	srav	negf	00 0111	7	7	BEL	71	47	G
addi	jr	00 1000	8	8	BS	72	48	H	
addiu	jalr		00 1001	9	9	HT	73	49	I
slt	movz		00 1010	10	a	LF	74	4a	J
slti	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.f	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.f	00 1101	13	d	CR	77	4d	M
xori		ceil.w.f	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.f	00 1111	15	f	SI	79	4f	O
(2)	mfhi		01 0000	16	10	DLE	80	50	P
mthi			01 0001	17	11	DC1	81	51	Q
mflo	movzf		01 0010	18	12	DC2	82	52	R
mtlo	movnf		01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
mult			01 1000	24	18	CAN	88	58	X
multu			01 1001	25	19	EM	89	59	Y
div			01 1010	26	1a	SUB	90	5a	Z
divu			01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.f	10 0000	32	20	Space	96	60	-
lh	addu	cvt.d.f	10 0001	33	21	!	97	61	a
lw	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.f	10 0100	36	24	\$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	-	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
			10 1110	46	2e	.	110	6e	n
			10 1111	47	2f	/	111	6f	o
swr			11 0000	48	30		112	70	p
cache			11 0001	49	31	1	113	71	q
11	tge	c.f.f	11 0010	50	32	2	114	72	r
lwcl	tgeu	c.unf	11 0011	51	33	3	115	73	s
lwcl	tlc	c.eqf	11 0100	52	34	4	116	74	t
pref	tltu	c.ueqf	11 0101	53	35	5	117	75	u
ldc1	teq	c.oltf	11 0100	54	36	6	118	76	v
ldc2	tne	c.ultf	11 0101	55	37	7	119	77	w
sc		c.csf	11 1000	56	38	8	120	78	x
swc1		c.nglef	11 1001	57	39	9	121	79	y
swc2		c.seqf	11 1010	58	3a	:	122	7a	z
		c.nglf	11 1011	59	3b	:	123	7b	{
		c.ltf	11 1100	60	3c	<	124	7c	
sdc1		c.ngef	11 1101	61	3d	=	125	7d	}
sdc2		c.lef	11 1110	62	3e	>	126	7e	~
		c.ngtf	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0

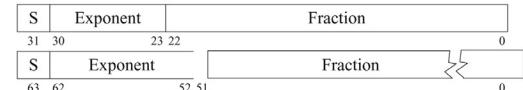
(2) opcode(31:26) == 17<sub>10</sub> (11<sub>hex</sub>); if fmt(25:21)==16<sub>10</sub> (10<sub>hex</sub>) f = s (single); if fmt(25:21)==17<sub>10</sub> (11<sub>hex</sub>) f = d (double)

### IEEE 754 FLOATING-POINT STANDARD

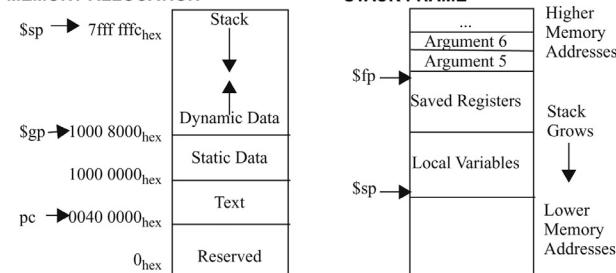
$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where Single Precision Bias = 127,  
Double Precision Bias = 1023.

### IEEE Single Precision and Double Precision Formats:



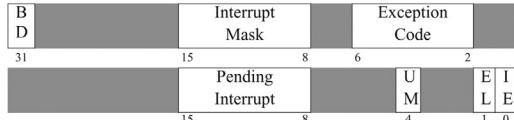
### MEMORY ALLOCATION



### DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7
Value of three least significant bits of byte address (Big Endian)							

### EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable

### EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

### SIZE PREFIXES

PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10 <sup>3</sup>	Kilo-	K	2 <sup>10</sup>	Kibi-	Ki	10 <sup>15</sup>	Peta-
10 <sup>6</sup>	Mega-	M	2 <sup>20</sup>	Mebi-	Mi	10 <sup>18</sup>	Exa-
10 <sup>9</sup>	Giga-	G	2 <sup>30</sup>	Gibi-	Gi	10 <sup>21</sup>	Zetta-
10 <sup>12</sup>	Tera-	T	2 <sup>40</sup>	Tebi-	Ti	10 <sup>24</sup>	Yotta-
							Yi